Towards Modular Reasoning for Realistic Programming Languages

Filip Sieczkowski

PhD Dissertation
IT University of Copenhagen

November 2013

Abstract

Computer programs, even the ones whose correctness is vital, are growing ever larger. Thus, if we are to be capable of verifying software, we need to do so *modularly*, i.e., reason separately about separate parts of the big programs and conclude that the whole program is correct based on correctness of its subcomponents. In recent years, we have seen a steady advance in modular techniques for software verification. However, these new developments are still far from encompassing all the features that modern programming languages utilize.

In this thesis, we study several approaches to extend modular reasoning techniques to more realistic programming languages. We consider three distinct directions, which deal with both traditional program logic style verification, as well as relational reasoning.

In **Verifying Object-Oriented Programs with Higher-Order Separation Logic in Coq** we construct a logic for giving modular separation logic specifications to object-oriented code that uses dynamic dispatch through interfaces. This powerful programming technique is commonly used in virtually all modern object-oriented languages, and most previous work only targeted the most common design patterns. We present a logic that is powerful enough to express the rarer, complex patterns of usage, while allowing for simple specification of simpler, more concrete interfaces. We follow this direction with **Formalized Verification of Snapshotable Trees: Separation and Sharing**, where we show that our approach scales to non-trivial programs. In this article we consider a family of sophisticated data structures that use complex sharing patterns, notoriously difficult to reason about with separation logic, and show how to verify one of them using our approach.

A Separation Logic for Fictional Sequential Consistency takes our work on separation logic in a different direction: based on a cutting-edge logic for reasoning about concurrent programs, it tackles the problem of real-world memory models, focusing on the common TSO model. The paper provides two interconnected logics: the TSO logic, in which one can reason about the additional observable behaviours, which the user can observe on the weak memory model, and the SC logic, in which reasoning is standard. Since most well-behaved code (e.g., well synchronized programs) cannot observe any additional behaviours admitted by the memory model, we thus provide the much

needed abstraction from the machine model for a large class of higher-level code.

Finally, in **A Concurrent Logical Relation** we investigate a different, *relational*, approach to reasoning about programs. We treat a concurrent extension of an ML-like language, build the first logical relation for a concurrent programming language, and prove a parallelization theorem, a long standing conjecture that describes, in terms of types and effects, when it is safe to run two expressions in parallel.

Acknowledgments

First and foremost, I would like to thank my advisors, Lars Birkedal and Peter Sestoft, for their continued guidance through these three years, and for outstanding patience and support. I could not imagine better supervisors.

A big thank-you to my collaborators, for the long and interesting discussions, and the ideas we shared. In particular, to my colleagues from the ToMeSo project: Jesper Bengtson, Jonas B. Jensen, Hannes Mehnert and Jacob Thamsborg, and to Kasper Svendsen; they have all been great company, and I will fondly remember the times spent together, both in and out of work.

Finally, to my the friends in Copenhagen, Aarhus, Wrocław and everywhere else, for sharing both in the good times, and the slightly worse. To the wonderful PLS group at the IT University for making it an exceptionally fun environment to work in. And to my family, for always being there when I needed them. Thank you all.

Filip Sieczkowski,
Copenhagen–Aarhus, 29th November 2013.

Contents

Introduction

The recent spread of computers and computer programs to virtually all areas of human life has increased the need for understanding whether these programs are correct. Especially in the case of safety critical systems, which are more and more computerized, we would like to be able to conclude whether the systems will run correctly. However, at the same time the systems have grown in size and complexity, to the extent where analyzing a complete program is virtually impossible. This underlines the need for *modular* techniques of reasoning about software. By modularity we mean, in this context, that we should be able to reason independently about separate components of the system, and deduce the properties of the whole system from the properties of these parts.

In the last decade, we have seen a steady advance in modular techniques for software verification. However, these new developments are still far from encompassing all the features that modern programming languages utilize. In this thesis, we concentrate on two such techniques: separation logic and Kripke logical relations. We consider three distinct directions in which we can extend modular reasoning techniques to more realistic programming languages. First, however, we present a short overview of the key problem.

1 The problems of shared mutable state.

One of the biggest problems in verification of real-life software has, for a long time, been the treatment of shared mutable state. This is a feature pervasive in almost all modern programming languages. In simple imperative languages the pointers allow the programmer to define complex data structures; in object-oriented programs the objects are stored on the heap and can be referenced from multiple different points of the program; even in most functional languages imperative, higher-order state is used to provide the means for more efficient implementations. Also in the concurrent programming languages, most communication primitives are usually developed in terms of shared memory.

To understand how shared mutable state constitutes a problem, consider the following code snippet:

$$\mathsf{x} := 1;$$
$$\mathsf{y} := 2$$

Let us try to specify what value can be stored in the cell x after the snippet executes. Clearly, it can be 1, as long as y denotes a different memory cell then x. However, what if both x and y denote the same location? In this case the value of x should be 2, since this is the final value written to that memory cell. This is an instance of the problem of *pointer aliasing*, and it is one of the problems underlying the considerations in this thesis.

From the program verification perspective, this problem basically reduces to being able to describe pointer aliasing — or, rather, lack of aliasing — in a

concise and modular way. However, reasoning about pointer aliasing directly, using explicit equality of the references, is problematic. In the case of our example, this is still relatively simple: we could informally specify our snippet by saying, for example "if $\mathsf{x} = \mathsf{y}$, then the value of x is 2, otherwise the value of x is 1". However, this approach does not readily scale even to the simplest data structures, especially if we want to keep the specification abstract. For example, if we want to append two singly-linked lists in constant time, using a pointer to the end of the list, we need to know that the two lists are not overlaid: the structures cannot share any pointers. If the list were overlaid, the list resulting from appending one to the other would become a cyclic structure, a result we might assume the programmer did not expect of an append function.

The problem runs even deeper, actually. Even if we manage to express the disjointness of the two lists in question, we do not know how the append action would affect any of the other lists we might know about. This makes specifying functions in a concise way virtually impossible. We discuss one of the ways to solve this problem in the following section.

2 Program logics: the case for separation.

We begin this section with a short overview of Hoare logic. Then, we proceed to discuss the basics of separation logic, motivating its design by how it helps with the problem of pointer aliasing.

Hoare logic [16] was one of the first attempts at reasoning about programs in a modular way. It was originally developed for a simple imperative language, but since its inception a lot of work was put into extending it to more realistic languages, a process that still continues, and of which this thesis is a small part. Its key characteristic is the use of *Hoare triples* of the form

$$\{\mathsf{P}\}\ \mathsf{c}\ \{\mathsf{Q}\}$$

where P and Q are *assertions* and c is the command that we are reasoning about. The assertions are usually interpreted as predicates on the program states, and the interpretation of a triple $\{\mathsf{P}\}\ \mathsf{c}\ \{\mathsf{Q}\}$ states that if we evaluate the program from a state s that satisfies P and the evaluation terminates in a state s', then s' satisfies Q. Assertions themselves form a logic, and we can use the reasoning within this logic via a rule of consequence:

$$\frac{\mathsf{P}_1 \vdash \mathsf{P}_2 \qquad \{\mathsf{P}_2\}\ \mathsf{c}\ \{\mathsf{Q}_2\} \qquad \mathsf{Q}_2 \vdash \mathsf{Q}_1}{\{\mathsf{P}_1\}\ \mathsf{c}\ \{\mathsf{Q}_1\}}.$$

Hoare triples can also be easily adapted to a language of expressions, by taking the postcondition Q to be parameterized on the final value of the expression.

The notion of Hoare triples allows us to reason about simple imperative code, but in fact we need a lot more power if we are to handle realistic programming languages. As Reynolds puts it in [25], where he introduces the notion of *specification* logic,

> [T]o obtain universal specifications about commands in Algol-like languages, it is necessary to extend the language of specifications well beyond the Hoare triples.

In fact, we can build a general higher-order logic of specifications, with Hoare triples being just one of the basic formulae. Another kind of basic formula might be, for example, a specification of a procedure. Adding this kind of specification would allow us to verify programs that use procedure calls, as shown in [25]. Sufficiently strong specification logics can also allow us to reason about correctness of recursive or mutually recursive functions in a general way, for example using Löb induction [7], or, as we show further on, specify object-oriented code that uses interfaces.

These developments, however, do not readily scale to shared mutable state. In fact, the way to reason in a modular way in the classical Hoare logic was through an admissible conjunction proof rule,

$$\frac{\{\mathsf{P}\}\ \mathsf{c}\ \{\mathsf{Q}\}}{\{\mathsf{P} \wedge \mathsf{R}\}\ \mathsf{c}\ \{\mathsf{Q} \wedge \mathsf{R}\}},$$

which usually requires side conditions dependent on the programming language for which the logic is designed. This rule, however, is not admissible if the state of the program includes shared mutable state: the changes to the heap that c makes might well invalidate R!

Separation Logic.

This brings us to separation logic. Introduced independently by Reynolds in [26], and Ishtiaq and O'Hearn in [18], separation logic extends the classic assertion logic of Hoare's in a way that makes a variant of the proof rule discussed above its cornerstone. Specifically, let us consider *when* would the conjunction rule hold: clearly, if the assertion R were made about a part of state that was *different*, disjoint from the part of the state that P was made about, such a property should hold. This observation leads us to a new assertion logic connective, called *separating conjunction* and written $*$. Intuitively, $\mathsf{P} * \mathsf{Q}$ holds in any state that can be split in two *disjoint* sub-states, one of which satisfies P, and the other Q. We also need some primitive way of specifying what values are stored in the heap; for this, we use a *points-to* predicate, written $\mathsf{l} \mapsto \mathsf{v}$, which asserts that the location l is allocated in the heap, and the value stored at l is indeed v.

Coming back to the example from Section 1, we can equip the snippet with a precondition

$$\mathsf{x} \mapsto \mathsf{u} * \mathsf{y} \mapsto \mathsf{v}.$$

This precondition ensures that x and y *can* actually be dereferenced, since we have a points-to assertion for both of these locations. But equally importantly, it states that $\mathsf{x} \neq \mathsf{y}$ — since if they were equal, we would not be able to split the state into two disjoint parts such that the same location were in both parts.

As long as our assertion logic also allows for existential quantification, these new connectives are a powerful tool for describing the shape and contents of disjoint data structures, such as lists or trees. Consider, for example, the following, classic specification of a singly-linked list:

$$\begin{aligned}\mathsf{list}(\mathsf{x}, \mathsf{vs}) \stackrel{\text{def}}{=} &(\mathsf{x} = \mathbf{null} \wedge \mathsf{vs} = \varepsilon) \\ &\vee \exists \mathsf{y}, \mathsf{v}, \mathsf{vs}'.\ x \mapsto (\mathsf{v}, \mathsf{y}) * \mathsf{list}(\mathsf{y}, \mathsf{vs}') \wedge \mathsf{vs} = \mathsf{v} \cdot \mathsf{vs}'\end{aligned}$$

This specification, defined by induction on the mathematical representation of the list, vs, ensures that no parts of the list are shared within the list. If we did not require this, our definition would admit circular lists, which would make reasoning about updates we might make to the list much more complicated. Moreover, if we have an assertion $\mathsf{list}(\mathsf{x}, \mathsf{us}) * \mathsf{list}(\mathsf{y}, \mathsf{vs})$, we are sure that no pointers except for the final null pointer are shared between the two lists. In contrast, a separate assertion that would describe that the lists are well-formed and not overlaid would have to be at least as complex as the definition itself — and it would not give us any knowledge about other lists that we might know about.

These definitions lead us to rephrase the Hoare logic rule for conjoining an assertion to a triple in a way that works in the presence of shared mutable state: the *frame rule*. As with its predecessor, it may need programming language–dependent side conditions, but its general form is

$$\frac{\{\mathsf{P}\}\ \mathsf{c}\ \{\mathsf{Q}\}}{\{\mathsf{P} * \mathsf{R}\}\ \mathsf{c}\ \{\mathsf{Q} * \mathsf{R}\}}.$$

This proof rule states that if we can split our state into two parts, one satisfying P and the other R, and if the part that satisfies P suffices to run the program c and end in a state that satisfies Q, then running the program in the state consisting of both parts will give us a state that again can be split in two parts: one that satisfies Q, and one that still satisfies R. Intuitively, this rule should hold, since the program does not "need" the part of the state described by R, so it should not modify it, but in some settings more complex arguments are needed to show that the rule is sound.

This ability to succinctly and accurately describe the shapes of data structures ties in well with some of the useful features of specification logics. For example, if the specification logic admits second-order quantification, existential quantifiers can be used to make the predicates that encode the shape of the data structure abstract [10]. This is similar in spirit to using existential quantification for data abstraction. Specific proof systems sometimes use more syntactic variants with a particular mode of use, an example developed for the object-oriented setting are *abstract predicates*, introduced by Parkinson and Bierman in [23].

This new expressive power does not come without a cost, however. One can easily notice that the separating conjunction does not admit all the usual structural rules that we would expect of a conjunction. In particular, contraction

rule does not hold in general, since we have

$$\mathsf{l} \mapsto \mathsf{v} \not\vdash \mathsf{l} \mapsto \mathsf{v} * \mathsf{l} \mapsto \mathsf{v}.$$

Admissibility of the weakening rule depends on the design and precise semantics of the logic. A good source for an overview of design considerations and techniques is, for example, a recent review by Jensen [19].

3 Relational reasoning and higher-order store.

In this section, we briefly discuss the development of the logical relation techniques to the point where they are able to handle languages with higher-order store.

The logical relations technique was developed in the 1970s by, among others, Tait [27] and Plotkin [24]. The technique was first developed for showing unary properties of languages, for instance the strong normalization of simply-typed lambda calculus, and later also for binary properties, most importantly contextual equivalence. The essence of the technique is interpreting syntactic types as relations that are determined by the structure of the type: for example, $[\![\sigma \to \tau]\!]$ should be determined from the relations $[\![\sigma]\!]$ and $[\![\tau]\!]$ in a way that guarantees closure under lambda abstraction and application; see [21, Chapter 8] for a good introduction to the technique, as well as more historical references.

In the most basic setting, the types can be interpreted simply as sets of well-typed values or expressions. However, this simple mode of interpretation does not readily scale to richer type systems, including polymorphic calculi. The crucial idea that allows us to interpret the universal quantification, is to have a space of *semantic* types, closed under intersections. Once we have such a space, we can give an interpretation to an open type that is well-formed in a type variable context, $\Delta \vdash \tau$. We interpret Δ as a sequence of semantic types, one for each variables, and the interpretation of the judgment is a map from the interpretation of Δ into semantic types. This treatment admits interpretations of universal and existential quantifiers via intersections and unions respectively, as long as the space of semantic types is closed under these operations. This idea, though stated differently, dates back to Girard's original paper on System F [15]. A good reference for the treatment of polymorphism is [21, Chapter 9].

Still more involved is the treatment of the recursive types. In contrast to universal and existential quantifiers, we cannot directly express the interpretation of the judgment $\Delta \vdash \mu\,\alpha.\ \tau$ in terms of the interpretation of $\Delta, \alpha \vdash \tau$ — at least not without enriching the space of semantic types with some extra structure. One of the possibilities is to ensure that the space of semantic types is non-empty and has a metric structure. Then, as long as the interpretation of $\Delta, \alpha \vdash \tau$ that we want to use to interpret the recursive type is a contractive map, we would be able to use Banach's fixed point theorem to give an interpretation of the recursive type. This approach was first explored by Amadio [5], and Abadi and Plotkin [1]; see for example Birkedal et al. [12] and the references therein for more extensive treatment.

For all their usefulness, the domain-theoretic models mentioned above are complex, and require solid mathematical grounding. Thus, simpler, more operational treatment of recursive types would also be useful. A line of research that provided such a technique is step-indexing. In this setting, one explicitly stratifies the space of semantic types by using a natural number that intuitively corresponds to the number of evaluation steps for which the type provides information. At index 0, the type does not provide any information, while at each higher index the information is more refined. Since the stratified relation is tied to the evaluation steps, this setup provides a relatively simple way to interpret iso-recursive types: unfolding a recursive value requires an evaluation step, so the interpretation of the recursive type only needs to depend on information from the previous stratum, and thus is well-defined. Step-indexing has been introduced by Appel and McAllester [6], and a good reference for this treatment of recursive types is Ahmed's work [3]. Note that the syntactic treatment has been since reconciled with the semantic one by Birkedal et al. in [11], by showing that the space of semantic types in the step-indexed models can also be seen as an ultrametric space.

The higher-order store, the final language extension we consider in this section, presents us with a different challenge still. This is due to the fact that we can no longer consider the semantic type in isolation: after all, we have to consider references, and whether they are well-typed should depend on the store. More explicitly, assume we have a space of semantic types T°. To be able to provide an interpretation of reference types, we need a space of semantic types that would know, at the least, what locations are defined. This gives us a domain equation of the following shape:

$$T = W \to T^\circ,$$

where W denotes the space of *worlds*, the semantic descriptions of heaps. In the case of a ground store, it is enough for the world to describe which locations are allocated, and so we could take $W = \mathcal{P}_{fin}(L)$, a finite set of locations. However, if we have higher-order store, we need to know what type is associated with any given location. This leaves us, in the simplest case, with an equation of the shape

$$W = L \rightharpoonup_{fin} T.$$

Solving this sort of recursive domain equations is the crucial challenge in modeling the higher-order store, and since the recursion occurs in a negative position, the solution is not trivial. That said, in the recent years different solutions for these types of domain equations have been provided, including [2, 4, 17, 12, 11].

Note that the notion of worlds given by the equation above is a very simplistic one. It can be extended to much more complex structures, see for example [4] or some of the recent work on models of concurrent calculi with higher-order state, e.g., [30].

Relational models and aliasing

The problem of aliasing is just as pressing in the relational setting, as it is when reasoning in unary models. Coming back to the code snippet of Section 1, let us consider the following equation:

$$\mathsf{e}_1; \mathsf{e}_2 \simeq \mathsf{e}_2; \mathsf{e}_1.$$

Should we be able to reorder the assignments in the example with this equality? Clearly we should only be allowed to do that if we know that `x` and `y` are not aliased, but this means that the equation does not hold in general. Thus, if we are to provide useful equivalences for programs that use higher-order store, we need to be able to express what aliasing patterns are permitted for the program equivalence to hold.

In the logical relations setting, one way to express such restrictions is to strengthen the underlying type system. To this end, we can use a *type-and-effect* system, which, in addition to the information on the type of an expression, also provides the information on side effects that it may perform. However, since the type and effect information should be determined statically, the precise locations that may or may not be aliased cannot be determined in general. Thus, the type-and-effect systems usually partition the heap into disjoint *regions*, with each reference being allocated in one of the regions. Thus, they can statically infer the *lack* of aliasing — as long as the references belong to different regions of the memory. The effects usually include read, write and allocate actions in a region, but it is possible to extend this set. Type-and-effect systems were introduced by Gifford and Lucassen in [14, 20], and have since been used, for example, to define static analyses [22] or even as a basis for a runtime system of SML [29, 13].

What interests us in the context of this thesis is the use of a type-and-effect system to enable relational reasoning, and in particular to justify effect-based program transformations. This possibility has been conjectured in the original papers of Gifford and Lucassen, but the relational models of higher-order programming languages were not strong enough to handle this kind of reasoning. However, with the development of rich relational models by Benton et al. [8, 9], this line of research has opened to new possibilities, and the work of Thamsborg and Birkedal [28] has extended and adapted it to the operational model, on which we base some of the work presented in this thesis.

4 The contents of this thesis.

This thesis aims to extend some of the modular reasoning techniques towards more realistic programming language features. In the following, we present a short overview of the subsequent chapters.

In Chapter 2, **Verifying Object-Oriented Programs with Higher-Order Separation Logic in Coq**, we define a higher-order separation logic

that allows us to give modular specification to object-oriented code that uses dynamic dispatch. We believe this an important development, since dynamic dispatch, typically using *interfaces* is a widespread and powerful feature of modern object-oriented programming languages. Most previous work targeted specific, widespread design patterns, but didn't readily scale to other modes of use that are relatively common when using interfaces. A particular example that we use in Section 2 of that chapter is reasoning about methods that take *any* instance of a given interface as an argument. This technique can be used, for example, to parameterize an algorithm on an underlying collection data structure. We show how, using higher-order separation logic, one can give an abstract specifications to an interface, and a concrete instantiation thereof to verify a class that implements the interface. Thus, specifications of methods that *use* the interface can also be expressed abstractly. Additionally, if these methods in turn are called by a client that knows which particular implementation is being used, the concrete instantiation of the interface can be used to easily regain the information about the state of the object.

The other aspect of our work in Chapter 2 is the *Coq* formalization. The framework that we lay down in Section 3 is, to a large extent, language independent, and can be used to experiment with different programming language features in a sound manner. The framework itself has been extended with tactic support, and eventually became the *Charge!* platform.

In Chapter 3, **Formalized Verification of Snapshotable Trees: Separation and Sharing**, we show how the techniques shown in Chapter 2 scale to realistic, sophisticated data structures. We begin by presenting a family of challenging case studies, the snapshotable trees, derived from a library of collections for *C#*. This data structure is a usual binary search tree in all but one aspects: it also provides a method to take a persistent *snapshot* of the collection in constant time. Since this time bound means we cannot simply copy the tree to build a snapshot, the memory has to be *shared* between the tree and its snapshots. This makes reasoning about the data structure challenging, particularly so if one wants to give it modular specifications. We also provide four implementations of the data structure, varying in terms of complexity of sharing involved — and, thus, the space usage.

We show how using higher-order separation logic one can achieve relatively modular specifications and use our formalization to prove one of the provided implementations, thus showing that our approach to verification of object-oriented programs does indeed scale to complex, realistic data structure.

We tackle an altogether different problem in Chapter 4, **A Separation Logic for Fictional Sequential Consistency**: instead of considering a sophisticated programming language, we turn to underlying memory models. In particular, we consider a simple, class-based, concurrent language running on a total store order (TSO) memory model. In TSO, each thread has an associated *store buffer* — a FIFO queue that buffers writes. Any time a value should be written into the memory, it is instead added to the buffer, and the writes from the buffer are flushed to the memory nondeterministically. This setup admits additional states that would not be observable with "standard" memory mod-

els, and is commonly used to model some real-life processors, notably of the x86 family.

We provide a separation logic for the TSO model, including some novel connectives that allow us to reason about the new states of the memory, but the real strength of our approach is that it allows us to abstract from the complex memory model, and use a much simpler, standard separation logic for a large class of programs. We achieve this by giving programs that utilize the weak behaviors in a restricted way specifications that provide their clients with a *fiction of sequential consistency*, thus bringing standard modular reasoning to much more realistic memory models.

Finally, in Chapter 5, **A Concurrent Logical Relation**, we again consider a different problem: giving a relational account of concurrency in the presence of higher-order store. We build on a logical relation for an ML-like language with higher-order store developed by Thamsborg and Birkedal, and extend it to support concurrency. To the best of our knowledge, this is the first such logical relation published. Since we are using a type-and-effect system to qualify not just the type of the expression, but also the kinds of side effects the program performs, this allows us to reason about correctness of concurrent program transformations — like the ones one might find in an optimizing compiler. One of such program transformations, proposed by Gifford and Lucassen as one of the arguments for introducing the type-and-effect systems, is automatic, effect-guided parallelization. We prove that, under certain conditions, this is indeed a valid optimization, thus showing that our logical relation is both strong and useful.

Additionally, we show how one can use relational reasoning about concurrent programs to show that certain low-level, nonblocking algorithms fulfill their specification, in this case, given by a high-level programming language construct instead of a logical formula.

5 Overview of the publications.

Listed below are the papers I have contributed to during the PhD studies. Included is a brief account of the role I have had in the research and writing phase.

1. **Verifying Object-Oriented Programs with Higher-Order Separation Logic in Coq**. Jesper Bengtson, Jonas Braband Jensen, Filip Sieczkowski and Lars Birkedal. In *Proceedings of ITP*, 2011. Forms Chapter 2 of this thesis.

   This paper formed the basis for what later became *Charge!* — a sound framework for interactive verification of object-oriented programs, build entirely within the *Coq* proof assistant. My share of the work on this theory was proportional; in particular, the *Coq* formalization was developed jointly by Jesper Bengtson, Jonas B. Jensen and myself, with all of us collaborating closely on all of its features.

In the writing of this paper, my input was concentrated on the sections that concerned formalization of abstract separation logic and its instantiation to the chosen subset of Java. The technical development formalized in *Coq* can be found at `http://itu.dk/people/birkedal/papers/hosl_coq-201105.tar.gz`.

2. **Formalized Verification of Snapshotable Trees: Separation and Sharing**. Hannes Mehnert, Filip Sieczkowski, Lars Birkedal and Peter Sestoft. In *Proceedings of VSTTE* 2012. Forms Chapter 3 of this thesis.

   This article was the first big case study conducted within the framework presented above, and informed some of the decisions taken with respect to proof automation in *Charge!* Hannes and me contributed the majority of this theory, and most of the formalization work fell to me.

   I have taken a smaller part in writing the article, focusing mostly on the treatment of iterators. The technical development formalized in *Coq* can be found at `http://itu.dk/people/hame/snapshots.tar.gz`.

3. **A Concurrent Logical Relation**. Lars Birkedal, Filip Sieczkowski and Jacob Thamsborg. In *Proceedings of CSL* 2012. Forms Chapter 5 of this thesis.

   This paper extends the theory developed earlier by Jacob and Lars to a concurrent setting. My contribution here was proportional; in particular I was responsible for proving the parallelization theorem, the main application we provide of our logical relation. Some of the formal proofs are included in the appendix attached to the paper.

4. **A Separation Logic for Fictional Sequential Consistency**. Filip Sieczkowski, Kasper Svendsen and Lars Birkedal. Under submission. Forms Chapter 4 of this thesis.

   I have developed this theory with help from Kasper and Lars. My contribution has been major, although Kasper and Lars made sure it evolved towards the principled, two-level structure the logic has in its current form.

   The writing has been a joint work, in which I have been responsible for explaining the programming language and the complex TSO logic (Sections 2 and 4). I have also written most of the technical appendix included in this thesis.

5. **A Kripke Logical Relation for Effect-based Program Transformations**. Lars Birkedal, Guilhem Jaber, Filip Sieczkowski and Jacob Thamsborg. Under submission. Not included in this thesis.

   This is an extended version of Jacob and Lars's ICFP 2011 article. My contributions to it were centered around extending the theory presented there to support region polymorphism, and preparing the text for the journal submission.

References

[1] M. Abadi and G. D. Plotkin. A per model of polymorphism and recursive types. In *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 355–365. IEEE, 1990. 5

[2] A. Ahmed. *Semantics of types for mutable state*. PhD thesis, Princeton University, 2004. 6

[3] A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *Programming Languages and Systems*, pages 69–83. Springer, 2006. 6

[4] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *ACM SIGPLAN Notices*, volume 44, pages 340–353. ACM, 2009. 6

[5] R. M. Amadio. Recursion over realizability structures. *Information and Computation*, 91(1):55–85, 1991. 5

[6] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, Sept. 2001. 6

[7] A. W. Appel, P.-A. Melliès, C. D. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 109–122, New York, NY, USA, 2007. ACM. 3

[8] N. Benton, A. Kennedy, L. Beringer, and M. Hofmann. Relational semantics for effect-based program transformations with dynamic allocation. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '07, pages 87–96, New York, NY, USA, 2007. ACM. 7

[9] N. Benton, A. Kennedy, L. Beringer, and M. Hofmann. Relational semantics for effect-based program transformations: Higher-order store. In *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, PPDP '09, pages 301–312, New York, NY, USA, 2009. ACM. 7

[10] B. Biering, L. Birkedal, and N. Torp-Smith. Bi-hyperdoctrines, higher-order separation logic, and abstraction. *ACM Trans. Program. Lang. Syst.*, 29(5), Aug. 2007. 4

[11] L. Birkedal, B. Reus, J. Schwinghammer, K. Støvring, J. Thamsborg, and H. Yang. Step-indexed Kripke models over recursive worlds. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages, POPL '11, pages 119–132, New York, NY, USA, 2011. ACM. 6

[12] L. Birkedal, K. Støvring, and J. Thamsborg. Realizability semantics of parametric polymorphism, general references, and recursive types. In *Foundations of Software Science and Computational Structures*, pages 456–470. Springer, 2009. 5, 6

[13] L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 171–183, New York, NY, USA, 1996. ACM. 7

[14] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 28–38, New York, NY, USA, 1986. ACM. 7

[15] J.-Y. Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. 2nd Scandinavian Logic Symposium, pages 63–92, Amsterdam, Netherlands, 1971. 5

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969. 2

[17] A. Hobor, R. Dockins, and A. W. Appel. A theory of indirection via approximation. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 171–184, New York, NY, USA, 2010. ACM. 6

[18] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *ACM SIGPLAN Notices*, volume 36, pages 14–26. ACM, 2001. 3

[19] J. B. Jensen. Techniques for model construction in separation logic. 2013. 5

[20] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. ACM. 7

[21] J. C. Mitchell. *Foundations for programming languages*, volume 1. MIT press Cambridge, 1996. 5

[22] F. Nielson and H. R. Nielson. Type and effect systems. In *Correct System Design*, pages 114–136. Springer, 1999. 7

[23] M. Parkinson and G. Bierman. Separation logic and abstraction. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 247–258, New York, NY, USA, 2005. ACM. 4

[24] G. D. Plotkin. *Lambda-definability and logical relations.* School of Artificial Intelligence, University of Edinburgh, 1973. 5

[25] J. C. Reynolds. Idealized algol and its specification logic. *Tools and notions for program construction*, pages 121–161, 1982. 2, 3

[26] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002. 3

[27] W. W. Tait. Intensional interpretations of functionals of finite type i. *The Journal of Symbolic Logic*, 32(2):198–212, 1967. 5

[28] J. Thamsborg and L. Birkedal. A Kripke logical relation for effect-based program transformations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 445–456, New York, NY, USA, 2011. ACM. 7

[29] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997. 7

[30] A. J. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical relations for fine-grained concurrency. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 343–356. ACM, 2013. 6

Verifying object-oriented programs with higher-order separation logic in Coq

Jesper Bengtson, Jonas Braband Jensen, Filip Sieczkowski, and Lars Birkedal

IT University of Copenhagen

Abstract. We present a shallow Coq embedding of a higher-order separation logic with nested triples for an object-oriented programming language. Moreover, we develop novel specification and proof patterns for reasoning in higher-order separation logic with nested triples about programs that use interfaces and interface inheritance. In particular, we show how to use the higher-order features of the Coq formalisation to specify and reason modularly about programs that (1) depend on some unknown code satisfying a specification or that (2) return objects conforming to a certain specification. All of our results have been formally verified in the interactive theorem prover Coq.

1 Introduction

Separation Logic [12,16] is a Hoare-style program logic for modular reasoning about programs that use shared mutable data structures. *Higher-order* separation logic [3] (HOSL) is an extension of separation logic that allows for quantification over predicates in both the assertion logic (the logic of pre- and post-conditions) and the specification logic (the logic of Hoare triples). HOSL was proposed with the purposes of (1) reasoning about data abstraction via quantification over resource invariants, and (2) making formalisations of separation logic easier by having one general expressive logic in which it is possible to define predicates, etc., needed for applications. In this article we explore these two purposes further; we discuss each in turn.

The first purpose (data abstraction) has been explored for a first-order language [4], for higher-order languages [9,11], and for reasoning about generics and delegates in object-oriented languages (without interfaces and without inheritance) [18]. In this article we show how HOSL can be used for modular reasoning about interfaces and interface-based inheritance in an object-oriented language like Java or $C\sharp$. Our current work is part of a research project in which we aim to formally specify and verify the C5 generic collection library [8], which is an extensive collection library that is used widely in practice and whose implementation makes extensive use of shared mutable data structures. A first case-study of one of the C5 data structures is described in [7]. C5 is written in $C\sharp$ and is designed mainly using interface inheritance, rather than class-to-class inheritance; different collection modules are related via an inheritance hierarchy among interfaces. For this reason we focus on verifying object-oriented programs that use interfaces and interface-based inheritance.

We explore the second purpose (formalisation) by developing a Coq formalisation of HOSL for an object-oriented class-based language and show through verified examples how it can be used to reason about interfaces and inheritance.

Our formalisation makes use of ideas from abstract separation logic [6] and thus consists of a general treatment of the assertion logic that works for many models and for a general operationally-inspired notion of semantic command. Our general treatment of the logic is also rich enough to cover so-called nested triples [17], which are useful for reasoning about unknown code, either in the form of closures or delegates [18] or, as we show here, in the form of code matching an interface. To reason about object-oriented programs, we instantiate the general development with the heap model for our object-oriented language and derive suitable proof rules for the language. This approach makes it easier in the future to experiment with other storage models and languages, e.g., variants of separation logic with fractional permissions.

Summary of contributions. We formalize a shallow Coq embedding of a higher-order separation logic for an object-oriented programming language. We have designed a system that allows us to write programs together with their specifications, and then prove that each program conforms to its specification. All meta-theoretical results have been verified in Coq[1].

We introduce a pattern for interface specifications that allows for a modular design. An interface specification is parametrised in such a way that any class implementing the interface can be given a suitably expressive specification by a simple instantiation of the interface specification. Moreover, we show how to use nested triples to, e.g., write postconditions in the assertion logic that require a returned object to match a certain specification. Our approach enables us to verify dynamically dispatched method calls, where the dynamic types of the objects are unknown.

Outline. The rest of this article is structured as follows. In Section 2 we demonstrate the patterns we use for writing interfaces by providing a small example program that uses interface inheritance and proving that it conforms to its specification. In Section 3 we cover the language and memory-model independent kernel of our Coq formalisation. In Section 4 we specialise our system to handle Java-like programs by providing constructs and a suitable memory model for a subset of Java. Section 5 covers related work, and Section 6 concludes.

2 Reasoning with interfaces

To demonstrate how our logic is applied, we will use the example of a class Cell that stores a single value and which is extended by a subclass Recell that maintains a backup of the last overwritten value and has an undo operation. This example is originally due to Abadi and Cardelli [1]; a variant of it was also used

[1] The Coq development accompanying this article can be found at `http://itu.dk/people/birkedal/papers/hosl_coq-201105.tar.gz`

```
interface ICell {
  int get();
  void set(int v);
}

class ProxySet {
  static void proxySet(ICell c, int v) {
    c.set(v);
  }
}

class Cell implements ICell {
  int value;

  Cell() { }
  int get() {
    return this.value;
  }
  void set(int v) {
    this.value = v;
  }
}
```

```
interface IRecell extends ICell {
  void undo();
}

class Recell implements IRecell {
  Cell cell;
  int bak;

  Recell() {
    this.cell = new Cell();
  }
  int get() {
    return this.cell.get();
  }
  void set(int v) {
    this.bak = this.cell.get();
    this.cell.set(v);
  }
  void undo() {
    this.cell.set(this.bak);
  }
}
```

Fig. 1. Java code for the Cell-Recell example with interface inheritance.

by Parkinson and Bierman [14] to show how their logic deals with class-to-class inheritance.

We add to this example a method proxySet, which calls the set method of a given object reference. It is a challenge to give a single specification to this method that is powerful enough to expose any additional side effects the set method might have in arbitrary subclasses. We will see in this section how our specification style achieves this, and it is sketched in Section 5 how this compares to related work.

Our model programming language is a subset of both Java and *C♯*. It leaves out class-to-class inheritance and focuses on interface inheritance. This mode of inheritance captures the essential object-oriented aspect of dynamic dispatch, while the code-reuse aspect has to be explicitly encoded with class composition. A Java implementation of the Cell-Recell example can be found in in Figure 1.

2.1 Interface ICell

Interface ICell from Figure 1 is modelled as a parametrised specification that states conditions for whether a class C behaves "Cell-like". In the following, *val* denotes the type of program values, in our case the union of integers, Booleans and object references. Also, $\mathit{UPred}(\mathit{heap})$ is the type of logical propositions over heaps, i.e., the spatial component of the assertion logic (see Section 3.1 for the

precise definition).

$$\begin{aligned} \mathit{ICell} \triangleq\ & \lambda C : \mathit{classname}.\quad \lambda T : \mathit{Type}.\quad \lambda R : \mathit{val} \to T \to \mathit{UPred}(\mathit{heap}). \\ & \lambda g : T \to \mathit{val}.\quad \lambda s : T \to \mathit{val} \to T. \\ & (\forall t : T.\ C{::}\mathsf{get}(\mathsf{this}) \mapsto \{\widehat{R}\ \mathsf{this}\ t\}\_\{\mathsf{r}.\ \widehat{R}\ \mathsf{this}\ t \wedge \mathsf{r} = g\ t\}) \wedge \quad (1) \\ & (\forall t : T.\ C{::}\mathsf{set}(\mathsf{this}, \mathsf{x}) \mapsto \{\widehat{R}\ \mathsf{this}\ t\}\_\{\widehat{R}\ \mathsf{this}\ (\widehat{s}\ t\ \mathsf{x})\}) \wedge \quad (2) \\ & (\forall t, v.\ g\ (s\ t\ v) = v) \quad (3) \end{aligned}$$

There is some notation to explain here. *ICell* is a function that takes five arguments and returns a result of type *spec*, which is the type of specifications. The logical connectives at the outer level ($\wedge$ and $\forall$) thus belong to the specification logic. The parameter R is the representation predicate of class C, so $R\ c\ t$ intuitively means that c is a reference to an object that is mathematically modelled by the value t of type T. The parameters g and s are functions that describe how get and set inspect and transform this mathematical value. They are constrained by (3) to ensure that get will actually return the value set with set.

The notation $C{::}m(\bar{p}) \mapsto \{P\}\_\{r.\ Q\}$ from (1) and (2) specifies that method m of class C has precondition P and postcondition Q. The arguments in a call will be bound to the names $\bar{p}$ in P and Q, and the return value will be bound to r in Q. We support both static and dynamic methods, where dynamic methods have an additional first argument, as seen in (1) and (2). The precise definition is given in Section 4.2.

The notation $\widehat{f}$ from (1) and (2) lifts a function f such that it operates on expressions, including program variables, rather than operating directly on *val*. It is a technical point that can be ignored for a first understanding of this example, but it is crucial for making HOSL work in a stack-based language. Details are in Section 3.2.

The type of T refers to the *Type* universe hierarchy in Coq.

2.2 Method proxySet

Consider method proxySet from Figure 1. Operationally, calling $\mathsf{proxySet}(c, v)$ does the same as calling $c.\mathsf{set}(v)$, and we seek a specification that reflects this. It is crucial for modularity that proxySet can be specified and verified only once and then used with any implementation of ICell that may be defined later. We give it the following specification.

$$\begin{aligned} \mathit{ProxySet\_spec} \triangleq\ & \forall C, T, R, g, s.\ \mathit{ICell}\ C\ T\ R\ g\ s \to \\ & \forall t : T.\ \mathsf{ProxySet}{::}\mathsf{proxySet}(\mathsf{c}, \mathsf{x}) \mapsto \{\mathsf{c} : C \wedge \widehat{R}\ \mathsf{c}\ t\}\_\{\widehat{R}\ \mathsf{c}\ (\widehat{s}\ t\ \mathsf{x})\} \end{aligned}$$

The assertion $\mathsf{c} : C$ means that the object referenced by c is of class C. Thus, the caller of proxySet can pass in an object reference of any class C as long as C can be shown to satisfy *ICell*.

This specification is as powerful as that of set in *ICell* since it essentially forwards it. Any class that behaves Cell-like should be able to encode the behaviour of its set method by a suitable choice of R and s. We will see in Section 2.6 that it, for instance, is possible to pass in a Recell and deduce how proxySet affects its backup value.

