Automatic Program Verification

PhD Dissertation

Kristoffer Just Arndal Andersen

Department of Computer Science
Aarhus University
Denmark

Colophon

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić.

Automatic Program Verification

A Dissertaion
Presented to the Faculty of Science and Technology
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Kristoffer Just Arndal Andersen
September 2019

Abstract

The expressive power of program logics for formal verification has risen explosively over the last three decades, as witnessed by the proliferation of separation logic variants for proving properties of non-trivial concurrent and distributed code. The complexity of abstractions and formal tools grows to match, and the return of investment in automated tool assistance becomes ever more evident.

Modern program logics like CAP and its descendants hone the insights of *resource logics*: that *ownership* of resources is key to building modular programs and proofs; and, that *protocols* describing how ownership is transferred between processes is a meaningful abstraction for describing systems with multiple concurrent actors, like shared-memory concurrency and distributed systems.

This thesis presents two approaches to adapting ideas from CAP-like program logics – frameworks for manual proofs of correctness – to *tools for automatic verification*, programs that assist the programmer in proving properties of their program.

We present Caper, the first tool of its kind to provide fully automatic proof-search for a CAP based program logic, providing *static* guarantees that a program conforms to its specification. A novel approach to interference reasoning based on *guard algebras* is key to enabling this tool, and we demonstrate its viability on a number of concurrent data structures and synchronization mechanisms.

The second half of the this thesis presents Distributed Protocol Combinators (DPC), a framework for disciplined *dynamic* testing, informed in design by techniques from program logics for distributed systems, specifically Disel. It represents a novel approach to describing protocols for distributed systems in a composable manner. The modular assembly of protocols inspire a methodology of specification development, analogous to programming, exploiting that specifications can be given an executable semantics and run like any other program. We demonstrate the approach on a number of case studies adapted from the literature on verification of distributed programming.

Resumé

Matematiske beskrivelser af programmers opførsel er grundstenen i formel verifikation – det at *bevise* at et program opføre sig som forventet under *alle* tænkelige forhold. Dette er idealet hvorefter al software udvikles, men oftest tyes der til uformel *testing*, der blot kan påvise fejl under *enkelte* omstændigheder. Grundig verifikation er arbejdsintensivt og fejlbehæftet, og investeringen af mandetimer ses oftest kun som investeringen værd i højrisiko foretagender hvor menneskeliv og mange års arbejde er på spil, som i rum- og luftfartsindustrien.

Programlogikker er regelsystemer der indfanger essencen af det matematiske argument der skal til for at vise korrektheden af et program, og i trit med at programmeringssprog er blevet mere avanceret har udviklerne af programlogikker mødt udfordringen med til stadighed mere avancerede teknikker.

Særligt flertrådet og distribuerede systemer har set store fremskridt i blot de seneste 5 år, og der er set større gennembrud i omfanget af programmer det nu er muligt at verificere. Der er dog tydeligt et behov for computerassistance i verifikationsopgaven. Det kan sænke adgangsbarrieren for hvornår det kan betale sig at benytte verifikation, og løfte niveauet for hvad der overhovedet er muligt at verificere.

I denne afhandling introducerer vi to værktøjer til *automatisk verifikation* af programmer i moderne programlogikker.

Caper er et værktøj der automatisk beviser korrektheden af flertrådet programmer der gør brug af såkaldte "fine-grained" operationer, frem for klassiske synkroniseringsmekanismer som låse. Det lader sig gøre via en ny teknik til at argumentere, at interaktionen mellem flere tråde ikke er ondartet. Vi demonstrerer effektiviteten af værktøjet på realistiske datastrukturere og synkroniseringsmekanismer.

I den anden halvdel af denne afhandling introduceres Distributed Protocol Combinators, et software bibliotek til programmeringssproget Haskell der gør det muligt at eksperimentere med specifikation af distribueret software på en let og billig, men formelt talt grundig facon. Ved at udnytte designprincipper fra programlogikker introducerer vi en ny, modulær måde at udtrykke *protokoller*: mønstre for kommunikation og samarbejde i systemer med flere aktøre. Modulariteten af disser inviterer til en oplagt måde at bygge testsscenarier, og vi kan udnytte den øvrige tradition for testing af Haskell programmer til at *generere* tests af distribuerede systemer. I visse tilfælde kan vi løfte eksisterende testingteknikker til decideret verifikation. Vi demonstrerer perspektiverne i metodikken på udvalgte eksempler på distribuerede systemer fra litteraturen.

Acknowledgments

First and foremost, my deepest gratitude is due Lars Birkedal, my advisor, for giving me this opportunity, and for introducing me to Thomas Dinsdale-Young, my co-advisor – if not by the letter, then in spirit. Without either this project would not have been.

I would like to thank my colleagues at The Department of Computer Science for such a wonderful environment through all these years, almost exactly 9 at the time of writing. Among them goes my deepest gratitude to Sofia Rasmussen and Anders Møller for such a no-nonsense approach to administrating the ph.d. programme. I would also like to thank Jan Midtgaard, Olivier Danvy, Erik Ernst, Jakob Thomsen and all the other staff and students of the programming language research group at Aarhus University for encouraging an early interest in PL.

Most enduring, is said, are friendships forged in fire. I have no doubt that my fellow ph.d. students Morten, Lau and Aleš will remember my time at Aarhus University for all the discussions, rants, right out arguments, stories, laughs, travels and–last, but certainly not least–the cups of coffee we shared over the years. They will all remain my among my fondest memories.

A heartfelt thank you to Ilya Sergey, Thomas Sibut-Pinot and Maria Schett at University College London, and the entire team of fantastically talented people at Wooga for their hospitality during my year of adventure.

My dear parents, brother Jens, and extended family of Rotvels and Nielsens have cheered for me all throughout, and I particularly apologize to Uncle Christian for all the parties I missed due to summer schools and other travels!

And to Camilla, the meaning of it all: what a journey it has been so far. I can't wait to see where it will take us next.

Aarhus
September 2019

Contents

1 Introduction

This section is adapted from the unpublished report "Automatic Verification of Fine-Grained Concurrency" by the author [7].

The work in this thesis hangs from a very sturdy family tree. It stems from research in program logics for verification, in particular the tradition of separation logic.

Program verification is the problem of *proving properties* about computer program behaviors, expressed in terms of mathematical descriptions of program behavior, their *semantics*. Early efforts in assigning mathematical meaning to programs were driven precisely by the possibility of reasoning formally about programs, as envisioned by e.g. Floyd, McCarthy and Hoare[33, 38, 55].

Program logics are mathematical frameworks founded on these precise formulations of programming language semantics. They codify semantic arguments about program behavior as inference rules, a format familiar to any computer scientists. A rule describing the behavior of a conditional statement of a simple imperative language might be presented as follows:

$$\frac{\left\{P \wedge \mathtt{e}\right\}\ \mathtt{s}_1\ \left\{Q\right\} \qquad \left\{P \wedge \neg\mathtt{e}\right\}\ \mathtt{s}_2\ \left\{Q\right\}}{\left\{P\right\}\ \mathtt{if(e)\ then}\ \{\mathtt{s}_1\}\ \mathtt{else}\ \{\mathtt{s}_2\}\ \left\{Q\right\}}$$

This rule states the following:

A computer about to execute the program

$$\mathtt{if(e)\ then}\ \{\mathtt{s}_1\}\ \mathtt{else}\ \{\mathtt{s}_2\}$$

enters a state described by the assertion Q if it starts in a state described by P provided that $\mathtt{s}_1$ could take the machine from states $P \wedge \mathtt{e}$ to Q and $\mathtt{s}_2$ could take the machine from states $P \wedge \neg\mathtt{e}$ to Q.

Throughout this thesis, *program execution* is described by small-step structural operational semantics, and program states will be, depending on the exact setting, an abstraction of shared data.

When "syntax directed" like the conditional rule (provided it is the only such rule describing the behavior of conditionals) they leave only decision of logical entailments to the verifier of the program - that is, the structure of the proof is unambiguously dictated by the syntactic structure of the program. Program logics for even the most basic of interesting programs, however, will involve human intuition

e. g.loop invariants must still be divined, as exemplified by a classic presentation of the While-rule for a simple imperative language:

$$\frac{P \Rightarrow I \qquad \left\{I \land \texttt{e}\right\} \texttt{s} \left\{I\right\} \qquad I \land \neg\texttt{e} \Rightarrow Q}{\left\{P\right\} \texttt{while(e)\{s\}} \left\{Q\right\}}$$

The choice of I is not inherent in the conclusion of the rule, and must be *chosen* by some method of deduction. This exact problem has spawned entire research areas of its own, which we will not dwell on in this thesis.

However, modern program logics are yet more sophisticated.

As any student tasked with showing an in-place linked list reversal algorithm correct by means of Hoare-style logic for an imperative programming language with pointers will attest to, the amount of bookkeeping required is prohibitive. In fact, the threshold for tool assistance to make an impact is very low, and recent advances in mechanization of such logics[16, 41] have proven incredibly fruitful in scaling more or less manual proofs to very large developments.

Separation logic[68] broke through around the turn of the millennium as a solution to the anguish faced by provers of imperative programs with mutable state: the bookkeeping was required in order to maintain that a local update by a program did not invalidate the global state of the program. This had previously been vaguely conventionalized by a *framing principle*: a program alters only the state its contract mentions, and everything else remains the same. By introducing a new connective, the *separating conjunction*, which formalized the otherwise by-convention maintained *frame rule*, separation logic allowed proofs of *local* program correctness to be composed in a modular way.

The introduction of concurrent execution followed a similar development in terms of program logics, but saw great leaps from the insights gained by the advances in separation logic. One of the earliest logics for concurrency is the Owicki-Gries logic, which formulated a correctness condition for concurrent programs as a *global* property, that every execution step had to maintain. As with imperative programs, global properties interfere with the ability to build proofs at scale as they do not allow for modular proofs of *local* correctness that can then be composed, and even reused, to argue global correctness.

The ideas from separation logic have evolved in the context of concurrent programming, where separating conjunction describes *disjointness* and *ownership* of program resources, and these, in turn, provide the right abstractions for expressing composable proofs of correctness for concurrent programs. The recent proliferation in concurrent separation logics (see Figure 1.1) all make use of the key insight that disjointness of resources enable safe parallel execution of programs, and exchange of resources is mediated through protocols that describe how ownership of resources changes throughout the execution.

In recent years, the insights gained in tackling the problem of concurrent execution has been lifted to the domain of distributed computation. In many ways, the settings are similar: both involve non-deterministic interleaving of computation, one as threads on a single machine, another as processes running on separate machines. Both involve the interaction of processes, one via shared memory, the other via message passing modeling the interaction of machines across a network. They can in fact model each other: the heap can be regarded as an actor on a network, and the network can be modeled as a collection of messages stored in a suitable data structure on the heap. The closeness of the domains helped the transfer of tools and techniques, resulting in e. g.modular verification of consensus algorithms using essences of frameworks developed for concurrent programs[34].

As pointed out, verifying that a program meets its specification can be a laborious process, and often involves a costly, human effort. Developing tools and techniques for alleviating this effort is therefore a project with a clear return of investment.

The earliest approaches to computer assisted verification was based on the "weakest precondition calculus" arising from the predicate transformer semantics of Dijkstra[22]. For example, the *verification condition generation* approach translates a program into a single logical formula, the validity of which implies the correctness of the source program. This is still the logical foundation for formal developments of program logics like Iris[41].

Deciding whether a program meets its specification very quickly becomes an undecidable problem. Modern programming languages includes facilities for unbounded loops and recursion, higher-order functions and other abstraction mechanisms that lead to undecidability in computationally viable fragments of logic. Here, tools assisting human provers operate in a very interesting space, as it can draw on the user as an *oracle*, to divine answers to undecideable problem, precisely like the inference of loop invariants as suggested earlier.

The key contribution of separation logic to the field of automated tools for program verification is the introduction of separation logic assertions as a symbolic representation of program state, enabling symbolic execution of resource manipulating programs, a technique solidified by the SmallFoot project[8]. This work spawned a whole family of tools, extending and varying the approach, and it is also in this branch of the program logic tree that we find this thesis rooted.

As we look ahead, I personally perceive the field of computer-assisted verification in general, and area of automation specifically, as the toolmakers of Theoretical Computer Scientist across disciplines. With evergrowing complexity of computer architectures and computing systems, the formalization efforts of algorithmics, cryptography,

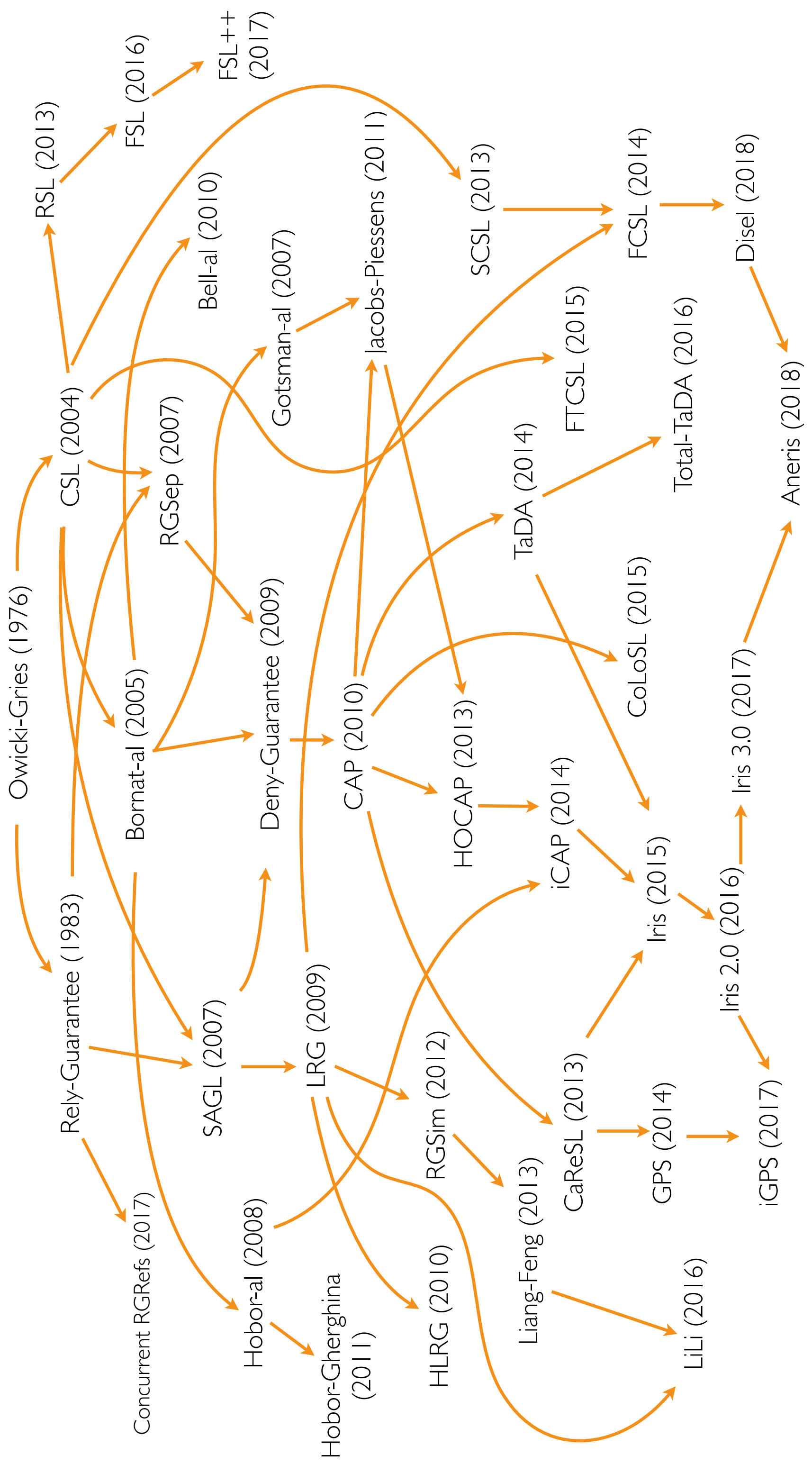


Figure 1.1: Ilya Sergey's infamous Family Tree of Concurrent Separation Logics

distributed systems design grow from valiant through heroic to infeasible. As recent as the 15th of August, 2019, Jens Palsberg remarked in a lecture on Quantum Computing that the current state of the art in quantum computers ($\sim$ 72 qubits) is at the very limit of what we can simulate with classical computers. Beyond that, no meaningful amount of testing, simulation or debugging is *possible*, by the very nature of quantum of computation. We absolutely need mature verification tools to handle programming for quantum computers.

With computer systems growing ever larger, the prospects of making an impact with advances in automatic verification beyond the research field itself are looking ever more promising.

1.1 Research Hypothesis & Objectives

This project begins as the field of program logics for concurrency is rapidly gaining in activity. The time was ripe to experiment with the transfer of insights from formal frameworks into tools. CAP-like logics introduced very abstract protocols into the mix of program logics, and this project started with the aim of taking this idea to completion and produce a prototype implementation of tool-supported verification in a CAP-like logic.

The central idea of this work is that, however much human ingenuity is needed in formal verification with program logics, there is a structure to the proofs derived from the program and its specification, and this begs the research question:

> To what extent can we automate the work involved in using program logics for verification of modern programs using fine-grained concurrency and distributed computing?

1.2 Scientific Contributions & Artifacts

The scientific body of work and their key contributions to this ph.d. project is outlined below. The final versions of manuscripts make up the technical content of this thesis, and forward references are provided. Their inclusion in the thesis is prefaced by a description of their differences from the published versions.

Software – Caper Tool

> Thomas Dinsdale-Young, Pedro da Rocha Pinto, and Kristoffer Just Andersen. *Caper (Source Code)*. URL: https://github.com/caper-tool/caper

As suggested in a CACM survey and position paper from 2013 [12], symbolic execution in the context of verifying concurrent code is an important open challenge for the state of the art, which Caper tackled head-on with beginnings around the same time. Caper is the first *fully automatic prover* for a modern CAP-like program logic. The key novelty of the tool setting it apart from the state-of-the-art at the time was the treatment of high-level protocols for shared memory, in Caper called *regions*. As highlighted in the accompanying paper, the key insights that facilitate this reasoning is:

- *guard algebras* for describing abstract resources used to facilitate interaction between threads according to protocols. Guard algebras land in a sweet-spot in terms of design for expressiveness and usability, being quite intuitive while still facilitating interesting concurrency patterns.

- techniques for reasoning about interference of threads, specifically facilitated by guards. In particular, Caper computes a sound interference relation based on a collection of simple transitions enabled by resources of the guard algebra.

- Backtracking proof search. Caper eschews computing abstractions at join-points or other state-space reducing techniques in favor of guided search, employing heuristics, including a novel use of abduction, to infer where to perform actions in protocols, create new instances of protocols etc.

The Caper project demonstrates the success of the approach, and the repository includes many interesting case studies.

I joined the project in early 2015, and helped bring the tool from a sketch to a full working prototype. The workings of Caper is described in the accompanying publication, included in Chapter 2.

Conference Publication – Caper: Fine-Grained Verification of Concurrent Programs

> Thomas Dinsdale-Young et al. "Caper." In: *ESOP: Proceedings of the 26th European Symposium on Programming*. 2017

The primary publication describing the motivation, design and utility of the Caper tool for verification of fine-grained concurrent programs. The paper illustrates the verification approach with a sequence of increasingly sophisticated examples of fine-grained data structures and concurrency primitives, and illustrates the modularity of the approach by showing that client programs of such library code can be verified, as well.

The paper is included in its entirety in Chapter 2, with expository sentences cut from the original to space limitations reintroduced in the presentation here. Additionally, a *postscript* has been added in Section 2.9, detailing the efforts on improving the tool made since the time of publication.

Technical Report – Caper Program Logic

> Thomas Dinsdale-Young et al. *Caper: Automatic Verification with Concurrent Abstract Predicates. Technical Appendix: Program Logic*. 2016. URL: `http://cs.au.dk/~kja/papers/caper-esop17/techreport.pdf`

To make precise the guarantees provided by the Caper tool we needed to develop a formal model of the object programming language and relate it to the specifications proven by Caper. As an investment in the foundations of this effort and my personal development as a student of logic and programming language semantics, the first substantial artifact produced by this project is this formalization of the CAP-like program logic in which Caper is conceptually seen to search for a proof. In some sense, this can be seen as the "specification" for the symbolic execution based technique employed by Caper: given a program that Caper deems valid, the evidence produced by the tool should be relatable to this program logic. This final link between tool and formal framework is left as future work, started by Thomas Dinsdale-Young as detailed in Section 2.9.

The object language of Caper is a procedural imperative programming language, with a standard structured call/return discipline, with local variables operating on a first-order store of single and contiguous memory cells.

Concurrency is thread-based with fine-grained operations performing atomic updated to the shared store, and the operational semantics are given as a standard thread-local small-step semantics that are then lifted to a thread-pool.

The logic is in many ways standard, but the development here stands apart from other efforts in that care has been taken to mimic the object language of Caper. "Cumbersome" features like non-local returns and local variables have been included, and a distinction of specification language syntax and semantics has been included, in that Caper programs include a syntax for specifications.

The report develops the precise semantics of shared regions, and includes a substantial soundness proof of the logic showing that a valid proof implies *safety*: all terminating executions run without faults to the end of a procedure or return a value according to the post-condition.

The soundness result is established by constructing a semantic model of the program logic. Assertions are given the interpretation of upwards-closed subsets of heaps paired with a region assignment, an "instrumentation" of the concrete state with a "ghost" state, describing the currently allocated regions, analogous to the role of a memory for currently allocated addresses. Heaps and region assignments form partial commutative monoids, and thus their up-closed subsets are a model of intuitionistic separation logic, in addition to assertions unique to Caper.

Specifications are then interpreted as down-closed sets of natural numbers, corresponding to the number of steps a statement is guaranteed safe.

The report is included in its entirety in Chapter 3.

Software – DPC

Kristoffer Just Arndal Andersen and Ilya Sergey. *DPC Haskell Package*. URL: `https://github.com/kandersen/dpc`

Distributed Protocol Combinators (DPC) is an approach to disciplined testing founded on insights from program logic.

The problem inspiring the work grew as I tried to formalize a greater case study for the DISEL framework in the proof assistant Coq. DISEL is a fully mechanized program logic for reasoning about distributed systems. It is a large, complicated system, with the additional barrier to entry of being encoded in Coq. DISEL uses a notion of network transitions to describe global protocols for communication, and the examples worked so far in the project were small, although sophisticated, protocols not involving more than 2 actors in a fairly shallow communication pattern.

I set out to formalize the pattern of distributed locking[43], and found the boilerplate definitions required for interacting with the example in the DISEL framework prohibitively large. Many thousands of lines of well-formedness definitions, lemmas stating preservation of properties... all proof (code) that followed straight from the definitions of the fairly regular protocols.

This led to the formulation of protocols in terms of *protlets*, a novel formulation of primitive interaction patterns that can be composed freely to express sophisticated communication patterns.

Their definition lends itself naturally to implementation, and I did so in the programming language Haskell. What we obtain is a disciplined testing framework for imperative programming on distributed systems. By exploiting the abstraction mechanisms of Haskell we can instantiate the framework with various notions of semantics for protlets, essentially plugging in different language run-times. A particularly interesting construction is when we insert a run-time that treats that performs a kind of symbolic execution, letting us express traces of distributed system executions as computational objects that can be queried and tested.

The essential design principle is the underlying theoretical framework of program logics, namely that of *program refinement*. The technique dates back to Abadi and Lamports work on refinement mappings [1], and expresses a program behavior as relative to the behavior of some other program, possibly in some other notion of computation. This approach is then used to relate the behavior of complicated programs, regarded as implementations, to other, simpler programs taken as specifications – perhaps programs for which static properties are known, or perhaps trivially conform to "business logic" requirements. Relational logics, for example Iris, are then formal frameworks for *proving* this relationship.

Fundamentally, every possible execution of the implementation must be "expressible" by some execution of the specification. In this sense, the implementation is richer, allowing for more executions than the specification (e.g. by allowing concurrent interleaving of program actions).

As with every verification effort, this can be difficult or labor intensive. But the same set-up can be used to express the presence of a bug: if we can identify an execution of the implementation that is *not* in the set of executions expressible in the specification, we have found a bug!

This is the basic methodology of testing, and DPC provides the toolkit for doing this in a disciplined fashion, where the implementation is the monadic language of Haskell and the specification is provided by the protlet language.

The implementation is freely available for experimentation.

I was the primary author of this work.

Conference Publication – Distributed Protocol Combinators

> Kristoffer Just Arndal Andersen and Ilya Sergey. "Distributed Protocol Combinators." In: *Practical Aspects of Declarative Languages - 21th International Symposium, PADL.* 2019

The primary publication on DPC, detailing the motivation, design and use of DPC as a testing and, in the right circumstances, a verification tool. A handful of case studies are described, and empirical comparisons are made.

I was the primary author of this publication.

Journal Publication – Protocol Combinators for Modeling, Testing, and Execution of Distributed Systems

> Kristoffer Just Arndal Andersen and Ilya Sergey. "Protocol Combinators for Modeling, Testing, and Execution of Distributed Systems." In: *Journal of Functional Programming* (2019). In Submission.

Extended version of [5], included in its entirety in Chapter 4.

I was the primary author of this publication.

1.3 Change of Research Environment

I visited Dr Ilya Sergey at University College London's Department of Computer Science during the spring of 2018, more specifically the Principles of Programming, Logic & Verification research group.

Dr Sergey and his team was investigating blockchain and cyptocurrency technologies, founded on prior work on formal verification of programs in general.

We identified excellent synergy in the origins of our work, both having grown from the study of separation logic and its applications in machine-aided verification.

The relationship was very fruitful, with the DPC framework being born from our discussions during my time physically at the department, and the two publications concerning it subsequently completed upon my return to the Department at Aarhus University.

1.3.1 *Leave of Absence*

I spent 6 months in the fall of 2017 on leave in Berlin, Germany, to accompany my wife during her own change of research environment, see [3] for more details. During my time there I worked as a Game Engineer at Wooga GmbH.

1.4 Co-Supervisory Roles

Anders Lindkvist & Troels Jensen Master's Students at the Department of Computer Science, Aarhus University, investigating denotational semantics of reactive programming with the aim of establishing contextual equivalence results for (the reactive version of) the Elm language. See [53].

Jonathan Sutton Master of Engineering Student at Imperial College London's Department of Computer Science. Extended the Caper tool for verification with mechanisms for verifying fault-tolerant code. The project is described in more detail in the Postscript added to the publication on Caper, Section 2.9.

1.5 Thesis Outline

The rest of the thesis consists of the manuscripts of the papers cited above.

Part i details the design and workings of the Caper tool for verification in Chapter 2 and the design of the underlying program logic in Chapter 3. They are reproduced with minor alterations from their state at publication, with the notable inclusion of a postscript detailing work performed on the project since the publication in Section 2.9.

Part ii consists wholly of Chapter 4 and describes the design and use of the DPC framework for modeling distributed system. The chapter consists of the manuscript of the invited extended journal version of our PADL 2019 submission, in consideration at the Journal of Functional Programming at the time of writing. It is reproduced here faithfully.

Part I

Caper

2

CAPER: Automatic Verification for Fine-grained Concurrency

The content of this chapter is adapted from [27]. Expository sections elided due to space limitations for publications have been included throughout, and a post-script, Section 2.9, has been included, detailing work on the project since time of publication.

2.1 Introduction

In recent years, much progress has been made in developing program logics for verifying the functional correctness of concurrent programs [24, 41, 72, 75, 81], with emphasis on fine-grained concurrent data structures. Reasoning about such programs is challenging since data is concurrently accessed by multiple threads: the reasoning must correctly account for interference between threads, which can often be subtle. Recent program logics address this challenge by using *resources* that are associated with some form of *protocol* for accessing shared data.

The concept of heap-as-resource was a fundamental innovation of separation logic [68]. It is possible to specify and verify a piece of code in terms of the resources that it uses. Further resources, which are preserved by the code, can be added by framing, provided that they are disjoint. Concurrent separation logic (CSL) [61] uses the observation that threads operating on disjoint resources do not interfere. This is embodied in the disjoint concurrency proof rule:

$$\frac{\left\{p_1\right\} c_1 \left\{q_1\right\} \qquad \left\{p_2\right\} c_2 \left\{q_2\right\}}{\left\{p_1 * p_2\right\} c_1 \| c_2 \left\{q_1 * q_2\right\}}$$

The *separating conjunction* connective '$*$' in the assertion $p_1 * p_2$ asserts that both p_1 and p_2 hold but for disjoint portions of the heap. In separation logic, the primitive resources are heap cells, represented $x \mapsto y$, meaning the heap at address x holds value y. A thread that owns a heap cell has an exclusive right to read, modify or dispose of it. The separating conjunction $x \mapsto y * y \mapsto z$ enforces disjointness: it requires x and y to be different addresses in the heap.

In fine-grained concurrent algorithms, however, threads use *shared* data, so a more flexible concept of resources is required. *Shared regions* [24] are one approach to this. A shared region encapsulates some underlying (concrete) resources, which may be accessed by multiple threads when they perform atomic operations. The region enforces a protocol that determines how threads can mutate the encapsulated resources. The region is associated with abstract resources called *guards* that determine the role that a thread can play in the protocol. Importantly, these resources determine what knowledge a thread can have about the region that is *stable* — i.e., continues to hold under the actions of other threads.

For example, consider a region that encapsulates a heap cell at address x. Associated with this region are two guards Inc and Dec. The protocol states that a thread with the Inc guard can increase the value stored at x, and a thread with the Dec guard can decrease the value stored at x. A thread holding the Inc guard can know that the value at x is *at most* the last value it observed; without the Dec guard, a thread cannot know that the value will not be decreased by another thread. Conversely, a thread holding the Dec guard can know a lower bound on the value at x. A thread that holds both guards can change the value arbitrarily and know it precisely, much as if it had the resource $x \mapsto y$ itself.

In this chapter we present Caper, a novel tool for automatic verification of fine-grained concurrent programs using separation logic. To verify a program, the user specifies the types of shared regions, defining their guards and protocols, and provides specifications for functions (and loop invariants) that use these regions. Caper uses a *region-aware* symbolic execution (§2.3.3) to verify the code, in the tradition of SmallFoot [8]. The key novelties of Caper's approach are:

- the use of *guard algebras* (§2.3.1) as a mechanism for representing and reasoning automatically about abstract resources, while supporting a range of concurrency verification patterns;
- techniques for automatically reasoning about interference on shared regions (§2.3.2), in particular, accounting for transitivity; and
- heuristics for non-deterministic proof search (§2.3), including the novel use of abduction to infer abstract updates to shared regions and guards.

We introduce our approach by considering a number of examples in §2.2. We emphasize that these examples are complete and self-contained — Caper can verify them without additional input. In §2.6 we evaluate Caper, reporting results for a range of examples. We discuss related work in §4.5 before concluding with remarks on future directions in §2.8.

The Caper tool is implemented in Haskell, and uses Z3 [19] and (optionally) E [69] to discharge proof obligations. The source code and examples are available [23], as is a soundness proof of the separation logic underlying Caper [26].

```
region SLock(r,x) {
  guards %LOCK * UNLOCK;
  interpretation {
    0 : x ↦ 0 &*& r@UNLOCK;
    1 : x ↦ 1;
  }
  actions {
    LOCK[_] : 0 ⇝ 1;
    UNLOCK : 1 ⇝ 0;
  }
}
function makeLock()
  requires true;
  ensures SLock(r,ret,0) &*& r@LOCK[1p]; {
    v := alloc(1);
    [v] := 0;
    return v;
}
function acquire(x)
  requires SLock(r,x,_) &*& r@LOCK[p];
  ensures SLock(r,x,1) &*& r@(LOCK[p] * UNLOCK); {
    b := CAS(x, 0, 1);
    if (b = 0) {
      acquire(x);
    }
}
function release(x)
  requires SLock(r,x,1) &*& r@UNLOCK;
  ensures SLock(r,x,_); {
    [x] := 0;
}
```

Figure 2.1: Caper listing for a spin lock implementation.

2.2 Motivating Examples

We begin by considering a series of examples that illustrate the programs and specifications that Caper is designed to prove. In each case, we discuss how Caper handles the example and why. In later sections, we will describe the rules and algorithms that underlie Caper in more detail. For each example, we give the *complete* source file, which Caper verifies with no further annotation.

2.2.1 *Spin Lock*

Figure 2.1 shows a typical annotated source file for Caper, which implements a simple fine-grained concurrent data structure: a spin lock. Note that &*& is Caper syntax for $*$ — separating conjunction.

Lines 1–11 define a *region type*, `SLock`, for spin locks. There are two kinds of assertions associated with regions, and their shape is dictated by region type definitions. `SLock(r,x,s)` is the assertion representing knowledge of a region `r` of type `SLock` with parameter `x` in abstract state `s`. The second are guard assertions of the form `r@(G)`, meaning we hold the guard `G` for region `r`.

Line 2 is a *guard algebra* declaration, indicating the guards associated with a given region type, and how they compose. There are two kinds of guard associated with `SLock` regions: `LOCK` guards, which are used to acquire the lock, and may be subdivided to allow multiple threads to compete for the lock (indicated by `%` in the `guards` declaration); and `UNLOCK` guards, which are used to release the lock, and are exclusive — only a thread holding the lock owns the `UNLOCK` guard.

Lines 3–6 declare a *region interpretation*: the resources held by the region when in each abstract state. `SLock` regions have two states: 0 represents that the lock is *available*, which is indicated concretely by the heap cell $\mathtt{x} \mapsto 0$; 1 represents that the lock has been *acquired*, which is indicated concretely by the heap cell $\mathtt{x} \mapsto 1$. In the *available* state the `UNLOCK` guard belongs to the region — a thread that transitions to the acquired state obtains this guard.

Finally, lines 7–10 declare the *actions* — the protocol governing the shared region. This embodies both the updates allowed for a given thread and the interference a thread must anticipate from the environment. A thread can transition an `SLock` region from abstract state 0 (available) to 1 (acquired) if it holds the `LOCK[p]` guard for any `p`. Similarly, a thread can transition an `SLock` region from acquired to available if it holds the `UNLOCK` guard.

The `makeLock` function allocates a new spin lock. It has the precondition `true` since it does not require any resources. In the postcondition, the function returns an `SLock` region with full permission to its `LOCK` guard (expressed by `r@(LOCK[1p])`). The logical variable `r` holds the identifier for the region, which is used to relate assertions that refer to the same region; it is implicitly existentially quantified as it occurs in the postcondition but neither in the precondition nor as a parameter to the function. The logical variable `ret` binds the return value, which is the address of the lock. When Caper symbolically executes the function body, at the `return` it has the resource $\mathtt{v} \mapsto 0$ but requires $\mathtt{SLock}(r, \mathtt{v}, 0)$. This missing resource is abduced: Caper backtracks searching for a place where it could have obtained the missing resource. Caper thus tries to construct the region before executing the `return` statement. Constructing the region consists of creating a fresh region identifier and adding the full guards for the region to the symbolic state (in this case `LOCK[1p] * UNLOCK`); the resources belonging to the region according to the interpretation are consumed (removed from the symbolic state). This is successful for the interpretation 0, leaving the guard `LOCK[1p]` for the new region. Caper can then successfully symbolically execute the `return` statement, since it has the resources required by the postcondition.

The `acquire` function attempts to acquire the lock. The precondition asserts that the spin lock is in an unknown state and that we have permission to acquire the lock in the form of the `LOCK[`p`]` guard. The postcondition asserts that the lock is in the acquired state (indicated by the 1 in the `SLock` predicate) and that we retain the `LOCK[`p`]` guard but have also acquired the `UNLOCK` guard.

The lock is acquired by performing an atomic compare-and-set (CAS) operation, which attempts to set the value stored at address `x` from 0 to 1. In symbolically executing the CAS, Caper determines that it needs to open the region because it does not have `x` $\mapsto -$. In opening the region, Caper branches on the interpretation of the region; it must show that both cases are correct. The CAS itself also introduces branches depending on whether it failed; these are quickly pruned as the CAS cannot fail if the region is in state 0, nor succeed if it is in state 1. Immediately after the atomic CAS, Caper must close the region. It does so by non-deterministically choosing among the interpretations.

If the initial state was 1, the CAS fails and Caper closes with state 1. Since the state is unchanged, this 'update' is permitted. After the atomic operation, Caper must stabilize the region; since the thread does not own the `UNLOCK` guard, another thread could transition the region to state 0. Consequently, after the CAS, Caper does not know which state the region is in. Since the CAS fails, the `if` condition succeeds. Caper then makes the recursive call to `acquire` using the specification, which allows it to obtain the postcondition in the end.

If the initial state was 0, the CAS succeeds and Caper closes with state 1. In doing so, the `UNLOCK` guard is acquired, since it is part of the interpretation of state 0, but not of state 1. Caper must then check that the update from state 0 to 1 is permitted by the available guards, `LOCK[`p`]` $*$ `UNLOCK`, which it is. After the CAS, the thread owns the `UNLOCK` guard so no other thread can change the state of the region, and so it is stable in state 1. The result of a successful CAS is 1, so Caper does not symbolically execute the body of the `if` in this case, and proceeds to check the postcondition, which is successful.

The verification of the `release` function proceeds along similar lines.

2.2.2 *Ticket Lock*

Figure 2.2 shows a Caper listing for a ticket lock. A ticket lock comprises two counters: the first records the next available ticket and the second records the ticket which currently holds the lock. (Note that the lock is "available" when the two counters are equal — i.e. the ticket holding the lock is the next available ticket.) To acquire the lock, a thread obtains a ticket by incrementing the first counter and waiting until the second counter reaches the value of the ticket it obtained. To release the lock, a thread simply increments the second counter.

In Caper, a ticket lock is captured by a `TLock` region, defined in lines 1–9 of Figure 2.2. In contrast to the `SLock` region, a `TLock` region has an infinite number of states: there is an abstract state for each integer `n`. The abstract state `n` of a `TLock` region represents the ticket that currently holds the lock. The guards associated with a `TLock` region represent the tickets: there is a unique guard `TICKET(n)` for each integer `n`. (This is indicated by the # in the `guards` declaration.) The region interpretation of state `n` ensures that:

- the first counter (`x`) is the next available ticket number, `m`, which is at least `n`;
- the second counter (`x+1`) is `n`, the lock-holding ticket number;
- all `TICKET` resources from `m` up belong to the region. (A set of indexed resources is expressed with a set-builder-style notation, as in `TICKET{k|k≥m}`.)

Note that `m` is implicitly existentially quantified in the interpretation.

A thread may acquire a ticket by incrementing the next-available-ticket counter and removing the corresponding `TICKET` guard from the region. Doing so does not affect the abstract state of the region, and can, therefore, happen at any time (no guards are required to do so). In order to increment the lock-holding-ticket counter, a thread must hold the `TICKET(n)` resource for the current value of the counter, `n`. We might, therefore, expect the `actions` declaration to be:

```
actions {
  TICKET(n) : n ⇝ n + 1;
}
```

This action declaration is, however, problematic for automation. Between symbolically executing atomic actions, Caper widens the set of possible abstract states for each region according to the *rely* relation for that region type. Suppose a `TLock` region is initially in state 0. If the thread does not hold the `TICKET(0)` guard, Caper must add the state 1 to the possible state set. If the thread does not hold `TICKET(1)`, Caper must add the state 2, and so on. In general, we cannot expect this widening process to terminate, so we must consider a *transitively-closed* rely relation. Caper cannot, in general, compute the transitive closure, but it *is* possible to *check* that a given `actions` declaration is transitively closed. We address this in §2.3.2. The proposed action declaration is, however, not transitive, since transitions from 0 to 1 and from 1 to 2 are possible, but the transitive transition from 0 to 2 is not possible in one step.

Instead, we use the `actions` declaration in Figure 2.2, which *is* transitively closed. It remedies the problem with the simple version by generalizing from a single increment to allow multiple increments.

```
region TLock(r,x) {
  guards #TICKET;
  interpretation {
    n : x ↦ m &*& (x + 1) ↦ n &*& r@TICKET{ k | k ≥ m } &*& m ≥
         n;
  }
  actions {
    n < m | TICKET{ k | n ≤ k, k < m } : n ⇝ m;
  }
}
function acquire(x)
  requires TLock(r,x,_);
  ensures TLock(r,x,n) &*& r@TICKET(n); {
    do {
        t := [x + 0];
        b := CAS(x + 0, t, t + 1);
    }
      invariant TLock(r,x,ni) &*& (b=0 ? true : r@TICKET(t) &*& t
           ≥ ni);
    while (b = 0);
    do {
        v := [x + 1];
    }
      invariant TLock(r,x,ni) &*& r@TICKET(t) &*& t ≥ ni &*& ni ≥
           v;
    while (v < t);
}
function release(x)
  requires TLock(r,x,n) &*& r@TICKET(n);
  ensures TLock(r,x,_); {
    v := [x + 1];
    [x + 1] := v + 1;
}
```

Figure 2.2: Caper listing for a ticket lock implementation.

This is achieved by placing a condition on the transition that `n`$\rightsquigarrow$`m` is only permitted when `n<m`, enforcing that the counter can only increase, as indicated before the vertical bar in the `actions` declaration. The guard `TICKET{k|n`$\leq$`k,k`$<$`m}` denotes the set of all guards `TICKET(k)` for `k` between `n` and `m-1` inclusive. This ensures that a thread can increment the counter past k only when it holds the `TICKET(`k`)` guard. For example, a thread holding `TICKET(n)` can transition the `TLock` region from abstract state `n` to `n+1`.

The precondition of `acquire` asserts that the ticket lock exists in some arbitrary abstract state. The postcondition ensures that the lock is in some state `n` and that the guard `TICKET(n)` has been acquired. The function contains two loops and Caper requires that we provide an invariant for each. The first loop, lines 13–18, increments the next available ticket. The invariant states that the region is in some state `ni` and that once the CAS succeeds ($\mathsf{b} = 1$) we have the `TICKET(t)` guard and `t` is at least `ni`; the conditional is expressed using the C-like `_?_:_` notation. Similarly to the `acquire` operation for the spin lock, Caper opens the region when symbolically executing the CAS operation. Since there is only one clause in the `TLock` region interpretation, Caper considers one generic case, rather than branching as in the spin lock. Immediately after symbolically executing the CAS operation, Caper needs to close the region. If the CAS succeeds Caper knows that the next available ticket `m` is `t+1`. Hence the guard `TICKET(t)` is not included in the set of guards `TICKET{k|k`$\geq$`m}` needed for closing the region, so Caper can transfer the guard `TICKET(t)` out of the `TLock` region. The next loop, lines 19–23, spins until the acquired ticket becomes the lock-holding ticket. Caper proceeds similarly to the first loop. After the loop, the invariant and failed loop test are sufficient to establish the postcondition.

The precondition of the `release` function expresses that the lock-holding ticket of the region is `n` and that we hold that ticket. Because we hold the guard `TICKET(n)`, we can make a transition from abstract state `n` to abstract state `n+1`. No other thread can make a transition, since to transition from `n` to `m` one needs to hold all the guards from `n` to `m-1`. Therefore, there is no interference from other threads on the second counter and we can update it without using a CAS loop. After the read on line 28, Caper knows that `v` holds value `n` by opening the region. To execute the write on line 29, Caper again opens the region in state `n`. Caper closes the region in a new state n_1, which must be the value of the `x+1` counter, i.e. $\mathsf{n}_1 = \mathsf{n} + 1$. The value of `m` is unchanged, but Caper must establish that $\mathsf{m} \geq \mathsf{n}_1 = \mathsf{n} + 1$, which follows from the fact that $\mathsf{m} \geq \mathsf{n}$ and that `TICKET(n)` (from the thread) is disjoint from `TICKET{k|k`$\geq$`m}` (from the region). Caper must also establish that the transition `n`$\rightsquigarrow$`n+1` is permitted by the actions for the available guards, which it is.

```
region Client(r,x,s,z) {
  guards 0;
  interpretation {
    0 : TLock(s,z,k) &*& (x ↦ a &*& (x+1) ↦ a ∨ s@TICKET(k));
  }
  actions {}
}
function set(x,z,w)
  requires Client(r,x,s,z,0) &*& TLock(s,z,_);
  ensures Client(r,x,s,z,0) &*& TLock(s,z,_); {
    acquire(z);
    [x] := w;
    [x + 1] := w;
    release(z);
}
```

Figure 2.3: A Caper listing of a client of the ticket lock.

2.2.2.1 *Client.*

Figure 2.3 shows an implementation of a simple client using the ticket lock. Here, the `Client(r,x,s,z)` region uses a ticket lock region `TLock(s,z)` to maintain the lock invariant that two cells, `x` and `x+1`, have the same value. The disjunction with the guard `s@TICKET(k)` makes it possible to temporarily break the invariant. The function `set(x,z,w)` sets the value of the two shared memory cells to `w`. Lines 12–13 are a critical section protected by the ticket lock. Note that the invariant is temporarily broken between the two writes.

In symbolically executing the call to `acquire` in line 11, Caper uses the postcondition of `acquire` to obtain `s@TICKET(k)` and `TLock(s,z,k)` for some `k`. When Caper symbolically executes line 12, it opens the `Client` region and must consider each of the disjuncts of the interpretation. It finds that the right-hand disjunct is not possible as we already hold the `TICKET` guard for the current abstract state `k` of the lock. Hence it obtains the points-to assertions for `x` and `x+1`, and can perform the write to `x`. Since the values stored at `x` and `x+1` are now different, Caper can only close the region by transferring `s@TICKET(k)` to the region. When Caper symbolically executes line 13, it again opens the `Client` region. This time it finds that the region holds `s@TICKET(k)` since we have the points-to predicates. After the assignment, the values stored at `x` and `x+1` are the same, and Caper can close the region while leaving us with the `s@TICKET(k)` resource which Caper then uses to satisfy the precondition of `release`.

2.2.3 *Stack-based Bag*

Figure 2.4 shows a Caper implementation of a concurrent bag based on Treiber's stack [74]. The stack is lock-free, and uses CAS operations

```
predicate bagInvariant(v);
region Bag(r,x) {
  guards 0;
  interpretation {
    0 : x ↦ y &*& BagList(s,y,_,_,0) &*& s@OWN;
  }
  actions {}
}
region BagList(s,y,v,z) {
  guards OWN;
  interpretation {
    0 : y = 0 ? true : y ↦ v &*& y+1 ↦ z
        &*& BagList(t,z,_,_,0) &*& t@OWN &*& bagInvariant(v);
    1 : s@OWN &*& y ↦ v &*& y+1 ↦ z &*& BagList(t,z,_,_,_);
  }
  actions {
    OWN : 0 ⇝ 1;
  }
}
function push(x,v)
  requires Bag(r,x,0) &*& bagInvariant(v);
  ensures Bag(r,x,0); {
    y := alloc(2); [y] := v;
    innerPush(x,y);
}
function innerPush(x,y)
  requires Bag(r,x,0) &*& y ↦ v &*& y+1 ↦ _ &*& bagInvariant(v)
      ;
  ensures Bag(r,x,0); {
    t := [x];
    [y + 1] := t;
    cr := CAS(x,t,y);
    if (cr = 0) {
      innerPush(x, y);
    }
}
function pop(x)
  requires Bag(r,x,0);
  ensures ret = 0 ? Bag(r,x,0) : Bag(r,x,0) &*& bagInvariant(ret)
      ; {
    t := [x];
    if (t = 0) { return 0; }
    t2 := [t + 1];
    cr := popCAS(x,t,t2);
    if (cr = 0) { ret := pop(x); return ret; }
    ret := [t];
    return ret;
}
function popCAS(x,t,t2)
  requires Bag(r,x,0) &*& BagList(rt,t,v,t2,_)
          &*& BagList(rt2,t2,_,_,_) &*& t != 0;
  ensures ret = 0 ∨ bagInvariant(v); {
    cr := CAS(x,t,t2); return cr;
}
```

Figure 2.4: Caper listing for a concurrent bag implementation.

to manipulate the head of a linked-list structure. To push a new item, a thread constructs a new head node and atomically updates the head pointer of the bag. When popping an item, a thread anticipates what the head node is before atomically updating the head pointer. In both cases, the function of the atomic compare-and-swap operation is to ensure that no other thread has manipulated the bag between operations. Unlike the preceding examples, the heap is fundamental to the implementation.

The specification is parametrised by an *abstract predicate* [9, 64] `bagInvariant` (line 1). The idea is that adding an item v to the bag requires transferring ownership of the predicate `bagInvariant`(v) to the bag, which is returned when the item is removed. Clients can decide how to instantiate `bagInvariant`.