2.3 Class Cell

A Java implementation of Cell can be found in Figure 1. We model constructors as static methods that allocate the object before running the initialisation code and return the allocated object, which is what happens in the absence of class-to-class inheritance.

We give class Cell the following specification, which is a conjunction of what we will call an *interface specification* and a *class specification*. These correspond respectively to the *dynamic* and *static* specifications in [14].

$$\mathit{Cell\_spec} \triangleq \exists R_{\mathrm{Cell}}.\ \mathit{ICell}\ \mathsf{Cell}\ \mathit{val}\ R_{\mathrm{Cell}}\ (\lambda v.\ v)\ (\lambda \_, v.\ v) \wedge \mathit{Cell\_class}\ R_{\mathrm{Cell}}$$

where

$$\begin{aligned}\mathit{Cell\_class} \triangleq\ & \lambda R_{\mathrm{Cell}} : \mathit{val} \to \mathit{val} \to \mathit{UPred}(\mathit{heap}).\\ & \mathsf{Cell{::}new}() \mapsto \{\mathit{true}\}\_\{\exists v.\ \widehat{R_{\mathrm{Cell}}}\ \mathsf{this}\ v\} \wedge\\ & (\forall v.\ \mathsf{Cell{::}get(this)} \mapsto \{\widehat{R_{\mathrm{Cell}}}\ \mathsf{this}\ v\}\_\{\mathsf{r}.\ \widehat{R_{\mathrm{Cell}}}\ \mathsf{this}\ v \wedge \mathsf{r} = v\}) \wedge\\ & (\forall v.\ \mathsf{Cell{::}set(this, x)} \mapsto \{\widehat{R_{\mathrm{Cell}}}\ \mathsf{this}\ v\}\_\{\widehat{R_{\mathrm{Cell}}}\ \mathsf{this}\ \mathsf{x}\})\end{aligned}$$

The representation predicate R_{Cell} is quantified such that its definition is visible only while proving the specifications of Cell, thus hiding the internal representation of the class from clients [4,13].

It is crucial that R_{Cell} is quantified outside both the class and the interface specification such that the representation predicate is the same in the two. In practice, a client will allocate a Cell by calling new, which establishes R_{Cell}; later, to model casting the object reference to its interface type, the client knows that *ICell* holds for this same R_{Cell}.

The specifications of get and set in *Cell_class* are identical to their counterparts in *ICell* when C, T, R, g, and s, are instantiated as in *Cell_spec*. In general, the class specification can be more precise than the interface specification, similarly to the dynamic and static specifications of [14].

To prove *Cell_spec*, the existential R_{Cell} is chosen as $\lambda c, v.\ c.\mathsf{value} \mapsto v$. We can then show that *Cell_class* R_{Cell} holds by verifying the method bodies of get, set and init, and the correctness of get and set can be used as a lemma in proving the interface specification. In this way, each method body is verified only once.

2.4 Interface IRecell

To show the analogy to interface inheritance at the specification level, we examine an interface for classes that behave Recell-like. The Java code for that is IRecell in Figure 1. The specification corresponding to this interface follows the same pattern as *ICell*:

$$\begin{aligned}\mathit{IRecell} \triangleq\ & \lambda C : \mathit{classname}.\quad \lambda T : \mathit{Type}.\quad \lambda R : \mathit{val} \to T \to \mathit{UPred}(\mathit{heap}).\\ & \lambda g : T \to \mathit{val}.\quad \lambda s : T \to \mathit{val} \to T.\quad \lambda u : T \to T.\\ & \mathit{ICell}\ C\ T\ R\ g\ s \wedge && (4)\\ & (\forall t : T.\ C{::}\mathsf{undo(this)} \mapsto \{\widehat{R}\ \mathsf{this}\ t\}\_\{\widehat{R}\ \mathsf{this}\ (u\ t)\}) \wedge && (5)\\ & (\forall t, v.\ g\ (u\ (s\ t\ v)) = g\ t) && (6)\end{aligned}$$

Notice that interface extension is modelled by referring to *ICell* in (4). We do not have to respecify get and set since they were already general enough in *ICell* due to it being parametric in g and s. Note how equation (6) specifies the abstract behaviour of undo via g and s.

There is a pattern to how we construct a specification-logic interface predicate from a Java interface declaration. For each method $m(x_1, \ldots, x_n)$, we add a parameter $f_m : T \to val^n \to (val \times T)$. The product $(val \times T)$ can be replaced with just val or T if the method should have no side effects or no return value, respectively. We then add a method specification of the form:

$$\forall t : T.\ C{::}m(\bar{p}) \mapsto \{\widehat{R}\ \mathsf{this}\ t\}\_\{\mathsf{r}.\ \widehat{R}\ \mathsf{this}\ (\pi_2\ (\widehat{f}_m\ \bar{p}\ t)) \wedge \mathsf{r} = \pi_1\ (\widehat{f}_m\ \bar{p}\ t)\}.$$

2.5 Class Recell

The specification of class Recell follows the same pattern as with Cell:

$$\begin{array}{rll} \mathit{Recell\_spec} \triangleq & \exists R_{\mathrm{Recell}} : \mathit{val} \to \mathit{val} \to \mathit{val} \to \mathit{UPred}(\mathit{heap}). & \\ & \mathit{IRecell}\ \mathsf{Recell}\ (\mathit{val} \times \mathit{val})\ R\ g\ s\ u \wedge \mathit{Recell\_class}\ R_{\mathrm{Recell}} & \\ \text{where} & R\ =\ \lambda \mathit{this}, (v, b).\ R_{\mathrm{Recell}}\ \mathit{this}\ v\ b, & g\ =\ \lambda (v, b).\ v, \\ & s\ =\ \lambda (v, b), v'.\ (v', v), & u\ =\ \lambda (v, b).\ (b, b), \end{array}$$

and *Recell_class* is defined analogously to *Cell_class*.

2.6 Class World

The correctness of the above specifications only matters if it enables client code to instantiate and use the classes. The client code in World demonstrates this:

```
class World {
  static ICell make() {
    Recell r = new Recell();
    r.set(5);
    ProxySet::proxySet(r, 3);
    r.undo();
    return r;
  }

  static void main() {
    ICell c = World::make();
    assert c.get() == 5;
  }
}
```

The body of make demonstrates the use of proxySet. Operationally, it should be clear that r has the value 3 and the backup value 5 after the call to proxySet. This can also be proved in our logic despite using a specification of proxySet that was verified without knowledge of Recell and its backup field.

Upon returning from make, we choose to forget that the returned object is really a Recell, upcasting it to ICell. Its precise class is not needed by the caller, main, which only needs to know that the returned object will return 5 from get.

We capture the interaction between these two methods with the following specification, in which $\mathit{FunI} : \mathit{spec} \to \mathit{UPred}(\mathit{heap})$ injects the specification logic

into the logic of propositions over heaps, thus generalising the concept of nested triples. Section 3.5 describes *FunI* in more detail.

$$\mathit{World\_spec} \triangleq \mathsf{World::main}() \mapsto \{\mathit{true}\}\_\{\mathit{true}\} \wedge$$
$$\mathsf{World::make}() \mapsto \{\mathit{true}\}\_\left\{ \begin{array}{l} \mathsf{r}.\ \exists C, T, R, g, s.\ \widehat{\mathit{FunI}}\ (\mathit{ICell}\ C\ T\ R\ g\ s) \wedge \\ \quad \exists t.\ \widehat{R}\ \mathsf{r}\ t \wedge g\ t = 5 \wedge \mathsf{r} : C \end{array} \right\}$$

The make method is specified to return an object whose class C is unknown, but we know that C satisfies *ICell*.

This pattern of returning an object of an unknown type that satisfies a particular specification often comes up in object-oriented programming: think of the method on a collection that returns an iterator, for example. The essence of this pattern is to have a parametrised specification $S : \mathit{classname} \to \mathit{spec}$ and a method specified as $D::m() \mapsto \{\mathit{true}\}\_\{\mathsf{r}.\ \exists C.\ \mathsf{r} : C \wedge \widehat{\mathit{FunI}}\ (S\ C)\}$. A more straightforward alternative to such a specification – one that does not require an embedding of the specification logic in the assertion logic – would be $\exists C.\ S\ C \wedge D::m() \mapsto \{\mathit{true}\}\_\{\mathsf{r}.\ \mathsf{r} : C\}$. However, this restricts the body of m to only being able to return objects of one class. The method body cannot, for example, choose at run time to return either a C_1 or a C_2, where both C_1 and C_2 satisfy S. We find that the most elegant way to allow the method body to make such a choice is to embed the specification in the postcondition.

Using the notion of validity from Definition 5 in Section 3.4 we can now prove that the whole program will behave according to specification:

Theorem 1. *(ProxySet_spec ∧ Cell_spec ∧ Recell_spec ∧ World_spec) is valid.*

3 Abstract representation

The core of our system is designed to be language independent. To allow for different memory models, we adopt the notion of separation algebras from Calcagno et al. [6]; we can then instantiate an assertion logic with any separation algebra suitable for the problem at hand. Commands are modelled as relations on the program state, which in turn consists of a mutable stack and a heap. Finally, we define an expressive specification logic that can be used to reason about semantic commands.

We use set-theoretic notation to describe our formalisation as this makes the theories easier to read; in Coq we model these sets as functions into *Prop*, which is the sort of propositions in Coq.

3.1 Uniform predicates

Definition 1 (Separation algebra). *A separation algebra is a partial, cancellative, commutative monoid* $(\Sigma, \circ, \mathbf{1})$ *where* Σ *is the carrier,* $\circ$ *is the monoid operator, and* $\mathbf{1}$ *is the unit element.*

Intuitively, Σ can be thought of as a type of heaps, and the $\circ$-operator as composition of disjoint heaps. Hence we refer to the elements of Σ as heaps. Two heaps are compatible, written $h_1 \# h_2$ if $h_1 \circ h_2$ is defined. A heap h_1 is a subheap of a h_2, written $h_1 \sqsubseteq h_2$, if there exists an h_3 such that $h_2 = h_1 \circ h_3$. We will commonly refer to a separation algebra by its carrier Σ.

A uniform predicate [5] over a separation algebra is a predicate on heaps and natural numbers; it is upwards closed in the heaps and downwards closed in the natural numbers.

$$\mathit{UPred}(\Sigma) \triangleq \{p \subseteq \Sigma \times \mathbb{N} \mid \forall g, m.\ \forall h \sqsupseteq g.\ \forall n \leq m.\ (g, m) \in p \to (h, n) \in p)\}$$

The upward closure in heaps ensures that we have an intuitionistic separation logic as is desirable for garbage-collected languages.

The natural numbers are used to connect the uniform predicates with the step-indexed specification logic – this connection will be covered in Section 3.5.

We define the standard connectives for the uniform predicates as in [5]:

$$\begin{array}{rcl@{\qquad}rcl}
\mathit{true} & \triangleq & \Sigma \times \mathbb{N} & \mathit{false} & \triangleq & \emptyset \\
p \wedge q & \triangleq & p \cap q & p \vee q & \triangleq & p \cup q \\
\forall x : U.\ f & \triangleq & \bigcap_{x:U} f\ x & \exists x : U.\ f & \triangleq & \bigcup_{x:U} f\ x \\
p \to q & \triangleq & \{(h, n) \mid \forall g \sqsupseteq h.\ \forall m \leq n.\ (g, m) \in p \to (g, m) \in q\} & & & \\
p * q & \triangleq & \{(h_1 \circ h_2, n) \mid h_1 \# h_2 \wedge (h_1, n) \in p \wedge (h_2, n) \in q\} & & & \\
p \mathrel{-\!\!*} q & \triangleq & \{(h, n) \mid \forall m \leq n.\ \forall h_1 \# h.\ (h_1, m) \in p \to (h \circ h_1, m) \in q\} & & &
\end{array}$$

For the quantifiers, U is of type *Type*, i.e. the sort of types in Coq, and f is any Coq function from U to $\mathit{UPred}(\Sigma)$. This allows us to quantify over *any* member of *Type* in Coq.

3.2 Stacks

Stacks are functions from variable names to values: $\mathit{stack} \triangleq \mathit{var} \to \mathit{val}$.

Two stacks are said to agree on a set V of variables if they assign the same value to all members of V: $s \simeq_V s' \triangleq \forall x \in V.\ s\ x = s'\ x$. In order to define operators that take values from the stack as arguments we introduce the notion of a *stack monad*. This approach is similar to that of Varming and Birkedal [20].

$$\mathit{sm}\ T \triangleq \{(f : \mathit{stack} \to T,\ V : \mathcal{P}(\mathit{var})) \mid \forall s, s'.\ s \simeq_V s' \to f\ s = f\ s'\}$$

Intuitively, V is an over-approximation of the free program variables in f. For any $m = (f, V) \in \mathit{sm}\ T$, we write $m\ s$ to mean $f\ s$ and $\mathit{fv}\ m$ to mean V.

Theorem 2. *sm is a monad with return operation* $\lambda x : T.\ ((\lambda\_.\ x), \emptyset)$ *and bind operation* $\lambda m : \mathit{sm}\ T.\ \lambda f : T \to \mathit{sm}\ U.\ ((\lambda s.\ f\ (m\ s)\ s),\ \mathit{fv}\ m \cup \bigcup_{t \in T} \mathit{fv}\ (f\ t))$.

We use the stack monad to model expressions (which can be evaluated to values using data from the stack), pure assertions (that represent logical propositions that are evaluated without using the heap), and assertions (that represent logical propositions that are evaluated using both the heap and the stack).

$$\mathit{expr} \triangleq \mathit{sm}\ \mathit{val} \qquad \mathit{pure} \triangleq \mathit{sm}\ \mathit{Prop} \qquad \mathit{asn}(\Sigma) \triangleq \mathit{sm}\ \mathit{UPred}(\Sigma)$$

We create an assertion logic by lifting all connectives from $\mathit{UPred}(\Sigma)$ into $\mathit{asn}(\Sigma)$. The definitions and properties of the liftings follow from the fact that sm is a monad (Theorem 2).We prove that both the uniform predicates and the assertions model separation logic [3].

Theorem 3. *For any separation algebra Σ, $\mathit{UPred}(\Sigma)$ and $\mathit{asn}(\Sigma)$ are complete BI-algebras.*

The stack monad is also used for the lifting operator $\widehat{f}$ that was introduced in Section 2.1. The operator takes a function f, and returns a function $\widehat{f}$ where any argument type T that is passed to f is replaced with $\mathit{sm}\ T$, and any return type U with $\mathit{sm}\ U$. As an example, the representation predicate R in the specification for ICell, which has type $\mathit{val} \to T \to \mathit{UPred}(\mathit{heap})$, is lifted to $\widehat{R}$ in the assertion-logic formulas of the specification. The resulting type for $\widehat{R}$ is $\mathit{sm}\ \mathit{val} \to \mathit{sm}\ T \to \mathit{sm}\ \mathit{UPred}(\mathit{heap})$, i.e. $\mathit{expr} \to \mathit{sm}\ T \to \mathit{asn}(\mathit{heap})$.

We have to make this lifting explicit in specifications because it restricts how program variables behave under substitution. We have that $(\widehat{f}\ e)[e'/x] = \widehat{f}\ (e[e'/x])$ for any $f : \mathit{val} \to \mathit{UPred}(\Sigma)$, but it is not the case that $(g\ e)[e'/x] = g\ (e[e'/x])$ for any $g : \mathit{expr} \to \mathit{asn}(\Sigma)$ because $g\ e$ may have more free program variables than those appearing in e, whereas $\widehat{f}\ e$ cannot, by construction. To make HOSL useful in a stack-based language, where such substitutions are commonplace, we therefore typically quantify over functions into $\mathit{UPred}(\Sigma)$ that we then lift to $\mathit{asn}(\Sigma)$ where needed.

3.3 Semantic commands

To obtain a language-independent core, we model commands as indexed relations on program states (each consisting of a stack and a heap) – a semantic command will relate, in a certain number of steps, a state either to another state or to an error. The only requirements we impose on these commands are that they do not relate to anything in zero steps, and that they satisfy a frame property that will allow us to infer a frame-rule for all semantic commands. Intuitively, the semantic commands can be seen as abstractions of rules of a step-indexed big-step operational semantics. More formally, we have the following definitions.

Definition 2 (pre-command). *A* pre-command $\tilde{c}$ *relates an initial state to either a terminal state or the special* **err** *state:*

$$\mathit{precmd} \triangleq \mathcal{P}(\mathit{stack} \times \Sigma \times ((\mathit{stack} \times \Sigma) \uplus \{\mathbf{err}\}) \times \mathbb{N})$$

We write $(s, h, \tilde{c}) \rightsquigarrow^n x$ to mean that $(s, h, x, n) \in \tilde{c}$.

Definition 3 (Frame property). *A pre-command $\tilde{c}$ has the frame property in case the following holds. If $(s, h_1, \tilde{c}) \not\rightsquigarrow^n$ **err** and $(s, h_1 \circ h_2, \tilde{c}) \rightsquigarrow^n (s', h')$ then there exists h_1' such that $h' = h_1' \circ h_2$ and $(s, h_1, \tilde{c}) \rightsquigarrow^n (s', h_1')$.*

Definition 4 (Semantic command). *A semantic command satisfies the frame property and does not evaluate to anything in zero steps.*

$$\mathit{semcmd} \triangleq \{\hat{c} \in \mathit{precmd} \mid \hat{c} \text{ has the frame property} \wedge \forall s, h, x.\ (s, h, \hat{c}) \not\rightsquigarrow^0 x\}$$

To facilitate the encoding of imperative programming languages in our framework, we create the following semantic commands that can be used as building blocks for that purpose. These commands are similar to the ones found in [6].

$$\mathbf{id} \qquad \mathbf{seq}\ \hat{c}_1\ \hat{c}_2 \qquad \hat{c}_1 + \hat{c}_2 \qquad \hat{c}^* \qquad \mathbf{assume}\ P \qquad \mathbf{check}\ P$$

Intuitively, these semantic commands are defined as follows: The **id**-command is the identity command – it does nothing; the **seq**-command executes two commands in sequence; the $+$-operator nondeterministically executes one of two commands; the *-command executes a command an arbitrary amount of times; the **assume**-command assumes a pure assertion that can be used to prove correctness of future commands; the **check**-command works like the **id**-command as long as a pure assertion can be inferred. Recall that pure assertions are logical formulas that are evaluated without using the heap.

Theorem 4. **id**, **seq**, $+$, *, **assume**, *and* **check** *are semantic commands.*

3.4 Specification logic

With the assertion logic and the semantic commands in place, we can define the specification logic. Semantically, a specification is a downwards-closed set of natural numbers; this allows us to reason about (mutually) recursive programs via step-indexing.

$$\mathit{spec} \triangleq \{S \subseteq \mathbb{N} \mid \forall m, n.\ m \leq n \wedge n \in S \to m \in S\}$$

The set *spec* is a complete Heyting algebra under the subset ordering, i.e., logical entailment ($\models$) is modelled as subset inclusion. Hence a specification S is *valid* if $S = \mathbb{N}$.

Given assertions P and Q, and semantic command $\hat{c}$, we define a Hoare triple specification:

$$\begin{aligned}\{P\}\hat{c}\{Q\} \triangleq \{n \mid \forall m \leq n.\ \forall k \leq m.\ \forall s, h.\ (h, m) \in P\ s \to (s, h, \hat{c}) \not\rightsquigarrow^k \mathbf{err} \wedge \\ \forall h', s'.\ (s, h, \hat{c}) \rightsquigarrow^k (s', h') \to (h', m - k) \in Q\ s'\}\end{aligned}$$

A program is proved correct by proving that its specification is valid:

Definition 5. *A specification is valid, written* $\models S$*, when true* $\models S$*.*

3.5 Connecting the assertion logic with the specification logic

We define an embedding of the specification logic into the assertion logic as follows:

$$\mathit{FunI} : \mathit{spec} \to \mathit{UPred}(\Sigma) \triangleq \lambda S.\ \Sigma \times S.$$

Lemma 1. *FunI is monotone, preserves implication, and has a left and a right adjoint, when spec and UPred(Σ) are treated as poset categories.*

From the second part of this lemma it follows that *FunI* preserves both finite and infinite conjunctions and disjunctions, which entails that all specification logic connectives are preserved by the translation.

3.6 Recursion

The specification connectives defined in the previous section are not enough for our purposes. When proving a program correct (by proving a formula of the form $\models S$), it is commonplace that the proof of one part of specification in S requires other parts of S – a typical example is recursive method calls, where the specification of the method called must be available in the context during its own verification. To accomplish this, we borrow the *later* operator ($\triangleright$) from Gödel-Löb logic (see [2]).

$$\triangleright S \triangleq \{n+1 \mid n \in S\} \cup \{0\}$$

This operator can be used via the Löb rule, which allows us to do induction on the step-indexes of the semantic commands.

$$\frac{\Gamma \wedge \triangleright S \models S \qquad 0 \in \Gamma \to 0 \in S}{\Gamma \models S}\ \textsc{Löb}$$

In the inductive case $\triangleright S$ is found on the left hand side of the turnstile and can hence be used to prove S.

4 Instantiation to an object-oriented language

We define a Java-like language with syntax of programs $\mathcal{P}$ shown below. The language is untyped and does not need syntax for interfaces; these exist in the specification logic only.

We use a shallow embedding for expressions, which we denote with e, as shown in Section 3.2.

$$\begin{array}{rcl}
\mathcal{P} & ::= & \mathcal{C}^* \qquad\qquad f \in \text{(field names)} \\
\mathcal{C} & ::= & \textbf{class}\ C\ f^*\ (m(\bar{x})\{c;\ \textbf{return}\ e\})^* \\
c & ::= & x := \textbf{alloc}\ C \mid x := e \mid x := y.f \mid x.f := e \mid x := y.m(\bar{e}) \\
 & \mid & x := C{::}m(\bar{e}) \mid \textbf{skip} \mid c_1;\ c_2 \mid \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \\
 & \mid & \textbf{while}\ e\ \textbf{do}\ c \mid \textbf{assert}\ e
\end{array}$$

In order to provide a concrete instance of the assertion logic, we construct a separation algebra of concrete heaps. The carrier set is $\mathit{heap} \triangleq (\mathit{ptr} \times \mathit{field}) \xrightarrow[\text{fin}]{} \mathit{val}$, with the values defined as the union of integers, Booleans and object references. The partial composition $h_1 \circ h_2$ is defined as $h_1 \cup h_2$ if $\text{dom}\ h_1 \cap \text{dom}\ h_2 = \emptyset$; otherwise the result is undefined. The unit of the algebra is the empty map, emp. We denote this separation algebra $(\mathit{heap}, \circ, \mathit{emp})$ with heap. The points-to predicate is defined as $v.f \mapsto v' \triangleq \{(h, n) \mid h \sqsupseteq [(v, f) \mapsto v']\}$.

$$\frac{}{\mathbf{skip} \sim_{\mathsf{sem}} \mathbf{id}}\ \textsc{Skip-Sem} \qquad \frac{c_1 \sim_{\mathsf{sem}} \hat{c_1} \qquad c_2 \sim_{\mathsf{sem}} \hat{c_2}}{c_1;\ c_2 \sim_{\mathsf{sem}} \mathbf{seq}\ \hat{c_1}\ \hat{c_2}}\ \textsc{Seq-Sem}$$

$$\frac{c \sim_{\mathsf{sem}} \hat{c}}{\mathbf{while}\ e\ \mathbf{do}\ c \sim_{\mathsf{sem}} \mathbf{seq}\ (\mathbf{seq}\ (\mathbf{assume}\ e)\ \hat{c})^*\ (\mathbf{assume}\ \neg e)}\ \textsc{While-Sem}$$

$$\frac{c_1 \sim_{\mathsf{sem}} \hat{c}_1 \qquad c_2 \sim_{\mathsf{sem}} \hat{c}_2}{\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \sim_{\mathsf{sem}} (\mathbf{seq}\ (\mathbf{assume}\ e)\ \hat{c}_1) + (\mathbf{seq}\ (\mathbf{assume}\ \neg e)\ \hat{c}_2)}\ \textsc{If-Sem}$$

Fig. 2. The skip, sequential composition, conditional and loop cases of the semantics relation

4.1 Semantics of the programming language

We define the semantics of the programming language commands by relating them to semantic commands instantiated with *heap* as the separation algebra. We write $c \sim_{\mathsf{sem}} \hat{c}$ to denote that the syntactic command c is related to the semantic command $\hat{c}$. The $\sim_{\mathsf{sem}}$ relation can be thought of as a function; it is defined as a relation only because this was more straightforward in Coq.

The commands **skip**, ;, **if**, and **while** can be related directly to composites of the general semantic commands, defined in Section 3.3. The definition of $\sim_{\mathsf{sem}}$ for these commands can be found in Figure 2. For the remaining commands, new semantic commands must be created.

In particular, for method calls, we define a semantic command

$$\mathbf{call}\ x\ C{::}m(\bar{e})\ \mathbf{with}\ c\ \hat{c}$$

that, intuitively, calls method m of class C with arguments $\bar{e}$ and assigns the return value to x; the command c is the method body, and $\hat{c}$ is its corresponding semantic command. This semantic command works uniformly for both static and dynamic methods, since in the dynamic case we can pass the object reference as an additional argument. The definition of this semantic command is shown in Figure 3. The definition makes use of a predicate

$$C{::}m(\bar{p})\{c; \mathbf{return}\ r\} \in \mathcal{P}$$

which holds in case method m in class C has parameters $\bar{p}$ and method body c in program $\mathcal{P}$. The program parameter $\mathcal{P}$ has been left implicit in the other rules. The notation $[\bar{p} \mapsto (\bar{e}\ s)]$ denotes a finite map that associates each p in $\bar{p}$ with the e at the corresponding position in $\bar{e}$ evaluated in stack s.

The requirement that the method body is related to the semantic command is not enforced by the construction of the semantic command, but rather by the definition of $\sim_{\mathsf{sem}}$ for respectively static and dynamic method calls:

$$\frac{c \sim_{\mathsf{sem}} \hat{c}}{x := C{::}m(\bar{e}) \sim_{\mathsf{sem}} \mathbf{call}\ x\ C{::}m(\bar{e})\ \mathbf{with}\ c\ \hat{c}}\ \textsc{SCall-Sem}$$

$$\frac{([\bar{p} \mapsto (\bar{e}\ s)], h, \hat{c}) \rightsquigarrow^n (s', h') \qquad C{::}m(\bar{p})\{c; \mathbf{return}\ r\} \in \mathcal{P} \qquad |\bar{p}| = |\bar{e}|}{(s, h, \mathbf{call}\ x\ C{::}m(\bar{e})\ \mathbf{with}\ c\ \hat{c}) \rightsquigarrow^{n+1} (s[x \mapsto (r\ s')], h')}\ \textsc{Call}$$

$$\frac{C{::}m(\bar{p})\{c; \mathbf{return}\ r\} \notin \mathcal{P}}{(s, h, \mathbf{call}\ x\ C{::}m(\bar{e})\ \mathbf{with}\ c\ \hat{c}) \rightsquigarrow^1 \mathbf{err}}\ \textsc{Call-Fail1}$$

$$\frac{C{::}m(\bar{p})\{c; \mathbf{return}\ r\} \in \mathcal{P} \qquad |\bar{p}| \neq |\bar{e}|}{(s, h, \mathbf{call}\ x\ C{::}m(\bar{e})\ \mathbf{with}\ c\ \hat{c}) \rightsquigarrow^1 \mathbf{err}}\ \textsc{Call-Fail2}$$

$$\frac{([\bar{p} \mapsto (\bar{e}\ s)], h, \hat{c}) \rightsquigarrow^n \mathbf{err} \qquad C{::}m(\bar{p})\{c; \mathbf{return}\ r\} \in \mathcal{P} \qquad |\bar{p}| = |\bar{e}|}{(s, h, \mathbf{call}\ x\ C{::}m(\bar{e})\ \mathbf{with}\ c\ \hat{c}) \rightsquigarrow^{n+1} \mathbf{err}}\ \textsc{Call-Fail3}$$

Fig. 3. Semantic call commands.

$$\frac{c \sim_{\mathsf{sem}} \hat{c} \qquad y : C}{x := y.m(\bar{e}) \sim_{\mathsf{sem}} \mathbf{call}\ x\ C{::}m(y, \bar{e})\ \mathbf{with}\ c\ \hat{c}}\ \textsc{DCall-Sem}$$

4.2 Syntactic Hoare triples and the concrete assertion logic

Hoare triples for syntactic commands are defined in the following manner:

$$\{P\}c\{Q\} \triangleq \forall \hat{c}.\ c \sim_{\mathsf{sem}} \hat{c} \to \{P\}\hat{c}\{Q\}.$$

From this definition we infer and prove sound Hoare rules for all commands of our language. To define the rule for method calls we first define the predicate that asserts the specification of methods, introduced in Section 2.1.

$$C{::}m(\bar{p}) \mapsto \{P\}\_\{r.\ Q\} \triangleq \exists c, e.\ wf(\bar{p}, r, P, Q, c) \land C{::}m(\bar{p})\{c; \mathbf{return}\ e\} \in \mathcal{P} \\ \land \{P\}c\{Q[e/r]\},$$

where *wf* is a predicate to assert the following static properties: the method parameter names do not clash; the pre- and postcondition do not use any stack variables other than the method parameters and this (the postcondition may also use the return variable); the method body does not modify the values of the method parameters or this.

Selected proof rules for syntactic commands are shown in Figure 4. Note the use of the *later* operator ($\triangleright$) in the method call rule; this means that this method call rule will often be used in connection with the Löb rule.

Theorem 5. *The rules in Figure 4 are sound with respect to the operational semantics.*

$$\frac{}{\models \{P\}\mathbf{skip}\{P\}}\ \textsc{Skip} \qquad \frac{}{\{P\}c_1\{Q\} \wedge \{Q\}c_2\{R\} \models \{P\}c_1;\ c_2\{R\}}\ \textsc{Seq}$$

$$\frac{}{\{P \wedge e\}c_1\{Q\} \wedge \{P \wedge \neg e\}c_2\{Q\} \models \{P\}\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\{Q\}}\ \textsc{If}$$

$$\frac{}{\{P \wedge e\}c\{P\} \models \{P\}\mathbf{while}\ e\ \mathbf{do}\ c\{P \wedge \neg e\}}\ \textsc{While} \qquad \frac{P \vdash e}{\models \{P\}\mathbf{assert}\ e\{P\}}\ \textsc{Assert}$$

$$\frac{}{\models \{\mathit{true}\}x :=\ \mathbf{alloc}\ C\{\forall^* f \in \mathit{fields}(C).\ x.f \mapsto \mathit{null}\}}\ \textsc{Alloc}$$

$$\frac{}{\models \{P\}x := e\{\exists v.\ P[v/x] \wedge x = e[v/x]\}}\ \textsc{Assign} \qquad \frac{}{\models \{x.f \mapsto \_\}x.f := e\{x.f \mapsto e\}}\ \textsc{Write}$$

$$\frac{P \vdash y.f \mapsto e}{\models \{P\}x := y.f\{\exists v.\ P[v/x] \wedge x = e[v/x]\}}\ \textsc{Read}$$

$$\frac{\Gamma \models \triangleright C::m(\bar{p}) \mapsto \{P\}\_\{r.\ Q\} \qquad |\bar{p}| = |y, \bar{e}|}{\Gamma \models \{y : C \wedge P[y, \bar{e}/\bar{p}]\}x := y.m(\bar{e})\{\exists v.\ Q[x, y[v/x], \bar{e}[v/x]/r, \bar{p}]\}}\ \textsc{DCall}$$

$$\frac{\Gamma \models \triangleright C::m(\bar{p}) \mapsto \{P\}\_\{r.\ Q\} \qquad |\bar{p}| = |\bar{e}|}{\Gamma \models \{P[\bar{e}/\bar{p}]\}x := C::m(\bar{e})\{\exists v.\ Q[x, \bar{e}[v/x]/r, \bar{p}]\}}\ \textsc{SCall}$$

$$\frac{P \vdash P' \qquad Q' \vdash Q}{\{P'\}c\{Q'\} \models \{P\}c\{Q\}}\ \textsc{Consequence} \qquad \frac{\forall x \in \mathit{fv}\ R.\ c\ \mathit{does\ not\ modify}\ x}{\{P\}c\{Q\} \models \{P * R\}c\{Q * R\}}\ \textsc{Frame}$$

$$\frac{P \vdash P' \qquad Q' \vdash Q \qquad \mathit{fv}\ P \subseteq \mathit{fv}\ P' \qquad \mathit{fv}\ Q \subseteq \mathit{fv}\ Q'}{C::m(\bar{p}) \mapsto \{P'\}\_\{r.\ Q'\} \models C::m(\bar{p}) \mapsto \{P\}\_\{r.\ Q\}}\ \textsc{Consequence-MSpec}$$

$$\frac{\mathit{fv}\ R \subseteq \{\mathsf{this}\} \cup \bar{p}}{C::m(\bar{p}) \mapsto \{P\}\_\{r.\ Q\} \models C::m(\bar{p}) \mapsto \{P * R\}\_\{r.\ Q * R\}}\ \textsc{Frame-MSpec}$$

Fig. 4. Specification logic rules for syntactic Hoare triples

5 Related work

Formalisations of higher-order separation logic have been proposed before, e.g. by Varming and Birkedal [20], who developed an Isabelle/HOL formalisation of HOSL for partial correctness for a simple imperative language with first-order mutually recursive procedures, using a denotational semantics of the programming language, and by Preoteasa [15], who developed a PVS formalisation for total correctness using a predicate-transformer semantics for a similar programming language.

Parkinson and Bierman treated an extended version of the Cell-Recell example in [14], improving upon their earlier work in [13]. Their approach is to tailor the specification logic to build in a form of quantification over families of rep-

resentation predicates following a fixed pattern determined by the inheritance tree of the program. This construction is known as *abstract predicate families* (APFs).

Where our logic allows quantification over a representation type T, as used in Section 2.1, APFs have a built-in notion of variable-arity predicates to achieve same effect: representation predicates of a subclass can add parameters to the representation predicate they inherit. Class Cell defines a two-parameter representation predicate family Val, which is extended to three arguments in Recell. A Recell r having value 2 and backup field 1 would be asserted as $Val(r, 2, 1)$. This assertion implies $Val(r, 2)$, which in turn implies $\exists b. Val(r, 2, b)$ if it is known that r is a Recell. Thus, casting to the two-argument representation predicate that would be necessary for calling $\{\exists v. Val(\mathsf{c}, v)\}$ $\mathsf{proxySet}(\mathsf{c}, \mathsf{x})$ $\{Val(\mathsf{c}, \mathsf{x})\}$ will lose any information about the backup field.

The logic of Parkinson and Bierman was extended by van Staden and Calcagno [19] to handle multiple inheritance, abstract classes and controlled leaking of facts about the abstract representation of either a single class or a class hierarchy. Using the latter feature, we observe that their logic can also be used to reason about the example in Section 2, by using parameters g and s to give a precise specification of proxySet. Instead of being functions, g and s would be abstract predicate families whose first argument would be an object reference used only for selecting the correct member of the APF.

Compared to the logics based on abstract predicate families, our logic allows families of not just predicates but also types, functions, class names or any other type that can be quantified over in Coq. This gives us strong typing of logical variables, and all this works without building it into the logic and requiring that quantifications and proofs follow the shape of the inheritance tree.

6 Conclusion and Future Work

We have presented a Coq implementation of a generic framework for higher-order separation logic. In this framework, instantiated with a simple object-oriented language, we have shown how HOSL can be used to reason about interfaces and interface inheritance.

Future work includes developing better support for automation via better use of tactics. Our Coq proofs of example programs are cluttered with manual reordering of the context because we do not yet have tactics to automate this. We also plan to integrate the current tool with an Eclipse front-end that is currently being researched within our project [10]. Moreover, we plan to use the tool for formal verification of interesting data structures from the C5 collection library.

Although it is not necessary for the code we mostly want to verify, proper support for class-to-class inheritance in both the logic and the design pattern would enable more direct comparison with related work. It would also make our Java subset more similar to actual Java.

Acknowledgements We are grateful for useful discussions with Hannes Mehnert, Matthew Parkinson, Peter Sestoft, Kasper Svendsen, and Jacob Thamsborg.

This worked was supported in part by the ToMeSo project, funded by the Danish Research Council.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1996.
2. A. W. Appel, P.-A. Melliès, C. D. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. In *Proceedings of POPL*, 2007.
3. B. Biering, L. Birkedal, and N. Torp-Smith. BI hyperdoctrines and higher-order separation logic. In *Proceedings of ESOP*, pages 233–247, 2005.
4. B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines, higher-order separation logic, and abstraction. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.
5. L. Birkedal, B. Reus, J. Schwinghammer, K. Støvring, J. Thamsborg, and H. Yang. Step-indexed kripke models over recursive worlds. In *Proceedings of POPL*, 2011.
6. C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *Proceedings of LICS*, pages 366–378, 2007.
7. J. Jensen, L. Birkedal, and P. Sestoft. Modular verification of linked lists with views via separation logic. *Journal of Object Technology*, 2011. To Appear. Preliminary version in FTfJP'10, available at www.itu.dk/people/birkedal/papers/views.pdf.
8. N. Kokholm and P. Sestoft. The C5 generic collection library for C# and CLI. Technical Report ITU-TR-2006-76, IT University of Copenhagen, 2006.
9. N. R. Krishnaswami, J. Aldrich, L. Birkedal, K. Svendsen, and A. Buisse. Design patterns in separation logic. In *In Proceedings of TLDI*, pages 105–116, 2009.
10. H. Mehnert. Kopitiam: modular incremental interactive full functional static verification of java code. In M. Bobaru, K. Havelund, G. Holzmann, and R. Joshi, editors, *Proceedings of the Third NASA Formal Methods Symposium (NFM 2011)*. NASA, April 2011.
11. A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable ADTs in hoare type theory. In *In Proc. of ESOP*, pages 189–204, 2007.
12. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL*, pages 1–19, 2001.
13. M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *Proceedings of POPL*, pages 247–258, 2005.
14. M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In *Proceedings of POPL*, pages 75–86, 2008.
15. V. Preoteasa. Frame rules for mutually recursive procedures manipulating pointers. *Theoretical Computer Science*, 2009.
16. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of LICS*, pages 55–74, 2002.
17. J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang. Nested Hoare triples and frame rules for higher-order store. In *Proceedings of CSL*, 2009.
18. K. Svendsen, L. Birkedal, and M. Parkinson. Verifying generics and delegates. In *Proceedings of ECOOP*, pages 175–199, 2010.
19. S. van Staden and C. Calcagno. Reasoning about multiple related abstractions with multistar. In *Proceedings of OOPSLA*, pages 504–519, 2010.
20. C. Varming and L. Birkedal. Higher-order separation logic in Isabelle/HOLCF. *Electr. Notes Theor. Comput. Sci.*, 218:371–389, 2008.

Formalized Verification of Snapshotable Trees: Separation and Sharing

Hannes Mehnert, Filip Sieczkowski, Lars Birkedal, and Peter Sestoft

IT University of Copenhagen
{hame, fisi, birkedal, sestoft}@itu.dk

Abstract. We use separation logic to specify and verify a Java program that implements snapshotable search trees, fully formalizing the specification and verification in the Coq proof assistant. We achieve *local* and *modular* reasoning about a tree and its snapshots and their iterators, although the implementation involves shared mutable heap data structures with no separation or ownership relation between the various data.
The paper also introduces a series of four increasingly sophisticated implementations and verifies the first one. The others are included as future work and as a set of challenge problems for full functional specification and verification, whether by separation logic or by other formalisms.

1 Introduction

This paper presents a family of realistic but compact challenge case studies for modular software verification. We fully specified and verified the first case study in Coq, using a domain-specific separation logic [10] and building upon our higher-order separation logic [2]. As future work we plan to verify the other implementations with the presented abstract interface specification. We believe this is the first mechanical formalization of this approach to modular reasoning about implementations that use shared heap data with no separation or ownership relation between the various data.

The family of case studies consists of a single interface specification for snapshotable trees, and four different implementations. A *snapshotable tree* is an ordered binary tree that represents a set of items and supports taking readonly *snapshots* of the set, in constant time, at the expense of slightly slower subsequent updates to the tree. A snapshotable tree also supports iteration (enumeration) over its items as do, e.g., the Java collection classes. The four implementations of the snapshotable tree interface all involve shared heap data as well as increasingly subtle uses of destructive heap update.

For practical purposes it is important that the same interface specification can support verification of multiple implementations with varying degrees of internal sharing and destructive update. Moreover, the specification must accommodate any number of data structure (tree) instances, each having any number of iterators and snapshots, each of which in turn can have any number of iterators. Most importantly, we show how we can have local reasoning (a frame rule) even though the tree and its snapshots share mutable heap data.

We welcome other solutions to the specification and verification of this case study; indeed R. Leino has already made one (unpublished) using Dafny [11].

The Java source code of the case studies of all four implementations and the Coq source is available at `http://www.itu.dk/people/hame/snapshots.tgz`.

Section 2 presents the interface of the case study data structure, shows an example use, and outlines four implementations. Section 3 gives a formal specification of the interface using separation logic and verifies the example code. Sections 4 and 5 verify the first implementation.

2 Case Study: Snapshotable Trees

The case study is a simplified version of snapshotable treesets from the C5 collection library [8].

2.1 Interface: Operations on Snapshotable Trees

Conceptually, a snapshot of a treeset is a readonly copy of the treeset. Subsequent updates to the tree do not affect any of its snapshots, so one can update the tree while iterating over a snapshot. Taking a snapshot must be a constant time operation, but subsequent updates to the tree may be slower after a snapshot has been taken. Implementations (Section 2.3) typically achieve this by making the tree and its snapshots share parts of their representation, gradually unsharing it as the tree gets updated, in a manner somewhat analogous to copy-on-write memory management schemes in operating systems.

All tree and snapshot implementations implement the same ITree interface:

```
public interface ITree extends Iterable<Integer> {
  public boolean contains(int x);
  public boolean add(int x);
  public ITree snapshot();
  public Iterator<Integer> iterator();
}
```

These operations have the following effect:

- `tree.contains(x)` returns true if the item is in the tree, otherwise false.
- `tree.add(x)` adds the item to the tree and returns true if the item was not already in the tree; otherwise does nothing and returns false.
- `tree.snapshot()` returns a readonly snapshot of the given tree. Updates to the given tree will not affect the snapshot. A snapshot cannot be made from a snapshot.
- `tree.iterator()` returns an iterator (also called enumerator, or stream) of the tree's items. Any number of iterators on a tree or snapshot may exist at the same time. Modifying a tree will invalidate all iterators on that tree (but not on its snapshots), so that the next operation on such an iterator will throw ConcurrentModificationException.

We include the somewhat complicated `iterator()` operation because it makes the distinction between a tree and its snapshots completely clear: While it is illegal to modify a tree while iterating over it, it is perfectly legal to modify the tree while iterating over one of its snapshots. Also, this poses an additional verification challenge when considering implementations with rebalancing (cases A2B1 and A2B2 in Section 2.3) because `tree.add(item)` may rebalance the tree in the middle of an iteration over a snapshot of the tree, and that should be legal and not affect the iteration.

Note that for simplicity, items are here taken to be integers; using techniques from [20] it is straightforward to extend our formal specification and verification to handle a generic version of snapshotable trees.

2.2 Example Client Code

To show what can be done with snapshots and iterators (and not without), consider this piece of client code. It creates a treeset `t`, adds three items to it, creates a snapshot `s` of the tree, and then iterates over the snapshot's three items while adding new items (6 and 9) to the tree:

```
ITree t = new Tree();
t.add(2); t.add(1); t.add(3);
ITree s = t.snapshot();
Iterator<Integer> it = s.iterator();
boolean lc = it.hasNext();
while (lc) {
  int x = it.next();
  t.add(x * 3);
  lc = it.hasNext();
}
```

2.3 Implementations of Snapshotable Trees

One may consider four implementations of treesets, spanned by two orthogonal implementation features. First, the tree may be unbalanced (A1) or it may be actively rebalanced (A2) to keep depth $O(\log n)$. Second, snapshots may be kept persistent, that is, unaffected by tree updates, either by path copy persistence (B1) or by node copy persistence (B2):

	Without rebalancing	With rebalancing
Path copy persistence	A1B1	A2B1
Node copy persistence	A1B2	A2B2

The implementation closest to that of the C5 library [8, section 13.10] is A2B2, which is still somewhat simplified: only integer items, no comparer argument, no update events, and so on. In this paper we formalize and verify only implementation A1B1; the verification of the more sophisticated implementations A1B2, A2B1 and A2B2 will be addressed in future work.

Nevertheless, for completeness and in the hope that others may consider this verification challenge, we briefly discuss all four implementations and the expected verification challenges here.

With *path copy persistence* (cases AxB1), adding an item to a tree will duplicate the path from the root to the added node, if this is necessary to avoid modifying any snapshot of the tree. Thus an update will create $O(d)$ new nodes where d is the depth of the tree.

With *node copy persistence* (cases AxB2), each tree node has a spare child reference. The first update to a node uses this spare reference, does not copy the node and does not update its parent; the node remains shared between the tree and its snapshots. Only the second update to a node copies it and updates its parent. Thus an update does not replicate the entire path to the tree root; the number of new nodes per update is amortized $O(1)$. See Driscoll [6] or [8].

To implement ordered trees without rebalancing (cases A1By), we use a Node class containing an item (here an integer) and left and right children; `null` is used to indicate the absence of a child. A tree or snapshot contains a stamp (indicating the "time" of the most recent update) and a reference to the root Node object; `null` if the tree is empty.

To implement rebalancing of trees (cases A2By), we use left-leaning red-black trees (LLRB) which encode 2-3 trees [1, 19], instead of general red-black trees [7] as in the C5 library. This reduces the number of rebalancing cases.

To implement iterators on a tree or snapshot we use a class TreeIterator that holds a reference to the underlying tree, a stamp (the creation "time" of the iterator) and a stack of nodes. The stamp is used to detect subsequent updates to the underlying tree, which will invalidate the iterator. Since snapshots cannot be updated, their iterators are never invalidated. The iterator's stack holds its current state: for each node in the stack, the node's own item and all items in the right subtree have yet to be output by the iterator.

Case A1B1 = no rebalancing, path copy persistence In this implementation there is shared data between a tree and its snapshots, but the shared data is not being mutated because the entire path from the root to an added node gets replicated. Hence no node reachable from the root of a snapshot, or from nodes in its iterators' stacks, can be affected by an update to the live tree; therefore no operation on a snapshot can be affected by operations on the live tree. Although this case is therefore the simplest case, it already contains many challenges in finding a suitable specification for trees, snapshots and iterators, and in proving the stack-based iterator implementation correct.

Case A2B1 = rebalancing, path copy persistence In this case there is potential mutation of shared data, because the rebalancing rotations seem to be able to affect nodes just off the fresh unshared path from a newly added node to the root. This could adversely affect an iterator of a snapshot because a reference from the iterator's node stack might have its right child updated (by a rotation), thus wrongly outputting the items of its right subtree twice or not at all. However, this does not happen because the receiver of a rotation (to be moved down) is

always a fresh node (we're in case B1 = path copy persistence) and moreover we consider only `add` operations (not `remove`), so the child being rotated (moved up) is also a fresh node and thus not on the stack of any iterator – the rebalancing was caused by this child being "too deep" in the tree. Hence if we were to support `remove` as well, then perhaps the implementation of rotations needs to be refined.

Case A1B2 = no rebalancing, node copy persistence In this case, there is mutation of shared data not observable by the client. For example, a left-child update to a tree node that is also part of a snapshot will move the snapshot's left-child value to the node's extra reference field, and destructively update the left child as required for the live tree. There should be no observable change to the snapshot, despite the change to the data representing it. The basic reason for correctness is that any snapshot involving an updated node will use the extra reference and hence not see the update; this is true for nodes reachable from the root of a snapshot as well as for nodes reachable from the stack of an iterator. When we need to update a node whose extra reference is already in use, we leave the old node alone and create a fresh copy of the node for use in the live tree; again, existing snapshots and their iterators do not see the update.

Case A2B2 = rebalancing, node copy persistence In this case there is mutation of shared data (due both to moving child fields to the extra reference in nodes, and due to rotations), not observable for the client. Since the updates caused by rotations are handled exactly like other updates, the correctness of rebalancing with respect to iterators seems to be more straightforward than in case A2B1.

3 Abstract Specification and Client Code Verification

We use higher-order separation logic [18, 3] to specify and verify the snapshotable tree data structure. We build on top of our intuitionistic formalization of HOSL in Coq [2] with semantics for an untyped Java-like language.

To allow implementations to share data between a tree, its snapshots, and iterators and still make it possible for clients to reason locally (to focus only on a single tree / snapshot / iterator), we will use an idea from [10] (see also the verification of Union-Find in [9]). The idea is to introduce an abstract predicate, here named H, global to each tree data structure consisting of a single tree, multiple snapshots, and multiple iterators. This abstract predicate H is parameterized by a finite set of disjoint *abstract structures*. We have three kinds of abstract structures: Tree, Snap, and Iter. The use of H enables a client of our specification to consider each abstract structure to be separate or disjoint from the rest of the abstract structures and thus the client can reason modularly about client code using only those abstract structures she needs; the rest can be framed out. Since the abstract predicate H is existentially quantified, the client has no knowledge of how an implementation defines H (see [3, 16] for more on abstract predicates in higher-order separation logic). The implementor

of the tree data structure has a global view on the tree with its snapshots and iterators, and is able to define which parts of the abstract structures are shared in the concrete heap. Section 4 defines H for the A1B1 case from Section 2.3.

The Tree abstract structure consists of a handle (reference) to the tree and a model, which is an ordered finite set, containing the elements of the tree. The Snap structure is similar to Tree. The Iter structure consists of a handle to the iterator and a model, which is a list containing the remaining elements for iteration. Because H is tree-global, exactly one Tree structure must be present ("the tree"), while the number of Snap and Iter structures is not constrained.

3.1 Specification of the ITree Interface

We now present the formal abstract specification of the ITree interface informally described in Section 2.1. The specification also contains five axioms, which are useful for a client and obligations to an implementor of the interface. The specification is parametrized over an implementation class C and the above-mentioned predicate H, and each method specification is universally quantified over the model τ, a finite set of integers and a finite set of abstract structures ϕ.

`interface ITree` {

$\{H(\{Tree(\texttt{this}, \tau)\} \uplus \phi)\}$	`contains(x)`	$\{\texttt{ret} = \texttt{x} \in \tau \wedge H(\{Tree(\texttt{this}, \tau)\} \uplus \phi)\}$
$\{H(\{Snap(\texttt{this}, \tau)\} \uplus \phi)\}$	`contains(x)`	$\{\texttt{ret} = \texttt{x} \in \tau \wedge H(\{Snap(\texttt{this}, \tau)\} \uplus \phi)\}$
$\{H(\{Tree(\texttt{this}, \tau)\} \uplus \phi)\}$	`add(x)`	$\{\texttt{ret} = \texttt{x} \notin \tau \wedge H(\{Tree(\texttt{this}, \{\texttt{x}\} \cup \tau)\} \uplus \phi)\}$
$\{H(\{Tree(\texttt{this}, \tau)\} \uplus \phi)\}$	`snapshot()`	$\{H(\{Snap(\texttt{ret}, \tau)\} \uplus \{Tree(\texttt{this}, \tau)\} \uplus \phi)\}$
$\{H(\{Snap(\texttt{this}, \tau)\} \uplus \phi)\}$	`iterator()`	$\{H(\{Iter(\texttt{ret}, [\tau])\} \uplus \{Snap(\texttt{this}, \tau)\} \uplus \phi) \wedge \texttt{ret} <: Iterator\}$

(a) $H(\{Tree(\texttt{t}, \tau)\} \uplus \phi) \vdash \texttt{t} : C$
(b) $H(\{Snap(\texttt{s}, \tau)\} \uplus \phi) \vdash \texttt{s} : C$
(c) $\tau = \tau' \wedge H(\{Tree(\texttt{t}, \tau)\} \uplus \phi) \vdash H(\{Tree(\texttt{t}, \tau')\} \uplus \phi)$
(d) $H(\{Snap(\texttt{s}, \tau)\} \uplus \phi) \vdash H(\phi)$
(e) $H(\{Iter(\texttt{it}, \alpha)\} \uplus \phi) \vdash H(\phi)$
}

These specifications can be read as follows:

- `contains` requires either a Snap or Tree structure (written as separate specifications) for the `this` handle and some set τ. The structure is unmodified in the postcondition, and the return value `ret` is true if the item `x` is in the set τ, otherwise false.
- `add` requires a Tree structure for the `this` handle and some set τ. The postcondition states that the given item `x` is added to the set τ. The return value indicates whether the tree was modified, which is the case if the item was not already present in the set τ.
- `snapshot` requires a Tree structure for the `this` handle and some set τ. The postcondition constructs a Snap structure for the returned handle `ret` and the set τ. So the Tree and the Snap structure contain the same elements.
- `iterator` requires a Snap structure for the `this` handle and some set τ. The postcondition constructs an Iter structure with the return handle and the set τ converted to an ordered list, written $[\tau]$. The returned handle conforms (written $<:$) to the Iterator specification shown in Section 3.2.

The five axioms state that (a) the static type of the tree is the given class C; (b) the static type of a snapshot is C; (c) the model τ of the tree can be replaced by an equal model τ' [1]; and we can forget about snapshots (d) and iterators (e).

In contrast to the description in Section 2.1 we leave iterators over the tree for future work. We could use the ramification operator [10] to express that any iterators over the tree become invalid when the tree is modified.

The abstract separation can be observed, e.g., in the specification of `add`: it only modifies the model of the Tree structure and does not affect the rest of the abstract structures (ϕ is preserved in the postcondition). Hence the client can reason about calls to `add` locally, independently of how many snapshots and iterators there are.

In our Coq formalization we do not have any syntax for interfaces at the specification logic level [2], but represent interfaces using Coq-level definitions. Appendix A contains the formal representations (ITree, Iterator, Stack).

3.2 Iterator Specification

Our iterator specification is also parametrized over a class IC and a predicate H, and each method specification is universally quantified over a list of integers α and a finite set of abstract structures ϕ.

```
interface Iterator<Integer> {
```
$\{H(\{Iter(\texttt{this}, \alpha)\} \uplus \phi)\}$ `hasNext()` $\{\texttt{ret} = (|\alpha| \neq 0) \wedge H(\{Iter(\texttt{this}, \alpha)\} \uplus \phi)\ \}$

$\{H(\{Iter(\texttt{this}, x :: \alpha)\} \uplus \phi)\}$ `next()` $\{\texttt{ret} = x \wedge H(\{Iter(\texttt{this}, \alpha)\} \uplus \phi)\}$

```
}
```

The specification of the Iterator interface requires an Iter structure with the `this` handle and some list α. The return value of the method `hasNext` captures whether the list α is non-empty. The Iter structure in the postcondition is not modified. The method `next` requires an Iter structure with a non-empty list $(x :: \alpha)$. The list head is returned and the model of the Iter structure is updated to the remainder of the list.

3.3 Client Code Verification

To verify the client code from Section 2.2 we assume we are given a class C such that ITree C H holds for some H and then verify the client code under the precondition $\{H(\{Tree(\texttt{t}, \{\})\})\}$.

Figure 1 gives a step-by-step proof of the client code from Section 2.2, with client code lines to the left and their postconditions to the right.

After inserting some items (line 1) to the tree, the model contains these items, {1, 2, 3}. In line 2, a snapshot `s` of the tree `t` is created. The invariant H now consists of the Tree structure and a Snap structure containing the same elements. For the client the abstract structures are disjoint, but in an implementation, they will be realized using sharing. Indeed, for the A1B1 implementation, the concrete

[1] This is explicit for technical reasons: in our implementation H is defined inside a monad [2], and the client should not have to discharge obligations inside the monad.

1:	`t.add(2);t.add(1);t.add(3);`	$\{H(\{Tree(\mathtt{t}, \{1,2,3\})\})\}$
2:	`ITree s = t.snapshot();`	$\{H(\{Tree(\mathtt{t}, \{1,2,3\})\} \uplus \{Snap(\mathtt{s}, \{1,2,3\})\})\}$
3:	`Iterator<Integer> it = s.iterator();`	$\{H(\{Tree(\mathtt{t}, \{1,2,3\})\} \uplus \{Snap(\mathtt{s}, \{1,2,3\})\} \uplus \{Iter(\mathtt{it}, [1,2,3])\})\}$
4:	`boolean lc = it.hasNext();`	$\{\mathtt{lc} = \mathtt{true} \wedge H(\{Tree(\mathtt{t}, \{1,2,3\})\} \uplus \{Snap(\mathtt{s}, \{1,2,3\})\} \uplus \{Iter(\mathtt{it}, [1,2,3])\})\}$
5:	`while (lc) {`	**invariant:** $\exists \alpha, \beta. \alpha @ \beta = [1,2,3] \wedge \mathtt{lc} = (|\beta| \neq 0) \wedge H(\{Tree(\mathtt{t}, \{1,2,3\} \cup \{3z | z \in \alpha\})\} \uplus \{Snap(\mathtt{s}, \{1,2,3\})\} \uplus \{Iter(\mathtt{it}, \beta)\})$
6:	`int x = it.next();`	$\{\alpha @ \beta = [1,2,3] \wedge \mathtt{lc} = (|\beta| \neq 0) \wedge \beta = \mathtt{x} :: \beta' \wedge H(\{Tree(\mathtt{t}, \{1,2,3\} \cup \{3z | z \in \alpha\})\} \uplus \{Snap(\mathtt{s}, \{1,2,3\})\} \uplus \{Iter(\mathtt{it}, \beta')\})\}$
7:	`t.add(x * 3);`	$\{\alpha @ \beta = [1,2,3] \wedge \mathtt{lc} = (|\beta| \neq 0) \wedge \beta = \mathtt{x} :: \beta' \wedge H(\{Tree(\mathtt{t}, \{1,2,3\} \cup \{3z | z \in \alpha\} \cup \{3\mathtt{x}\})\} \uplus \{Snap(\mathtt{s}, \{1,2,3\})\} \uplus \{Iter(\mathtt{it}, \beta')\})\}$
8:	`lc = it.hasNext();`	$\{\alpha @ \beta = [1,2,3] \wedge \mathtt{lc} = (|\beta'| \neq 0) \wedge \beta = \mathtt{x} :: \beta' \wedge H(\{Tree(\mathtt{t}, \{1,2,3\} \cup \{3z | z \in \alpha\} \cup \{3\mathtt{x}\})\} \uplus \{Snap(\mathtt{s}, \{1,2,3\})\} \uplus \{Iter(\mathtt{it}, \beta')\})\}$
9:	`}`	$\{H(\{Tree(\mathtt{t}, \{1,2,3,6,9\})\} \uplus \{Snap(\mathtt{s}, \{1,2,3\})\})\}$

Fig. 1. Client code verification

heap will be as shown in Figure 2, where all the nodes are shared between the tree and the snapshot.

In line 3 an iterator `it` over the snapshot `s` is created. To apply the call rule of the `iterator` method, only the Snap structure is taken into account, the rest (the Tree structure) is framed out inside of H (via appropriate instantiation of ϕ in the `iterator` specification). The result is that an Iter structure is constructed, whose model contains the same values as the model of the snapshot, but converted to an ordered list. We introduce the loop condition `lc` in line 4, and again use abstract framing to call `hasNext`.

Lines 5–9 contain a while loop with loop condition `lc`. The loop invariant splits the iteration list $[1,2,3]$ into the list α containing the elements already iterated over and the list β containing the remainder. The loop variable `lc` is false iff β is the empty list. The invariant H contains the Tree structure whose model is the initial set {1, 2, 3} joined with the set of the elements of α, each multiplied by 3. H also contains the Iter and the Snap structures.

We omit detailed explanation of the remaining lines of verification.

Note that in the final postcondition, the client sees two disjoint structures (axiom (e) is used to forget the empty iterator), but in the A1B1 implementation, the concrete heap will involve sharing, as shown in Figure 3. Only the left subtree is shared by the tree and the snapshot; the root and right subtree were unshared by the first call to `add` in the loop.

In summary, we have shown the following theorem, which says that given any H and any classes C and IC satisfying the ITree and Iterator interface

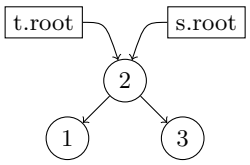


Fig. 2. Heap after snapshot construction

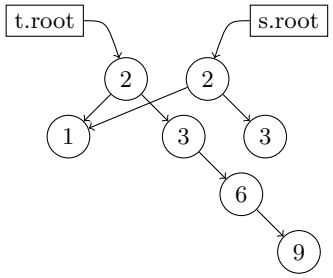


Fig. 3. Live heap after loop

specifications, the client code satisfies its specification. The postcondition states that snapshot `s` contains 1, 2 and 3, and tree `t` contains additionally 6 and 9.

Theorem 1. $\forall H.\forall C.\forall IC.ITree\ C\ H \wedge Iterator\ IC\ H$
$\vdash \{H(\{Tree(\mathtt{t}, \{\})\})\}\ client\_code\ \{H(\{Tree(\mathtt{t}, \{1,2,3,6,9\})\} \uplus \{Snap(\mathtt{s}, \{1,2,3\})\})\}$

4 Implementation A1B1

In this section we show the partial correctness verification of the A1B1 implementation with respect to the abstract specification from the previous section. This involves defining a concrete H and showing that the methods satisfy the required specifications for this concrete H. The development has been formally verified in Coq (as has the client program verification above).

The Coq formalization uses a shallow embedding of higher-order separation logic, developed for verification of OO programs using interfaces. See [2].

Invariant H is radically different depending on whether snapshots of the tree are present or not. The reason is that method `add` mutates the existing tree if there are no snapshots present, see Section 5 for details. Here we focus on the case where snapshots are present.

The A1B1Tree class stores its data in three fields: the `root` node, a boolean field `isSnapshot`, indicating whether it is a snapshot, and a field `hasSnapshot`, indicating whether it has snapshots. The stamp field mentioned in Section 2.3 is only required for iterators over the tree and so not further discussed here.

The Node class is a nested class of the A1B1Tree with three fields, `item` containing its value, and a handle to the right (`rght`) and left (`left`) subtree.

In the following we use standard separation logic connectives, in particular the separating conjunction $*$ and the points to predicate $\mapsto$.

We now define our concrete H and also the realization of the abstract structures. We first explain the realization of Tree and Snap; the Iter structure is described in Section 4.1. Recall that ϕ ranges over finite sets of abstract structures (Tree, Snap, Iter), with exactly one Tree structure, and recall that H, given a ϕ, returns a separation logic predicate. The definition of H is:

$$H(\phi) \triangleq \exists\sigma.wf(\sigma) \wedge heap(\sigma)\ *\ \sigma \vDash \phi$$

Here σ is a finite map of type ptr $\to$ ptr $\times \mathbb{Z} \times$ ptr, with ptr being the type of Java pointers (handles), corresponding to the Node class. The map σ must be well-formed (pure predicate $wf(\sigma)$), which simply means that all pointers in the codomain of σ are either `null` or in the domain of σ.

The *heap* function maps σ to a separation logic predicate, which describes the realization of σ as a linked structure in the concrete heap, starting with $\top$:

$$\begin{aligned} heap(\sigma) \triangleq fold\ (\lambda \mathtt{p}\ n\ Q.\ match\ n\ with\ (\mathtt{pl}, v, \mathtt{pr}) \Rightarrow \\ \mathtt{p.left} \mapsto \mathtt{pl}\ *\ \mathtt{p.item} \mapsto v\ *\ \mathtt{p.rght} \mapsto \mathtt{pr}\ *\ Q)\ \top\ \ \sigma \end{aligned}$$

Finally, we present the definition of $\sigma \vDash \phi$ (we defer the definition of $\sigma \vDash \{Iter(\_,\_)\}$ to the following subsection):

$$\begin{aligned} \sigma \vDash \phi \uplus \psi &\triangleq \sigma \vDash \phi\ *\ \sigma \vDash \psi \\ \sigma \vDash \{Tree(\mathtt{ptr}, \tau)\} &\triangleq \exists \mathtt{p}. Node(\sigma, \mathtt{p}, \tau) \wedge \mathtt{ptr.root} \mapsto \mathtt{p}\ * \\ &\quad \mathtt{ptr.isSnapshot} \mapsto \mathtt{false}\ *\ \mathtt{ptr.hasSnapshot} \mapsto \mathtt{true} \\ \sigma \vDash \{Snap(\mathtt{ptr}, \tau)\} &\triangleq \exists \mathtt{p}. Node(\sigma, \mathtt{p}, \tau) \wedge \mathtt{ptr.root} \mapsto \mathtt{p}\ * \\ &\quad \mathtt{ptr.isSnapshot} \mapsto \mathtt{true}\ *\ \mathtt{ptr.hasSnapshot} \mapsto \mathtt{false} \end{aligned}$$

The spatial structure of all the nodes is covered by $heap(\sigma)$ so $\sigma \vDash \phi$ just needs to describe the additional heap taken up by Tree, Snap, and Iter structures.

The pure $Node$ predicate is defined inductively on τ below. It is used to express that τ is the finite set of items reachable from $\mathtt{p}$ in σ.

$$\begin{aligned} Node(\sigma, \mathtt{p}, \tau) \triangleq\ & \big(\mathtt{p} = \mathtt{null} \wedge \tau = \{\}\big) \vee \\ & \big(\mathtt{p} \in dom(\sigma)\ \wedge \exists \mathtt{pl}, v, \mathtt{pr}.\ \sigma[\mathtt{p}] = (\mathtt{pl}, v, \mathtt{pr}) \wedge \\ & \exists \tau_l, \tau_r. \tau = \tau_l \cup \{v\} \cup \tau_r \wedge \\ & (\forall x \in \tau_l. x < v) \wedge (\forall x \in \tau_r. x > v) \wedge \\ & Node(\sigma, \mathtt{pl}, \tau_l) \wedge Node(\sigma, \mathtt{pr}, \tau_r)\big) \end{aligned}$$

The sortedness constraint (a strict total order) in the $Node$ predicate enforces implicitly that σ has the right shape: σ cannot contain cycles and the left and right subtrees must be disjoint. The set τ is split into three sets, one with strictly smaller elements (τ_l), the singleton v and with strictly bigger elements (τ_r).

4.1 Iterator

The TreeIterator class implements the Iterator interface. It contains a single field, `context`, which is a stack of Node objects.

The constructor of the TreeIterator pushes all nodes on the leftmost path of the tree onto the stack. The method `next` pops the top node from the stack and returns the value held in that node. Before returning, it pushes the leftmost path of the node's right subtree (if any) onto the stack. The method `hasNext` returns true if and only if the stack is empty.

The verification of the iterator depends on the following specification of a stack class, generic in C. The specification is parametrized over a representation type T and existentially over a representation predicate SR (of type classname $\to$

$(\mathsf{val} \to T \to \mathsf{HeapAsn}) \to \mathsf{val} \to T^* \to \mathsf{HeapAsn}$). The second argument is the predicate P (of type $\mathsf{val} \to T \to \mathsf{HeapAsn}$) , which holds for every stack element. This specification is kept in the style of [17], although we use a different logic.

```
class Stack<C> {
```

$\top$	`new()` $SR\ C\ P$ `ret` nil
$SR\ C\ P$ `this` α	`empty()` `ret` $= (\alpha = \mathsf{nil}) \wedge SR\ C\ P$ `this` α
$SR\ C\ P$ `this` $\alpha * P$ `x` $t \wedge$ `x` $: C$	`push(x)` $SR\ C\ P$ `this` $(t :: \alpha)$
$SR\ C\ P$ `this` $(t :: \alpha)$	`pop()` P `ret` $t\ * SR\ C\ P$ `this` α
$SR\ C\ P$ `this` $(t :: \alpha)$	`peek()` P `ret` $t * (\forall u. P$ `ret` $u \mathrel{-\!\!*} SR\ C\ P$ `this` $(u :: \alpha))$

(a) $P\ v\ t \vdash P'\ v\ t \implies SR\ C\ P\ v\ \alpha \vdash SR\ C\ P'\ v\ \alpha$

```
}
```

For the purpose of specifying the iterators over snapshotable trees, we instantiate the type T with $\mathbb{Z}^*$; the model of a node on the stack is a list of integers. Intuitively, this list corresponds to the node value and the element list of its right subtree. The iterator is modelled as a list that is equal to the concatenation of the elements of the stack. We also require that the topmost element of the stack is nonempty (if present). This intuition is formalized in the interpretation of the Iter structure, where SR is a representation predicate of a stack:

$$\sigma \vDash \{Iter(\mathtt{p}, \alpha)\} \triangleq \exists \mathtt{st}.\ \mathtt{p.context} \mapsto \mathtt{st}\ *\ \exists \beta. stack\_inv(\beta, \alpha) \wedge SR\ \mathtt{Node}\ (NS\ \sigma)\ \mathtt{st}\ \beta.$$

To make this definition complete, we provide the definitions of *stack_inv*, which connects the representation of the stack with the representation of the iterator, and the definition of the NS predicate.

$$stack\_inv(xss, ys) \triangleq ys = \mathsf{concat}(xss) \wedge \begin{cases} \top & \text{iff } xss = \mathsf{nil} \\ xs \neq \mathsf{nil} & \text{iff } xss = xs :: xss' \end{cases}$$

$$NS\ \sigma\ \mathtt{node}\ \alpha \triangleq Node(\sigma, \mathtt{node}, \tau) \wedge\ \alpha = [\{x \in \tau | x \geq \mathtt{node.item}\}]$$

These definitions, along with an assumption that SR is the representation predicate of Stack (i.e., fulfills all the method specifications and axioms of *Stack_spec*) suffice to show the correctness of Iter-dependent methods. The axiom present in *Stack_spec* is needed to preserve iterators if some new memory is added to σ: it allows us to replace $(NS\ \sigma)$ with $(NS\ \sigma')$ as a representation predicate of stack objects under certain side conditions.

5 On the Verification of Implemented Code

We now give an intuitive description of how the A1B1 implementation was verified, given the concrete H defined above. We verified the complete implementation in Coq but only discuss the `add` method here. We used Kopitiam [13] to transform the Java code into SimpleJava, the fragment represented in Coq.

Method `add` calls method `addRecursive` with the root node to insert the item into the binary tree, respecting the ordering. Method `addRecursive`, shown below, must handle several cases:

- if there are no snapshots present, then
  - if the item `x` is already in the tree, then the heap is not modified.
  - if the item `x` is not in the tree, then a new node is allocated and destructively inserted into the tree.
- if there are snapshots present, then
  - if the item `x` is already in the tree, then the heap is not modified.
  - if the item `x` is not in the tree, then a new node is allocated and every node on the path from the root to the added node is replicated, so that the snapshots are unimpaired.

The implementation of `addRecursive` walks down the tree until a node with the same value, or a leaf, is reached. It uses the call stack to remember the path in the tree. If a node was added, either the entire path from the root to the added node is duplicated (if snapshots are present) or the handles to the left or right subtree are updated (happens destructively exactly once, the parent of the added node updates its left or right handle, previously pointing to `null`):

```
Node addRecursive (Node node, int item, RefBool updated) {
  Node res = node;
  if (node == null) {
    updated.value = true;
    res = new Node(item);
  } else {
    if (item < node.item) {
      Node newLeft = addRecursive(node.left, item, updated);
      if (updated.value && this.hasSnapshot)
        res = new Node(newLeft, node.item, node.rght);
      else
        node.left = newLeft;
    } else if (node.item < item) {
      Node newRght = addRecursive(node.rght, item, updated);
      if (updated.value && this.hasSnapshot)
        res = new Node(node.left, node.item, newRght);
      else
        node.rght = newRght;
    } //else item == node.item so no update
  }
  return res;
}
```

We now show the pre- and postcondition of `addRecursive` for the two cases where snapshots are present. If the item is already present in the tree, the pre- and postcondition are equal:

$$\{\texttt{updated.value} \mapsto \texttt{false} \ * \ \texttt{this.hasSnapshot} \mapsto \texttt{true} \ * \ heap(\sigma) \ * \ wf(\sigma) \wedge Node(\sigma, \texttt{node}, \tau) \wedge \texttt{item} \in \tau\}$$
$$\texttt{addRecursive(node, item, updated)}$$
$$\{\texttt{updated.value} \mapsto \texttt{false} \ * \ \texttt{this.hasSnapshot} \mapsto \texttt{true} \ * \ heap(\sigma) \ * \texttt{ret} = \texttt{node}\}$$

The postcondition in the case that the item is added to the tree extends the map σ to σ', for which the heap layout and the well-formedness condition must hold. The Node predicate uses σ' and the finite set is extended with `item`:

$$\begin{gathered}\{\texttt{updated.value} \mapsto \texttt{false} \ * \ \texttt{this.hasSnapshot} \mapsto \texttt{true} \ * \\ heap(\sigma) \ * \ wf(\sigma) \wedge Node(\sigma, \texttt{node}, \tau) \wedge \texttt{item} \notin \tau\} \\ \texttt{addRecursive(node, item, updated)} \\ \{\texttt{updated.value} \mapsto \texttt{true} \ * \ \texttt{this.hasSnapshot} \mapsto \texttt{true} \ * \\ \exists \sigma'. \sigma \subseteq \sigma' \wedge heap(\sigma') \ * \ wf(\sigma') \wedge Node(\sigma', \texttt{ret}, \{\texttt{item}\} \cup \tau)\}\end{gathered}$$

The call to `addRecursive` inside of `add` is verified for each specification of `addRecursive` independently.

To summarize Sections 4 and 5, we state the following theorem, which says there exists an H that given a stack fulfilling the stack specification, the TreeIterator class meets the Iterator specification and the A1B1 class meets the ITree specification, and the constructor for the A1B1Tree establishes the H predicate.

Theorem 2. $\exists H. Stack\_spec \vdash Iterator\ TreeIterator\ H \wedge ITree\ A1B1\ H \wedge \{\top\}\ A1B1Tree()\ \{H(\{Tree(\texttt{ret}, \{\})\})\}$

The client code, independently verified, can be safely linked with the A1B1 implementation!

6 Related Work

Malecha and Morrisett [12] presented a formalization of a Ynot implementation of B-trees with an iterator method. In their case, the iterator and the tree also share data in the concrete heap. However, they can only reason about "single-threaded" uses of trees and iterators: their specification of the iterator method transforms the abstract tree predicate into an abstract iterator predicate, which prohibits calling tree methods until the client turns the iterator back into a tree. In our setup, we have one tree, but multiple snapshots and iterators, and the tree can be updated after an iterator has been created. To permit sharing between a tree and an iterator, Malecha and Morrisett use fractional permissions, where we use the H predicate. They work in an axiomatic extension of Coq, whereas our proofs are done in a shallowly embedded program logic, since our programs are written in an existing programming language (Java).

Dinsdale-Young et al. [5] present another approach to reasoning about shared data structures, which gives the client a fiction of disjointness. Roughly speaking, they define a new abstract program logic for each module (they can be combined) for abstract client reasoning. Their approach allows one to give a client specification similar to ours, but without using the H and with the abstract structures (Tree / Snap / Iter) being predicates in the (abstract) program logic. This has the advantage that one can use ordinary framing for local reasoning.

Dinsdale-Young et al. [4] also presented an approach to reasoning about sharing. Sharing can happen in certain regions, and the module implementor

has to define a protocol that describes how data in the shared region can evolve. What corresponds to our abstract structures can now be seen as separation logic predicates and thus one can use ordinary framing for local reasoning.

In both approaches [5] and [4] the module implementor has more proof obligations than in our approach: In [5] he must show that the abstract operations satisfy some conditions related to the realization of the abstract structures in the concrete heap. In [4] she must show related properties phrased in terms of certain stability conditions.

Compared to the work of Dinsdale-Young et al., our approach has the advantage that it is arguably simpler, with no need to introduce new separation (or context) algebras for the modules. That is why we could build our formalization on an implementation of standard separation logic in Coq.

Rustan Leino made a solution for a custom implementation of this data structure (A1B1) using Dafny [11]. Dafny verifies that if a snapshot is present, the nodes are shared and not mutated by the tree operations. His solution does not (yet) verify the content of the tree, snapshots or iterators. Our verification specifies the concrete heap layout. Dafny does not support abstract specification due to the lack of inheritance. The trusted code base is different: Dafny relies on Boogie, Z3 and the CLR, whereas our proof trusts Coq.

7 Conclusion and Future Work

We have presented snapshotable trees as a challenge for formalized modular reasoning about mutable data structures that use sharing extensively, and given an abstract specification of the ITree interface. Moreover, we have presented a formalization of the A1B1 implementation of snapshotable trees.

The overall size of the formalization effort is roughly 5000 lines of Coq code and it takes 2 hours to `Qed` the proofs. This is quite big compared to other formalization efforts of imperative programs in Coq, such as Hoare Type Theory / Ynot [14, 15]. The main reason is that we are working in a shallowly embedded program logic for a Java-like language, whereas Hoare Type Theory / Ynot is an axiomatic extension of Coq. Thus our formalization includes both the operational semantics of the Java subset and the soundness theorems for the program logic; also, Java program variables cannot simply be represented by Coq variables.

We also plan to verify the even subtler implementations A1B2, A2B1 and A2B2, which are expected to provide further insight into the challenges of dealing with shared mutable data and unobservable state changes. Through those more complex applications of separation logic we hope to learn more about desirable tool support, including how to automate the "obvious" reasoning that currently requires much thought and excessive amounts of proof code. Although we have not formally verified these implementations yet, we are fairly certain they would match the interface specification presented in Section 3. In all four implementations the tree is conceptually separate from its snapshots, which is the property required by the interface, and the invariant H allows us to describe the heap layout very precisely, using techniques shown in Section 4.

Finally, we would like to explore how to combine the advantages of our approach and those of Dinsdale-Young's approach discussed above.