In Caper, the head pointer and the linked-list nodes are encapsulated by separate regions: `Bag` and `BagList`, respectively. Note that there is an apparent hierarchy between `Bag` and `BagList` regions: a `Bag` refers to a `BagList`, which may, in turn, refer to another `BagList` and so on. In this way, we can use regions to model inductive data structures such as linked lists. While regions can fulfill a similar role to inductive predicates, they are semantically distinct. Regions are shared globally, and so may refer to each other in arbitrary, even cyclical ways. Although it appears as though the regions are nested, semantically all regions exist at the same level. In this example, we achieve a hierarchy through ownership of guards: the top-level `Bag` holds the `OWN` guard for the first `BagList`, which holds the `OWN` guard for the second, and so on.

A `Bag` region is simple, in that it has no guards and is always in one abstract state. It simply permits sharing of the resources it holds. The interpretation of abstract state 0 of a region `Bag(r,x)` holds a pointer to the first, possibly null, linked-list node at `x` in addition to its `OWN` guard.

The `BagList(s,y,v,z)` region type represents a list node (or a null terminator) with payload value `v` at address `y` and a pointer to the successor `z` at `y+1`. A `BagList` region is in one of two states, depending on whether it belongs to the bag or not. Abstract state 0 means that the region belongs to the bag, in which case it can represent either a null-pointer terminating the list or a list node with a value and successor, represented by another `BagList` region. The region also holds the `OWN` guard to the successor and the `bagInvariant` predicate for its value. The abstract state 1 represents a list node that has been popped. The interpretation therefore includes the region's own `OWN` guard and knowledge of the successor, but *not* the successor's `OWN` guard or the `bagInvariant` predicate.

Since the bag can be used concurrently, we do not specify exactly which elements it contains at a given time. Instead, our specification of push and pop focuses on ownership transfer of elements pushed to and popped from the bag.

The precondition of push asserts only knowledge of the bag and ownership of the bagInvariant for the value v to be put in the bag. In the postcondition, the bagInvariant resource is absent, as it is transferred to the bag. The push function allocates a new list node and then delegates to a CAS loop in innerPush.

The specification of innerPush is similar to that of push, but the precondition requires the list node that is to be pushed. Note that the successor of the node, y+1, is initially undetermined. In lines 29–30, innerPush loads the current head of the list into the tail pointer of the new node via the variable t. To do so, Caper opens the Bag region, getting access to x, and closes it again. The observed value is then written as the successor of the new node, at address y+1, without opening any regions. To account for the head of the stack having changed since it was read, a CAS is used to update the head pointer. To symbolically execute the CAS in line 31, Caper again opens the Bag region. If successful, it must close and restore the Bag region. This means a new BagList region in state 0 must be created for the new head. Upon creation, Caper creates the OWN guard for the new region, which is given to the Bag region, closing it in state 0. If the CAS does not succeed, it means another thread updated the stack between lines 29 and 31, and innerPush recursively tries again.

The specification of the pop function states that, from a Bag, pop produces either null (0), in case the bag is empty, or a value satisfying the bagInvariant predicate. The idea is that the concrete value comes from the underlying linked list, and the corresponding bagInvariant(v) predicate is removed from a BagList region. The pop function attempts to CAS the head pointer of the bag to the successor of the first link, effectively removing the first element from the bag. It first reads the head pointer into t, which requires opening the Bag region. At this point, we obtain the BagList region, which can be freely duplicated, although the OWN guard remains in the Bag region. After the Bag region is closed again, we must stabilize the BagList region to account for the fact that another thread could remove the head node from the stack. That is, its abstract state could now be 0 or 1. If the head pointer was 0, then the bag was empty and so 0 is returned. Otherwise, t points to a node, and line 41 reads its successor pointer into t2. This involves opening the BagList region previously obtained. It is not necessary to open the Bag region again at this stage. The call to popCAS at line 42 attempts to update the head node to the successor of the head node. We give this CAS operation a specification (lines 48–50) to assist Caper. To symbolically execute that CAS, Caper opens the Bag region containing x and the OWN guard for the BagList region for the head of the stack. The BagList region for the previously seen head (with region identifier rt) and the BagList region for the current head (if it is different) are also opened. Caper determines that the CAS can only succeed if these two regions are the same. In this case, the `bagInvariant` is transferred to the thread, the `OWN` guard for the successor is transferred to the `Bag` region, and the `OWN` guard for the head is transferred to its own `BagList` region. In this process, the state of the `BagList` region for the old head is updated from 0 to 1. This is allowed since we have access to its `OWN` guard. If the CAS fails, nothing is changed and the pop is retried. On success, the return value is read from the `BagList` region that is now in state 1. This value corresponds to the previously-obtained `bagInvariant` predicate.

2.3 Proof System

Caper's proof system is based on the logic of CAP [24], using improvements from iCAP [72] and TaDA [81]. The logic is a separation logic with shared regions.

Each shared region has a unique *region identifier*. A region has an associated *region type*, which determines the resources and protocol associated with the region. Region types $\mathbf{T}(r, \bar{x})$ are parametrised, with the first parameter (r) always being the region identifier. A region also has an *abstract state*. In Caper's logic, *region assertions* describe the type and state of a region. The region assertion $\mathbf{T}(r, \bar{x}, y)$ asserts the existence of a region with identifier r and type $\mathbf{T}(r, \bar{x})$ in abstract state y. Region assertions are freely duplicable; i.e., they satisfy the equivalence: $\mathbf{T}(r, \bar{x}, y) \iff \mathbf{T}(r, \bar{x}, y) * \mathbf{T}(r, \bar{x}, y)$. Moreover, region assertions with the same region identifier must agree on the region type and abstract state: $\mathbf{T}(r, \bar{x}, y) * \mathbf{T}'(r, \bar{x}', y') \implies (\mathbf{T}, \bar{x}, y) = (\mathbf{T}', \bar{x}', y')$.

Shared regions are also associated with (ghost) resources called *guards*. Which guards can be associated with a region, as well as their significance and behavior is determined by the type of the region. Guards are interpreted as elements of a partial commutative monoid (PCM), referred to as a *guard algebra*. That is, they have a partial composition operator that is associative, commutative and has a unit. This is sufficient for them to behave as separation logic resources (see e.g. [25]). In Caper's logic, *guard assertions* assert ownership of guard resources. The guard assertion $r@(G_1 * \ldots * G_n)$ asserts ownership of the guards $G_1, \ldots, G_n$ associated with region identifier r. Guard assertions are not in general duplicable, but they distribute with respect to $*$: $r@(G_1 * G_2) \iff r@G_1 * r@G_2$.

The region type determines how a shared region is used. A region type definition determines the following properties of regions of that type: the guard algebra associated with the region; the abstract states of the region and their concrete *interpretation*; and the *actions* that can be used to update the state of the region, and which guards are required in order to perform each action. Region type definitions determine two derived relations, Rely and Guar (for *guarantee*), which

$$\frac{(P(\bar{z})|G : e_1(\bar{z}) \rightsquigarrow e_2(\bar{z})) \in \mathrm{Actions}(\mathbf{T}(r,\bar{x})) \qquad P(\bar{w})}{(e_1(\bar{w}), e_2(\bar{w})) \in \mathrm{Guar}(\mathbf{T}(r,\bar{x}), G)}$$

$$\frac{}{(x,x) \in \mathrm{Guar}(\mathbf{T}(r,\bar{x}), G)} \qquad \frac{(x,y),(y,z) \in \mathrm{Guar}(\mathbf{T}(r,\bar{x}), G)}{(x,z) \in \mathrm{Guar}(\mathbf{T}(r,\bar{x}), G)}\ (\dagger)$$

$$\frac{(x,y) \in \mathrm{Guar}(\mathbf{T}(r,\bar{x}), G)}{(x,y) \in \mathrm{Guar}(\mathbf{T}(r,\bar{x}), G * G')}$$

$$\frac{(P(\bar{z})|G : e_1(\bar{z}) \rightsquigarrow e_2(\bar{z})) \in \mathrm{Actions}(\mathbf{T}(r,\bar{x})) \qquad P(\bar{w}) \qquad G * G' \text{ defined}}{(e_1(\bar{w}), e_2(\bar{w})) \in \mathrm{Rely}(\mathbf{T}(r,\bar{x}), G')}$$

$$\frac{(x,y),(y,z) \in \mathrm{Rely}(\mathbf{T}(r,\bar{x}), G)}{(x,z) \in \mathrm{Rely}(\mathbf{T}(r,\bar{x}), G)}\ (\dagger) \qquad \frac{(x,y) \in \mathrm{Rely}(\mathbf{T}(r,\bar{x}), G * G')}{(x,y) \in \mathrm{Rely}(\mathbf{T}(r,\bar{x}), G)}$$

$$\frac{}{(x,x) \in \mathrm{Rely}(\mathbf{T}(r,\bar{x}), G)}$$

Figure 2.5: Rules defining the Guar and Rely relations.

are defined in Figure 2.5, based on the actions for the region types. The relation $\mathrm{Rely}(\mathbf{T}(r,\bar{x}), G)$ consists of all state transitions that a thread's environment may make to a region of type $\mathbf{T}(r,\bar{x})$, if the thread owns guard G. The relation $\mathrm{Guar}(\mathbf{T}(r,\bar{x}), G)$ consists of all state transitions that a thread itself may make to a region of type $\mathbf{T}(r,\bar{x})$, if it owns guard G.

2.3.1 *Guards*

The underlying logic of Caper permits the guard algebra for a region to be an arbitrary PCM. However, in order to reason automatically about guards, Caper must be able to effectively compute solutions to certain problems within the PCM. To this end, Caper provides a number of constructors for guard algebras for which these problems are soluble. These constructors are inspired by common patterns in concurrency verification, and are useful for many examples. The three automation problems are as follows.

Frame Inference. Given guard assertions `A` and `B`, find a `C` (if it exists) such that $\mathtt{A} \vdash \mathtt{B} * \mathtt{C}$. This problem has two applications:

- Computing the Guar relation requires determining if the guard currently available to the thread (`A`) entails the guard required to perform some action (`B`);
- At call sites, returns and when closing regions, symbolic execution consumes assertions (i.e. checks that the assertion holds and

removes the corresponding resources from the symbolic state). Frame inference (for guards) does this for guard assertions.

Composition and Compatibility. Given guard assertions `A` and `B`, determine their composition `A` $*$ `B` and the condition for it being defined. Whether it is defined will be a pure assertion on the free variables of `A` and `B`. This also has two applications:

- Computing the Rely relation requires determining when the guard currently owned by the thread (`A`) is compatible with the guard required to perform some action (`B`);
- At entry points, after calls and when opening regions, symbolic execution produces assertions (i.e. adds resources and assumptions corresponding to the assertion to the symbolic state). Composition does this for guards.

Least Upper-bounds. Given guard assertions `A` and `B`, compute `C` such that `C` $\vdash$ `A`, `C` $\vdash$ `B` and for any `D` with `D` $\vdash$ `A` and `D` $\vdash$ `B`, `D` $\vdash$ `C`. This is used to compose two actions, which will be guarded by the least upper-bound of the two guards.

2.3.1.1 *Supported Guard Algebras.*

We present the guard algebras that are supported by Caper. Each guard algebra has a maximal element, the *full guard*, which is generated for a region when it is initially created.

Trivial guard algebra. The trivial guard algebra, `0` in Caper syntax, consists of one element which is the unit. This algebra is used when a region has no roles associated with it: it can be used in the same way by all threads at all times.

All-or-nothing guard algebra. An all-or-nothing guard algebra consists of a single element distinct from the unit, which is the full guard. In Caper syntax, this algebra is represented by the name chosen for the point; for instance, `GUARD` would be the all-or-nothing algebra with point `GUARD`.

Permissions guard algebra. A permissions guard algebra `%GUARD` has the full resource `GUARD[1p]` which can be subdivided into smaller permissions `GUARD[`π`]`. The typical model for permissions is as fractions in the interval $[0,1]$. This allows for (non-zero) permissions to be split arbitrarily often. Caper implements a different theory of permissions. This theory also allows arbitrary splitting, but also requires that `GUARD[`π`]` $*$ `GUARD[`π`]` is undefined for any non-zero π.

The theory can be encoded into the theory of *atomless Boolean algebras* — a Boolean algebra is said to be atomless if for all $a > \bot$ there exists a' with $a > a' > \bot$. The encoding defines the PCM operator as $p_1 * p_2 = p_1 \vee p_2$ if $p_1 \wedge p_2 = \bot$ (and undefined otherwise). Conveniently, the first-order theory of atomless Boolean algebras is complete and therefore decidable (initially reported by Tarski [73], proved by Ershov [31], and see e.g. [15] for details).

CAPER implements three different proof procedures for the theory of permissions. One uses the encoding with Boolean algebras and passes the problem to the first-order theorem prover E [69]. A second checks for the satisfiability of a first-order permissions formula directly. The third encodes the satisfiability problem with bit-vectors and passes it to the SMT solver Z3 [19]. Dockins et al. [28] previously proposed a tree-share model of permissions, which is also a model of this theory. Le et al. [50] have developed decision procedures for entailment checking based on this model, which could also be used by CAPER.

Counting guard algebra. A counting guard algebra `|GUARD|` consists of *counting guards* similar to the counting permissions of Bornat et al. [10]. For $n \geq 0$, `GUARD|`n`|` expresses n counting guards. An *authority guard* tracks the number of counting guards that have been issued. For $n \geq 0$, `GUARD|-1` $- n$`|` expresses the authority guard with n counting guards issued. The PCM operator is defined as:

$$\texttt{GUARD|}n\texttt{|} * \texttt{GUARD|}m\texttt{|} = \texttt{GUARD|}n+m\texttt{|} \quad \text{if } (n \geq 0 \wedge m \geq 0) \vee \\ (n < 0 \wedge m \geq 0 \wedge n+m < 0) \vee (n \geq 0 \wedge m < 0 \wedge n+m < 0).$$

This ensures that the authority is unique (e.g. `GUARD|-1|` $*$ `GUARD|-2|` is undefined) and that owning `GUARD|-1|`, which is the full guard, guarantees that no other thread may have a counting guard.

Indexed guard algebra. An indexed guard algebra `#GUARD` consists of sets of individual guards `GUARD(`n`)` where n ranges over integers. A set of such individual guards is expressed using a set-builder notation: `GUARD`$\{x \mid P\}$ describes the set of all guards `GUARD`(x) for which P holds of x. The full guard is `GUARD`$\{x \mid \texttt{true}\}$. The notation `GUARD`(n) is syntax for `GUARD`$\{x \mid x = n\}$. The PCM operator is defined as `GUARD`$S_1 *$ `GUARD`$S_2 =$ `GUARD`$(S_1 \cup S_2)$ if $S_1 \cap S_2 = \emptyset$.

The automation problems reduce to testing conditions concerning sets, specifically set inclusion. Sets in CAPER are not first-class entities: they are always described by a logical predicate that is a (quantifier-free) arithmetic formula. For sets characterized in this way, set inclusion can be characterized as: $\{x \mid P\} \subseteq \{x \mid Q\} \iff \forall x. P \Rightarrow Q$. Advanced SMT solvers, such as Z3 [19], have support for first-order quantification, and CAPER exploits this facility to handle the verification conditions concerning set inclusion.

Product guard algebra construction. Given guard algebras M and N, the product construction $M * N$ consists of pairs $(m, n) \in M \times N$, where the PCM operation is defined pointwise: $(m_1, n_1) * (m_2, n_2) = (m_1 * m_2, n_1 * n_2)$. The unit is the pair of units and the full resource is the pair of full resources.

Sum guard algebra construction. Given guard algebras M and N, the sum construction $M + N$ is the discriminated (or disjoint) union of M and N, up to identifying the units and identifying the full resources of the two PCMs. The PCM operation embeds the operations of each of the constituent PCMs, with composition between elements of different PCMs undefined.

Within the Caper implementation, guards are represented as maps from guard names to parameters that depend on the guard type. For this to work, Caper disallows multiple guards with the same name in a guard algebra definition. For instance, `INSERT * %INSERT` is not legal. The sum construction is implemented by rewriting where necessary by the identities the construction introduces.

2.3.2 *Interference Reasoning*

There are two sides to interference: on one side, a thread should only perform actions that are anticipated by the environment, expressed by the Guar relation; on the other, a thread must anticipate all actions that the environment could perform, expressed by the Rely relation. Each time a thread updates a region, it must ensure that the update is permitted by the Guar with respect to the guards it owns (initially) for that region. Moreover, the symbolic state between operations and frames for non-atomic operations must be *stabilized* by closing the set of states they might be in under the Rely relation. Caper must therefore be able to compute with these relations effectively.

The biggest obstacle to effective computation is that the relations are transitively closed. Transitivity is necessary, at least for Rely, since the environment may take arbitrarily many steps in between the commands of a thread. However, computing the transitive closure in general is a difficult problem. For instance, consider a region that has the (unguarded) action `: n ~> n + 1`. From this action, we should infer the relation $\{(n, m) \mid n \leq m\}$, as the reflexive-transitive closure of $\{(n, n+1) \mid n \in \mathbb{Z}\}$. It is generally beyond the ability of SMT solvers to compute transitive closures, although some (limited) approaches have been proposed [30].

Caper employs two techniques to deal with the transitive closure problem. This first is that, if the state space of the region is finite, then it is possible to compute the transitive closure directly. Caper uses a variant of the Floyd-Warshall algorithm [32] for computing shortest paths in a weighted graph. The 'weights' are constraints (first-order formulae) with conjunction as the 'addition' operation and disjunction as the 'minimum' operation.

The second technique is to add composed actions until the set of actions is transitively closed. When the actions are transitively closed, the Rely and Guar relations can be computed without further accounting for transitivity (i.e. the (†) rules in Figure 2.5 can be ignored). For two actions $P \mid G : e_1 \rightsquigarrow e_2$ and $P' \mid G' : e'_1 \rightsquigarrow e'_2$ (assuming the only common variables are region parameters) their composition is $P, P', e_2 = e'_1 \mid G \sqcup G' : e_1 \rightsquigarrow e'_2$ where $\sqcup$ is the least-upper-bound operation on guards. Using frame inference for guards, we can check if one action subsumes another — that is, whether any transition permitted by the second is also permitted by the first.

CAPER uses the following process to reach a transitive set of actions. First, consider the composition of each pair of actions currently in the set and determine if it is subsumed by any action in the set. If all compositions are already subsumed then the set is transitive. Otherwise, add all compositions that are not subsumed and repeat. Since this process is not guaranteed to terminate (for example, for `n ⇝ n + 1`), CAPER will give up after a fixed number of iterations fail to reach a transitive set. Note that adding composite actions does not change the Rely and Guar relations, since these are defined to be transitively closed.

If CAPER is unable to reach a transitive set of actions, the Rely relation is over-approximated by the universal relation, while the Guar is under-approximated. This is sound, since CAPER can prove strictly less in such circumstances, although the over-approximation is generally too much to prove many examples.

It is often practical to represent the transition system for a region type in a way that CAPER can determine its transitive closure. For example, instead of the action `: n ⇝ n + 1` we can give the action `n < m | : n ⇝ m`, which CAPER can prove subsumes composition with itself. Since CAPER tries to find a transitive closure, it is often unnecessary to provide a set of actions that is transitively closed. For instance, given the actions `0 ≤n, n <m | A : n ⇝-m` and `0 <n | B : -n ⇝n`, CAPER adds the following actions to reach a transitive closure:

```
    0 ≤ n, n < m | A * B : n ⇝ m;
    0 < n, n < m | A * B : -n ⇝ -m;
    0 < n, n < m | A * B : -n ⇝ m;
```

2.3.3 *Symbolic Execution*

CAPER's proof system is based on symbolic execution, where programs are interpreted over a domain of symbolic states. Symbolic states represent separation logic assertions, but are distinct from the syntactic assertions of the CAPER input language. To verify that code satisfies a specification, the code is symbolically executed from a symbolic state corresponding to the precondition. The symbolic execution may be non-deterministic — for instance, to account for conditional statements

— and so produces a set of resulting symbolic states. If each of these symbolic states entails the postcondition, then the code satisfies the specification.

A symbolic state $S = (\Delta, \Pi, \Sigma, \Xi, \Upsilon) \in \mathsf{SState}$ consists of:

- $\Delta \in \mathsf{VarCtx} = \mathsf{SVar} \xrightharpoonup{\text{fin}} \mathsf{Sort}$, a *variable context* associating logical sorts with symbolic variables;
- $\Pi \in \mathsf{Pure} = \mathsf{Cond}^*$, a context of *pure conditions* (over the symbolic variables) representing logical assumptions;
- $\Sigma \in \mathsf{Preds} = \mathsf{Pred}^*$, a context of *predicates* (over the symbolic variables) representing owned resources;
- $\Xi \in \mathsf{Regions} = \mathsf{RId} \xrightharpoonup{\text{fin}} \mathsf{RType}_\bot \times \mathsf{Exp}_\bot \times \mathsf{Guard}$, a finite map of region identifiers to an (optional) region type, an (optional) expression representing the state of the region, and an guard expression representing the owned guards for the region;
- $\Upsilon \in \mathsf{ProgVars} = \mathsf{ProgVar} \xrightharpoonup{\text{fin}} \mathsf{Exp}$, a map from program variables to expressions representing their current values.

We take the set of symbolic variables SVar to be countably infinite. Symbolic variables are considered distinct from program variables (ProgVar), which occur in the syntax of the program code (Stmt), and assertion variables (AssnVar), which occur in syntactic assertions (Assn). Currently, the set of sorts supported by Caper is $\mathsf{Sort} = \{\mathbf{Val}, \mathbf{Perm}, \mathbf{RId}\}$. That is, a variable can represent a program value (i.e. an integer), a permission, or a region identifier.

We do not formally define the syntax of (symbolic) expressions (Exp) and conditions (Cond). Expressions include symbolic variables, as well as arithmetic operators and operators on permissions. Conditions include a number of relational propositions over expressions, such as equality and inequality ($<$). They can also express rely and guarantee relations and inclusions between sets. A context of pure conditions is a sequence of conditions, interpreted as a conjunction. Symbolic execution generates entailment between contexts of conditions as verification conditions. The practical limitation on conditions is that these entailments should be checkable automatically by means of provers such as SMT solvers.

A (spatial) predicate $P(\bar{e}) \in \mathsf{Pred}$ consists of a predicate name P and a list of expressions $\bar{e}$. Two types of predicates are given special treatment and have their own syntax: individual heap cells $a \mapsto b$ (where a is the address and b the value stored), and blocks of heap cells $a \mapsto \#\mathrm{cells}(n)$ (where a is the starting address and n is the number of consecutive heap cells). All other predicates are abstract.

A region map associates region identifiers with knowledge and resources for the given region. The knowledge consists of the type of

$$\frac{\begin{array}{c} \mathrm{dom}(\gamma) = S \\ \mathrm{range}(\gamma) = \Gamma \qquad \forall x, y \in S.\, \gamma(x) = \gamma(y) \implies x = y \qquad \Delta \cap \Gamma = \emptyset \end{array}}{\mathit{freshSub}(\Delta, \Gamma, S, \gamma)}$$

$$\frac{\begin{array}{c} s'_1, \ldots, s'_n \notin \Delta \qquad \Xi = [r_1 \mapsto (t_1, s_1, G_1), \ldots, r_n \mapsto (t_n, s_n, G_n)] \\ P = (s_1, s'_1) \in \mathrm{Rely}(t_1, G_1); \ldots ; (s_n, s'_n) \in \mathrm{Rely}(t_n, G_n) \\ \Xi' = [r_1 \mapsto (t_1, s'_1, G_1), \ldots, r_n \mapsto (t_n, s'_n, G_n)] \end{array}}{\mathit{stabilise}(\Delta, \Xi, s'_1 : \mathbf{Val}; \ldots ; s'_n : \mathbf{Val}, P, \Xi')}$$

$$\frac{\begin{array}{c} \Phi(f) = (\bar{x}, A_{\mathrm{pre}}, A_{\mathrm{post}}) \qquad \mathit{freshSub}(\varepsilon, \Delta, \mathrm{vars}(A_{\mathrm{pre}}) \cup \bar{x}, \gamma) \\ \mathit{produce}(A_{\mathrm{pre}}, \gamma) : (\Delta, \varepsilon, \varepsilon, \emptyset) \rightsquigarrow \mathbb{S}_0 \\ \forall (\Delta_0, \Pi_0, \Sigma_0, \Xi_0) \in \mathbb{S}_0.\, \exists \mathbb{S}_1.\, \vdash C : (\Delta_0, \Pi_0, \Sigma_0, \Xi_0, [\bar{x} \mapsto \gamma(\bar{x})]) \rightsquigarrow \mathbb{S}_1 \\ \forall (\Delta_1, \Pi_1, \Sigma_1, \Xi_1, \Upsilon_1) \in \mathbb{S}_1.\, \exists \Delta'_1, \Pi'_1, \Xi'_1.\, \mathit{stabilise}(\Delta_1, \Xi_1, \Delta'_1, \Pi'_1, \Xi'_1) \\ \exists \Gamma_1, \gamma'.\, \mathit{freshSub}(\Delta_1; \Delta'_1, \Gamma_1, \mathrm{vars}(A_{\mathrm{post}}) \setminus (\mathrm{vars}(A_{\mathrm{pre}}) \cup \bar{x}), \gamma') \\ \exists \mathbb{S}_2.\, \mathit{consume}(A_{\mathrm{post}}, \gamma \cup \gamma') : (\Delta_1; \Delta'_1, \Pi_1; \Pi'_1, \Gamma_1, \varepsilon, \Sigma_1, \Xi'_1) \rightsquigarrow \mathbb{S}_2 \\ \forall (\Delta_2, \Pi_2, \Gamma_2, P_2, \Sigma_2, \Xi_2) \in \mathbb{S}_2.\, \Delta_2, \Pi_2 \vdash \Gamma_2, P_2 \end{array}}{\Psi, \Phi \vdash \texttt{function}\ f(\bar{x})\{C\}}$$

Figure 2.6: Function correctness judgement.

the region, which is a pair of a region type name and list of expressions representing the parameters, and an expression describing the current state of the region. The resources consist of a guard. It is possible to have a guard for a region without knowing the type or state of the region, so these two components can be unspecified ($\perp$).

Figure 2.6 gives the correctness judgment for functions that forms the basis of CAPER's proof system. The judgment is parametrised by a context of region declarations Ψ, and a context of function specifications Φ. (Both contexts are left implicit in the sub-judgments.) The conditions break down into four key steps:

1. A symbolic state is generated corresponding to the function's precondition A_{pre}. This is captured by the judgment

$$\mathit{produce}(A_{\mathrm{pre}}, \gamma) : (\Delta, \varepsilon, \varepsilon, \emptyset) \rightsquigarrow \mathbb{S}_0$$

   which adds resources and assumptions to an initially empty symbolic state.

2. The body of the function is symbolically executed. This is captured by the judgment

$$\vdash C : (\Delta_0, \Pi_0, \Sigma_0, \Xi_0, [\bar{x} \mapsto \gamma(\bar{x})]) \rightsquigarrow \mathbb{S}_1.$$

3. The regions of the resulting symbolic states are stabilized to account for possible interference from other threads. This is captured by the judgment

$$\mathit{stabilise}(\Delta_1, \Xi_1, \Delta'_1, \Pi'_1, \Xi'_1).$$

(Note that stabilization also occurs at each interleaving step in the symbolic execution.)

4. Each final symbolic state is checked against the postcondition. This is captured by the judgment

$$\mathit{consume}(A_{\mathrm{post}}, \gamma \cup \gamma') : (\Delta_1, \Pi_1, \Gamma, \varepsilon, \Sigma_1, \Xi_1) \rightsquigarrow \mathbb{S}_2$$

which removes resources and generates verification conditions that are sufficient for the symbolic state to entail the postcondition. These verification conditions are checked by the judgment $\Delta_2, \Pi_2 \vdash \Gamma_2, P_2$.

Step 1 uses the judgment *produce* $\subseteq (\mathsf{Assn} \times (\mathsf{AssnVar} \rightharpoonup \mathsf{Exp})) \times \overline{\mathsf{SState}} \times \mathcal{P}(\overline{\mathsf{SState}})$, where $\overline{\mathsf{SState}} = \mathsf{VarCtx} \times \mathsf{Pure} \times \mathsf{Preds} \times \mathsf{Regions}$. The *produce* judgment (we adopt the produce/consume nomenclature of Verifast [39]) adds resources and assumptions to the symbolic state corresponding to a given syntactic assertion. It is parametrised by a substitution from assertion variables to expressions. In producing the precondition, this substitution maps the assertion variables occurring in the precondition and the function parameters (treated as assertion variables) to fresh symbolic variables. This is captured by the *freshSub* judgment. These fresh symbolic variables are bound in the initial variable context (Δ), while the initial context of conditions (ε), context of predicates (ε) and region map ($\emptyset$) are all empty. The judgment produces a set of symbolic states (sans program variable context). This set should be interpreted disjunctively: each of the symbolic states is possible after producing the assertion.

Step 2 uses the symbolic execution judgment: $(\vdash - : - \rightsquigarrow -) \subseteq \mathsf{Stmt} \times \mathsf{SState} \times \mathcal{P}(\mathsf{SState})$. This judgment updates the symbolic state according to the symbolic execution rules for program statements. The initial program variable context is given by mapping the function parameters $\bar{x}$ (treated as program variables) to the corresponding logical expressions $\gamma(\bar{x})$.

Step 3 uses the judgment *stabilise* $\subseteq \mathsf{VarCtx} \times \mathsf{Regions} \times \mathsf{VarCtx} \times \mathsf{Pure} \times \mathsf{Regions}$. This judgment relates an initial region map (in a given context) with a new region map that accounts for interference, with an extended context and additional pure conditions. The judgment is defined by the rule given in Figure 2.6. This rule creates a fresh variable (s'_i) to represent the new state of each region and asserts that it is related to the old state (s_i) in accordance with the rely relation for the given region. To account for the region type or state being unknown, we extend the definition of Rely with the following two rules:

$$\frac{}{(x, y) \in \mathrm{Rely}(\bot, G)} \qquad \frac{}{(\bot, y) \in \mathrm{Rely}(\mathbf{T}(r, \bar{x}), G)}$$

Step 4 uses the judgment *consume* $\subseteq (\mathsf{Assn} \times (\mathsf{AssnVar} \rightharpoonup \mathsf{Exp})) \times \widehat{\mathsf{SState}} \times \mathcal{P}(\widehat{\mathsf{SState}})$, where $\widehat{\mathsf{SState}} = \mathsf{VarCtx} \times \mathsf{Pure} \times \mathsf{VarCtx} \times \mathsf{Pure} \times$

Preds $\times$ Regions. The *consume* judgment removes resources and adds assertions to the symbolic state. The symbolic state is extended with a second variable context representing existentially quantified variables and a second context of pure conditions representing logical assertions. As an example, consuming the assertion `x |-> 2` where the predicates include $a \mapsto b$ can remove that predicate, adding the assertions $[\![\mathtt{x}]\!]_\gamma = a$ and $2 = b$ (where γ is the assertion variable substitution). Any assertion variables occurring in the postcondition that are neither parameters of the function nor occur in the precondition are treated as existentially quantified. The *freshSub* judgment is used again to generate a context and substitution for these variables.

It remains to check that the assertions arising from consuming the postcondition follow from the assumptions. This is achieved with the entailment judgment: $(-, - \vdash -, -) \subseteq \mathsf{VarCtx} \times \mathsf{Pure} \times \mathsf{VarCtx} \times \mathsf{Pure}$. The judgment is defined by:

$$\Delta, \Pi \vdash \Gamma, P \quad \overset{\text{def}}{\iff} \quad \forall \delta \in [\![\Delta]\!] \,.\, [\![\Pi]\!]_\delta \implies \exists \delta' \in [\![\Gamma]\!] \,.\, [\![P]\!]_{\delta \cup \delta'}$$

Here, $[\![\Delta]\!]$ is the set of variable assignments agreeing with context Δ and $[\![\Pi]\!]_\delta$ is the valuation of the conjunction of the conditions Π in the variable assignment δ.

The *produce* judgment. The rules for the *produce* judgment are given in Figure 2.7. The rules follow the syntax of the assertion to be produced. For a separating conjunction (`&*&`), first the left assertion is produced and then the right. Producing a conditional expression (`?:`) introduces non-determinism: we generate cases for whether the condition is true or false. For the true case, the first assertion is produced together with the condition; for the false case, the second assertion is produced together with the negated condition. Note that this non-determinism is demonic, in that the proof must deal with all cases. Producing a pure assertion simply adds it to the logical assumptions (interpreting the assertion variables through the substitution γ). Producing a predicate assertion adds it to the predicate context. As a special case, the points-to predicate also adds logical assumptions, expressed by $\mathcal{C}_{\text{cell}}$, which express that addresses must be positive and no two cells can have the same address (we elide the formal definition here).

The remaining two rules concern regions: producing a region and a guard assertion respectively. In each case, a region descriptor r_0 is created — in the first case, including the region type and state but the empty guard, and in the second case, including a guard but no region type or state. We non-deterministically consider two cases: when the region identified by z already exists in the symbolic state, and when it represents a completely fresh region. The first case is handled by rmerge, which non-deterministically merges the region with one of the existing regions. The second case is handled by rnew, which associates the region with a fresh identifier (i) and adds assumptions that this

$$\frac{\begin{array}{c}\mathit{produce}(A_1,\gamma) : S \rightsquigarrow \{S_i\}_{i\in I}\\ \forall i \in I.\, \mathit{produce}(A_2,\gamma) : S_i \rightsquigarrow \left\{S_{i,j}\right\}_{j\in J_i}\end{array}}{\mathit{produce}(A_1 \;\&{*}\&\; A_2,\gamma) : S \rightsquigarrow \left\{S_{i,j} \,\middle|\, i \in I, j \in J_i\right\}}$$

$$\frac{\begin{array}{c}\mathit{produce}(A_1,\gamma) : (\Delta,\Pi;\llbracket p\rrbracket_\gamma,\Sigma,\Xi) \rightsquigarrow \mathbb{S}_1\\ \mathit{produce}(A_2,\gamma) : (\Delta,\Pi;\neg\llbracket p\rrbracket_\gamma,\Sigma,\Xi) \rightsquigarrow \mathbb{S}_2\end{array}}{\mathit{produce}(p \mathbin{?} A_1 : A_2,\gamma) : (\Delta,\Pi,\Sigma,\Xi) \rightsquigarrow \mathbb{S}_1 \cup \mathbb{S}_2}$$

$$\frac{}{\mathit{produce}(p,\gamma) : (\Delta,\Pi,\Sigma,\Xi) \rightsquigarrow (\Delta,\Pi;\llbracket p\rrbracket_\gamma,\Sigma,\Xi)}$$

$$\frac{}{\mathit{produce}(P(\bar{e}),\gamma) : (\Delta,\Pi,\Sigma,\Xi) \rightsquigarrow (\Delta,\Pi,\Sigma;P(\llbracket\bar{e}\rrbracket_\gamma),\Xi)}$$

$$\frac{}{\mathit{produce}(e_1 \mapsto e_2,\gamma) : (\Delta,\Pi,\Sigma,\Xi) \rightsquigarrow (\Delta,\Pi;\mathcal{C}_{\text{cell}}(\llbracket e_1\rrbracket_\gamma,\Sigma),\Sigma;\llbracket e_1\rrbracket_\gamma \mapsto \llbracket e_2\rrbracket_\gamma,\Xi)}$$

$$\frac{\begin{array}{c}r_0 = (\mathbf{T}(\llbracket z,\bar{e}\rrbracket_\gamma),\llbracket s\rrbracket_\gamma,0) \qquad \mathbb{S}_1 = \text{rmerge}(\Delta,\Pi,\Sigma,\Xi,\llbracket z\rrbracket_\gamma,r_0)\\ i \notin \text{dom}(\Xi) \qquad \mathbb{S}_2 = \text{rnew}(\Delta,\Pi,\Sigma,\Xi,\llbracket z\rrbracket_\gamma,r_0,i)\end{array}}{\mathit{produce}(\mathbf{T}(z,\bar{e},s),\gamma) : (\Delta,\Pi,\Sigma,\Xi) \rightsquigarrow \mathbb{S}_1 \cup \mathbb{S}_2}$$

$$\frac{\begin{array}{c}r_0 = (\bot,\bot,\llbracket G\rrbracket_\gamma) \qquad \mathbb{S}_1 = \text{rmerge}(\Delta,\Pi,\Sigma,\Xi,\llbracket z\rrbracket_\gamma,r_0)\\ i \notin \text{dom}(\Xi) \qquad \mathbb{S}_2 = \text{rnew}(\Delta,\Pi,\Sigma,\Xi,\llbracket z\rrbracket_\gamma,r_0,i)\end{array}}{\mathit{produce}(z@(G),\gamma) : (\Delta,\Pi,\Sigma,\Xi) \rightsquigarrow \mathbb{S}_1 \cup \mathbb{S}_2}$$

$$\begin{array}{l}\text{rmerge}(\Delta,\Pi,\Sigma,\Xi,a,r_0) =\\ \qquad \left\{(\Delta,\Pi;a = i';\Pi',\Sigma,\Xi[i' \mapsto r']) \,\middle|\, \exists r.\, \Xi(i) = r \wedge \mathit{mergeRegion}(r,r_0,\Pi',r')\right\}\\ \text{rnew}(\Delta,\Pi,\Sigma,\Xi,a,r_0,i) = \left\{(\Delta,\Pi;a = i;\bigwedge\{i \neq i' \mid i' \in \text{dom}(\Xi)\},\Sigma,\Xi[i \mapsto r_0])\right\}\end{array}$$

Figure 2.7: Selected rules for the *produce* judgment.

$$\dfrac{\begin{array}{c}(A, q) \in \{(A_1, \llbracket p \rrbracket_\gamma), (A_2, \neg \llbracket p \rrbracket_\gamma)\} \\ \mathit{consume}(A, \gamma) : (\Delta, \Pi, \Gamma, P; q, \Sigma, \Xi) \rightsquigarrow \mathbb{S}\end{array}}{\mathit{consume}(p \mathbin{?} A_1 : A_2, \gamma) : (\Delta, \Pi, \Gamma, P, \Sigma, \Xi) \rightsquigarrow \mathbb{S}}$$

$$\dfrac{\begin{array}{c}\mathrm{fv}(\llbracket p \rrbracket_\gamma) \subseteq \Delta \qquad \mathit{consume}(A_1, \gamma) : (\Delta, \Pi; \llbracket p \rrbracket_\gamma, \Gamma, P, \Sigma, \Xi) \rightsquigarrow \mathbb{S}_1 \\ \mathit{consume}(A_2, \gamma) : (\Delta, \Pi; \neg \llbracket p \rrbracket_\gamma, \Gamma, P, \Sigma, \Xi) \rightsquigarrow \mathbb{S}_2\end{array}}{\mathit{consume}(p \mathbin{?} A_1 : A_2, \gamma) : (\Delta, \Pi, \Gamma, P, \Sigma, \Xi) \rightsquigarrow \mathbb{S}_1 \cup \mathbb{S}_2}$$

$$\dfrac{\Xi(i) = ((R, \bar{x}), s, G) \qquad s \neq \bot}{\begin{array}{c}\mathit{consume}(R(r, \bar{e}, a), \gamma) : (\Delta, \Pi, \Gamma, P, \Sigma, \Xi) \\ \rightsquigarrow \{(\Delta, \Pi, \Gamma, P; \llbracket r \rrbracket_\gamma = i; \llbracket r, \bar{e} \rrbracket_\gamma = \bar{x}; \llbracket a \rrbracket_\gamma = s, \Sigma, \Xi)\}\end{array}}$$

$$\dfrac{\Xi(i) = (t, s, H) \qquad \mathit{takeGuard}(\Delta, t, H, \llbracket G \rrbracket_\gamma, \Gamma', F, P')}{\begin{array}{c}\mathit{consume}(r@G, \gamma) : (\Delta, \Pi, \Gamma, P, \Sigma, \Xi) \\ \rightsquigarrow \{(\Delta, \Pi, \Gamma; \Gamma', P; P', \Sigma, \Xi[i \mapsto (t, s, F)])\}\end{array}}$$

Figure 2.8: Selected rules for the *consume* judgment.

is distinct from all other identifiers. Merging regions is governed by the *mergeRegion* judgment, which combines two region descriptors into one, producing a series of assumptions that are necessary for the merger to be well-defined. We elide the details here.

While producing a region or guard assertion can introduce a lot of non-determinism, it is typically the case that most of the non-deterministic choices will have inconsistent assumptions. For example, when producing an assertion that corresponds to an already known region, it would be inconsistent to merge that with a different region, or treat it as a completely new one. Since anything follows from inconsistent assumptions, we can immediately prune such cases.

The *consume* judgment. A selection of rules for the *consume* judgment are given in Figure 2.8. Unlike with *produce*, the syntax of the assertion may not uniquely determine which rule to apply. For instance, there are two rules for consuming a conditional assertion. The first rule consumes the conditional by consuming either the condition and the first assertion or the negated condition and the second assertion. (This are somewhat analogous to the $\vee$-introduction rules of natural deduction.) The second rule non-deterministically *assumes* the truth or falsity of the condition, consuming the first or second assertion in the respective case. This requires that only assumption variables can occur in the condition ($\mathrm{fv}(\llbracket p \rrbracket_\gamma) \subseteq \Delta$), since otherwise the context of assumptions would be ill-formed. Here, we are exploiting the law

$$\frac{\begin{array}{c}\Xi(r) = (\mathbf{T}(\bar{x}), s, G) \\ \Psi_{\mathrm{I}}(\mathbf{T}) = \{(P_i, e_i, A_i)\}_{i \in I} \qquad \forall i \in I.\, \mathit{freshSub}(\Delta, \Gamma_i, \mathrm{vars}(P_i, e_i, A_i), \gamma_i) \\ \forall i \in I.\, \mathit{produce}(A_i, \gamma_i) : (\Delta; \Gamma_i, \Pi; [\![\Psi_{\mathrm{P}}(\mathbf{T})]\!]_{\gamma_i} = \bar{x}; s = [\![e_i]\!]_{\gamma_i}; [\![P_i]\!]_{\gamma_i}, \Sigma, \Xi) \rightsquigarrow \mathbb{S}_i\end{array}}{\mathit{open}(r) : (\Delta, \Pi, \Sigma, \Xi) \rightsquigarrow \bigcup \{\mathbb{S}_i\}_{i \in I}}$$

$$\frac{\begin{array}{c}\Xi(r) = (\mathbf{T}(\bar{x}), s, G) \\ s' \notin \Delta \qquad \Gamma = (s' : \mathbf{Val}) \qquad P = ((s, s') \in \mathrm{Guar}_{\Psi}(\mathbf{T}(\bar{x}), G))\end{array}}{\mathit{update}(r, \Delta, \Xi, \Gamma, P, \Xi[r \mapsto (\mathbf{T}(\bar{x}), s', G)])}$$

$$\frac{\begin{array}{c}\Xi(r) = (\mathbf{T}(\bar{x}), s, G) \\ (P_0, e_0, A_0) \in \Psi_{\mathrm{I}}(\mathbf{T}) \qquad \mathit{freshSub}(\Delta; \Gamma, \Gamma_0, \mathrm{vars}(P_0, e_0, A_0), \gamma) \\ \mathit{consume}(A_0, \gamma) : (\Delta, \Pi, \Gamma; \Gamma_0, P; [\![\Psi_{\mathrm{P}}(\mathbf{T})]\!]_{\gamma} = \bar{x}; s = [\![e_0]\!]_{\gamma}; [\![P_0]\!]_{\gamma}, \Sigma, \Xi) \rightsquigarrow \mathbb{S}\end{array}}{\mathit{close}(r) : (\Delta, \Pi, \Gamma, P, \Sigma, \Xi) \rightsquigarrow \mathbb{S}}$$

$$\frac{\begin{array}{c}\mathit{openRegions} : (\Delta, \Pi, \Sigma, \Xi, \varepsilon) \rightsquigarrow \mathbb{S} \\ \forall (\Delta_1, \Pi_1, \Sigma_1, \Xi_1, \bar{r}) \in \mathbb{S}.\, \exists \mathbb{S}_1.\, \mathit{atomic}(\alpha) : (\Delta_1, \Pi_1, \Sigma_1, \Upsilon) \rightsquigarrow \mathbb{S}_1 \\ \forall (\Delta_2, \Pi_2, \Sigma_2, \Upsilon_2) \in \mathbb{S}_1.\, \exists \Gamma_2, P_2, \Xi_2.\, \mathit{updateRegions}(\bar{r}, \Delta_2, \Xi_1, \Gamma_2, P_2, \Xi_2) \\ \exists \bar{s}, \Gamma_2', P_2', \Xi_2'.\, \mathit{createRegions}(\bar{s}, \Delta_2; \Gamma_2, \Xi_2, \Gamma_2', P_2', \Xi_2') \\ \exists \mathbb{S}_2.\, \mathit{closeRegions}(\bar{s}, \bar{r}) : (\Delta_2, \Pi_2, \Gamma_2; \Gamma_2', P_2; P_2', \Sigma_2, \Xi_2') \rightsquigarrow \mathbb{S}_2 \\ \forall (\Delta_3, \Pi_3, \Gamma_3, P_3, \Sigma_3, \Xi_3) \in \mathbb{S}_3.\, (\Delta_3, \Pi_3 \vdash \Gamma_3, P_3) \wedge (\Delta_3; \Gamma_3, \Pi_3; P_3, \Sigma_3, \Xi_3, \Upsilon_2) \in \mathbb{S}_3\end{array}}{\vdash \langle \alpha \rangle : (\Delta, \Pi, \Sigma, \Xi, \Upsilon) \rightsquigarrow \mathbb{S}_3}$$

Figure 2.9: Symbolic execution rule for atomic statements.

of the excluded middle for pure assertion p: that is $p \vee \neg p$ holds. [1] Consuming a region assertion asserts that there is a corresponding region with the specified type and state. Consuming a guard assertion makes use of the judgment *takeGuard*$(\Delta, t, H, G, \Gamma, P, F)$, which expresses that guard G can be removed from guard H leaving the frame F, under conditions P, given the region type t. Frame inference is used to discharge *takeGuard* obligations.

The symbolic execution judgment. The symbolic execution judgment expresses how executing a program statement affects the symbolic state. Most of the symbolic execution rules are standard, except that when statements are sequenced together, the intermediate symbolic state is stabilized (using the *stabilise* judgment). The other novelty is the symbolic execution of atomic statements (read, write and CAS operations), which require access to the shared regions. The symbolic execution rule is given in Figure 2.9. It consists of six steps:

1. Regions are opened with the *openRegions* judgment. This is based on the *open* judgment, which opens a single region by producing

1 N.B. since our assertion language is limited, this does not require a classical meta-logic.

its interpretation (by case analysis on the possible interpretations).

2. The atomic statement is symbolically executed with the *atomic* judgment. This cannot affect the shared regions.

3. The regions are updated with the *updateRegions* judgment. This applies the *update* judgment to each of the regions, updating the state arbitrarily in a manner that is consistent with the guarantee for the available guard.

4. New regions may be created with the *createRegions* judgment. These regions must be distinct from the existing ones, and will be created along with the full guard for the region type. At this point, these new regions are open.

5. All of the open regions are closed with the *closeRegions* judgment. This applies *close* for each region, which consumes the interpretation.

6. The generated assertions are checked to follow from the assumptions, and the assertions are treated as assumptions in the new symbolic state.

2.4 Guard Reasoning

As we have seen, each region in Caper is associated with guards, which are ghost resources that are used to control how the region is accessed. In CAP [24], guards (or tokens) have the form $[\textsc{Guard}(\bar{x})]_{\pi}$. That is, a guard is a parametrised name with an associated (non-zero) fractional permission. The parametrised name determines some actions on the region, and any thread owning a non-zero fraction of the corresponding guard can perform those actions.

In TaDA [81], guards are generalized. Each region type is associated with a *guard algebra*, which is a partial commutative monoid (PCM). The PCM determines how the guards for a region can be composed. Actions can then be associated with arbitrary guards to enforce the desired protocol.

The benefit from this is that it is often easier to express protocols. For example, consider a concurrent set where either all threads are inserting elements or all threads are removing elements. In CAP, this protocol can be enforced by allowing the full permission for inserting, $[\textsc{Insert}]_1$, to be traded for the full permission for removing, $[\textsc{Remove}]_1$, (and vice-versa) by performing an action on the region. Requiring full permission (represented by 1) ensures that all threads must agree to the change between insert and remove modes; no thread can have an outstanding fraction when the change occurs. This approach was adopted in [82], and requires a significant amount of bookkeeping to manage the trades. By contrast, with an ad-hoc guard algebra we can choose to identify $[\textsc{Insert}]_1$ and $[\textsc{Remove}]_1$ (although not $[\textsc{Insert}]_\pi$ and $[\textsc{Remove}]_\pi$ for $\pi < 1$). This means that no actions are required to change mode, so there is no bookkeeping overhead.

2.4.1 *Automation Problems*

To automate reasoning about guards in Caper, we must be able to solve three problems: entailment with framing, composition and compatibility, and least upper-bounds. Entailment in Caper is intuitionistic: if $A \vdash B$ then $A * C \vdash B$. This means that it is always sound to discard resources. Because of this, we can view entailment as inducing an ordering on guards, where $A \vdash B$ is interpreted as "B is at most A".