References

1. A. Andersson. Balanced search trees made simple. In F. Dehne et al., editors, *Algorithms and Data Structures. LNCS 709*, pages 60–71. Springer-Verlag, 1993.
2. J. Bengtson, J. B. Jensen, F. Sieczkowski, and L. Birkedal. Verifying object-oriented programs with higher-order separation logic in Coq. In *ITP 2011*, 2011.
3. B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines, higher-order separation logic, and abstraction. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.
4. T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In T. DHondt, editor, *ECOOP 2010*, volume 6183 of *LNCS*, pages 504–528. Springer Berlin / Heidelberg, 2010.
5. T. Dinsdale-Young, P. Gardner, and M. Wheelhouse. Abstraction and refinement for local reasoning. In *VSTTE*, pages 199–215, Berlin, Heidelberg, 2010. Springer.
6. J. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan. Making data structures persistent. *Journal of Computer and Systems Sciences*, 38(1):86–124, 1989.
7. L. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th FCS, Ann Arbor, Michigan*, pages 8–21, 1978.
8. N. Kokholm and P. Sestoft. The C5 Generic Collection Library for C# and CLI. Technical Report ITU-TR-2006-76, IT University of Copenhagen, January 2006.
9. N. Krishnaswami. *Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic*. PhD thesis, Carnegie Mellon University, 2011. Forthcoming.
10. N. R. Krishnaswami, L. Birkedal, and J. Aldrich. Verifying event-driven programs using ramified frame properties. In *TLDI*, pages 63–76. ACM, 2010.
11. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning. LNCS 6355*, pages 348–370, 2010.
12. G. Malecha and G. Morrisett. Mechanized verification with sharing. In *7th International Colloquium on Theoretical Aspects of Computing*, Sept. 2010.
13. H. Mehnert. Kopitiam: Modular incremental interactive full functional static verification of java code. In M. Bobaru, K. Havelund, G. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, volume 6617, pages 518–524. Springer, 2011.
14. A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: dependent types for imperative programs. In J. Hook and P. Thiemann, editors, *Proc. of 13th ACM ICFP 2008*, pages 229–240. ACM, 2008.
15. A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the verification of heap-manipulating programs. In *Proceedings of POPL*, 2010.
16. M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *Proceedings of POPL*, pages 247–258, 2005.
17. R. Petersen, L. Birkedal, A. Nanevski, and G. Morrisett. A realizability model for impredicative hoare type theory. In S. Drossopoulou, editor, *ESOP 2008*, volume 4960 of *Lecture Notes in Computer Science*, pages 337–352. Springer, 2008.
18. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. *IEEE Proc. of 17th Symp. on Logic in CS*, Nov 2002.
19. R. Sedgewick. Left-leaning red-black trees. At http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf.
20. K. Svendsen, L. Birkedal, and M. Parkinson. Verifying generics and delegates. In *ECOOP'10*, pages 175–199. Springer-Verlag, 2010.

A Appendix

We define here the ITree and the Iterator interface specification as Coq definitions, as well as the Stack class. We use the name *SPred* for the finite set of abstract structures containing exactly one Tree structure and any number of Snap and Iter structures.

The notation $\widehat{f}$ lifts the function f such that it operates on expressions rather than values.

A detailed explanation of the notation and of lifting can be found in [2].

$$
\begin{aligned}
\mathit{ITree} \triangleq\ & \lambda C : \mathit{classname}.\ \lambda H : \mathcal{P}_{fin}(\mathit{SPred}) \to \mathit{HeapAsn}. \\
& (\forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\mathtt{contains}(\mathtt{this}, \mathtt{x}) \mapsto \\
& \quad \{\widehat{H}(\{\widehat{\mathsf{Tree}}(\mathtt{this}, \tau)\} \uplus \phi)\}\_\{\mathtt{r}.\ \widehat{H}(\{\widehat{\mathsf{Tree}}(\mathtt{this}, \tau)\} \uplus \phi) \wedge \mathtt{r} = (\mathtt{x} \in \tau)\}) \\
& \wedge (\forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\mathtt{contains}(\mathtt{this}, \mathtt{x}) \mapsto \\
& \quad \{\widehat{H}(\{\widehat{\mathsf{Snap}}(\mathtt{this}, \tau)\} \uplus \phi)\}\_\{\mathtt{r}.\ \widehat{H}(\{\widehat{\mathsf{Snap}}(\mathtt{this}, \tau)\} \uplus \phi) \wedge \mathtt{r} = (\mathtt{x} \in \tau)\}) \\
& \wedge (\forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\mathtt{add}(\mathtt{this}, \mathtt{x}) \mapsto \\
& \quad \{\widehat{H}(\{\widehat{\mathsf{Tree}}(\mathtt{this}, \tau)\} \uplus \phi)\}\_\{\mathtt{r}.\ \widehat{H}(\{\widehat{\mathsf{Tree}}(\mathtt{this}, \{\mathtt{x}\} \cup \tau)\} \uplus \phi) \wedge \mathtt{r} = (\mathtt{x} \notin \tau)\}) \\
& \wedge (\forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\mathtt{snapshot}(\mathtt{this}) \mapsto \\
& \quad \{\widehat{H}(\{\widehat{\mathsf{Tree}}(\mathtt{this}, \tau)\} \uplus \phi)\}\_\{\mathtt{r}.\ \widehat{H}(\{\widehat{\mathsf{Tree}}(\mathtt{this}, \tau), \widehat{\mathsf{Snap}}(\mathtt{r}, \tau)\} \uplus \phi)\}) \\
& \wedge (\forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\mathtt{iterator}(\mathtt{this}) \mapsto \\
& \quad \{\widehat{H}(\{\widehat{\mathsf{Snap}}(\mathtt{this}, \tau)\} \uplus \phi)\}\_\{\mathtt{r}.\ \exists IC : \mathit{classname}.\ \mathit{Iterator}\ IC\ H \wedge \mathtt{r} : IC \wedge \\
& \qquad \widehat{H}(\{\widehat{\mathsf{Snap}}(\mathtt{this}, \tau), \widehat{\mathsf{Iter}}(\mathtt{r}, [\tau])\} \uplus \phi)\}) \\
& \wedge (\forall v : \mathit{val}.\ \forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}). \\
& \quad (H(\{\mathit{Tree}(v, \tau)\} \uplus \phi) \implies v : C) \wedge (H(\{\mathit{Snap}(v, \tau)\} \uplus \phi) \implies v : C)) \\
& \wedge (\forall v : \mathit{val}.\ \forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}). \\
& \quad (H(\{\mathit{Snap}(v, \tau)\} \uplus \phi) \vdash H(\phi))) \\
& \wedge (\forall v : \mathit{val}.\ \forall \alpha : \mathbb{Z}^*.\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}). \\
& \quad (H(\{\mathit{Iter}(v, \alpha)\} \uplus \phi) \vdash H(\phi))) \\
& \wedge (\forall v : \mathit{val}.\ \forall \tau, \tau' : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}). \\
& \quad \tau = \tau' \implies (H(\{\mathit{Tree}(v, \tau)\} \uplus \phi) \vdash H(\{\mathit{Tree}(v, \tau')\} \uplus \phi)))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Iterator} \triangleq\ & \lambda C : \mathit{classname}.\ \lambda H : \mathcal{P}_{fin}(\mathit{SPred}) \to \mathit{HeapAsn}. \\
& (\forall \alpha : \mathbb{Z}^*.\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\mathtt{hasNext}(\mathtt{this}) \mapsto \\
& \quad \{\widehat{H}(\{\widehat{\mathsf{Iter}}(\mathtt{this}, \alpha)\} \uplus \phi)\}\_\{\mathtt{r}.\ \widehat{H}(\{\widehat{\mathsf{Iter}}(\mathtt{this}, \alpha)\} \uplus \phi) \wedge \mathtt{r} = (\alpha \neq \mathsf{nil})\}) \\
& \wedge (\forall x : \mathbb{Z}.\ \forall \alpha : \mathbb{Z}^*.\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\mathtt{next}(\mathtt{this}) \mapsto \\
& \quad \{\widehat{H}(\{\widehat{\mathsf{Iter}}(\mathtt{this}, x{::}\alpha)\} \uplus \phi)\}\_\{\mathtt{r}.\ \widehat{H}(\{\widehat{\mathsf{Iter}}(\mathtt{this}, \alpha)\} \uplus \phi) \wedge \mathtt{r} = x\})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Stack\_spec} \triangleq\ & \forall T : \mathit{Type}.\\
& \exists SR : \mathit{classname} \to (\mathit{val} \to T \to \mathit{HeapAsn}) \to \mathit{val} \to T^* \to \mathit{HeapAsn}.\\
& (\forall C : \mathit{classname}.\ \forall P : \mathit{val} \to T \to \mathit{HeapAsn}.\\
& \quad \texttt{Stack::new}() \mapsto \{\top\}\_\{\texttt{r}.\ \widehat{SR}\ C\ P\ \texttt{r}\ \mathsf{nil}\}\\
& \wedge (\forall \alpha : T^*.\ \texttt{Stack::empty}(\texttt{this}) \mapsto\\
& \quad \{\widehat{SR}\ C\ P\ \texttt{this}\ \alpha\}\_\{\texttt{r}.\ \widehat{SR}\ C\ P\ \texttt{this}\ \alpha \wedge \texttt{r} = (\alpha = \mathsf{nil})\})\\
& \wedge (\forall \alpha : T^*.\ \forall t : T.\ \texttt{Stack::push}(\texttt{this}, \texttt{x}) \mapsto\\
& \quad \{\widehat{SR}\ C\ P\ \texttt{this}\ \alpha * \widehat{P}\ \texttt{x}\ t \wedge \texttt{x} : C\}\_\{\widehat{SR}\ C\ P\ \texttt{this}\ (t :: \alpha)\})\\
& \wedge (\forall \alpha : T^*.\ \forall t : T.\ \texttt{Stack::pop}(\texttt{this}, \texttt{x}) \mapsto\\
& \quad \{\widehat{SR}\ C\ P\ \texttt{this}\ (t :: \alpha)\}\_\{\texttt{r}.\ \widehat{P}\ \texttt{r}\ t * \widehat{SR}\ C\ P\ \texttt{this}\ \alpha\})\\
& \wedge (\forall \alpha : T^*.\ \forall t : T.\ \texttt{Stack::peek}(\texttt{this}, \texttt{x}) \mapsto\\
& \quad \{\widehat{SR}\ C\ P\ \texttt{this}\ (t :: \alpha)\}\_\{\texttt{r}.\ \widehat{P}\ \texttt{r}\ t *\\
& \quad\ (\forall u : T.\ \widehat{P}\ \texttt{r}\ u \mathrel{-\!\!*} \widehat{SR}\ C\ P\ \texttt{this}\ (u :: \alpha))\}))\\
& \wedge (\forall C : \mathit{classname}.\ \forall P, P' : \mathit{val} \to T \to \mathit{HeapAsn}.\\
& \quad (\forall v : \mathit{val}.\ \forall t : T.\ (P\ v\ t \vdash P'\ v\ t)) \implies\\
& \quad\ \forall v : \mathit{val}.\ \forall \alpha : T^*.\ (SR\ C\ P\ v\ \alpha \vdash SR\ C\ P'\ v\ \alpha))
\end{aligned}
$$

A Separation Logic for Fictional Sequential Consistency

Filip Sieczkowski
IT University of Copenhagen
fisi@itu.dk

Kasper Svendsen
Aarhus University
ksvendsen@cs.au.dk

Lars Birkedal
Aarhus University
birkedal@cs.au.dk

Abstract

Modern multiprocessors and mainstream concurrent programming languages do not provide the strong *sequentially consistent* shared memory that has been assumed by most work on program logics for concurrent programs. Rather, they typically implement *weak* memory models and the programmer can observe reorderings of memory operations caused by the weak behavior. In particular, programmers have to reason about weak behavior when implementing low-level efficient algorithms, e.g., synchronization primitives. However, in practice, low-level algorithms are often implemented in such a way that clients do not have to reason about weak behavior; instead clients can pretend that they are working with a sequentially consistent model.

In this paper we present a new program logic, iCAP-TSO, for a TSO memory model. The logic supports formal reasoning at both levels of abstraction that programmers use informally. Our logic can be used to verify efficient low-level implementations of libraries against specifications that provide clients with a *fiction of sequential consistency*. Hence client-side reasoning can be done as in earlier program logics that assumed a sequentially consistent memory model.

1. Introduction

What? Modern multiprocessors and mainstream concurrent programming languages do not provide the strong *sequentially consistent* (SC) shared memory that has been assumed by most work on program logics for concurrent programs. Rather, they typically implement *weak* memory models with observable buffering and reorderings of memory operations. In the case of the Total Store Order (TSO) memory model, each thread is connected to main memory via a FIFO store buffer that buffers writes before committing them to main memory.

It is possible to extend existing program logics to a setting with a TSO memory model by reasoning explicitly about these store buffers. Ridge [14] defines a Rely-Guarantee-based program logic for reasoning about x86-TSO assembly code and Wehrman [18] defines a separation logic for a low-level language with a TSO memory model. Both of these logics can be used to reason about TSO programs that exhibit *racy* behavior. Importantly, large classes of programs (e.g., well-synchronized programs) only exhibit sequentially consistent behavior. Intuitively, for most programs, explicit reasoning about store buffers is thus unnecessary and if we are forced to reason about store buffers, we will never be able to *scale* our techniques to *realistic* systems. Ridge's and Wehrman's logics both suffer from this scalability issue, by requiring explicit reasoning about store buffers even for well-behaved clients. The main challenge is thus to develop a program logic that supports *simple* high-level reasoning for well-behaved clients *and* low-level reasoning about *racy* library implementations.

In this paper we present a new program logic, iCAP-TSO, for a TSO memory model. This logic features *two* higher-order separation logics, a TSO logic for reasoning about racy low-level code and an SC logic with *standard* separation logic assertions and proof rules for reasoning about well-behaved client code. Crucially, our logic supports *simple* reasoning about programs that only exhibit sequentially consistent behaviors despite using libraries with *racy* implementations. Intuitively, these libraries provide clients with a *fiction of sequential consistency* and our logic exploits this to simplify reasoning about clients.

How? In the TSO memory model, each thread is connected to main memory via a FIFO store buffer, modeled as a sequence of (address, value) pairs, see, e.g., [15]. When a value is written to an address, the write is recorded in the writing thread's store buffer. Threads can commit these buffered writes to main memory at any point in time. When reading from a location, threads first consult their own store buffer and only consult main memory if their own store buffer does not contain a buffered write for the given location. If it does, the thread reads the value of the last buffered write in its own store buffer. Each thread thus has its own *subjective* view of the current state of memory, which might differ from other threads.

In contrast, in a sequentially consistent memory model, threads read and write directly to main memory and thus share an *objective* view of the current state of the memory. In separation logics for languages with sequentially consistent memory models we thus use assertions such as $\mathsf{x} \mapsto 1$, which express an *objective* property about the value of location x. Since in the TSO setting each thread has a subjective view of the state, in order to preserve the standard proof rules for reading and writing, we need a *subjective* interpretation of pre- and postconditions.

In our SC logic we thus have specifications of the form

$$[\mathsf{P}]\ \mathsf{e}\ [\mathsf{r}.\mathsf{Q}],$$

which express that if e is executed by thread t from an initial state that satisfies P *from the point of view* of t and e terminates with value v, then the terminal state satisfies $\mathsf{Q}[\mathsf{v}/\mathsf{r}]$ from the point of view of thread t. Informally, an assertion P holds from point of view of a thread t if P's assertion about the heap hold from the point of view of main memory and t's store buffer and no other thread's store buffer contains pending writes to these parts of the heap. In particular, $\mathsf{x} \mapsto \mathsf{v}$ holds from the point of view of thread t, if the value of x that t can observe is v. We shall see that this interpretation justifies the standard separation logic read and write rules.

What about transfer of resources? In separation logics for sequentially consistent memory models, assertions about resources are objective and can thus be transferred freely. However, since assertions in the SC logic are interpreted subjectively, they may not hold from the point of view of other threads. Clearly, to transfer resources between threads, their views of the resources must match. In general, we thus cannot reason directly about code that facilitates resource transfer in the SC logic. To reason about such code, we use the TSO logic, which allows us to reason about the complete TSO machine state, including store buffers. Importantly, in cases where the implementation that facilitates the resource transfer between two threads *does* ensure that their views match, we can verify the implementation against an SC specification! This gives us a *fiction of sequential consistency* and allows us to reason about *clients* using the SC logic.

Example. To illustrate, consider a simple spin-lock library with acquire and release methods. We can specify the lock in the SC logic as follows.

$$
\begin{array}{l}
\exists \mathsf{isLock}, \mathsf{locked} : \mathsf{Prop}_{\mathsf{sc}} \times \mathsf{Val} \to \mathsf{Prop}_{\mathsf{sc}}. \\
\quad \forall \mathsf{R} : \mathsf{Prop}_{\mathsf{sc}}.\ \mathsf{stable}(\mathsf{R}) \ \Rightarrow \\
\qquad [\mathsf{R}]\ \mathsf{Lock}()\ [\mathsf{r}.\ \mathsf{isLock}(\mathsf{R}, \mathsf{r})] \\
\qquad \wedge\ [\mathsf{isLock}(\mathsf{R}, \mathbf{this})]\ \mathsf{Lock.acquire}()\ [\mathsf{locked}(\mathsf{R}, \mathbf{this}) * \mathsf{R}] \\
\qquad \wedge\ [\mathsf{locked}(\mathsf{R}, \mathbf{this}) * \mathsf{R}]\ \mathsf{Lock.release}()\ [\top] \\
\qquad \wedge\ \mathsf{valid}(\forall \mathsf{x} : \mathsf{Val}.\ \mathsf{isLock}(\mathsf{R}, \mathsf{x}) \Leftrightarrow \mathsf{isLock}(\mathsf{R}, \mathsf{x}) * \mathsf{isLock}(\mathsf{R}, \mathsf{x})) \\
\qquad \wedge\ \forall \mathsf{x} : \mathsf{Val}.\ \mathsf{stable}(\mathsf{isLock}(\mathsf{R}, \mathsf{x})) \wedge \mathsf{stable}(\mathsf{locked}(\mathsf{R}, \mathsf{x}))
\end{array}
$$

Here $\mathsf{Prop}_{\mathsf{sc}}$ is the type of propositions of the SC logic, and isLock and locked are thus abstract representation predicates. $\mathsf{isLock}(\mathsf{R}, \mathsf{x})$ expresses that x is a lock protecting the resource invariant R and $\mathsf{locked}(\mathsf{R}, \mathsf{x})$ expresses that the lock x is indeed locked. Acquiring the lock grants ownership of R, while releasing the lock requires the client to relinquish ownership of R. Since the resource invariant R is universally quantified, this is a very strong specification; in particular, the client is free to instantiate R with any SC proposition. This specification requires the resource invariant to be stable, $\mathsf{stable}(\mathsf{R})$. The reason is that R could in general refer to shared resources and to reason about shared resources we need to ensure we only use assertions that are closed under interference from the environment. This is what stability expresses.

Note that this specification is expressed in the SC logic and the specification of the acquire method thus grants ownership of the resource R from the caller's point of view. Likewise, the release method only requires that R holds from the caller's point of view. This specification thus provides a fiction of sequential consistency, by allowing transfer of SC resources. Crucially, since the lock specification is an SC specification, we can reason about well-synchronized clients *entirely* using the standard proof rules of the SC logic. We illustrate this by verifying a shared bag in Section 3.

Using the TSO logic we can verify that an *efficient* spin-lock implementation satisfies this specification. The spin-lock that we verify is inspired by the Linux spin-lock implementation [1] which allows the release to be buffered. To verify the implementation we must prove that between releasing and acquiring the lock, the releasing and acquiring threads' views of the resource R match. Intuitively, this is the case because if R holds from the point of view of the releasing thread, once the buffered release makes it to main memory, R holds objectively. This style of reasoning relies on the ordering of buffered writes. To capture this style of reasoning, we introduce a new operator in the TSO logic for expressing such ordering dependencies. In Section 5 we illustrate how to use the TSO logic to verify the spin-lock.

$$
\begin{array}{l}
\mathsf{Val} \ni \mathsf{v} ::= \mathsf{x} \mid \mathbf{null} \mid \mathbf{this} \mid \mathsf{o} \mid \mathsf{n} \mid \mathsf{b} \mid \mathbf{()} \\
\mathsf{Exp} \ni \mathsf{e} ::= \mathsf{v} \mid \mathbf{let}\ \mathsf{x} = \mathsf{e}_1\ \mathbf{in}\ \mathsf{e}_2 \mid \mathbf{if}\ \mathsf{v}\ \mathbf{then}\ \mathsf{e}_1\ \mathbf{else}\ \mathsf{e}_2 \mid \mathbf{new}\ \mathsf{C}(\bar{\mathsf{v}}) \\
\qquad \mid \mathsf{v.f} \mid \mathsf{v}_1\mathsf{.f} := \mathsf{v}_2 \mid \mathsf{v.m}(\bar{\mathsf{v}}) \mid \mathbf{CAS}(\mathsf{v}_1\mathsf{.f}, \mathsf{v}_2, \mathsf{v}_3) \mid \mathbf{fork}(\mathsf{v.m})
\end{array}
$$

Figure 1. Syntax of the programming language. In the definition of values, n ranges over machine integers, b over booleans, and o over object references. In the definition of expressions, f ranges over the field names, and m over the method names.

Technical Overview. While our proof system consists of two logics, SC triples are in fact interpreted as TSO triples, through an embedding that formalizes our subjective interpretation of SC assertions. Soundness of the standard read and write rules in the SC logic follow from this subjective interpretation. Soundness of the standard structural rules in the SC logic follow from the fact that this embedding is well-behaved with respect to the connectives and quantifiers of the TSO logic (Lemma 2).

iCAP-TSO builds on iCAP [16], a recent extension of higher-order separation logic [3] for modular reasoning about concurrent higher-order programs with shared mutable state.

Summary of Contributions. We provide a new program logic, iCAP-TSO, for a TSO memory model, which features:

- a novel logic for reasoning about low-level *racy* code on a TSO memory model, called TSO logic; this logic features new connectives for expressing ordering dependencies introduced by store buffers
- a *standard* separation logic, called SC logic, that allows *simple* reasoning from the perspective of a single thread
- and, importantly, a *fiction of sequential consistency* which allows us to reason about ownership transfer within the SC logic

Moreover, we prove soundness of iCAP-TSO with respect to a model. Using this logic we verify an *efficient* spin-lock implementation against an SC lock specification. Crucially, this means that we can reason about well-synchronized clients *entirely* using standard separation logic proof rules!

Outline. In Section 2 we introduce the programming language that we reason about and its operational semantics. Section 3 illustrates how the fiction of sequential consistency allows us to reason about shared resources using standard separation logic. Section 4 introduces the TSO logic and connectives introduced to reason about store buffers. In Section 5 we illustrate the use of the TSO logic to verify an efficient spin-lock. In Section 6 we discuss the iCAP-TSO soundness theorem. Finally, in Sections 7 and 8 we discuss related work and future work and conclude. Details and proofs can be found in the accompanying technical report [12].

2. Language

We build our logic for a simple, class-based programming language. For simplicity of both semantics and the logic, the language uses let-bindings and expressions, but it stays relatively low-level by ensuring that all the values are machine-word size. The values include variables, natural numbers, booleans, unit, object references (pointers), the null pointer and the special variable **this**. The expressions include values, let bindings, conditionals, constructor and method calls, field reads and writes, atomic compare-and-swap expression and a fork call. The syntax of values and expressions is shown in Figure 1. The class and method definitions are standard and so omitted here; they can be found in the accompanying technical report.

Following the Views framework [7], the operational semantics is split into two components: a thread-local small-step semantics labeled with actions that occur during the step, and action semantics that defines the effect of the action on the machine state — in our case, the heap and the store buffer pool. In the thread-local semantics, a thread, consisting of a thread identifier and expression takes a single step of evaluation to a finite map of threads that contains besides the original thread also the threads spawned by this step. It also emits the action that describes the interaction with the memory state. For instance, the WRITE rule in Figure 3 applies when the expression associated with thread t is an assignment (possibly in some evaluation context). It reduces by replacing the assignment with a unit value, and emits a write action that states that thread t wrote the value v to the field f of object o.

The non-fault memory state consists of a heap — a finite map from pairs of an object reference and a field to semantic values (i.e., all the values that are not variables) — and a store buffer pool, which contains a sequence of buffered updates for each of the finitely many thread identifiers. The memory can also be in a fault state (written $\lightning$), which means that an error in the execution of the program has occurred. The action semantics interprets the actions as functions from memory states to sets of memory states: if it is impossible for the action to occur in the given state, the result is an empty set; if, however, the action may occur in the given state but it would be erroneous, the result is the fault state. Consider the write action emitted by reducing an assignment. In Figure 4 we can see the interpretation of the action: there are three distinct cases. The action is successful if there is a store buffer associated with the thread that emitted the action and the object is allocated in the heap, and has the appropriate field. In this case, the write gets added to the end of the buffer. However, the write action can have two additional outcomes: if there is no store buffer associated with the thread in the store buffer pool, the initial state had to be ill-formed, and so the interpretation of the action is an empty set; however, if the thread is defined, but the reference to the field $o.f$ is not found in the heap, the program is erroneous, and the interpretation of the action is a fault state.

The state of a complete program consists of the thread pool and a memory state, and is consistent if the memory state is a fault, or the domain of the store buffer pool equals the domain of the thread pool. The complete semantics proceeds by reducing one of the threads using the thread-local semantics, then interpreting the resulting action with the action semantics, and reducing to a memory state in the resultant set, as in Figure 2. Note how in some cases, notably read, this might require "guessing" the return value, and checking that the guess was right using the action semantics. Some of the cases of the semantics are written out in Figures 3 and 4. In particular, we show the reduction and action semantics that correspond to the (nondeterministic) flushing of a store buffer: a flush action can be emitted by a thread at any time, and the action is interpreted by flushing the oldest buffered write to the memory. Note also the rules for the compare-and-swap expression: similarly to reading, the return value has to be guessed by the thread local semantics. In the case when the guess does not match the state of the memory, the result of the action semantics is empty set, much like when reading. However, if the compare-and-swap succeeds and the return value matches the outcome, the new value is not just written to the store buffer: instead, the *whole* content of the buffer, including the update resulting from compare-and-swap, is written to main memory. Thus, this expression can serve as a synchronization primitive.

Note that our operational semantics is relatively high-level with respect to the canonical TSO semantics [15]. However, it exhibits all the necessary weak behaviors of the TSO model, and as such can serve as a vehicle for studying the problem.

$$\frac{t \in \operatorname{dom} T \qquad (t, T(t)) \xrightarrow{a} T' \qquad \mu' \in [\![a]\!](\mu)}{(\mu, T) \to (\mu', (T - t) \uplus T')}$$

Figure 2. Single step evaluation of a thread pool.

$$\frac{}{(t, E[\mathsf{o.f} := \mathsf{v}]) \xrightarrow{write(t,o,f,v)} \{(t, E[()])\}}\ \textsc{Write}$$

$$\frac{}{(t, E[\mathsf{o.f}]) \xrightarrow{read(t,o,f,v)} \{(t, E[v])\}}\ \textsc{Read}$$

$$\frac{}{(t, E[\mathsf{CAS}(\mathsf{o.f}, \mathsf{v_o}, \mathsf{v_n})]) \xrightarrow{cas(t,o,f,v_o,v_n,r)} \{(t, E[r])\}}\ \textsc{CAS}$$

$$\frac{}{(t, e) \xrightarrow{flush(t)} \{(t, e)\}}\ \textsc{Flush}$$

Figure 3. Selected cases of the thread-local operational semantics.

3. Reasoning in the SC logic

The SC logic of iCAP-TSO allows us to reason about well-behaved code, *using standard separation logic*, without having to reason about store buffers. Naturally, this class of well-behaved code includes standard mutable data structures without any sharing. We can thus easily verify a list library in the SC logic against the standard separation logic specification as it enforces a *unique* owner. Crucially, this class of well-behaved code also includes the large class of well-synchronized programs *with sharing*. Using the lock specification from the Introduction and the fiction of sequential consistency that it provides, we illustrate how to verify a well-synchronized program with sharing in the SC logic. In particular, we verify a shared bag library implemented using a list protected by a lock.

The SC logic. The SC logic is an intuitionistic higher-order separation logic. Recall that the SC logic features Hoare triples of the form $[\mathsf{P}]\ \mathsf{e}\ [\mathsf{r}.\ \mathsf{Q}]$, where P and Q are SC assertions. Formally, SC assertions are terms of type $\mathsf{Prop}_{\mathsf{SC}}$. SC assertions include the usual connectives and quantifiers of higher-order separation logic and language specific assertions such as points-to, $\mathsf{x.f} \mapsto \mathsf{v}$, for asserting the value of field f of object x.

Recall that SC triples employ a subjective interpretation of the pre- and postcondition: $[\mathsf{P}]\ \mathsf{e}\ [\mathsf{r}.\ \mathsf{Q}]$ expresses that if thread t executes the expression e from an initial state where P holds from the point of view of thread t and e terminates with value v then $\mathsf{Q}[v/\mathsf{r}]$ holds for the terminal state from the point of view of thread t. An assertion P holds from the point of view of a thread t if P's assertions about the heap hold from the point of view of t's store buffer and main memory *and* no other thread's store buffer contains a buffered write to these parts of the heap. The assertion $\mathsf{x.f} \mapsto \mathsf{v}$ thus holds from the point of view of thread t if

- the value of the most recently buffered write to $\mathsf{x.f}$ in t's store buffer is v
- or t's store buffer does not contain any buffered writes to $\mathsf{x.f}$ and the value of $\mathsf{x.f}$ in main memory is v

and no other threads store buffer contains a buffered write to $\mathsf{x.f}$. The condition that no other thread's store buffer can contain a buffered write to $\mathsf{x.f}$ ensures that flushing of store buffers cannot invalidate $\mathsf{x.f} \mapsto \mathsf{v}$ from the point of view of a given thread.

If $\mathsf{x.f} \mapsto \mathsf{v}$ holds from the point of view of thread t and thread t attempts to read $\mathsf{x.f}$ it will thus read the value v either from main memory or its own store buffer. Likewise, if $\mathsf{x.f} \mapsto \mathsf{v_1}$ holds from the point of view of thread t and thread t writes $\mathsf{v_2}$ to $\mathsf{x.f}$, afterwards

$$[\![read(t,o,f,v)]\!](h,U) = \begin{cases} \{(h,U)\} & \text{if } (o,f) \in \operatorname{dom} h \text{ and } \text{lookup}(o.f, U(t), h) = v \\ \emptyset & \text{if } t \notin \operatorname{dom} U \text{ or } (o,f) \in \operatorname{dom} h \text{ and } \text{lookup}(o.f, U(t), h) \neq v \\ \{\lightning\} & \text{if } (o,f) \notin \operatorname{dom} h \end{cases}$$

$$[\![write(t,o,f,v)]\!](h,U) = \begin{cases} \{(h, U[t \mapsto U(t) \cdot (o,f,v)])\} & \text{if } (o,f) \in \operatorname{dom} h \text{ and } t \in \operatorname{dom} U \\ \{\lightning\} & \text{if } (o,f) \notin \operatorname{dom} h \\ \emptyset & \text{if } t \notin \operatorname{dom} U \end{cases}$$

$$[\![cas(t,o,f,v_o,v_n,r)]\!](h,U) = \begin{cases} \{(\text{flush}(h, U(t) \cdot (o,f,v_n)), U[t \mapsto \varepsilon])\} & \text{if } (o,f) \in \operatorname{dom} h,\ r = \textbf{true} \text{ and } \text{lookup}(o.f, U(t), h) = v_o \\ \{(h,U)\} & \text{if } (o,f) \in \operatorname{dom} h,\ r = \textbf{false} \text{ and } \text{lookup}(o.f, U(t), h) \neq v_o \\ \{\lightning\} & \text{if } (o,f) \notin \operatorname{dom} h \\ \emptyset & \text{otherwise} \end{cases}$$

$$[\![flush(t)]\!](h,U) = \begin{cases} \{(h[(o,f) \mapsto v], U[t \mapsto \alpha])\} & \text{if } U(t) = (o,f,v) \cdot \alpha \text{ and } (o,f) \in \operatorname{dom} h \\ \emptyset & \text{if } t \notin \operatorname{dom}(U),\ U(t) = \varepsilon, \text{ or } (o,f) \notin \operatorname{dom} h \end{cases}$$

Figure 4. Selected cases of the action semantics. The lookup function finds the newest value associated with the field, including the store buffer, while the flush function applies all the updates from the store buffer to the heap in order.

$$\frac{}{[\mathsf{x.f} \mapsto \mathsf{v}]\ \mathsf{x.f}\ [\mathsf{r}.\ \mathsf{x.f} \mapsto \mathsf{v} * \mathsf{r} = \mathsf{v}]}\ \text{S-READ} \qquad \frac{}{[\mathsf{x.f} \mapsto \mathsf{v}_1]\ \mathsf{x.f} := \mathsf{v}_2\ [\mathsf{r}.\ \mathsf{x.f} \mapsto \mathsf{v}_2]}\ \text{S-WRITE} \qquad \frac{\triangleright([\mathsf{P}]\ \mathsf{C}(\overline{\mathsf{x}})\ [\mathsf{r}.\ \mathsf{Q}])}{[\mathsf{P}[\overline{\mathsf{v}}/\overline{\mathsf{x}}]]\ \textbf{new}\ \mathsf{C}(\overline{\mathsf{v}})\ [\mathsf{r}.\ \mathsf{Q}[\overline{\mathsf{v}}/\overline{\mathsf{x}}]]}\ \text{S-NEW}$$

$$\frac{\triangleright([\mathsf{P}]\ \mathsf{C.m}(\overline{\mathsf{y}})\ [\mathsf{r}.\ \mathsf{Q}])}{[\mathsf{P}[\overline{\mathsf{v}}/\overline{\mathsf{y}}, \mathsf{x}/\textbf{this}] * \mathsf{x} : \mathsf{C}]\ \mathsf{x.m}(\overline{\mathsf{v}})\ [\mathsf{r}.\ \mathsf{Q}[\overline{\mathsf{v}}/\overline{\mathsf{y}}, \mathsf{x}/\textbf{this}]]}\ \text{S-CALL} \qquad \frac{[\mathsf{P}]\ \mathsf{e}_1\ [\mathsf{r}.\ \mathsf{Q}] \qquad [\mathsf{Q}[\mathsf{x}/\mathsf{r}]]\ \mathsf{e}_2\ [\mathsf{r}.\ \mathsf{R}]}{[\mathsf{P}]\ \textbf{let}\ \mathsf{x} = \mathsf{e}_1\ \textbf{in}\ \mathsf{e}_2\ [\mathsf{r}.\ \mathsf{R}]}\ \text{S-BIND} \qquad \frac{}{[\top]\ \mathsf{v}\ [\mathsf{r}.\ \mathsf{r} = \mathsf{v}]}\ \text{S-VAL}$$

$$\frac{\mathsf{P}_1 \vdash \mathsf{P}_2 \qquad [\mathsf{P}_2]\ \mathsf{e}\ [\mathsf{r}.\ \mathsf{Q}_2] \qquad \mathsf{Q}_2 \vdash \mathsf{Q}_1}{[\mathsf{P}_1]\ \mathsf{e}\ [\mathsf{r}.\ \mathsf{Q}_1]}\ \text{S-CONS} \qquad \frac{[\mathsf{P}]\ \mathsf{e}\ [\mathsf{r}.\ \mathsf{Q}] \qquad \mathsf{stable}(\mathsf{R})}{[\mathsf{P} * \mathsf{R}]\ \mathsf{e}\ [\mathsf{r}.\ \mathsf{Q} * \mathsf{R}]}\ \text{S-FRAME}$$

Figure 5. Selected proof rules for the sequentially consistent logic.

$\mathsf{x.f} \mapsto \mathsf{v}_2$ holds from the point of view of thread t. We thus get the standard rules for reading and writing to a field in our SC logic (rules S-READ and S-WRITE in Figure 5.)

The S-NEW and S-CALL rules are the object allocation and method call rules of the SC logic. The assertion $\mathsf{x} : \mathsf{C}$ expresses that x refers to an object of class C. The method call rule (S-CALL) thus allows us to reason about a call to a method m of a class C. The later operator, written $\triangleright$ in the premise of the S-CALL rule expresses that the given specification must hold *one step later*. Among other things, these later operators internalizes a notion of steps of evaluation. This allows us to reason about mutually recursive methods using Löb induction. The later operator goes back to Gödel-Löb logic and has been used in several models and logics for reasoning about higher-order code, e.g., [2, 11, 17]. Note the side condition that R is stable in the S-FRAME rule: this is a requirement similar to the one we have seen in the specification of the lock in the Introduction, and asserts that the interference of the environment cannot invalidate the frame.

A list library. The standard separation logic specification for a list library enforces a unique owner and thus ensures that the underlying heap representation is not shared. We can express the standard list specification in our SC logic as follows.

$$\begin{aligned} &\exists \mathsf{lst} : \mathsf{Val} \times \mathit{seq}\ \mathsf{Val} \to \mathsf{Prop}_{\mathsf{SC}}. \\ &\quad [\top]\ \mathsf{List}()\ [\mathsf{r}.\ \mathsf{lst}(\mathsf{r}, \varepsilon)] \wedge \\ &\quad \forall \mathsf{l}.\ [\mathsf{lst}(\textbf{this}, \mathsf{l})]\ \mathsf{List.push}(\mathsf{x})\ [\mathsf{lst}(\textbf{this}, \mathsf{x} :: \mathsf{l})] \wedge \\ &\quad \forall \mathsf{l}.\ [\mathsf{lst}(\textbf{this}, \mathsf{l})] \\ &\qquad\quad \mathsf{List.pop}() \\ &\qquad \left[\begin{array}{l} \mathsf{r}.\ (\mathsf{r} = \textbf{null} * \mathsf{l} = \varepsilon * \mathsf{lst}(\textbf{this}, \mathsf{l})) \\ \qquad \vee\ (\mathsf{r} = \mathit{hd}(\mathsf{l}) * \mathsf{lst}(\textbf{this}, \mathit{tail}(\mathsf{l}))) \end{array}\right] \wedge \\ &\quad \forall \mathsf{x} : \mathsf{Val}.\ \forall \mathsf{l}_1, \mathsf{l}_2 \in \mathit{seq}\ \mathsf{Val}.\ \ \mathsf{valid}(\mathsf{lst}(\mathsf{x}, \mathsf{l}_1) * \mathsf{lst}(\mathsf{x}, \mathsf{l}_2) \Rightarrow \bot) \end{aligned}$$

Here Val is the type of values of our programming language and $\mathit{seq}\ \mathsf{Val}$ the type of finite sequences with elements of type Val. The lst predicate is a representation predicate relating a mathematical model of a list with its concrete heap representation. We use higher-order quantification to existentially quantify the lst predicate and thus hide the internal data representation of implementations from clients.

Consider the specification of the list constructor. Due to the subjective interpretation of SC triples the constructor specification expresses that if thread t creates a new list, then there exists a new list from the point of view of thread t. It says nothing about the list from the point of view of other threads (other than that other thread's store buffers cannot contain buffered writes to the list). Likewise, for the other methods: if the list contains the values l from the point of view of thread t and thread t pushes an element x, afterwards the list contains the values $\mathsf{x} :: \mathsf{l}$ from the point of view of t.

This specification also explicitly rules our sharing, by asserting that lst is non-duplicable. Due to this subjective interpretation and

```
Bag {
  Lock lock; List elms;

  pop() {
    let x = this.lock in let y = this.elms in
      x.acquire(); let z = y.pop() in x.release(); z
  }

  ...
}
```

Figure 6. Sketch of a shared bag implementation.

```
 pop() {
[bag(this, P)]
    let x = this.lock in
    let y = this.elms in
      x.acquire();
[this.elms ↦ y ∗ locked(x, R_bag(P)) ∗ R_bag(P)]
      let z = y.pop() in
[locked(x, R_bag(P)) ∗ R_bag(P) ∗ (z = null ∨ P(z))]
        x.release();
[isLock(x, R_bag(P)) ∗ (z = null ∨ P(z))]
        z
[r. r = null ∨ P(r)]
  }
```

Figure 7. Proof outline of the bag Pop method.

the lack of sharing, the verification of a linked-list implementation against this specification is as in separation logic for sequentially consistent memory models. We can instantiate the lst representation predicate with the usual definition and verify the implementation using the usual proof rules.

We can thus reason about mutable data structures without any sharing in the SC logic as if we were in standard separation logic. However, the real power of our proof system comes from the fact that we can still reason as in standard separation logic about sharing of such lists through libraries that provide a fiction of sequential consistency.