The problem of entailment with framing is: given guards A and B, find a guard C, if it exists, such that $A \vdash B * C$. This problem has two key applications. The first is in determining if an action for a region is applicable in order to compute the Guarantee relation. For this, we do not need to know what the frame C is, but only if the entailment holds. The second is in computing entailments and frames for symbolic execution. If we currently have guard A as an available resource and we make a function call which requires guard B in its precondition, then a positive solution is necessary to make the call. Moreover, since the guard C is not required by the call, we can treat it as a frame. Note that for both of these applications it is sound to under-approximate the solution; that is, an implementation could say no frame exists, or find a non-maximal frame, and we would only be able to prove less as a consequence.

The problem of composition and compatibility is: given guards A and B, determine the composition $A * B$ if it is defined. The problem also has two key applications. The first is in determining if an action for a region is applicable in order to compute the Rely relation. For this we need to know if the guard associated with the action is compatible with the guard that we hold; if so, then another thread could have it and hence perform the action, so it should be accounted for in the Rely. The second is in combining resources that are obtained during symbolic execution. For instance, if we have a frame A and the postcondition of a call gives B then we would continue with the guard $A * B$. If the combination is not defined, then we can stop the symbolic execution at this point, since the state is unreachable. Note that it is sound to over-approximate compatibility — it is sound to include more actions in the Rely relation, and to continue symbolic execution from an unreachable state. It is sound to under-approximation composition — that is, return a guard that is smaller than the actual composition — since it is sound to discard resources.

The problem of least upper-bounds is: given guards A and B, find the least guard C such that $C \vdash A$ and $C \vdash B$. The purpose of computing least upper-bounds is to determine if the system of actions for a region is transitive. That is, if guard A permits the transition from x to y according to some action, and guard B permits the transition from y to z, then the least upper-bound C of A and B should permit the transition directly from x to z. Note that it is sound to under-approximate the least upper-bound, since it is not necessary to know that the system of actions is transitive, although we should only conclude that it is transitive when it truly is.

2.4.2 *Guard Implementation*

In Caper, guard algebras are constructed from a number of simple combinators. While this does not allow a fully general choice of guard algebra, the combinators are sufficiently flexible to express useful patterns, and will be extended to include further combinators in future (e.g. to support counting permissions [10]). The combinators also provide a concise way of expressing guard algebras.

In addition to the PCM structure, a guard algebra is equipped with a designated point representing the full resource. This full resource is provided when a region is created; moreover, it plays a significant role in some of the constructions.

Trivial guard algebra. The trivial guard algebra, `0` in Caper syntax, consists of a single element which is both the unit and the full resource. This algebra is used when a region does not have any roles associated with it: it can be used in the same way by all threads at all times.

All-or-nothing guard algebra. An all-or-nothing guard algebra consists of a single element distinct from the unit, which is the full resource. In Caper syntax, this algebra is represented by the name chosen for the point; for instance, `GUARD` would be the all-or-nothing algebra with point `GUARD`.

Permissions guard algebra. A permissions guard algebra `%GUARD` has the full resource `GUARD[1p]` which can be subdivided into smaller permissions `GUARD[`π`]`. The typical model for permissions is as fractions in the interval $[0, 1]$. This allows for (non-zero) permissions to be split arbitrarily often. Caper implements a different theory of permissions. This theory also allows arbitrary splitting, but also requires that `GUARD[`π`]` $*$ `GUARD[`π`]` is undefined for any non-zero π.

The theory can be encoded into the theory of *atomless Boolean algebras* — a Boolean algebra is said to be atomless if for all $a > \bot$ there exists a' with $a > a' > \bot$. The encoding defines the PCM operator as follows:

$$p_1 * p_2 = \begin{cases} p_1 \vee p_2 & \text{if } p_1 \wedge p_2 = \bot \\ \text{undefined} & \text{otherwise.} \end{cases}$$

A benefit of this encoding is that the first-order theory of atomless Boolean algebras is well known to be complete and therefore decidable (initially reported by Tarski [73], proved by Ershov [31], and see e.g. [15] for details).

Caper implements three different proof procedures for the theory of permissions. One uses the encoding with Boolean algebras and passes the problem to the first-order theorem prover E [69]. A second checks for the satisfiability of a first-order permissions formula directly. The third encodes the satisfiability problem with bit-vectors and passes it to the SMT solver Z3 [19].

Dockins et al. [28] previously proposed a tree-share model of permissions, which is also a model of this theory. Le et al. [50] have developed decision procedures for entailment checking based on this model, which could also be used by Caper.

Indexed guard algebra. An indexed guard algebra `#GUARD` consists of sets of individual guards `GUARD(`n`)` where n ranges over integers. A set of such individual guards is expressed using a set-builder-style notation: $\mathtt{GUARD}\{x \mid P\}$ describes the set of all guards $\mathtt{GUARD}(x)$ for which P holds of x. The full guard is $\mathtt{GUARD}\{x \mid \mathtt{true}\}$, and officially $\mathtt{GUARD}(n)$ is syntax for $\mathtt{GUARD}\{x \mid x = n\}$. The PCM operator on the indexed guard algebra can be defined as:

$$\mathtt{GUARD}S_1 * \mathtt{GUARD}S_2 = \begin{cases} \mathtt{GUARD}(S_1 \cup S_2) & \text{if } S_1 \cap S_2 = \emptyset \\ \text{undefined} & \text{otherwise} \end{cases}$$

The automation problems reduce to testing conditions concerning sets, specifically set inclusion. Sets in Caper are not first-class entities: they are always described by a logical predicate that is a (quantifier-free) arithmetic formula. For sets characterised in this way, set inclusion can be characterised as:

$$\{x \mid P\} \subseteq \{x \mid Q\} \iff \forall x.\, P \Rightarrow Q$$

Advanced SMT solvers, such as Z3 [19], have support for first-order quantification, and Caper exploits this facility to handle the verification conditions concerning set inclusion.

Product guard algebra construction. Given guard algebras M and N, the product construction $M * N$ consists of pairs $(m, n) \in M \times N$, where the PCM operation is defined pointwise: $(m_1, n_1) * (m_2, n_2) = (m_1 * m_2, n_1 * n_2)$. The unit is the pair of units and the full resource is the pair of full resources.

Sum guard algebra construction. Given guard algebras M and N, the sum construction $M + N$ is the discriminated (or disjoint) union of M and N, up to identifying the units and identifying the full resources of the two PCMs. The PCM operation embeds the operations of each of the constituent PCMs, with composition between elements of different PCMs undefined.

With these combinators, the guard algebra proposed for the set could be defined as `%INSERT + %REMOVE`.

Within the CAPER implementation, guards are represented as maps from guard names to parameters, which at present are either a unit (for all-or-nothing guards) or a permission expression (for permission guards). The unit is represented as the empty map. For this representation to work, CAPER disallows multiple guards with the same name in a guard algebra definition. For instance, `INSERT * %INSERT` is not legal. The sum construction is implemented by rewriting where necessary by the identities the construction introduces.

The Road not Taken.

An alternative approach to that taken in CAPER would be to specify guard algebras by defining their constituents together with equations for rewriting them. Where possible, these equations could be used to generate a confluent rewriting system (via the Knuth-Bendix completion algorithm [44]), which could be used for solving the automation problems.

This approach could allow more flexibility in defining guard algebras. However, it may be more difficult or even impossible to define some algebras, such as permissions, in this way. Moreover, it would allow the user to specify guard algebras that cannot be computed with effectively. This would make it more difficult for users to specify appropriate guard algebras. By contrast, the combinators are succinct and effective, so they are easier for a user to work with.

2.5 Interference Reasoning

There are two sides to interference: on one side, a thread should only perform actions that are anticipated by the environment; on the other, a thread must anticipate all actions that the environment could perform. In the logic of CAPER, the actions that a thread can perform and must anticipate are captured by the Guarantee and Rely relations respectively. Each time a thread updates a region, it must ensure that the update is permitted by the Guarantee with respect to the guards it owns (initially) for that region. Moreover, the symbolic state between operations and frames for non-atomic operations must be *stabilized* by closing the set of states they might be in under the Rely relation. CAPER must therefore be able to compute with these relations effectively.

The biggest obstacle to effective computation is that the relations are transitively closed. Transitivity is necessary, at least for Rely, since the environment may take arbitrarily many steps in between the commands of a thread. However, computing the transitive closure in general is a difficult problem. For instance, consider a region that has the following action

```
    0 : n ⇝ n + 1;
```

From this action, we should infer the relation $\{(n, m) \mid n \leq m\}$, as the reflexive-transitive closure of $\{(n, n+1) \mid n \in \mathbb{Z}\}$. It is generally beyond the ability of SMT solvers to compute transitive closures. El Ghazi et al. [30] have proposed an approach to encoding transitive closure problems so that they can be handled by SMT solvers; however, there are limits to what this can achieve.

Caper employs two techniques to deal with the transitive closure problem. This first is that, if the state space of the region is finite, then it is possible to compute the transitive closure directly. Caper uses a variant of the Floyd-Warshall algorithm [32] for computing shortest paths in a weighted graph. The 'weights' are constraints (first-order formulae) with conjunction as the 'addition' operation and disjunction as the 'minimum' operation.

The second technique is to check if the set of actions for a region are already transitive. In such a case, the relations will be transitive immediately, so no effort is required to compute the closure. In Caper, this is implemented by showing that for any pair of actions that can be sequenced there is a third action that permits this sequence directly. If the two sequenced actions have guards A and B respectively, then the guard for the third action should be at most the least guard that entails both A and B. This is why we need to solve the least upper-bound problem for guards.

For the action `0 : n ⇝ n + 1`, Caper can accurately determine the relations if it knows the state space is finite. (Currently, Caper only knows this if all the state interpretations are for constant values. In practice, we would not expect the direct computation approach to work well with large finite state spaces.) Otherwise, Caper over-approximates the Rely relation as the universal relation (which is essentially useless, but sound). To get the desired result in the infinite-state case, the user could instead specify the action

```
    n < m | 0 : n ⇝ m;
```

Caper can determine that if this action is sequenced with itself then the resulting transition (from n to m' with $n < m = n' < m'$) is permitted by the same action. Consequently, it knows that the actions are transitive and thus can compute the Rely and Guarantee relations exactly.

An approach (not yet implemented in Caper) that would allow more flexibility would be to automatically add a limited number of actions to a region in order to achieve transitivity. Essentially, if it is determined that the sequencing of two actions is not already covered by an existing action, that action can be added and the process repeated. The number of iterations should be bounded, since this will not terminate in general. As an example, consider the following actions

```
0 ≤ n, n < m | A : n ⇝ -m;
0 < n        | B : -n ⇝ n;
```

It should be possible to determine that the following actions can be obtained by sequencing these:

```
0 ≤ n, n < m | A ∗ B : n ⇝ m;
0 < n, n < m | A ∗ B : -n ⇝ -m;
0 < n, n < m | A ∗ B : -n ⇝ m;
```

After adding these, the set of actions is transitively closed.

The Road not Taken.

An alternative to computing the transitive closure of the relations would be to use their transitive interior. (The transitive interior of a relation is the greatest transitive relation included within the relation.) It is not immediately clear if this would be any simpler to automate, however, it would at the very least be unintuitive from a user's perspective. For instance, the transitive interior of $\{(n, n+1) \mid n \in \mathbb{Z}\}$ is the empty relation, so the action `0 : n ⇝ n + 1` would not actually permit any updates to the region.

Another alternative would be for the user to specify stable sets of abstract states for a region, instead of the actions for the region. For example, to specify that a thread holding the guard `A` can only see the state of the region increase, we might write:

```
A : { m | m ≥ n };
```

It may be easier to stabilize assertions in such a setting, since one need only widen the set of states for a region to one that is known to be stable. Conversely, determining which transitions are permitted by the Guarantee relation may be more difficult, since a transition is only permitted if it preserves all sets that another thread may consider to be stable. This approach in some ways resembles that of the Views Framework [25], where the stable assertions are determined a priori and updates are only permitted if they preserve all stable frames. This alternative approach to specifying regions would be compatible with Caper's existing approach, and may therefore be an interesting extension for future investigation.

2.6 Evaluation

We have successfully applied Caper to verify a number of concurrent algorithms. In §2.2, we discussed the spin lock and ticket lock, whose

Name	Code (lines)	Annotations (lines)	Time (s)
SpinLock	17 / 17	17 / 18	0.21 / 0.35
TicketLock(Client)	33 (41) / 24 (32)	19 (29) / 17 (27)	0.77 (0.82) / 2.22 (2.23)
ReadWriteLock	36 / 37	25 / 28	3.32 / 15.98
BoundedReadWriteLock	55 / 57	36 / 41	31.01 / 127.34
CASCounter	20 / 20	15 / 16	0.08 / 0.14
BoundedCounter	25 / 25	20 / 21	4.01 / 20.14
IncDec	29 / 29	19 / 21	0.10 / 0.36
ReferenceCount	31 / 30	22 / 24	0.22 / 0.73
ForkJoin(Client)	17 (32) / 17 (32)	16 (30) / 17 (31)	0.05 (0.07) / 0.07 (0.09)
Barrier(Client*)	71 (127) / 77 (130)	31 (60) / 35 (67)	28.22 (30.50) / 26.96 (31.44)
BagStack†	35 / 30	26 / 26	3.22 / 11.98
Queue‡	60 / 58	37 / 38	177.33 / 179.82

* flags: `-c 0` † flags: `-c 1 -o 3` ‡ flags: `-c 2 -o 3`

Table 2.1: Examples (recursive/iterative).

specifications guarantee mutual exclusion. We have verified a reader-writer lock, whose specification permits multiple readers or a single writer to enter their critical sections concurrently. This example uses counting permissions, but we have also verified a bounded version that does not. We have also verified a number of counter implementations with specifications that enforce monotonicity (CASCounter, BoundedCounter and IncDec), and an atomic reference counter (ReferenceCount). We have verified a library for joining on forked threads and a client that waits for the child thread to terminate before presenting the work done by the child. We have also verified a synchronization barrier and a client that uses it to synchronize threads incrementing and decrementing a counter. Finally, we have verified two implementations of a bag: the stack of §2.2.3, and a concurrent queue. We summarize these examples in Table 2.1, which shows the number of lines of code and annotation and verification times for recursive and iterative versions of each example. By default, Caper can create up to two regions at a time (`-c 2`) and open up to two regions (`-o 2`). The BagStack and Queue examples require opening up to three regions. The BarrierClient requires creating no regions, because of an issue with Caper's failure handling implementation.

From the verification times, we can observe that the versions that use loops tend to take longer than the recursive versions. This is due to case analysis which propagates through loops, but is abstracted in function calls. The high verification times for the bounded examples are largely due to Caper computing transitive closures for finite-state regions. The barrier example also takes significant time to compute the transitive closure of an infinite-state region. The BagStack and Queue use nested regions, and Queue has a complicated transition system, which combine to give a long verification time.

There are three key areas where Caper could use significant improvement. Firstly, proof search could be improved, for instance by directing the choice of regions to open and abstracting multiple branches into one. Currently, successful proofs may take some time and failing proofs take even longer. Secondly, Caper heuristics used in abduction require improvement, including loop invariants, this should allow more algorithms to be proved. Thirdly, Caper's annotations limit the expressivity of specifications to some extent. For instance, there is no support for regions with abstract states other than integers. Despite these limitations, we believe that Caper demonstrates the viability of our approach, and provides a good basis for further investigation.

2.7 Related Work

Caper is a tool for automating proofs in a concurrent separation logic with shared regions, aimed at proving functional correctness for fine-grained concurrent algorithms. The logic is in the spirit of concurrent abstract predicates (CAP) [24] taking inspiration from recent developments in concurrent separation logic such as iCAP [72], TaDA [81], Views [25], CaReSL [75], FCSL [58] and Iris [41].

Smallfoot [8] pioneered symbolic execution for separation logic. While it can prove functional correctness, it has limited support for concurrency and so cannot prove fine-grained concurrent algorithms.

SmallfootRG [13] extended Smallfoot to the more expressive logic RGSep [76]. The tool uses shared resources that are annotated with invariants and actions that can be performed over these resources. The actions that can be performed are not guarded, which leads to very weak specifications: it can prove memory safety, but not functional correctness. The abstraction of stabilization employed by SmallfootRG is different than the transitivity-based technique of Caper. SmallfootRG uses abstract interpretation to weaken assertions such that they are stable, where the abstract domain is based on symbolic assertions. Requiring (or ensuring) that a set of actions is transitively closed can be seen as an abstraction that terminates in a single step.

CAVE [77] built on SmallfootRG's action inference to prove linearisability [37] of concurrent data structures. That is, CAVE can prove that the operations of a concurrent data structure are atomic with respect to each other, and satisfy an abstract functional specification. Caper cannot yet prove linearisability, although it could in future support abstract atomicity in the style of TaDA [81]. On the other hand, CAVE cannot prove functional correctness of non-linearisable examples such as a spin lock.

Other mechanized — but not automatic — approaches based on separation logic include Verifast [39] and the Coq mechanization of fine-grained concurrent separation logic [58, 70]. Both approaches support an expressive assertion language, including higher-order predicates. They are able to prove functional correctness properties for fine-grained concurrent programs. Direct comparisons are, however, difficult. Programs and specifications need adaptation, often more than simple translation, resulting in different and sometimes weaker specifications. This is due in part to a smaller core set of operations and in part to a lack of features and expressivity of logic. However, when examples are comparable, the annotation overhead of the Caper examples is lower, often significantly. For example, the spin lock requires 87 lines of annotation in Verifast, compared to 18 in Caper, while the ticket lock requires 123 lines compared to 17. Verifast takes 0.11s to check each of these examples.

Viper [56] is a verification infrastructure for program verification based on permissions. It supports an expressive permission model that includes fractional permissions and symbolic permissions. It would be interesting to develop a front end for Viper that implements Caper's verification approach. A challenging issue is whether (and how) the non-deterministic proof search can be encoded in Viper's intermediate language.

2.8 Conclusions

We have presented Caper, the first automatic proof tool for a separation logic with CAP-style shared regions, and discussed the significant innovations that it involves. As a prototype, Caper provides a foundation for exploring the possibilities for automation with such a logic. Support for a number of different features will significantly increase the scope of examples that Caper can handle. We anticipate adding support for the following: additional guard algebra constructions; richer logical data types, such as sets and inductive data types; support for abstract and inductive predicates; and support for separation at the level of abstract states in the spirit of FCSL [58] and CoLoSL [66]. We would like to investigate inferring loop invariants and other annotations. We would like to make Caper more usable by providing proofs and failed proofs in a format that can easily be navigated and interpreted by a user. To this end, Caper already provides an interactive proof mode that allows a user to drive the proof search. This enables exploration of, in particular, failing proofs, which has proven valuable in development of the tool and the accompanying examples. A further goal is to put Caper on a rigorous footing by formalizing its logic in a proof assistant (such as Coq) and using Caper to generate program proofs that can be checked in the proof assistant or by a verified checker.

2.9 Postscript

Since the publication of the conference paper, work proceeded on the tool itself. As remarked in the conclusion above, we had several visions for the progress of Caper as a verification tool. By the nature of research and research groups, the project proceeded along a course shaped by the circumstances of the people involved.

2.9.1 *Fault-Tolerant Programming*

Caper as a software project is fairly well-developed. The architecture of the program and the sub-problems are fairly well delineated using the various abstraction and organizational techniques of Haskell, the implementation language. The code is accompanied by an extensive automated test suite, and it is easy to add more, the testing harness is built with auto-discoverability of tests in mind.

This extensibility proved fruitful for Jonathan Sutton, MEng, of Imperial College London, who undertook the project of extending Caper with facilities for reasoning about fault-tolerant code, leveraging the work by Ntzik, Rocha Pinto, and Gardner[60]. The work formed the substance of his MEng thesis, supervised by Prof Philippa Gardner with support from Julian Sutherland and myself.

Fault-tolerant reasoning is a departure from an underlying assumption in most program logics: that *computers* do not crash. Ntzik and collaborators enhanced a program logic in the Views family of concurrent separation logics[25] with notions of machine crash to model power failures, disc malfunction etc. Data-centers and highly available distributed systems are natural examples of systems designed around the possibility of hardware malfunction, and the main case study of the work by Ntzik[60] is indeed the ARIES recovery algorithm, a form of error recovery using write-ahead logging known from the world of data bases.

Concretely, the machine model is generalized to distinguish two notions of state: volatile and durable. Volatile state does not persist across machine crashes while the durable state does. This models the distinction between e.g. RAM and disk. Assertions are then generalized to describe a volatile and a durable component. Finally, triples are generalized to account for an invariant of the durable state at any given execution point, and a *recovery handler* C_R for a given statement C, a piece of code that ensures restoring the machine to a valid state upon restarting from a fault. The entire construction is summarized by and put to work in the *gault abstraction rule* from [60]:

$$\frac{S \vdash \left\{P_V \mid P_D\right\} \; \mathsf{C} \; \left\{Q_V \mid Q_D\right\} \qquad S \vdash \left\{\mathsf{true} \mid S\right\} \; \mathsf{C}_R \; \left\{\mathsf{true} \mid R\right\}}{R \vdash \left\{P_V \mid P_D\right\} \; [\mathsf{C}] \; \left\{Q_V \mid Q_D\right\}}$$

It is easy to see that it concretizes to the familiar Views-style concurrent separation logic that we also find in Caper by taking the durable state P_D, Q_D as true, the invariants as `true` and the recovery code C_R as `skip` as the function itself.

The key observation here that makes this work interesting in relation to Caper, is that there is nothing inherent in the choice of assertion language, or even programming language. The generalization to fault-tolerant code is a construction on the program logic, proven sound wrt to the Views-framework, of which the logic of Caper is an instance in spirit. Moreover, the proof-burden – the actual verification work – reduces to simple triples, *cf.* the above abstraction rule. The thesis of Sutton's work is that Caper *as a software platform for verification can be leveraged to prove specifications of fault-tolerant fine-grained concurrent code.*

Concretely, the work resulted in a branch of Caper with the modifications outlined above. The assertion language was enriched to speak of volatile and durable resources, and function-level specifications was given recovery-code annotations and durable resource invariants. The symbolic execution engine was extended to verify that every execution step preserved the durable resource invariants. Furthermore, the programming language of Caper was enriched with primitives for manipulating durable state in the form of a simple file-based logging system, with primitives like "append to log" and "read last log entry". Finally, an encoding of the ARIES write-ahead recovery algorithm was used to verify a fault-tolerant bank transfer scenario.

The missing ingredient is some preprocessing of specifications in order to have Caper perform the necessary proof burdens: as it stands, there is no tool-enforced connection between recovery code and the code it recovers. As such, a user needs to manually construct the proof-obligations by creating copies of the recovery function with every durable state invariant as a pre-condition.

Finally, a graceful up-stream merge of the work with a user-friendly support for opting into the functionality remains.

2.9.2 *Region-Levels*

TaDA, a logic for "Time and Data Abstraction"[81] is another Views-Style program logic, that, similarly to fault-tolerant concurrent separation logic, has verification burdens similar to those of Caper.

The key addition to the traditional program logic is the "atomic triple", which bakes in the notion of linearizability. Operationally, the atomic triple encodes the proof-obligation of identifying the linearization point of a procedure. The linearization point is the atomic operation after which the procedure being verified is said to logically take place. Linearizability is a well-studied correctness condition for abstraction preservation in concurrent implementations of data-structures[37].

From a symbolic execution perspective, the atomic triples of TaDA is a generalization of the atomic primitives of Caper. The proof system of Caper allows access to shared resources via these operations. In terms of spec, this means that a program can access resources held by regions precisely during the (symbolic) execution of an atomic primitive. A proof can open any number of regions at once, perform the operation, and close all regions again, providing proofs that the updates lie within the Guarantee relation implied by the specifications of these regions.

Opening the same region twice leads to logical inconsistencies, but this is avoided in Caper as any given symbolic state maintains the invariant that all region identifiers at hand are distinct, and, if they are not, merges the information in their associated region assertions to make it so. Regions can contain yet more regions, but these are not allowed to be opened by the same atomic operation, preventing cyclical references in regions letting Caper open the same region twice.

The situation, however, is more subtle in TaDA. The atomic triples allow for multiple openings of regions in several steps of execution, precisely enabling the problematic behaviours outlined above. To counter this, TaDA instruments symbolic states with *region levels*, a partial order on regions. Every region identifier is associated with an element of the order, and opening that region bounds the *open regions* by that element. Any *new* regions encountered through this action is added to the symbolic state at the next higher level. Finally, the proof system of TaDA ensures that only regions of higher levels than the open regions are opened. This prevents the logical inconsistencies resulting from opening regions twice.

This functionality was identified as the groundwork for introducing atomic triples to the specification language of Caper in the wake of the publication at ESOP. A branch of the Caper project exists with a sketch of extending the specification language with primitives for expressing constraints on levels. These are modeled as real numbers at the level of the underlying SMT constraint language, giving a partially ordered set with constants as desired. The symbolic execution engine adds the desired constraints at the appropriate places, namely when opening, creating and closing regions. This is a special case of the atomic triples of TaDA, where it spans but a single atomic operation.

What remains for this feature to be complete is an effort to figure out to what extent the tool can alleviate the burden of annotating all regions with levels. In many (if not all) instances, at least the collection of use-cases that the Caper project contains, the region structure is "well-behaved". In the spirit of automatic verification, it is apparent that the syntactic structure of the specification can hint at the appropriate level annotations for the proof. Or, perhaps, with the backtracking mechanisms of Caper at hand, that an appropriate annotation can be searched for: there are only so many possible orderings of for a finite set of regions.

2.9.3 *Proof generation*

Verification of a program is an absolute guarantee of correctness with respect to the formal framework in which the proof is made. It is, however, important to remember that ultimately these methods are tools for real people to solve problems. The human aspect of the process is important to consider when we move from considering program verification an academic discipline, and promote it to an aspect of software engineering.

Formal frameworks for verification, like the program logics on which Caper and many other tools are founded, leave room for human "errors". Wrong assumptions or misunderstandings on the part of the user can result in improper use e. g.assuming totality from a partial correctness proof, or race freedom from a simple type checker's acceptance of a concurrent program. Specification of programs is inherently a modeling exercise and can also leave room for "errors" as the requirements for a piece of code are interpreted and translated into the specification language of the verification framework. What might appear to be a correct specification might not capture subtleties of functional requirements e. g.does a sorting algorithm need to be stable, or does a a collection data structure need to be thread safe?

In turn, the user of the formal framework rely on its soundness and proper construction. For formal frameworks like program logics, this means logical consistency and proofs of soundness: everything proven correct is true, or, conversely, false things cannot be proven correct. Modern solutions to ensuring this can be found in Iris and the Views frameworks for constructing program logics.[25, 41]. They are in some sense schemas for forming verification frameworks, with sufficient conditions for ensuring correctness of the resulting logic. Moreover, the correctness of these constructions, given the assumptions of the correctness conditions, have been formalized in the proof assistant Coq for complete certainty.

What e. g.Views and Iris (and Coq, for that matter) achieve, is a small *trusted kernel*, a well-defined collection of assumptions or constructions that are well understood and proven correct. Then, rules for combining or using these primitives are proven sound, with the implication that anything built from these is a sound program logic.

Caper is a tool for essentially finding proofs of correctness within a program logic. Its search is *inspired* by the rules of the program logic, but there is no formal connection between the behavior of the tool and the formal program logic that it supposedly embodies. This means that Caper, as a component of the verification framework, must be *trusted*.

Caper is by many measures a complex piece of software, handling much bookkeeping in the symbolic execution engine: backtracking, scoping of variables, bindings with SMT solver libraries and plumbing of data between all these components. More than a few bugs discovered by the authors of the tool amounted to simple typos, omissions and copy/paste errors in seemingly innocuous *glue code*, that marshaled data between e.g. the symbolic state and the layer interacting with the underlying solver.

To reduce the amount of code that needs to be trusted, Caper was built from the start with *certificate generation* in mind. The symbolic execution engine produces a proof-tree as a certificate of acceptance of a specification. The certificate is in practice a formal statement of the correctness proof in Coq, drawing on a library containing a formalization of the Caper program logic, and a proof script that guides the Coq proof checker to successful completion. Where the Caper tool consulted SMT solvers, it can leave holes for the user of the certificate to complete e.g. tricky existential reasoning, if the standard arithmetic and order theoretic tactics in Coq cannot discharge the proof obligation.

A prototype of this system was developed by Thomas Dinsdale-Young, is in basic working order and available as a branch of the Caper repository[23].

3 Caper Tech Report

3.1 Introduction

This chapter contains a formal development of the program logic on which the Caper tool for automatic verification of fine-grained concurrent programs is built.

The chapter is structured as follows: Sections 3.2 and 3.2.3 gives an overview of how the concrete syntax used in Caper source files translate to the abstract syntax of the program logic used in the formal development. The abstract syntax and the operational semantics are detailed in Sections 3.3 and 3.4. The assertion logic to 3.6. The semantics of the program logic are developed in 3 stages. Section 3.7 details the semantic model that the interpretation of Section 3.8 translates the abstract syntax into. This translation is finally proven sound in Section 3.9.

The object language features a few things that are non-standard in the sense of a core, imperative calculus for developing program logics. Really, only the heap commands and basic control flow constructs are important.

But, as a tool for automating verification of programs, it is reasonable to push the object language closer to "the real thing": modern imperative programming languages and hence the algorithms designed for them make use of local variables and local returns to structure and organize code.

The way that ownership of capabilities is modeled is different from the literature. E.g. Iris is essentially a framework parameterized by choice of a partial commutative monoid that, for a given program or problem can be instantiated "large enough" to model the relationships between capability resources that is desired.

The model of Caper makes a different choice based on two ideas: by not having a parameterized framework of a model, but instead a "large enough" model once and for all, we achieve true modularity of verification. Morally, if two modules are proven correct using Iris, but on two different sets of assumptions, they are not directly composable. This is avoided in Caper.

In addition we have the issue of automation and implementability (and to a lesser extent usability). Products, sums, parameters and permissions give an expressive and convenient vocabulary for expressing many protocols on shared state.

Immediately, the combinators provided do not allow for higher order monoid expressions like Iris.

3.2 Syntax Summary

In this section we give a brief overview of how the concrete syntax of Caper corresponds to the abstract syntax of the program logic.

3.2.1 *Object Language*

The object language is a core imperative language with first-order dynamically allocated heap storage in addition to mutable local variables and multiple return points. The latter two features sets Caper apart from standard core imperative languages used in the literature, but is more representative of modern imperative languages.

$\mathsf{s} \in \mathsf{Stmt} ::=$	$\mathtt{s_1;s_2}$	sequencing
$\mid$	$\mathtt{if(e)\ then\ \{s_1\}\ else\ \{s_2\}}$	conditional branch
$\mid$	$\mathtt{while(e)\{s\}}$	zero-or-more iteration
$\mid$	$\mathtt{x := e}$	local assignment
$\mid$	$\mathtt{x := alloc(e)}$	allocation
$\mid$	$\mathtt{x := CAS(e_1,e_2,e_3)}$	compare-and-swap
$\mid$	$\mathtt{[e_1] := e_2}$	heap write
$\mid$	$\mathtt{x := [e]}$	heap read
$\mid$	$\mathtt{x := f(\vec{e})}$	function call
$\mid$	$\mathtt{return\ e}$	value return
$\mid$	$\mathtt{fork\ f(\vec{e})}$	spawn thread
$\mid$	$\mathtt{skip}$	no-op

$$\mathsf{FunDef} ::= \mathtt{function\ f(\vec{x})\{s\}}$$

$$\mathsf{Program} ::= \mathsf{FunDef}^*\mathtt{;function\ main()\{s\}}$$

3.2.2 *Specification Language*

The assertion language is a standard, first-order intuitionistic separation logic extended with resources representing knowledge and ownership of shared state and permissions to update that state. Shared state is modeled by "regions", and permissions to manipulate the state is expressed through "guards". A region declaration consists of a guard algebra declaration (GAD), a region interpretation declaration (ID) and an action declaration (AD). The guard algebra describes which permissions can be owned for a particular region, and how those permissions may be shared, split and combined. The region interpretation describes by way of a "state variable" the configuration of the heap owned by the region. The action interpretation describes the protocol governing the shared state by assigning legal updates to the state variable to guards.

The P in the grammar ranges over assertions in first-order separation logic extended with guard and region assertions as detailed later in this appendix.

$$
\begin{array}{lcll}
\mathit{GAD}\ g & ::= & G & \text{Indivisible Guard} \\
& \mid & \%G & \text{Divisible Guard} \\
& \mid & \#G & \text{Parameterized Guard} \\
& \mid & g * g & \text{Product Construction} \\
& \mid & g + g & \text{Sum Construction} \\
& \mid & 0 & \text{Nil Guard}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{ID}\ i & ::= & \cdot \\
& \mid & i, (\Delta).\Pi \mid e : P
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{AD}\ a & ::= & \cdot \\
& \mid & a, (\Delta).\Pi \mid G : e_1 \rightsquigarrow e_2
\end{array}
$$

Well-specified programs and individual functions are specified with respect to a region declaration context:

$$\mathit{RegionContext}\ R \quad ::= \quad \bullet \mid R, \mathbb{T}(r, \vec{x})(g, i, a)$$

The variable r binds the name of the region itself (think `this` from mainstream OO programming languages) and the variables $\vec{x}$ bind the names of the resources held by the region. These names are bound in the region interpretation and action declaration.

Well-specified programs are also specified with respect to a function specification context, assigning pre- and postconditions to function names, expressed as assertions ranging over the parameters of the function and its return value.

$$\mathit{SpecCtxt}\ \Phi \quad ::= \quad \bullet \mid \Phi,\ \mathtt{f} : (\Gamma, \vec{\mathtt{x}})\{P\}\{r.Q\}$$

Individual functions and fragments of code are verified "Hoare style", demonstrating that programs run from states satisfying a precondition P either run to completion, reaching a state satisfying Q, or return locally a value satisfying U:

$$R; \Phi; \Gamma \vdash \left\{P\right\}\ \mathtt{s}\ \left\{Q \mid r.U\right\}$$

There is no surface syntax for this judgment as constructing them is precisely the function of Caper.

3.2.3 *Spin Lock Example*

In order to demonstrate the specification syntax of Caper we here describe a fully verified program, relating it to the surface syntax in the process. The following listing demonstrates a simple spin-lock, for simplicity, but it will exercise most all the features of the program logic.

First, the surface syntax of the region declarations for the spin-lock:

```
region SLock(r,x) {
  guards %LOCK * UNLOCK;
  interpretation {
    0 : x ↦ 0 &*& r@UNLOCK;
    1 : x ↦ 1;
  }
  actions {
    LOCK[_] : 0 ⇝ 1;
    UNLOCK : 1 ⇝ 0;
  }
}
```

The set of primitive guards is $\{\texttt{LOCK}, \texttt{UNLOCK}\}$, and the guard algebra declaration, call it G_{SL}, $\%\texttt{LOCK} * \texttt{UNLOCK}$, describing that the LOCK resource is divisible, while the UNLOCK resource is not.

The region interpretation, I_{SL}, is described by two clauses, in abstract syntax written as

$$(\cdot).\top \mid 0 : \mathtt{x} \mapsto 0 * r@\texttt{UNLOCK},$$
$$(\cdot).\top \mid 1 : \mathtt{x} \mapsto 1$$

Hence, the region declaration context for this example consists of the single SLock declaration:

$$R := \texttt{SLock}(r, x)(G_{SL}, I_{SL}, A_{SL})$$

The concrete syntax for the unlock implementation is as follows:

```
function unlock(x)
  requires SLock(r,x,1) &*& r@UNLOCK;
  ensures SLock(r,x,_); {
    [x] := 0;
}
```

This represents the function specification written as follows:

$$\texttt{unlock} :$$
$$(r : \mathsf{Region}, \mathtt{x} : \mathsf{Val})\{\texttt{SLock}(r, x, 1) * r@\texttt{UNLOCK}\}\{ret.\exists s.\texttt{SLock}(r, x, s)\}$$

3.3 Syntax of Object Language

Convention 1 (Sets)**.** Sets are typeset with capitalized italicized font when there is no canonical name, e.g. *Type*, *Heap* but $\mathbb{N}$ and $\mathbb{Z}$.

Convention 2 (Sets of Syntax)**.** We typeset sets of syntax with capitalized teletype, e.g. Fun.

Definition 3 (Var, Variable Names)**.** We suppose a set Var ranged over by $\mathtt{x}, \mathtt{y}$ etc.

Definition 4 (Fun, Function Names)**.** We suppose a set Fun, ranged over by $\mathtt{f}, \mathtt{g}$ etc.

Definition 5 (Expr, Syntactic Expressions)**.** We define Expr as arithmetic and boolean expressions over integer literals and variables drawn from Var. We let e range over syntactic expressions.

$$\begin{array}{rcll}
\mathsf{e} \in \mathsf{Expr} & ::= & \mathtt{x} & \text{variables} \\
& \mid & n & \text{constants} \\
& \mid & \mathsf{e} + \mathsf{e} & \text{arithmetic operations} \\
& \mid & \mathsf{e} - \mathsf{e} & \\
& \mid & \mathsf{e} * \mathsf{e} & \\
& \mid & \mathsf{e} / \mathsf{e} & \\
& \mid & \mathsf{e} = \mathsf{e} & \text{integer comparisons} \\
& \mid & \mathsf{e} \neq \mathsf{e} & \\
& \mid & \mathsf{e} < \mathsf{e} & \\
& \mid & \mathsf{e} \leq \mathsf{e} & \\
& \mid & \mathsf{e} \geq \mathsf{e} & \\
& \mid & \mathsf{e} > \mathsf{e} &
\end{array}$$

Definition 6 (FunDef, Syntactic Function Definitions)**.** We define FunDef by the following grammar:

$$\mathsf{FunDef} ::= \mathtt{function}\ \mathtt{f}(\vec{\mathtt{x}})\{\mathtt{s}\}$$

Definition 7 (Program, Caper Programs)**.** The set of syntactic Caper Programs is defined as the set of sequences of function definitions with a designated `main` function as the entry point, i.e.

$$\mathsf{Program} := \mathsf{FunDef}^{*}; \mathtt{function}\ \mathtt{main}()\{\mathtt{s}\}$$

3.4 Operational Semantics

Definition 8 (*Addr*, Machine Addresses)**.**

$$n \in \mathit{Addr} := \mathbb{N}$$

Definition 9 (*Val*, Program Values)**.**

$$n \in \mathit{Val} := \mathbb{Z}$$

Note that $\mathit{Addr} \subseteq \mathit{Val}$. We can distinguish valid addresses in our meta-mathematics simply by considering $n \in \mathit{Addr}$ versus $n \notin \mathit{Addr}$.

Definition 10 (*Heap*, Global Machine Memory)**.**

$$h \in \mathit{Heap} := \mathit{Addr} \overset{\text{fin}}{\rightharpoonup} \mathit{Val}$$

Heaps form a partial commutative monoid with the empty heap, written $\emptyset$ as unit and disjoint union, written $\uplus$, as composition, defined as follows:

$$h_1 \uplus h_2 = \begin{cases} \bot & \mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) \neq \emptyset \\ \mathrm{lookup}(h_1, h_2) & \text{otherwise} \end{cases}$$

$$\mathrm{lookup}(h_1, h_2) = \lambda a. \begin{cases} h_1(a) & a \in \mathrm{dom}(h_1) \\ h_2(a) & a \in \mathrm{dom}(h_2) \\ \bot & \text{otherwise} \end{cases}$$

As always, a partial commutative monoid is partially ordered by the canonical extension order:

$$h_1 \sqsubseteq h_2 \overset{\text{def}}{\iff} \exists h_3.\ h_1 \uplus h_3 = h_2$$

Observe that $\sqsubseteq$ is both

1. *reflexive*, as $\emptyset \uplus h = h \uplus \emptyset = h$ for all h;
2. *transitive*, as if we have heaps to extend h_1 to h_2 and h_2 to h_3, the composition of the two pieces will extend h_1 to h_3, since $\uplus$ is associative;
3. *anti-symmetric*, as having heaps to extend h_1 to h_2 and vice versa implies that the pieces are empty, hence $h_1 = h_2$. That is, if $h_1 \cdot h_1' = h_2$ and $h_2 \cdot h_2' = h_1$ it means that $(h_2 \cdot h_2') \cdot h_1' = h_2$. If $h_2 \cdot (h_2' \cdot h_1') = h_2$ then it must be that $h_2' \cdot h_1' = \emptyset$, meaning so must both of h_2' and h_1'.

Definition 11 (*Stack*, Thread Local Stack)**.**

$$\sigma \in \mathit{Stack} := \mathsf{Var} \rightarrow \mathit{Val}$$

We silently lift the action of stacks on variables to sequences of variables when the intention is clear, e.g. $\sigma(\vec{\mathtt{x}}) = \sigma(\mathtt{x}_1), \sigma(\mathtt{x}_2)..., \sigma(\mathtt{x}_n)$ for a sequence $\vec{x}$ of length n.

Definition 12 (*Cont*, Continuations)**.** We define the set of continuations by the following grammar, where $\sigma \in \mathit{Stack}$, $\mathtt{s} \in \mathsf{Stmt}$ and $\mathtt{x} \in \mathsf{Var}$:

$$\kappa \in \mathit{Cont} ::= \mathtt{s} \mid \mathtt{x}{:=}(\sigma, \kappa)$$

We use the second construction to represent the frames of the call stack.

Definition 13 (*ThreadId*, Thread Identifiers)**.** We suppose a set *ThreadId* of thread identifiers, ranged over by *id*.

Definition 14 (*Thread*, Threads)**.**

$$t \in \mathit{Thread} := \mathit{Stack} \times \mathit{Cont}$$

Definition 15 (*ThreadPool*, Thread pools)**.**

$$T \in \mathit{ThreadPool} := \mathit{ThreadId} \overset{\text{fin}}{\rightharpoonup} \mathit{Thread}$$

Definition 16 (*Program*, Machine Configurations)**.** We enrich machine configurations with a distinguished faulting state denoted $\lightning$:

$$\mathit{Program} := (\mathit{Heap} \uplus \{\lightning\}) \times \mathit{ThreadPool}$$

Definition 17 (*Env*, Function Environments)**.** Function environments are partial mappings from function names to parameters and bodies. The set of function environments are generated by the following grammar, where $\mathtt{f} \in \mathsf{Fun}, \vec{\mathtt{x}} \in \mathsf{Var}^*$ and $\mathtt{s} \in \mathsf{Stmt}$:

$$E \in \mathit{Env} ::= \cdot \mid E, \mathtt{f} : (\vec{\mathtt{x}}, \mathtt{s})$$

where f does not appear in the E it extends. An environment E induces a partial map

$$\mathsf{Fun} \overset{\text{fin}}{\rightharpoonup} (\mathsf{Var}^*, \mathsf{Stmt})$$

and we denote a well-defined lookup as $E(\mathtt{f}) = (\vec{\mathtt{x}}, \mathtt{s})$.

Definition 18 ($[\![-]\!]_{\_}$, Expression Evaluation)**.** We define a *total* interpretation of expressions with regards to a stack as follows:

$$[\![-]\!]_{\_} : \mathsf{Expr} \times \mathit{Stack} \to \mathit{Val}$$

$$\begin{aligned}
[\![\mathtt{x}]\!]_\sigma &= \sigma(\mathtt{x}) \\
[\![\mathsf{e}_1 + \mathsf{e}_2]\!]_\sigma &= [\![\mathsf{e}_1]\!]_\sigma + [\![\mathsf{e}_2]\!]_\sigma \\
[\![\mathsf{e}_1 / \mathsf{e}_2]\!]_\sigma &= \begin{cases} [\![\mathsf{e}_1]\!]_\sigma \,/\, [\![\mathsf{e}_2]\!]_\sigma & \text{if } [\![\mathsf{e}_2]\!]_\sigma \neq 0 \\ 0 & \text{otherwise} \end{cases} \\
[\![\mathsf{e}_1 \leq \mathsf{e}_2]\!]_\sigma &= \begin{cases} \\ 1 & \text{if } [\![\mathsf{e}_1]\!]_\sigma \leq [\![\mathsf{e}_2]\!]_\sigma \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

We omit the remaining, straightforward cases.

Notice that uninitialized variables are not handled exceptionally as the stack is a *total* function. Intuitively, referencing uninitialized local variables returns an arbitrary but consistent value - multiple accesses of the same variables returns the same value.

We present the operational semantics as a Views-style labeled transition system, with a label interpretation providing the semantics of atomic actions: heap operations and faulting.

Definition 19 (*Action*, Atomic Action Labels)**.** The set of atomic action labels is described by the following grammar, where n ranges over *Addr* and v ranges over *Val*.

$$\begin{array}{rl} \alpha \in \mathit{Action} ::= & \mathtt{id} \\ | & \lightning \\ | & \mathtt{alloc}(n, v) \\ | & \mathtt{read}(n, v) \\ | & \mathtt{write}(n, v) \\ | & \mathtt{CAS}(n, v, v, v) \end{array}$$

Convention 20 ($- \mapsto -$, Map Extension)**.** We denote the extension of maps by the usual $m[k \mapsto v]$ notation, and lift it to sequences of matching lengths, using $m[\vec{k} \mapsto \vec{v}]$ to stand for $m[k_1 \mapsto v_1]...[k_n \mapsto v_n]$, with circumstances ensuring matching lengths.

Definition 21 ($[\![-]\!]$, Atomic Action Interpretation)**.** We interpret atomic action labels as functions from atomic action labels to heaps or a designated faulting element $\lightning$.

$$[\![-]\!] : \mathit{Action} \to \mathit{Heap} \to \mathcal{P}(\mathit{Heap} \uplus \{\lightning\})$$

$$\begin{aligned} [\![\mathtt{id}]\!]\,(h) &= \{h\} \\ [\![\lightning]\!]\,(h) &= \{\lightning\} \end{aligned}$$

$$[\![\mathtt{read}(x, v)]\!]\,(h) = \begin{cases} \{h\} & \text{if } x \in \mathrm{dom}(h) \text{ and } h(x) = v \\ \emptyset & \text{if } x \in \mathrm{dom}(h) \text{ and } h(x) \neq v \\ \{\lightning\} & \text{if } x \notin \mathrm{dom}(h) \end{cases}$$

$$[\![\mathtt{write}(x, v)]\!]\,(h) = \begin{cases} \{h[x \mapsto v]\} & \text{if } x \in \mathrm{dom}(h) \\ \{\lightning\} & \text{otherwise} \end{cases}$$

$$[\![\mathtt{CAS}(x, v, v', b)]\!]\,(h) = \begin{cases} \{h[x \mapsto v']\} & \text{if } x \in \mathrm{dom}(h),\, h(x) = v \text{ and } b \neq 0 \\ \{h\} & \text{if } x \in \mathrm{dom}(h),\, h(x) \neq v \text{ and } b = 0 \\ \emptyset & \text{if } x \in \mathrm{dom}(h) \text{ and } h(x) = v \text{ but } b = 0, \text{ or} \\ & \qquad\qquad h(x) \neq v \text{ and } b \neq 0 \\ \{\lightning\} & \text{if } x \notin \mathrm{dom}(h) \end{cases}$$

$$[\![\mathtt{alloc}(v, n)]\!]\,(h) = \begin{cases} \{h[n \mapsto \_]...[n + (v-1) \mapsto \_]\} & \text{if } n, ..., n + (v-1) \notin \mathrm{dom}(h) \\ & \quad \text{and } v > 0 \\ \emptyset & \text{if } n \text{ or } n+1 \text{ or ... or } n + (v-1) \\ & \quad \in \mathrm{dom}(h) \text{ and } v > 0 \\ \{\lightning\} & \text{if } v < 1 \end{cases}$$

We lift a function

$$[\![\alpha]\!] : \mathit{Heap} \to \mathcal{P}(\mathit{Heap} \uplus \{\lightning\})$$

to

$$(\mathit{Heap} \uplus \{\lightning\}) \to \mathcal{P}(\mathit{Heap} \uplus \{\lightning\})$$

by letting $[\![\alpha]\!]\,(\lightning) = \{\lightning\}$.