A shared bag. To illustrate, consider a shared bag implemented using a list. Each shared bag maintains a list of elements and a lock to ensure exclusive access to the list of elements. Each bag method acquires the lock and calls the corresponding method of the list library before releasing the lock. In Figure 6 we sketch an implementation of a shared bag.

We take the following specification, which allows unrestricted sharing of the bag, to be our specification of a shared bag. This is not the most general specification we can express, but it suffices to illustrate that verification of the shared bag against such specifications is standard. Since the specification allows unrestricted sharing (the bag predicate is duplicable), no client can know the contents of the bag; instead, the specification allows clients to associate ownership of additional resources (expressed using the predicate P) with each element in the bag.

$$
\begin{array}{l}
\exists \mathsf{bag} : \mathsf{Val} \times (\mathsf{Val} \to \mathsf{Prop}_{\mathsf{SC}}) \to \mathsf{Prop}_{\mathsf{SC}}.\ \forall \mathsf{P} : \mathsf{Val} \to \mathsf{Prop}_{\mathsf{SC}}. \\
\quad (\forall \mathsf{x} : \mathsf{Val}.\ \mathsf{stable}(\mathsf{P}(\mathsf{x}))) \ \Rightarrow \\
\qquad [\top]\ \mathsf{Bag}()\ [\mathsf{r}.\ \mathsf{bag}(\mathsf{r}, \mathsf{P})] \wedge \\
\qquad [\mathsf{bag}(\mathbf{this}, \mathsf{P}) * \mathsf{P}(x)]\ \mathsf{Bag.push}(\mathsf{x})\ [\top] \wedge \\
\qquad [\mathsf{bag}(\mathbf{this}, \mathsf{P})]\ \mathsf{Bag.pop}()\ [\mathsf{r}.\ (\mathsf{P}(\mathsf{r}) \vee \mathsf{r} = \mathbf{null})] \wedge \\
\qquad \forall \mathsf{x} : \mathsf{Val}.\ \mathsf{valid}(\mathsf{bag}(\mathsf{x}, \mathsf{P}) \Leftrightarrow \mathsf{bag}(\mathsf{x}, \mathsf{P}) * \mathsf{bag}(\mathsf{x}, \mathsf{P}))
\end{array}
$$

Pushing an element x thus requires the client to transfer ownership of $\mathsf{P}(x)$ to the bag. Likewise, either pop returns **null** or the client receives ownership of the resources associated with the returned element.

To verify the implementation against this specification, we first have to define the abstract bag representation predicate. To define bag we first need to define the resource invariant of the lock. Intuitively, the lock owns the list of elements and the resources associated with the elements currently in the list. This is expressed by the following resource invariant $\mathsf{R}_{bag}(\mathsf{x}, \mathsf{P})$, where x refers to the list of elements.

$$\mathsf{R}_{bag}(\mathsf{x}, \mathsf{P}) \stackrel{\text{def}}{=} \exists \mathsf{l} : seq\ \mathsf{Val}.\ \mathsf{lst}(\mathsf{x}, \mathsf{l}) * \circledast_{\mathsf{y} \in mem(\mathsf{l})} \mathsf{P}(\mathsf{y})$$

The bag predicate asserts *read-only* ownership of the lock and elms fields, and that the lock field refers to a lock with the above resource invariant.

$$\mathsf{bag}(\mathsf{x}, \mathsf{P}) \stackrel{\text{def}}{=} \exists \mathsf{y}, \mathsf{z} : \mathsf{Val}.\ \mathsf{x.lock} \mapsto \mathsf{y} * \mathsf{x.elms} \mapsto \mathsf{z} * \mathsf{isLock}(\mathsf{y}, \mathsf{R}_{bag}(\mathsf{z}, \mathsf{P}))$$

Now, we are ready to verify the bag methods. The most interesting method is Pop, as it actually returns the resources associated with the elements it returns. Figure 7 contains a proof outline of Pop. The crucial thing to note is that from the specification of the lock, once a thread t acquires the lock, it receives ownership of the resource invariant $\mathsf{R}_{bag}(\mathsf{P})$ *from the point of view of* t. This is exactly what we need to use the SC specification of the List.pop method with the S-CALL rule!

In general, we can thus reason about well-synchronized code with sharing in the SC logic as in separation logics for sequentially consistent memory models.

4. TSO logic and connectives

In this section we describe the TSO logic and introduce our new TSO connectives that allow us to reason about the kinds of weak behaviors that occur in low-level concurrency libraries.

We can express the configurations that lead to these weak behaviors by extending the space of states over which the assertions are built — in the case of our TSO model, we include the store buffer pool as an additional component of the memory state. However, reasoning about the buffers directly would be extremely complicated and contrary to the spirit of program logics. Hence, we introduce new logical connectives that help express these new configurations in a useful, abstract manner, and provide appropriate reasoning rules.

The triples and assertions of the TSO logic First, however, we need to consider how the TSO logic is built. As mentioned in the Introduction, its propositions extend the propositions of SC logic by adding the store buffer pool component. Just like SC assertions, this space forms a higher-order intuitionistic separation logic, with the usual rules for reasoning about assertion entailment. However, we are still reasoning about the code running in some thread and we often need to state properties that hold of its — or other thread's — store buffer. Thus, formally, the typing rule for the TSO logic triples is as follows:

$$\frac{\mathsf{P} : \mathsf{TId} \to \mathsf{Prop}_{\mathsf{TSO}} \qquad \mathsf{Q} : \mathsf{TId} \to \mathsf{Val} \to \mathsf{Prop}_{\mathsf{TSO}}}{\{\mathsf{P}\}\ \mathsf{e}\ \{\mathsf{Q}\} : \mathsf{Spec}}$$

where TId is the type of thread identifiers and Spec is the type of specifications. We usually keep this quantification over thread identifiers implicit, by introducing appropriate syntactic sugar for the TSO-level connectives. The logic also include another family

$$\frac{}{\triangleright\overline{\mathsf{x.f} \mapsto \mathsf{v}} \vdash_{\mathrm{w}} \mathsf{x.f} \mapsto \mathsf{v}, \top}\ \text{W-Ax} \qquad \frac{\mathsf{P} \vdash_{\mathrm{w}} \mathsf{x.f} \mapsto \mathsf{v}, \mathsf{R}}{\mathsf{P} * \triangleright\overline{\mathsf{Q}} \vdash_{\mathrm{w}} \mathsf{x.f} \mapsto \mathsf{v}, \mathsf{Q} * \mathsf{R}}\ \text{W-}* \qquad \frac{\mathsf{Q} \vdash_{\mathrm{w}} \mathsf{x.f} \mapsto \mathsf{v}, \mathsf{R}}{\mathsf{P}\ \mathcal{U}\ \mathsf{Q} \vdash_{\mathrm{w}} \mathsf{x.f} \mapsto \mathsf{v}, \mathsf{R}}\ \text{W-}\mathcal{U} \qquad \frac{\mathsf{P} \vdash_{\mathrm{r}} \mathsf{x.f} \mapsto \mathsf{v}}{\langle \mathsf{P} \rangle\ \mathsf{x.f}\ \langle \mathsf{r}.\ \mathsf{P} * \mathsf{r} = \mathsf{v} \rangle^{C}}\ \text{A-Read}$$

$$\frac{\mathsf{P} \vdash_{\mathrm{w}} \mathsf{x.f} \mapsto \mathsf{v}', \mathsf{Q}}{\langle \mathsf{P} \rangle\ \mathsf{x.f} := \mathsf{v}\ \langle \_.\ \mathsf{P}\ \mathcal{U}\ (\ulcorner \mathsf{Q} * \mathsf{x.f} \mapsto \mathsf{v} \urcorner) \rangle^{C}}\ \text{A-Write} \qquad \frac{\mathsf{P} \vdash_{\mathrm{w}} \mathsf{x.f} \mapsto \mathsf{v}, \mathsf{Q}}{\langle \mathsf{P} \rangle\ \mathbf{CAS}(\mathsf{x.f}, \mathsf{v}', \mathsf{v})\ \langle \mathsf{r}.\ \mathsf{r} = \mathbf{true} * \ulcorner \mathsf{Q} * \mathsf{x.f} \mapsto \mathsf{v}' \urcorner \rangle^{C}}\ \text{A-CAS-True}$$

$$\frac{\mathsf{P} \vdash_{\mathrm{r}} \mathsf{x.f} \mapsto \mathsf{v} \qquad \mathsf{v} \neq \mathsf{v}_2}{\langle \mathsf{P} \rangle\ \mathbf{CAS}(\mathsf{x.f}, \mathsf{v}_1, \mathsf{v}_2)\ \langle \mathsf{r}.\ \mathsf{r} = \mathbf{false} * P \rangle^{C}}\ \text{A-CAS-False} \qquad \frac{\langle \mathsf{P} \rangle\ \mathsf{e}\ \langle \mathsf{Q} \rangle^{C} \qquad \mathsf{atomic(e)}}{\{\mathsf{P}\}\ \mathsf{e}\ \{\mathsf{Q}\}}\ \text{A-Start}$$

$$\frac{\{\mathsf{P}\}\ \mathsf{e}_1\ \{\mathsf{r}.\ \mathsf{Q}(\mathsf{r})\} \qquad \{\mathsf{Q}(\mathsf{x})\}\ \mathsf{e}_2\ \{\mathsf{r}.\ \mathsf{R}(\mathsf{r})\}}{\{\mathsf{P}\}\ \mathbf{let}\ \mathsf{x} = \mathsf{e}_1\ \mathbf{in}\ \mathsf{e}_2\ \{\mathsf{r}.\ \mathsf{R}(\mathsf{r})\}}\ \text{Bind} \qquad \frac{\{\mathsf{P}\}\ \mathsf{e}\ \{\mathsf{Q}\} \qquad \mathsf{stable(R)}}{\{\mathsf{P} * \mathsf{R}\}\ \mathsf{e}\ \{\mathsf{Q} * \mathsf{R}\}}\ \text{Frame} \qquad \frac{\{\overline{\mathsf{P}}\}\ \mathsf{e}\ \{\overline{\mathsf{Q}}\}}{[\mathsf{P}]\ \mathsf{e}\ [\mathsf{Q}]}\ \text{S-Shift}$$

Figure 8. Select rules of the TSO logic.

of Hoare triples, the atomic triples, with the following typing rule:

$$\frac{\mathsf{atomic}(e) \qquad \mathsf{P} : \mathsf{TId} \to \mathsf{Prop}_{\mathsf{TSO}} \qquad \mathsf{Q} : \mathsf{TId} \to \mathsf{Val} \to \mathsf{Prop}_{\mathsf{TSO}}}{\langle \mathsf{P} \rangle\ \mathsf{e}\ \langle \mathsf{Q} \rangle^{C} : \mathsf{Spec}}$$

As the rule states, these triples can only be used to reason about atomic expressions — read, write and compare-and-swap. This feature is inherited from iCAP, as a means of reasoning about the way the shared state changes at the atomic updates. We give an example of such reasoning in Section 5.

Two spaces of assertions and the connection between them. As mentioned in the Introduction, our proof system consists of two logics, each with a corresponding space of assertions. To provide the fiction of sequential consistency and show SC specifications for implementations whose correctness involves reasoning about buffered updates, we need to use of both of these spaces in the TSO logic. To this end we use an embedding of $\mathsf{Prop}_{\mathsf{SC}}$ into $\mathsf{Prop}_{\mathsf{TSO}}$.

The canonical embedding is denoted $\ulcorner - \urcorner : \mathsf{Prop}_{\mathsf{SC}} \to \mathsf{Prop}_{\mathsf{TSO}}$. The idea is that $\ulcorner \mathsf{P} \urcorner$ holds in a state that does include store buffers if P holds in the state where we ignore the buffers *and* none of the buffers contain buffered updates to the locations mentioned by P. The intuition behind this embedding is that P should hold *in main memory*. The condition that no buffered updates whatsoever to the state in P are allowed may at first seem overly restrictive. However, it is crucial to our setup, in particular to allow resource transfer and the fiction of sequential consistency. This is due to the nondeterministic nature of the store buffers: if more than one thread contains a buffered update to the same location, no thread can be certain what values of that location it can observe. Thus, we can maintain a reasonable level of complexity only as long as a thread can only write to a location for which no *other* thread has buffered updates. However, *transferring* a resource must mean passing the right to update it to a different thread. Thus, the thread that gives up a resource also can have no buffered writes at the point when the transfer occurs, so no updates of the resource can be buffered at all when the transfer occurs.

For a concrete example of what this embedding means, consider an assertion $\mathsf{x.f} \mapsto \mathsf{v} : \mathsf{Prop}_{\mathsf{SC}}$. Clearly, we can use our embedding to get $\ulcorner \mathsf{x.f} \mapsto \mathsf{v} \urcorner$ — an assertion that means precisely that the reference x.f is defined, its associated value in the heap is v, and there are no buffered updates in any of the store buffers to the field x.f.

Reasoning about buffered updates. Even though there must be no buffered updates of the resource when we transfer resources between threads, most of the time we reason about situations where buffered updates *are* possible. Therefore we define two additional connectives that allow us to reason about the presence of buffered updates. The first of these is a binary function $\ulcorner - \ \mathbf{in}\ - \urcorner : \mathsf{Prop}_{\mathsf{SC}} \times \mathsf{TId} \to \mathsf{Prop}_{\mathsf{TSO}}$. Intuitively, $\ulcorner \mathsf{P}\ \mathbf{in}\ t \urcorner$ means that P holds from the perspective of thread t — including the possible buffered updates in the store buffer of t. However, for the reasons discussed prior, we have to disallow buffered updates to the locations described by P in any of the other store buffers.

The second of this pair of connectives is the one that allows us to express how the state *changes* due to an update. Because of its role, it has a certain temporal feel: in fact, it behaves in a way that is somewhat similar to the classic "until" operator. Intuitively, $\mathsf{P}\ \mathcal{U}_t\ \mathsf{Q}$ means that there exists a buffered update in the store buffer of thread t, such that until this update is flushed the assertion P holds, while after the update gets written to memory, the assertion Q holds. Thus, it can be used to describe the ordering dependencies introduced by the presence of store buffers. This intuition should become clearer by observing the proof rules in Figure 8 (explained in the following).

Coming back to our example, consider first the assertion $\ulcorner \mathsf{x.f} \mapsto \mathsf{v}\ \ \mathbf{in}\ \ t \urcorner$. This proposition holds in any state where all the buffered updates to x.f are in the store buffer associated with thread t — including the boundary case when there are no buffered updates to x.f at all. Thus, if we are reasoning from the perspective of thread t, we know precisely the value we can read from the field x.f, but no other thread has any knowledge about the state, save that x.f is defined. On the other hand, in the state described by $\ulcorner \mathsf{x.f} \mapsto 1 \urcorner\ \mathcal{U}_t\ \ulcorner \mathsf{x.f} \mapsto 2 \urcorner$, we have information that any thread can use: we know that the value of x.f in the heap is 1, and that there exists a buffered update in thread t, such that before that update there is no updates to x.f, i.e., it is the first update to x.f in the store buffer of t, and after it gets flushed $\ulcorner \mathsf{x.f} \mapsto 2 \urcorner$ holds — so the update must set x.f to 2. Additionally, we know that there are no other buffered updates to x.f. This means that the thread t can observe the value of x.f to be 2, while all of the other threads can observe it to be 1. Note that, since $\mathcal{U}$ is a binary operator on $\mathsf{Prop}_{\mathsf{TSO}}$, it is possible to use it to express multiple buffered updates.

Since most of the time we are reasoning from the perspective of a particular thread, we also include some syntactic sugar: $\mathcal{U}$ is a shorthand for an update in the current thread, while $\overline{\mathcal{U}}$ is a shorthand for an update in some thread *other than the current one*. We also use $\overline{p}$ as a shorthand for $\ulcorner p\ \mathbf{in}\ t \urcorner$, where t is the current thread.

Reading and writing state. The presence of additional connectives that mention the state makes reading fields of an object and writing to them more involved than in standard separation logic. We deal with this by introducing additional judgments that specify when we can read or write a value. Intuitively, $\mathsf{P} \vdash_{\mathrm{r}} \mathsf{x.f} \mapsto \mathsf{v}$ specifies that, from the perspective of the current thread x.f has the value v — and so the read expression can proceed. Similarly, $\mathsf{P} \vdash_{\mathrm{w}} \mathsf{x.f} \mapsto \mathsf{v}, \mathsf{Q}$ means that in the state specified by P the current thread has permission to write to x.f, the value of x.f from its perspective is v, and that once an update to x.f reaches main memory,

Q will additionally hold *in main memory*. In this sense, Q behaves like a frame; however, since the rule is used to reason about writes, which affect the store buffers, we can learn something about the order of writes in the buffer, which we would not be able to do using the frame rule. Note that the reading of the write judgment corresponds to the intuitive understanding of the postcondition of the A-WRITE rule in Figure 8: the write we are performing introduces a buffered update, before which the precondition still holds, and after which $\ulcorner \mathsf{Q} * \mathsf{x.f} \mapsto \mathsf{v} \urcorner$ holds. This setup allows us to retain some knowledge of the order in which updates reach the main memory in an abstract manner; a property that is crucial to allow resource transfer in the presence of the store buffers. Note also that the A-CAS-TRUE rule can be viewed as a combination of writing and flushing. As mentioned above, a write adds a buffered write to the appropriate store buffer, which is expressed using $\mathcal{U}$ in A-WRITE. However, if at that moment we *flush* the store buffer, we get a condition under which the right-hand side of $\mathcal{U}$ is defined to hold: thus, we get to know that $\ulcorner \mathsf{Q} * \mathsf{x.f} \mapsto \mathsf{v}' \urcorner$ holds in the postcondition of A-CAS-TRUE.

Also of interest are some of the proof rules for the write judgment (the read judgment behaves in a similar manner, with some complications when reading from $\overline{\mathcal{U}}$). Note how in rules W-AX and W-* the later operators ($\triangleright$) appear. These arise from the fact that the model is defined using guarded recursion to break circularities and, since they play the role of guards, they can be removed at the atomic steps of the proof, as expressed by the rules. Moreover, the rules also match the intuition we gave about the store buffer related connectives. First, the judgment means that it is enough that we know the value of $\mathsf{x.f}$ from the thread's perspective: hence, we are interested in the most recent updates, and look to the right of $\mathcal{U}$ in W-$\mathcal{U}$. For the same reason, it is enough to use $\overline{-}$ in W-AX. Second, due to the way we interpret the "frame" in the write judgment, we only need Q to hold from the thread's perspective in W-*. This is also why we can use the frame taken from the right-hand-side of $\mathcal{U}$ in W-$\mathcal{U}$: the update that we are reasoning about in the write judgment will happen *after* the buffered update asserted in the definition of $\mathcal{U}$, so the frame taken from after the first update will hold in the memory by the time the later one reaches it.

Stability and stabilization. There is one potentially worrying issue with the explanation of the $\mathcal{U}_t$ operator given in this section: since at any point in the program a *flush* action can occur nondeterministically, how can we know that there still exists a buffered update as asserted by $\mathcal{U}_t$? After all, it might have been flushed to the memory. This is the question of *stability*[1] of the until operator — and the answer is that it is unstable by design. The rationale behind this choice is simple: Suppose we had made it stable by allowing the possibility that the buffered update has already been flushed. Then, if we were to read a field that had a buffered write to it in a different thread, we would not know whether the write was still buffered or had been flushed, and so we would not know what value we read. With the current definition, when we read, we know that the update is still buffered and so the result of the read is known. However, we only allow reasoning with unstable assertions in the *atomic* triples, i.e., when reasoning about a single read, write or compare-and-swap expression. Hence, we need a way to make $\mathcal{U}$ stable. For this reason, we define an explicit *stabilization* operator, $(\!|-|\!)$. It is a closure operator, which means we have $\mathsf{P} \vdash (\!|P|\!)$. Moreover, for stable assertions, the other direction $(\!|\mathsf{P}|\!) \vdash \mathsf{P}$ also holds. The important part, however, is how stabilization behaves with respect to $\mathcal{U}$: provided P and Q are stable, we have $(\!|\mathsf{P}\ \mathcal{U}_t\ \mathsf{Q}|\!) \dashv\vdash (\mathsf{P}\ \mathcal{U}_t\ \mathsf{Q}) \vee \mathsf{Q}$. This does indeed correspond with our intuition — even for stable assertions P and Q, the interference can flush the buffered update that is asserted in the definition of $\mathcal{U}$, which would transition to a state in which Q holds. However, since P and Q are stable, this is also the *only* problem that the interference could cause. Explicit stabilization is not a new idea: it has been used recently in the context of program logics, most commonly in connection with rely-guarantee reasoning. In particular, Wickerson studies explicit stabilization in RGSep in his PhD thesis [19, Chapter 3], and Ridge [14] uses it to reason about the x86-TSO.

Considering stability of the other TSO-level connectives exposes a different issue: the fact that there are two distinct parts to the interference a thread is subject to — the effect of store buffer actions, and the effect other threads can have on shared state. We follow the CAP approach [8] of reasoning about shared state using shared regions with protocols. See Section 5 for an example. We thus have two notions of stability: $\mathsf{r{-}stable}$ and $\mathsf{b{-}stable}$. The first of these considers the shared regions, and is the only kind of stability that concerns the SC-logic; it corresponds to stability in iCAP [16]. The second only considers the additional guarantees required by the TSO logic. Stability is defined as the conjunction of the two. Thanks to the restrictions of the contents of the store buffers imposed by the embeddings, for any $\mathsf{P} : \mathsf{Prop}_{\mathsf{SC}}$ we have $\mathsf{b{-}stable}(\ulcorner \mathsf{P} \urcorner)$ and $\mathsf{b{-}stable}(\ulcorner \mathsf{P}\ \mathbf{in}\ t \urcorner)$. These two facts in conjunction with the explicit stabilization operator suffice for reasoning about low level implementations. Although important for reasoning about shared regions, stability under region interference is much as in iCAP, so we elide the proof rules for inferring that a predicate is $\mathsf{r{-}stable}$ and refer the reader to the technical report for details [12].

Interpretation of the SC logic. Note that the same intuition that lies behind the SC logic, discussed in the previous section, is expressed by the $\ulcorner - \ \mathbf{in}\ t \urcorner$ embedding. This is more formally expressed by the rule S-SHIFT in Figure 8 (recall $\overline{\mathsf{P}}$ is syntactic sugar for $\lambda t.\ \ulcorner \mathsf{P}\ \mathbf{in}\ t \urcorner$), which states that the two ways of expressing that a triple holds from the perspective of the current thread are equivalent. In fact, we take this rule as the *definition* of the SC triples, and so we can prove that the SC triples actually form a standard separation logic by proving that the proof rules of SC logic correspond to admissible rules in the TSO logic. This is expressed by the following theorem:

Theorem 1 (Soundness of SC logic)**.** *The SC logic is sound wrt. its interpretation within TSO logic, i.e., the proof rules (in particular those shown in Figure 5) composed with the rule* S-SHIFT *are admissible rules of the TSO logic.*

For most of the proof rules, the soundness follows directly; the only ones that require additional properties to be proved are the frame, consequence, and standard quantifier rules, which additionally require the following property:

Lemma 2. *The embeddings $\ulcorner - \urcorner$ and $\ulcorner - \ \mathbf{in}\ t \urcorner$ distribute over quantifiers and separating conjunction, and preserve entailment.*

The formal statement of this property, along with the proof, can be found in the accompanying technical report.

5. Reasoning in the TSO logic

In Section 3 we illustrated that the fiction of sequential consistency provided by the lock specification allows us to reason about shared mutable data structures shared through locks, without explicitly reasoning about the underlying weak memory model. Of course, to verify a lock implementation against this lock specification, we *do* have to reason about the weak memory model. In this section we illustrate how to achieve this using our TSO logic. We focus on the use of the TSO-connectives introduced in Section 4 to describe

[1] Recall an assertion is stable, if it cannot be invalidated by the environment.

```
Lock {
  bool locked;

  Lock() {
    CAS(this.locked, false, null)
  }

  acquire() {
    let x = CAS(this.locked, true, false) in
      if x then () else acquire()
  }

  release() {
    this.locked := false
  }
}
```

Figure 9. Spin-lock implementation.

the machine states of the spin-lock and elide the details related to the use of concurrent abstract predicates.

The spin-lock implementation that we wish to verify is given in Figure 9. It uses a compare-and-swap (CAS) instruction to attempt to acquire the lock, but only a primitive write instruction to release the lock. While CAS flushes the store buffer of the thread that executes the CAS, a primitive write does not. To verify this implementation, we thus have to explicitly reason about the possibility of buffered releases in store buffers.

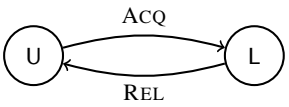
Specification. In the Introduction we introduced a lock specification expressed in our SC logic. When verifying the spin-lock implementation, we actually verify the implementation against the following slightly stronger specification, from which we can easily derive the SC specification.

$$\begin{aligned}
&\exists \mathsf{isLock}, \mathsf{locked} : \mathsf{Prop}_{\mathsf{sc}} \times \mathsf{Val} \to \mathsf{Prop}_{\mathsf{sc}}.\\
&\quad \forall \mathsf{R} : \mathsf{Prop}_{\mathsf{sc}}.\ \mathsf{stable}(\mathsf{R}) \Rightarrow\\
&\qquad \{\overline{\mathsf{R}}\}\ \mathsf{Lock}()\ \{\mathsf{r}.\ \overline{\mathsf{isLock}(\mathsf{R},\mathsf{r})}\}\\
&\qquad \wedge\ \{\overline{\mathsf{isLock}(\mathsf{R},\mathsf{x})}\}\ \mathsf{Lock.acquire}()\ \{\overline{\mathsf{locked}(\mathsf{R},\mathsf{x})} * \ulcorner \mathsf{R} \urcorner\}\\
&\qquad \wedge\ \{\overline{\mathsf{locked}(\mathsf{R},\mathsf{x}) * \mathsf{R}}\}\ \mathsf{Lock.release}()\ \{\top\}\\
&\qquad \wedge\ \mathsf{valid}(\forall \mathsf{x} : \mathsf{Val}.\ \mathsf{isLock}(\mathsf{R},\mathsf{x}) \Leftrightarrow \mathsf{isLock}(\mathsf{R},\mathsf{x}) * \mathsf{isLock}(\mathsf{R},\mathsf{x}))\\
&\qquad \wedge\ \forall \mathsf{x} : \mathsf{Val}.\ \mathsf{stable}(\mathsf{isLock}(\mathsf{R},\mathsf{x})) \wedge \mathsf{stable}(\mathsf{locked}(\mathsf{R},\mathsf{x}))
\end{aligned}$$

Note that this stronger specification is expressed using TSO triples. This specification of the Acquire method is slightly stronger: this specification asserts that upon termination of Acquire, the resource invariant R holds in main memory and there are no buffered writes affecting R in *any* store buffer ($\ulcorner \mathsf{R} \urcorner$). The weaker SC specification only asserts that the resource invariant R holds from the point of view of the acquiring thread and that there are no buffered writes affecting R in *any of the other threads'* store buffers ($\overline{\mathsf{R}}$).

Lock protocol. To verify the spin-lock implementation against the above specification, we first need to define the abstract representation predicates isLock and locked. Following CAP [8] and iCAP [16], to reason about sharing iCAP-TSO extends separation logic with shared regions, with protocols governing the resources owned by each shared region. In the case of the spin-lock, upon allocation of a new spin-lock the idea is to allocate a new shared region governing the state of the spin-lock and ownership of the resource invariant.

Conceptually, a spin-lock can be in one of two states: locked and unlocked. In iCAP-TSO, we express this formally using the following labeled transition system.



The above labeled transition system specifies an abstract model of the lock; to relate it to the concrete implementation, for each abstract state (L and U), we choose an assertion that describes the resources the spin-lock owns in the given abstract state.

Since acquiring the lock flushes the store buffer of the acquiring thread, the locked state is fairly simple. In the locked state the spin-lock owns the locked field, which contains the value true in main memory and there are no buffered writes to locked in any store buffer. The spin-lock x with resource invariant R thus owns the resources described by $I_{\mathsf{L}}(\mathsf{x},\mathsf{R},\mathsf{n})$ in the abstract locked state.

$$I_{\mathsf{L}}(\mathsf{x},\mathsf{R},\mathsf{n}) = \ulcorner \mathsf{x.locked} \mapsto \mathsf{true} \urcorner$$

Due to the possibility of buffered releases in store buffers, the unlocked state is more complicated. In the unlocked state,

- either locked is false in main memory and there are no buffered writes to locked in any store buffer
- or locked is true in main memory, and there is *exactly one* store buffer with a buffered write to locked, and the value of this buffered write is false

Furthermore, in case there is a buffered write to locked that changes its value from true to false, then, once the buffered write reaches main memory, the resource invariant holds in main memory. Intuitively, the resource invariant holds from the point of view of the releasing thread before the lock is released; hence any buffered writes affecting the resource invariant must reach main memory before the buffered release. We can express this ordering dependency using the until operator as follows.

$$\begin{aligned}
I_{\mathsf{U}}(\mathsf{x},\mathsf{R},\mathsf{n}) = \exists \mathsf{t} : \mathsf{TId}.\ (\!|\ &\ulcorner \mathsf{x.locked} \mapsto \mathsf{true} \urcorner\ \mathcal{U}_{\mathsf{t}}\\
&\ulcorner \mathsf{x.locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]_1^{\mathsf{n}} \urcorner\ |\!|)
\end{aligned}$$

Here $[\textsc{Rel}]_1^{\mathsf{n}}$ is a CAP action permission used to ensure that only the current holder of the lock can release the lock. Since this is orthogonal to the underlying memory model, we refer the interested reader to the technical report [12] for details.

Since both arguments of $\mathcal{U}_t$ are stable, as explained in Section 4, $I_{\mathsf{U}}(\mathsf{x},\mathsf{R},\mathsf{n})$ is equivalent to the following assertion.

$$\begin{aligned}
&\exists \mathsf{t} : \mathsf{TId}.\\
&\quad \ulcorner \mathsf{x.locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]_1^{\mathsf{n}} \urcorner\ \vee\\
&\quad (\ulcorner \mathsf{x.locked} \mapsto \mathsf{true} \urcorner\ \mathcal{U}_t\ \ulcorner \mathsf{x.locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]_1^{\mathsf{n}} \urcorner)
\end{aligned}$$

The first disjunct corresponds to the case where the release has already made its way to main memory and the second disjunct to the case where it is still buffered.

The definition of isLock in terms of I_{L} and I_{U} now follows standard iCAP.[2] The isLock predicate asserts the existence of a shared region governed by the above labeled transition system, where the resources owned by the shared region in the two abstract states are given by I_{L} and I_{U}. It further asserts that the abstract state of the shared region is either locked or unlocked and also a non-exclusive right to acquire the lock.

Proof outline. To verify the spin-lock implementation, it remains to verify each method against the specification instantiated with the concrete isLock and locked predicates. To illustrate the reasoning related to the weak memory model, we focus on the verification of the acquire method and the compare-and-swap instruction in par-

[2] See the accompanying technical report for a formal definition of isLock.

ticular. The full proof outline is given the accompanying technical report.

As the name suggests, the resources owned by a shared region are shared between all threads. Atomic instructions are allowed to access and modify resources owned by shared regions, provided they follow the protocol imposed by the region. In the case of the spin-lock, the spin-lock region owns the shared locked field and we thus need to follow the spin-lock protocol to access and modify the locked field. Since the precondition of acquire asserts that the lock is either in the locked or unlocked state, we need to consider two cases.

If the spin-lock region is already locked, then the compare-and-swap fails and we remain in the locked state. This results in the following proof obligation:

$$\begin{array}{l}\langle \triangleright I_{\mathsf{L}}(\mathbf{this}, \mathsf{R}, \mathsf{n})\rangle \\ \quad \mathsf{CAS}(\mathbf{this}.\mathsf{locked}, \mathbf{true}, \mathbf{false}) \\ \langle \mathsf{r}.\ \triangleright I_{\mathsf{L}}(\mathbf{this}, \mathsf{R}, \mathsf{n}) * \mathsf{r} = \mathsf{false}\rangle\end{array}$$

That is, if locked contains the value true from the point of view of a thread t, then CAS'ing from false to true in thread t will fail. This is easily shown to hold by rule A-CAS-FALSE.

If the spin-lock region is unlocked, then the compare-and-swap may or may not succeed, depending on whether the buffered release has made it to main memory and which thread performed the buffered release. If it succeeds, the acquiring thread transitions the shared region to the locked state and takes ownership of the resource invariant; otherwise, the shared region remains in the unlocked state. This results in the following proof obligation:

$$\begin{array}{l}\langle \triangleright I_{\mathsf{U}}(\mathbf{this}, \mathsf{R}, \mathsf{n})\rangle \\ \quad \mathsf{CAS}(\mathbf{this}.\mathsf{locked}, \mathbf{true}, \mathbf{false}) \\ \langle \mathsf{r}.\ \exists \mathsf{y} \in \{\mathsf{U}, \mathsf{L}\}.\ \triangleright I_{\mathsf{y}}(\mathbf{this}, \mathsf{R}, \mathsf{n}) * Q(\mathsf{y}, \mathsf{r}, \mathsf{n})\rangle\end{array}$$

where

$$\begin{aligned} Q(\mathsf{y}, \mathsf{r}, \mathsf{n}) = (\mathsf{y} = U * \mathsf{r} = \mathsf{false}) \ \vee \\ (\mathsf{y} = L * [\textsc{Rel}]_1^{\mathsf{n}} * \ulcorner \mathsf{R} \urcorner * \mathsf{r} = \mathsf{true}) \end{aligned}$$

Rewriting the explicit stabilization to a disjunction and commuting in $\triangleright$, this reduces to the following proof obligation:

$$\begin{array}{l}\langle \exists \mathsf{t} : \mathsf{TId}.\ \triangleright \ulcorner \mathsf{x}.\mathsf{locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]_1^{\mathsf{n}} \urcorner \vee \\ \quad (\triangleright \ulcorner \mathsf{x}.\mathsf{locked} \mapsto \mathsf{true} \urcorner \, \mathcal{U}_{\mathsf{t}} \, \triangleright \ulcorner \mathsf{x}.\mathsf{locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]_1^{\mathsf{n}} \urcorner)\rangle \\ \quad \mathsf{CAS}(\mathbf{this}.\mathsf{locked}, \mathbf{true}, \mathbf{false}) \\ \langle \mathsf{r}.\ \exists \mathsf{y} \in \{\mathsf{U}, \mathsf{L}\}.\ \triangleright I_{\mathsf{y}}(\mathbf{this}, \mathsf{R}, \mathsf{n}) * Q(\mathsf{y}, \mathsf{r}, \mathsf{n})\rangle\end{array}$$

In case the second disjunct holds and there exists buffered releases in store buffer t, the CAS will succeed if executed by thread t and fail if executed by any other thread. To prove this obligation, we thus do case analysis on whether t is our thread or not. This leaves us with three proof obligations (after strengthening the post-condition):

- either the buffered release is in our store buffer

$$\begin{array}{l}\langle \triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{true} \urcorner \, \mathcal{U} \, \triangleright \ulcorner \mathsf{x}.\mathsf{locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]_1^{\mathsf{n}} \urcorner\rangle \\ \quad \mathsf{CAS}(\mathbf{this}.\mathsf{locked}, \mathbf{true}, \mathbf{false}) \\ \langle \mathsf{r}.\ \triangleright I_{\mathsf{L}}(\mathbf{this}, \mathsf{R}, \mathsf{n}) * [\textsc{Rel}]_1^{\mathsf{n}} * \ulcorner \mathsf{R} \urcorner * \mathsf{r} = \mathsf{true}\rangle\end{array}$$

- or in some other thread's store buffer

$$\begin{array}{l}\langle \triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{true} \urcorner \, \overline{\mathcal{U}} \, \triangleright \ulcorner \mathsf{x}.\mathsf{locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]_1^{\mathsf{n}} \urcorner\rangle \\ \quad \mathsf{CAS}(\mathbf{this}.\mathsf{locked}, \mathbf{true}, \mathbf{false}) \\ \langle \mathsf{r}.\ \triangleright I_{\mathsf{U}}(\mathbf{this}, \mathsf{R}, \mathsf{n}) * \mathsf{r} = \mathsf{false}\rangle\end{array}$$

- or it has already been flushed

$$\begin{array}{l}\langle \triangleright \ulcorner \mathsf{x}.\mathsf{locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]_1^{\mathsf{n}} \urcorner\rangle \\ \quad \mathsf{CAS}(\mathbf{this}.\mathsf{locked}, \mathbf{true}, \mathbf{false}) \\ \langle \mathsf{r}.\ \triangleright I_{\mathsf{L}}(\mathbf{this}, \mathsf{R}, \mathsf{n}) * [\textsc{Rel}]_1^{\mathsf{n}} * \ulcorner \mathsf{R} \urcorner * \mathsf{r} = \mathsf{true}\rangle\end{array}$$

These three proof obligations are easily discharged using rules A-CAS-TRUE and A-CAS-FALSE.

Note that our TSO logic forces us to consider exactly those four cases that intuitively one has to consider when considering a TSO weak memory model.

6. Soundness

We prove soundness of iCAP-TSO with respect to a kind of Kripke model in which worlds consist of allocated regions and their associated protocols. Since iCAP-TSO inherits iCAP's impredicative protocols [16], worlds need to be recursively defined. Hence we use a meta-theory that supports the definition of sets by guarded recursion, namely the so-called internal language of the topos of trees [4]. As the overall structure of the model construction remains unchanged from iCAP, we elide the details. We refer the interested reader to the accompanying technical report [12] for details and proofs.

Following the Views framework [7], TSO assertions (terms of type $\mathsf{Prop}_{\mathsf{TSO}}$) are modeled as predicates over *instrumented states*. In addition to the underlying machine state, instrumented states contain shared regions and protocols. Soundness is proven by relating the machine semantics with an instrumented semantics that, for instance, enforces that clients obey the chosen protocols when accessing shared state. This relation is expressed through an erasure function, $\lfloor - \rfloor$, that erases an instrumented state to a set of machine states.

The soundness theorem is stated in terms of the following $eval(\mu, T, q)$ predicate, which asserts that for any terminating execution of the thread pool T from initial state μ, the predicate q must hold for the terminal state and thread pool. The $eval$ predicate is defined as a guarded recursive predicate (the recursive occurrence of $eval$ is guarded by $\triangleright$), to express that each step of evaluation in the machine semantics corresponds to a step in the topos of trees.

$$\begin{aligned} eval(\mu, T, q) \stackrel{\text{def}}{=} (\mathrm{irr}(\mu, T) \wedge (\mu, T) \in q) \ \vee \\ (\forall T', \mu'.\ (\mu, T) \to (\mu', T') \Rightarrow \triangleright eval(\mu', T', q)) \end{aligned}$$

Here $\mathrm{irr}(\mu, T)$ means that (μ, T) is irreducible. We can now state the soundness of iCAP-TSO.

Theorem 3 (Soundness). *If* $\{\mathsf{P}\}e\{\mathsf{r}.\mathsf{Q}\}$ *and* $\mu \in \lfloor [\![\mathsf{P}]\!](t) \rfloor$ *then*

$$eval(\mu, [t \mapsto e], \lambda(\mu', T).\ \mu' \in \lfloor [\![\mathsf{Q}]\!](t)(T(t)) \rfloor)$$

This theorem expresses that if a specification $\{\mathsf{P}\}e\{\mathsf{r}.\ \mathsf{Q}\}$ holds and the execution of the thread pool $[t \mapsto e]$ with a single thread t from an initial state μ in the erasure of P terminates (including threads spawned by t), then the terminal state is in the erasure of Q instantiated with the return value $T(t)$ of thread t.