Definition 22 ($- \vdash - \xrightarrow{-} -$, Thread Semantics)**.** We define the semantics of individual threads as a labeled transition relation, with the set of labels consisting of atomic action labels extended by a designated $\texttt{fork}(\texttt{f},\vec{v})$ label; letting

$$\{\mathit{Label}\} := \mathit{Action} + \{\texttt{fork}(\texttt{f},\vec{v}) \mid \texttt{f} \in \mathsf{Fun}, \vec{v} \in \mathit{Val}^*\}$$

we define the relation

$$- \vdash - \xrightarrow{-} - \subset \mathit{Env} \times \mathit{Thread} \times \{\mathit{Label}\}\ \mathit{timesThread}$$

Seq
$$\frac{E \vdash (\sigma, \mathtt{s}_1) \xrightarrow{\alpha} (\sigma', \mathtt{s}_1')}{E \vdash (\sigma, \mathtt{s}_1;\mathtt{s}_2) \xrightarrow{\alpha} (\sigma', \mathtt{s}_1';\mathtt{s}_2)}$$

Skip
$$\frac{}{E \vdash (\sigma, \mathtt{skip;s}) \xrightarrow{\mathtt{id}} (\sigma, \mathtt{s})}$$

IfTrue
$$\frac{[\![\mathtt{e}]\!]_\sigma \neq 0}{E \vdash (\sigma, \mathtt{if}(\mathtt{e})\ \mathtt{then}\ \{\mathtt{s}_1\}\ \mathtt{else}\ \{\mathtt{s}_2\}) \xrightarrow{\mathtt{id}} (\sigma, \mathtt{s}_1)}$$

IfFalse
$$\frac{[\![\mathtt{e}]\!]_\sigma = 0}{E \vdash (\sigma, \mathtt{if}(\mathtt{e})\ \mathtt{then}\ \{\mathtt{s}_1\}\ \mathtt{else}\ \{\mathtt{s}_2\}) \xrightarrow{\mathtt{id}} (\sigma, \mathtt{s}_2)}$$

WhileTrue
$$\frac{[\![\mathtt{e}]\!]_\sigma \neq 0}{E \vdash (\sigma, \mathtt{while}(\mathtt{e})\{\mathtt{s}\}) \xrightarrow{\mathtt{id}} (\sigma, \mathtt{s};\mathtt{while}(e)\{s\})}$$

WhileFalse
$$\frac{[\![\mathtt{e}]\!]_\sigma = 0}{E \vdash (\sigma, \mathtt{while}(\mathtt{e})\{\mathtt{s}\}) \xrightarrow{\mathtt{id}} (\sigma, \mathtt{skip})}$$

FunctionCall
$$\frac{E(\mathtt{f}) = (\vec{\mathtt{x}}, \mathtt{s}) \qquad \sigma'(\vec{\mathtt{x}}) = [\![\vec{\mathtt{e}}]\!]_\sigma}{E \vdash (\sigma, \mathtt{x}{:=}\mathtt{f}(\vec{\mathtt{e}})) \xrightarrow{\mathtt{id}} (\sigma, \mathtt{x}{:=}(\sigma', \mathtt{s}))}$$

FunctionCallStep
$$\frac{E \vdash \kappa \xrightarrow{\alpha} \kappa'}{E \vdash (\sigma, \mathtt{x}{:=}\kappa) \xrightarrow{\alpha} (\sigma, \mathtt{x}{:=}\kappa')}$$

LocalReturn
$$\frac{}{E \vdash (\sigma, \mathtt{x}{:=}(\sigma', \mathtt{return\ e;s})) \xrightarrow{\mathtt{id}} (\sigma[\mathtt{x} \mapsto [\![\mathtt{e}]\!]_{\sigma'}], \mathtt{skip})}$$

Return
$$\frac{}{E \vdash (\sigma, \mathtt{x}{:=}(\sigma', \mathtt{return\ e})) \xrightarrow{\mathtt{id}} (\sigma[\mathtt{x} \mapsto [\![\mathtt{e}]\!]_{\sigma'}], \mathtt{skip})}$$

DefaultReturn
$$\frac{}{E \vdash (\sigma, \mathtt{x}{:=}(\sigma', \mathtt{skip})) \xrightarrow{\mathtt{id}} (\sigma[\mathtt{x} \mapsto v], \mathtt{skip})}$$

$$\textsc{LocalAssign}\quad \frac{}{E \vdash (\sigma, \mathtt{x} := \mathtt{e}) \xrightarrow{\mathtt{id}} (\sigma[\mathtt{x} \mapsto [\![e]\!]_\sigma], \mathtt{skip})}$$

$$\textsc{Fork}\quad \frac{}{E \vdash (\sigma, \mathtt{fork}\ \mathtt{f}(\vec{\mathtt{e}})) \xrightarrow{\mathtt{fork}(\mathtt{f},[\![\vec{\mathtt{e}}]\!]_\sigma)} (\sigma, \mathtt{skip})}$$

$$\textsc{AtomicWrite}\quad \frac{[\![\mathtt{e}_1]\!]_\sigma = n \qquad n \in \mathit{Addr}}{E \vdash (\sigma, [\mathtt{e}_1] := \mathtt{e}_2) \xrightarrow{\mathtt{write}(n,[\![\mathtt{e}_2]\!]_\sigma)} (\sigma, \mathtt{skip})}$$

$$\textsc{AtomicWriteFault}\quad \frac{[\![\mathtt{e}_1]\!]_\sigma \notin \mathit{Addr}}{E \vdash (\sigma, [\mathtt{e}_1] := \mathtt{e}_2) \xrightarrow{\lightning} (\sigma, \mathtt{skip})}$$

$$\textsc{AtomicRead}\quad \frac{[\![\mathtt{e}]\!]_\sigma = n \qquad n \in \mathit{Addr}}{E \vdash (\sigma, \mathtt{x} := [\mathtt{e}]) \xrightarrow{\mathtt{read}(n,v)} (\sigma[\mathtt{x} \mapsto v], \mathtt{skip})}$$

$$\textsc{AtomicReadFault}\quad \frac{[\![\mathtt{e}]\!]_\sigma \notin \mathit{Addr}}{E \vdash (\sigma, [\mathtt{x}] := \mathtt{e}) \xrightarrow{\lightning} (\sigma, \mathtt{skip})}$$

$$\textsc{CAS}\quad \frac{[\![\mathtt{e}_1]\!]_\sigma = n \qquad n \in \mathit{Addr}}{E \vdash (\sigma, \mathtt{x} := \mathtt{CAS}(\mathtt{e}_1,\mathtt{e}_2,\mathtt{e}_3)) \xrightarrow{\mathtt{CAS}(n,[\![\mathtt{e}_2]\!]_\sigma,[\![\mathtt{e}_3]\!]_\sigma,b)} (\sigma[\mathtt{x} \mapsto b], \mathtt{skip})}$$

$$\textsc{CASFault}\quad \frac{[\![\mathtt{e}_1]\!]_\sigma \notin \mathit{Addr}}{E \vdash (\sigma, \mathtt{x} := \mathtt{CAS}(\mathtt{e}_1,\mathtt{e}_2,\mathtt{e}_3)) \xrightarrow{\lightning} (\sigma, \mathtt{skip})}$$

$$\textsc{Alloc}\quad \frac{[\![\mathtt{e}]\!]_\sigma = n \qquad 1 \leq n}{E \vdash (\sigma, \mathtt{x} := \mathtt{alloc}(\mathtt{e})) \xrightarrow{\mathtt{alloc}(n,v)} (\sigma[\mathtt{x} \mapsto v], \mathtt{skip})}$$

$$\textsc{AllocFault}\quad \frac{[\![\mathtt{e}]\!]_\sigma < 1}{E \vdash (\sigma, \mathtt{x} := \mathtt{alloc}(\mathtt{e})) \xrightarrow{\lightning} (\sigma, \mathtt{skip})}$$

Lemma 23 (Determinism of Thread Semantics)**.** For any thread $(\sigma, \mathtt{s})$, if $E \vdash (\sigma, \mathtt{s}) \xrightarrow{\alpha} (\sigma_1, \mathtt{s}_1)$ and $E \vdash (\sigma, \mathtt{s}) \xrightarrow{\alpha'} (\sigma_2, \mathtt{s}_2)$ then $s_1 = s_2$.

Definition 24 ($- \vdash - \xrightarrow{-} -$, Threadpool Semantics)**.** We define the operational semantics of a thread pool as a atomic action labeled transition relation.

$$- \vdash - \xrightarrow{-} - \subseteq \mathit{Env} \times \mathit{ThreadPool} \times \mathit{Action} \times \mathit{ThreadPool}$$

$$\frac{E \vdash t \xrightarrow{\texttt{fork}(\texttt{f},\vec{v})} t' \qquad E(\texttt{f}) = (\vec{\texttt{x}}, \texttt{s}) \qquad \sigma(\vec{\texttt{x}}) = \vec{v}}{E \vdash T \mid\mid t \xrightarrow{\texttt{id}} T \mid\mid t' \mid\mid (\sigma, \texttt{s})}$$

$$\frac{E \vdash t \xrightarrow{\alpha} t' \qquad \alpha \in \mathit{Action}}{E \vdash T \mid\mid t \xrightarrow{\alpha} T \mid\mid t'}$$

Definition 25 ($- \vdash - \xrightarrow{-} -$, Program Semantics)**.** We define the operational semantics of programs as a single step transition relation

$$- \vdash - \to - \subseteq \mathit{Env} \times \mathit{Program} \times \mathit{Program}$$

$$\frac{E \vdash T \xrightarrow{\alpha} T' \qquad h' \in [\![\alpha]\!]\,(h)}{E \vdash (h, T) \to (h', T')}$$

3.5 Assertion Logic

Definition 26 (Type, Logical Types)**.** We define a set of primitive types over which we will allow quantification:

$$\mathsf{Type} \quad \tau ::= \mathsf{Val} \mid \mathsf{Perm} \mid \mathsf{Region}$$

Definition 27 (*LVar*, Logical Variables)**.** We suppose a set *LVar* ranged over by x, y etc.

Definition 28 (Context, Logical Contexts)**.** Well-formed logical contexts binds logical variables to primitive types:

$$\Gamma ::= \cdot \mid \Gamma, x : \tau$$

where x does not occur in the Γ it extends. A context Γ induces a partial map from variables to types, and we denote a defined look-up by $\Gamma(x) = \tau$.

Definition 29 (*RegionTypeName*, Region Type Names)**.** We suppose a set of region type names ranged over by $\mathbb{T}$.

Definition 30 (*RegionType*, Region Types)**.** We define the set of region types as parameterized region type names, according to the following definition:

$$\mathit{RegionType} := \{\mathbb{T}(r, \vec{x} : \vec{\tau}) \mid r \in \mathit{LVar}, \vec{x} \in \mathit{LVar}^*, \vec{\tau} \in \mathsf{Type}^*\}$$

Definition 31 (*PrimitiveGuard*, Primitive Guard Symbols)**.** We suppose an infinite set of primitive guard symbols, usually ranged over by G.

Definition 32 (Term, Assertion Logic Terms)**.** We define the syntax of assertions with the following grammar. The types τ described in the grammar are simple types over which we allow quantification; this is not a higher-order logic.

$M, N, O \in \mathsf{Term} ::=$	x	logical variables
	$\mid \top \mid \bot \mid M \wedge N \mid M \vee N$	propositional formulae
	$\mid M \Rightarrow N \mid \neg N$	
	$\mid M \mathbin{?} N : O$	conditionals
	$\mid \forall x : \tau.M \mid \exists x : \tau.M$	quantification
	$\mid M =_\tau N$	primitive equality
	$\mid M * N \mid \texttt{emp}$	separating conjunction
	$\mid M \mapsto N \mid M \mapsto [N]$	points-to assertions
	$\mid M@(N) \mid \mathbb{T}(M, N, O)$	region assertions
	$\mid G \mid G[M] \mid G(M)$	guard resources
	$\mid 0p \mid 1p \mid\sim M \mid M \cdot N$	permission expressions
	$\mid \texttt{compatible}(M, N)$	
	$\mid \texttt{x} \mid n \mid M + N$	program expressions
	$\mid M - N \mid M * N \mid M/N$	
	$\mid M \leq N \mid M < N$	
	$\mid M \geq N \mid M > N$	
	$\mid \epsilon \mid M, N$	list expressions

Definition 33 ($- \vdash - : \mathsf{Val}$, Program Value Expressions)**.** Program value expressions precisely reflect arithmetic expressions and comparisons as per the object language. We do not distinguish between expressions in the assertion logic and expressions in programs; the only difference is that assertions additionally allow for logical variables.

Well-typed program value expressions are given by the following rules:

$$- \vdash - : \mathsf{Val} \subseteq \mathsf{Context} \times \mathsf{Term}$$

$$\frac{\Gamma(x) = \mathsf{Val}}{\Gamma \vdash x : \mathsf{Val}} \qquad \frac{}{\Gamma \vdash \texttt{x} : \mathsf{Val}} \qquad \frac{}{\Gamma \vdash n : \mathsf{Val}}$$

$$\frac{\Gamma \vdash M : \mathsf{Val} \qquad \Gamma \vdash N : \mathsf{Val} \qquad \mathrm{op} \in \{+, -, *, /\}}{\Gamma \vdash M \ \mathrm{op}\ N : \mathsf{Val}}$$

$$\frac{\Gamma \vdash M : \mathsf{Val} \qquad \Gamma \vdash N : \mathsf{Val} \qquad \mathrm{comp} \in \{<, \leq, \geq, >\}}{\Gamma \vdash M \ \mathrm{comp}\ N : \mathsf{Val}}$$

Definition 34 ($- \vdash - : \mathsf{Perm}$, Permission Assertions)**.** Well-typed permission expression are given by the following rules:

$$- \vdash - : \mathsf{Perm} \subseteq \mathsf{Context} \times \mathsf{Term}$$

$$\frac{\Gamma(x) = \mathsf{Perm}}{\Gamma \vdash x : \mathsf{Perm}} \qquad \frac{}{\Gamma \vdash 0p : \mathsf{Perm}} \qquad \frac{}{\Gamma \vdash 1p : \mathsf{Perm}}$$

$$\frac{\Gamma \vdash M : \mathsf{Perm}}{\Gamma \vdash\sim M : \mathsf{Perm}} \qquad \frac{\Gamma \vdash M : \mathsf{Perm} \qquad \Gamma \vdash N : \mathsf{Perm}}{\Gamma \vdash M \cdot N : \mathsf{Perm}}$$

Definition 35 ($- \vdash - : \mathsf{Pure}$, Pure Assertions)**.** Well-typed pure assertions are given by the following rules. They are "pure" in that they do not contain spatial or region assertions.

$$- \vdash - : \mathsf{Pure} \subseteq \mathsf{Context} \times \mathsf{Term}$$

$$\frac{}{\Gamma \vdash \bot : \mathsf{Pure}} \qquad \frac{}{\Gamma \vdash \top : \mathsf{Pure}}$$

$$\frac{\Gamma \vdash M : \mathsf{Pure} \qquad \Gamma \vdash N : \mathsf{Pure}}{\Gamma \vdash M \wedge N : \mathsf{Pure}} \qquad \frac{\Gamma \vdash M : \mathsf{Pure} \qquad \Gamma \vdash N : \mathsf{Pure}}{\Gamma \vdash M \vee N : \mathsf{Pure}}$$

$$\frac{\Gamma \vdash M : \mathsf{Pure} \qquad \Gamma \vdash N : \mathsf{Pure}}{\Gamma \vdash M \Rightarrow N : \mathsf{Pure}} \qquad \frac{\Gamma \vdash M : \mathsf{Pure}}{\Gamma \vdash \neg M : \mathsf{Pure}}$$

$$\frac{\Gamma \vdash M : \tau \qquad \Gamma \vdash N : \tau}{\Gamma \vdash M =_\tau N : \mathsf{Pure}} \qquad \frac{\Gamma, x : \tau \vdash M : \mathsf{Pure}}{\Gamma \vdash \forall x : \tau . M : \mathsf{Pure}}$$

$$\frac{\Gamma, x : \tau \vdash M : \mathsf{Pure}}{\Gamma \vdash \exists x : \tau . M : \mathsf{Pure}} \qquad \frac{\Gamma \vdash M : \mathsf{Perm} \qquad \Gamma \vdash N : \mathsf{Perm}}{\Gamma \vdash \mathtt{compatible}(M, N) : \mathsf{Pure}}$$

Definition 36 ($- \vdash - : \mathsf{Guard}$, Guard Expressions)**.** Well-typed guard expressions are given by the following rules, wherein G ranges over the set of primitive guards:

$$- \vdash - : \mathsf{Guard} \subseteq \mathsf{Context} \times \mathsf{Term}$$

$$\frac{}{\Gamma \vdash G : \mathsf{Guard}} \qquad \frac{\Gamma \vdash M : \mathsf{Perm}}{\Gamma \vdash G[M] : \mathsf{Guard}} \qquad \frac{\Gamma \vdash M : \mathsf{Val}}{\Gamma \vdash G(M) : \mathsf{Guard}}$$

$$\frac{\Gamma \vdash M : \mathsf{Guard} \qquad \Gamma \vdash N : \mathsf{Guard}}{\Gamma \vdash M * N : \mathsf{Guard}}$$

Definition 37 ($- \vdash - : \mathsf{Region}$, Region Identifiers)**.**

$$- \vdash - : \mathsf{Region} \subseteq \mathsf{Context} \times \mathsf{Term}$$

$$\frac{\Gamma(x) = \mathsf{Region}}{\Gamma \vdash x : \mathsf{Region}}$$

Definition 38 ($-; - \vdash - : \mathsf{Assn}$, Assertions)**.** Well-typed assertions are given by the following rules.

$$-; - \vdash - : \mathsf{Assn} \subseteq \mathcal{P}(\mathit{RegionType}) \times \mathsf{Context} \times \mathsf{Term}$$

$$\frac{\Gamma \vdash M : \mathsf{Pure}}{R; \Gamma \vdash M : \mathsf{Assn}}$$

$$\frac{\Gamma \vdash M : \mathsf{Pure} \qquad R; \Gamma \vdash N : \mathsf{Assn} \qquad \mathrm{dom}(R); \Gamma \vdash O : \mathsf{Assn}}{R; \Gamma \vdash M \mathbin{?} N : O : \mathsf{Assn}}$$

$$\frac{R; \Gamma \vdash M : \mathsf{Assn} \qquad R; \Gamma \vdash N : \mathsf{Assn}}{R; \Gamma \vdash M * N : \mathsf{Assn}}$$

$$\frac{\Gamma \vdash M : \mathsf{Val} \qquad \Gamma \vdash N : \mathsf{Val}}{R; \Gamma \vdash M \mapsto N : \mathsf{Assn}} \qquad \frac{\Gamma \vdash M : \mathsf{Val} \qquad \Gamma \vdash N : \mathsf{Val}}{R; \Gamma \vdash M \mapsto [N] : \mathsf{Assn}}$$

$$\frac{}{R; \Gamma \vdash \mathtt{emp} : \mathsf{Assn}} \qquad \frac{\Gamma \vdash x : \mathsf{Region} \qquad \Gamma \vdash M : \mathsf{Guard}}{R; \Gamma \vdash x@(M) : \mathsf{Assn}}$$

$$\frac{\begin{array}{c} \mathbb{T}(r, \vec{x} : \vec{\tau}) \in R \\ \Gamma \vdash x : \mathsf{Region} \qquad |\vec{M}| = |\vec{\tau}| \qquad \Gamma \vdash M_i : \tau_i \qquad \Gamma \vdash N : \mathsf{Val} \end{array}}{R; \Gamma \vdash \mathbb{T}(x, \vec{M}, N) : \mathsf{Assn}}$$

Assertions also contain the usual first order logic connectives:

$$\frac{R; \Gamma \vdash M : \mathsf{Assn} \qquad R; \Gamma \vdash N : \mathsf{Assn} \qquad \mathrm{op} \in \{\vee, \wedge, \Rightarrow\}}{R; \Gamma \vdash M \mathrel{\mathrm{op}} N : \mathsf{Assn}}$$

$$\frac{R; \Gamma, x : \tau \vdash M : \mathsf{Assn} \qquad \mathrm{quan} \in \{\forall, \exists\}}{R; \Gamma \vdash \mathrm{quan}\; x.M : \mathsf{Assn}} \qquad \frac{\mathrm{c} \in \{\top, \bot\}}{R; \Gamma \vdash \mathrm{c} : \mathsf{Assn}}$$

$$\frac{\Gamma \vdash M : \tau \qquad \Gamma \vdash N : \tau}{R; \Gamma \vdash M =_\tau N : \mathsf{Assn}} \qquad \frac{R; \Gamma \vdash M : \mathsf{Assn}}{R; \Gamma \vdash \neg M : \mathsf{Assn}}$$

3.5.1 *Entailment Logic*

Definition 39 (Syntactic Entailment of Assertions)**.** We define syntactic entailment on those as a relation:

$$-;- \mid - \vdash - \subseteq \mathcal{P}(\mathit{RegionType}) \times \mathsf{Context} \times \mathcal{P}(\mathsf{Assn}) \times \mathsf{Assn}$$

An entailment $R; \Gamma \mid P_1, ..., P_n \vdash Q$ is well-formed when

$$R; \Gamma \vdash P_i : \mathsf{Assn} \qquad\qquad R; \Gamma \vdash Q : \mathsf{Assn}$$

The rules of the entailment logic are a standard first-order intuitionistic separation logic.

3.6 Specification Logic

3.6.1 *Specification Syntax*

Definition 40 (*GAD*, Guard Algebra Declaration)**.** A guard algebra declaration is an expression from the following grammar of guard algebra combinators, where G ranges over primitive guard symbols:

$$\begin{array}{rcll} GAD\ g & ::= & G & \text{Indivisible Guard} \\ & | & \%G & \text{Divisible Guard} \\ & | & \#G & \text{Parameterized Guard} \\ & | & g * g & \text{Product Construction} \\ & | & g + g & \text{Sum Construction} \\ & | & 0 & \text{Nil Guard} \end{array}$$

A guard algebra declaration is well-formed when a primitive guard symbol G occurs at most once in the declaration.

Definition 41 (*ID*, Region Interpretation Declaration)**.** Region interpretations are a collection of clauses of the form

$$(\Delta).\Pi \mid e : P$$

where Δ is a logical context and Π, e and P are terms of the assertion logic.

A single clause is well-formed with respect to a set of region types R, variable r and variables $\vec{x}$ of types $\vec{\tau}$ if

- $r : \mathsf{Region}, \vec{x} : \vec{\tau}, \Delta \vdash \Pi : \mathsf{Pure}$
- $r : \mathsf{Region}, \vec{x} : \vec{\tau}, \Delta \vdash e : \mathsf{Val}$
- $R; r : \mathsf{Region}, \vec{x} : \vec{\tau}, \Delta \vdash P : \mathsf{Assn}$

A whole region interpretation declaration is well-formed with respect to a region declaration context R, identifier r and variables $\vec{x}$ of types $\vec{\tau}$ when, in addition to each clause being well-formed.

Definition 42 (*AD*, Region Action Declaration)**.** A region action declaration is a collection of clauses of the form

$$(\Delta).\Pi \mid G : e_1 \rightsquigarrow e_2$$

where Δ is a logical context, P, G, e_1 and e_2 are terms in the assertion logic.

A single clause is well-formed with respect to a variable r and variables $\vec{x}$ of types $\vec{\tau}$ when

- $r : \mathsf{Region}, \vec{x} : \vec{\tau}, \Delta \vdash \Pi : \mathsf{Pure}$
- $r : \mathsf{Region}, \vec{x} : \vec{\tau}, \Delta \vdash G : \mathsf{Guard}$

- $r : \mathsf{Region}, \vec{x} : \vec{\tau}, \Delta \vdash e_1 : \mathsf{Val}$
- $r : \mathsf{Region}, \vec{x} : \vec{\tau}, \Delta \vdash e_2 : \mathsf{Val}$

Definition 43 (RegionDecl, Region Declarations)**.** A region declaration is a triple of a guard algebra declaration, a region interpretation declaration and a region action declaration:

$$\mathsf{RegionDecl} := \{(g, i, a) \mid g \in GAD, i \in ID, a \in AD\}$$

A region declaration (g, i, a) is well-formed with respect to a collection of region types R, logical variable r and logical variables $\vec{x}$ of types $\vec{\tau}$ when:

- g is a well-formed guard algebra declaration
- i is a well-formed region interpretation declaration with respect to R, r and $\vec{x} : \vec{\tau}$
- a is a well-formed action declaration with respect to r and $\vec{x} : \vec{\tau}$

Definition 44 (RegionContext, Region Contexts)**.** A region declaration contexts is defined by the following grammar

$$R ::= \cdot \mid R, \mathbb{T}(r, \vec{x} : \vec{\tau})(g, i, a)$$

We denote the set of region types associated in the region context R as $\mathrm{dom}(R)$.

A region context R is well-formed when each individual region declaration (g, i, a) is well-formed with respect to the associated region type parameters r, $\vec{x}$ and $\vec{\tau}$, and the collection of all the region types $\mathrm{dom}(R)$, and each region type is associated at most once.

These conditions are enough for a well-formed region context to induce a partial mapping from region types to well-formed region declarations, and we denote a well-defined lookup as follows:

$$R(\mathbb{T}(r, \vec{x} : \vec{\tau})) = (g, i, a)$$

Definition 45 ($I(-)$, Interpretation Function)**.** A well-formed region context induces a function on well-typed region assertions to spatial assertions, intuitively computing the "symbolic" interpretation of a region assertion. The region context in question will be clear from context.

For a region context R and a well-formed region assertion

$$\mathrm{dom}(R); \Gamma \vdash \mathbb{T}(x, \vec{M}, N) : \mathsf{Assn}$$

we define the interpretation as follows: Let $\mathbb{T}(r, \vec{x} : \vec{\tau})$ be the associated region type in R. Then, for each clause in the region interpretation, say

$$(\Delta).\Pi \mid e : P$$

We can build the assertion

$$\mathrm{dom}(R); \Gamma \vdash (\exists \Delta.\Pi \wedge N = e \wedge P)[x, \vec{M}/r, \vec{x}] : \mathsf{Assn}$$

where we close the assertions in the clause by the concrete parameters at hand, and the local context is closed existentially. Then, the interpretation is the disjunction of all such clauses:

$$I(\mathbb{T}(x, \vec{M}, N)) = \bigvee_{(\Delta).\Pi | e:P} (\exists \Delta.\Pi \wedge N = e \wedge P)[x, \vec{M}/r, \vec{x}]$$

Definition 46 (*SpecCtxt*, Function Specification Context)**.** Function specification contexts assign specifications to function symbols. They are generated by the following grammar, where Γ is a logical context, and $\vec{y}$ is a sequence of logical variables:

$$\Phi \in \mathit{SpecCtxt} ::= \cdot \mid \Phi, \mathtt{f} : (\Gamma, \vec{y})\{P\}\{r.Q\}$$

A function specification context is well-formed with regards to a set of region types R when

$$R; \Gamma, \vec{y} : \mathsf{Val} \vdash P : \mathsf{Assn} \qquad R; \Gamma, \vec{y} : \mathsf{Val}, r : \mathsf{Val} \vdash Q : \mathsf{Assn}$$

where P and Q do not mention any program variables. Furthermore we require that $\mathtt{f}$ occurs at most once in Φ.

These conditions are enough for a well-formed function specification context to induce a partial mapping from function names to function specifications, and we denote a successful lookup with

$$\Phi(\mathtt{f}) = (\Gamma, \vec{y})\{P\}\{r.Q\}$$

Definition 47 (Spec, Statement Specifications)**.** The set of syntactic statement specifications is defined by the following grammar, where $P, Q, U \in \mathsf{Term}$, $r \in \mathit{LVar}$ and $\mathtt{s} \in \mathsf{Stmt}$.

$$s \in \mathsf{Spec} ::= \{P\}\mathtt{s}\{Q \mid r.U\}$$

A statement specification is well-typed wrt. a region context R, function specification context Φ (well-typed wrt. $\mathrm{dom}(R)$) and logical context Γ according to the following judgment:

$$\frac{\begin{array}{l}\mathrm{dom}(R); \Gamma \vdash P : \mathsf{Assn} \\ \mathrm{dom}(R); \Gamma \vdash Q : \mathsf{Assn} \\ \mathrm{dom}(R); \Gamma, r : \mathsf{Val} \vdash U : \mathsf{Assn}\end{array}}{R; \Phi; \Gamma \vdash \Big\{P\Big\}\ \mathtt{s}\ \Big\{Q \mid r.U\Big\} : \mathsf{Spec}}$$

Definition 48 ($-$ atomic, Atomic Statements)**.** We declare a subset of Stmt as atomic:

$$\frac{}{\mathtt{x} := \mathtt{CAS}(\mathtt{e}_1,\mathtt{e}_2,\mathtt{e}_3)\ \text{atomic}} \qquad \frac{}{[\mathtt{e}_1] := \mathtt{e}_2\ \text{atomic}}$$

$$\frac{}{\mathtt{x} := [\mathtt{e}]\ \text{atomic}}$$

Definition 49 (Atomic Statement Specifications)**.** The set of syntactic atomic statement specifications is given by the following grammar where $P, Q \in \mathsf{Term}$, $\mathtt{s} \in \mathsf{Stmt}$ with $\mathtt{s}$ atomic:

$$\mathsf{Atomic} ::= \langle P \rangle \mathtt{s} \langle Q \rangle$$

An atomic statement specification is well-typed wrt. to collection of region identifiers S, a region declaration R and a logical context Γ according to the following judgment:

$$\frac{R; \Gamma \vdash P : \mathsf{Assn} \qquad R; \Gamma \vdash Q : \mathsf{Assn} \qquad \forall x \in S.\ \Gamma \vdash x : \mathsf{Region}}{R; \Gamma \vdash_S \langle P \rangle\ \mathtt{s}\ \langle Q \rangle : \mathsf{Atomic}}$$

Definition 50 (Region Opening)**.** We define a judgment for expressing the opening of regions, given by the following grammar where I is some finite index set P, Q_i are assertions, Δ_i are contexts and $\vec{r}_i$ are sequences of region identifiers:

$$[P] \text{ open } [\{(\Delta_i).(Q_i, \vec{r}_i\}_{i \in I}]$$

An opening judgment is well-formed with regards to a set of region-types R and a logical context Γ according to the following judgment:

$$\frac{\begin{array}{l} R; \Gamma \vdash P : \mathsf{Assn} \\ \forall i \in I. R; \Gamma, \Delta_i, \vec{r}_i : \mathsf{Region} \vdash Q_i : \mathsf{Assn} \end{array}}{R; \Gamma \vdash [P] \text{ open } [\{(\Delta_i).(Q_i, \vec{r}_i\}_{i \in I}]}$$

Definition 51 (Region Closing)**.** We define a judgment for closing regions, given by the following grammar where P, Q are assertions and $\vec{r}$ a sequence of region identifiers:

$$[P] \text{ close}(\vec{r})\ [Q]$$

A closing judgment is well-formed with regards to a set of region types R and a logical context Γ according to the following judgment:

$$\frac{R; \Gamma \vdash P : \mathsf{Assn} \qquad R; \Gamma \vdash Q : \mathsf{Assn}}{R; \Gamma \vdash [P] \text{ close}(\vec{r})\ [Q]}$$

Convention 52. For two sequences of assertion logic expressions of equal length n, $\vec{x}$ and $\vec{y}$, we use the notation $\vec{x} = \vec{y}$ as notation for the assertion $x_1 = y_1 * \ldots * x_n = y_n$.

Definition 53 ($\mathsf{FunctionSpec}$, Function Specifications)**.** The set of syntactic function specifications is defined by the following grammar, letting $\mathtt{f} \in \mathsf{Fun}$, $\vec{\mathtt{x}} \in \mathsf{Var}^*$, $P, Q \in \mathsf{Term}$ are assertions that does not mention program variables and $\mathtt{s} \in \mathsf{Stmt}$, $\Gamma \in \mathsf{Context}$ and $\vec{y} \in LVar^*$.

$$\mathsf{FunctionSpec} ::= \mathtt{f}(\vec{\mathtt{x}})(\Gamma, \vec{y})\{P\}\mathtt{s}\{r.Q\}$$

A function specification is well-typed wrt. a region context R and a function specification context Φ if its constituents form a well-typed statement specification.

$$\frac{R;\Phi;\Gamma,\vec{y} : \mathsf{Val} \vdash \left\{P * \vec{\mathtt{x}} = \vec{y}\right\}\ \mathtt{s}\ \left\{\forall r.Q \mid r.Q\right\} : \mathsf{Spec}}{R;\Phi \vdash \mathtt{f}(\vec{\mathtt{x}})(\Gamma,\vec{y})\{P\}\mathtt{s}\{r.Q\} : \mathsf{FunctionSpec}}$$

Definition 54 ($\lceil - \rceil$, Function Specification Erasure (Specification))**.** We define

$$\lceil - \rceil : \mathsf{FunctionSpec}^* \rightarrow \mathit{SpecCtxt}$$

as follows:

$$\begin{aligned}\lceil \epsilon \rceil &= \cdot \\ \lceil \vec{\mathtt{f}}, \mathtt{f}_i(\vec{\mathtt{x}})(\Gamma,\vec{y})\{P\}\mathtt{s}\{r.Q\} \rceil &= \lceil \vec{\mathtt{f}} \rceil, \mathtt{f}_i : (\Gamma,\vec{y})\{P\}\{r.Q\}\end{aligned}$$

Definition 55 (ProgramSpec, Program Specifications)**.** The set of syntactic program specifications is defined as sequences of region declarations and function specifications:

$$\mathsf{ProgramSpec} := \mathsf{RegionDecl}^*; \mathsf{FunctionSpec}^*$$

A program specification is well-typed when each individual function specification is well-typed wrt. to the region and function specification contexts described by the region and function declarations, respectively:

$$\frac{\vec{r}; \lceil \vec{\mathtt{f}} \rceil \vdash \mathtt{f}_i : \mathsf{FunctionSpec}, \qquad \forall \mathtt{f}_i \in \vec{\mathtt{f}}}{\vdash \vec{r}; \vec{\mathtt{f}} : \mathsf{ProgramSpec}}$$

3.6.2 *Specification Rules*

3.6.2.1 *Program Specification*

We say that we can derive a program specification when we can derive each of the individual function specifications with respect to the region and function contexts specified by the program specification.

$$\frac{\vec{r}; \lceil \vec{\mathtt{f}} \rceil \vdash \mathtt{f}_i, \qquad \forall \mathtt{f}_i \in \vec{\mathtt{f}}}{\vdash \vec{r}; \vec{\mathtt{f}}}$$

3.6.2.2 *Function Specification*

We say that we can derive a function specification when we can derive its implementation with respect to the pre- and post-condition imposed by the function specification.

$$\frac{R;\Phi;\Gamma,\vec{y} : \mathsf{Val} \vdash \left\{P * (\vec{\mathtt{x}} = \vec{y})\right\}\ \mathtt{s}\ \left\{\forall r.Q \mid r.Q\right\}}{R;\Phi \vdash \mathtt{f}(\vec{\mathtt{x}})(\Gamma,\vec{y})\{P\}\mathtt{s}\{r.Q\}}$$

3.6.2.3 *Statement Specification*

Definition 56 (mod(−), Modifies Sets)**.** We define the set of local variables modified by a given statement as follows:

$$\begin{aligned}
\mathrm{mod} &: \mathsf{Stmt} \to \mathcal{P}(\mathsf{Var}) \\
\mathrm{mod}(\mathtt{s}_1;\mathtt{s}_2) &= \mathrm{mod}(\mathtt{s}_1) \cup \mathrm{mod}(\mathtt{s}_2) \\
\mathrm{mod}(\mathtt{if}(\mathtt{e})\ \mathtt{then}\ \{\mathtt{s}_1\}\ \mathtt{else}\ \{\mathtt{s}_2\}) &= \mathrm{mod}(\mathtt{s}_1) \cup \mathrm{mod}(\mathtt{s}_2) \\
\mathrm{mod}(\mathtt{while}(\mathtt{e})\{\mathtt{s}\}) &= \mathrm{mod}(\mathtt{s}) \\
\mathrm{mod}(\mathtt{x} := \mathtt{alloc}(\mathtt{e})) &= \{\mathtt{x}\} \\
\mathrm{mod}(\mathtt{x} := \mathtt{CAS}(\mathtt{e}_1,\mathtt{e}_2,\mathtt{e}_3)) &= \{\mathtt{x}\} \\
\mathrm{mod}(\mathtt{x} := \mathtt{e}) &= \{\mathtt{x}\} \\
\mathrm{mod}([\mathtt{e}_1] := \mathtt{e}_2) &= \emptyset \\
\mathrm{mod}(\mathtt{x} := [\mathtt{e}]) &= \{\mathtt{x}\} \\
\mathrm{mod}(\mathtt{x} := \mathtt{f}(\vec{\mathtt{e}})) &= \{\mathtt{x}\} \\
\mathrm{mod}(\mathtt{return}\ \mathtt{e}) &= \emptyset \\
\mathrm{mod}(\mathtt{fork}\ \mathtt{f}(\vec{\mathtt{e}})) &= \emptyset \\
\mathrm{mod}(\mathtt{skip}) &= \emptyset
\end{aligned}$$

$$\dfrac{\Phi(\texttt{f}) = (\Gamma, \vec{y})\{P\}\{r.Q\}}{R;\Phi;\Gamma,\vec{y} : \mathsf{Val} \vdash \left\{P * (\vec{\mathtt{e}} = \vec{y})\right\}\ \texttt{x:=f}(\vec{\mathtt{e}})\ \left\{\exists r.\mathtt{x} = r * Q \mid r.U\right\}}$$

$$\dfrac{}{R;\Phi;\Gamma \vdash \left\{\top\right\}\ \texttt{return e}\ \left\{Q \mid r.\mathtt{e} = r\right\}}$$

$$\dfrac{R;\Phi;\Gamma \vdash \left\{P * \mathtt{e} \neq 0\right\}\ \mathtt{s}_1\ \left\{Q \mid r.U\right\} \qquad R;\Phi;\Gamma \vdash \left\{P * \mathtt{e} = 0\right\}\ \mathtt{s}_2\ \left\{Q \mid r.U\right\}}{R;\Phi;\Gamma \vdash \left\{P\right\}\ \texttt{if}(\mathtt{e})\ \texttt{then}\ \{\mathtt{s}_1\}\ \texttt{else}\ \{\mathtt{s}_2\}\ \left\{Q \mid r.U\right\}}$$

$$\dfrac{R;\Phi;\Gamma \vdash \left\{I * \mathtt{e} \neq 0\right\}\ \mathtt{s}\ \left\{I \mid r.U\right\}}{R;\Phi;\Gamma \vdash \left\{I\right\}\ \texttt{while}(\mathtt{e})\{\mathtt{s}\}\ \left\{I * \mathtt{e} = 0 \mid r.U\right\}}$$

$$\dfrac{}{R;\Phi;\Gamma \vdash \left\{\mathtt{e} = v * v > 0\right\}\ \mathtt{x} := \texttt{alloc}(\mathtt{e})\ \left\{\exists n.\mathtt{x} = n * n \mapsto [v] \mid r.U\right\}}$$

$$\dfrac{}{R;\Phi;\Gamma \vdash \left\{P\right\}\ \texttt{skip}\ \left\{P \mid r.U\right\}}$$

$$\dfrac{R;\Phi;\Gamma \vdash \left\{P\right\}\ \mathtt{s}_1\ \left\{S \mid r.U\right\} \qquad R;\Phi;\Gamma \vdash \left\{S\right\}\ \mathtt{s}_2\ \left\{Q \mid r.U\right\}}{R;\Phi;\Gamma \vdash \left\{P\right\}\ \mathtt{s}_1;\mathtt{s}_2\ \left\{Q \mid r.U\right\}}$$

$$\dfrac{}{R;\Phi;\Gamma \vdash \left\{\mathtt{x} = \mathtt{e}_0\right\}\ \mathtt{x} := \mathtt{e}\ \left\{\mathtt{x} = \mathtt{e}[\mathtt{e}_0/\mathtt{x}] \mid r.U\right\}}$$

$$\dfrac{\Phi(\texttt{f}) = (\Gamma, \vec{y})\{P\}\{r.Q\}}{R;\Phi;\Gamma \vdash \left\{P * (\vec{\mathtt{e}} = \vec{y})\right\}\ \texttt{fork f}(\vec{\mathtt{e}})\ \left\{\top \mid r.U\right\}}$$

$$\dfrac{R;\Phi;\Gamma \vdash \left\{P\right\}\ \mathtt{s}\ \left\{Q \mid r.U\right\} \qquad \mathrm{mod}(\mathtt{s}) \cap F = \emptyset}{R;\Phi;\Gamma \vdash \left\{P * F\right\}\ \mathtt{s}\ \left\{Q * F \mid r.U\right\}}$$

$$\dfrac{R;\Gamma \mid P \vdash P' \qquad R;\Phi;\Gamma \vdash \left\{P'\right\}\ \mathtt{s}\ \left\{Q' \mid r.U'\right\} \qquad R;\Gamma \mid Q' \vdash Q \qquad R;\Gamma, r : \mathsf{Val} \mid U' \vdash U}{R;\Phi;\Gamma \vdash \left\{P\right\}\ \mathtt{s}\ \left\{Q \mid r.U\right\}}$$

$$\dfrac{\begin{array}{l} R;\Gamma \vdash [P]\ \mathrm{open}\ [\{(\Delta_i).(P_i', \vec{r}_i)\}_{i \in I}] \\ \forall i \in I. \\ R;\Gamma,\Delta_i \vdash_{\{\vec{r}_i\}} \langle P_i' \rangle\ \mathtt{s}\ \langle Q_i * \mathit{newRegion}(\vec{n}_i)\rangle \\ R;\Gamma,\Delta_i \vdash [Q_i * \mathit{newRegion}(\vec{n}_i)]\ \mathrm{close}(\vec{r}_i, \vec{n}_i)\ [Q] \end{array}}{R;\Phi;\Gamma \vdash \left\{P\right\}\ \mathtt{s}\ \left\{\mathrm{stabilize}(Q) \mid r.U\right\}}$$

3.6.2.4 *Atomic Statement Specifications*

$$\frac{}{R;\Gamma \vdash_S \langle \mathsf{e}_1 \mapsto \_ \rangle\ [\mathsf{e}_1] := \mathsf{e}_2\ \langle \mathsf{e}_1 \mapsto \mathsf{e}_2 \rangle}$$

$$\frac{}{R;\Gamma \vdash_S \langle \mathsf{e} = n * n \mapsto v \rangle\ \mathtt{x} := [\mathsf{e}]\ \langle \mathtt{x} = v * n \mapsto v \rangle}$$

$$\frac{}{\begin{array}{c} R;\Gamma \vdash_S \langle \mathsf{e}_1 = a * a \mapsto v * \mathsf{e}_2 = old * \mathsf{e}_3 = new \rangle \\ \mathtt{x} := \mathtt{CAS}(\mathsf{e}_1,\mathsf{e}_2,\mathsf{e}_3) \\ \langle (\mathtt{x} \neq 0 * v = old * a \mapsto new) \vee (\mathtt{x} = 0 * v \neq old * a \mapsto v \rangle \end{array}}$$

3.7 Model

We here describe the constructions in Sets that we use to interpret the specification logic.

Definition 57 (*Perm*, Permissions)**.** We define the set of *Perm* as the free atomless boolean algebra over an infinite set.

Definition 58 (*RegionId*, Region Identifiers)**.** We assume a set of region identifiers, *RegionId*.

Definition 59 (*LVal*, Logical Values)**.** We define the set of logical values to be the disjoint union of program values, permissions and region identifiers:

$$LVal := Val + Perm + RegionId$$

recalling the definition of program values from Definition 9.

Definition 60 (*AbstractState*, Abstract States)**.** We define the set of abstract region states as the set of program values:

$$AbstractState := Val$$

Definition 61 (PCMZ, Partial Commutative Monoid with Zero)**.** A PCMZ M is a 4-tuple $(|M|, \epsilon, 0, \cdot)$ consisting of an underlying set $|M|$ with a distinguished element ϵ (the unit), a distinguished element 0 (the zero) and a partial composition on that set (multiplication or simply 'composition', noted $\cdot$) satisfying the following requirements for all $a, b, c \in |M|$, where $x \downarrow y$ means '$x \cdot y$ is defined'.

- $a \downarrow b$ implies ($b \downarrow a$ and $a \cdot b = b \cdot a$).
- $\epsilon \downarrow a$ and $\epsilon \cdot a = a$.
- $0 \downarrow a$ and $0 \cdot a = 0$.
- ($b \downarrow c$ and $a \downarrow (b \cdot c)$) implies ($a \downarrow b$ and $(a \cdot b) \downarrow c$ and $a \cdot (b \cdot c) = (a \cdot b) \cdot c$).

We note that the *free* PCMZ over a set X is the usual, initial construction of a structure over an underlying set. Note that by initiality, any function f from X into the underlying set of another PCMZ, say $|M|$, gives us a canonical PCMZ homomorphism from the free PCMZ over X to M, denoted $\bar{f}$.

Definition 62 (*Guard*, The PCMZ of Primitive Guards)**.** We define *Guard* as the free PCMZ over a set of expressions over *PrimitiveGuard*, call it $|Guard|$, built according to the following grammar wherein G ranges over *PrimitiveGuard*, π ranges over *Perm* and x ranges over *LVar*:

$$\begin{aligned} |Guard| ::= {} & G \\ \mid {} & G[\pi] \\ \mid {} & G(x) \end{aligned}$$

Definition 63 (*GuardAlgebra*, Guard Algebras)**.**

$$\sum_{G:PCMZ} |Guard| \to |G|$$

Definition 64 (*GuardAlgebraAssignment*, Guard Algebra Assignment)**.** We define a guard algebra assignment as a partial finite map from region types to guard algebras,

$$GuardAlgebraAssignment := RegionType \overset{\text{fin}}{\rightharpoonup} GuardAlgebra$$

Definition 65 (*RegionAssignment*, Region Assignments)**.** We define region assignments as partial finite maps from region identifiers to triples describing their (indexed) region type, state, and which guards we have in possession.

$$\begin{aligned} & RegionAssignment := RegionId \overset{\text{fin}}{\rightharpoonup} \\ & \quad ((RegionType \times LVal^*) \times AbstractState \times |Guard|) \end{aligned}$$

We refer to the first, second and third component of the codomain as $\alpha(r)$.type, $\alpha(r)$.state and $\alpha(r)$.guards for $\alpha \in RegionAssignment, r \in \mathrm{dom}(\alpha)$.

Region assignments form a partial commutative monoid where composition is defined as follows - we use $\bot$ to denote undefined values, i.e. $f(a) = \bot$ if $a \notin \mathrm{dom}(f)$.

$$(a_1 \cdot a_2)(r) =$$

$$\begin{cases} a_1(r) & a_2(r) = \bot \\ a_2(r) & a_1(r) = \bot \\ (a_1(r).\text{type}, a_1(r).\text{state}, a_1(r).\text{guards} \cdot a_2(r).\text{guards}) & a_1(r).\text{type} = a_2(r).\text{type} \\ & \wedge\ a_1(r).\text{state} = a_2(r).\text{state} \\ \bot & \text{otherwise} \end{cases}$$

Observe that the empty map is a unit. Associativity and commutativity is inherited by the properties of equality and the PCM *Guard*.