7. Related Work

Our work builds directly on iCAP [16], which is an extension of separation logic for modular reasoning about concurrent higher-order programs with shared mutable state. Our work extends the model of iCAP with store buffers to implement a TSO memory model, extends the iCAP logic with TSO-connectives for reasoning about these store buffers and, importantly, defines a new logic for reasoning about sequentially consistent clients.

There has been a lot of previous work on reasoning about TSO-like and even weaker memory models. Owens [13] defines a property on the set of sequentially consistent traces of an x86 program, which ensures that every TSO trace of the program has a "memory equivalent" sequentially consistent trace. Owens shows how this property allows clients of synchronization primitives to reason as if the underlying memory model was sequentially consistent, despite

racy implementations of these synchronization primitives. In particular, he proves that all traces of spin-lock well-synchronized x86 programs satisfy this property. Unfortunately, Owens' approach is non-compositional: while Owens proves similar results for multiple synchronization primitives *in isolation*, these results do not apply to clients that *combine* two or more of these synchronization primitives. Our approach is compositional in this sense and we can reason about clients that combine multiple libraries with racy implementations that have been verified *independently*, without having to re-verify the library implementations.

Ridge [14] proposes a Rely-Guarantee-based proof system for reasoning about assembly code on the x86-TSO memory model. To ensure sound reasoning in the presence of a TSO memory model, the proof system enforces that the rely relation includes possible interference from write buffers. Ridge uses this proof system to verify an implementation of Simpson's four slot algorithm. Ridge's proof system does not provide anything like the fiction of sequential consistency provided by our logic or Owens' approach; in particular, to reason about clients of Simpson's four slot algorithm in Ridge's system, we would still have to explicitly reason about possible interference from store buffers.

In his thesis, Wehrman [18] proposes a program logic for reasoning about low-level code in a language with built-in locks and a TSO memory model. Wehrman's logic is based on separation logic, which he also extends with temporal operators to reason about the orderings introduced by the weak memory model. Wehrman's temporal operators differ from ours; in particular, Wehrman's temporal operators concern all store buffers, whereas our temporal operator refers to a particular store buffer. Wehrman's objective is to allow local reasoning about racy *implementations* and his logic does not provide any fiction of sequential consistency to simplify reasoning about *clients*. In his conclusion, Wehrman points out that it would be beneficial with a program logic for a weak memory model that reduces to a standard logic for non-racy clients. This is exactly what our SC logic and fiction of sequential consistency provides!

Another verification technique for programs in the TSO memory model, which does not involve explicit reasoning about store buffers, is to restrict programs to follow a certain programming discipline that ensures that all program behaviors can be simulated by a sequentially consistent machine. Cohen and Schirmer [6] propose such a programming discipline based on ownership. With this approach one can, of course, only reason about programs that follow the discipline. For instance, as Cohen and Schirmer remark, an efficient spin-lock implementation with a buffered release, like the one we verify in Section 5, does not obey their programming discipline. Essentially, the problem is one of compositionality: The programming discipline is formalized as a property of the set of reachable states of a *complete* program. This is a non-compositional property and the authors do not provide a compositional method for reasoning about it.

Gotsman et al. [10] propose another approach for providing clients with a fiction of sequential consistency based on linearizability. By relating racy library implementations on a TSO architecture with abstract specifications on an SC architecture, they can reason about data-race free clients that call racy libraries using an SC memory model. Their approach is only compositional for *non-interacting* libraries: assuming two libraries are linearizable with respect to SC specifications and the set of locations accessed by the libraries are disjoint, then the composition of the two libraries is linearizable with respect to the composition of the specifications. Their approach also requires the client and any libraries to be non-interacting. Our work does not suffer from either of these restrictions. Furthermore, these restrictions make it very unclear how to extend the their approach to higher-order languages, as higher-order languages blur the lines between library and client. In contrast, our work extends easily to a higher-order programming language, by following the iCAP methodology [16].

8. Conclusion and Future Work

We have presented a new proof system, iCAP-TSO, to support modular and scalable reasoning for a language with a TSO memory model. The proof system consists of two logics, a TSO logic for reasoning about *racy* low-level code and an SC logic for reasoning about well-behaved code. This class of well-behaved code includes standard mutable data structures without any sharing but also well-synchronized code *with sharing*. For this class of well-behaved code the SC logic allows us to reason much as in standard separation logic, without having to worry about the underlying weak memory model.

In particular, we use the TSO logic to verify an efficient spin-lock implementation against an SC specification. This allows us to reason about well-synchronized clients with sharing entirely from within the SC logic! We illustrate this by verifying a standard list library and using it to verify a shared bag library. Both of these libraries can be verified entirely in the SC logic and their proofs are as in separation logics for sequentially consistent memory models.

We think of iCAP-TSO as a first steps towards more automated/interactive tools for reasoning about the TSO memory model. In this paper we have focused on the foundational issues of constructing a logic that allows simple reasoning for well-behaved code. As future work it would be interesting to try to extend tools like [5, 9] to support mostly automated verification in the SC logic, while requiring more interactive verification of low-level racy code in the TSO logic.

Acknowledgements

We thank Mark Batty, Aleš Bizjak, Susmit Sarkar, and Peter Sewell for helpful discussions on this work. This research was supported in part by the ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU).

References

[1] Linux kernel mailing list, Nov. 1999. spin_unlock optimization(i386).

[2] A. W. Appel, P.-A. Melliès, C. D. Richards, and J. Vouillon. A Very Modal Model of a Modern, Major, General Type System. In *Proceedings of POPL*, 2007.

[3] B. Biering, L. Birkedal, and N. Torp-Smith. BI-Hyperdoctrines, Higher-order Separation Logic, and Abstraction. *ACM TOPLAS*, 2007.

[4] L. Birkedal, R. Møgelberg, J. Schwinghammer, and K. Støvring. First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees. In *Proceedings of LICS*, 2011.

[5] A. Chlipala. Mostly-automated Verification of Low-level Programs in Computational Separation Logic. In *Proceedings of PLDI*, 2011.

[6] E. Cohen and B. Schirmer. From Total Store Order to Sequential Consistency: A Practical Reduction Theorem. In *Proceedings of ITP*, 2010.

[7] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: Compositional Reasoning for Concurrent Programs. In *Proceedings of POPL*, 2013.

[8] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent Abstract Predicates. In *Proceedings of ECOOP*, 2010.

[9] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular Shape Analysis. In *Proceedings of PLDI*, 2007.

[10] A. Gotsman, M. Musuvathi, and H. Yang. Show No Weakness: Sequentially Consistent Specifications of TSO Libraries. In *Proceedings of DISC*, 2012.

[11] A. Hobor, R. Dockins, and A. Appel. A theory of indirection via approximation. In *Proceedings of POPL*, 2010.

[12] NN. A Separation Logic for Fictional Sequential Consistency. Technical report, 2013.

[13] S. Owens. Reasoning about the Implementation of Concurrency Abstractions on x86-TSO. In *Proceedings of ECOOP*, 2010.

[14] T. Ridge. A Rely-Guarantee proof system for x86-TSO. In *Proceedings of VSTTE*, 2010.

[15] P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. O. Myreen. x86-TSO: A Rigorous and Usable Programmers Model for x86 Multiprocessors. In *Communications of the ACM*, 2010.

[16] K. Svendsen and L. Birkedal. Impredicative Concurrent Abstract Predicates. 2013. Under Submission.

[17] A. Turon, D. Dreyer, and L. Birkedal. Unifying Refinement and Hoare-Style Reasoning in a Logic for Higher-Order Concurrency. In *Proceedings of ICFP*, 2013.

[18] I. Wehrman. *Weak-Memory Local Reasoning*. PhD thesis, University of Texas, 2012. Dissertation draft.

[19] J. Wickerson. *Concurrent verification for sequential programs*. PhD thesis, University of Cambridge, 2012.

A Concurrent Logical Relation

Lars Birkedal, Filip Sieczkowski, and Jacob Thamsborg

IT University of Copenhagen, Denmark
{birkedal, fisi, thamsborg}@itu.dk

Abstract

We present a logical relation for showing the correctness of program transformations based on a new type-and-effect system for a concurrent extension of an ML-like language with higher-order functions, higher-order store and dynamic memory allocation.

We show how to use our model to verify a number of interesting program transformations that rely on effect annotations. In particular, we prove a Parallelization Theorem, which expresses when it is sound to run two expressions in parallel instead of sequentially. The conditions are expressed solely in terms of the types and effects of the expressions. To the best of our knowledge, this is the first such result for a concurrent higher-order language with higher-order store and dynamic memory allocation.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases logics of programs, verification, semantics

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Relational reasoning about program equivalence is useful for reasoning about the correctness of program transformations, data abstraction (representation independence), compiler correctness, etc. The standard notion of program equivalence is contextual equivalence and in recent years, there have been many improvements in reasoning methods for higher-order ML-like languages with general references, based on bisimulations, e.g., [1, 2, 3], traces [4], game semantics [5], and Kripke logical relations, e.g., [6, 7, 8, 9].

In this paper we present the first Kripke logical relation for reasoning about equivalence of a *concurrent* higher-order ML-like language with higher-order store and dynamic memory allocation.

To state and prove useful equivalences about concurrent programs, it is necessary to have some way of restricting the contexts under which one proves equivalences. This point was made convincingly in the recent paper by Liang et. al. [10], who presented a rely-guarantee-based simulation for verifying concurrent program transformations for a first-order imperative language (with first-order store). Here is a very simple example illustrating the point. Consider two expressions

$$e_1 \equiv x := 1; y := 1 \quad \text{and} \quad e_2 \equiv y := 1; x := 1.$$

Here x and y are variables of type refint. The expressions e_1 and e_2 are not contextually equivalent. (To see why, consider expression $e_3 \equiv x := 0;\ y := 0$, and note that running e_1 in parallel with e_3 may result in a state with $!x = 0$ and $!y = 1$, but that cannot be the case when we run e_2 in parallel with e_3.) The issue is, of course, that the context may also modify the references x and y. On the other hand, if we know that no other threads have access to x or y, then it should be the case that e_1 and e_2 are equivalent. We can express this restriction on the contexts using a refined region-based type-and-effect system.

Conference title on which this volume is based on.
Editors: Billy Editor, Bill Editors; pp. 61–91

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We first recall that a type-and-effect system is a type system that classifies programs according to which side effects the programs may have. A variety of effect systems have been proposed for higher-order programming languages, e.g., [11, 12, 13], see [14] for a recent overview. Effect systems can often be understood as specifying the results of a static analysis, in the sense that it is possible to automatically infer types and effects. Effect systems can be used for different purposes: they were originally proposed by Lucassen and Gifford [12] for parallelization purposes but they have also, e.g., been used as the basis for implementing ML using a stack of regions for memory management [13, 15]. In a recent series of papers, Benton et. al. have argued that another important point of effect systems is that they can be used as the basis for effect-based program transformations, e.g., compiler optimizations, [16, 17, 18, 19], see also [20]. The idea is that certain program transformations are only sound under additional assumptions about which effects program phrases may, or rather may not, have.

Now, returning to our example, we refine the types of x and y to be $\mathsf{ref}_\rho\mathsf{int}$ and $\mathsf{ref}_\sigma\mathsf{int}$, respectively. Intuitively, this expresses that x and y are references in different regions, but it does not put any restrictions on whether other threads may access x or y. Thus, when we type e_1 and e_2 we will use two contexts of region variables, one for public regions that can be used by other concurrently running threads, and one for private regions that are under the control of the present thread. This idea is inspired by recent work on concurrent separation logic, e.g., [21, 22, 23, 24]. We use a vertical bar to separate public and private regions: the typing context

$$\rho, \sigma \mid \emptyset \mid x : \mathsf{ref}_\rho\mathsf{int}, y : \mathsf{ref}_\sigma\mathsf{int}$$

expresses that ρ and σ are public regions, whereas the typing context

$$\emptyset \mid \rho, \sigma \mid x : \mathsf{ref}_\rho\mathsf{int}, y : \mathsf{ref}_\sigma\mathsf{int}$$

expresses that ρ and σ are private regions. The expressions e_1 and e_2 are well-typed in the latter context and, with this refined typing, they are indeed contextually equivalent, because our type-and-effect system guarantees that no well-typed context can access regions ρ or σ. (The expressions are also well-typed in the former context, but not contextually equivalent with that refined typing.)

In this paper we present a step-indexed Kripke logical relations model of a type-and-effect system with public and private regions for a concurrent higher-order language with general references. Our model is constructed over the operational semantics of the programming language, and builds on recent work by Thamsborg and Birkedal on logical relations for the sequential sub-language [20]. Note that the type-and-effect annotations are just annotations; the operational semantics of the language is standard and regions only exist in our semantic model, not in the operational semantics.

As an important application of our model we prove a Parallelization Theorem, which expresses when it is sound to run two expressions in parallel instead of sequentially. To the best of our knowledge, this is the first such result for a higher-order language with higher-order store and dynamic memory allocation. Here is a very simple instance of the theorem. Consider two expressions

$$e_1 \equiv y :=\! !\, x + !\, y \quad \text{and} \quad e_2 \equiv z :=\! !\, x + !\, z,$$

each well-typed in a context

$$\emptyset \mid \rho_x, \rho_y, \rho_z \mid x : \mathsf{ref}_{\rho_x}\mathsf{int}, y : \mathsf{ref}_{\rho_y}\mathsf{int}, z : \mathsf{ref}_{\rho_z}\mathsf{int},$$

i.e., where x, y, and z are references in distinct private regions. In this context, running e_1 and e_2 sequentially is contextually equivalent to running e_1 and e_2 in parallel. Intuitively, this also makes sense: e_1 and e_2 update references in distinct regions, and it is unproblematic that they both read (but not write) from the same region.

As mentioned, this was a simple instance of the Parallelization Theorem. We stress that the theorem is expressed solely in terms of the type and efffects of the expressions e_1 and e_2, so a compiler may automatically infer that it is safe to parallelize two expressions by looking at the inferred effect types, and without reasoning about all interleavings. Moreover, the theorem applies to contexts and expressions with general higher types (not just with references to integers and unit types). Note that the distinction between private and public regions is also crucial here (parallelization would not be sound if the effects of the expressions were on public regions).

Our type-and-effect system crucially also includes a region-masking rule. Traditionally, this rule has been used to hide local effects on regions, which makes it possible to view a computation as pure even if it uses effects locally and makes the effect system stronger, in the sense that it can justify more program transformations. Here we also observe that the masking rule can be used for introducing private regions, since the masking rule intuitively guarantees that effects on a region are not leaked to the context. It is well-known that region-masking makes the model construction for a sequential language technically challenging, see the extensive discussion in [20]. Here it is yet more challenging because of concurrency; we explain how our model ensures soundness of the masking rule in Section 3.

The extension with concurrency also means that when we define the logical relation for contextual approximation and relate two computations e_1 and e_2, we cannot simply require relatedness after e_1 has completed evaluation (as in the sequential case), since other threads should be allowed to execute as well. We explain our approach to relating concurrent computations in Section 3; it is informed by recent soundness proofs of unary models of concurrent separation logic [25, 26].

Another challenge arises from the fact that since our language includes dynamically allocated general references, the existence of the logical relation is non-trivial; in particular, the set of Kripke worlds must be recursively defined. Here we build on our earlier work [27] and define the worlds as a solution to a recursive metric-space equation. Indeed, to focus on the essential new aspects due to the extension with concurrency, we deliberately choose to use the exact same notion of worlds as we used for the sequential sub-language in [20]. In the same vein, we here consider a monomorphically typed higher-order programming language with general references, but leave out universal and existential types as well as recursive types. However, we want to stress that since our semantic techniques (step-indexed Kripke logical relations over recursively defined worlds) do indeed scale well to universal, existential, and recursive types, e.g. [27, 9], it is possible to extend our model to a language with such types. We conjecture that it is also possible to extend our model to richer effect systems involving region and effect polymorphism, but we have not done so yet.

2 Language and Typing

We consider a standard call-by-value lambda calculus with general references, and extended with parallel composition and an atomic construct. We assume countably infinite, pairwise disjoint sets of *region variables* $\mathcal{RV}$ (ranged over by ρ), *locations* $\mathcal{L}$ (ranged over by l) and program variables (ranged over by x, y, f). As usual, the reduction relation is between configurations, $(e \,|\, h) \longmapsto (e' \,|\, h')$ where heaps $\mathcal{H}$ are finite maps from locations to

$$
\begin{array}{rcl}
\pi & ::= & rd_\rho \mid wr_\rho \mid al_\rho \\
\varepsilon & ::= & \pi_1, \ldots, \pi_n \\
\tau & ::= & 1 \mid \mathsf{int} \mid \tau_1 \times \tau_2 \mid \mathsf{ref}_\rho\, \tau \\
 & & \mid \tau_1 \to_\varepsilon^{\Pi,\Lambda} \tau_2 \\
v & ::= & x \mid \langle\rangle \mid \langle v_1, v_2 \rangle \\
 & & \mid \mathsf{fun}\, f(x).e \mid l \\
e & ::= & v \mid \mathsf{proj}_i\, v \mid v\, e \mid \mathsf{ref}\, v \mid !\, v \\
 & & \mid v_1 := v_2 \mid \mathsf{par}\ e_1\ \mathsf{and}\ e_2 \\
 & & \mid \mathsf{cas}\ (v_1, v_2, v_3) \mid \mathsf{atomic}\ e \\
E & ::= & [\,] \mid v\, E \mid \mathsf{par}\ E\ \mathsf{and}\ e_2 \\
 & & \mid \mathsf{par}\ e_1\ \mathsf{and}\ E
\end{array}
$$

Figure 1 Syntax

$$
\begin{array}{rcl}
(E[\mathsf{proj}_i\, \langle v_1, v_2 \rangle] \mid h) & \longmapsto & (E[v_i] \mid h) \\
(E[(\mathsf{fun}\, f(x).e)\, v] \mid h) & \longmapsto & (E[e[\mathsf{fun}\, f(x).e/f, v/x]] \mid h) \\
(E[\mathsf{ref}\, v] \mid h) & \longmapsto & (E[l] \mid h[l \mapsto v]) \text{ if } l \notin dom(h) \\
(E[l := v] \mid h) & \longmapsto & (E[\langle\rangle] \mid h[l{:=}v]) \text{ if } l \in dom(h) \\
(E[!\, l] \mid h) & \longmapsto & (E[h(l)] \mid h) \quad \text{if } l \in dom(h) \\
(E[\mathsf{par}\ v_1\ \mathsf{and}\ v_2] \mid h) & \longmapsto & (E[\langle v_1, v_2 \rangle] \mid h) \\
(E[\mathsf{cas}\ (l, n_1, n_2)] \mid h) & \longmapsto & (E[1] \mid h[l{:=}n_2]) \\
 & & \text{if } l \in dom(h) \text{ and } h(l) = n_1 \\
(E[\mathsf{cas}\ (l, n_1, n_2)] \mid h) & \longmapsto & (E[0] \mid h) \\
 & & \text{if } l \in dom(h) \text{ and } h(l) \neq n_1 \\
(E[\mathsf{atomic}\ e] \mid h) & \longmapsto & (E[v] \mid h') \\
 & & \text{if } (e \mid h) \longmapsto^* (v \mid h') \\
(E[\mathsf{atomic}\ e] \mid h) & \longmapsto & (E[\mathsf{atomic}\ e] \mid h)
\end{array}
$$

Figure 2 Operational semantics

values. Figures 1 and 2 give the syntax and operational semantics; we denote the set of expressions $\mathcal{E}$ and the set of values $\mathcal{V}$. The evaluation contexts allow parallel evaluation inside par expressions, and there is a new primitive reduction covering the case when the two subcomputations have terminated. For technical simplicity, we allow an atomic e expression to reduce to itself, possibly introducing more divergence than the diverging behaviours of e. The syntax is kept minimal; in examples we may use additional syntactic sugar, e.g., writing let $x = e_1$ in e_2 for $(\mathsf{fun}\, f(x).e_2)\, e_1$ for some fresh f. For $e \in \mathcal{E}$, we write $\mathrm{FV}(e)$ and $\mathrm{FRV}(e)$ for the sets of free program variables and region variables, respectively; also we define $\mathrm{rds}\, \varepsilon = \{\rho \in \mathcal{RV} \mid rd_\rho \in \varepsilon\}$ and similarly for writes and allocation.

The form of the judgments of our type-and-effect system is standard with one important refinement: regions are partitioned into *public* and *private* regions, with the purpose of restricting interference from the environment. In greater detail, a typing judgement looks like this:

$$\Pi \mid \Lambda \mid \Gamma \vdash e : \tau, \varepsilon.$$

The Γ, e and τ are the usual: the *variable context* Γ assigns types to program variables in the expression e, with the resulting type of τ. To get an idea of — or rather an upper bound of — the side-effects of e, we split the heap into *regions*; these are listed in Π and Λ. We track memory accesses by adding a set ε of *effects* of the form rd_ρ, wr_ρ and al_ρ, where ρ is a region. Roughly, a computation with effect rd_ρ may read one or more locations in region ρ, and similarly for writes and allocation. This setup goes back to Lucassen and Gifford [12].

The novelty, as mentioned in the Introduction, is our partition of regions into the *public* ones Π and the *private* ones Λ. As opposed to the rest of the judgment, this public-private division does not make promises about the behavior of e. Instead, it states the expectations that e has of the environment: threads running in parallel with e may — in a well-typed manner — read, write and allocate in the public regions but must leave the private regions untouched.

When running parallel threads, the private regions of the parent are shared between the children, and so are public from their point of view; this is reflected in the typing

$$\frac{}{\Pi \,|\, \Lambda \,|\, \Gamma, x : \tau \vdash x : \tau,\ \emptyset} \qquad \frac{}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash \langle\rangle : 1,\ \emptyset} \qquad \frac{\Pi \,|\, \Lambda \,|\, \Gamma \vdash v : \tau_1 \times \tau_2,\ \varepsilon}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash \mathsf{proj}_i\, v : \tau_i,\ \varepsilon}$$

$$\frac{\Pi \,|\, \Lambda \,|\, \Gamma \vdash v_1 : \tau_1,\ \varepsilon_1 \qquad \Pi \,|\, \Lambda \,|\, \Gamma \vdash v_2 : \tau_2,\ \varepsilon_2}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash \langle v_1, v_2\rangle : \tau_1 \times \tau_2,\ \varepsilon_1 \cup \varepsilon_2} \qquad \frac{\Pi \,|\, \Lambda \,|\, \Gamma, f : \tau_1 \to^{\Pi,\Lambda}_{\varepsilon} \tau_2, x : \tau_1 \vdash e : \tau_2,\ \varepsilon}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash \mathsf{fun}\, f(x).e : \tau_1 \to^{\Pi,\Lambda}_{\varepsilon} \tau_2,\ \emptyset}$$

$$\frac{\Pi \,|\, \Lambda \,|\, \Gamma \vdash v : \tau_1 \to^{\Pi,\Lambda}_{\varepsilon} \tau_2,\ \varepsilon_1 \qquad \Pi \,|\, \Lambda \,|\, \Gamma \vdash e : \tau_1,\ \varepsilon_2}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash v\, e : \tau_2,\ \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon} \qquad \frac{\Pi \,|\, \Lambda \,|\, \Gamma \vdash v : \tau,\ \varepsilon \qquad \rho \in \Pi, \Lambda}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash \mathsf{ref}\, v : \mathsf{ref}_\rho \tau,\ \varepsilon \cup \{al_\rho\}}$$

$$\frac{\Pi \,|\, \Lambda \,|\, \Gamma \vdash v_1 : \mathsf{ref}_\rho \tau,\ \varepsilon_1 \qquad \Pi \,|\, \Lambda \,|\, \Gamma \vdash v_2 : \tau,\ \varepsilon_2}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash v_1 := v_2 : 1,\ \varepsilon_1 \cup \varepsilon_2 \cup \{wr_\rho\}} \qquad \frac{\Pi \,|\, \Lambda \,|\, \Gamma \vdash v : \mathsf{ref}_\rho \tau,\ \varepsilon}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash\, !\, v : \tau,\ \varepsilon \cup \{rd_\rho\}}$$

$$\frac{\Pi \,|\, \Lambda, \rho \,|\, \Gamma \vdash e : \tau,\ \varepsilon}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash e : \tau,\ \varepsilon - \rho}\ (\rho \notin \mathrm{FRV}(\Gamma, \tau)) \qquad \frac{\cdot \,|\, \Pi, \Lambda \,|\, \Gamma \vdash e : \tau,\ \varepsilon}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash \mathsf{atomic}\ e : \tau,\ \varepsilon}\ (\mathrm{als}\,\varepsilon \subseteq \mathrm{rds}\,\varepsilon \cap \mathrm{wrs}\,\varepsilon)$$

$$\frac{\Pi, \Lambda \,|\, \cdot \,|\, \Gamma \vdash e_1 : \tau_1,\ \varepsilon_1 \qquad \Pi, \Lambda \,|\, \cdot \,|\, \Gamma \vdash e_2 : \tau_2,\ \varepsilon_2}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash \mathsf{par}\ e_1\ \mathsf{and}\ e_2 : \tau_1 \times \tau_2,\ \varepsilon_1 \cup \varepsilon_2}$$

$$\frac{\Pi \,|\, \Lambda \,|\, \Gamma \vdash v_1 : \mathsf{ref}_\rho \mathsf{int},\ \varepsilon_1 \qquad \Pi \,|\, \Lambda \,|\, \Gamma \vdash v_2 : \mathsf{int},\ \varepsilon_2 \qquad \Pi \,|\, \Lambda \,|\, \Gamma \vdash v_3 : \mathsf{int},\ \varepsilon_3}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash \mathsf{cas}\ (v_1, v_2, v_3) : \mathsf{int},\ \{wr_\rho, rd_\rho\} \cup \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3}$$

$$\frac{\Pi \,|\, \Lambda \,|\, \Gamma \vdash e : \tau_1,\ \varepsilon_1 \qquad \Pi, \Lambda \vdash \tau_1 \leq \tau_2 \qquad \varepsilon_1 \subseteq \varepsilon_2}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash e : \tau_2,\ \varepsilon_2}\ (\mathrm{FRV}(\varepsilon_2) \subseteq \Pi, \Lambda)$$

$$\frac{}{\Theta \vdash \tau \leq \tau}\ (\mathrm{FRV}(\tau) \subseteq \Theta) \qquad \frac{\Theta \vdash \tau_1 \leq \tau_1' \qquad \Theta \vdash \tau_2 \leq \tau_2'}{\Theta \vdash \tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'}$$

$$\frac{\Theta \vdash \tau_1' \leq \tau_1 \qquad \Theta \vdash \tau_2 \leq \tau_2' \qquad \varepsilon_1 \subseteq \varepsilon_2 \qquad \Pi_1 \subseteq \Pi_2 \qquad \Lambda_1 \subseteq \Lambda_2}{\Theta \vdash \tau_1 \to^{\Pi_1,\Lambda_1}_{\varepsilon_1} \tau_2 \leq \tau_1' \to^{\Pi_2,\Lambda_2}_{\varepsilon_2} \tau_2'}\ (\mathrm{FRV}(\varepsilon_2), \Pi_2, \Lambda_2 \subseteq \Theta)$$

Figure 3 Typing and subtyping relations. Notice that for a typing judgement $\Pi \,|\, \Lambda \,|\, \Gamma \vdash e : \tau,\ \varepsilon$ we always have $\mathrm{FRV}(\Gamma, \tau, \varepsilon) \subseteq \Pi \cup \Gamma$.

rule for parallel composition, c.f. Figure 3. Note that the parent thread only continues once both children have terminated; as a consequence, the parent regains ownership of its private regions before it goes on. Running an expression atomically temporarily makes all regions private. The side condition is a technical necessity. Finally, new, private regions are introduced by the so-called masking rule:

$$\frac{\Pi \,|\, \Lambda, \rho \,|\, \Gamma \vdash e : \tau,\ \varepsilon}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash e : \tau,\ \varepsilon - \rho}\ (\rho \notin \mathrm{FRV}(\Gamma, \tau))$$

The subtraction of ρ in the conclusion removes any read, write or allocation effects tagged with ρ. The reading of the masking rule is that we make a brand new, empty region ρ for e to use, but once e has terminated we forget about ρ again; this works out since the side condition prevents e from leaking locations from ρ. Traditionally, the masking rule has been used to do memory-management [13] as well as a means of hiding local effects to facilitate effect-based program transformations [17, 20]. Here we make another use of the rule: we observe that, moreover, e cannot leak locations from ρ while running and so ρ is a *private* region for the duration of e. After all, the only means of inter-thread communication is shared memory. Note that from the perspective of the context, this rule allows to *remove* a private region, and prepare a setup for application of the parallel composition.

All the typing rules are in Figure 3. Note how reference types are tagged with the region

where the location resides and that function arrows are tagged with the latent effects as well as with the public and private regions that the function expects; the latter is natural once we remember that a function is basically just a suspended, well-typed expression.

Because of the nondeterminism arising from par and shared references, the definition of contextual equivalence could take into account both may- and must-convergence. In this paper we only consider may-equivalence and formally we define (may-) contextual approximation by:

▶ **Definition 1.** $\Pi \,|\, \Lambda \,|\, \Gamma \vdash e \lesssim_{\downarrow} e' : \tau, \varepsilon$ if and only if for all h and C typed such that $C : (\Pi \,|\, \Lambda \,|\, \Gamma \vdash \tau, \varepsilon) \rightsquigarrow (\cdot \,|\, \cdot \,|\, \cdot \vdash \mathsf{int}, \emptyset)$, whenever $(C[e] \,|\, h) \downarrow$ then $(C[e'] \,|\, h) \downarrow$.

Here, as usual, $(e \,|\, h) \downarrow$ means that $(e \,|\, h) \longmapsto^{*} (v \,|\, h')$ for some value v and some h'. Typing for contexts is defined in a standard way; selected rules are presented in Appendix A.1.

Contextual equivalence, $\Pi \,|\, \Lambda \,|\, \Gamma \vdash e \approx e' : \tau, \varepsilon$, is then defined as $\Pi \,|\, \Lambda \,|\, \Gamma \vdash e \lesssim_{\downarrow} e' : \tau, \varepsilon$ and $\Pi \,|\, \Lambda \,|\, \Gamma \vdash e' \lesssim_{\downarrow} e : \tau, \varepsilon$. Note that the diverging behaviours introduced by our operational semantics of atomic e do not influence may-contextual equivalence.

3 Definition of the logical relation

Semantic Types and Worlds We give a Kripke or world-indexed logical relation. This is a fairly standard approach to modeling dynamic allocation; in combination with higher-order store, however, it comes with a fairly standard problem: the *type-world circularity*. Roughly, semantic types are indexed over worlds and worlds contain semantic types, so both need to be defined before the other. A specific instance of this circularity was solved recently by Thamsborg and Birkedal [20] based on metric-space theory developed by Birkedal et. al. [27]; we re-use that solution here. Semantic types (and worlds) are constructed as a fixed-point of a endo-functor on a certain category of metric-spaces. We do not care about that, though; we just give the result of the construction. In addition, we largely ignore the fact that we actually deal in metric spaces and not just plain sets; the little metric machinery we need is deferred to Appendix A.2.

There is a set $\mathbf{T}$ of *semantic types* and a set $\mathbf{W}$ of *worlds*; types are world-indexed relations on values and worlds describe the regions and type-layouts of heaps, roughly speaking. Take a type $\mu \in \mathbf{T}$ and apply it to a world $w \in \mathbf{W}$ and you get an indexed relation on values, i.e., $\mu(w) \subseteq \mathbb{N} \times \mathcal{V} \times \mathcal{V}$. These relations are downwards closed in the first coordinate; we read $(k, v_1, v_2) \in \mu(w)$ as saying that v_1 and v_2 are related at type μ up to approximation k assuming world w.

We assume a countably infinite set of region names $\mathcal{RN}$; a world $w \in \mathbf{W}$ contains finitely many such $|w| \subseteq_{\mathit{fin}} \mathcal{RN}$. Some of these $\mathrm{dom}(w) \subseteq |w|$ are *live* and the rest are *dead*. To each live region $r \in \mathrm{dom}(w)$ we associate a finite partial bijection $w(r)$ on locations decorated with types, i.e., $w(r) \subseteq_{\mathit{fin}} \mathcal{L} \times \mathcal{L} \times \widehat{\mathbf{T}}$ such that for $(l_1, l_2, \mu), (m_1, m_2, \nu) \in w(r)$ we have that both $l_1 = m_1$ and $l_2 = m_2$ imply $l_1 = m_1$, $l_2 = m_2$ and $\mu = \nu$. We write $\mathrm{dom}_1(w(r))$ for the set of left hand side locations in the bijection and $\mathrm{dom}_2(w(r))$ for the right hand side ones; different regions must have disjoint left and right hand side locations. For convenience, we set $\mathrm{dom}_1^{\mathrm{A}}(w) = \bigcup_{r \in A \cap \mathrm{dom}(w)} \mathrm{dom}_1(w(r))$ whenever $A \subseteq |w|$, and we write $\mathrm{dom}_1(w)$ for $\mathrm{dom}_1^{|w|}(w)$, i.e., the set of all left hand side locations. Similarly for the right hand side.

Worlds evolve and types adapt. Triples of two locations and a type can be added to a live region, as long as different regions remain disjoint. Orthogonal to this, one can add a fresh, i.e., neither live nor dead, region name with an associated empty partial bijection. And one can kill any live region, rendering it dead and losing the associated the partial bijection in

the process. The reflexive, transitive closure of all three combined is a preorder $\sqsubseteq$ on worlds; it is a crucial property of types that they respect this, i.e., that $w \sqsubseteq w' \implies \mu(w) \subseteq \mu(w')$ for any two $w, w' \in \mathbf{W}$ and any $\mu \in \mathbf{T}$. This is *type monotonicity* and it prevents values from fleeing types over time.

Finally, to tie the knot, there is an isomorphism $\iota : \widehat{\mathbf{T}} \to \mathbf{T}$ from the odd types stored in worlds to proper types. Whenever a type is extracted from a world it needs to be coerced by this isomorphism before it can be applied to some world.

The Logical Relation and Interpretation of Types Often, a logical relation goes like this: two computations are related if they (from related heaps) reduce to related values (and heaps); this is the *extensional* view: we do not care about the intermediate states. As we consider concurrency, however, a computation can be interrupted and so we need to start caring. In our setup, public regions are accessible from the environment. To address this, we assume that before each reduction step, the public regions hold related values; in return, we promise related values after the step. In other words, the *granularity of extensionality* is just one step for the public regions. For private regions, however, there is no interference and the granularity is an entire computation as usual. This is the fundamental idea; it is how we propose to stay extensional in the face of concurrency.

Without further ado, let us look into the cornerstone of our model: the safety relation defined in Figure 6; auxiliary relations are defined in Figure 8. What does it mean to have

$$(k, h_1^\circ, h_2^\circ, e_1, e_2, h_1, h_2) \in \mathbf{safe}_{\tau,\varepsilon}^{\Pi,\Lambda,A,R}\, w^\circ, w?$$

Overall, it says that after environment interference, we can match the behavior of e_1, i.e., termination or any one-step reduction, by zero or more steps of e_2; match in the sense of (re-)establishing certain relations, including safety itself. Safety is a *local* property of a pair of computations, this is crucial: it has no knowledge of computations running concurrently and h_1 and h_2 are the *local heaps*, i.e., the parts of the global heaps that the e_1 respectively e_2 control exclusively. The computations consider $R(\Pi)$ to be their public, $R(\Lambda)$ to be their private and A to be their *anonymous* regions. The latter intuitively are private regions that have been masked out: they exist only for the duration of these computations, but we have to track them to deny the environment access; this is another difficulty imposed by concurrency. Safety is indexed by a world w as well; note that worlds are *global* things: all concurrent threads share one world, i.e., they agree about the division of the heap into regions and the types associated to locations. Finally k is intuitively the number of steps we are safe for, h_1° and h_2° are the (private parts of) the initial local heaps, τ is the expected return type, ε the effects and w° the initial world.

We unroll the definition in writing. The first pair of big square brackets — the *prerequisites* — translates to 'the environment interferes'. This yields a new world w' subject to the constraints of the environment transition relation: no public, private or anonymous regions are killed, and the latter two see no allocation either. The actual contents of the public regions are unknown, but we are free to assume that they hold related values of the proper type, at least where we have read effects; this is the *public heaps* g_1 and g_2 in the precondition relation. In addition we have *frames* f_1 and f_2 that cover the remainder of the world and a triple-split relation that ensures coherence between the domains of corresponding parts of the world and the heaps, see Figures 4 and 8.

The left hand side is irreducible in the *termination branch* and takes one step in the *progress branch*. In either case, we must match this in zero or more steps on the right hand side, not touching the frame; this means finding a future world w'' and relating a number of things. The choice of future world is restricted by the self transition relation: we must not

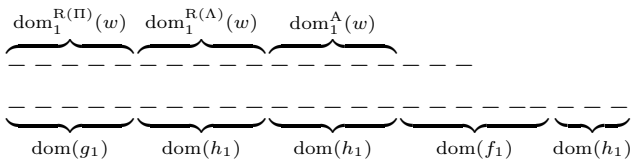


Figure 4 The left hand side of the triple-split relation. The top dashed line is $\mathrm{dom}_1(w)$, the bottom dashed line $\mathrm{dom}(g_1 \cdot h_1 \cdot f_1)$. The local heap h_1 has a *private* part matching the private regions, an *anonymous* part matching the anonymous regions and an *off-world* part outside the domain of the world. The frame f_1 must cover regions that are neither public, private nor anonymous.

kill private or public regions, but we can allocate in them, and regions that we know nothing about must be left untouched; this is our promise to the environment. In the termination branch, we are furthermore required to kill off all anonymous regions as the computation is done; any new regions added in the progress branch go to the set of anonymous regions. In both branches, the changes made to the public heap must be well-typed and permitted by the effects and, if we are done, we check the changes made to (the private part of) the local heaps as well; the fact that the public heaps are compared across a single stage and the (private parts of) the local heaps are compared across the entire computations is the crux of the idea of having different granularities of extensionality.

In addition to performing actual allocation, we have the possibility of moving existing locations from, say, the off-world part of the local heap into the public heap or the private part of the local heap; this is a subtle point that permits the actual allocation of new locations and the corresponding extension of the world to be temporarily out of sync.

We have glossed over one aspect of safety: the right hand side takes steps in the ordinary operational semantics, but the left hand side works in the *instrumented operational semantics*. A reduction $(e \mid h) \to_\mu^n (e' \mid h')$ in the latter implies a similar reduction in former; in addition it counts the steps of a reduction with all atomic commands 'unfolded' (with unfolding itself counting one step) and it records all heap accesses; the formal definition is deferred to the Appendix. We need the former for compatibility of the atomic typing rule below: atomic commands really unfold as they execute, hence we must count the number of 'unfolded' steps. It is less immediate that we must test the actual reads, writes and allocations, recorded by μ, against the effects described by ε, as done in the progress branch of safety. But if omitted, our present proof of the Parallelization Theorem falls short, since it relies on the following simple, but crucial commutation property:

▶ **Lemma 2.** *If we have* $l \notin \mu$ *and* $(e \mid h) \to_\mu^n (e' \mid h')$*, then* $(e \mid h[l \mapsto v]) \to_\mu^n (e' \mid h'[l \mapsto v])$.

The actual logical relation is given in Figure 7. The existentially quantified $a \in \mathbb{N}$ is the minimal number of anonymous regions required to run; apart from that it uses safety in a straightforward way. There is some asymmetry to these definitions: the anonymous regions A are required to exist (and be empty) in the world beforehand, but are killed off in the termination branch; also the precondition on the (private parts of) the initial local heaps is in the logical relation whereas the postcondition lives in the termination branch. The interpretation of types is in Figure 5. Interpreting the function type looks daunting, but a function is just a suspended expression with a single free variable, hence we have to restate most of the logical relation in the definition. Apart from that, we just remark that the $R(\rho) \notin \mathrm{dom}(w)$ case of reference interpretation is part of an approach to handling dangling pointers (due to region masking) proposed recently in [20]; similarly for the $R(\mathrm{FRV}(\varepsilon)) \not\subseteq \mathrm{dom}(w)$.

$$[\![1]\!]^R w = \{(k,(),()) \mid k \in \mathbb{N}\} \qquad [\![\mathsf{int}]\!]^R w = \{(k,n,n) \mid k \in \mathbb{N} \wedge n \in \mathbb{Z}\}$$

$$[\![\tau_1 \times \tau_2]\!]^R w = \left\{(k,(v_{11},v_{21}),(v_{12},v_{22})) \mid (k,v_{11},v_{12}) \in [\![\tau_1]\!]^R w \wedge (k,v_{21},v_{22}) \in [\![\tau_2]\!]^R w\right\}$$

$$[\![\mathsf{ref}_\rho \tau]\!]^R w = \begin{cases} \left\{ \begin{array}{c} (k,l_1,l_2) \mid \exists \mu \in \widehat{\mathbf{T}}.\, (l_1,l_2,\mu) \in w(R(\rho)) \wedge \\ \forall w' \sqsupseteq w.\, [\![\tau]\!]^R w' \overset{k}{=} (\iota\, \mu)(w') \end{array} \right\} & R(\rho) \in \mathrm{dom}(w) \\ \{(k,v_1,v_2) \mid k \in \mathbb{N} \wedge v_1, v_2 \in \mathcal{V}\} & R(\rho) \notin \mathrm{dom}(w) \end{cases}$$

$$[\![\tau_1 \to^{\Pi,\Lambda}_\varepsilon \tau_2]\!]^R w =$$

$$\begin{cases} \left\{ \begin{array}{l} (k, \mathsf{fun}\, f(x).e_1, \mathsf{fun}\, f(x).e_2) \mid \exists a \in \mathbb{N}.\, \forall j < k.\, \forall w' \sqsupseteq w. \\ \quad \forall A \subseteq \mathrm{dom}(w').\, \forall v_1, v_2 \in \mathcal{V}.\, \forall h_1, h_2, h_1', h_2' \in \mathcal{H}. \\ \quad \left[\begin{array}{l} R(\mathrm{FRV}(\varepsilon)) \subseteq \mathrm{dom}(w') \wedge A \# R(\Pi \cup \Lambda) \wedge |A| \geq a \wedge w'(A) = \emptyset \wedge \\ (j,v_1,v_2) \in [\![\tau_1]\!]^R w' \wedge h_1' \subseteq h_1 \wedge h_2' \subseteq h_2 \wedge (j,h_1',h_2') \in \mathbf{P}^{\Lambda,R}_\varepsilon\, w' \end{array} \right] \Rightarrow \\ \quad\quad (j,h_1',h_2',(\mathsf{fun}\, f(x).e_1)\, v_1, (\mathsf{fun}\, f(x).e_2)\, v_2, h_1, h_2) \in \mathbf{safe}^{\Pi,\Lambda,A,R}_{\tau_2,\varepsilon} w', w' \end{array} \right\} & \\ & R(\mathrm{FRV}(\varepsilon)) \subseteq \mathrm{dom}(w) \\ \{(k,v_1,v_2) \mid k \in \mathbb{N} \wedge v_1, v_2 \in \mathcal{V}\} & R(\mathrm{FRV}(\varepsilon)) \nsubseteq \mathrm{dom}(w) \end{cases}$$

Figure 5 Interpretation of types. We require $R : \mathcal{RV} \rightharpoonup_{fin} \mathcal{RN}$ injective with $\mathrm{FRV}(\tau) \subseteq \mathrm{dom}(R)$. We assume $R(\mathrm{FRV}(\tau)) \subseteq |w|$ above, otherwise we define $[\![\tau]\!]^R w$ to be the empty set. We get that $[\![\tau]\!]^R \in \mathbf{T}$.

To conclude this subsection we give a theorem that, combined with the upcoming compatibility, means that logical relatedness implies contextual may-approximation. The proof is in Appendix A.4 and it is not hard, but it is worth noting that we need a proof at all: with sequential languages, this is a result one reads off the definition of the logical relation.

▶ **Theorem 3** (May-Equivalence). *Assume that* $\cdot \mid \cdot \mid \cdot \models e_1 \preceq e_2 : \mathsf{int}, \emptyset$ *holds. Take any* $h_1, h_2 \in \mathcal{H}$*. If there are* e_1', h_1' *with* $(e_1 \mid h_1) \longmapsto^* (e_1' \mid h_1')$ *such that* $\mathrm{irr}(e_1' | h_1')$ *holds, then there is* $n \in \mathbb{Z}$ *such that* $e_1' = \underline{n}$ *and* h_2' *such that* $(e_2 \mid h_2) \longmapsto^* (\underline{n}, h_2')$*.*

Compatibility of the Logical Relation The logical relation is compatible, i.e., respects all typing rules. This is a *sine qua non* of logical relations; it implies the fundamental lemma stating that every well-typed expression is related to itself. And, as discussed just above, it makes the logical relation approximate contextual may-approximation:

▶ **Theorem 4.** $\Pi \mid \Lambda \mid \Gamma \models e_1 \preceq e_2 : \tau, \varepsilon$ *implies* $\Pi \mid \Lambda \mid \Gamma \vdash e_1 \lesssim_\downarrow e_2 : \tau, \varepsilon$*.*

Compatibility means that each typing rule induces a lemma by reading the (unary) typing judgments as the corresponding (binary) logical relations.The three most interesting of these have to do with concurrency and the divide between public and private regions; they are listed here and proofs are given in Appendix A.6:

▶ **Lemma 5.** $\Pi \mid \Lambda, \rho \mid \Gamma \models e_1 \preceq e_2 : \tau, \varepsilon$ *implies* $\Pi \mid \Lambda \mid \Gamma \models e_1 \preceq e_2 : \tau, \varepsilon - \rho$ *provided that* $\rho \notin \mathrm{FRV}(\Gamma, \tau)$*.*

▶ **Lemma 6.** $\cdot \mid \Pi, \Lambda \mid \Gamma \models e_1 \preceq e_2 : \tau, \varepsilon$ *implies* $\Pi \mid \Lambda \mid \Gamma \models \mathsf{atomic}\ e_1 \preceq \mathsf{atomic}\ e_2 : \tau, \varepsilon$ *if* $\mathrm{als}\, \varepsilon \subseteq \mathrm{rds}\, \varepsilon \cap \mathrm{wrs}\, \varepsilon$*.*

▶ **Lemma 7.** $\Pi, \Lambda \mid \cdot \mid \Gamma \models e_1 \preceq e_2 : \tau, \varepsilon$ *and* $\Pi, \Lambda \mid \cdot \mid \Gamma \models e_1^\dagger \preceq e_2^\dagger : \tau^\dagger, \varepsilon^\dagger$ *together imply* $\Pi \mid \Lambda \mid \Gamma \models \mathsf{par}\ e_1\ \mathsf{and}\ e_1^\dagger \preceq \mathsf{par}\ e_2\ \mathsf{and}\ e_2^\dagger : \tau \times \tau^\dagger, \varepsilon \cup \varepsilon^\dagger$*.*

$$
\begin{array}{l}
(k, h_1^\circ, h_2^\circ, e_1, e_2, h_1, h_2) \in \mathbf{safe}_{\tau,\varepsilon}^{\Pi,\Lambda,A,R}\, w^\circ, w \\
\quad \iff \\
\forall j \le k.\, \forall w', g_1, g_2, f_1, f_2. \\
\quad \Big[\mathbf{envtran}^{\Pi,\Lambda,A,R}\, w, w' \wedge (j, g_1, g_2) \in \mathbf{P}_\varepsilon^{\Pi,R}\ w' \wedge \\
\quad (g_1, h_1, f_1, g_2, h_2, f_2) \in \mathbf{splits}^{\Pi,\Lambda,A,R}\, w'\Big] \Rightarrow \\
\quad \Big[\mathrm{irr}(e_1 | g_1 \cdot h_1 \cdot f_1) \Rightarrow \\
\quad\quad \exists e_2', w'', h_1', h_2', g_1', g_2'. \\
\quad\quad\quad (e_2 \mid g_2 \cdot h_2 \cdot f_2) \overset{*}{\longmapsto} (e_2' \mid g_2' \cdot h_2' \cdot f_2) \wedge \mathbf{selftran}^{\Pi,\Lambda,A,R}\, w', w'' \wedge \\
\quad\quad\quad \emptyset = (A \cap \mathrm{dom}(w'')) \cup (\mathrm{dom}(w'') \setminus \mathrm{dom}(w')) \wedge g_1 \cdot h_1 = g_1' \cdot h_1' \wedge \\
\quad\quad\quad (g_1', h_1', f_1, g_2', h_2', f_2) \in \mathbf{splits}^{\Pi,\Lambda,\emptyset,R}\, w'' \wedge (j, g_1, g_2, g_1', g_2') \in \mathbf{Q}_\varepsilon^{\Pi,R}\ w', w'' \wedge \\
\quad\quad\quad (j, e_1, e_2') \in [\![\tau]\!]^R\, (w'') \wedge \exists h_1'' \subseteq h_1', h_2'' \subseteq h_2'.\, (j, h_1^\circ, h_2^\circ, h_1'', h_2'') \in \mathbf{Q}_\varepsilon^{\Lambda,R}\ w^\circ, w''\Big] \wedge \\
\quad \Big[\forall e_1', h_1^\dagger, \mu, n \le j.\, (e_1 \mid g_1 \cdot h_1 \cdot f_1) \to_\mu^n (e_1' \mid h_1^\dagger) \Rightarrow \\
\quad\quad \exists e_2', w'', A', h_1', h_2', g_1', g_2'. \\
\quad\quad\quad (e_2 \mid g_2 \cdot h_2 \cdot f_2) \overset{*}{\longmapsto} (e_2' \mid g_2' \cdot h_2' \cdot f_2) \wedge \mathbf{selftran}^{\Pi,\Lambda,A,R}\, w', w'' \wedge \\
\quad\quad\quad A' = (A \cap \mathrm{dom}(w'')) \cup (\mathrm{dom}(w'') \setminus \mathrm{dom}(w')) \wedge h_1^\dagger = g_1' \cdot h_1' \cdot f_1 \wedge \\
\quad\quad\quad (g_1', h_1', f_1, g_2', h_2', f_2) \in \mathbf{splits}^{\Pi,\Lambda,A',R}\, w'' \wedge (j - n, g_1, g_2, g_1', g_2') \in \mathbf{Q}_\varepsilon^{\Pi,R}\ w', w'' \wedge \\
\quad\quad\quad \mu \in \mathbf{effs}_{\varepsilon,h_1'}^{A',R}\, w'' \wedge (j - n, h_1^\circ, h_2^\circ, e_1', e_2', h_1', h_2') \in \mathbf{safe}_{\tau,\varepsilon}^{\Pi,\Lambda,A',R}\, w^\circ, w''\Big]
\end{array}
$$

Figure 6 Safety. The predicate is defined by well-founded induction. Nontrivial requirements are: $\Pi \# \Lambda$, $\mathrm{FRV}(\tau, \varepsilon) \subseteq \Pi \cup \Lambda$, $\mathrm{FV}(e_1, e_2) = \emptyset$, $R : \Pi \cup \Lambda \hookrightarrow |w^\circ|$, $R(\mathrm{FRV}(\varepsilon)) \subseteq \mathrm{dom}(w^\circ)$ and $w \sqsupseteq w^\circ$ with $\mathrm{dom}(w^\circ) \cap R(\Pi \cup \Lambda) \subseteq \mathrm{dom}(w)$, $A \subseteq \mathrm{dom}(w)$ and $A \# R(\Pi \cup \Lambda)$. See Figure 8 for auxiliary definitions. We refer to the contents of the big square brackets as the *prerequisites*, the *termination branch* and the *progress branch*, respectively.

$$
\begin{array}{l}
\Pi \mid \Lambda \mid \Gamma \models e_1 \preceq e_2 : \tau, \varepsilon \\
\quad \iff \\
\exists a \in \mathbb{N}.\, \forall k \in \mathbb{N}.\, \forall w \in \mathbf{W}.\, \forall R : \Pi \cup \Lambda \hookrightarrow |w|.\, \forall A \subseteq \mathrm{dom}(w). \\
\quad \forall \gamma_1, \gamma_2 \in \mathcal{V}^{|\Gamma|}.\, \forall h_1, h_2, h_1', h_2' \in \mathcal{H}. \\
\quad\quad \Big[R(\mathrm{FRV}(\varepsilon)) \subseteq \mathrm{dom}(w) \wedge A \# R(\Pi \cup \Lambda) \wedge |A| \ge a \wedge \forall r \in A.\, w(r) = \emptyset \wedge \\
\quad\quad (k, \gamma_1, \gamma_2) \in [\![\Gamma]\!]^R w \wedge h_1' \subseteq h_1 \wedge h_2' \subseteq h_2 \wedge (k, h_1', h_2') \in \mathbf{P}_\varepsilon^{\Lambda,R} w\Big] \Rightarrow \\
\quad\quad\quad (k, h_1', h_2', e_1[\gamma_1/\Gamma], e_2[\gamma_2/\Gamma], h_1, h_2) \in \mathbf{safe}_{\tau,\varepsilon}^{\Pi,\Lambda,A,R} w, w.
\end{array}
$$

Figure 7 The logical relation with anonymous regions. We require that $\Pi \# \Lambda$, $\mathrm{FRV}(\Gamma, \tau, \varepsilon) \subseteq \Pi \cup \Lambda$ and, as always, that $\mathrm{FV}(e_1, e_2) \in |\Gamma|$.

$$\mathbf{envtran}^{\Pi,\Lambda,A,R}\, w, w' \iff w \sqsubseteq w' \land \forall r \in \mathrm{dom}(w) \cap (R(\Pi \cup \Lambda) \cup A).\, r \in \mathrm{dom}(w') \\ \land \forall r \in \mathrm{dom}(w) \cap (R(\Lambda) \cup A).\, w(r) = w'(r).$$

$$\mathbf{selftran}^{\Pi,\Lambda,A,R}\, w, w' \iff w \sqsubseteq w' \land \forall r \in \mathrm{dom}(w) \setminus A.\, r \in \mathrm{dom}(w') \\ \land \forall r \in \mathrm{dom}(w) \setminus (R(\Pi \cup \Lambda) \cup A).\, w(r) = w'(r).$$

$$(g_1, h_1, f_1, g_2, h_2, f_2) \in \mathbf{splits}^{\Pi,\Lambda,A,R}\, w \iff \\ \mathrm{dom}(h_1) \# \mathrm{dom}(g_1) \# \mathrm{dom}(f_1) \land \mathrm{dom}(h_2) \# \mathrm{dom}(g_2) \# \mathrm{dom}(f_2) \land \\ \mathrm{dom}_1^{\mathrm{R}(\Pi)}(w) = \mathrm{dom}(g_1) \land \mathrm{dom}_1^{\mathrm{R}(\Lambda)\cup\mathrm{A}}(w) \subseteq \mathrm{dom}(h_1) \land \\ \mathrm{dom}_1^{\mathrm{dom(w)}\setminus(\mathrm{R}(\Pi\cup\Lambda)\cup\mathrm{A})}(w) \subseteq \mathrm{dom}(f_1) \land \\ \mathrm{dom}_2^{\mathrm{R}(\Pi)}(w) = \mathrm{dom}(g_2) \land \mathrm{dom}_2^{\mathrm{R}(\Lambda)\cup\mathrm{A}}(w) \subseteq \mathrm{dom}(h_2) \land \\ \mathrm{dom}_2^{\mathrm{dom(w)}\setminus(\mathrm{R}(\Pi\cup\Lambda)\cup\mathrm{A}))}(w) \subseteq \mathrm{dom}(f_2).$$

$$(k, h_1, h_2) \in \mathbf{P}_{\varepsilon}^{\Theta,R}\, w \iff \mathrm{dom}(h_1) = \mathrm{dom}_1^{\mathrm{R}(\Theta)}(w) \land \mathrm{dom}(h_2) = \mathrm{dom}_2^{\mathrm{R}(\Theta)}(w) \land \\ \forall r \in R(\Theta) \cap \mathrm{dom}(w).\, \forall (l_1, l_2, \mu) \in w(r). \\ r \in R(\mathrm{rds}\, \varepsilon) \Rightarrow k > 0 \Rightarrow (k-1, h_1(l_1), h_2(l_2)) \in (\iota\, \mu)(w).$$

$$(k, h_1, h_2, h_1', h_2') \in \mathbf{Q}_{\varepsilon}^{\Theta,R}\, w, w' \iff \\ \mathrm{dom}(h_1) = \mathrm{dom}_1^{\mathrm{R}(\Theta)}(w) \land \mathrm{dom}(h_2) = \mathrm{dom}_2^{\mathrm{R}(\Theta)}(w) \land \\ \mathrm{dom}(h_1') = \mathrm{dom}_1^{\mathrm{R}(\Theta)}(w') \land \mathrm{dom}(h_2') = \mathrm{dom}_2^{\mathrm{R}(\Theta)}(w') \land \\ \big(\forall r \in R(\Theta) \cap \mathrm{dom}(w).\, \forall (l_1, l_2, \mu) \in w(r). \\ \quad [h_1(l_1) = h_1'(l_1) \land h_2(l_2) = h_2'(l_2)] \lor [r \in R(\mathrm{wrs}\, \varepsilon) \land \\ \quad k > 0 \Rightarrow (k-1, h_1'(l_1), h_2'(l_2)) \in (\iota\, \mu)(w')]\big) \land \\ \big(\forall r \in R(\Theta) \cap \mathrm{dom}(w). \\ \quad \forall (l_1, l_2, \mu) \in w'(r) \setminus w(r).\, r \in R(\mathrm{als}\, \varepsilon) \land \\ \quad k > 0 \Rightarrow (k-1, h_1'(l_1), h_2'(l_2)) \in (\iota\, \mu)(w')\big).$$

$$\mu \in \mathbf{effs}_{\varepsilon,h}^{A,R}\, w \iff \{l \mid rd_l \in \mu\} \cap \mathrm{dom}_1(w) \subseteq \mathrm{dom}_1^{\mathrm{R}(\mathrm{rds}\, \varepsilon)\cup\mathrm{A}}(w) \land \\ \{l \mid wr_l \in \mu\} \cap \mathrm{dom}_1(w) \subseteq \mathrm{dom}_1^{\mathrm{R}(\mathrm{wrs}\, \varepsilon)\cup\mathrm{A}}(w) \land \\ \{l \mid al_l \in \mu\} \cap \mathrm{dom}_1(w) \subseteq \mathrm{dom}_1^{\mathrm{R}(\mathrm{als}\, \varepsilon)\cup\mathrm{A}}(w) \land \\ \{l \mid rd_l \in \mu \lor wr_l \in \mu \lor al_l \in \mu\} \setminus \mathrm{dom}_1(w) \subseteq \mathrm{dom}(h).$$

Figure 8 Six auxiliary definitions. The *environment transition* and *self transition* relations are defined for $\Pi \# \Lambda$, $R : \Pi \cup \Lambda \hookrightarrow |w|$, $A \subseteq \mathrm{dom}(w)$ and $R(\Pi \cup \Lambda) \# A$. The *triple-split* relation has the same prerequisites. The *precondition* relation is defined for $R : \mathcal{RV} \rightharpoonup_{fin} |w|$ injective with $\Theta \cup \mathrm{FRV}(\varepsilon) \subseteq \mathrm{dom}(R)$. The *postcondition* relation additionally requires $w' \sqsupseteq w$ such that $\mathrm{dom}(w) \cap R(\Theta) \subseteq \mathrm{dom}(w')$. Finally the *actual-effects* relation expects $R : \mathcal{RV} \rightharpoonup_{fin} |w|$ injective with $\mathrm{FRV}(\varepsilon) \subseteq \mathrm{dom}(R)$ and $A \subseteq \mathrm{dom}(w)$.

4 Applications

4.1 Parallelization Theorem: Disjoint Concurrency

We now explain our Parallelization Theorem, which gives us an easy way to prove properties about the common case of disjoint concurrency, where disjointness is captured using private regions and effect annotations.

▶ **Theorem 8** (Parallelization). *Assuming that*

1. $\Pi, \Lambda \mid \cdot \mid \Gamma \vdash e_1 : \tau_1, \varepsilon_1$,
2. $\Pi, \Lambda \mid \cdot \mid \Gamma \vdash e_2 : \tau_2, \varepsilon_2$,
3. $\operatorname{rds}\varepsilon_1 \cup \operatorname{wrs}\varepsilon_1 \cup \operatorname{rds}\varepsilon_2 \cup \operatorname{wrs}\varepsilon_2 \subseteq \Lambda$,
4. $\operatorname{rds}\varepsilon_1 \cap \operatorname{wrs}\varepsilon_2 = \operatorname{rds}\varepsilon_2 \cap (\operatorname{wrs}\varepsilon_1 \cup \operatorname{als}\varepsilon_1) = \operatorname{wrs}\varepsilon_1 \cap \operatorname{wrs}\varepsilon_2 = \emptyset$,

the following property holds:

$$\Pi \mid \Lambda \mid \Gamma \models \langle e_1, e_2 \rangle \cong \mathsf{par}\ e_1\ \mathsf{and}\ e_2 : \tau_1 \times \tau_2, \varepsilon_1 \cup \varepsilon_2.$$

Intuitively, item 3 keeps the environment from detecting anything, and item 4 prevents the two computations from talking among themselves, thereby making them independent; the $\operatorname{als}\varepsilon_1$ in item 4 is a technicality that we cannot do without. We showed a concrete simple application of this theorem in the Introduction. More generally, example usage includes situations where we operate on two imperative data structures (say linked lists or graphs); if we only mutate parts of the data structures that are in different regions, then we may safely parallelize operations on the data structures.

The masking rule makes it possible to do more optimizations via the Parallelization Theorem: Consider, for simplicity, the familiar example of an efficient implementation *fib* of the Fibonacci function using two local references. We can use the masking rule to give it type and effect $\mathsf{int} \to^{\emptyset,\emptyset}_{\emptyset} \mathsf{int}, \emptyset$. This allows us to view the imperative implementation as pure, and thus by Theorem 8 we find that it is sound to optimize two sequential calls to *fib* to two parallel calls. This may sound like a simple optimization, but the point is that a compiler can perform it automatically, just based on the effect types. It also underlines how we are able to reason about more involved behaviors of concurrent threads, even though the type system provides only rough bounds on interference through the private-public distinction.

The proof of the Parallelization Theorem is quite tricky. Please see the appendix for an informal overview of the proof and the technical details.

4.2 Non-disjoint Concurrency

We now exemplify how our logical relations model can also be used to reason compositionally about equivalences of fine-grained concurrent programs operating on public regions.

Consider the following type

$$\tau \equiv \mathsf{ref}_\rho \mathsf{int} \to^{\rho,\emptyset}_{\{rd_\rho, wr_\rho\}} 1$$

of functions that take an integer reference in a public region, possibly read and write from the reference, and return unit. The following two functions

$$\mathsf{fun}\,\mathsf{inc}_1(x).\ \mathsf{let}\ y = !\,x\ \mathsf{in}\ \mathsf{let}\ z = y + 1\ \mathsf{in}\ \mathsf{if}\ \mathsf{cas}\ (x, y, z)\ \mathsf{then}\ \langle\rangle\ \mathsf{else}\ \mathsf{inc}_1\ (x) \quad \text{and} \quad \mathsf{fun}\,\mathsf{inc}_2(x).\ \mathsf{atomic}\ (x := !\,x + 1)$$

both have type τ. (We have allowed ourselves to use a standard conditional expression; 1 corresponds to true and 0 to false.) Both functions increment the integer given in their reference arguments; inc_1 uses the fine-grained compare-and-swap to do it atomically, whereas

inc_2 uses the brute-force atomic operation. Using our logical relations model, we can prove that inc_1 and inc_2 are contextually equivalent:

$$\rho \mid \cdot \mid \cdot \vdash \mathsf{inc}_1 \approx \mathsf{inc}_2 : \tau, \emptyset. \tag{1}$$

Hence, replacing inc_2 with inc_1 in any well-typed client gives two contextually equivalent expressions. Thus our logical relation models a form of *data abstraction for concurrency* (where we abstract over the granularity of concurrency in the module).

We now show how to use the equivalence of inc_1 and inc_2 to derive equivalences of two different clients using the fine-grained concurrency implementation inc_1.

To this end, consider the following two client programs of type

$$\sigma \equiv \tau \to^{\rho,\emptyset}_{\emptyset} \mathsf{ref}_\rho \mathsf{int} \to^{\rho,\emptyset}_{\{rd_\rho, wr_\rho\}} \mathsf{int},$$

$$\mathsf{fun}\, \mathsf{c}_1(\mathsf{inc}).\lambda\, n.\mathsf{inc}\, n; \mathsf{inc}\, n; !\, n \quad \text{and} \quad \mathsf{fun}\, \mathsf{c}_2(\mathsf{inc}).\lambda\, n.(\mathsf{par}\ \mathsf{inc}\, n\ \mathsf{and}\ \mathsf{inc}\, n); !\, n$$

Note that c_1 makes two sequential calls to inc, whereas c_2 runs the two calls in parallel. Because of the use of compare-and-swap in inc_1, we would hope that the $\mathsf{c}_1\, \mathsf{inc}_1$ and $\mathsf{c}_2\, \mathsf{inc}_1$ are contextually equivalent (in typing context $\rho \mid \emptyset \mid \emptyset$). We can prove that this is indeed the case using compositional reasoning as follows. Using our logical relation, we prove that $\mathsf{c}_1\, \mathsf{inc}_2$ is contextually equivalent to $\mathsf{c}_2\, \mathsf{inc}_2$, i.e.,

$$\rho \mid \cdot \mid \cdot \vdash \mathsf{c}_1\, \mathsf{inc}_2 \approx \mathsf{c}_2\, \mathsf{inc}_2 : \mathsf{ref}_\rho \mathsf{int} \to^{\rho,\emptyset}_{\{rd_\rho, wr_\rho\}} \mathsf{int}, \emptyset. \tag{2}$$

Finally, we conclude that $\mathsf{c}_1\, \mathsf{inc}_1$ is contextually equivalent to $\mathsf{c}_2\, \mathsf{inc}_1$ by transitivity of contextual equivalence (using (1), (2) and (1) again for the respective steps):

$$\mathsf{c}_1\, \mathsf{inc}_1 \approx \mathsf{c}_1\, \mathsf{inc}_2 \approx \mathsf{c}_2\, \mathsf{inc}_2 \approx \mathsf{c}_2\, \mathsf{inc}_1$$

This proof illustrates an important point: to show equivalence of two clients of a module implemented using fine-grained concurrency, it suffices to show that the clients are equivalent wrt. a coarse-grained implementation, and that the coarse-grained implementation is equivalent to the fine-grained implementation. This is often a lot simpler than trying to show the equivalence of the clients wrt. the fine-grained implementation directly. We can think of the coarse-grained implementation of the module (here inc_2) as the *specification* of the module and the fine-grained implementation (here inc_1) as its *implementation*.

The formal proofs of (1) and (2) follow by straightforward induction.

5 Discussion

Gifford and Lucassen [11, 12] originally proposed type-and-effect systems as a static analysis for determining which parts of a higher-order imperative program could be implemented using parallelism. Here we are able to express the formal correctness of these ideas in a succinct way by having a parallel construct in our programming language and establishing the Parallelization Theorem.

In Section 4.2 we showed how contextual equivalence can be used to *state* that compare-and-swap can be used to implement a simple form of locking, and how our logical relations model could be used to *prove* this statement. We believe that it should be possible to give similar succinct statements and proofs of other implementations of synchronization. For instance, we are currently working on a similar relational specification and correctness proof of Peterson's mutual exclusion algorithm, which involves (for succinctness of specification) extending the language with a primitive notion of critical section.

As mentioned earlier, we have deliberately used the same definition of worlds here as in [20]. As discussed there [20, Section 8.2], this notion of world has somewhat limited expressiveness: the only heap invariants we can state are those that relate values at two locations by a semantic type. To increase expressiveness, it would thus be interesting to extend our model using ideas from [9], where worlds are defined using state-transition systems, and then investigate more examples of equivalences.

Recently, Liang et. al. [10] have proposed RGSim, a simulation based on rely-guarantee, to verify program transformations in a concurrent setting. Their actual definition [10, Definition 4] bears some resemblance to our safety relation; indeed, an early draft of *loc.cit.* was a source of inspiration. They have no division of the heap into public and private parts, instead they give a pair of rely and guarantee that, respectively, constrain the interference of the environment and the actions of the computation. Their approach is essentially untyped; one point of view is that we 'auto-instantiate' the many parameters of their simulation based on our typing information. They consider first-order languages with ground store; this obviously keeps life simple, but the example equivalences they give are not.

Our simple example of data abstraction for concurrency in Section 4.2 suggests that there could be a relationship to linearizability. In [28], Filipović et. al. show a formal connection between linearizability and simulation relations, for a simple first-order imperative programming language. We intend to explore whether a similar kind of formal relationship can be established in our higher-order setting.

6 Conclusion and Future Work

We have presented a logical relations model of a new type-and-effect system for a concurrent higher-order ML-like language with general references. We have shown how to use the model for reasoning about both disjoint and non-disjoint concurrency. In particular, we have proved the first automatic Parallelization Theorem for such a rich language.

In this paper, we have focused on may contextual equivalence. Future work includes investigating models for must contextual equivalence. Since our language allows the encoding of countable nondeterminism, must equivalence is non-trivial, and will probably involve indexing over ω_1 rather than ω [29]. Future work also includes extending the model to region and effect polymorphism, as well as the extension to more expressive worlds, and to other concurrency constructs such as fork-join.

In this paper we have used logical relations for reasoning about contextual equivalence for a concurrent higher-order imperative language with a type-and-effect system. In the future, it would be interesting to explore also the application of other methods, such as bisimulations and game semantics.

The authors would like to thank Jan Schwinghammer and Xinyu Feng for discussions of aspects of this work.

References

1 V. Koutavas and M. Wand, "Small bisimulations for reasoning about higher-order imperative programs," in *POPL*, 2006.

2 D. Sangiorgi, N. Kobayashi, and E. Sumii, "Environmental bisimulations for higher-order languages," *TOPLAS*, 2011.

3 E. Sumii, "A complete characterization of observational equivalence in polymorphic λ-calculus with general references," in *CSL*, 2009.

4 J. Laird, "A fully abstract trace semantics for general references," in *ICALP*, 2007.

5 A. Murawski and N. Tzevelekos, "Game semantics for good general references," in *LICS*, 2011.

6 A. Ahmed, "Semantics of types for mutable state," Ph.D. dissertation, Princeton University, 2004.

7 A. Ahmed, D. Dreyer, and A. Rossberg, "State-dependent representation independence," in *POPL*, 2009.

8 L. Birkedal, J. Thamsborg, and K. Støvring, "Realizability semantics of parametric polymorphism, general references, and recursive types," in *FOSSACS*, 2009.

9 D. Dreyer, G. Neis, and L. Birkedal, "The impact of higher-order state and control effects on local relational reasoning," in *ICFP 2010*. ACM, 2010, pp. 143–156.

10 H. Liang, X. Feng, and M. Fu, "A rely-guarantee-based simulation for verifying concurrent program transformations," in *POPL*, 2012.

11 D. Gifford and J. Lucassen, "Integrating functional and imperative programming," in *LISP and Functional Programming*, 1986.

12 J. Lucassen and D. Gifford, "Polymorphic effect systems," in *POPL*, 1988.

13 M. Tofte and J.-P. Talpin, "Implementation of the typed call-by-value λ-calculus using a stack of regions," in *Proceedings of POPL*, 1994.

14 F. Henglein, H. Makholm, and H. Niss, "Effect types and region-based memory management," in *Advanced Topics in Types and Programming Languages*, B. Pierce, Ed. MIT Press, 2005.

15 L. Birkedal, M. Tofte, and M. Vejlstrup, "From region inference to von Neumann machines via region representation inference," in *POPL*, 1996.

16 N. Benton, A. Kenney, M. Hofmann, and L. Beringer, "Reading, writing and relations: Towards extensional semantics for effect analyses," in *APLAS*, 2006.

17 N. Benton and P. Buchlovsky, "Semantics of an effect analysis for exceptions," in *TLDI*, 2007.

18 N. Benton, L. Beringer, M. Hofmann, and A. Kennedy, "Relational semantics for effect-based program transformations with dynamic allocation," in *PPDP*. ACM, 2007.

19 ——, "Relational semantics for effect-based program transformations: Higher-order store," in *PPDP*. ACM, 2009.

20 J. Thamsborg and L. Birkedal, "A Kripke logical relation for effect-based program transformations," in *ICFP*, 2011.

21 P. W. O'Hearn, "Resources, concurrency, and local reasoning," *TCS*, 2007.

22 M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis, "Deny-guarantee reasoning," in *ESOP*, 2009.

23 V. Vafeiadis and M. Parkinson, "A marriage of rely/guarantee and separation logic," in *CONCUR*, 2007.

24 X. Feng, R. Ferreira, and Z. Shao, "On the relationship between concurrent separation logic and assume-guarantee reasoning," in *ESOP*, 2007.

25 V. Vafeiadis, "Concurrent separation logic and operational semantics," in *MFPS*, 2011.

26 A. Buisse, L. Birkedal, and K. Støvring, "A step-indexed Kripke model of separation logic for storable locks," in *MFPS*, 2011.

27 L. Birkedal, B. Reus, J. Schwinghammer, K. Støvring, J. Thamsborg, and H. Yang, "Step-indexed Kripke models over recursive worlds," in *POPL*, 2011.

28 I. Filipovic, P. W. O'Hearn, N. Rinetzky, and H. Yang, "Abstraction for concurrent objects," *TCS*, 2010.

29 J. Schwinghammer and L. Birkedal, "Step-indexed relational reasoning for countable nondeterminism," in *CSL*, 2011.

A Appendix

A.1 Typing of Contexts

As usually, typing rules for contexts are simple extensions of the type basic type system. It is, however, interesting to observe the region dynamics in the analogues of masking and parallel composition rules, since these allow contexts (when read top-to-bottom) to "kill off" private regions and change public regions into private ones. Selected rules are presented in Figure 9.

$$\frac{}{[\,] : (\Pi \,|\, \Lambda \,|\, \Gamma \vdash \tau,\, \varepsilon) \rightsquigarrow (\Pi \,|\, \Lambda \,|\, \Gamma \vdash \tau,\, \varepsilon)}$$

$$\frac{C : (\Pi \,|\, \Lambda \,|\, \Gamma \vdash \tau,\, \varepsilon) \rightsquigarrow (\Pi' \,|\, \Lambda', \rho \,|\, \Gamma' \vdash \tau',\, \varepsilon')}{C : (\Pi \,|\, \Lambda \,|\, \Gamma \vdash \tau,\, \varepsilon) \rightsquigarrow (\Pi' \,|\, \Lambda' \,|\, \Gamma' \vdash \tau',\, \varepsilon' - \rho)}\ (\rho \notin \mathrm{FRV}(\Gamma', \tau'))$$

$$\frac{C : (\Pi \,|\, \Lambda \,|\, \Gamma \vdash \tau,\, \varepsilon) \rightsquigarrow (\Pi', \Lambda' \,|\, \cdot \,|\, \Gamma' \vdash \tau_1,\, \varepsilon_1) \qquad \Pi', \Lambda' \,|\, \cdot \,|\, \Gamma' \vdash e : \tau_2,\, \varepsilon_2}{\mathsf{par}\ C\ \mathsf{and}\ e : (\Pi \,|\, \Lambda \,|\, \Gamma \vdash \tau,\, \varepsilon) \rightsquigarrow (\Pi' \,|\, \Lambda' \,|\, \Gamma' \vdash \tau_1 \times \tau_2,\, \varepsilon_1 \cup \varepsilon_2)}$$

$$\frac{C : (\Pi \,|\, \Lambda \,|\, \Gamma \vdash \tau,\, \varepsilon) \rightsquigarrow (\Pi' \,|\, \Lambda' \,|\, \Gamma' \vdash \tau_1 \to^{\Pi',\Lambda'}_{\varepsilon'} \tau_2,\, \varepsilon_1) \qquad \Pi' \,|\, \Lambda' \,|\, \Gamma' \vdash e : \tau_1,\, \varepsilon_2}{C\, e : (\Pi \,|\, \Lambda \,|\, \Gamma \vdash \tau,\, \varepsilon) \rightsquigarrow (\Pi' \,|\, \Lambda' \,|\, \Gamma' \vdash \tau_2,\, \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon')}$$

$$\frac{C : (\Pi \,|\, \Lambda \,|\, \Gamma \vdash \tau,\, \varepsilon) \rightsquigarrow (\Pi' \,|\, \Lambda' \,|\, \Gamma' \vdash \tau_1,\, \varepsilon_2) \qquad \Pi' \,|\, \Lambda' \,|\, \Gamma' \vdash e : \tau_1 \to^{\Pi',\Lambda'}_{\varepsilon'} \tau_2,\, \varepsilon_1}{e\, C : (\Pi \,|\, \Lambda \,|\, \Gamma \vdash \tau,\, \varepsilon) \rightsquigarrow (\Pi' \,|\, \Lambda' \,|\, \Gamma' \vdash \tau_2,\, \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon')}$$

$$\frac{C : (\Pi \,|\, \Lambda \,|\, \Gamma \vdash \tau,\, \varepsilon) \rightsquigarrow (\Pi' \,|\, \Lambda' \,|\, \Gamma', f : \tau_1 \to^{\Pi,\Lambda}_{\varepsilon'} \tau_2, x : \tau_1 \vdash \tau_2,\, \varepsilon')}{\mathsf{fun}\, f(x).C : (\Pi \,|\, \Lambda \,|\, \Gamma \vdash \tau,\, \varepsilon) \rightsquigarrow (\Pi' \,|\, \Lambda' \,|\, \Gamma' \vdash \tau_1 \to^{\Pi,\Lambda}_{\varepsilon'} \tau_2,\, \emptyset)}$$

Figure 9 Selected typing rules for contexts

A.2 Well-definedness of Interpretation of Types

As explained in the main text, we import the worlds and types of Thamsborg and Birkedal [20] wholesale; hence their non-trivial construction is for free. The details of the construction, including metric prerequisites, can be found in Appendices A.1 and A.2 of the long version of *loc. cit.*; it is available online.[1] Thus it only remains to verify that interpreting a syntactic type actually gives a semantic ditto.