This gives rise to canonical partial order known as the extension order, here written in full: we say $a_1 \sqsubseteq a_2$ iff $\mathrm{dom}(a_1) \subseteq \mathrm{dom}(a_2)$ and for all $r \in \mathrm{dom}(a_1)$:

1. $a_1(r).\mathrm{type} = a_2(r).\mathrm{type}$
2. $a_1(r).\mathrm{state} = a_2(r).\mathrm{state}$
3. $a_1(r).\mathrm{guards} \sqsubseteq a_2(r).\mathrm{guards}$

Definition 66 (Abstract Configuration)**.** Abstract machine configurations consists of a heap and region assignment pair:

$$\mathit{Heap} \times \mathit{RegionAssignment}$$

Abstract configurations form a partial commutative monoid by pointwise lifting of both unit and composition. This also gives rise to the pointwise extension order:

$$(h_1, a_1) \leq (h_2, a_2) \overset{\text{def}}{\iff} \exists(h_3, a_3).\ h_1 \cdot h_3 = h_2 \wedge a_1 \cdot a_3 = a_2$$

Definition 67 (*Assertion*, Assertions)**.** We define assertions as upwards closed subsets of abstract configurations, ordered according to the canonical extension order.

$$\mathit{Assertion} := \mathcal{P} \uparrow (\mathit{Heap} \times \mathit{RegionAssignment})$$

Definition 68 (*RegionInterpretation*, Region Interpretations)**.**

$$\mathit{RegionInterpretation} := \mathit{AbstractState} \pm \mathit{Assertion}$$

Definition 69 (*LTS*, Labeled Transition Systems)**.**

$$\mathit{LTS} := |\mathit{Guard}| \xrightarrow{\text{mon}} \mathcal{P}(\mathit{AbstractState} \times \mathit{AbstractState})$$

Definition 70 (*RegionTypeAssignment*, Region Type Assignments)**.** A region type assignment associates a region interpretation, labeled transition system and guard algebra with each (instantiated) region type.

$$\begin{aligned}\mathit{RegionTypeAssignment} :=&(\mathit{RegionType} \times \mathit{LVal}^*) \pm \\ &\qquad \mathit{RegionInterpretation} \times \mathit{LTS} \times \mathit{GuardAlgebra}\end{aligned}$$

We refer to the three components of the type assignment

$$\rho(\mathbb{T}(r, \vec{x} : \vec{\tau}), \vec{v})$$

as follows:

$$\rho(\mathbb{T}(r, \vec{x} : \vec{\tau}), \vec{v}).\mathrm{int} \qquad \rho(\mathbb{T}(r, \vec{x} : \vec{\tau}), \vec{v}).\mathrm{lts} \qquad \rho(\mathbb{T}(r, \vec{x} : \vec{\tau}), \vec{v}).\mathrm{GA}$$

The choice of guard algebra must be independent of the instantiated values, i.e. for a given $\rho \in \mathit{RegionTypeAssignment}$, for any region type $\mathbb{T}(r, \vec{x} : \vec{\tau})$ and two choices of values $\vec{v}$ and $\vec{u}$ it must hold that

$$\rho(\mathbb{T}(r, \vec{x} : \vec{\tau}), \vec{v}) = \rho(\mathbb{T}(r, \vec{x} : \vec{\tau}), \vec{u})$$

Definition 71 ($GA(-)$, RTA Algebra Assignment Erasure)**.** We denote the erasure of region interpretation and labeled transition system from a region type assignment ρ as $GA(\rho)$, erasing a

$$(\mathit{RegionType} \times \mathit{LVal}^*) \overset{\text{fin}}{\rightharpoonup} \mathit{RegionInterpretation} \times \mathit{LTS} \times \mathit{GuardAlgebra}$$

to a

$$\mathit{RegionType} \overset{\text{fin}}{\rightharpoonup} \mathit{GuardAlgebra}$$

3.8 Interpretation

Definition 72 (Interpretation of Primitive Types)**.** We interpret the primitive types into the corresponding sets:

$$\begin{aligned} [\![\mathsf{Val}]\!] &= \mathit{Val} \\ [\![\mathsf{Perm}]\!] &= \mathit{Perm} \\ [\![\mathsf{Region}]\!] &= \mathit{RegionId} \end{aligned}$$

Definition 73 (Interpretation of Logical Variable Contexts)**.** We interpret program variable contexts as substitutions, valuations or partial maps to values of the appropriate type:

$$\begin{aligned} [\![\cdot]\!] &= \emptyset \\ [\![\Gamma, x : \tau]\!] &= \{\gamma[x \mapsto v] \mid \gamma \in [\![\Gamma]\!], v \in [\![\tau]\!]\} \end{aligned}$$

Definition 74 (Interpretation of Program Expressions)**.** We interpret program expressions as functions from interpretations of their contexts into *Val*.

$$[\![\Gamma \vdash M : \mathsf{Val}]\!] : [\![\Gamma]\!] \times \mathit{Stack} \to \mathit{Val}$$

$$\begin{aligned} [\![\Gamma \vdash x : \mathsf{Val}]\!]\,(\gamma, \sigma) &= \gamma(x) \\ [\![\Gamma \vdash \mathtt{x} : \mathsf{Val}]\!]\,(\gamma, \sigma) &= \sigma(\mathtt{x}) \\ [\![\Gamma \vdash n : \mathsf{Val}]\!]\,(\gamma, \sigma) &= n \\ [\![\Gamma \vdash M \text{ op } N : \mathsf{Val}]\!]\,(\gamma, \sigma) &= [\![\Gamma \vdash M : \mathsf{Val}]\!]\,(\gamma, \sigma) \text{ op } [\![\Gamma \vdash N : \mathsf{Val}]\!]\,(\gamma, \sigma) \\ [\![\Gamma \vdash M \prec N : \mathsf{Val}]\!]\,(\gamma, \sigma) &= \begin{cases} 1 & \text{if } [\![\Gamma \vdash M : \mathsf{Val}]\!]\,(\gamma, \sigma) \prec [\![\Gamma \vdash N : \mathsf{Val}]\!]\,(\gamma, \sigma) \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

Definition 75 (Interpretation of Permission Expressions)**.** We interpret permission expressions as functions from interpretations of their contexts into *Perm*.

$$[\![\Gamma \vdash M : \mathsf{Perm}]\!] : [\![\Gamma]\!] \to \mathit{Perm}$$

$$\begin{aligned}
[\![\Gamma \vdash x : \mathsf{Perm}]\!]\,(\gamma) &= \gamma(x)\\
[\![\Gamma \vdash 0p : \mathsf{Perm}]\!]\,(\gamma) &= \bot\\
[\![\Gamma \vdash 1p : \mathsf{Perm}]\!]\,(\gamma) &= \top\\
[\![\Gamma \vdash\sim M : \mathsf{Perm}]\!]\,(\gamma) &= ([\![\Gamma \vdash M : \mathsf{Perm}]\!]\,(\gamma))^{C}\\
[\![\Gamma \vdash M \cdot N : \mathsf{Perm}]\!]\,(\gamma) &= [\![\Gamma \vdash M : \mathsf{Perm}]\!]\,(\gamma) \wedge [\![\Gamma \vdash N : \mathsf{Perm}]\!]\,(\gamma)
\end{aligned}$$

Definition 76 (Interpretation of Region Expressions)**.** We interpret region expressions as functions from interpretations of their contexts to region identifiers:

$$[\![\Gamma \vdash M : \mathsf{Region}]\!] : [\![\Gamma]\!] \to \mathit{RegionId}$$

$$[\![\Gamma \vdash x : \mathsf{Region}]\!]\,(\gamma) = \gamma(x)$$

Definition 77 (Interpretation of Pure Assertions)**.** We interpret pure assertions as functions from interpretations of their contexts into 2. The codomain 2 is a complete boolean algebra, so we omit the usual, pointwise definitions of the standard logical connectives.

$$[\![\Gamma \vdash M : \mathsf{Pure}]\!] : [\![\Gamma]\!] \times \mathit{Stack} \to 2$$

$$\begin{aligned}
&[\![\Gamma \vdash M =_\tau N : \mathsf{Pure}]\!]\,(\gamma,\sigma) \iff\\
&\quad [\![\Gamma \vdash M : \tau]\!]\,(\gamma,\sigma) =_\tau [\![\Gamma \vdash N : \tau]\!]\,(\gamma,\sigma)
\end{aligned}$$

$$\begin{aligned}
&[\![\Gamma \vdash \forall x : \tau.M : \mathsf{Pure}]\!]\,(\gamma,\sigma) \iff\\
&\quad \forall v \in [\![\tau]\!]\,.\,[\![\Gamma, x : \tau; \sigma \vdash M : \mathsf{Pure}]\!]\,(\gamma[x \mapsto v],\sigma)
\end{aligned}$$

$$\begin{aligned}
&[\![\Gamma \vdash \exists x : \tau.M : \mathsf{Pure}]\!]\,(\gamma,\sigma) \iff\\
&\quad \exists v \in [\![\tau]\!]\,.\,[\![\Gamma, x : \tau; \sigma \vdash M : \mathsf{Pure}]\!]\,(\gamma[x \mapsto v],\sigma)
\end{aligned}$$

$$\begin{aligned}
&[\![\Gamma \vdash \mathtt{compatible}(M,N) : \mathsf{Pure}]\!]\,(\gamma,\sigma) \iff\\
&\quad [\![\Gamma \vdash M : \mathsf{Perm}]\!]\,(\gamma) \wedge [\![\Gamma \vdash N : \mathsf{Perm}]\!]\,(\gamma) \neq \bot
\end{aligned}$$

Definition 78 (Interpretation of Guard Expressions)**.** We interpret guard expressions as expressions in *Guard*, the free PCMZ over primitive guard symbols:

$$[\![\Gamma \vdash G : \mathsf{Guard}]\!] : [\![\Gamma]\!] \times \mathit{Stack} \to |\mathit{Guard}|$$

$$\begin{aligned}
[\![\Gamma \vdash G : \mathsf{Guard}]\!]\,(\gamma,\sigma) &= G\\
[\![\Gamma \vdash G[M] : \mathsf{Guard}]\!]\,(\gamma,\sigma) &= G[[\![\Gamma \vdash M : \mathsf{Perm}]\!]\,(\gamma)]\\
[\![\Gamma \vdash G(M) : \mathsf{Guard}]\!]\,(\gamma,\sigma) &= G([\![\Gamma \vdash M : \mathsf{Val}]\!]\,(\gamma,\sigma))\\
[\![\Gamma \vdash M * N : \mathsf{Guard}]\!]\,(\gamma,\sigma) &= [\![\Gamma \vdash M : \mathsf{Guard}]\!]\,(\gamma,\sigma) \cdot [\![\Gamma \vdash N : \mathsf{Guard}]\!]\,(\gamma,\sigma)
\end{aligned}$$

Definition 79 (Region Type Guard Algebra Assignment)**.** We denote the set of guard algebra assignments corresponding to a well-formed region type context R as $GA(R)$, and define it as follows:

$$GA(R) := \{\rho \mid \mathrm{dom}(\rho) = R\}$$

Definition 80 (Interpretation of Assertions)**.** We interpret assertions as functions from interpretations of their contexts into upwards closed subsets of heaps and region assignments. Upwards closed subsets of elements drawn from a partial commutative monoid has enough structure to support standard intuitionistic separation logic, so we omit the standard, pointwise interpretations for now.

$$\llbracket R; \Gamma \vdash M : \mathsf{Assn} \rrbracket : GA(R) \times \llbracket \Gamma \rrbracket \times \mathit{Stack} \rightarrow$$

$$\mathcal{P} \uparrow (\mathit{Heap} \times \mathit{RegionAssignment})$$

$$\llbracket R;\Gamma \vdash M : \mathsf{Assn} \rrbracket(\rho,\gamma,\sigma) = \begin{cases} \top & \text{if } \llbracket \Gamma \vdash M : \mathsf{Pure} \rrbracket(\gamma,\sigma) \\ \bot & \text{otherwise} \end{cases}$$

$$\llbracket R;\Gamma \vdash M \mathbin{?} N : O : \mathsf{Assn} \rrbracket(\rho,\gamma,\sigma) = \begin{cases} \llbracket R;\Gamma \vdash N : \mathsf{Assn} \rrbracket(\rho,\gamma,\sigma) & \text{if } \llbracket \Gamma \vdash M : \mathsf{Pure} \rrbracket(\gamma,\sigma) \\ \llbracket R;\Gamma \vdash O : \mathsf{Assn} \rrbracket(\rho,\gamma,\sigma) & \text{otherwise} \end{cases}$$

$$\begin{aligned}
&\llbracket R;\Gamma \vdash x@(M) : \mathsf{Assn} \rrbracket(\rho,\gamma,\sigma) = \\
&\quad \{(h,\alpha) \mid \forall r,G,f. \\
&\qquad \overline{f}(\llbracket \Gamma \vdash M : \mathsf{Guard} \rrbracket(\gamma,\sigma)) \sqsubseteq_G \overline{f}(\alpha(r).\mathrm{guards}) \\
&\qquad \wedge r = \llbracket \Gamma \vdash x : \mathsf{Region} \rrbracket(\gamma) \\
&\qquad \wedge (G,f) = \rho(\pi_1(\alpha(r).\mathrm{type}))\}
\end{aligned}$$

$$\begin{aligned}
&\left\llbracket R;\Gamma \vdash \mathbb{T}(x,\vec{M},N) : \mathsf{Assn} \right\rrbracket(\rho,\gamma,\sigma) = \\
&\quad \{(h,\alpha) \mid \forall r,s,\vec{v}.\ \exists \vec{x},\vec{\tau}. \\
&\qquad \alpha(r).\mathrm{type} = (\mathbb{T}(r,\vec{x}:\vec{\tau}),\vec{v}) \wedge \alpha(r).\mathrm{state} = s \\
&\qquad \wedge r = \llbracket \Gamma \vdash x : \mathsf{Region} \rrbracket(\gamma) \\
&\qquad \wedge v_i = \llbracket \Gamma \vdash M_i : \tau_i \rrbracket(\gamma,\sigma) \\
&\qquad \wedge s = \llbracket \Gamma \vdash N : \mathsf{Val} \rrbracket(\gamma,\sigma)\}
\end{aligned}$$

$$\begin{aligned}
&\llbracket R;\Gamma \vdash M \mapsto N : \mathsf{Assn} \rrbracket(\rho,\gamma,\sigma) = \{(h,\alpha) \mid x \in \mathit{Addr} \wedge h(x) = \llbracket \Gamma \vdash N : \mathsf{Val} \rrbracket(\gamma,\sigma) \\
&\quad \wedge x = \llbracket \Gamma \vdash M : \mathsf{Val} \rrbracket(\gamma,\sigma)\}
\end{aligned}$$

$$\begin{aligned}
&\llbracket R;\Gamma \vdash M \mapsto [N] : \mathsf{Assn} \rrbracket(\rho,\gamma,\sigma) = \{(h,\alpha) \mid \exists \vec{v}.\ x \in \mathit{Addr} \wedge n > 0 \wedge h(x,\ldots,x+(n-1)) = \vec{v} \\
&\quad \wedge x = \llbracket \Gamma \vdash M : \mathsf{Val} \rrbracket(\gamma,\sigma) \\
&\quad \wedge n = \llbracket \Gamma \vdash N : \mathsf{Val} \rrbracket(\gamma,\sigma)\}
\end{aligned}$$

$$\llbracket R;\Gamma \vdash M * N : \mathsf{Assn} \rrbracket(\rho,\gamma,\sigma) = \llbracket R;\Gamma \vdash M : \mathsf{Assn} \rrbracket(\rho,\gamma,\sigma) * \llbracket R;\Gamma \vdash N : \mathsf{Assn} \rrbracket(\rho,\gamma,\sigma)$$

$$\llbracket R;\Gamma \vdash \mathtt{emp} : \mathsf{Assn} \rrbracket(\rho,\gamma,\sigma) = \{(\mathtt{emp},\alpha) \mid \forall \alpha\}$$

$$\llbracket R;\Gamma \vdash M \wedge N : \mathsf{Assn} \rrbracket(\rho,\gamma,\sigma) = \llbracket R;\Gamma \vdash M : \mathsf{Assn} \rrbracket(\rho,\gamma,\sigma) \cap \llbracket R;\Gamma \vdash N : \mathsf{Assn} \rrbracket(\rho,\gamma,\sigma)$$

$$\llbracket R;\Gamma \vdash M \vee N : \mathsf{Assn} \rrbracket(\rho,\gamma,\sigma) = \llbracket R;\Gamma \vdash M : \mathsf{Assn} \rrbracket(\rho,\gamma,\sigma) \cup \llbracket R;\Gamma \vdash N : \mathsf{Assn} \rrbracket(\rho,\gamma,\sigma)$$

Convention 81. For particular ρ and γ we write $\llbracket R;\Gamma \vdash M : \mathsf{Assn} \rrbracket(\rho,\gamma)$ to denote the function

$$\lambda\sigma.\, \llbracket R;\Gamma \vdash M : \mathsf{Assn} \rrbracket(\rho,\gamma,\sigma)$$

Definition 82 (Interpretation of Entailments)**.** We interpret a well-typed derivation of entailment of assertions as functions from the interpretations of their environment into 2. The domain of interpretation of assertions is a complete BI-algebra, and entailment corresponds to subset inclusion.

$$\llbracket R;\Gamma \mid \Delta \vdash Q \rrbracket : \mathit{GA}(R) \times \llbracket \Gamma \rrbracket \times \mathit{Stack} \to 2$$

$$\begin{aligned}
&\llbracket R;\Gamma \mid \Delta \vdash Q \rrbracket(\rho,\gamma,\sigma) \iff \\
&\bigwedge_{P\in\Delta} \llbracket R;\Gamma \vdash P : \mathsf{Assn} \rrbracket(\rho,\gamma,\sigma) \subseteq \llbracket R;\Gamma \vdash Q : \mathsf{Assn} \rrbracket(\rho,\gamma,\sigma)
\end{aligned}$$

3.8.1 *Interpretation of Specifications*

We need some auxiliary definitions before we can define the interpretation of specifications.

Definition 83 (GADef$(-,-)$, Definedness of Guard w.r.t. Guard Algebra)**.**

$$\begin{aligned}&\mathrm{GADef} \subseteq \mathit{GuardAlgebra} \times \mathit{Guard}\\ &\mathrm{GADef}((G; f), gs) \overset{\text{def}}{\iff} \bar{f}(gs) \neq \bot_G\end{aligned}$$

Definition 84 (wf$(-,-,-)$, Well-Formedness of Region Assignments)**.**

$$\begin{aligned}&\mathrm{wf} \subseteq \mathit{GuardAlgebraAssignment} \times \mathit{RegionAssignment}\\ &\mathrm{wf}(\rho, a) \overset{\text{def}}{\iff} \forall r \in \mathrm{dom}(a), \mathrm{GADef}(\rho(\pi_1(a(r).\mathrm{type})), a(r).\mathrm{guards})\end{aligned}$$

Definition 85 (collapse$^{-}_{-}(-,-)$, Abstract Configuration Collapse)**.** Collapsing combines the information in open regions and the current heap into a single assertion. It uses a region type assignment to interpret regions and a set of regions that are not opened in order to collapse only the open regions.

$$\begin{aligned}&\mathrm{collapse}^{\rho}_{s} : (\mathit{Heap} \times \mathit{RegionAssignment}) \to \mathcal{P}(\mathit{Heap} \times \mathit{RegionAssignment})\\ &\mathrm{collapse}^{\rho}_{s}(h, a) = \{(h, a)\} * \circledast_{r \in (\mathrm{dom}(a) \setminus s)}\ \rho(r).\mathrm{int}(a(r).\mathrm{type})\end{aligned}$$

Definition 86 ($\lfloor - \rfloor^{-}_{-}$, Erasure)**.** We define a notion of erasure - the concrete heaps that are all described by the same abstract configuration. It is defined with respect to a region type assignment and a collection of region as that are not to be collapsed:

$$\begin{aligned}\lfloor - \rfloor^{-}_{-} :&(\mathit{Heap} \times \mathit{RegionAssignment})\\ &\times \mathit{RegionTypeAssignment}\\ &\times \mathcal{P}(\mathit{RegionId})\\ &\to \mathcal{P}(\mathit{Heap})\\ \lfloor (h, a) \rfloor^{\rho}_{s} :=&\{h' \mid (h', a') \in \mathrm{collapse}^{\rho}_{s}(h, a) \wedge \mathrm{wf}(GA(\rho), a')\}\end{aligned}$$

Definition 87 (Interpretation of Guard Algebra Declaration)**.** We interpret a guard algebra declaration as a guard algebra: a PCMZ M and a function $f : |Guard| \to |M|$, as follows:

$$\begin{aligned}
[\![0]\!] &= (2, \{\}) \quad \text{(the empty function)} \\
[\![G]\!] &= (3, \{G \mapsto 1\}) \\
[\![\%G]\!] &= (Perm, \{G[\pi] \mapsto \pi\}) \\
[\![\#G]\!] &= (3^{\mathsf{Val}}, \{G(v) \mapsto [v \mapsto 1\}) \\
[\![X * Y]\!] &= (M \times N, (i \circ f) \cup (j \circ g)) \\
&\quad \text{where} \\
&\quad\quad i : M \to M \times N \\
&\quad\quad i(m) = (m, \epsilon_N) \\
&\quad\quad j : N \to M \times N \\
&\quad\quad j(n) = (\epsilon_M, n) \\
&\quad\quad (M, f) = [\![X]\!] \\
&\quad\quad (N, g) = [\![Y]\!] \\
&\quad\quad \text{and } f \text{ and } g \text{ have disjoint domains} \\
[\![X + Y]\!] &= (M + N, [f, 0] \cup [g, 1]) \\
&\quad \text{where} \\
&\quad\quad (M, f) = [\![X]\!] \\
&\quad\quad (N, g) = [\![Y]\!] \\
&\quad\quad \text{and } f \text{ and } g \text{ have disjoint domains}
\end{aligned}$$

We can extend $f : |Guard| \to |M|$ to a total function by mapping everything else to zero. From this, we have a PCMZ homomorphism $\bar{f} : Guard \to M$.

Moreover, if generated from the above constructions, it will be surjective, and so $\mathrm{im}(\bar{f}) \cong M$ (as PCMZs). $\bar{f}$ defines an equivalence relation $\ker(\bar{f})$ on *Guard*, with $Guard/\ker(\bar{f}) \cong \mathrm{im}(\bar{f})$.

Definition 88 (Interpretation of Region Interpretation Declarations)**.** We interpret a region interpretation declaration, well-formed with respect to a set of region types R, identifier r and variables $\vec{x} : \vec{\tau}$ as a function from interpretations of these contexts, to region interpretations:

$$GA(R) \times [\![r : \mathsf{Region}, \vec{x} : \vec{\tau}]\!] \to RegionInterpretation$$

Given a guard algebra assignment $\rho \in GA(R)$, a substitution $\gamma \in [\![r : \mathsf{Region}, \vec{x} : \vec{\tau}]\!]$, for each clause $(\Delta).\Pi \mid e : P$ we *conditionally* extend the resulting map from *AbstractState* to *Assertion* by the following:

$$v \mapsto [\![R; r : \mathsf{Region}, \vec{x} : \vec{\tau}, \Delta \vdash P : \mathsf{Assn}]\!] \, (\rho, \gamma \cdot \delta, \sigma)$$

if there is a $\delta \in [\![\Delta]\!]$ such that

$$[\![R; r : \mathsf{Region}, \vec{x} : \vec{\tau}, \Delta \vdash \Pi : \mathsf{Pure}]\!] \, (\gamma \cdot \delta, \sigma)$$

and $v = \llbracket \Gamma \vdash e : \mathsf{Val} \rrbracket (\gamma \cdot \delta, \sigma)$ hold, for any σ at all, as none of these assertions can mention program variables.

Since no clause overlap, there will be at most one clause for which the condition is satisfied, hence this is a well-defined partial map.

Definition 89 (Interpretation of Action Declarations)**.** We can region interpret an action declaration, well-formed with respect to a set of region types R, identifier r and variables $\vec{x} : \vec{\tau}$, as describing an *LTS* as function of interpretations of its context:

$$GA(R) \times \llbracket r : \mathsf{Region}, \vec{x} : \vec{\tau} \rrbracket \to LTS$$

Given a guard algebra assignment $\rho \in GA(R)$, a substitution $\gamma \in \llbracket r : \mathsf{Region}, \vec{x} : \vec{\tau} \rrbracket$, for each clause $(\Delta).P \mid G : e_1 \rightsquigarrow e_2$ we extend the mapping as follows:

$$F \mapsto \{(\llbracket \Gamma \vdash e_1 : \mathsf{Val} \rrbracket (\gamma \cdot \delta, \sigma), \llbracket \Gamma \vdash e_2 : \mathsf{Val} \rrbracket (\gamma \cdot \delta, \sigma)) \mid$$

$$\llbracket \Gamma \vdash G : \mathsf{Guard} \rrbracket (\rho, \gamma \cdot \delta) = F\}^*$$

if there exists a $\delta \in \llbracket \Delta \rrbracket$ such that

$$\llbracket R; r : \mathsf{Region}, \vec{x} : \vec{\tau}, \Delta \vdash P : \mathsf{Assn} \rrbracket (\rho, \gamma \cdot \delta, \sigma)$$

for any stack σ at all, where * indicates the reflexive, transitive closure of the set.

Definition 90 (Guard Compatibility Complement)**.** We can define the "compatible complement" to a collection of guards $M \subseteq |Guard|$ with respect to a particular guard algebra G as follows:

$$\overline{M} = \{g' \in |Guard| \mid \forall g \in M, GADef(G, g \cdot g')\}$$

Definition 91 (Rely$(-)$, The Rely Relation)**.** We can describe the interference allowed by other threads by help of a region type assignment ρ, considering all the transitions allowed by guards compatible with ours:

$$\mathrm{Rely}(\rho) \subseteq (Heap \times RegionAssignment) \times (Heap \times RegionAssignment)$$

We say two abstract configurations (h_1, a_1) and (h_2, a_2) are related iff

- $h_1 = h_2$
- $\mathrm{dom}(\alpha_1) \subseteq \mathrm{dom}(\alpha_2)$
- $\forall r \in \mathrm{dom}(a_1).a_1(r).\mathrm{guards} = a_2(r).\mathrm{guards}$
- $\forall r \in \mathrm{dom}(a_1), (a_1(r).\mathrm{state}, a_2(r).\mathrm{state}) \in \rho(a_1(r).\mathrm{type}).\mathrm{lts}(\overline{a_1(r).\mathrm{guards}})$, where the guard compatibility complement is with respect to the guard algebra $\rho(a_1(r).\mathrm{type}).\mathrm{GA}$

Definition 92 (Interpretation of Region Contexts)**.** We interpret a well-formed region context R into a region type assignment:

$$[\![R]\!] : \mathit{RegionTypeAssignment}$$

For each associated $\mathbb{T}(r, \vec{x} : \vec{\tau})(g, i, a)$, let $[\![g]\!]$ be the guard algebra corresponding to g. Then, let $[\![i]\!]\,([\![g]\!], \vec{v})$ and $[\![a]\!]\,([\![g]\!], \vec{v})$ be the region interpretation and LTS corresponding to i and a, respectively, under valuation $\vec{v}$, such that $v_i \in [\![\tau_i]\!]$, of the region parameters. Then we extend the resulting map as follows:

$$(\mathbb{T}(r, \vec{x} : \vec{\tau}), \vec{v}) \mapsto ([\![i]\!]\,([\![g]\!], \vec{v}), [\![a]\!]\,([\![g]\!], \vec{v}), [\![g]\!])$$

This definition is well-formed as a map as each region type is bound once in the region context.

Definition 93 (Stabilization)**.** We define the stabilization of an assertion P as intuitively the strongest weaker stable assertion:

$$\text{stabilize}_\rho(P) := \{q \mid \exists p \in P.\ (p, q) \in \text{Rely}(\rho)\}$$

We remark that by reflexivity of the rely relation, it is always the case that $P \leq \text{stabilize}_\rho(P)$.

Definition 94 ($\text{stable}_{-}(-)$, Assertion Stability)**.** An assertion P is stable with respect to region type assignment ρ when

$$\text{stabilize}_\rho(P) \leq P$$

Hence, stable assertions are equivalent to their stabilizations.

Definition 95 ($- \Vdash_{-} \{-\}\{-\}$, Semantic Action Judgment)**.** We define $\alpha \Vdash_\rho \{P\}\{Q\}$ to hold if and only if, for all $R \in \mathit{Assertion}$ such that $\text{stable}(R)$ we have $[\![\alpha]\!]\,(\lfloor P * R \rfloor^\rho_\emptyset) \subseteq \lfloor Q * R \rfloor^\rho_\emptyset$.

Lemma 96 (Locality of Action Judgment)**.** For all stable assertions R,

$$\alpha \Vdash_\rho \{P\}\{Q\} \Rightarrow \alpha \Vdash_\rho \{P * R\}\{Q * R\}$$

Proof. Chose the frame in the assumption to be $R * R'$ where R' is the given frame in the conclusion of the Lemma. Stable assertions are closed under $*$. □

Definition 97 ($\preceq$, Semantic Entailment)**.**

$$P \preceq Q \overset{\text{def}}{\iff} \texttt{id} \Vdash_\rho \{P\}\{Q\}$$

Lemma 98. Assertions ordered by semantic entailment forms a partial order.

Proof. The action of the identity action on sets of heap is the identity. Hence, assertions under semantic entailment forms a partial order because sets ordered by set inclusion does. □

Lemma 99 (Frame Property of Semantic Entailment)**.**

$$P \preceq Q \Rightarrow (\forall R.\mathrm{stable}(R) \Rightarrow P * R \preceq Q * R)$$

Proof. Directly from Lemma 96. □

Lemma 100 (Entailment is Semantic Entailment)**.**

$$R;\Gamma \mid \Delta \vdash Q \Rightarrow \forall\rho \in [\![R]\!], \forall\gamma \in [\![\Gamma]\!], \forall\sigma.$$
$$\bigwedge_{P\in\Delta} [\![R;\Gamma \vdash P : \mathsf{Assn}]\!](\rho,\gamma,\sigma) \preceq [\![R;\Gamma \vdash Q : \mathsf{Assn}]\!](\rho,\gamma,\sigma)$$

Lemma 101 ($\preceq$-Closure of Action Judgment)**.** If

$$P \preceq P' \qquad \alpha \Vdash_\rho \{P'\}\{Q'\} \qquad Q' \preceq Q$$

then

$$\alpha \Vdash_\rho \{P\}\{Q\}$$

Proof. We proceed by direct proof. Suppose a stable frame R and suppose a configuration (h,a) in the assertion $P * R$. By the definition of assertions, that means we can split (h,a) into $(h_P,a_P) \in P$ and $(h_R,a_R) \in R$. By $P \preceq P'$, we get that $(h_P,a_P) \in P'$. Hence, by chosing the frame R in the assumption, we get that $[\![\alpha]\!]((h_P,a_P)) \in Q'$, and again, in Q, precisely as desired. □

Definition 102 (safe, Execution Safety)**.** We define safe to be a recursively defined relation on

$$\mathit{RegionTypeAssignment} \times \mathbb{N} \times \mathit{Env} \times \mathit{Assertion} \times \mathit{Stack} \times$$
$$\mathit{Cont} \times (\mathit{Stack} \to \mathit{Assertion}) \times (\mathit{Stack} \to \mathit{Val} \to \mathit{Assertion}),$$

with $\mathrm{safe}_n^\rho(E,P,\sigma,\kappa,Q,U)$ intuitively capturing that execution of κ in stack σ satisfying P will run and, within n steps, safely halt execution in a stack satisfying Q or return a value v satisfying $U(v)$.

We say $\mathrm{safe}_0(E,P,\sigma,\kappa,Q,U)$ always holds, and $\mathrm{safe}_{n+1}(E,P,\sigma,\kappa,Q,U)$ holds iff ...

1. ... when $\kappa = \texttt{skip}$ then
   a) $\texttt{id} \Vdash_\rho \{P\}\{Q(\sigma)\}$
2. ... when $\kappa = \texttt{return e}$ or $\kappa = \texttt{return e;s}$, for some `e` and `s`, then
   a) $\texttt{id} \Vdash_\rho \{P\}\{U(\sigma)([\![\texttt{e}]\!]_\sigma)\}$
3. ... when there is a forking step $E \vdash (\sigma,\kappa) \xrightarrow{\texttt{fork}(\texttt{f},\vec{v})} (\sigma',\kappa')$ for some `f` and $\vec{v}$, there exists $P', F : \mathit{Assertion}$ and $(\vec{\texttt{x}},\texttt{s})$ such that,
   a) $E(\texttt{f}) = (\vec{\texttt{x}},\texttt{s})$
   b) $\texttt{id} \Vdash_\rho \{P\}\{P' * F\}$

c) $\mathsf{safe}^{\rho}_{n}(E, P', \sigma', \kappa', Q, U)$

d) $\mathsf{safe}^{\rho}_{n}(E, F, [\vec{\mathtt{x}} \mapsto \vec{v}], \mathtt{s}, \lambda\_.\top, \lambda\_.\lambda\_.\top)$

4. ... when there is a non-forking step $E \vdash (\sigma, \kappa) \xrightarrow{\alpha} (\sigma', \kappa')$ there is a P' : *Assertion* such that

   a) $\alpha \Vdash_{\rho} \{P\}\{P'\}$

   b) $\mathsf{safe}^{\rho}_{n}(E, P', \sigma', \kappa', Q, U)$

We omit the super-script ρ when it is clear from context. At any one time there will only be a single such region type assignment in play.

Definition 103 (Environment/Function Spec Agreement).

$- \vDash^{-}_{-} - : - \subseteq \mathit{Env} \times \mathit{RegionTypeAssignment} \times \mathbb{N} \times \mathsf{Fun} \times \mathsf{FunctionSpec}$

An environment implements a function specification of $\mathtt{f}$ for n steps, noted

$$E \vDash^{\rho}_{n} \mathtt{f} : (\Gamma, \vec{y})\{P\}\{r.Q\}$$

if and only if there exists $\vec{\mathtt{x}} \in \mathsf{Var}^{*}$ and $\mathtt{s} \in \mathsf{Stmt}$ such that $E(\mathtt{f}) = (\vec{\mathtt{x}}, \mathtt{s})$ and, for all $\gamma \in [\![\Gamma, \vec{y} : \mathsf{Val}]\!]$, $\sigma \in \mathit{Stack}$,

$$\begin{aligned}\mathsf{safe}^{\rho}_{n}(E, &[\![R; \Gamma, \vec{y} : \mathsf{Val} \vdash P * (\vec{\mathtt{x}} = \vec{y}) : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma),\\ &\sigma,\\ &\mathtt{s},\\ &\lambda\sigma.\,[\![R; \Gamma, \vec{y} : \mathsf{Val} \vdash \forall r.Q : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma),\\ &\lambda\sigma.\lambda v.\,[\![R; \Gamma, \vec{y} : \mathsf{Val}, r : \mathsf{Val} \vdash Q : \mathsf{Assn}]\!]\,(\rho, \gamma[r \mapsto v], \sigma))\end{aligned}$$

Definition 104 (Environment/Specification Context Agreement).

$$- \vDash^{-}_{-} - \subseteq \mathit{Env} \times \mathit{RegionTypeAssignment} \times \mathbb{N} \times \mathit{SpecCtxt}$$

An environment E implements a specification context Φ for n steps, noted $E \vDash^{\rho}_{n} \Phi$ when, for each individual $\mathtt{f} \in \mathrm{dom}(\Phi)$, $E \vDash^{\rho}_{n} \mathtt{f} : \Phi(\mathtt{f})$.

As with the safe predicate, we elide the region type assignment when clear from the context.

Definition 105 (Interpretation of "Triples"). We define the interpretation of a Hoare "triple" as a function

$$\left[\!\!\left[R; \Phi; \Gamma \vdash \left\{P\right\} \; \mathtt{s} \; \left\{Q \mid r.U\right\} : \mathsf{Spec} \right]\!\!\right] : [\![R]\!] \times [\![\Gamma]\!] \to \mathcal{P}\downarrow(\mathbb{N})$$

defined as follows:

$$\begin{aligned}&\left[\!\!\left[R; \Phi; \Gamma \vdash \left\{P\right\} \; \mathtt{s} \; \left\{Q \mid r.U\right\} : \mathsf{Spec} \right]\!\!\right] (\rho, \gamma) =\\ &\quad \{n \in \mathbb{N} \mid \forall E : \mathit{Env}.(\forall n' < n.E \vDash^{\rho}_{n'} \Phi) \Rightarrow \forall \sigma : \mathit{Stack}.\\ &\qquad \mathsf{safe}^{\rho}_{n}(E,\\ &\qquad\quad [\![\mathrm{dom}(R); \Gamma \vdash P : \mathsf{Assn}]\!]\,(GA(\rho), \gamma, \sigma),\\ &\qquad\quad \sigma,\\ &\qquad\quad \mathtt{s},\\ &\qquad\quad \lambda\sigma.\,[\![\mathrm{dom}(R); \Gamma \vdash Q : \mathsf{Assn}]\!]\,(GA(\rho), \gamma, \sigma),\\ &\qquad\quad \lambda\sigma.\lambda v.\,[\![\mathrm{dom}(R); \Gamma, r : \mathsf{Val} \vdash U : \mathsf{Assn}]\!]\,(GA(\rho), \gamma[r \mapsto v], \sigma))\}\end{aligned}$$

Lemma 106 (Interpretation of Triples is Well-Defined)**.** For all $\rho \in [\![R]\!]$, $\gamma \in [\![\Gamma]\!]$,

$$\left[\!\!\left[R; \Phi; \Gamma \vdash \left\{P\right\} \ \mathtt{s} \ \left\{Q \mid r.U\right\} : \mathsf{Spec}\right]\!\!\right] (\rho, \gamma)$$

is downwards closed.

Definition 107 (Interpretation of Atomic Triples)**.** We define the interpretation of atomic triples as a function of well-typed triples as follows:

$$[\![R; \Gamma \vdash_S \langle P \rangle \ \mathtt{s} \ \langle Q \rangle : \mathsf{Atomic}]\!] : [\![R]\!] \times [\![\Gamma]\!] \times \mathit{Stack} \rightarrow 2$$

where $[\![R; \Gamma \vdash_S \langle P \rangle \ \mathtt{s} \ \langle Q \rangle]\!] (\rho, \gamma, \sigma)$ holds if and only if there is α, σ' such that

1. $E \vdash (\sigma, \mathtt{s}) \xrightarrow{\alpha} (\sigma', \mathtt{skip})$
2. For all $\vec{r} \in [\![S]\!]$ and stable assertions R,

$$[\![\alpha]\!] \left(\lfloor [\![R; \Gamma \vdash P : \mathsf{Assn}]\!] (\rho, \gamma, \sigma) * R \rfloor^{\rho}_{\vec{r}} \right)$$
$$\subseteq$$
$$\lfloor [\![R; \Gamma \vdash Q : \mathsf{Assn}]\!] (\rho, \gamma, \sigma') * R \rfloor^{\rho}_{\vec{r}}$$

Definition 108 (Interpretation of Open Region Judgment)**.** We define the interpretation of open region judgments as a function of well-typed judgments as follows:

$$[\![R; \Gamma \vdash [P] \text{ open } [\{(\Delta_i).(Q_i, \vec{r}_i)\}_{i \in I}]]\!] : [\![R]\!] \times [\![\Gamma]\!] \times \mathit{Stack} \rightarrow 2$$

where $[\![R; \Gamma \vdash [P] \text{ open } [\{(\Delta_i).(Q_i, \vec{r}_i)\}_{i \in I}]]\!] (\rho, \gamma, \sigma)$ holds if and only if for all $p \in [\![P]\!]$ and frame r, there is an index $i \in I$ with $\delta \in [\![\Delta_i]\!]$ and $q \in [\![Q_i]\!] (\gamma\delta)$ with $\vec{y} \in [\![\vec{r}_i]\!] (\gamma\delta)$ such that there is an updated frame r' with

1. $(r, r') \in \text{Rely}(\rho)$
2. $\lfloor p \cdot r \rfloor^{\rho}_{\emptyset} \subseteq \lfloor q \cdot r' \rfloor^{\rho}_{\vec{y}}$

Definition 109 (Interpretation of Close Region Judgment)**.** We define the interpretation of close region judgments as a function of well-typed judgments as follows:

$$[\![\Gamma \vdash [P] \text{ close}(\vec{r}) \ [Q]]\!] : [\![R]\!] \times [\![\Gamma]\!] \times \mathit{Stack} \rightarrow 2$$

where $[\![\Gamma \vdash [P] \text{ close}(\vec{r}) \ [Q]]\!] (\rho, \gamma, \sigma)$ holds if and only if, for all $p \in [\![R; \Gamma \vdash P : \mathsf{Assn}]\!] (\rho, \gamma, \sigma)$ and any choice of frame r, there is a $q \in [\![R; \Gamma \vdash Q : \mathsf{Assn}]\!] (\rho, \gamma, \sigma)$ and an updated frame r' such that

1. $(r, r') \in \text{Rely}(\rho)$

2. $\lfloor p \cdot r \rfloor^{\rho}_{\vec{r}} \subseteq \lfloor q \cdot r' \rfloor^{\rho}_{\emptyset}$

Definition 110 (Interpretation of Function Specifications)**.** We interpret well-typed function specifications as functions from interpretations of a function context into down-closed sets of natural numbers

$$[\![R;\Phi \vdash \mathtt{f}(\vec{\mathtt{x}})(\Gamma,\vec{y})\{P\}\mathtt{s}\{r.Q\} : \mathsf{FunctionSpec}]\!] : [\![R]\!] \to \mathcal{P}\downarrow(\mathbb{N})$$

$$\begin{aligned}&[\![R;\Phi \vdash \mathtt{f}(\vec{\mathtt{x}})(\Gamma,\vec{y})\{P\}\mathtt{s}\{r.Q\} : \mathsf{FunctionSpec}]\!]\,(\rho) = \\ &\bigcap_{\gamma\in[\![\Gamma,\vec{y}:\mathsf{Val}]\!]} \left[\!\!\left[R;\Phi;\Gamma,\vec{y}:\mathsf{Val} \vdash \left\{P * (\vec{\mathtt{x}} = \vec{y})\right\}\ \mathtt{s}\ \left\{\forall r.Q \mid r.Q\right\} : \mathsf{Spec}\right]\!\!\right](\rho,\gamma)\end{aligned}$$

Definition 111 (Interpretation of Program Specifications)**.** We interpret well-typed program specifications into 2:

$$[\![\vdash \vec{r};\vec{\mathtt{f}} : \mathsf{ProgramSpec}]\!] \iff \forall n \in \mathbb{N}.\ \lfloor \vec{\mathtt{f}} \rfloor \vDash^{[\![\vec{r}]\!]}_{n} \lceil \vec{\mathtt{f}} \rceil$$

3.9 Soundness

3.9.1 *Soundness of Specification Logic*

3.9.1.1 *Soundness of Statement Specifications*

Lemma 112 (Required for "Soundness of Function Call")**.** Suppose P : *Assertion* and Q : Term that does not mention program variables. If

$$\begin{aligned}&\mathrm{safe}_n(E,P,\sigma',\mathtt{s},\\ &\qquad [\![\mathrm{dom}(R);\Gamma \vdash \forall r:\mathsf{Val}.Q : \mathsf{Assn}]\!]\,(\rho,\gamma),\\ &\qquad \lambda v.\,[\![R;\Gamma,r:\mathsf{Val} \vdash Q : \mathsf{Assn}]\!]\,(\rho,\gamma[r\mapsto v]))\end{aligned}$$

then, for any stack σ, program variable $\mathtt{x}$ and U : *Stack* $\to$ *Val* $\to$ *Assertion*:

$$\mathrm{safe}_n(E,P,\sigma,\mathtt{x}{:=}(\sigma',\mathtt{s}),[\![\mathrm{dom}(R);\Gamma \vdash \exists r:\mathsf{Val}.\mathtt{x} = r * Q : \mathsf{Assn}]\!]\,(\rho,\gamma),U).$$

Proof. Proceed by Induction on n; assume i.e. the lemma holds at all $n' < n$.

Suppose the antecedent along with a σ, $\mathtt{x}$ and U. We now have to show.

$$\mathrm{safe}_n(E,P,\sigma,\mathtt{x}{:=}(\sigma',\mathtt{s}),[\![\mathrm{dom}(R);\Gamma \vdash \exists r:\mathsf{Val}.\mathtt{x} = r * Q : \mathsf{Assn}]\!]\,(\gamma),U).$$

To show safe is to show the code in question safe when it halts, returns and steps. We know that the code $\mathtt{x}{:=}(\sigma',\mathtt{s})$ neither immediately returns or halts, it must take a (possibly forking) execution step. By the operational semantics, which step depends on the shape of the code of the running function, s. By case analysis of the operational semantics, there are 4 cases, and we handle each in sequence.

Halt Suppose $\mathtt{s} = \mathtt{skip}$. By assumption on $\mathtt{s}$, we then know that

$$P \preceq [\![\mathrm{dom}(R); \Gamma \vdash \forall r.Q : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma')$$

We can now step

$$\frac{}{E \vdash (\sigma, \mathtt{x}{:=}(\sigma', \mathtt{skip})) \xrightarrow{\mathtt{id}} (\sigma[\mathtt{x} \mapsto v], \mathtt{skip})}$$

for some v and thus have to find a P' : *Assertion* such that

(a) $P \preceq P'$

(b) $\mathsf{safe}_{n-1}(E, P', \sigma[\mathtt{x} \mapsto v], \mathtt{skip}, [\![\mathrm{dom}(R); \Gamma \vdash \exists r : \mathsf{Val}.\mathtt{x} = r * Q : \mathsf{Assn}]\!]\,(\rho, \gamma))$

Pick $P' := [\![\mathrm{dom}(R); \Gamma \vdash \exists r : \mathsf{Val}.\mathtt{x} = r * Q : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma[\mathtt{x} \mapsto v])$, and (b) is immediate. To show (a) is to show:

$$P \preceq [\![\mathrm{dom}(R); \Gamma \vdash \exists r : \mathsf{Val}.\mathtt{x} = r * Q : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma[\mathtt{x} \mapsto v])$$

By transitivity of $\preceq$ it suffices to show that

$$[\![\mathrm{dom}(R); \Gamma \vdash \forall r.Q : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma') \preceq$$

$$[\![\mathrm{dom}(R); \Gamma \vdash \exists r : \mathsf{Val}.\mathtt{x} = r * Q : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma[\mathtt{x} \mapsto v])$$

Since Q does not mention program variables,

$$[\![\mathrm{dom}(R); \Gamma \vdash \forall r.Q : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma') = [\![\mathrm{dom}(R); \Gamma \vdash \forall r.Q : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma[\mathtt{x} \mapsto v])$$

and thus, by Lemma 100 it suffices to show that

$$R; \Gamma \mid \forall r.Q \vdash \exists r.\mathtt{x} = r * Q,$$

which is a simple derivation:

$$\cfrac{\cfrac{\cfrac{}{R; \Gamma \mid \forall r.Q \vdash \mathtt{x} = \mathtt{x}} \qquad \cfrac{}{R; \Gamma \mid \forall r.Q \vdash Q}}{R; \Gamma \mid \forall r.Q \vdash \mathtt{x} = \mathtt{x} * Q}}{R; \Gamma \mid \forall r.Q \vdash \exists r.\mathtt{x} = r * Q}$$

Return This case is completely analogous to the previous, except we now have a specific value rather than v; observe the previous case made no explicit use of v.

Non-forking step If $\mathtt{s}$ is such that

$$E \vdash (\sigma', \mathtt{s}) \xrightarrow{\alpha} (\sigma'', \mathtt{s}')$$

then we know from the safety of $\mathtt{s}$ that there is a P' such that $P \preceq P'$ and

$$\begin{aligned}\mathsf{safe}_{n-1}(E,\ & \\ & P',\\ & \sigma'',\\ & \mathtt{s}',\\ & [\![\mathrm{dom}(R); \Gamma \vdash \forall r.Q : \mathsf{Assn}]\!]\,(\rho, \gamma),\\ & \lambda v.\, [\![R; \Gamma, r : \mathsf{Val} \vdash Q : \mathsf{Assn}]\!]\,(\rho, \gamma[r \mapsto v]))\end{aligned}$$

holds. Further, we know by the operational semantics that

$$E \vdash (\sigma, \mathtt{x}{:=}(\sigma', \mathtt{s})) \xrightarrow{\alpha} (\sigma, \mathtt{x}{:=}(\sigma'', \mathtt{s}'))$$

and thus we need to find a P' such that

(a) $P \preceq P'$

(b) $\mathsf{safe}_{n-1}(E, P', \sigma, \mathtt{x}{:=}(\sigma', \mathtt{s}'), [\![\exists r : \mathsf{Val}.\mathtt{x} = r * Q]\!]\,(\gamma))$

We choose the P' given by assumption, and (a) is immediate.

(b) follows by induction hypothesis, with the necessary premise given by the assumption on $\mathtt{s}$.

Forking step The case of forking steps is analogous to non-forking steps, and hinges on the same observations on the assumption that $\mathtt{s}$ is safe and the hypothesis. □

Lemma 113 (Argument-Parameter Substitution). For any P that does not contain program variables, σ and σ' such that $\sigma'(\vec{\mathtt{x}}) = [\![\vec{\mathtt{e}}]\!]_\sigma$ it's the case that

$$[\![R; \Gamma \vdash P * (\vec{\mathtt{e}} = \vec{y}) : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma) \preceq$$

$$[\![R; \Gamma \vdash P * (\vec{\mathtt{x}} = \vec{y}) : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma')$$

Proof. First, we observe that by the interpretation of assertions, we can compute on the left hand side:

$$\begin{aligned}&[\![R; \Gamma, \vec{y} : \mathsf{Val} \vdash P * (\vec{\mathtt{e}} = \vec{y}) : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma)\\ =\ &[\![R; \Gamma, \vec{y} : \mathsf{Val} \vdash P : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma) * [\![R; \Gamma, \vec{y} : \mathsf{Val} \vdash (\vec{\mathtt{e}} = \vec{y}) : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma)\end{aligned}$$

and then the right:

$$\begin{aligned}&[\![R; \Gamma, \vec{y} : \mathsf{Val} \vdash P * (\vec{\mathtt{x}} = \vec{y}) : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma')\\ =\ &[\![R; \Gamma, \vec{y} : \mathsf{Val} \vdash P : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma') * [\![R; \Gamma, \vec{y} : \mathsf{Val} \vdash (\vec{\mathtt{x}} = \vec{y}) : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma')\end{aligned}$$

Observing that

$$[\![R; \Gamma, \vec{y} : \mathsf{Val} \vdash P : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma) = [\![R; \Gamma, \vec{y} : \mathsf{Val} \vdash P : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma')$$

since P does not refer to program variables, we now have an entailment of the shape $R * P \preceq R * Q$, so by Lemma 99 it suffices to show that:

$$[\![R; \Gamma, \vec{y} : \mathsf{Val} \vdash (\vec{\mathtt{e}} = \vec{y}) : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma) \preceq [\![R; \Gamma, \vec{y} : \mathsf{Val} \vdash (\vec{\mathtt{x}} = \vec{y}) : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma')$$

By another appeal to Lemma 99 it suffices to show that

$$[\![R; \Gamma, \vec{y} : \mathsf{Val} \vdash \mathtt{e}_i = y_i : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma) \preceq$$

$$[\![R; \Gamma, \vec{y} : \mathsf{Val} \vdash \mathtt{x}_i = y_i : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma')$$

for each i. This follows easily by computation, which reveals that this indeed holds since $\preceq$ is reflexive:

$$\begin{aligned} &\llbracket R;\Gamma,\vec{y}:\mathsf{Val}\vdash \mathsf{e}_i = y_i : \mathsf{Assn}\rrbracket\,(\rho,\gamma,\sigma)\\ =\ &\llbracket \mathsf{e}_i\rrbracket_\sigma =_{\mathsf{Val}} \gamma(y_i)\\ =\ &\llbracket \mathtt{x}_i\rrbracket_{\sigma'} =_{\mathsf{Val}} \gamma(y_i)\\ =\ &\llbracket R;\Gamma,\vec{y}:\mathsf{Val}\vdash \mathtt{x}_i = y_i : \mathsf{Assn}\rrbracket\,(\rho,\gamma,\sigma') \end{aligned}$$

and we are done. □

Lemma 114 (Soundness of Function Call)**.** Suppose that

$$\Phi(\mathtt{f}) = (\Gamma,\vec{y})\{P\}\{r.Q\}.$$

Then for any $n\in\mathbb{N}$, $\rho\in\llbracket R\rrbracket$ and $\gamma\in\llbracket\Gamma,\vec{y}:\mathsf{Val}\rrbracket$,

$$n\in$$

$$\left\llbracket R;\Phi;\Gamma,\vec{y}:\mathsf{Val}\vdash \left\{P*(\vec{\mathsf{e}}=\vec{y})\right\}\ \mathtt{x}{:=}\mathtt{f}(\vec{\mathsf{e}})\ \left\{\exists r:\mathsf{Val}.\mathtt{x}=r*Q \mid r.U\right\}\right\rrbracket(\rho,\gamma).$$

Proof. Suppose an environment E such that for all $n'<n$ we have $E\vDash^{\rho}_{n'}\Phi$. Suppose further a stack σ. We now need to show that

$$\begin{aligned} \mathsf{safe}_n(&E,\llbracket R;\Gamma,\vec{y}:\mathsf{Val}\vdash P*(\vec{\mathsf{e}}=\vec{y}) : \mathsf{Assn}\rrbracket\,(\rho,\gamma,\sigma),\\ &\sigma,\\ &\mathtt{x}{:=}\mathtt{f}(\vec{\mathsf{e}}),\\ &\llbracket R;\Gamma,\vec{y}:\mathsf{Val}\vdash \exists r.\mathtt{x}=r*Q : \mathsf{Assn}\rrbracket\,(\rho,\gamma),\\ &\lambda v.\,\llbracket R;\Gamma,\vec{y}:\mathsf{Val},r:\mathsf{Val}\vdash U : \mathsf{Assn}\rrbracket\,(\rho,\gamma[r\mapsto v])) \end{aligned}$$

We thus need to show Case 4 of Definition 102, where we take a regular execution step as the only applicable rule is

$$\frac{E(\mathtt{f})=(\vec{\mathtt{x}},\mathtt{s})\qquad \sigma'(\vec{\mathtt{x}})=\llbracket\vec{\mathsf{e}}\rrbracket_\sigma}{E\vdash(\sigma,\mathtt{x}{:=}\mathtt{f}(\vec{\mathsf{e}}))\xrightarrow{\mathtt{id}}(\sigma,\mathtt{x}{:=}(\sigma',\mathtt{s}))}$$

for some $\sigma' : \mathit{Stack}$ such that $\sigma'(\vec{\mathtt{x}})=\llbracket\vec{\mathsf{e}}\rrbracket_\sigma$.

Hence $\mathtt{s}'=\mathtt{x}{:=}(\sigma',\mathtt{s})$ and $\alpha=\mathtt{id}$. We now need to choose a suitable P', such that

(a) $\llbracket R;\Gamma,\vec{y}:\mathsf{Val}\vdash P*(\vec{\mathsf{e}}=\vec{y}) : \mathsf{Assn}\rrbracket\,(\rho,\gamma,\sigma)\preceq P'$.

(b) $\mathsf{safe}_{n-1}(E,P',\sigma,\mathtt{x}{:=}(\sigma',\mathtt{s})),\llbracket\exists r.\mathtt{x}=r*Q\rrbracket\,(\rho,\gamma),\lambda v.\,\llbracket U\rrbracket\,(\rho,\gamma[r\mapsto v]))$

We choose $P' := \llbracket R;\Gamma,\vec{y}:\mathsf{Val}\vdash P*(\vec{\mathtt{x}}=\vec{y}) : \mathsf{Assn}\rrbracket\,(\rho,\gamma,\sigma')$.

The statement (b) follows by appeal to Lemma 112, where the premise is obtained by instantiating $\forall n'<n.E\vDash\Phi$ at $n-1$, giving us that s is safe to run from $\llbracket R;\Gamma,\vec{y}:\mathsf{Val}\vdash P*(\vec{\mathtt{x}}=\vec{y}) : \mathsf{Assn}\rrbracket\,(\rho,\gamma,\sigma')$. (b) is then immediate from the lemma.

(a) follows by Lemma 113. □

Lemma 115 (Soundness of Value Return)**.** For all $n \in \mathbb{N}$, $\rho \in [\![R]\!]$, $\gamma \in [\![\Gamma]\!]$ and assertion Q,

$$n \in \left[\!\!\left[R; \Phi; \Gamma \vdash \left\{\top\right\} \texttt{return e} \left\{Q \mid r.\mathsf{e} = r\right\} : \mathsf{Spec}\right]\!\!\right] (\rho, \gamma)$$

Proof. Suppose an environment E such that for all $n' < n$, $E \vDash^{\rho}_{n'} \Phi$. Suppose furthermore a stack σ. We need to show that

$$\begin{aligned}&\mathsf{safe}_n(E, [\![\mathsf{dom}(R); \Gamma \vdash \top : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma)\\&\qquad \sigma,\\&\qquad \texttt{return e},\\&\qquad \lambda\sigma'.\,[\![\mathsf{dom}(R); \Gamma \vdash Q : \mathsf{Assn}]\!]\,(\gamma, \sigma')\\&\qquad \lambda\sigma'.\lambda v.\,[\![R; \Gamma, r : \mathsf{Val} \vdash \mathsf{e} = r : \mathsf{Assn}]\!]\,(\rho, \gamma[r \mapsto v], \sigma'))\end{aligned}$$

We thus need to show that

$$[\![\mathsf{dom}(R); \Gamma \vdash \top : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma) \preceq$$
$$[\![R; \Gamma, r : \mathsf{Val} \vdash \mathsf{e} = r : \mathsf{Assn}]\!]\,(\rho, \gamma[r \mapsto [\![\mathsf{e}]\!]_\sigma], \sigma)$$

which is immediate by calculation:

$$\begin{aligned}&[\![R; \Gamma, r : \mathsf{Val} \vdash \mathsf{e} = r : \mathsf{Assn}]\!]\,(\rho, \gamma[r \mapsto [\![\mathsf{e}]\!]_\sigma], \sigma)\\=\;& [\![\mathsf{e}]\!]_\sigma =_{\mathsf{Val}} [\![\mathsf{e}]\!]_\sigma\end{aligned}$$

which always holds. □

Lemma 116 (Soundness of If)**.** For all assertions P, Q, U, and for all $n \in \mathbb{N}$, $\gamma \in [\![\Gamma]\!]$ and $\rho \in [\![Rho]\!]$, if

1. $n \in \left[\!\!\left[R; \Phi; \Gamma \vdash \left\{P * \mathsf{e} \neq 0\right\} \texttt{s}_1 \left\{Q \mid r.U\right\}\right]\!\!\right] (\rho, \gamma)$ and
2. $n \in \left[\!\!\left[R; \Phi; \Gamma \vdash \left\{P * \mathsf{e} = 0\right\} \texttt{s}_2 \left\{Q \mid r.U\right\}\right]\!\!\right] (\rho, \gamma)$

then

$$n \in \left[\!\!\left[R; \Phi; \Gamma \vdash \left\{P\right\} \texttt{if(e) then \{s}_1\texttt{\} else \{s}_2\texttt{\}} \left\{Q \mid r.U\right\}\right]\!\!\right] (\rho, \gamma)$$

Proof. Suppose an environment E such that for all $n' < n$ we have $E \vDash^{\rho}_{n'} \Phi$. Suppose further a stack σ. We now need to show that

$$\begin{aligned}&\mathsf{safe}_n(E, [\![\mathsf{dom}(R); \Gamma \vdash P : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma),\\&\qquad \sigma,\\&\qquad \texttt{if(e) then \{s}_1\texttt{\} else \{s}_2\texttt{\}},\\&\qquad [\![\mathsf{dom}(R); \Gamma \vdash Q : \mathsf{Assn}]\!]\,(\rho, \gamma),\\&\qquad \lambda v.\,[\![R; \Gamma, r : \mathsf{Val} \vdash U : \mathsf{Assn}]\!]\,(\rho, \gamma[r \mapsto v]))\end{aligned}$$

Proceed by cases on the whether result of $[\![\mathsf{e}]\!]_\sigma$ is different from or equal to zero.

In either case we have to show Case 4 of Definition 102, where we take a regular execution step.

In the case where $[\![\mathsf{e}]\!]_\sigma \neq 0$, there is only one applicable rule,

$$\frac{[\![\mathsf{e}]\!]_\sigma \neq 0}{E \vdash (\sigma, \mathtt{if}(\mathtt{e})\ \mathtt{then}\ \{\mathtt{s}_1\}\ \mathtt{else}\ \{\mathtt{s}_2\}) \xrightarrow{\mathtt{id}} (\sigma, \mathtt{s}_1)}$$

Hence, we need to show that there is a P' such that

$$[\![\mathrm{dom}(R); \Gamma \vdash P : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma) \preceq P'$$

and

$$\begin{aligned}\mathrm{safe}_{n-1}(E, P',\\ &\sigma,\\ &\mathtt{s}_1,\\ &[\![\mathrm{dom}(R); \Gamma \vdash Q : \mathsf{Assn}]\!]\,(\rho, \gamma),\\ &\lambda v.\,[\![R; \Gamma, r : \mathsf{Val} \vdash U : \mathsf{Assn}]\!]\,(\rho, \gamma[r \mapsto v]))\end{aligned}$$

We chose $P' := [\![\mathrm{dom}(R); \Gamma \vdash P * \mathsf{e} \neq 0 : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma)$. To show the entailment, we compute on both sides. First the right, exploiting that e does not refer to logical variables.

$$\begin{aligned}&[\![\mathrm{dom}(R); \Gamma \vdash P * \mathsf{e} \neq 0 : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma)\\ =\,&[\![\mathrm{dom}(R); \Gamma \vdash P : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma) * [\![\mathsf{e}]\!]_\sigma \neq_{\mathsf{Val}} 0\end{aligned}$$

Observing that $\top$ is unit to $*$, we can frame the interpretation of P away by Lemma 99, and it remains to show that

$$\top \preceq [\![\mathsf{e}]\!]_\sigma \neq_{\mathsf{Val}} 0$$

which holds as $[\![\mathsf{e}]\!]_\sigma \neq 0$ by assumption.

The safety requirement on s_1 follows precisely by the assumption on s_1.

The case of $[\![\mathsf{e}]\!] = 0$ is analogous. □

Lemma 117 (Soundness of Sequencing)**.** For all $n \in \mathbb{N}$, if, for any $\gamma \in [\![\Gamma]\!]$ and $\rho \in [\![R]\!]$,

$$n \in \left[\!\!\left[R; \Phi; \Gamma \vdash \left\{P\right\}\ \mathtt{s}_1\ \left\{R \mid r.U\right\} : \mathsf{Spec}\right]\!\!\right](\rho, \gamma)$$

$$n \in \left[\!\!\left[R; \Phi; \Gamma \vdash \left\{R\right\}\ \mathtt{s}_2\ \left\{Q \mid r.U\right\} : \mathsf{Spec}\right]\!\!\right](\rho, \gamma)$$

then

$$n \in \left[\!\!\left[R; \Phi; \Gamma \vdash \left\{P\right\}\ \mathtt{s}_1;\mathtt{s}_2\ \left\{Q \mid r.U\right\} : \mathsf{Spec}\right]\!\!\right](\rho, \gamma)$$

Proof. Proceed by induction: assume the lemma holds for all $n' < n$. We now show it holds for n.

Suppose an environment E such that for all $n' < n.E \vDash^{\rho}_{n'} \Phi$. Suppose further a stack σ. We now need to show that

$$\begin{aligned}\mathrm{safe}_n(E, &[\![\mathrm{dom}(R);\Gamma \vdash P : \mathsf{Assn}]\!]\,(\rho,\gamma),\\ &\sigma,\\ &\mathtt{s}_1;\mathtt{s}_2,\\ &[\![\mathrm{dom}(R);\Gamma \vdash Q : \mathsf{Assn}]\!]\,(\rho,\gamma),\\ &\lambda v.\,[\![U]\!]\,(\rho,\gamma[r \mapsto v]))\end{aligned}$$

We case on whether $\mathtt{s}_1 = \mathtt{skip}$, $\mathtt{s}_1 = \mathtt{return\ e}$ or not; whether to skip, return or step.

Skip If $\mathtt{s}_1 = \mathtt{skip}$, we obtain from the first assumption that

$$[\![\mathrm{dom}(R);\Gamma \vdash P : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma) \preceq [\![\mathrm{dom}(R);\Gamma \vdash R : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma)$$

holds. We also observe that the only possible step is

$$\frac{}{E \vdash (\sigma, \mathtt{skip};\mathtt{s}_2) \xrightarrow{\mathtt{id}} (\sigma, \mathtt{s}_2)}$$

We thus now need to show that there is a P' : *Assertion* such that

(a) $[\![\mathrm{dom}(R);\Gamma \vdash P : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma) \preceq P'$

(b)

$$\begin{aligned}\mathrm{safe}_{n-1}(E,\\ &P',\\ &\sigma,\\ &\mathtt{s}_2, [\![\mathrm{dom}(R);\Gamma \vdash Q : \mathsf{Assn}]\!]\,(\rho,\gamma),\\ &\lambda v.\,[\![R;\Gamma, r : \mathsf{Val} \vdash U : \mathsf{Assn}]\!]\,(\rho,\gamma[r \mapsto v]))\end{aligned}$$

We chose $P' := [\![\mathrm{dom}(R);\Gamma \vdash R : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma)$ and we have both by assumption.

Return If $\mathtt{s}_1 = \mathtt{return\ e}$ or $\mathtt{return\ e};\mathtt{s'}_1$, we obtain by assumption on $\mathtt{s}_1$ that

$$[\![\mathrm{dom}(R);\Gamma \vdash P : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma) \preceq [\![R;\Gamma, r : \mathsf{Val} \vdash U : \mathsf{Assn}]\!]\,(\rho,\gamma[r \mapsto [\![\mathtt{e}]\!]_\sigma],\sigma)$$

which is precisely the requirement for

$$\begin{aligned}\mathrm{safe}_n(E,\\ &[\![\mathrm{dom}(R);\Gamma \vdash P : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma),\\ &\sigma,\\ &\mathtt{return\ e};\mathtt{s}_2,\\ &[\![\mathrm{dom}(R);\Gamma \vdash Q : \mathsf{Assn}]\!]\,(\rho,\gamma),\\ &\lambda v.\,[\![R;\Gamma, r : \mathsf{Val} \vdash U : \mathsf{Assn}]\!]\,(\rho,\gamma[r \mapsto v]))\end{aligned}$$

Forking Step If

$$E \vdash (\sigma, \mathtt{s}_1) \xrightarrow{\mathtt{fork}(\mathtt{f}, \vec{v})} (\sigma', \mathtt{s}_1')$$

we obtain by assumption on $\mathtt{s}_1$ variables and code $(\vec{\mathtt{x}}, \mathtt{s})$, assertions P and F' such that

(a) $E(\mathtt{f}) = (\vec{\mathtt{x}}, \mathtt{s})$

(b) $[\![\mathrm{dom}(R); \Gamma \vdash P : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma) \preceq P' * F$

(c)

$$\begin{aligned}\mathrm{safe}_{n-1}(&E,\\ &P',\\ &\sigma',\\ &\mathtt{s}_1',\\ &[\![\mathrm{dom}(R); \Gamma \vdash Q : \mathsf{Assn}]\!]\,(\rho, \gamma),\\ &\lambda v.\, [\![R; \Gamma, r : \mathsf{Val} \vdash U : \mathsf{Assn}]\!]\,(\rho, \gamma[r \mapsto v]))\end{aligned}$$

(d) $\mathrm{safe}_{n-1}(E, F, [\vec{\mathtt{x}} \mapsto \vec{v}], \mathtt{s}, \lambda\_.\top, \lambda\_.\lambda\_.\top)$

Hence, the compound statement can step as follows:

$$\frac{E \vdash (\sigma, \mathtt{s}_1) \xrightarrow{\mathtt{fork}(\mathtt{f}, \vec{v})} (\sigma', \mathtt{s}_1')}{E \vdash (\sigma, \mathtt{s}_1;\mathtt{s}_2) \xrightarrow{\mathtt{fork}(\mathtt{f}, \vec{v})} (\sigma', \mathtt{s}_1';\mathtt{s}_2)}$$

and by the definition of safety we now have to show the existence of $(\vec{\mathtt{x}}, \mathtt{s})$ and P, F' such that

(a) $E(\mathtt{f}) = (\vec{\mathtt{x}}, \mathtt{s})$

(b) $[\![\mathrm{dom}(R); \Gamma \vdash P : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma) \preceq P' * F$

(c)

$$\begin{aligned}\mathrm{safe}_{n-1}(&E,\\ &P',\\ &\sigma',\\ &\mathtt{s}_1';\mathtt{s}_2,\\ &[\![\mathrm{dom}(R); \Gamma \vdash Q : \mathsf{Assn}]\!]\,(\rho, \gamma),\\ &\lambda v.\, [\![R; \Gamma, r : \mathsf{Val} \vdash U : \mathsf{Assn}]\!]\,(\rho, \gamma[r \mapsto v])\end{aligned}$$

(d) $\mathrm{safe}_{n-1}(E, F, [\vec{\mathtt{x}} \mapsto \vec{v}], \mathtt{s}, \lambda\_.\top, \lambda\_.\lambda\_.\top)$

We choose precisely the givens obtained by the assumption $\mathtt{s}_1$. Items (a), (b) and (d) are thus given directly. (c) follows by the induction hypothesis.

Non-Forking Step If $\mathtt{s}_1$ can step according to

$$E \vdash (\sigma, \mathtt{s}_1) \xrightarrow{\alpha} (\sigma', \mathtt{s}_1')$$

we obtain by assumption on $\mathtt{s}_1$ an assertion P' such that

(a) $\alpha \Vdash_\rho \{[\![\mathrm{dom}(R); \Gamma \vdash P : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma)\}\{P'\}$

(b)

$$\begin{aligned}\mathrm{safe}_{n-1}(&E,\\ &P',\\ &\sigma',\\ &\mathtt{s}_1',\\ &[\![\mathrm{dom}(R); \Gamma \vdash R : \mathsf{Assn}]\!]\,(\rho, \gamma),\\ &\lambda v.\,[\![R; \Gamma, r : \mathsf{Val} \vdash U : \mathsf{Assn}]\!]\,(\rho, \gamma[r \mapsto v]))\end{aligned}$$

That $\mathtt{s}_1$ can step implies that the compound statement can step according to

$$\frac{E \vdash (\sigma, \mathtt{s}_1) \xrightarrow{\alpha} (\sigma', \mathtt{s}_1')}{E \vdash (\sigma, \mathtt{s}_1;\mathtt{s}_2) \xrightarrow{\alpha} (\sigma', \mathtt{s}_1';\mathtt{s}_2)}$$

and hence we need to show the existence of P' such that

(a) $\alpha \Vdash_\rho \{[\![\mathrm{dom}(R); \Gamma \vdash P : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma)\}\{P'\}$

(b)

$$\begin{aligned}\mathrm{safe}_{n-1}(&E,\\ &P',\\ &\sigma',\\ &\mathtt{s}_1';\mathtt{s}_2,\\ &[\![\mathrm{dom}(R); \Gamma \vdash Q : \mathsf{Assn}]\!]\,(\rho, \gamma),\\ &\lambda v.\,[\![R; \Gamma, r : \mathsf{Val} \vdash U : \mathsf{Assn}]\!]\,(\rho, \gamma[r \mapsto v]))\end{aligned}$$

We chose P' obtained before and get (a) immediately. (b) follows by the induction hypothesis. □

Lemma 118 (Soundness of While)**.** For all $n \in \mathbb{N}$, $\rho \in [\![R]\!]$ and $\gamma \in [\![\Gamma]\!]$, if

$$n \in \left[\!\!\left[R; \Phi; \Gamma \vdash \left\{P * \mathtt{e} \neq 0\right\}\ \mathtt{s}\ \left\{I \mid r.U\right\}\right]\!\!\right](\rho, \gamma)$$

then

$$n \in \left[\!\!\left[R; \Phi; \Gamma \vdash \left\{I\right\}\ \mathtt{while(e)\{s\}}\ \left\{I * \mathtt{e} = 0 \mid r.U\right\}\right]\!\!\right](\rho, \gamma)$$

Proof. Proceed by Induction. Assume that the lemma as stated holds for all $n' < n$. We now show it holds for n.

Suppose an environment E such that for all $n' < n$ we have $E \vDash^{\rho}_{n'} \Phi$. Suppose further a stack σ. We now need to show that

$$\begin{aligned}\mathsf{safe}_n(E, &[\![\mathrm{dom}(R);\Gamma \vdash I : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma),\\ &\sigma,\\ &\texttt{while(e)\{s\}},\\ &[\![\mathrm{dom}(R);\Gamma \vdash I * \mathsf{e} = 0 : \mathsf{Assn}]\!]\,(\rho,\gamma),\\ &\lambda v.\,[\![R;\Gamma, r : \mathsf{Val} \vdash U : \mathsf{Assn}]\!]\,(\rho,\gamma[r \mapsto v])).\end{aligned}$$

Proceed by cases on the whether result of $[\![\mathsf{e}]\!]_\sigma$ is different from or equal to zero. In either case we have to show Case 4 of Definition 102, where we take a regular execution step.

Non-Zero In the case where $[\![\mathsf{e}]\!]_\sigma \neq 0$, there is only one applicable rule,

$$\frac{[\![\mathsf{e}]\!]_\sigma \neq 0}{E \vdash (\sigma, \texttt{while(e)\{s\}}) \xrightarrow{\texttt{id}} (\sigma, \texttt{s;while(e)\{s\}})}$$

Hence, we need to show that there is a P' such that

$$[\![\mathrm{dom}(R);\Gamma \vdash I : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma) \preceq P'$$

and

$$\begin{aligned}\mathsf{safe}_{n-1}(E, &P'\\ &\sigma,\\ &\texttt{s;while(e)\{s\}},\\ &[\![\mathrm{dom}(R);\Gamma \vdash I * \mathsf{e} = 0 : \mathsf{Assn}]\!]\,(\rho,\gamma),\\ &\lambda v.\,[\![R;\Gamma, r : \mathsf{Val} \vdash U : \mathsf{Assn}]\!]\,(\rho,\gamma[r \mapsto v])).\end{aligned}$$

We chose $P' := [\![\mathrm{dom}(R);\Gamma \vdash I * \mathsf{e} \neq 0 : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma)$. We thus need to show that

$$[\![\mathrm{dom}(R);\Gamma \vdash I : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma) \preceq [\![\mathrm{dom}(R);\Gamma \vdash I * \mathsf{e} \neq 0 : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma)$$

Which follows knowing $[\![\mathsf{e}]\!]_\sigma \neq 0$. Finally we need to show that

$$\begin{aligned}\mathsf{safe}_{n-1}(E, &[\![\mathrm{dom}(R);\Gamma \vdash I * \mathsf{e} \neq 0 : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma),\\ &\sigma,\\ &\texttt{s;while(e)\{s\}},\\ &[\![\mathrm{dom}(R);\Gamma \vdash I * \mathsf{e} = 0 : \mathsf{Assn}]\!]\,(\rho,\gamma),\\ &\lambda v.\,[\![R;\Gamma, r : \mathsf{Val} \vdash U : \mathsf{Assn}]\!]\,(\rho,\gamma[r \mapsto v])).\end{aligned}$$

which precisely corresponds to showing that

$$n - 1 \in \left[\!\!\left[R;\Phi;\Gamma \vdash \left\{I * \mathsf{e} \neq 0\right\} \texttt{s;while(e)\{s\}} \left\{I * \mathsf{e} = 0 \mid r.U\right\}\right]\!\!\right](\rho,\gamma)$$

We have by assumption on s that

$$n-1 \in \left[\!\left[R;\Phi;\Gamma \vdash \left\{ I * \mathsf{e} \neq 0 \right\} \ \mathtt{s} \ \left\{ I \mid r.U \right\} \right]\!\right] (\rho,\gamma)$$

and the induction hypothesis gives us

$$n-1 \in \left[\!\left[R;\Phi;\Gamma \vdash \left\{ I \right\} \ \mathtt{while(e)\{s\}} \ \left\{ I * \mathsf{e} = 0 \mid r.U \right\} \right]\!\right] (\rho,\gamma)$$

The two preceding statements are by the Soundness of Sequencing enough to give us precisely the desired conclusion.

Zero In the case where $[\![\mathsf{e}]\!]_\sigma = 0$, there is only one applicable rule,

$$\frac{[\![\mathsf{e}]\!]_\sigma = 0}{E \vdash (\sigma, \mathtt{while(e)\{s\}}) \xrightarrow{\mathtt{id}} (\sigma, \mathtt{skip})}$$

Hence, we need to show that there is a P' such that $[\![I]\!] (\rho,\gamma,\sigma) \preceq P'$ and

$$\begin{aligned}\mathsf{safe}_{n-1}(E, P',\ & \\ & \sigma, \\ & \mathtt{skip}, \\ & [\![\mathsf{dom}(R);\Gamma \vdash I * \mathsf{e} = 0 : \mathsf{Assn}]\!] (\rho,\gamma), \\ & \lambda v.\, [\![R;\Gamma, r : \mathsf{Val} \vdash U : \mathsf{Assn}]\!] (\rho, \gamma[r \mapsto v]).\end{aligned}$$

We chose $P' := [\![I * \mathsf{e} = 0]\!] (\rho,\gamma,\sigma)$, and the safety requirement holds by reflexivity of $\preceq$. To show the entailment we argue as in the the looping case: We can frame I on each side of the entailment by Lemma 99, and it remains to show that

$$\top \preceq [\![\mathsf{dom}(R);\Gamma \vdash \mathsf{e} = 0 : \mathsf{Assn}]\!] (\rho,\gamma,\sigma)$$

which always holds as $[\![\Gamma \vdash \mathsf{e} : \mathsf{Val}]\!] (\gamma,\sigma) = [\![\mathsf{e}]\!]_\sigma$ when e is free of program variables, as here. □

Lemma 119 (Soundness of Skip)**.** For all assertions P and U and $\rho \in [\![R]\!]$ and $\gamma \in [\![\Gamma]\!]$, for all $n \in \mathbb{N}$,

$$n \in \left[\!\left[R;\Phi;\Gamma \vdash \left\{ P \right\} \ \mathtt{skip} \ \left\{ P \mid r.U \right\} : \mathsf{Spec} \right]\!\right] (\rho,\gamma)$$

Proof. Suppose an environment E such that for all $n' < n$ we have $E \vDash^{\rho}_{n'} \Phi$. Suppose further a stack σ. We now need to show that

$$\begin{aligned}\mathsf{safe}_{n}(E, & [\![\mathsf{dom}(R);\Gamma \vdash P : \mathsf{Assn}]\!] (\rho,\gamma,\sigma) \\ & \sigma, \\ & \mathtt{skip}, \\ & [\![\mathsf{dom}(R);\Gamma \vdash P : \mathsf{Assn}]\!] (\rho,\gamma), \\ & \lambda v.\, [\![R;\Gamma, r : \mathsf{Val} \vdash U : \mathsf{Assn}]\!] (\rho, \gamma[r \mapsto v]))\end{aligned}$$

which means we need to show that

$$[\![\mathrm{dom}(R);\Gamma \vdash P : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma) \preceq [\![\mathrm{dom}(R);\Gamma \vdash P : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma).$$

This follows by reflexivity of $\preceq$. $\square$

Lemma 120 (Soundness of Local Assignment)**.** For all n, $\rho \in [\![R]\!]$, $\gamma \in [\![\Gamma]\!]$,

$$n \in \left[\!\!\left[R;\Phi;\Gamma \vdash \left\{\mathtt{x} = n\right\}\ \mathtt{x} := \mathsf{e}\ \left\{\mathtt{x} = \mathsf{e}[n/\mathtt{x}] \mid r.U\right\} : \mathsf{Spec}\right]\!\!\right](\rho,\gamma).$$

Proof. Suppose an environment E such that for all $n' < n$ we have $E \vDash^{\rho}_{n'} \Phi$. Suppose further a stack σ. We now need to show

$$\begin{aligned}\mathsf{safe}_n(E, &[\![\mathrm{dom}(R);\Gamma \vdash \mathtt{x} = n : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma),\\ &\sigma,\\ &\mathtt{x} := \mathsf{e},\\ &[\![\mathrm{dom}(R);\Gamma \vdash \mathtt{x} = \mathsf{e}[n/\mathtt{x}] : \mathsf{Assn}]\!]\,(\rho,\gamma),\\ &\lambda v.\,[\![U]\!]\,(\rho,\gamma[r \mapsto v])).\end{aligned}$$

There is only one applicable transition,

$$\frac{}{E \vdash (\sigma, \mathtt{x} := \mathsf{e}) \xrightarrow{\mathtt{id}} (\sigma[\mathtt{x} \mapsto [\![\mathsf{e}]\!]_\sigma], \mathtt{skip})}$$

So we pick $P' := [\![\mathrm{dom}(R);\Gamma \vdash \mathtt{x} = \mathsf{e}[n/\mathtt{x}] : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma[\mathtt{x} \mapsto [\![\mathsf{e}]\!]_\sigma])$ and have to show that

(a)

$$\begin{aligned}&[\![\mathrm{dom}(R);\Gamma \vdash \mathtt{x} = n : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma)\\ \preceq\, &[\![\mathrm{dom}(R);\Gamma \vdash \mathtt{x} = \mathsf{e}[n/\mathtt{x}] : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma[\mathtt{x} \mapsto [\![\mathsf{e}]\!]_\sigma])\end{aligned}$$

(b)

$$\begin{aligned}\mathsf{safe}_{n-1}(E,\ &\\ &[\![\mathrm{dom}(R);\Gamma \vdash \mathtt{x} = \mathsf{e}[n/\mathtt{x}] : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma[\mathtt{x} \mapsto [\![\mathsf{e}]\!]_\sigma]),\\ &\sigma[\mathtt{x} \mapsto [\![\mathsf{e}]\!]_\sigma],\\ &\mathtt{skip},\\ &[\![\mathrm{dom}(R);\Gamma \vdash \mathtt{x} = \mathsf{e}[n/\mathtt{x}] : \mathsf{Assn}[n/\mathtt{x}]]\!]\,(\rho,\gamma),\\ &\lambda v.\,[\![R;\Gamma, r : \mathsf{Val} \vdash U : \mathsf{Assn}]\!]\,(\rho,\gamma[r \mapsto v]))\end{aligned}$$

(b) is immediate by Lemma 119.
(a) follows by computation; first on the left hand side:

$$\begin{aligned}&[\![\mathrm{dom}(R);\Gamma \vdash \mathtt{x} = n : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma)\\ \iff &\sigma(\mathtt{x}) =_{\mathsf{Val}} n\end{aligned}$$

Then on the right hand side:

$$\begin{aligned}&[\![\mathrm{dom}(R); \Gamma \vdash \mathtt{x} = \mathsf{e}[n/\mathtt{x}] : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma[\mathtt{x} \mapsto [\![\mathsf{e}]\!]_\sigma])\\ \iff& [\![\mathsf{e}]\!]_\sigma =_{\mathsf{Val}} [\![\Gamma \vdash \mathsf{e}[n/\mathtt{x}] : \mathsf{Val}]\!]\,(\gamma, \sigma[\mathtt{x} \mapsto [\![\mathsf{e}]\!]_\sigma)\end{aligned}$$

Hence, by definition of semantic entailment, it suffices to show that, assuming $\sigma(\mathtt{x}) = n$, for any expression e and any value v it holds that

$$[\![\mathsf{e}]\!]_\sigma =_{\mathsf{Val}} [\![\Gamma \vdash \mathsf{e}[n/\mathtt{x}] : \mathsf{Val}]\!]\,(\gamma, \sigma[\mathtt{x} \mapsto v])$$

We proceed by induction on the structure of e:

Constant If $\mathsf{e} = m$, for a constant m, the result is immediate.

Variable If $\mathsf{e} = \mathtt{y}$ for some program variable $\mathtt{y}$, there are two cases: if $\mathtt{x} = \mathtt{y}$, we calculate

$$\begin{aligned}[\![\Gamma \vdash \mathsf{e}[n/\mathtt{x}] : \mathsf{Val}]\!]\,(\gamma, \sigma[\mathtt{x} \mapsto v]) &= [\![\Gamma \vdash \mathtt{x}[n/\mathtt{x}] : \mathsf{Val}]\!]\,(\gamma, \sigma[\mathtt{x} \mapsto v])\\ &= n\\ &= \sigma(\mathtt{x})\\ &= [\![\mathtt{x}]\!]_\sigma\\ &= [\![\mathsf{e}]\!]_\sigma\end{aligned}$$

as desired. If $\mathtt{x} \neq \mathtt{y}$ we calculate

$$\begin{aligned}[\![\Gamma \vdash \mathsf{e}[n/\mathtt{x}] : \mathsf{Val}]\!]\,(\gamma, \sigma[\mathtt{x} \mapsto v]) &= [\![\Gamma \vdash \mathtt{y}[n/\mathtt{x}] : \mathsf{Val}]\!]\,(\gamma, \sigma[\mathtt{x} \mapsto v])\\ &= [\![\Gamma \vdash \mathtt{y} : \mathsf{Val}]\!]\,(\gamma, \sigma[\mathtt{x} \mapsto v])\\ &= \sigma(\mathtt{y})\\ &= [\![\mathtt{y}]\!]_\sigma\\ &= [\![\mathsf{e}]\!]_\sigma\end{aligned}$$

Binary Operator If $\mathsf{e} = \mathsf{e}_1 \circledast \mathsf{e}_2$ for some binary operation, we can calculate as follows:

$$\begin{aligned}[\![\Gamma \vdash \mathsf{e} : \mathsf{Val}[n/\mathtt{x}]]\!]\,(\gamma, \sigma[\mathtt{x} \mapsto v]) &= [\![\Gamma \vdash \mathsf{e}_1 \circledast \mathsf{e}_2 : \mathsf{Val}[n/\mathtt{x}]]\!]\,(\gamma, \sigma[\mathtt{x} \mapsto v])\\ &= [\![\Gamma \vdash \mathsf{e}_1[n/\mathtt{x}] : \mathsf{Val}]\!]\,(\gamma, \sigma[\mathtt{x} \mapsto v])\\ &\quad \circledast [\![\Gamma \vdash \mathsf{e}_2 : \mathsf{Val}[n/\mathtt{x}]]\!]\,(\gamma, \sigma[\mathtt{x} \mapsto v])\\ &= [\![\mathsf{e}_1]\!]\,(\sigma) \circledast [\![\mathsf{e}_2]\!]\,(\sigma)\\ &= [\![\mathsf{e}_1 \circledast \mathsf{e}_2]\!]_\sigma\\ &= [\![\mathsf{e}]\!]_\sigma\end{aligned}$$

where the third equality holds by the induction hypotheses of e_1 and e_2. □

Lemma 121 (Soundness of Frame)**.** For any $\rho \in [\![R]\!]$, $\gamma \in [\![\Gamma]\!]$ and $\sigma \in \mathit{Stack}$, assuming $\mathrm{mod}(\mathtt{s}) \cap F = \emptyset$, if

$$n \in \left[\!\!\left[R;\Phi;\Gamma \vdash \left\{P\right\} \;\mathtt{s}\; \left\{Q \mid r.U\right\} : \mathsf{Spec}\right]\!\!\right] (\rho,\gamma,\sigma)$$

then

$$n \in \left[\!\!\left[R;\Phi;\Gamma \vdash \left\{P * F\right\} \;\mathtt{s}\; \left\{Q * F \mid r.U\right\} : \mathsf{Spec}\right]\!\!\right] (\rho,\gamma,\sigma).$$

Proof. We proceed by strong induction on n. Suppose an n such that it lies in the interpretation of $R;\Phi;\Gamma \vdash \left\{P\right\} \;\mathtt{s}\; \left\{Q \mid r.U\right\}$ We now have to show that n lies in the interpretation of $R;\Phi;\Gamma \vdash \left\{P * F\right\} \;\mathtt{s}\; \left\{Q * F \mid r.U\right\}$, which we proceed to do so according to definition:

Suppose an environment E such that for all $n' < n$ we have $E \vDash^{\rho}_{n'} \Phi$. Suppose further a stack σ. We now need to show

$$\begin{aligned}\mathsf{safe}_n(E, &[\![R;\Gamma \vdash P : \mathsf{Assn}]\!] (\rho,\gamma,\sigma),\\ &\sigma,\\ &\mathtt{s},\\ &[\![R;\Gamma \vdash Q : \mathsf{Assn}]\!] (\rho,\gamma),\\ &\lambda v.\,[\![R;\Gamma, r : \mathsf{Val} \vdash U : \mathsf{Assn}]\!] (\rho,\gamma[r \mapsto v]))\end{aligned}$$

There are 4 cases, according to the definition of safe:

Skip If $\mathtt{s} = \mathtt{skip}$, then we must show that

$$\mathtt{id} \Vdash_{\rho} \{[\![R;\Gamma \vdash P * F : \mathsf{Assn}]\!] (\rho,\gamma,\sigma)\}\{[\![R;\Gamma \vdash Q * F : \mathsf{Assn}]\!] (\rho,\gamma,\sigma)\}$$

but by the assumption on n, we also get that

$$\mathtt{id} \Vdash_{\rho} \{[\![R;\Gamma \vdash P : \mathsf{Assn}]\!] (\rho,\gamma,\sigma)\}\{[\![R;\Gamma \vdash Q : \mathsf{Assn}]\!] (\rho,\gamma,\sigma)\}$$

By the semantics of assertions, the interpretation of assertions commutes with $*$, and we then have by Lemma 99 precisely what we need to show.

Return This case is analogous to the case of `skip`.

Forking Step In this case, we assume that $E \vdash (\sigma, \mathtt{s}) \xrightarrow{\mathtt{fork}((,)f,\vec{v})} (\sigma', \kappa')$ for some `f` and $\vec{v}$. We now have to provide P', F' and $(\vec{\mathtt{x}}, \mathtt{s})$ such that

1. $E(\mathtt{f}) = (\vec{\mathtt{x}}, \mathtt{s})$
2. $\mathtt{id} \Vdash_{\rho} \{[\![R;\Gamma \vdash P * F : \mathsf{Assn}]\!] (\rho,\gamma,\sigma)\}\{P' * F'\}$
3. $\mathsf{safe}_{n-1}(E, P', \sigma', \kappa', [\![R;\Gamma \vdash Q * F : \mathsf{Assn}]\!] (\rho,\gamma), [\![U]\!])$
4. $\mathsf{safe}_{n-1}(E, F, \sigma', \mathtt{s}, \lambda\_.\top, \lambda\_.\lambda\_.\top)$

By appeal to the assumption on n, we get precisely the pieces we need, however, we get that $\texttt{id} \Vdash_\rho \{[\![R;\Gamma \vdash P : \mathsf{Assn}]\!]\}\{P' * F'\}$. Here, we instead of just P', we chose $P' * [\![R;\Gamma \vdash F : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma')$, our original frame. Hence, by Lemma 96 we get Item 2. Item 4 is then still immediate by assumption. It remains to show Item 3, which now amounts to showing that

$$\mathsf{safe}_{n-1}(E, P' * [\![R;\Gamma \vdash F : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma'), \kappa', [\![Q * F]\!]\,(\rho,\gamma), [\![U]\!])$$

which we get by assumption on n combined with the induction hypothesis.

Non-Forking Step Analogous to the forking case without the complication of the forked thread.

□

Lemma 122. If for any σ, $Q'(\sigma) \preceq Q(\sigma)$ and $\mathsf{safe}_n(E,P,\sigma,\mathtt{s},Q',U)$ then $\mathsf{safe}_n(E,P,\sigma,\mathtt{s},Q,U)$.

Proof. Proceed by strong induction on n.

Skip If $\mathtt{s} = \texttt{skip}$, we must show $P \preceq Q(\sigma)$ knowing $P \preceq Q'(\sigma)$ by assumption. This follows by transitivity of $\preceq$.

Return If $\mathtt{s}$ returns some expression $\mathtt{e}$, we must show $P \preceq U(\sigma)([\![\mathtt{e}]\!]_\sigma)$, but we get this immediately by assumption.

Forking Step Here we must find assertions P', F such that

1. $P \preceq P' * F$
2. $\mathsf{safe}_{n-1}(E, P', \sigma', \mathtt{s}', Q, U)$
3. $\mathsf{safe}_{n-1}(E, F, [\vec{\mathtt{x}} \mapsto [\![\vec{\mathtt{e}}]\!]_\sigma], \mathtt{s}, \top, \top)$

We get the choice of assertions by the assumption on $\mathtt{s}$, and Items 1 and 3 are immediate while Item 2 follows by the induction hypothesis.

Non-forking step Here we must find assertion P' such that

1. $\alpha \Vdash_\rho \{P\}\{P'\}$
2. $\mathsf{safe}_{n-1}(E, P', \sigma', \mathtt{s}', Q, U)$

Both again follow immediately by assumption and the induction hypothesis. □

Lemma 123 (Soundness of Consequence)**.** For any $\rho \in [\![R]\!]$, $\gamma \in [\![\Gamma]\!]$ and $\sigma \in \mathit{Stack}$, we have that, assuming

$$R;\Gamma \mid P \vdash P' \qquad R;\Gamma \mid Q' \vdash Q \qquad R;\Gamma \mid U' \vdash U,$$

then for any n and $\gamma \in [\![\Gamma]\!]$, if

$$n \in \left[\!\!\left[R;\Phi;\Gamma \vdash \left\{P'\right\} \ \mathtt{s} \ \left\{Q' \mid r.U'\right\} : \mathsf{Spec}\right]\!\!\right] (\rho,\gamma,\sigma)$$

then

$$n \in \left[\!\!\left[R;\Phi;\Gamma \vdash \left\{P\right\} \ \mathtt{s} \ \left\{Q \mid r.U\right\} : \mathsf{Spec}\right]\!\!\right] (\rho,\gamma,\sigma)$$

Proof. We proceed according to the proof of the Soundness of Fork: by strong induction on n, and then by case on the definition of safety.

In each case, we appeal Lemma 101 in order to weaken or strengthen the assertions involved as needed, remarking that the assumed syntactic entailments give us e.g.

$$[\![R;\Gamma \vdash P : \mathsf{Assn}]\!] (\rho,\gamma,\sigma) \preceq [\![R;\Gamma \vdash P' : \mathsf{Assn}]\!] (\rho,\gamma,\sigma).$$

Again, the return and skipping cases are alike, so we show one. The same is true of the forking and the non-forking case so we show the more complicated of the two.

Skip In the case of $\mathtt{s} = \mathtt{skip}$, we are to show that $P \preceq Q$, knowing $P' \preceq Q'$, $P \preceq P'$ and $Q' \preceq Q$, which is directly the statement of Lemma 101.

Forking Step In the case that $\mathtt{s} = \mathtt{fork}\ \mathtt{f}(\vec{\mathtt{e}})$, we are to give an implementation $(\vec{\mathtt{x}},\mathtt{s})$ and assertions R and F such that

1. $P \preceq R * F$
2. $\mathsf{safe}_{n-1}(E,R,\sigma,\mathtt{s}',Q,U)$
3. $\mathsf{safe}_{n-1}(E,F,[\vec{\mathtt{x}} \mapsto [\![\vec{\mathtt{e}}]\!]_\sigma],\mathtt{s},\top,\top)$

The choices of $(\vec{\mathtt{x}},\mathtt{s})$, R and F are given by the assumption of the lemma. The first item follows from transitivity of $\preceq$, knowing $P \preceq P'$ and $P' \preceq R * F$. The remaining two items follow by assumption with an appeal to Lemma 122. □

Lemma 124 (Soundness of Fork)**.** Assuming $\Phi(\mathtt{f}) = (\Gamma,\vec{y})\{P\}\{r.Q\}$, it is the case that, for all n, for all $\rho \in [\![R]\!]$, $\gamma \in [\![\Gamma]\!]$:

$$n \in$$

$$\left[\!\!\left[R;\Phi;\Gamma,\vec{y} : \mathsf{Val} \vdash \left\{P * (\vec{\mathtt{e}} = \vec{y})\right\} \ \mathtt{fork}\ \mathtt{f}(\vec{\mathtt{e}}) \ \left\{\top \mid r.U\right\} : \mathsf{Spec}\right]\!\!\right] (\rho,\gamma)$$

Proof. Suppose an environment E such that for all $n' < n$ we have $E \vDash^{\rho}_{n'} \Phi$. Suppose further a stack σ. We now need to show

$$\begin{aligned}\mathsf{safe}_n(E,&[\![R;\Gamma,\vec{y} : \mathsf{Val} \vdash P * (\vec{\mathtt{e}} = \vec{y}) : \mathsf{Assn}]\!] (\rho,\gamma,\sigma),\\ &\sigma,\\ &\mathtt{fork}\ \mathtt{f}(\vec{\mathtt{e}}),\\ &[\![R;\Gamma,\vec{y} : \mathsf{Val} \vdash \top : \mathsf{Assn}]\!] (\rho,\gamma),\\ &\lambda v.\, [\![R;\Gamma,\vec{y} : \mathsf{Val}, r : \mathsf{Val} \vdash U : \mathsf{Assn}]\!] (\rho,\gamma[r \mapsto v]))\end{aligned}$$

By case analysis of first the statement in question and then the operational rules, we see that the only applicable case of safe_n is the case where

$$E \vdash (\sigma, \mathtt{fork\ f}(\vec{\mathtt{e}})) \xrightarrow{\mathtt{fork}(\mathtt{f}, [\![\vec{\mathtt{e}}]\!]_\sigma)} (\sigma, \mathtt{skip}).$$

This means showing that there exists a P', F and $(\vec{\mathtt{x}}, \mathtt{s})$ such that the following four items hold:

(a) $E(\mathtt{f}) = (\vec{\mathtt{x}}, \mathtt{s})$.

(b) $[\![R; \Gamma, \vec{y} : \mathsf{Val} \vdash P * (\vec{\mathtt{e}} = \vec{y}) : \mathsf{Assn}]\!] (\rho, \gamma, \sigma) \preceq P' * F$

(c)

$$\begin{aligned}\mathsf{safe}_{n-1}(E,\ & \\ & P', \\ & \sigma, \\ & \mathtt{skip}, \\ & [\![R; \Gamma, \vec{y} : \mathsf{Val} \vdash \top : \mathsf{Assn}]\!] (\rho, \gamma), \\ & \lambda v. [\![R; \Gamma, \vec{y} : \mathsf{Val}, r : \mathsf{Val} \vdash U : \mathsf{Assn}]\!] (\rho, \gamma[r \mapsto v]))\end{aligned}$$

(d) $\mathsf{safe}_{n-1}(E, F, [\vec{\mathtt{x}} \mapsto [\![\vec{\mathtt{e}}]\!]_\sigma], \mathtt{s}, \lambda\_.\top, \lambda\_\lambda\_.\top)$

By assumption that $\mathtt{f}$ is specified by Φ, we obtain $(\vec{\mathtt{x}}, \mathtt{s})$ that satisfy (a). We choose

$$P' := \top \qquad F := [\![R; \Gamma, \vec{y} : \mathsf{Val} \vdash P * (\vec{\mathtt{x}} = \vec{y}) : \mathsf{Assn}]\!] (\rho, \gamma, [\vec{\mathtt{x}} \mapsto [\![\vec{\mathtt{e}}]\!]_\sigma])$$

(c) follows easily: the interpretation $[\![\top]\!] (\rho, \gamma, \sigma)$ for any arguments is $\top$, and $\top \preceq \top$, as required by safety of $\mathtt{skip}$. (d) follows from the assumption that $E \vDash^{\rho}_{n-1} \Phi$ by appeal to weakening of the conclusions: we can always weaken to $\top$.

Left is (b), which follows precisely from 113. □

Lemma 125 (Soundness of Allocation)**.** For all $\rho \in [\![R]\!]$, $\gamma \in [\![\Gamma]\!]$ and σ, for any n it holds that n lies in

$$\left[\!\!\left[R; \Gamma \vdash \left\{\mathtt{e} = v * v > 0\right\} \mathtt{x} := \mathtt{alloc}(\mathtt{e}) \left\{\exists n. \mathtt{x} = n * n \mapsto [v] \mid r.U\right\} : \mathsf{Spec}\right]\!\!\right] (\rho, \gamma, \sigma)$$

Proof. Suppose an environment E such that for all $n' < n$ we have $E \vDash^{\rho}_{n'} \Phi$. Suppose further a stack σ. We now need to show

$$\begin{aligned}\mathsf{safe}_n(E, & [\![R; \Gamma \vdash \mathtt{e} = v * v > 0 : \mathsf{Assn}]\!] (\rho, \gamma, \sigma), \\ & \sigma, \\ & \mathtt{x} := \mathtt{alloc}(\mathtt{e}), \\ & [\![R; \Gamma \vdash \exists n. \mathtt{x} = n * n \mapsto [v] : \mathsf{Assn}]\!] (\rho, \gamma), \\ & \lambda v. [\![R; \Gamma, r : \mathsf{Val} \vdash U : \mathsf{Assn}]\!] (\rho, \gamma[r \mapsto v]))\end{aligned}$$

According to the operational semantics, the allocation can either complete successfully or fault, depending on whether e denotes a non-zero natural number.

If it does not, i.e. $[\![\mathtt{e}]\!]_\sigma < 1$ it is the case that $E \vdash (\sigma, \mathtt{x} := \mathtt{alloc}(\mathtt{e})) \xrightarrow{\lightning} (\sigma, \mathtt{skip})$ and we have to pick a P' according to Case 4 of the definition of safe$_n$. We chose $\bot$ and have to show:

1. $\lightning \Vdash_\rho \{[\![R;\Gamma \vdash \mathtt{e} = v * v > 0 : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma)\}\{\bot\}$

2. $$\begin{aligned}\text{safe}_n(&E,\\ &\bot,\\ &\sigma,\\ &\mathtt{skip},\\ &[\![R;\Gamma \vdash \exists n.\mathtt{x} = n * n \mapsto [v] : \mathsf{Assn}]\!]\,(\rho,\gamma),\\ &\lambda v.\,[\![R;\Gamma, r : \mathsf{Val} \vdash U : \mathsf{Assn}]\!]\,(\rho,\gamma[r \mapsto v])\end{aligned}$$

Item 2 is vacuously true as $\bot \preceq P$ holds of any P. To argue Item 1 we observe that any configuration in the in the interpretation of $\mathtt{e} = v$ and $v > 0$ would contradict that $v < 1$, hence there are none. Therefore, that interpretation is empty, and the inclusion required in Item 1 is empty. (The action of faulting on a set of heaps is thus also irrelevant - the lifting of actions to sets of heaps preserve the empty set).

If $\mathtt{e} > 0$, it is the case that $E \vdash (\sigma, \mathtt{x} := \mathtt{alloc}(\mathtt{e})) \xrightarrow{\mathtt{alloc}([\![\mathtt{e}]\!]_\sigma, n)} (\sigma[\mathtt{x} \mapsto n], \mathtt{skip})$ for some address n and we thus need to find a P', for which we pick the postcondition as stated in the lemma, such that

1. $$\begin{aligned}&\mathtt{alloc}([\![\mathtt{e}]\!]_\sigma, n) \Vdash_\rho\\ &\{[\![R;\Gamma \vdash \mathtt{e} = v * v > 0 : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma)\}\\ &\{[\![R;\Gamma \vdash \exists n.\mathtt{x} = n * n \mapsto [v] : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma[\mathtt{x} \mapsto n])\}\end{aligned}$$

2. $$\begin{aligned}\text{safe}_n(&E,\\ &[\![R;\Gamma \vdash \exists n.\mathtt{x} = n * n \mapsto [v] : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma[\mathtt{x} \mapsto n]),\\ &\sigma,\\ &\mathtt{skip},\\ &[\![R;\Gamma \vdash \exists n.\mathtt{x} = n * n \mapsto [v] : \mathsf{Assn}]\!]\,(\rho,\gamma),\\ &\lambda v.\,[\![R;\Gamma, r : \mathsf{Val} \vdash U : \mathsf{Assn}]\!]\,(\rho,\gamma[r \mapsto v])\end{aligned}$$

Item 2 holds by the definition of safety of skip. To show the first item, we proceed directly by definition of the semantic action judgment: suppose a stable frame, and suppose a heap h in the erasure of the conjunction of said frame and $[\![R;\Gamma \vdash \mathsf{e} = v * v > 0 : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma)$. For h we know that $[\![\mathsf{e}]\!]_\sigma = v$ and $v > 0$, consistent with the assumption on e.

Any h' in the result of applying the action of the allocation action would satisfy that $m \in \mathrm{dom}(h')$ if m is such that $n <= m < n + (v-1)$. This is ensured knowing $v > 0$.

Finally, we need to show that h' lies in the erasure of

$$[\![R;\Gamma \vdash \exists n.\mathtt{x} = n * n \mapsto [v] : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma[\mathtt{x} \mapsto n]).$$

For this to be the case, there needs to be an n such $n = n$ and $n \mapsto v$ is satisfied by the heap. This is precisely the case for h'. Hence we are done. □

Lemma 126 (Soundness of Atomic Embedding)**.** Assuming

$$[\![R;\Gamma \vdash [P]\ \mathsf{open}\ [\{(\Delta_i).(P_i', \vec{r}_i)\}]]\!]\,(\rho,\gamma,\sigma)$$

and, for all $i \in I$ and $\delta_i \in [\![\Delta_i]\!]$,

$$[\![R;\Gamma,\Delta_i \vdash_{\{\vec{r}_i\}} \langle P_i'\rangle\ \mathsf{s}\ \langle Q_i * \mathit{newRegion}(\vec{n}_i)\rangle]\!]\,(\rho,\gamma\delta_i,\sigma)$$

$$[\![R;\Gamma,\Delta_i \vdash [Q_i * \mathit{newRegion}(\vec{n}_i)]\ \mathsf{close}(\vec{r}_i,\vec{n}_i)\ [Q]]\!]\,(\rho,\gamma\delta_i,\sigma)$$

then for all $n \in \mathbb{N}$,

$$n \in \left[\!\!\left[R;\Phi;\Gamma \vdash \left\{P\right\}\ \mathsf{s}\ \left\{\mathsf{stabilize}(Q) \mid r.U\right\}\right]\!\!\right]\,(\rho,\gamma,\sigma)$$

Proof. Assume

1. $[\![R;\Gamma \vdash [P]\ \mathsf{open}\ [\{(\Delta_i).(P_i', \vec{r}_i)\}]]\!]\,(\rho,\gamma,\sigma)$
2. $[\![R;\Gamma,\Delta_i \vdash_{\{\vec{r}_i\}} \langle P_i'\rangle\ \mathsf{s}\ \langle Q_i * \mathit{newRegion}(\vec{n}_i)\rangle]\!]\,(\rho,\gamma\cdot\delta_i,\sigma)$
3. For all $i \in I$, $[\![R;\Gamma,\Delta_i \vdash [Q_i * \mathit{newRegion}(\vec{n}_i)]\ \mathsf{close}(\vec{r}_i,\vec{n}_i)\ [Q]]\!]\,(\rho,\gamma\cdot\delta_i,\sigma)$

Suppose an environment E such that for all $n' < n$ we have $E \vDash^{\rho}_{n'} \Phi$. Suppose further a stack σ. We now need to show that

$$\begin{aligned}\mathsf{safe}_n(&E, [\![R;\Gamma \vdash P : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma),\\ &\sigma,\\ &\mathsf{s},\\ &[\![R;\Gamma \vdash \mathsf{stabilize}(Q) : \mathsf{Assn}]\!]\,(\rho,\gamma),\\ &\lambda v.\,[\![R;\Gamma, r : \mathsf{Val} \vdash U : \mathsf{Assn}]\!]\,(\rho,\gamma[r \mapsto v])).\end{aligned}$$

Since s is atomic by Assumption 2, there is only the possibility that the code take a non-forking step according to the operational semantics, one that immediately reaches the `skip` configuration.

Hence, we can only step according to

$$E \vdash (\sigma, \mathsf{s}) \xrightarrow{\alpha} (\sigma', \mathtt{skip})$$

and we need to find a P' such that

1. $\alpha \Vdash_\rho \{[\![R;\Gamma \vdash P : \mathsf{Assn}(\rho,\gamma,\sigma)]\!]\}\{P'\}$
2. $\mathsf{safe}_{n-1}(E, P', \sigma', \mathtt{skip}, [\![R;\Gamma \vdash \mathsf{stabilize}(Q) : \mathsf{Assn}]\!]\,(\rho,\gamma), [\![U]\!])$

We chose $P' := [\![R;\Gamma \vdash \mathsf{stabilize}(Q) : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma')$, and 2 is immediate. To show 1, assume a stable frame R. We now need to argue that

$$[\![\alpha]\!]\,(\lfloor [\![P]\!]\,(\rho,\gamma,\sigma) * R\rfloor^{\rho}_{\emptyset}) \subseteq \lfloor [\![\mathsf{stabilize}(Q)]\!]\,(\rho,\gamma,\sigma') * R\rfloor^{\rho}_{\emptyset}$$

Suppose an abstract configuration (h,a) in $[\![P]\!] * R$. Hence, we can split (h,a) into $p = (h_P, a_P) \cdot (h_R, a_R) = r$. By Assumption 1, we thus get an index $i \in I$, a $\delta \in [\![\Delta]\!]_i$ and $p' \in [\![P'_i]\!]$ and $\vec{y} \in [\![\vec{r}_i]\!]$ and an r' such that $(r, r') \in \mathrm{Rely}(\rho)$ and $\lfloor p \cdot r\rfloor^{\rho}_{\emptyset} \subseteq \lfloor p' \cdot r'\rfloor^{\rho}_{\vec{y}}$.

By Assumption 2 and since

$$p' = (h_{P'}, a_{P'}) \in [\![P_i]\!]\,,$$

we get

$$([\![\alpha]\!]\,(h_{P'}), a_{P'}) \in [\![Q_i * \mathit{newRegion}(\vec{n}_i)]\!]\,.$$

By Assumption 3 we thus get that there is a frame r'' and assertion $q \in [\![Q]\!]$ such that

$$\lfloor([\![\alpha]\!]\,(h_{P'}), a_{P'}) \cdot r\rfloor^{\rho}_{\vec{r}_i,\vec{n}_i} \subseteq \lfloor q \cdot r''\rfloor^{\rho}_{\emptyset}\,.$$

By the transitivity of the rely relation, we know that r'' is in the assertion R as R is stable, i.e. closed under the rely relation.

Since this is shown for any configuration in $[\![P]\!] * R$ we have that $\lfloor [\![P]\!] * R\rfloor^{\rho}_{\emptyset} \subseteq \lfloor [\![Q]\!] * R\rfloor^{\rho}_{\emptyset}$. It remains to observe that $Q \subseteq \mathsf{stabilize}(Q)$ as remarked in Definition 93, and hence, by Lemma 99 we get

$$\lfloor [\![P]\!] * R\rfloor^{\rho}_{\emptyset} \subseteq \lfloor [\![\mathsf{stabilize}(Q)]\!] * R\rfloor^{\rho}_{\emptyset}$$

as desired. $\square$

3.9.1.2 *Soundness of Atomic Statement Specifications*

Lemma 127 (Soundness of Atomic Write)**.** For all $\rho \in R$, $\gamma \in \Gamma$ and σ,

$$[\![R;\Gamma \vdash_S \langle \mathsf{e}_1 \mapsto \_\rangle\ [\mathsf{e}_1] := \mathsf{e}_2\ \langle \mathsf{e}_1 \mapsto \mathsf{e}_2\rangle : \mathsf{Atomic}]\!]\,(\rho,\gamma,\sigma)$$

Proof. There are two possible execution steps for the atomic write. If e_1 does not denote a valid address, the write does not succeed, and the code steps according to

$$E \vdash (\sigma, [\mathsf{e}_1] := \mathsf{e}_2) \xrightarrow{\lightning} (\mathtt{skip}, \sigma)$$

and we have to argue that for any $s \in [\![S]\!]$ and stable frame R

$$[\![\lightning]\!]\,(\lfloor [\![R;\Gamma \vdash \mathsf{e}_1 \mapsto \_ : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma) * R\rfloor^{\rho}_{s}) \subseteq$$

$$\lfloor [\![R;\Gamma \vdash \mathsf{e}_1 \mapsto \mathsf{e}_2 : \mathsf{Assn}]\!]\,(\rho,\gamma,\sigma) * R\rfloor^{\rho}_{s}$$

which is trivial as any heap satisfying the precondition will satisfy that $[\![\mathsf{e}_1]\!]_\sigma \in \mathit{Addr}$, which is a contradiction. There are hence no heaps satisfying the precondition, and the inclusion in the semantic action judgment is thus vacuously satisfied.

Hence, there is only one transition allowed by the operational semantics. The code $[\mathsf{e}_1] := \mathsf{e}_1$ can step according to

$$\frac{[\![\mathsf{e}_1]\!]_\sigma \in \mathit{Addr}}{E \vdash (\sigma, [\mathsf{e}_1] := \mathsf{e}_2) \xrightarrow{\mathtt{write}([\![\mathsf{e}_1]\!]_\sigma, [\![\mathsf{e}_2]\!]_\sigma)} (\sigma, \mathtt{skip})}$$

We thus have to argue that the following holds for all $s \in [\![S]\!]$ and stable frames R:

$$[\![\mathtt{write}([\![\mathsf{e}_1]\!]_\sigma, [\![\mathsf{e}_2]\!]_\sigma)]\!] \left(\lfloor [\![R; \Gamma \vdash \mathsf{e}_1 \mapsto \_ : \mathsf{Assn}]\!] (\rho, \gamma, \sigma) * R \rfloor_s^\rho\right) \subseteq$$

$$\lfloor [\![R; \Gamma \vdash \mathsf{e}_1 \mapsto \mathsf{e}_2 : \mathsf{Assn}]\!] (\rho, \gamma, \sigma) * R \rfloor_s^\rho$$

We proceed according to definition. Suppose a stable frame R. And suppose a heap h in

$$\lfloor [\![\mathsf{e}_1 \mapsto \_]\!] (\rho, \gamma, \sigma) * R \rfloor_\emptyset^\rho .$$

We now have to argue that

$$[\![\mathtt{write}([\![\mathsf{e}_1]\!]_\sigma, [\![\mathsf{e}_2]\!])]\!] (h)$$

lies in

$$\lfloor [\![\mathsf{e}_1 \mapsto \mathsf{e}_2]\!] (\rho, \gamma, \sigma) * R \rfloor_\emptyset^\rho .$$

Since $[\![\mathsf{e}_1]\!]_\sigma$ is an address by assumption, we know that the heap update is successful, and we know

$$h[[\![\mathsf{e}_1]\!]_\sigma \mapsto [\![\mathsf{e}_2]\!]_\sigma]$$

is in the assertion

$$[\![R; \Gamma \vdash \mathsf{e}_1 \mapsto \mathsf{e}_2 : \mathsf{Assn}]\!] (\rho, \gamma, \sigma)$$

and hence in its erasure. □

Lemma 128 (Soundness of Atomic Read)**.** For all $\rho \in [\![R]\!]$, $\gamma \in [\![\Gamma]\!]$ and σ,

$$[\![R; \Gamma \vdash_S \langle \mathsf{e} = n * n \mapsto v \rangle\ \mathtt{x} := [\mathsf{e}]\ \langle \mathtt{x} = v * n \mapsto v \rangle : \mathsf{Atomic}]\!] (\rho, \gamma, \sigma)$$

Proof. There are two possible transitions for an atomic write, depending on whether the expression e denotes a valid address or not.

In the case that $[\![\mathsf{e}]\!]_\sigma \notin \mathit{Addr}$, the code transitions according to

$$E \vdash (\sigma, \mathtt{x} := [\mathsf{e}]) \xrightarrow{\lightning} (\mathtt{skip}, \sigma)$$

and we have to argue that for any $s \in [\![S]\!]$ and stable frame R we have

$$[\![\lightning]\!] \left(\lfloor [\![R;\Gamma \vdash \mathsf{e} = n * n \mapsto v : \mathsf{Assn}]\!] \, (\rho,\gamma,\sigma) * R \rfloor_s^\rho\right) \subseteq$$

$$\lfloor [\![R;\Gamma \vdash \mathtt{x} = v * n \mapsto v : \mathsf{Assn}]\!] \, (\rho,\gamma,\sigma) * R \rfloor_s^\rho$$

which is trivial as any heap satisfying the precondition will satisfy that $[\![\mathsf{e}]\!]_\sigma \in \mathit{Addr}$, which is a contradiction. There are hence no heaps satisfying the precondition, and the inclusion in the semantic action judgment is thus vacuously satisfied.

Hence, there is only one transition allowed by the operational semantics. The code $\mathtt{x} := [\mathsf{e}]$ can step as follows for some value v':

$$\frac{[\![\mathsf{e}]\!]_\sigma \in \mathit{Addr}}{E \vdash (\sigma, \mathtt{x} := [\mathsf{e}]) \xrightarrow{\mathtt{read}([\![\mathsf{e}]\!]_\sigma, v')} (\sigma[\mathtt{x} \mapsto v'], \mathtt{skip})}$$

Now it remains to show that

$$[\![\mathtt{read}([\![\mathsf{e}]\!]_\sigma, v')]\!] \left(\lfloor [\![R;\Gamma \vdash \mathsf{e} = n * n \mapsto v : \mathsf{Assn}]\!] \, (\rho,\gamma,\sigma) * R \rfloor_s^\rho\right) \subseteq$$

$$\lfloor [\![R;\Gamma \vdash \mathtt{x} = v * n \mapsto v : \mathsf{Assn}]\!] \, (\rho,\gamma,\sigma) * R \rfloor_s^\rho$$

We proceed directly by definition of the semantic action judgment. Suppose a stable R, and suppose further a heap h in

$$\lfloor [\![R;\Gamma \vdash \mathsf{e} = n * n \mapsto v : \mathsf{Assn}]\!] \, (\rho,\gamma,\sigma) * R \rfloor_\emptyset^\rho .$$

We know that

$$h([\![\mathsf{e}]\!]_\sigma) = v,$$

hence $v' = v$. Then, by the action interpretation,

$$[\![\mathtt{read}([\![\mathsf{e}]\!]_\sigma, v)]\!] \, (h) = \{h\}.$$

Hence it suffices to show that

$$h \in \lfloor [\![R;\Gamma \vdash \mathtt{x} = v * n \mapsto v : \mathsf{Assn}]\!] \, (\rho,\gamma,\sigma[\mathtt{x} \mapsto v]) \rfloor_\emptyset^\rho .$$

By calculation, the semantics of the postcondition, it suffices to show that h satisfies that $\sigma[\mathtt{x} \mapsto v](\mathtt{x}) = v$, which is trivially satisfied by h, and $h(n) = v$, which it does, as we know $h([\![\mathsf{e}]\!]_\sigma) = v$ and $[\![\mathsf{e}]\!]_\sigma = n$.

Hence, h is in the erasure of that assertion. $\square$

Lemma 129 (Soundness of CAS)**.** For all $\rho \in [\![R]\!]$, $\gamma \in [\![\Gamma]\!]$ and σ,

$$\begin{gathered}[\![R;\Gamma \vdash_S \langle \mathsf{e}_1 = a * a \mapsto v * \mathsf{e}_2 = \mathit{old} * \mathsf{e}_3 = \mathit{new} \rangle \\ \mathtt{x} := \mathtt{CAS}(\mathsf{e}_1,\mathsf{e}_2,\mathsf{e}_3) \\ \langle (\mathtt{x} \neq 0 * v = \mathit{old} * a \mapsto \mathit{new}) \vee (\mathtt{x} = 0 * v \neq \mathit{old} * a \mapsto v) \rangle]\!](\rho,\gamma,\sigma)\end{gathered}$$

Proof. Whether the CAS operation succeeds or not is independent of whether the CAS operation completes successfully or not. This depends on whether $\mathtt{e}_1$ denotes a legal address.

In the case that $[\![\mathtt{e}_1]\!]_\sigma \notin \mathit{Addr}$, the code transitions according to

$$E \vdash (\sigma, \mathtt{x} := \mathtt{CAS}(\mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3)) \xrightarrow{\lightning} (\mathtt{skip}, \sigma)$$

and we have to argue that for any $s \in S$ and stable assertion R

$$[\![\lightning]\!]\,(\lfloor [\![R; \Gamma \vdash \mathtt{e}_1 = a * a \mapsto v * \mathtt{e}_2 = \mathit{old} * \mathtt{e}_3 = \mathit{new} : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma) * R \rfloor_s^\rho) \subseteq$$

$$\lfloor [\![R; \Gamma \vdash (\ldots) \vee (\ldots) : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma) * R \rfloor_s^\rho$$

which is trivial as any heap satisfying the precondition will satisfy that $[\![\mathtt{e}]\!]_\sigma \in \mathit{Addr}$, which is a contradiction. There are hence no heaps satisfying the precondition, and the inclusion in the semantic action judgment is thus vacuously satisfied.

Hence, there is only one transition allowed by the operational semantics, namely that the CAS operation completes. The code can step as follows for some value v':

$$\frac{[\![\mathtt{e}_1]\!]_\sigma \in \mathit{Addr}}{E \vdash (\sigma, \mathtt{x} := \mathtt{CAS}(\mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3)) \xrightarrow{\mathtt{CAS}(b, [\![\mathtt{e}_1]\!]_\sigma, [\![\mathtt{e}_2]\!]_\sigma, [\![\mathtt{e}_3]\!]_\sigma)} (\sigma[\mathtt{x} \mapsto b], \mathtt{skip})}$$

for some value b. We proceed in two analogous cases according to whether $b = 0$ - we show the negative case in which the CAS succeeds.

We have to show that for any $s \in [\![S]\!]$ and stable frames R

$$\begin{aligned} &[\![\mathtt{CAS}(b, [\![\mathtt{e}_1]\!]_\sigma, [\![\mathtt{e}_2]\!]_\sigma, [\![\mathtt{e}_3]\!]_\sigma)]\!] \\ &\qquad (\lfloor [\![R; \Gamma \vdash \mathtt{e}_1 = a * a \mapsto v * \mathtt{e}_2 = \mathit{old} * \mathtt{e}_3 = \mathit{new} : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma) * R \rfloor_s^\rho) \\ &\subseteq \lfloor [\![R; \Gamma \vdash (\ldots) \vee (\ldots) : \mathsf{Assn}]\!]\,(\rho, \gamma, \sigma) * R \rfloor_s^\rho \end{aligned}$$

We proceed directly by definition of the semantic action judgment. Suppose a heap h that lies in the initial assertion.

We know $[\![\mathtt{e}_1]\!]_\sigma \in \mathit{Addr}$, and furthermore, we know that $[\![\mathtt{e}_1]\!]_\sigma \in \mathrm{dom}(h)$. We also know that $h([\![\mathtt{e}_1]\!]_\sigma) = v$. Hence, the interpretation of the heap effect depends on whether $v = \mathit{old}$ or not. In this case it must since we know by assumption that $b \neq 0$.

If it does, $[\![\mathtt{CAS}([\![\mathtt{e}_1]\!]_\sigma, [\![\mathtt{e}_2]\!]_\sigma, [\![\mathtt{e}_3]\!]_\sigma, b)]\!]\,(h) = h[[\![\mathtt{e}_1]\!]_\sigma \mapsto [\![\mathtt{e}_2]\!]_\sigma]$. Hence, we need to show that, knowing $b \neq 0$,

$$(h[[\![\mathtt{e}_1]\!]_\sigma \mapsto [\![\mathtt{e}_2]\!]_\sigma], a) \in$$

$$[\![R; \Gamma \vdash (\mathtt{x} \neq 0 * v = \mathit{old} * a \mapsto \mathit{new}) \vee (\mathtt{x} = 0 * v \neq \mathit{old} * a \mapsto v) : \mathsf{Assn}]\!]$$

$$(\rho, \gamma, \sigma[\mathtt{x} \mapsto b])$$

By the semantics of assertions, it is sufficient to demonstrate that it lies in one of the disjuncts, where we obviously choose to show

$$(h[[\![\mathtt{e}_1]\!]_\sigma \mapsto [\![\mathtt{e}_2]\!]_\sigma], a) \in [\![R; \Gamma \vdash \mathtt{x} \neq 0 * v = \mathit{old} * a \mapsto \mathit{new} : \mathsf{Assn}]\!]$$

$$(\rho, \gamma, \sigma[\mathtt{x} \mapsto b])$$

We know $b \neq 0$, $[\![\mathsf{e}_1]\!]_\sigma = a$ and $[\![\mathsf{e}_2]\!]_\sigma = v$, hence the three constraints are immediate.

The alternate case is analogous. □

3.9.1.3 *Soundness of Program Specifications*

Lemma 130 (Soundness of Program Specifications)**.** Given a well-typed program specification $\vdash \vec{r}; \vec{\mathtt{f}}$, assume for each $\mathtt{f}_i$ that, for all $n \in \mathbb{N}$,

$$n \in [\![\vec{r}; \lceil \vec{\mathtt{f}} \rceil \vdash \mathtt{f}_i : \mathsf{FunctionSpec}]\!] \,.$$

Then, $[\![\vdash \vec{r}; \vec{\mathtt{f}} : \mathsf{ProgramSpec}]\!]$ holds.

Proof. We are to show that, for any given $n \in \mathbb{N}$, $\lfloor \vec{\mathtt{f}} \rfloor \vDash_n \lceil \vec{\mathtt{f}} \rceil$.

By assumption, for every function spec $\mathtt{f}_i = \mathsf{g}(\vec{\mathtt{x}})(\Gamma, \vec{y})\{P\}\mathsf{s}\{r.Q\}$, for any n and $\gamma \in [\![\Gamma, \vec{y} : \mathsf{Val}]\!]$, $\rho \in [\![R]\!]$,

$$n \in \left[\!\!\left[\vec{r}; \lceil \vec{\mathtt{f}} \rceil ; \Gamma, \vec{y} : \mathsf{Val} \vdash \left\{P * (\vec{\mathtt{x}} = \vec{y})\right\} \mathsf{s} \left\{\forall r.Q \mid r.Q\right\} : \mathsf{Spec}\right]\!\!\right] (\rho, \gamma).$$

This in particular means that for any E such that for any $n' < n$ we have $E \vDash^{\rho}_{n'} \lceil \vec{\mathtt{f}} \rceil$, it holds for any $\sigma \in \mathit{Stack}$ that

$$\begin{aligned}\mathsf{safe}_n(E, &[\![\mathrm{dom}(R); \Gamma, \vec{y} : \mathsf{Val} \vdash P * (\vec{\mathtt{x}} = \vec{y}) : \mathsf{Assn}]\!] (\rho, \gamma, \sigma),\\ &\sigma,\\ &\mathtt{s},\\ &\lambda\sigma'. [\![\mathrm{dom}(R); \Gamma, \vec{y} : \mathsf{Val} \vdash \forall r.Q : \mathsf{Assn}]\!] (\rho, \gamma, \sigma'),\\ &\lambda\sigma'.\lambda v. [\![\mathrm{dom}(R); \Gamma, \vec{y} : \mathsf{Val}, r : \mathsf{Val} \vdash Q : \mathsf{Assn}]\!] (\rho, \gamma[r \mapsto v], \sigma'))\end{aligned}$$

This is precisely Definition 103, of environment/function spec agreement, and in summary we have

$$\forall E. E \vDash^{\rho}_{n'} \lceil \vec{\mathtt{f}} \rceil \Rightarrow E \vDash^{\rho}_{n} \mathsf{g} : (\Gamma, \vec{y})\{P\}\{r.Q\}$$

Since we have this for every $\mathsf{g} \in \mathrm{dom}(\lceil \vec{\mathtt{f}} \rceil)$, we have that

$$\forall E. E \vDash^{\rho}_{n'} \lceil \vec{\mathtt{f}} \rceil \Rightarrow E \vDash^{\rho}_{n} \lceil \vec{\mathtt{f}} \rceil \,.$$

If we instantiate E to $\lfloor \vec{\mathtt{f}} \rfloor$ we obtain precisely

$$\lfloor \vec{\mathtt{f}} \rfloor \vDash^{\rho}_{n'} \lceil \vec{\mathtt{f}} \rceil \Rightarrow \lfloor \vec{\mathtt{f}} \rfloor \vDash^{\rho}_{n} \lceil \vec{\mathtt{f}} \rceil \,.$$

Since we have this for any $n' < n$, and for any n, we can generalize to

$$\forall n. (\forall n' < n. \lfloor \vec{\mathtt{f}} \rfloor \vDash^{\rho}_{n'} \lceil \vec{\mathtt{f}} \rceil) \Rightarrow \lfloor \vec{\mathtt{f}} \rfloor \vDash^{\rho}_{n} \lceil \vec{\mathtt{f}} \rceil$$

which by induction lets us conclude

$$\forall n. \lfloor \vec{\mathtt{f}} \rfloor \vDash^{\rho}_{n} \lceil \vec{\mathtt{f}} \rceil$$

as desired. □

Theorem 131 (Soundness of Program Logic)**.** If $\vdash \vec{r}; \vec{\mathtt{f}} : \mathsf{ProgramSpec}$ is derivable in the program logic, then $[\![\vdash \vec{r}; \vec{\mathtt{f}} : \mathsf{ProgramSpec}]\!]$ holds.

Part II

Distributed Protocol Combinators

4

DPC: Protocol Combinators for Modeling, Testing and Execution of Distributed Systems

Distributed systems are hard to get right, model, test, debug, and teach. Their textbook definitions, typically given in a form of replicated state machines, are concise, yet prone to introducing programming errors if naïvely translated into runnable implementations.

In this work, we present *Distributed Protocol Combinators* (DPC), a declarative programming framework that aims to bridge the gap between specifications and runnable implementations of distributed systems, and facilitate their modeling, testing, and execution. DPC builds on the ideas from the state-of-the art logics for compositional systems verification. The contribution of DPC is a novel family of program-level primitives, which facilitates construction of larger distributed systems from smaller components, streamlining the usage of the most common asynchronous message-passing communication patterns, and providing machinery for testing and user-friendly dynamic verification of systems. This paper describes the main ideas behind the design of the framework and presents its implementation in Haskell. We introduce DPC through a series of characteristic examples and showcase it on a number of distributed protocols from the literature.

This paper extends our preceeding conference publication [5] with an exploration of randomised testing for protocols and their implementations, and an additional case study demonstrating bounded model checking of protocols.

4.1 Introduction

Distributed fault-tolerant systems are at the heart of modern electronic services, spanning such aspects of our lives as healthcare, online commerce, transportation, entertainment and cloud-based applications. From engineering and reasoning perspectives, distributed systems are amongst the most complex pieces of software being developed nowadays. The complexity is not only due to the intricacy of the underlying protocols for multi-party interaction, which should be resilient to execution faults, packet loss and corruption, but also due to hard performance and availability requirements [14].

The issue of system correctness is traditionally addressed by employing a wide range of *whole-system* testing methodologies, with more recent advances in integrating techniques for formal verification into the system development process [29, 36, 59]. In an ongoing effort of developing a *verification* methodology enabling the *reuse* of formal proofs about distributed systems in the context of an open world, the Disel logic, built on top of the Coq proof assistant [18], has been proposed as the first framework for mechanised verification of distributed systems, enabling modular proofs about protocol composition [71, 79].

The main construction of Disel is a *distributed protocol* $\mathcal{P}$—an operationally described replicated *state-transition system* (STS), which captures the shape of the state of each node in the system, as well as what it *can* or *cannot* do at any moment, depending on its state. Even though a protocol $\mathcal{P}$ is not an executable program and cannot be immediately run, one can still use it as an *executable specification* of the system, in order to prove the system's intrinsic properties. For instance, reasoning at the level of a protocol, one can establish that a property I : $\mathsf{SystemState} \rightarrow \mathsf{Prop}$ is an inductive invariant *wrt.* a protocol $\mathcal{P}$.[1] A somewhat simplified main judgement of Disel, $\mathcal{P} \vdash c$, asserts that an actual system implementation c will *not* violate the operational specification of $\mathcal{P}$. Therefore, if this holds, one can infer that any execution of a program c, will not violate the property I, proved for protocol $\mathcal{P}$. Disel also features a full-blown program logic, implemented as a Hoare Type Theory [57], which allows one to ascribe pre- and post-conditions to distributed programs, enforcing them via Coq's dependent types, at the expense of frequently requiring the user to write lengthy proof scripts.

While expressive enough to implement and verify, for instance, a crash-recovery service on top of a Two-Phase Commit [71], unfortunately, Disel, as a systems *implementation* tool, is far from being user-friendly, and is not immediately applicable for rapid prototyping of composite distributed systems, their testing and debugging. Neither can one use it for teaching without assuming students' knowledge of Coq and Separation Logic [62]. Furthermore, system implementations in Disel must be encoded in terms of low-level send/receive primitive, obscuring the high-level protocol design.

In this work, we give a practical spin to Disel's main idea—disentangling protocol *specifications* from runnable, possibly highly optimised, systems *implementations*, making the following contributions:

- We distil a number of high-level distributed interaction patterns, which are common in practical system implementations, and capture them in a form of a novel family of *Distributed Protocol Combinators* (DPC)—a set of versatile higher-order programming primitives. DPC allow one to implement systems concisely, while

1 Examples of such properties include global-systems invariants, used, in particular, to reason about the whole system reaching a consensus [65, 67].

still being able to benefit from protocol-based specifications for the sake of testing and specification-aware debugging.

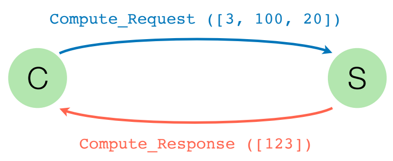
- We implement DPC in Haskell, providing a set of specification and implementation primitives, parameterised by a monadic interface, which allow for multiple interpretations of protocol-oriented distributed implementations.
- We provide a rich toolset for testing, running, and visual debugging of systems implemented via DPC:
  - visual exploration tools for tracing protocol execution.
  - tools for guided random execution of *implementations*, enabling testing implementations against their protocol specifications as properties in the sense of Claessen and Hughes [17].
  - a language for expressing protocol invariants, and tools for checking them on the (bounded) state space of the protocol.
- We showcase DPC on a variety of distributed systems, ranging from a simple RPC-based cloud calculator and its variations, to distributed locking [43], Two-Phase Commit [35], and Paxos consensus [46, 47].

4.2 Specifying and Implementing Systems with DPC

In this work, we focus on message-passing asynchronous distributed systems, where each node maintains its internal state while interacting with others by means of sending and receiving messages. That is, the messages, which can be sent and received at any moment, with arbitrary delays, drops, and rearrangements, are the only medium of communication between the nodes. DPC takes the common approach of thinking of message-passing systems as shared-memory systems, in which each message in transit is allocated in a virtual shared "message soup", where it lingers until it is delivered to the recipient [71, 80].

The exact implementation of the *per-node* internal state might differ from one node to another, as it is virtually unobservable by other participants of the system. However, in order for the whole system to function correctly, it is required that each node's behaviour would be at least coherent with some notion of *abstract state*, which is used to describe the interaction protocol.

In the remainder of this section, we will build an intuition of designing a system "top-down". We will start from its specification in terms of a protocol that defines the abstract state and governs the message-passing discipline, going all the way down to the implementation that defines the state concretely and possibly combines several protocols together. For this, we use a standard example of a distributed calculator.



4.2.1 *Describing Distributed Interaction*

In a simple cloud calculator, a node takes one of two possible roles: that of a *client* or that of a *server*. A client may send a request along with data to be acted upon to the server (e. g., a list of numbers `[3, 100, 20]` to compute the sum of), and the server in turn responds with the result of the computation, as shown on the diagram on the right. For uniformity of implementation, all message payloads, including the response of the server, are lists of integers. Notice that this description does not restrict e. g., the order in which a server must process incoming requests from the clients, leaving a lot of room for potential optimisations on the implementation side.

In order to capture the behavioural contract describing the interaction between clients and servers, we need to be able to outlaw some unwelcome communication scenarios. For instance, in our examples, it would be out of protocol for the server to respond with a wrong answer (in general an issue of safety) or to the wrong client (in general an issue of security). A convenient way to restrict the communication rules between distributed parties is by introducing the *abstract state* describing specific "life stages" of a client and a server, as well as associated messages that trigger changes in this state—altogether forming an STS, a well-known way to abstractly describe and reason about distributed protocols [48, 49].

Let us now describe our calculator protocol as a collection of coordinated transition systems. The client's part in the protocol originates in a state `ClientInit` containing the input it is going to send to the server, as well as the server's identity. From this state, it can send a message to server `s` with the payload `[3, 100, 20]`. It then must wait, in a blocking state, for a response from the server.[2] Upon having received the message, the client proceeds to a third and final state, `ClientDone`. From here, no more transitions are possible, and the client's role in the protocol is completed. A schematic outline of the client protocol is depicted in Fig. 4.1 (a).

In our simplified scenario, the protocol for the server (Fig. 4.1, (b)) can be captured by just one state, `ServerReady`, so that receiving the request and responding to it with a correct result is observed as "atomic" by other parties, and hence, is denoted by a single composite transition. In other words, at the specification level, the server immediately reacts to the request by sending a response.

2 Remember that this is a specification-level blocking, the implementation can actually do something useful in the same time, just not (observably) related to this protocol!

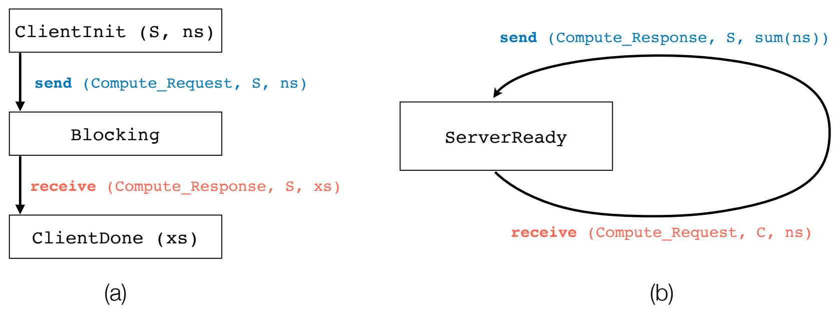

ClientInit (S, ns)
send (Compute_Request, S, ns)
Blocking
receive (Compute_Response, S, xs)
ClientDone (xs)
(a)
send (Compute_Response, S, sum(ns))
ServerReady
receive (Compute_Request, C, ns)
(b)

Figure 4.1: State transitions for a client (a) and a server (b) in the calculator protocol.

Notice that the protcol places no demands on the number of clients, servers or unrelated nodes in the network, nor does it restrict the number of instances of the protocol are running in a given network. The specification is "local" to the parties involved (which in general can number arbitrarily many).

This “request/respond” communication pattern is so common in distributed programming that it is worth making explicit. We will refer to this pattern as a pure *remote procedure call* (RPC) and take it as our first combinator for protocol-based implementation of distributed systems.

4.2.2 *Specifying the Protocol*

We can capture the RPC-shaped communication in DPC by first enumerating all possible states of nodes in the protocol in a single data type. For the calculator, the states can be directly translated from the description above to the following Haskell data type:

```
data S = ClientInit NodeID [Int]
       | ClientDone [Int]
       | ServerReady
```

`NodeID` is a type synonym for `Int`, but any type with equality would serve. `ClientInit` contains the name of the server and the list to sum. `ClientDone` contains the response from the server. Next, we describe the only kind of exchange that takes place in a network of clients and servers communicating by following the RPC discipline. We do so by specifying when a client can produce a request in a protocol, and how the server computes the response. Perhaps, a bit surprisingly, no more information is needed, as the pattern dictates that clients await responses from servers, and the server responds immediately. This is the reason why we need only enumerate two states for the client, eliding the one for blocking, as per Fig. 4.1 (a): the framework adds the third during execution by wrapping the states in a type with an additional `Blocking` constructor.[3] The following definition of `compute` outlines the specification of the protocol's STSs:

```
compute :: Alternative f ⇒ ([Int] → Int) → Protlet f S
compute f = RPC "compute" clientStep serverStep
  where
    clientStep s = case s of
      ClientInit server args → pure (server, args, ClientDone)
      _ → empty
    serverStep args s = case s of
      ServerReady → pure ([f args], ServerReady)
      _ → empty
```

As per its type, `compute` takes a client-provided function of type `[Int] → Int`, which is used by the server to perform calculations. The result of `compute` is of type `Protlet f S`, where `S` is the data type of our STS states defined just above and `f` is a type-former encapsulating possible non-determinism in a protocol specification. This is is a standard pattern for programming "with effects" in the pure fragment of Haskell. Later constructions will make integral use of non-determinism to, e. g., decide on the next transition depending on the external inputs, and the parameter `f` serves to restrict what notion of non-determinism is used in the definition of protocols.[4] For now, the result of `compute` is entirely deterministic, but must still be "wrapped" in the constructors of the non-deterministic effect, here `pure` and `empty` indicating a single result and the absence of results, respectively.

Protlets (*aka.* "small protocols") are the main building blocks of our framework. A distributed protocol can be thought of as a family of protlets, each of which corresponds to a logically independent piece of functionality and can be captured by a fixed interaction pattern between nodes. In a system, each node can act according to one or more protlets, executing the logic corresponding to them sequentially, or in parallel. For this example, there is just the one exchange of messages, so a single protlet makes for the complete protocol description.

Our framework provides several constructors to build protlets from the data type description for the protocol state space and the operational semantics of its transitions. In the example above, `RPC` is a data constructor, which encodes the protlet logic by means of two functions. Its first argument, `clientStep`, prescribes that from `ClientInit` state, a node can send `args` to node `server`, and the response payload is later wrapped via `ClientDone` to form the succesor state. The second argument, `serverStep`, says that the state `ServerReady` can serve a request in one step: receiving `args` and responding with `f args` in a singleton list, continuing in the same state. We have now completely captured the above intuitions and transition system of the calculator in less than ten lines of Haskell.

3 See the discussion of executing specification in Section 4.3
4 One can think of any protocol, whose diagram has a fork, as non-deterministic.

4.2.3 *Executing the Specification*

The immediate benefits of having an executable operational specification of a protocol is to be able to run it, locally and without needing full deployment across a network, ensuring that it satisfies basic sanity checks and more complex invariants.

The execution model for protlets is a small-step operational semantics, with the granularity of transitions being that of the involved protlets. We take as machine configurations the entire network of nodes and their abstract states.

In case several protlets of a similar shape are involved (e. g., a node is involved in two or more RPCs), we distinguish them by introducing protlet labels, a solution that is standard for program logics for concurrency [24, 70]. Having introduced protlet labels, we can logically partition the local state of each node along the protlet instance space, maintaining a per protlet instance local state portion for each node. We represent this operational machine configuration as the datatype `SpecNetwork`, which is an instantiation of an abstract structure of a network state `NetworkState`, representing the global environment and a local state for each node in the network. The generality allows code reuse across the framework. For execution, the global environment is a protocol specification for each instance label. The per-node state consists of a protlet state for each protocol instance, and a message queue. The intention is that the operational semantics updates one node's one protlet's state at a time.

```
data NetworkState global local = NetworkState {
    _globalState :: global,
    _localStates :: Map NodeID local
  }

type SpecNetwork f s =
  NetworkState (Map Label [Protlet f s])
               (Map Label (NodeState s), [Message])
```

The following describes a network for the calculator protocol with two nodes (identified by `0` and `1`), both running just one protlet (labelled with `0`), for the input for the example from Section 4.2.1:

```
addNetwork :: Alternative f ⇒ SpecNetwork f S
addNetwork = initializeNetwork nodeStates protocols
  where
    nodeStates = [ (server, [(0, ServerReady)])
                 , (client, [(0, ClientInit server [3, 100, 20])]) ]
    protocols  = [ (0, [compute sum]) ]

    server, client :: NodeID
    (server, client) = (0, 1)
```

Here, `initializeNetwork` is a convenience function to initialize the `SpecNetwork` datastructures from human writeable descriptions in the form of association lists.

In any given network configuration, many actions can be possible. A node may be ready to initiate an RPC, or it (or another node entirely) might be ready to receive a message—many such actions may be enabled and relevant at once.[5] As the purpose of running the specification is to trace the possible behaviors in the protocol, we choose the next action to execute in the network by leaving the resolution to the *user* of the semantics. To do so, we implement the executable small-step relation as a monad-parameterised function capturing the possibility of non-determinism (hence `Alternative f`). This makes the implementation of the operational semantics simple, yet general, as it just needs to describe an `f`-ary choice or `f`-full collection of possible transitions at each step:

```
step :: (Monad f, Alternative f) ⇒ SpecNetwork f s → f (SpecNetwork f s)
```

The network can be "run" by iterating this small-step execution function with a suitable instance of `f`, a standard construction in implementation of a non-determinism in monadic interpreters.

For example, we can instantiate the non-determinism to the classic choice of the list monad [52], which leads to enumerating every possible action. We can then iterate the function `step` by choosing an arbitrary transition, as captured by the `simulateNetworkIO` function used in the following interaction with the library, where we explore the "depth" of a single run of the protocol.

```
> length <$> simulateNetworkIO addNetwork
4
```

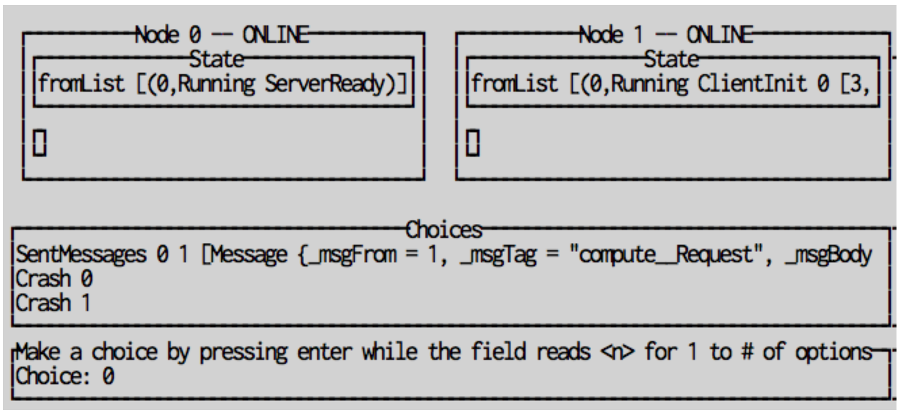
This is coherent with the first example we envisioned *wrt.* the protocol: there is (1) the initial state; (2) the state with the client awaiting response, but the message undelivered; (3) the state with the client waiting and the server having sent a response; and finally, (4) a terminal state with the client done.

The non-determinism can be similarly resolved by enumerating all possible paths through a protocol, up to a certain trace length if the execution space is not finite. If the state space of a network is finite, this can yield actual finite-space model checking procedures. In the following subsection, we will explore another alternative to resolving the non-determinism, yielding an unusual yet very useful execution method.

4.2.4 *Interactive Exploration with GUI*

By delegating the decision of which transition to follow to the user of an application that performs this simulation, we can allow the client of the framework to explore the network behaviour interactively. The DPC library provides a command-line GUI application facilitating interactive exploration of distributed networks step-by-step. Provided

5 And their abundance is precisely why reasoning about distributed systems is hard.



```
Node 0 — ONLINE
State
fromList [(0,Running ServerReady)]
[]

Node 1 — ONLINE
State
fromList [(0,Running ClientInit 0 [3,
[]

Choices
SentMessages 0 1 [Message {_msgFrom = 1, _msgTag = "compute_Request", _msgBody
Crash 0
Crash 1

Make a choice by pressing enter while the field reads <n> for 1 to # of options
Choice: 0
```

Figure 4.2: The interactive exploration tool, loaded with the calculator protocol.

an initial network specification like the one described previously, one can start the session by typing the following:

```
> runGUI addNetwork
```

This yields the interface displayed in Fig. 4.2. By choosing specific transitions in sequence, the user can evolve and inspect the network at each step of execution. This is useful for protocol design and debugging, and can help understand the dynamics of a protocol, and the kinds of communication patterns it describes.

For example, in Fig. 4.3 we show the subsequent prompt after showing the selection of Option 1:

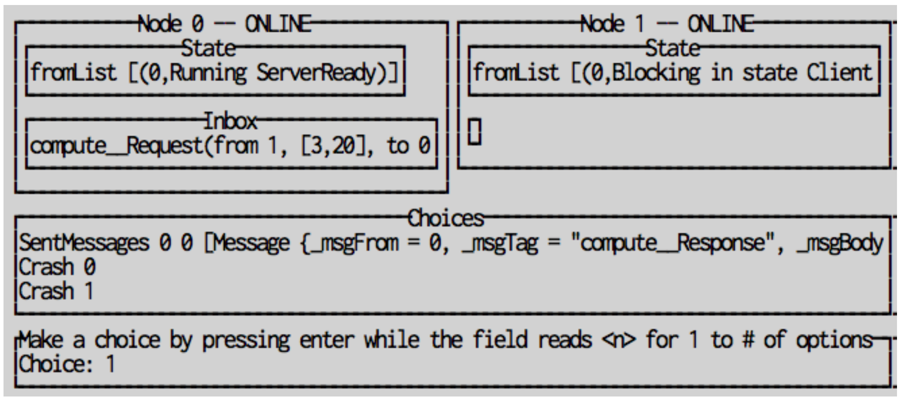
```
SentMessages 0 1 [Message {_msgFrom = 1, _msgTag = "compute_Request", . . .
```

`SentMessages` is a human readable piece of data that represents the option of sending in protocol instance `0`, from node `1` the message with sender `1` of tag `"compute_Request"`.

The format chosen is the debug serialization format provided by Haskell's **`Show`** and **`Read`** type classes for ease of experimenting: any data of the sort displayed to the user can be directly copied and used in scripts or command prompts. Here, the recipient and message content is elided for issues of screenspace, but as the window is enlarged, so is the depth of information provided to the client of the framework.

The state view then shows that Node `0` now has said message waiting for it in the soup, and Node `1` is now blocking. The user is then presented with subsequent possible choices, here the option for the calculator to receive the request and send the response in one atomic action, as dictated by the protocol.

Additionally, as can be seen in Fig. 4.2, in the interactive tool we enrich the possible transitions at every step with the possibility of a node to go off-line. In effect, it means it will stop processing messages, modelling a benign (non-byzantine) fault. Other nodes cannot observe this and will "perceive" the node as not responding. It is implemented



```
Node 0 — ONLINE
State
fromList [(0,Running ServerReady)]
Inbox
compute_Request(from 1, [3,20], to 0

Node 1 — ONLINE
State
fromList [(0,Blocking in state Client
[]

Choices
SentMessages 0 0 [Message {_msgFrom = 0, _msgTag = "compute_Response", _msgBody
Crash 0
Crash 1

Make a choice by pressing enter while the field reads <n> for 1 to # of options
Choice: 1
```

Figure 4.3: Choosing option 1 in the prompt from Fig. 4.2.

by eliding the actions performed by the off-line node when computing the set of possible actions. This, however, becomes very useful when we move to explore protocols that allow for partial responses among a collection of nodes, as in the case of crash-resilient consensus protocols.

4.2.5 *Protocol-Aware Distributed Implementations*

Distributed systems protocols serve as key components of some of the largest software systems in use. The actions taken in the protocol are governed by programs outside the key protocol primitives, so it is vital that implementations can integrate with software components in real general-purpose languages. We here present such a language with primitives for sending and receiving messages as an embedded domain-specific language (EDSL) in Haskell. This allows use of the entire Haskell toolkit in engineering efficient optimised implementations relying on distributed interaction.

Naturally, as implementations deviate from the protocols (in the way they, e. g., implement internal state), we want to ensure that the they still adhere to the protocol as specified. To achieve this, we introduce primitives for *annotating* implementations with protocol-specific assertions. These annotations can be ignored by execution-oriented interpretations aiming for efficiency rather than verification guarantees.

The following code implements a calculator server in plain Haskell using **do**-notation to sequence effectful computations. The effects are described by type class constraints on `m`, the monad used for sequencing: `MessagePassing` provides a `send` and `receive` primitive, and `ProtletAnnotations S` provide the `enactingServer` primitive over the state-space `S`. The type `S` is the data type defined in Section 4.2.2, and denotes the abstract state space we wish to relate to sub-computations in our implementation, as explained below.

```
addServer :: (ProtletAnnotations S m, MessagePassing m) ⇒ Label → m a
addServer label = loop
```

```
  where
    loop = do
      enactingServer (compute sum) $ do
        Message client _ args _ _ ← spinReceive [(label, "Compute__Request")]
        send client label "Compute__Response" [sum args]
      loop
```

By using type classes describing operations, we allow for several different interpretations of this code. For instance, by interpreting the `send` and `receive` as POSIX Socket operations, we obtain a subroutine in the IO Monad, Haskell's effectful fragment, that we can integrate into any larger development with no interpretive overhead. The `spinReceive` operation is defined using recursion and a primitive `receive` operation that attempts to receive an incoming message with a tag from amongst a list of candidate message tags in a non-blocking manner.

The body of `addServer` is annotated with a `(compute sum)` protlet, enforcing that the server responds to the client atomically (in terms of message passing) and to perform the **sum** function (or something observationally equivalent) on the supplied arguments. By bracketing the `receive` and `send` in the `enactingServer` primitive, the implementation declares its intent to conform to the server role of the RPC, as dictated by the protocol. Once we have a client to play the other role in the protocol, we will demonstrate how this intent can be checked dynamically. The message tags that appear in the code are *by convention* the tags used in the RPC protocol, i. e., the name of the protocol with a suffix indicating the role in the RPC that the message plays.

In contrast Disel and other static verification frameworks that enforce protocol adherence via (dependent) type systems (embedded in Coq or other proof assistants) [45, 71], we verify protocol properties dynamically. The tradeoff is that of coverage versus annotation and proof overhead. We can, through exploiting executable specifications, check that *a single run of a program* adheres to a protocol. Notice that `addServer` is, like the specification of the `compute` protlet, agnostic in the number and kinds of other nodes in the network. Its behaviour is locally and completely described by its implementation, and is segregated from interfering with unrelated protlets via the `label` parameter. We refer the reader to the development for a number of client component implementations.

Let us now reap the benefits of protocol-aware distributed programming enabled by DPC and *dynamically check* that the implementations do indeed follow the abstract protocols. We achieve this by interpreting the EDSL into a datatype of abstract syntax trees (AST) that makes it possible to inspect their evaluations at run time. We give a small-step structural operational semantics to this language, and, precisely like the exectuable specifications, lift the evaluation of a single program to that of an entire network of programs, by assigning each program a node identifier in the network, as show below. Here, Node 1 runs the client implementation and Node 0 runs the server.

```
addConf :: (ProtletAnnotations S m, MessagePassing m) ⇒ ImplNetwork m Int
addConf = initializeImplNetwork [
            (1, addClient 0 20 3 0)
          , (0, addServer 0)
          ]
```

The similarity to specification-level configurations is not incidental: `ImplNetwork` is another instantiation of `NetworkState`:

```
type ImplNetwork m a = NetworkState [Message] (m a)
```

Here, the global state (of type `ImplNetwork m Int`, with `m` constrained as in `addServer`/`addClient`) is just the message soup, and the node-local state is the program itself. An interpreter for such configurations is implemented by the following function:

```
traceRoundRobin :: ImplNetwork (AST s) a → [TraceAction s]
```

Here, the `AST` data type is the HOAS AST for message-passing implementations to be interpreted in. The result of running the network is a (possibly infinite but productive) list of `TraceAction`s. Trace actions describe "events" in the network: messages received and messages sent. We can simulate a full run of the network by using the trace actions to resolve the non-determinism of choices in the operational semantics.

For utility we here bake in a round robin schedule to ensure fair execution, but provide more general interpreters parametrised by a schedule and returning richer results, e.g.

```
runWithSchedule :: [NodeID] →
                   ImplNetwork (AST s) a →
                   [(TraceAction s, ImplNetwork (AST s) a)]
```

We can verify that our implementation indeed adheres to the desired protocol by the trace produced by `traceRoundRobin` on a network configuration, ensuring that (a) every observable action is compatible with the state that the node is supposed to be in, and (b) checking the messages expected from these states. For this, we implement yet another operational semantics, where the machine configuration is a protocol state for every node id, and the program is a trace of primitive actions. We here call it `checkTrace`. The interpreter faults if the current action is not applicable to the state, or sends or receives messages not prescribed by the specification. We can run the adherence checker on a *prefix* (e. g., of length 15) of the infinite trace as follows:

```
> checkTrace addNetwork $ take 15 $ traceRoundRobin addConf
Right ()
```

The result of `Right ()` indicates success: the trace did indeed conform to the protlet annotations of the program, assuming the initial state of the implementations in `addConf` conformed to an initial abstract state corresponding to the the network state of `addNetwork`.

What happens if we introduce a mistake in the implementation? For instance, what if we run the client implementation *twice*? We can illustrate this by altering `addConf` to demand the `addClient` to be run twice in succession:

```
addConf = initializeImplNetwork [
    (1, addClient 0 3 20 0 » addClient 0 100 11 0)
  , (0, addServer 0)
  ]
```

The checker then reports an error, as this is not allowed by the protocol: the client would have brought itself to the terminal state `ClientDone` by the first RPC, and, hence, cannot proceed.

```
> checkTrace addNetwork $ take 30 $ traceRoundRobin addConf
*** Exception: Node 1 expected to initiate rpc compute
               Node is in state: ClientDone [23]
```

In a different scenario, if we erroneously annotate the server as intending to serve a **`product`** function (instead of **`sum`**), we will fail protocol adherence, because the specification does not agree on the content of the messages.

```
addServer :: (ProtletAnnotations S m, MessagePassing m) ⇒ Label → m a
addServer label = loop
 where
   loop = do
     enactingServer (compute product) {- !!ERROR!! -} $ do
       ...

> checkTrace addNetwork $ take 15 $ checkTrace addConf
*** Exception: The server response did not follow the protocol from state
               ServerReady
                 Expected: [60]
                 Got: [23]
```

Of course, here we only observe the error because our *single* instantiation of the client's payload, `[20, 3]`, happened to disagree on the **`sum`** and **`product`** function. What if the payload was `[0,1]`?

By enriching dynamic testing with protocol adherence checks we believe we can achieve greater assurances of the correctness of our implementations without resorting to use full-blown verification frameworks [36, 71].

4.2.6 *Introducing Randomised Testing to Distributed Systems*

Naturally, the dynamic testing demonstrated in the preceeding section is only as good as the creativity and insight of the developer. The originators of QuickCheck observed that pure functional programs with executable specifications acting on first-order data is a natural setting for exploiting randomised testing: instead of carefully crafting pathological cases to demonstrate absence of errors, carefully craft a generator for random program input and write more programs to decide whether the program under test performs as expected [17].

DPC is a natural fit for this approach: we have an executable specification of a protocol, along with a re-interpretable DSL for implementing these protocols. This suggests that we generate random input data for the implementations, and use our executable specification as ground truth for correctness of implementations.

First, we parameterise the initial configurations of the specification and implementation execution configurations by the numbers that the client want operated on.

```
addNetwork :: Int → Int → SpecNetwork f S
addConf :: Int → Int → ImplNetwork m [Int]
```

With this in hand we can formulate universally quantified boolean properties (indexed boolean-valued expressions) that can then be evaluated on randomly generated inputs. At it's most basic, we wish the evaluation of the implemantation to conform to the specification, a property here formulated using `checkTrace` as described in Section 4.2.5:

```
prop_simpleAddNetwork :: Int → Int → Bool
prop_simpleAddNetwork x y =
  let trace = take 100 $ traceRoundRobin (addConf x y) in
    checkTrace (addNetwork x y) trace == Right ()
```

The prefix of `prop_` is a convention that allows for discovery of properties by the QuickCheck toolset. The function `traceRoundRobin` is a pure interpreter for the implementation language that schedules nodes in a fair round-robin fashion. QuickCheck can now help us exorcise bugs of the class previously identified as problematic for unit testing:

```
> quickCheck prop_simpleAddNetwork
*** Failed! (after 2 tests and 2 shrinks):
Exception:
  The server response did not follow the protocol from state: ServerReady
    Expected: [0]
    Got: [1]
0
1
```

QuickCheck reports that on payload `[0, 1]`, the server implementation violated the server specification: it replied to the client with 1 rather than 0 as (erroneously) dictated by the specification.

QuickCheck can also help us with the problem of systematically testing a class of errors unique to the setting of non-deterministic concurrent computation via message-passing, namely that of programs not accounting for all possible schedules. As the number of instructions per process increases, the number of possible schedules grows exponentially, and aggressively so. Corner cases are also difficult to foresee, so in lieu of formal verification, the possibility of randomly exercising possible schedules is worth pursuing. A schedule arises from the non-deterministic interleaving of threads, but a concrete schedule can be represented by a sequence of integers, in Haskell a value of type [NodeID], indicating the order of execution of the nodes in the distributed system. We generalize roundRobinTrace to traceSchedule, parametrised by the specific schedule to use.

```
arbitraryScheduleFor :: [NodeID] → Gen [NodeID]
arbitraryScheduleFor s = infiniteListOf (elements s)

prop_simpleAddNetworkArbSchedule :: Int → Int → Property
prop_simpleAddNetworkArbSchedule x y =
    forAll (arbitraryScheduleFor (nodes conf)) $ λschedule →
      let trace = take 100 $ fst <$> runWithSchedule schedule conf in
          checkTrace (addNetwork x y) trace == Right ()
  where
    conf = addConf x y
```

We use the forAll combinator to build a Property, in essence a generalization of a boolean valued expression to a function taking a random seed (and some additional configuration controlling the generation process). Here, forAll is used to explicitly supply the generator arbitraryScheduleFor to be used for generating traces as opposed to the implicit inference of appropriate generators for x and y via type classes (The existing instance for [Int] simple genereates a finite list of random integers). QuickCheck uses the convention of naming generators 'arbitrary'. This now let's us exercise the implementation for bugs arising due to pathological execution orders of each node in the network. It appears robust to arbitrary interleavings of execution:

```
> quickCheck (withMaxSuccess 10000 prop_simpleAddNetworkArbSchedule)
+++ OK, passed 10000 tests.
```

We believe we here have illustrated the applicability of randomised testing to build a dicipline of testing for distributed systems. By exploiting that the specification language of DPC is executable, we leverage existing technologies to give us a light-weight process for writing convincingly correct implementations of distributed components.

4.3 Framework Internals

4.3.1 *The Specification Language*

A full distributed system specification consists of a collection of nodes, each assigned a unique node identifier, and a collection of protlets for each instance label. A node owns local state, partitioned according to protocol instance labels. A protlet describes one exchange pattern between parties. A collection of protlets over the same state space then describe an entire protocol.

In the overview we saw the simplest protlet, the pure RPC, but through exploration of examples and case studies, we have discovered a number of such patterns, each more general than the previous. These are implemented as extensions to the `Protlet` data type. One such is the broadcast protlet, integral for describing multi-party protocols. We elide the other protlet constructors, which can be found in our implementation.

```
data Protlet f s =
   | RPC         String (ClientStep s) (ServerStep s)
   | Broadcast   String (Broadcast s)  (Receive f s) (Send f s)
   | ...
```

The component functions of the protlets reuse a number of common type abbreviations, here `ClientStep`, `Send` etc. All are at work in the above listing. This common structure unifies their implementation in the operational semantics. The expansion of, e. g., the `Broadcast` synonym is as follows:

```
type Broadcast s = s → Maybe ([(NodeID, [Int])], [(NodeID, [Int])] → s)
```

This models a "partial" function on states `s`, saying under which conditions a node can initiate a broadcast, by enumerating the recipients and the body of the messages to them, along with a continuation processing the received answers with their associated senders. This continuation is stored in the implicit blocking state during actual execution of the specification.

The specification language is given a non-deterministic operational semantics as described in Section 4.2.3. Recall the network step function:

```
step :: (Monad f, Alternative f) ⇒ SpecNetwork f s → f (SpecNetwork f s)
```

It is implemented by computing an `f`-full of possible transitions for every node in the network and combining the result of taking all possible transitions on the current network. The key operation of `step` is a dispatch on the current protocol state of a node:

```
case state of
  BlockingOn _ tag f nodeIDs k →
    resolveBlock label tag f nodeID inbox nodeIDs k
  Running s → do
    protlet ← fst <$> oneOf (_globalState Map.! label)
    stepProtlet nodeID s inbox label protlet
```

The constructors `BlockingOn` and `Running` are supplied by the framework. The first is used to track the terms under which a node is blocking: what message(s) it needs to continue and from whom. `resolveBlock` computes whether the conditions are met for the current node to continue.

Here, `_globalState` is the mapping of collections of protlets (i. e., a protocol) from instance labels. We then choose between protlets using `oneOf::[a] → f a`. The function `stepProtlet` dispatches control based on a case distinction on the protlet constructor: for example, here is the branch for the `Broadcast` protlet:

```
stepProtlet :: (Monad m, Alternative m) =>
  NodeID -> s -> [Message] -> Label -> Protlet m s ->  m (
      Transition s)
stepProtlet nodeID state inbox label protlet = case protlet of
  ...
  Broadcast name broadcast receive respond ->
    tryBroadcast label name broadcast nodeID state inbox <|>
                      -- (1)
    tryReceive label (name ++ "__Broadcast") receive nodeID
        state inbox <|>  -- (2)
    trySend label respond nodeID state inbox
                                      -- (3)
  ...
```

A node attempting to advance a protocol using the `Broadcast` protlet can do so if it is (1) a client ready to perform a broadcast; (2) a server ready to receive such a broadcast; or (3) a server that is ready to respond to a broadcast. The **`try`** functions all follow the same structure: check that the user-provided protlet component functions apply, and if so, generate an appropriate transition. The result of each call is combined using <|>, the choice operator for the `Alternative` instance for `m`. For instance, here is the signature of one such function for `Broadcast`:

```
tryBroadcast :: Alternative f ⇒ Label → String → Broadcast s →
                NodeID → s → [Message] → f (Transition s)
```

There is one such function for every protlet component function, five in total.

Interpretations of Protocols. As described in Section 4.2.3, the operational semantics of protocols can be instantiated to obtain different interpretations. We here look at bounded model checking mentioned in passing in the overview. We can use the **`List`** monad to enumerate all execution paths in a breadth-first manner:

```
simulateNetworkTraces :: SpecNetwork [] s → [[SpecNetwork [] s]]
```

This yields a list-of-lists where the nth list contains all possible states after n steps of execution, in a breadth first enumeration of the state space. Each constituent list of states is necessarily finite, but the list-of-lists need not be in the case of infinite network executions. By virtue of Haskell's lazy evaluation, such a computational object is useful. We can write a procedure that, given a trace, applies a boolean predicate at every step of the trace.

```
checkTraceInvariant :: Invariant m s Bool → m → [SpecNetwork f s] →
                       Maybe Int
```

The `Invariant` data type is an abbreviation for a boolean predicate on the type `s` that additionally takes some "meta-data" `m`, like "roles" in a protocol, needed to express the invariant. The procedure `checkTrace` returns **Nothing** to signify that there were no violations of the invariant, while it returns **Just** n to report that the nth state was the first state to violate the invariant. With this language of predicates we can build invariants and with the aforementioned checking procedure we can perform (bounded) checking that an invariant is in fact inductive (i. e., holds for each state). In the case of a finite state space, this amounts to real verification of inductive invariants. The most sophisticated example we have successfully specified is an inductive invariant for a Two-Phased Commit protocol [71], for which we refer the curious reader to the implementation.

4.3.2 *The Implementation Language*

The monadic langugage for message-passing programs is implemented as an EDSL in Haskell. This has the benefit of providing all the standard tools for writing Haskell programs; all the abstraction mechanisms and organisational principles are at hand to write sophisticated software, including lazy evaluation, higher-order functions, algebraic data types and more. By virtue of the modularity offered by the approach of EDSLs, it is straightforward to give multiple interpretations of such programs.

At the time of this writing DPC's implementation fragment came with three interpretations of the monadic interface:

1. The AST monad used for dynamic verification of implementation adherence of the implementations to protocols, and covered in detail in Section 4.2.5.
2. A shared-memory based interpretation where nodes are represented as threads, and message passing is performed by writing to shared message queues using non-blocking concurrency primitives.
3. An interpretation for distributed message passing.

In the third case (true distribution), we give an interpretation into **IO** computations performing message passing through POSIX Sockets. For this, each computation needs an "address book" mapping NodeIDs to physical addresses (concretely, IP adresses and ports). Additionally, each program will have access to a local mailbox, represented by a message buffer being filled by a local thread whose only function is to listen for messages. These two pieces of data are collected in a record of type NetworkContext. Computations running in such a context are idiomatically captured in a type synonym over the ReaderT monad transformer:

```
newtype SocketRunnerT m a = SocketRunnerT {
    runSocketRunnerT :: ReaderT NetworkContext m a }
```

What follows is the implementation of the send primitive in this particular instance of the message-passing interface:

```
instance MonadIO m ⇒ MessagePassing (SocketRunnerT m) where
  send to lbl tag body = do
    thisID ← this
    let p = encode $ Message thisID tag body to lbl
    peerSocket ← (!to) <$> view addressBook
    void . liftIO $ Socket.send peerSocket p mempty
```

The code for sending messages is, thus, implemented in a form of a Reader-like computation over an **IO**-capable monad m as indicated by the MonadIO constraint. It starts by building a Message containing the supplied tag, body, receiver (to) and label, along with the executing nodes ID, as supplied by another primitive, this. It then uses encode to serialize this message into bytestring p. Then, p is sent to the appropriate peerSocket, as resolved by the addressBook, using the **System**.Socket.Send operation from the POSIX Socket library for Haskell. The monadic glue code (and the rest of the Haskell toolkit) is interpreted by choosing an appropriate base monad for the interpretation, e. g., the **IO** monad. Ultimately, we build the following function for running the system:

```
defaultMain :: NetworkDescription → NodeID → SocketRunner a → IO ()
```

It takes a NetworkDescription, which maps NodeIDs to physical addresses, a NodeID with which to identify this node, and a computation in the above described interpretation of message passing programs. The result is an **IO** () computation that establishes (if run on each machine) a fully connected mesh network with every node in the supplied network description, and then proceeds to run the supplied computation, passing messages accordingly. This interpretation can be used to facilitate integration of DPC-based implementations with real Haskell code once they have been assured to comply with their protocols.

4.4 Evaluation

The implementation of DPC is publicly available online for extensions and experimentation.[6] We now report on our experience of using DPC for implementing and validating some commonly used distributed systems.

4.4.1 *More Examples*

In order to evaluate the framework, we have encoded a number of textbook distributed protocols, translating their specifications to the abstractions of DPC. By doing so, we were aiming to answer the following research questions:

6 https://github.com/kandersen/dpc

Protocol	Impl	Protlets	LOC	`RPC`	`ARPC`	`Notif`	`Broad`	`OneOf`	`Quorum`
Calculator	✓	1	10	✓	✓			✓	
Lock Server		4	73	✓	✓	✓			
Concurrent Database		3	23	✓					
Two-Phase Commit		2	43				✓		
Paxos	✓	2	42						✓

Table 4.1: A summary for implemented systems: protocol, runnable implementation, count of constituent protlets, size of encoding (lines of code), employed combinators.

1. Are our `Protlet`-based combinator sufficiently expressive to capture a variety of distributed systems from the standard literature in a natural way?
2. Is it common to have realistic protocols that require *more than one* combinator, i. e., can be efficiently decomposed into multiple `Protlet`s?
3. What is the implementation burden for encoding systems using DPC?

The statistics for our experiments is summarised in Table 4.1.

The framework has been shaped by the explorations of protocols that we have made, but we believe that the answer to Q1 is affirmative, supported by the variety of protocols we have so far explored. The answer to Q2 is also affirmative. Complex protocols from literature decompose into interactions shaped as RPCs, notifications, *etc.*, and we manage to capture all of them in protlets. Simply put, for every arrow in a diagram of the network indicating a communication channel, the protocol has a protlet detailing the exchanges occuring across that channel. For instance the two-phase protocols like Paxos and Two-Phase Commit (2PC) naturally decompose into two broadcast/quorum phases, while more asymmetric protocols like distributed locking [43] requires as many as four protlets.

Regarding Q3, the lines of code versus complexity of protocol are indicative of a positive relationship between complexity and effort to encode a protocol, which is desireable. That is, a lot of complexity is encapsulated by the treatment of combinators, so the coding effort in the framework is very light.

The nature of the verification that the framework enables is naturally not strictly sound (as it is dynamic), but techniques like bounded model checking are readily explorable. With it, we have been able to validate, e. g., correctness for the 2PC protocol [71], a not an insignificant proof burden.

The framework also affords exploration in other directions than we have mentioned so far. We have experimented with enriching the message passing language with operations for shared-memory concurrency and thread-based parallelism. The database example in the table uses *node-local* threads to maintain a database that is served by two different threads. Our approach to dynamic checking of protocol adherance scales to concurrency, and we have a concurrent Calculator server serving *multiple* arithmetic functions *in parallel*.

4.4.2 *Case Study: Constructing and Running Paxos Consensus*

For a representative exploration of the capabilities of DPC we turn to a study of the Paxos Consensus [14, 34, 46]. Paxos solves a problem of reaching a consensus on a single value agreed upon across multiple nodes, of which a subset acts as proposers (who suggest the values) and another, complementary subset acts as acceptors (who reach an agreement). The nature of the Paxos algorithm lends itself well to interactive exploration and the specification should be robust to issues that appear specifically in distributed systems, like arbitrary interleaving of messages, message reorderings, and nodes going offline. The tools we have developed so far are enough to explore these aspects of the protocol.

We can specify this protocol in DPC with relatively little code. We further generalise the `Broadcast` combinator to "quorums" — broadcasts that await only a certain number of responses before proceeding. We introduce another entry in our `Protlet` datatype for capturing this pattern.

```
data Protlet f s = ...
   | Quorum String Rational (Broadcast s) (Receive s) (Send f s)
```

The `Quorum` protlet is and acts identical to the Broadcast protlet, but it is further instrumented by a rational number indicating the number of responses to await before proceeding. We encode the dissection of nodes into proposers and acceptors directly in the state of the protocol, similar to how we dissected the state space of the cloud server along Client/Server lines. The proposer starts in `(ProposerInit b v as)` with the desire to propose to acceptors as the value `v` with priority (*ballot*) `b`. We encode this with a quorom protlet:

```
prepare :: Alternative f => Label -> Int -> Protlet f PState
prepare label n = Quorum "prepare" ((fromIntegral n % 2) + 1)
    propositionCast ...
  where
    propositionCast = \case
      ProposerInit b v as -> Just (zip as (repeat [b]),
          propositionReceive b v as)
      _ -> Nothing
```

Here, `prepare` is parameterised by the number of participants. Hence, the protlet dictates we should wait for a majority quorum, to avoid ties in the system. The listing shows the initiation of the first broadcast as representative of the rest of the implementation. The proposer starts in an `ProposerInit` state, in which it initiates a broadcast poll of all `as` acceptors, sending its ballot `b`.

The second phase of the protocol is encoded as another `Quorum` protlet, where the proposers react to the outcome of the responses on the first polling. The interactive exploration tool can be used to explore, for instance, the robustness of the protocol with respect to crashing participants versus crashing proposers, and why a quorum size of $\left(\frac{n}{2}+1\right)$ acceptors is sufficient for reaching consensus.

The explored implementation demonstrates use of the *state* monad to organise the acceptor as an effectful program, and a *callback* to provide the ballot to the proposer, using features of Haskell, while retaining the benefits of the framework. Neither effect is possible to express at the protocol specification level.

4.4.3 *Case Study 2: Specifying and Model Checking Two-Phased Commit*

For a representative of the verification capabilities of DPC we turn to a study of the Two-Phase Commit protocol encoded in Disel [71]. There, it was properly formalized in the Disel framework, so it translates readily to DPC. This case serves as a study of the same work done using a light-weight, formally-guided approach, as opposed to a fully formal framework.

The Disel encoding of the 2PC protocol as described by Weikum and Vossen [78] assumes a single *coordinator*, often known as a "proposer" in similar treatments, and a static collection of participants, or acceptors. The coordinator asks the participants to agree or disagree with a particular transaction, and the consensus is communicated back to the acceptors once the coordinator has tallied all votes. The state space of the nodes in the protocol is the most complicated we have studied so far:

```
data State = CoordinatorInit [NodeID]
           | CoordinatorCommit [NodeID]
           | CoordinatorAbort [NodeID]
           | ParticipantInit
           | ParticipantGotRequest NodeID
           | ParticipantRespondedYes NodeID
           | ParticipantRespondedNo NodeID
           | ParticipantCommit NodeID
           | ParticipantAbort NodeID
```

The coordinator maintains a list of participants to poll for acceptance, and each participant records the server to respond to upon contact. The server proceeds from `CoordinatorInit` to either `CoordinatorCommit` or `CoordinatorAbort` depending on the outcome of the polling round, and from there back to `CoordinatorInit` upon sending the result of the poll to all participants. Each participant starts in the `ParticipantInit` state, and proceeds to `ParticipantGotRequest` upon being polled by a coordinator. From here it can accept or reject the request as desired, this is left up to the implementaion: it is however *specified* that it must move to `ParticipantRespondedYes` or `ParticipantRespondedNo`, respectively. From there it moves to `ParticipantCommit` or `ParticipantAbort` as appropriate when learning of the outcome of the poll at large from the coordinator.

A safety specification of the protocol is traditionally given in a form of *inductive invariant*—a property that is satisfied by the initial state of the system and is preserved by each modification it undergoes—ultimately implying that "nothing bad happens". Finding an invariant strong enough, so it would adequately capture the relevant properties system, is a work of art, and inevitably requires a human prover's assistance [63]. However, once an invariant is defined, it can be checked mechanically. Invariants are usually defined by conjoining global predicates on the states of each node without regard for the specification of the protocol. That is, they declaratively express the legal states of the *entire system* without mention of legal ways to get there.

Invariants in DPC are expressed as predicates on the specification level state space, and in particular, it was straight forward to adapt the Coq-formulated inductive invariant of the 2PC system to a Haskell function deciding the same property for a 2PC specification.

The invariant checker enabled by DPC appears to be a very useful tool. In this particular case we can borrow an invariant from Disel formalization that expresses full correctness of the 2PC protocol, but it is conceivable that "smaller" properties might be of interest. For local examples, that a particular state is never entered, or that the payload of a particular state satisfies a predicate. For examples of more global properties it might be that no two nodes are in a particular state at once.

The entire invariant is specified as a disjunction:

```
tpcInvariant :: TPCInv
tpcInvariant = everythingInit
          <||> phaseOne
          <||> phaseTwo
```

The invariant asserts that all nodes in the system are either in the initial state, the started phase one or all participants have been polled and the system is in phase two. The type synonym `TPCInv` simply wraps the `Invariant` type introduced in Section 4.3.1, instantiated by the `State` type of this particular case study. The operator <||> simply lifts boolean disjunction to the `Invariant` type.

Let us illustrate the implementation details of the invariant looking at the implementation the `phaseOne` invariant and its components.

```
phaseOne :: TPCInv
phaseOne =
  forCoordinator coordinatorPhaseOne <&&>
  forallParticipants participantPhaseOne
```

We here use domain specific combinators that have been built to make it convenient to refer to the nodes in the state space by their roles as opposed to their `NodeID`. The predicate `participantPhaseOne` is a large disjunction enumerating that a particular node is either in the initial state because it hasn't been polled by the coordinator; or it is in the `GotRequest` state because it has been polled exactly once; or it has responded yes or no, sending the appropriate messages to the coordinator:

```
participantPhaseOne :: NodeID → TPCInv
participantPhaseOne pt = do
  cn ← getCoordinator
  foldOr [
    runningInState ParticipantInit pt
    <&&> noMessageFromTo pt cn
    <&&> messageAt pt "Prepare_Broadcast" [] cn,

    runningInState (ParticipantGotRequest cn) pt
    <&&> noOutstandingMessagesBetween pt cn,

    runningInState (ParticipantRespondedYes cn) pt
    <&&> noMessageFromTo cn pt
    <&&> messageAt cn "Prepare_Response" [1] pt,

    runningInState (ParticipantRespondedNo cn) pt
    <&&> noMessageFromTo cn pt
    <&&> messageAt cn "Prepare_Response" [0] pt
    ]
```

The utility predicates like `noMessageFromTo` are "primitives" provided by the `Invariant` library that are reusable across specifications. They express general properties like state of the message soup pertaining to a particular node or set of nodes.

With an invariant like this in hand, we can check that the specification satisfies this property at every step, i. e., that the invariant is *inductive*:

```
> checkInvariantTraces tpcInvariant initNetworkMetadata . take 15 $
                      simulateNetworkTraces initNetwork

Nothing
```

What we see is exhaustive bounded model checking of the specification: the trace enumeration via `simulateNetworkTraces` evolves the network in a breadth-first manner, returning a list of frontiers, while `checkInvariantTraces` iterates through this "tree" and ensures that the supplied invariant is never violated. Additionally, it is supplied with protocol specific meta-data, some global context accessible to the invariant, which in this case includes the assignment of roles in the protocol to the node identifiers used in `initNetwork`.

Here, we look 15 frontiers deep, a grossly exponential number of states, but enough for the protocol to have run at least once.

While by no means a wild feat of engineering, this brings hard verification to a very light-weight toolkit at very little cost to developers of algorithms. By comparison, the equivalent *formal* verification in DISEL is more than 2,000 lines of definitions and well-formedness property proofs, not counting the invariant it self, *before* the formalization begins any proof-work. Here, we start exploring the behaviour and nuances of the protocol of interest in as little as 75 lines in the case of 2PC.

4.5 Related Work

Declarative programming for distributed systems. In the past five years, several works were published proposing mechanised formalisms for scalable verification of distributed protocols, both in synchronous [29] and asynchronous setting [71, 80]. All those verification frameworks allow for executable implementations, yet the encoding overhead is prohibitively high, and no abstractions for specific interaction patterns are provided in any of them. Most of the DSLs for distributed systems we are aware of are implemented by means of extracting code rather than by means of a shallow DSL embedding [42, 51, 54]. MACE [42], a C++ language extension and source-to-source compiler, provides a suite of tools for generating and model checking distributed systems. DISTALGO [54] and SPLAY [51] extract implementations from protocol descriptions.

In a recent work, Brady [11] has described a discipline of protocol-aware programming in IDRIS, in which adherence of an implementation to a protocol is ensured by the host language's dependent type system, similarly to DISEL, but in a more lightweight form. That approach provides strong static safety guarantees; however, it does not provide dedicated combinators for specific protocol patterns, e. g., broadcasts or quorums.

Similarly to our DPC-based language for defining protocol combinators, the P programming language by Desai et al. [20] has been introduced as a way to facilitate modular construction of distributed systems. In P, a program is a collection of machines. Machines communicate with each other asynchronously through events. In order to implement a protocol, the programmer must specify the structure of the machines and events. P programs can be verified via the built-in PTESTER tool, and are compiled to C as a executables. Therefore, P approach introduces a gap between the verified and the executable artifacts.

More recently, the the MODP system [21] has been built on top of it. MODP is an extension of P, which allows for more complex programs to be built. In MODP, users can implement systems as individual modules, and combine them into a larger module *horizontally*, i. e., by means of Disel-style Rely-Guarantee-based composition [40]. While similar in spirit to DPC, ModP's composition framework appears to be more *coarse-grained* then what we have described. For instance, even though ModP has been used to define and test a version of Paxos, it is not clear how to implement a in it a combinator such as our `Quorum`, which can be reused across multiple protocols.

Relation to Disel. DPC's protlets adapt Disel's protocols, that are phrased exclusively in terms of *low-level* send / receive commands, which should be instrumented with protocol-specific logic for each new construction. While it is possible to derive DPC's protlets in Disel, extracting them and ascribing them suitable types requires large annotation overhead. To wit, only the protocol description for Two-Phase Commit in Disel takes nearly 400 LOC of Coq, while the *entire* protocol, implementation *and* invariant for 2PC in DPC take only 243 LOC of Haskell. We believe that providing a concise reusable specification to advanced DPC protlets, such as `Quorum`, allowing for verification of, e. g., Paxos, would be an interesting research challenge by itself.

The idea of exploiting random exploration of process interleavings in asynchrounous settings in general is not a new one. For instance, generating and controlling schedules of execution have been central in lines of work surrounding concurrency errors in web applications [2]. We here similarly demonstrate the applicability of the approach in a lightweight framework inspired by a program logic.

4.6 Conclusion and Future Work

Declarative programming over distributed protocols is possible and, we believe, can lead to new insights, such as better understanding on how to structure systems implementations. Even though there are several known limitations to the design of DPC (for instance, in order to define new combinators, one needs to extend `Protlet`), we consider our approach beneficial and illuminating for the purposes of prototyping, exploration, and teaching distributed system design. In the future, we are going to explore the opportunities, opened by DPC, for randomised protocol testing and lightweight verification with refinement types.

[1] Martín Abadi and Leslie Lamport. "The Existence of Refinement Mappings." In: *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*. 1988.

[2] Christoffer Quist Adamsen, Anders Møller, Rezwana Karim, Manu Sridharam, Frank Tip, and Koushik Sen. "Repairing Event Race Errors by Controlling Nondeterminism." In: *Proc. 39th International Conference on Software Engineering (ICSE)*. 2017.

[3] Camilla Arndal Andersen. *Taste Analysis via Electroencephalography*. Aarhus Universitet, Institut for Ingeniørvidenskab, 2019.

[4] Kristoffer Just Arndal Andersen and Ilya Sergey. *DPC Haskell Package*. URL: `https://github.com/kandersen/dpc`.

[5] Kristoffer Just Arndal Andersen and Ilya Sergey. "Distributed Protocol Combinators." In: *Practical Aspects of Declarative Languages - 21th International Symposium, PADL*. 2019.

[6] Kristoffer Just Arndal Andersen and Ilya Sergey. "Protocol Combinators for Modeling, Testing, and Execution of Distributed Systems." In: *Journal of Functional Programming* (2019). In Submission.

[7] Kristoffer Just Andersen. "Automatic Verification of Fine-Grained Concurrency." Report in fulfilment of Qualification Exam. 2017.

[8] Josh Berdine, Cristiano Calcagno, and Peter W O'Hearn. "Smallfoot: Modular automatic assertion checking with separation logic." In: *Formal Methods for Components and Objects*. Springer. 2006.

[9] Bodil Biering, Lars Birkedal, and Noah Torp-Smith. "BI Hyperdoctrines and Higher-order Separation Logic." In: *ESOP*. 2005.

[10] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. "Permission Accounting in Separation Logic." In: *POPL*. 2005, pp. 259–270.

[11] Edwin Brady. "Type-driven Development of Concurrent Communicating Systems." In: *Computer Science (AGH)* 18.3 (2017).

[12] Cristian Cadar and Koushik Sen. "Symbolic Execution for Software Testing: Three Decades Later." In: *Commun. ACM* 56.2 (Feb. 2013).

[13] Cristiano Calcagno, Matthew Parkinson, and Viktor Vafeiadis. "Modular safety checking for fine-grained concurrency." In: *SAS 2007*. Springer Berlin/Heidelberg. 2007.

[14] Tushar Chandra, Robert Griesemer, and Joshua Redstone. "Paxos made live: an engineering perspective." In: *PODC*. ACM, 2007.

[15] C. C. Chang and H. Jerome Keisler. *Model Theory*. Studies in Logic and the Foundations of Mathematics. Elsevier Science, 1990.

[16] Adam Chlipala. "Mostly-automated verification of low-level programs in computational separation logic." In: *PLDI*. 2011.

[17] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs." In: *ACM Sigplan Notices* 46.4 (2011).

[18] Coq Development Team. *The Coq Proof Assistant Reference Manual*. 2018.

[19] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008.

[20] Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. "P: safe asynchronous event-driven programming." In: *PLDI*. ACM, 2013.

[21] Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit Seshia. "Compositional Programming and Testing of Dynamic Distributed Systems." In: *PACMPL* 2.OOPSLA (2018).

[22] Edsger W. Dijkstra. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs." In: *Commun. ACM* 18.8 (Aug. 1975).

[23] Thomas Dinsdale-Young, Pedro da Rocha Pinto, and Kristoffer Just Andersen. *Caper (Source Code)*. URL: https://github.com/caper-tool/caper.

[24] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. "Concurrent Abstract Predicates." In: *ECOOP*. 2010.

[25] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. "Views: compositional reasoning for concurrent programs." In: *POPL*. 2013.

[26] Thomas Dinsdale-Young, Pedro da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. *Caper: Automatic Verification with Concurrent Abstract Predicates. Technical Appendix: Program Logic*. 2016. URL: http://cs.au.dk/~kja/papers/caper-esop17/techreport.pdf.

[27] Thomas Dinsdale-Young, Pedro da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. "Caper." In: *ESOP: Proceedings of the 26th European Symposium on Programming*. 2017.

[28] Robert Dockins, Aquinas Hobor, and Andrew W. Appel. “A Fresh Look at Separation Algebras and Share Accounting.” English. In: *Programming Languages and Systems*. Ed. by Zhenjiang Hu. Vol. 5904. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009.

[29] Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. “PSync: a partially synchronous language for fault-tolerant distributed algorithms.” In: *POPL*. ACM, 2016.

[30] Aboubakr Achraf El Ghazi, Mana Taghdiri, and Mihai Herda. “First-Order Transitive Closure Axiomatization via Iterative Invariant Injections.” In: *NASA Formal Methods*. Springer, 2015.

[31] Yurii Leonidovich Ershov. “Decidability of the elementary theory of distributive lattices with relative complements and the theory of filters.” In: *Algebra i Logika* 3 (1964).

[32] Robert W. Floyd. “Algorithm 97: Shortest Path.” In: *Commun. ACM* 5.6 (June 1962).

[33] Robert W. Floyd. “Assigning Meanings to Programs.” In: *Program Verification: Fundamental Issues in Computer Science*. Ed. by Timothy R. Colburn, James H. Fetzer, and Terry L. Rankin. Dordrecht: Springer Netherlands, 1993.

[34] Álvaro García-Pérez, Alexey Gotsman, Yuri Meshman, and Ilya Sergey. “Paxos Consensus, Deconstructed and Abstracted.” In: *ESOP*. Vol. 10801. LNCS. Springer, 2018.

[35] James N Gray. “Notes on data base operating systems.” In: *In Operating Systems*. Springer, 1978, pp. 393–481.

[36] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. “IronFleet: proving practical distributed systems correct.” In: *SOSP*. ACM, 2015.

[37] Maurice P. Herlihy and Jeannette M. Wing. “Linearizability: a correctness condition for concurrent objects.” In: *ACM Trans. Program. Lang. Syst.* 12.3 (July 1990).

[38] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming.” In: *Commun. ACM* 12.10 (1969).

[39] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. “VeriFast: A powerful, sound, predictable, fast verifier for C and Java.” In: *NASA Formal Methods* (2011).

[40] Cliff B. Jones. “Tentative Steps Toward a Development Method for Interfering Programs.” In: 5.4 (1983).

[41] 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.

[42] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. "Mace: Language Support for Building Distributed Systems." In: *PLDI*. ACM, 2007.

[43] Martin Kleppmann. *How to do distributed locking*. `https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html`. 2016.

[44] Donald E Knuth and Peter B Bendix. "Simple word problems in universal algebras." In: *Automation of Reasoning*. Springer, 1983.

[45] Morten Krogh-Jespersen, Amin Timani, Marit Edna Ohlenbusch, and Lars Birkedal. *Aneris: A Logic for Node-Local, Modular Reasoning of Distributed Systems*. 2018.

[46] Leslie Lamport. "The Part-Time Parliament." In: *ACM TOPLAS* 16.2 (1998).

[47] Leslie Lamport. *Paxos Made Simple*. Self-published note. 2001.

[48] Leslie Lamport and Fred B. Schneider. "Formal Foundation for Specification and Verification." In: *Distributed Systems: Methods and Tools for Specification, An Advanced Course*. Vol. 190. LNCS. Springer, 1985.

[49] Butler W. Lampson. "How to Build a Highly Available System Using Consensus." In: *WDAG*. 1996.

[50] Xuan Bach Le, Cristian Gherghina, and Aquinas Hobor. "Programming Languages and Systems: 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings." In: Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. Chap. Decision Procedures over Sophisticated Fractional Permissions.

[51] Lorenzo Leonini, Etienne Riviere, and Pascal Felber. "SPLAY: Distributed Systems Evaluation Made Simple (or How to Turn Ideas into Live Systems in a Breeze)." In: *NSDI*. USENIX Association, 2009.

[52] Sheng Liang, Paul Hudak, and Mark P. Jones. "Monad Transformers and Modular Interpreters." In: *POPL*. ACM Press, 1995.

[53] Anders Lindkvist and Troels Fleischer Skov Jensen. *Contextextual Equivalence in the Elm Language*. Speciale, Insitut for Datalogi / Computer Science Department. Aarhus Universitet, Datalogisk Institut.

[54] Yanhong A. Liu, Scott D. Stoller, Bo Lin, and Michael Gorbovitski. "From Clarity to Efficiency for Distributed Algorithms." In: *OOPSLA*. ACM, 2012.

[55] John McCarthy. “A Basis for a Mathematical Theory of Computation, Preliminary Report.” In: *Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*. IRE-AIEE-ACM ’61 (Western). Los Angeles, California: ACM, 1961.

[56] Peter Müller, Malte Schwerhoff, and Alexander J Summers. “Viper: A Verification Infrastructure for Permission-Based Reasoning.” In: *Verification, Model Checking, and Abstract Interpretation*. Springer. 2016.

[57] Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. “Ynot: Dependent Types for Imperative Programs.” In: *ICFP*. 2008.

[58] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. “Communicating State Transition Systems for Fine-Grained Concurrent Resources.” In: *ESOP*. 2014.

[59] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. “How Amazon web services uses formal methods.” In: *Commun. ACM* 58.4 (2015).

[60] Gian Ntzik, Pedro da Rocha Pinto, and Philippa Gardner. “Fault-Tolerant Resource Reasoning.” In: *APLAS*. 2015.

[61] Peter W. O’Hearn. “Resources, Concurrency and Local Reasoning.” In: *Theor. Comput. Sci.* 375.1-3 (Apr. 2007).

[62] Peter W. O’Hearn, John C. Reynolds, and Hongseok Yang. “Local Reasoning about Programs that Alter Data Structures.” In: *CSL*. Vol. 2142. LNCS. Springer, 2001.

[63] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. “Ivy: safety verification by interactive generalization.” In: *PLDI*. ACM, 2016.

[64] Matthew J. Parkinson and Gavin M. Bierman. “Separation logic and abstraction.” In: *POPL*. 2005.

[65] George Pîrlea and Ilya Sergey. “Mechanising blockchain consensus.” In: *CPP*. ACM, 2018.

[66] Azalea Raad, Jules Villard, and Philippa Gardner. “CoLoSL: Concurrent Local Subjective Logic.” In: *ESOP*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015.

[67] Robbert van Renesse and Deniz Altinbuken. “Paxos Made Moderately Complex.” In: *ACM Comp. Surv.* 47.3 (2015).

[68] John C Reynolds. “Separation logic: A logic for shared mutable data structures.” In: *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*. IEEE. 2002.

[69] Stephan Schulz. “System description: E 1.8.” In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer Berlin Heidelberg, 2013.

[70] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. "Mechanized Verification of Fine-grained Concurrent Programs." In: *36th ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2015)*. 2015.

[71] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. "Programming and proving with distributed protocols." In: *PACMPL* 2.POPL (2018).

[72] Kasper Svendsen and Lars Birkedal. "Impredicative Concurrent Abstract Predicates." In: *ESOP*. 2014.

[73] Alfred Tarski. "Arithmetical classes and types of Boolean algebras." In: *Bulletin of the American Mathematical Society* 55 (1 1949).

[74] R Kent Treiber. *Systems programming: Coping with parallelism*. Tech. rep. RJ 5118. IBM Almaden Research Center, 1986.

[75] Aaron Turon, Derek Dreyer, and Lars Birkedal. "Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency." In: *ICFP*. 2013.

[76] Viktor Vafeiadis. "Modular fine-grained concurrency verification." PhD thesis. University of Cambridge, Computer Laboratory, 2008.

[77] Viktor Vafeiadis. "Automatically proving linearizability." In: *Computer Aided Verification*. Springer. 2010.

[78] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[79] James R. Wilcox, Ilya Sergey, and Zachary Tatlock. "Programming Language Abstractions for Modularly Verified Distributed Systems." In: *SNAPL*. 2017.

[80] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. "Verdi: a framework for implementing and formally verifying distributed systems." In: *PLDI*. ACM, 2015.

[81] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. "TaDA: A Logic for Time and Data Abstraction." In: *ECOOP 2014–Object-Oriented Programming*. Springer Berlin Heidelberg, 2014.

[82] Pedro da Rocha Pinto, Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, and Mark Wheelhouse. "A Simple Abstraction for Complex Concurrent Indexes." In: *OOPSLA*. 2011.