In metric terms, this means that for any syntactic type τ and any $R : \mathcal{RV} \multimap_{fin} \mathcal{RN}$ with $\mathrm{FRV}(\tau) \subseteq \mathrm{dom}(R)$ we must have

$$[\![\tau]\!]^R \in \mathbf{T} = \mathbf{W} \to_{mon} \mathit{URel}(\mathcal{V}),$$

i.e., it should map worlds to indexed, downwards closed relations on values in a non-expansive and monotone manner. To prove this, we need two lemmas on the pre- and postcondition relations:

▶ **Lemma 9.** *We have* $\mathbf{P}^{\Theta,R}_{\varepsilon}\ w \in \mathit{URel}(\mathcal{H})$. *And for* $w_1, w_2 \in \mathbf{W}$ *with* $w_1 \overset{n}{=} w_2$ *we have* $\mathbf{P}^{\Theta,R}_{\varepsilon}\ w_1 \overset{n}{=} \mathbf{P}^{\Theta,R}_{\varepsilon}\ w_2$ *as well.*

[1] `www.itu.dk/people/thamsborg/longcarnival.pdf`

▶ **Lemma 10.** *We have* $\mathbf{Q}_\varepsilon^{\Theta,R}\ w, w' \in \mathit{URel}(\mathcal{H} \times \mathcal{H})$. *And for* $w_1, w_1', w_2, w_2' \in \mathbf{W}$ *with* $w_1 \overset{n}{=} w_2$ *and* $w_1' \overset{n}{=} w_2'$ *we have* $\mathbf{Q}_\varepsilon^{\Theta,R}\ w_1, w_1' \overset{n}{=} \mathbf{Q}_\varepsilon^{R}\ w_2, w_2'$ *too.*

The proofs a quite straightforward. Then, by simultaneous induction, we can prove the following two propositions and we are done:

▶ Proposition 11. We have $[\![\tau]\!]^R \in \mathbf{T}$.

▶ Proposition 12. For $w_1^\circ, w_1, w_2^\circ, w_2 \in \mathbf{W}$ we have that $w_1^\circ \overset{n}{=} w_2^\circ$ together with $w_1 \overset{n}{=} w_2$ implies $\mathbf{safe}_{\tau,\varepsilon}^{\Pi,\Lambda,A,R}\ w_1^\circ, w_1 \overset{n}{=} \mathbf{safe}_{\tau,\varepsilon}^{\Pi,\Lambda,A,R}\ w_2^\circ, w_2$.

A.3 Instrumented Operational Semantics

$$
\begin{aligned}
(E[\mathsf{proj}_i\ \langle v_1, v_2\rangle]\,|\,h) &\to^1_\emptyset (E[v_i]\,|\,h)\\
(E[(\mathsf{fun}\ f(x).e)\ v]\,|\,h) &\to^1_\emptyset (E[e[f{:=}\mathsf{fun}\ f(x).e, x{:=}v]]\,|\,h)\\
(E[\mathsf{ref}\ v]\,|\,h) &\to^1_{\{al_l\}} (E[l]\,|\,h[l \mapsto v]) \text{ if } l \notin \mathit{dom}(h)\\
(E[l := v]\,|\,h) &\to^1_{\{wr_l\}} (E[\langle\rangle]\,|\,h[l{:=}v]) \text{ if } l \in \mathit{dom}(h)\\
(E[!\,l]\,|\,h) &\to^1_{\{rd_l\}} (E[h(l)]\,|\,h) \quad \text{if } l \in \mathit{dom}(h)\\
(E[\mathsf{par}\ v_1\ \mathsf{and}\ v_2]\,|\,h) &\to^1_\emptyset (E[\langle v_1, v_2\rangle]\,|\,h)\\
(E[\mathsf{cas}\ (l, n_1, n_2)]\,|\,h) &\to^1_{\{rd_l, wr_l\}} (E[\mathsf{1}]\,|\,h[l{:=}n_2])\\
&\qquad \text{if } l \in \mathit{dom}(h) \text{ and } h(l) = n_1\\
(E[\mathsf{cas}\ (l, n_1, n_2)]\,|\,h) &\to^1_{\{rd_l\}} (E[\mathsf{0}]\,|\,h)\\
&\qquad \text{if } l \in \mathit{dom}(h) \text{ and } h(l) \neq n_1\\
(E[\mathsf{atomic}\ e]\,|\,h) &\to^{n+1}_\mu (E[v]\,|\,h') \text{ if } (e\,|\,h) \overset{*}{\to}{}^n_\mu (v\,|\,h')\\
(E[\mathsf{atomic}\ e]\,|\,h) &\to^1_\emptyset (E[\mathsf{atomic}\ e]\,|\,h)
\end{aligned}
$$

$$
\begin{aligned}
(e\,|\,h) \overset{*}{\to}{}^0_\emptyset (e'\,|\,h') &\iff e = e' \wedge h = h'\\
(e\,|\,h) \overset{*}{\to}{}^n_\mu (e''\,|\,h'') &\iff (e\,|\,h) \to^{n_1}_{\mu_1} (e'\,|\,h') \wedge\\
&\qquad (e'\,|\,h') \overset{*}{\to}{}^{n_2}_{\mu_2} (e''\,|\,h'') \wedge\\
&\qquad n = n_1 + n_2 \wedge \mu = \mu_1 \cup \mu_2.
\end{aligned}
$$

Instrumented operational semantics is presented above. $(e\,|\,h) \to^n_\mu (e'\,|\,h')$ is defined for $e, e' \in \mathcal{E}$ with $\mathrm{FV}(e, e') = \emptyset$, $h, h' \in \mathcal{H}$, $n \geq 1$ and $\mu \subseteq \{al_l \mid l \in \mathcal{L}\} \cup \{wr_l \mid l \in \mathcal{L}\} \cup \{rd_l \mid l \in \mathcal{L}\}$; the starred version additionally permits a zero as superscript. Note that $(e|h) \longmapsto (e'|h')$ if and only if there are n and μ such that $(e\,|\,h) \to^n_\mu (e'\,|\,h')$.

A.4 Proof of May-Equivalence Theorem

Theorem 3. *Assume that* $\cdot\,|\,\cdot\,|\,\cdot \models e_1 \preceq e_2 : \mathsf{int}, \emptyset$ *holds. Take any* $h_1, h_2 \in \mathcal{H}$. *If there are* e_1', h_1' *with* $(e_1\,|\,h_1) \overset{*}{\longmapsto} (e_1'\,|\,h_1')$ *such that* $\mathrm{irr}(e_1'|h_1')$ *holds, then there is* $n \in \mathbb{Z}$ *such that* $e_1' = \underline{n}$ *and* h_2' *such that* $(e_2\,|\,h_2) \overset{*}{\longmapsto} (\underline{n}, h_2')$.

Proof. Pick, by assumption, $n \in \mathbb{N}$ such that $(e_1\,|\,h_1) \overset{n}{\longmapsto} (e_1'\,|\,h_1')$. We name the configurations in this multi-step reduction $(e_1^0\,|\,h_1^0), (e_1^1\,|\,h_1^1), \ldots, (e_1^n\,|\,h_1^n)$ in order, i.e., such that

$$
\begin{aligned}
(e_1\,|\,h_1) = (e_1^0\,|\,h_1^0) \longmapsto (e_1^1\,|\,h_1^1) &\longmapsto \cdots\\
&\longmapsto (e_1^n\,|\,h_1^n) = (e_1'\,|\,h_1').
\end{aligned}
$$

For each of these standard reductions, there is a corresponding instrumented reduction. This means, that we can pick $n_1, n_2, \ldots, n_n \in \mathbb{N}$ non-zero and $\mu_1, \mu_2, \ldots, \mu_n$ such that

$$(e_1^0 \mid h_1^0) \to_{\mu_1}^{n_1} (e_1^1 \mid h_1^1) \to_{\mu_2}^{n_2} \cdots \to_{\mu_n}^{n_n} (e_1^n \mid h_1^n).$$

For convenience, we furthermore write $N_m = \sum_{i=m+1}^{n} n_i$ for each $0 \leq m \leq n$.

Now let $a \in \mathbb{N}$ be the required number of anonymous regions according to the definition of the logical relation, and let $w \in \mathbf{W}$ be any world with a empty, live regions and nothing else. We now prove by induction that for all $0 \leq m \leq n$ the following holds:

$$\begin{aligned}&\exists e_2', h_2', w'.\, (e_2 \mid h_2) \longmapsto^{*} (e_2' \mid h_2') \wedge \\ &\quad \mathrm{dom}(h_1^m) \supseteq \mathrm{dom}_1(w') \wedge \mathrm{dom}(h_2') \supseteq \mathrm{dom}_2(w') \wedge \\ &\quad (N_m, [], [], e_1^m, e_2', h_1^m, h_2') \in \mathbf{safe}_{\mathsf{int},\emptyset}^{\emptyset,\emptyset,\mathrm{dom}(w'),\emptyset} w, w'.\end{aligned}$$

The base case is easy by the definition of the logical relation: Pick $e_2' = e_2$, $h_2' = h_2$, and $w' = w$ and apply the assumption that $\cdot \mid \cdot \mid \cdot \models e_1 \preceq e_2 : \mathsf{int}, \emptyset$.

For the inductive case, let $0 < m \leq n$ and unroll the assumptions for $m-1$: we have e_2', h_2', w' with $(e_2 \mid h_2) \longmapsto^{*} (e_2' \mid h_2')$ and $\mathrm{dom}(h_1^{m-1}) \supseteq \mathrm{dom}_1(w')$, $\mathrm{dom}(h_2') \supseteq \mathrm{dom}_2(w')$, and

$$(N_{m-1}, [], [], e_1^{m-1}, e_2', h_1^{m-1}, h_2') \in \mathbf{safe}_{\mathsf{int},\emptyset}^{\emptyset,\emptyset,\mathrm{dom}(w'),\emptyset} w, w'.$$

By the overall setup, we know that $(e_1^{m-1} \mid h_1^{m-1}) \to_{\mu_m}^{n_m} (e_1^m \mid h_1^m)$ and the progress branch of safety provides for us: There are e_2'' and h_2'' with $(e_2' \mid h_2') \longmapsto^{*} (e_2'' \mid h_2'')$ and hence $(e_2 \mid h_2) \longmapsto^{*} (e_2'' \mid h_2'')$. There is w'' with $\mathrm{dom}(h_1^m) \supseteq \mathrm{dom}_1(w'')$ and $\mathrm{dom}(h_2'') \supseteq \mathrm{dom}_2(w')$ and, finally,

$$(N_m, [], [], e_1^m, e_2'', h_1^m, h_2'') \in \mathbf{safe}_{\mathsf{int},\emptyset}^{\emptyset,\emptyset,\mathrm{dom}(w''),\emptyset} w, w''.$$

We have now finished the induction proof; all that remains is to observe that the property proved implies the overall goal in the case $m = n$ by the termination branch of safety. ◀

A.5 Properties of the Pre- and Postcondition Relations

▶ **Lemma 13** (Precondition Separation). *Assume that we have* $\Pi \# \Lambda$, $h_1 = f_1 \restriction_{\mathrm{dom}_1^{\mathrm{R}(\Lambda)}(w)}$, $g_1 = f_1 \restriction_{\mathrm{dom}_1^{\mathrm{R}(\Pi)}(w)}$, $h_2 = f_2 \restriction_{\mathrm{dom}_2^{\mathrm{R}(\Lambda)}(w)}$ *and* $g_2 = f_2 \restriction_{\mathrm{dom}_2^{\mathrm{R}(\Pi)}(w)}$. *Then it holds that*

$$\begin{aligned}&(k, h_1, h_2) \in \mathbf{P}_{\varepsilon}^{\Lambda,R}\ w \wedge (k, g_1, g_2) \in \mathbf{P}_{\varepsilon}^{\Pi,R}\ w \\ &\quad \Longleftrightarrow \\ &(k, f_1, f_2) \in \mathbf{P}_{\varepsilon}^{\Pi\cup\Lambda,R}\ w\end{aligned}$$

▶ **Lemma 14** (Postcondition Separation). *Assume that we have* $\Pi \# \Lambda$, $h_1 = f_1 \restriction_{\mathrm{dom}_1^{\mathrm{R}(\Lambda)}(w)}$, $g_1 = f_1 \restriction_{\mathrm{dom}_1^{\mathrm{R}(\Pi)}(w)}$, $h_2 = f_2 \restriction_{\mathrm{dom}_2^{\mathrm{R}(\Lambda)}(w)}$, $g_2 = f_2 \restriction_{\mathrm{dom}_2^{\mathrm{R}(\Pi)}(w)}$, $h_1' = f_1' \restriction_{\mathrm{dom}_1^{\mathrm{R}(\Lambda)}(w')}$, $g_1' = f_1' \restriction_{\mathrm{dom}_1^{\mathrm{R}(\Pi)}(w')}$, $h_2' = f_2' \restriction_{\mathrm{dom}_2^{\mathrm{R}(\Lambda)}(w')}$ *and* $g_2' = f_2' \restriction_{\mathrm{dom}_2^{\mathrm{R}(\Pi)}(w')}$. *Then*

$$\begin{aligned}&(k, h_1, h_2, h_1', h_2') \in \mathbf{Q}_{\varepsilon}^{\Lambda,R}\ w, w' \wedge \\ &(k, g_1, g_2, g_1', g_2') \in \mathbf{Q}_{\varepsilon}^{\Pi,R}\ w, w' \\ &\quad \Longleftrightarrow \\ &(k, f_1, f_2, f_1', f_2') \in \mathbf{Q}_{\varepsilon}^{\Pi\cup\Lambda,R}\ w, w'.\end{aligned}$$

▶ **Lemma 15** (Precondition Composition)**.**

$$(k, h_1, h_2) \in \mathbf{P}_{\varepsilon}^{\Theta,R} w \wedge (k, h_1, h_2, h_1', h_2') \in \mathbf{Q}_{\varepsilon}^{\Theta,R} w, w'$$
$$\Longrightarrow$$
$$(k, h_1', h_2') \in \mathbf{P}_{\varepsilon}^{\Theta,R} w'.$$

▶ **Lemma 16** (Postcondition Composition)**.**

$$(k, h_1, h_2, h_1', h_2') \in \mathbf{Q}_{\varepsilon}^{\Theta,R} w, w' \wedge$$
$$(k, h_1', h_2', h_1'', h_2'') \in \mathbf{Q}_{\varepsilon}^{\Theta,R} w', w''$$
$$\Longrightarrow$$
$$(k, h_1, h_2, h_1'', h_2'') \in \mathbf{Q}_{\varepsilon}^{\Theta,R} w, w''.$$

A.6 Some Cases of Proof of Compatibility

Lemma 5. *$\Pi \,|\, \Lambda, \rho \,|\, \Gamma \models e_1 \preceq e_2 : \tau, \varepsilon$ implies $\Pi \,|\, \Lambda \,|\, \Gamma \models e_1 \preceq e_2 : \tau, \varepsilon - \rho$ provided that $\rho \notin \mathrm{FRV}(\Gamma, \tau)$.*

Proof. Let $a \in \mathbb{N}$ be the required number of anonymous regions from the assumption. To prove the desired, we unroll the definition of the logical relation: choose $a+1$, pick arbitrary $k^\circ \in \mathbb{N}$, $w^\circ \in \mathbf{W}$, $R : \Pi \cup \Lambda \hookrightarrow |w^\circ|$, $A^\circ \subseteq \mathrm{dom}(w^\circ)$, $\gamma_1, \gamma_2 \in \mathcal{V}^{|\Gamma|}$ and $h_1^\circ, h_2^\circ, h_1'^\circ, h_2'^\circ \in \mathcal{H}$. Assume that $R(\mathrm{FRV}(\varepsilon - \rho)) \subseteq \mathrm{dom}(w^\circ)$, $A^\circ \# R(\Pi \cup \Lambda)$, $|A^\circ| \geq a$, $\forall r \in A^\circ. w^\circ(r) = \emptyset$, $(k^\circ, \gamma_1, \gamma_2) \in [\![\Gamma]\!]^R w^\circ$, $h_1'^\circ \subseteq h_1^\circ$, $h_2'^\circ \subseteq h_2^\circ$, and $(k^\circ, h_1'^\circ, h_2'^\circ) \in \mathbf{P}_{\varepsilon-\rho}^{\Lambda,R} w^\circ$. We must show that $(k^\circ, h_1'^\circ, h_2'^\circ, e_1[\gamma_1/\Gamma], e_2[\gamma_2/\Gamma], h_1^\circ, h_2^\circ) \in \mathbf{safe}_{\tau,\varepsilon-\rho}^{\Pi,\Lambda,A^\circ,R} w^\circ, w^\circ$.

We need, obviously, to make use of the assumption of the lemma. To do so, we claim that to have

$$(k, h_1'^\circ, h_2'^\circ, e_1, e_2, h_1, h_2) \in \mathbf{safe}_{\tau,\varepsilon-\rho}^{\Pi,\Lambda,A,R} w^\circ, w \tag{3}$$

it suffices to know that

$$(k, h_1'^\circ, h_2'^\circ, e_1, e_2, h_1, h_2) \in \mathbf{safe}_{\tau,\varepsilon}^{\Pi,\Lambda,\rho,A\setminus\{r\},R[\rho\mapsto r]} w^\circ, w \tag{4}$$

whenever $r \in A$ holds. Observe first, that if we can prove this, then we are done by an easy application of the assumption of the lemma. So all that remains is to prove the claim; this we do by well-founded induction on $k \in \mathbb{N}$.

Assume that the claim holds for all naturals strictly less than $k \in \mathbb{N}$; we will try to prove it for k. To prove (3) we proceed to unroll the definition of safety. Pick arbitrary $j \leq k$, $w' \in \mathbf{W}$ and $g_1, g_2, f_1, f_2 \in \mathcal{H}$. Assume that the prerequisites hold, i.e., that we have $\mathbf{envtran}^{\Pi,\Lambda,A,R} w, w'$, $(j, g_1, g_2) \in \mathbf{P}_{\varepsilon-\rho}^{\Pi,R} w'$ and $(g_1, h_1, f_1, g_2, h_2, f_2) \in \mathbf{splits}^{\Pi,\Lambda,A,R} w'$. Before we look into the termination and progress branches, observe that the w' and the g_1, g_2, f_1, f_2 match the prerequisites of safety for (4) as well, since $R(\Lambda) \cup A = R[\rho \mapsto r](\Lambda, \rho) \cup A \setminus \{r\}$.

So we follow the termination branch and assume that $\mathrm{irr}(e_1 | g_1 \cdot h_1 \cdot f_1)$ holds. From (4) we get e_2', w'', h_1', h_2', g_1' and g_2' with properties aplenty:

- $(e_2 \,|\, g_2 \cdot h_2 \cdot f_2) \xmapsto{*} (e_2' \,|\, g_2' \cdot h_2' \cdot f_2)$.
- $\mathbf{selftran}^{\Pi,\Lambda,\rho,A\setminus\{r\},R[\rho\mapsto r]} w', w''$.
- $\emptyset = ((A \setminus \{r\}) \cap \mathrm{dom}(w'')) \cup (\mathrm{dom}(w'') \setminus \mathrm{dom}(w'))$.
- $g_1 \cdot h_1 = g_1' \cdot h_1'$.
- $(g_1', h_1', f_1, g_2', h_2', f_2) \in \mathbf{splits}^{\Pi,\Lambda,\rho,\emptyset,R[\rho\mapsto r]} w''$.

- $(j, e_1, e_2) \in [\![\tau]\!]^{R[\rho\mapsto r]}(w'')$.
- $(j, g_1, g_2, g_1', g_2') \in \mathbf{Q}_{\varepsilon}^{\Pi, R[\rho\mapsto r]}\ w', w''$.
- $\exists h_1'' \subseteq h_1', h_2'' \subseteq h_2'.\, (j, h_1'^{\circ}, h_2'^{\circ}, h_1'', h_2'') \in \mathbf{Q}_{\varepsilon}^{\Lambda,\rho,R[\rho\mapsto r]}\ w^{\circ}, w''$.

Now note that we must have $r \in \mathrm{dom}(w'')$ and so we define the world w''' by $w'' \to_{mask(r)} w'''$, i.e., we kill off region r. So we need to discharge a number of obligations; most are straightforward, but the type of the resulting expressions as well as the postcondition on the local heaps take a bit of scrutiny: Note first that we have

$$[\![\tau]\!]^{R[\rho\mapsto r]}(w'') \subseteq [\![\tau]\!]^{R[\rho\mapsto r]}(w''') = [\![\tau]\!]^{R}(w'')$$

by type monotonicity and since $\rho \notin \mathrm{FRV}(\tau)$; this means that the resulting expressions are indeed well-typed. Now define heaps $h_1'' \subseteq h_1'$ and $h_2'' \subseteq h_2'$ by demanding that $\mathrm{dom}(h_1'') = \mathrm{dom}_1^{\mathrm{R}(\Lambda)}(w''')$ and $\mathrm{dom}(h_2'') = \mathrm{dom}_2^{\mathrm{R}(\Lambda)}(w''')$, i.e., we restrict to the private parts of the local heaps, not including the locations (formerly) in region r. One easily verifies that this gives

$$(j, h_1^{\circ}, h_2^{\circ}, h_1'', h_2'') \in \mathbf{Q}_{\varepsilon-\rho}^{\Lambda,R}\ w^{\circ}, w'''$$

and the termination branch is done.

Finally we get to the progress branch: we assume that there are $e_1', h_1^{\dagger}$, μ and $n \leq j$ such that $(e_1 \mid g_1 \cdot h_1 \cdot f_1) \to_{\mu}^{n} (e_1' \mid h_1^{\dagger})$. From (4) we get e_2', w'', A', h_1', h_2', g_1', g_2' with a range of properties:

- $(e_2 \mid g_2 \cdot h_2 \cdot f_2) \xmapsto{*} (e_2' \mid g_2' \cdot h_2' \cdot f_2)$.
- $\mathbf{selftran}^{\Pi,\Lambda,\rho,A\setminus\{r\},R[\rho\mapsto r]}\ w', w''$.
- $A' = (A \setminus \{r\} \cap \mathrm{dom}(w'')) \cup (\mathrm{dom}(w'') \setminus \mathrm{dom}(w'))$.
- $h_1^{\dagger} = g_1' \cdot h_1' \cdot f_1$.
- $\mu \in \mathbf{effs}_{\varepsilon,h_1'}^{A',R[\rho\mapsto r]}\ w''$.
- $(g_1', h_1', f_1, g_2', h_2', f_2) \in \mathbf{splits}^{\Pi,\Lambda,\rho,A',R[\rho\mapsto r]}\ w''$.
- $(j-n, g_1, g_2, g_1', g_2') \in \mathbf{Q}_{\varepsilon}^{\Pi,R[\rho\mapsto r]}\ w', w''$.
- $(j-n, h_1'^{\circ}, h_2'^{\circ}, e_1', e_2', h_1', h_2') \in \mathbf{safe}_{\tau,\varepsilon}^{\Pi,\Lambda,\rho,A',R[\rho\mapsto r]}\ w^{\circ}, w''$.

Now we must have $r \in \mathrm{dom}(w'')$ and letting $A'' = (A \cap \mathrm{dom}(w'')) \cup (\mathrm{dom}(w'') \setminus \mathrm{dom}(w'))$ gives $r \in A''$ as well as $A'' \setminus \{r\} = A'$. With A'' as the choice of new anonymous regions, the obligations in proving the termination branch of (3) are easily met; we just remark that we rely on the induction hypothesis to establish

$$(j-n, h_1'^{\circ}, h_2'^{\circ}, e_1', e_2', h_1', h_2') \in \mathbf{safe}_{\tau,\varepsilon-\rho}^{\Pi,\Lambda,A'',R} w^{\circ}, w''.$$

◀

There are many details to the proof, but the idea is simple: we simultaneously view the computation in two ways, both with ρ as a private region and with ρ masked out. From the safety of former, we then get safety of the latter. In the masked out case, ρ is no longer a private region variable and the region name r associated with ρ joins the anonymous regions. In most cases, however, (region names associated with) private regions and anonymous regions are treated the same, so there is work to do only when anonymous regions are considered in isolation.

Lemma 6. $\cdot \mid \Pi, \Lambda \mid \Gamma \models e_1 \preceq e_2 : \tau, \varepsilon$ *implies* $\Pi \mid \Lambda \mid \Gamma \models \mathsf{atomic}\ e_1 \preceq \mathsf{atomic}\ e_2 : \tau, \varepsilon$ *if* $\mathrm{als}\,\varepsilon \subseteq \mathrm{rds}\,\varepsilon \cap \mathrm{wrs}\,\varepsilon$.

Proof. Let $a \in \mathbb{N}$ be the required number of anonymous regions from the assumption. To prove the desired, we unroll the definition of the logical relation: choose a, pick arbitrary $k^\circ \in \mathbb{N}$, $w^\circ \in \mathbf{W}$, $R : \Pi \cup \Lambda \hookrightarrow |w^\circ|$, $A \subseteq \mathrm{dom}(w^\circ)$, $\gamma_1, \gamma_2 \in \mathcal{V}^{|\Gamma|}$ and $h_1^\circ, h_2^\circ, h_1'^\circ, h_2'^\circ \in \mathcal{H}$. Assume that $R(\mathrm{FRV}(\varepsilon)) \subseteq \mathrm{dom}(w^\circ)$, $A \# R(\Pi \cup \Lambda)$, $|A| \geq a$, $\forall r \in A.\, w^\circ(r) = \emptyset$, $(k^\circ, \gamma_1, \gamma_2) \in [\![\Gamma]\!]^R w^\circ$, $h_1'^\circ \subseteq h_1^\circ$, $h_2'^\circ \subseteq h_2^\circ$, and $(k^\circ, h_1'^\circ, h_2'^\circ) \in \mathbf{P}_\varepsilon^{\Lambda,R} w^\circ$. We must show that $(k^\circ, h_1'^\circ, h_2'^\circ, \mathsf{atomic}\ e_1[\gamma_1/\Gamma], \mathsf{atomic}\ e_2[\gamma_2/\Gamma], h_1^\circ, h_2^\circ) \in \mathbf{safe}_{\tau,\varepsilon}^{\Pi,\Lambda,A,R} w^\circ, w^\circ$.

We need to generalize a bit to handle the loopy behavior of atomic: we prove that for any $k \leq k^\circ$ and any $w \in \mathbf{W}$ with $\mathbf{envtran}^{\Pi,\Lambda,A,R} w^\circ, w$ we have

$$(k, h_1'^\circ, h_2'^\circ, \mathsf{atomic}\ e_1[\gamma_1/\Gamma], \mathsf{atomic}\ e_2[\gamma_2/\Gamma], h_1^\circ, h_2^\circ) \in \mathbf{safe}_{\tau,\varepsilon}^{\Pi,\Lambda,A,R} w^\circ, w.$$

This we do by well-founded induction on k. Unroll the definition of safety. Pick arbitrary $j \leq k$, $w' \in \mathbf{W}$ and $g_1, g_2, f_1, f_2 \in \mathcal{H}$. Assume that the prerequisites hold, i.e., that we have $\mathbf{envtran}^{\Pi,\Lambda,A,R} w, w'$, $(j, g_1, g_2) \in \mathbf{P}_\varepsilon^{\Pi,R} w'$ and $(g_1, h_1, f_1, g_2, h_2, f_2) \in \mathbf{splits}^{\Pi,\Lambda,A,R} w'$. Atomic commands always have the possibility of looping; hence we need not consider the termination branch. So we get to the progress branch: we assume that there are $e_1', h_1^\dagger$, μ and $n \leq j-1$ such that $(\mathsf{atomic}\ e_1[\gamma_1/\Gamma] \,|\, g_1 \cdot h_1 \cdot f_1) \to_\mu^{n+1} (e_1' \,|\, h_1^\dagger)$ and we must match this.

Observe first that $\mathbf{envtran}^{\Pi,\Lambda,A,R} w^\circ, w'$ holds. If, now, the configuration loops to itself, we take no steps on the right hand side and conclude with the induction hypothesis. So we are left to consider the case where $(e_1[\gamma_1/\Gamma] \,|\, g_1 \cdot h_1 \cdot f_1) \to_\mu^n (e_1' \,|\, h_1^\dagger)$ and $e_1' \in \mathcal{V}$. Let $m \in \mathbb{N}$ be the number of reduction steps, we name the configurations $(e_1^0 \,|\, h_1^0), (e_1^1 \,|\, h_1^1), \ldots, (e_1^m \,|\, h_1^m)$ in order and pick $n_1, n_2, \ldots, n_m \in \mathbb{N}$ non-zero and $\mu_1, \mu_2, \ldots, \mu_m$ such that

$$\begin{aligned}(e_1[\gamma_1/\Gamma] \,|\, g_1 \cdot h_1 \cdot f_1) = (e_1^0 \,|\, h_1^0) &\to_{\mu_1}^{n_1} (e_1^1 \,|\, h_1^1) \to_{\mu_2}^{n_2} \cdots \\ &\to_{\mu_m}^{n_m} (e_1^m \,|\, h_1^m) = (e_1' \,|\, h_1^\dagger)\end{aligned}$$

with $n = n_1 + n_2 + \cdots + n_m$ and $\mu = \mu_1 \cup \mu_2 \cup \cdots \cup \mu_m$. For convenience, we furthermore write $N_j = \sum_{i=j+1}^m n_i$ for each $0 \leq j \leq m$.

It is time for the crux: for any $0 \leq j \leq m$ there are $w'', A' \subseteq \mathrm{dom}(w''), h_1', e_2', h_2'$ such that

- $(e_2[\gamma_2/\Gamma] \,|\, g_2 \cdot h_2 \cdot f_2) \overset{*}{\longmapsto} (e_2' \,|\, h_2' \cdot f_2)$
- $\mathbf{selftran}^{\cdot,\Pi,\Lambda,A,R} w', w''$
- $h_1' \cdot f = h_1^j$
- $\bigcup_{i=1}^j \mu_i \setminus (\mathrm{dom}_1^{\mathrm{R}(\Pi\cup\Lambda)}(w'') \setminus \mathrm{dom}_1^{\mathrm{R}(\Pi\cup\Lambda)}(w')) \in \mathbf{effs}_{\varepsilon,h_1'}^{A',R} w''$
- $([], h_1', f_1, [], h_2', f_2) \in \mathbf{splits}^{\cdot,\Pi,\Lambda,A',R} w''$
- $(N_j + j - n, g_1 \cdot h_1, g_2 \cdot h_2, e_1^j, e_2', h_1', h_2') \in \mathbf{safe}_{\tau,\varepsilon}^{\cdot,\Pi,\Lambda,A',R} w', w''$.

Notice the abuse of notation in item four: we do not really remove locations, we remove any actual effects tagged with the listed locations. The base case follows from the assumption of the lemma; the induction simply by unrolling the safety. In the end, we are able to use the properties in the case $j = m$ to produce a right hand side reduction in the overall proof. There are many details to this; here we just remark that it would be more more pleasant to have

$$\bigcup_{i=1}^j \mu_i \in \mathbf{effs}_{\varepsilon,h_1'}^{A',R} w''$$

as item four. Unfortunately, that would break in the inductive step, and we are stuck with the more complex version, which again forces the side condition on the rule. ◀

Lemma 7. *$\Pi, \Lambda \mid \cdot \mid \Gamma \models e_1 \preceq e_2 : \tau, \varepsilon$ and $\Pi, \Lambda \mid \cdot \mid \Gamma \models e_1^\dagger \preceq e_2^\dagger : \tau^\dagger, \varepsilon^\dagger$ together implies $\Pi \mid \Lambda \mid \Gamma \models \mathsf{par}\ e_1\ \mathsf{and}\ e_1^\dagger \preceq \mathsf{par}\ e_2\ \mathsf{and}\ e_2^\dagger : \tau \times \tau^\dagger, \varepsilon \cup \varepsilon^\dagger$.*

Proof. Let $a_1, a_2 \in \mathbb{N}$ be the required numbers of anonymous regions from the respective assumptions. To prove the desired, we unroll the definition of the logical relation: choose $a_1 + a_2$, pick arbitrary $k^\circ \in \mathbb{N}$, $w^\circ \in \mathbf{W}$, $R : \Pi \cup \Lambda \hookrightarrow |w^\circ|$, $A^\circ \subseteq \mathrm{dom}(w^\circ)$, $\gamma_1, \gamma_2 \in \mathcal{V}^{|\Gamma|}$ and $h_1^\circ, h_2^\circ, h_1'^\circ, h_2'^\circ \in \mathcal{H}$. Assume that $R(\mathrm{FRV}(\varepsilon - \rho)) \subseteq \mathrm{dom}(w^\circ)$, $A^\circ \# R(\Pi \cup \Lambda)$, $|A^\circ| \geq a_1 + a_2$, $\forall r \in A^\circ. w^\circ(r) = \emptyset$, $(k^\circ, \gamma_1, \gamma_2) \in [\![\Gamma]\!]^R\, w^\circ$, $h_i'^\circ \subseteq h_i^\circ$, and $(k^\circ, h_1'^\circ, h_2'^\circ) \in \mathbf{P}^{\Lambda,R}_{\varepsilon_1 \cup \varepsilon_2}\, w^\circ$. We must show that $(k^\circ, h_1'^\circ, h_2'^\circ, \mathsf{par}\ e_1\ \mathsf{and}\ e_1^\dagger[\gamma_1/\Gamma], \mathsf{par}\ e_2\ \mathsf{and}\ e_2^\dagger[\gamma_2/\Gamma], h_1^\circ, h_2^\circ) \in \mathbf{safe}^{\Pi,\Lambda,A^\circ,R}_{\tau_1 \times \tau_2, \varepsilon_1 \cup \varepsilon_2}\, w^\circ, w^\circ$.

We need, obviously, to make use of the assumptions of the lemma. To this end, we claim that to have

$$(k, h_1'^\circ, h_2'^\circ, \mathsf{par}\ e_1^1\ \mathsf{and}\ e_1^2, \mathsf{par}\ e_2^1\ \mathsf{and}\ e_2^2, \\ h_1 \cdot h_1^1 \cdot h_1^2, h_2 \cdot h_2^1 \cdot h_2^2) \in \mathbf{safe}^{\Pi,\Lambda,A_1 \cup A_2,R}_{\tau_1 \times \tau_2, \varepsilon_1 \cup \varepsilon_2}\, w^\circ, w, \tag{5}$$

it suffices to know that

$$(k, h_1, h_2) \in \mathbf{P}^{\Lambda,R}_{\varepsilon_1 \cup \varepsilon_2}\, w \tag{6}$$

$$(h, h_1'^\circ, h_2'^\circ, h_1, h_2) \in \mathbf{Q}^{\Lambda,R}_{\varepsilon_1 \cup \varepsilon_2}\, w^\circ, w \tag{7}$$

$$(k, \emptyset, \emptyset, e_1^1, e_2^1, h_1^1, h_2^1) \in \mathbf{safe}^{\Pi \cup \Lambda, \cdot, A_1, R}_{\tau_1, \varepsilon_1}\, w^\circ, w \tag{8}$$

$$(k, \emptyset, \emptyset, e_1^2, e_2^2, h_1^2, h_2^2) \in \mathbf{safe}^{\Pi \cup \Lambda, \cdot, A_2, R}_{\tau_2, \varepsilon_2}\, w^\circ, w, \tag{9}$$

whenever $A_1 \# A_2$ and $\mathrm{dom_i}^{A_j}(w) \subseteq \mathrm{dom}(h_i^j)$ holds. Observe, that if we can prove this, then we are done by splitting A° into two parts of appropriate sizes followed by an easy application of the assumptions of the lemma. The remaining part is to prove the claim, this we do by complete induction on k.

Assume the claim holds for all naturals strictly less then $k \in \mathbb{N}$; we will now prove it also holds for k. To prove 5 we unroll the definition of safety. Pick arbitrary $j \leq k, w' \in \mathbf{W}$ and $g_1, g_2, f_1, f_2 \in \mathcal{H}$. Assume the prerequisites hold, i.e., that we have $\mathbf{envtran}^{\Pi,\Lambda,A_1 \cup A_2,R}\, w, w'$, $(j, g_1, g_2) \in \mathbf{P}^{\Pi,R}_{\varepsilon_1 \cup \varepsilon_2}\, w'$ and $(g_1, h_1 \cdot h_1^1 \cdot h_1^2, f_1, g_2, h_2 \cdot h_2^1 \cdot h_2^2, f_2) \in \mathbf{splits}^{\Pi,\Lambda,A_1 \cup A_2,R}\, w'$. Observe that we can instantiate assumptions 8 and 9 by taking $g_i \cdot h_i$ as the public part of the heap, and adding the spare private part to the frame.

We follow the termination branch first, and assume that $\mathrm{irr}(\mathsf{par}\ e_1^1\ \mathsf{and}\ e_1^2 | g_1 \cdot h_1 \cdot h_1^1 \cdot h_1^2)$. This means each of the subexpressions is also irreducible so from safety assumptions we learn, among other things, that $e_1^1, e_1^2 \in \mathcal{V}$. However, this means that $(\mathsf{par}\ e_1^1\ \mathsf{and}\ e_1^2 | g_1 \cdot h_1 \cdot h_1^1 \cdot h_1^2) \longmapsto (\langle e_1^1, e_1^2 \rangle | g_1 \cdot h_1 \cdot h_1^1 \cdot h_1^2)$, which contradicts the irreducibility assumption.

For the progress branch, we are left with three possibilities of reduction: either both e_1^1 and e_1^2 are values, or the reduction happens inside one of them. We omit the case for reduction inside e_1^2, since it is completely symmetric to the one for e_1^1; the other two cases follow.

Assume that $(e_1^1 \mid g_1 \cdot h_1 \cdot h_1^1 \cdot h_1^2 \cdot f_1) \to_\mu^n (e_1'^1 \mid h_1^\dagger)$ for some $n \leq j$. From 8 we get $e_2'^1$, w'', $h_1', h_2', g_1', g_2', h_1'^1$ and $h_2'^1$, along with the following properties:

- $(e_2^1 | g_2 \cdot h_2 \cdot h_2^1 \cdot h_2^2 \cdot f_2) \longmapsto^* (e_2'^1 | g_2' \cdot h_2' \cdot h_2'^1 \cdot h_2^2 \cdot f_2)$,
- $\mathbf{selftran}^{\Pi \cup \Lambda, \cdot, A_1, R}\, w', w''$,
- $A_1' = (A_1 \cap \mathrm{dom}(w'')) \cup (\mathrm{dom}(w'') \setminus \mathrm{dom}(w'))$,
- $h_1^\dagger = g_1' \cdot h_1' \cdot h_1'^1 \cdot h_1^2 \cdot f_1$,
- $\mu \in \mathbf{effs}^{A_1', R}_{\varepsilon_2, h_1'^1}\, w''$

- $(g'_1 \cdot h'_1, h'^1_1, f_1 \cdot h^2_1, g'_2 \cdot h'_2, h'^1_2, f_2 \cdot h^2_2) \in \mathbf{splits}^{\Pi\cup\Lambda,\cdot,A'_2,R}\, w''$,
- $(j-n, g_1 \cdot h_1, g_2 \cdot h_2, g'_1 \cdot h'_1, g'_2 \cdot h'_2) \in \mathbf{Q}^{\Pi\cup\Lambda,R}_{\varepsilon_1}\ w', w''$,
- $(j-n, \emptyset, \emptyset, e'^1_1, e'^1_2, h'^1_1, h'^2_1) \in \mathbf{safe}^{\Pi\cup\Lambda,\cdot,A'_2,R}_{\tau_1,\varepsilon_1}\, w^\circ, w''$.

Obviously, we can now replay the reduction sequence for the whole right-hand-side expression. By postcondition separation and postcondition weakening lemmas, we can also split up the postcondition into separate parts for Π and Λ, the first of which is needed by progress branch. The other obligations of the progress branch are simple, leaving us with the final safety requirement: $(j-n, h'^\circ_1, h'^\circ_2, \mathsf{par}\ e'^1_1\ \mathsf{and}\ e^2_1, \mathsf{par}\ e'^1_2\ \mathsf{and}\ e^2_2, h'_1 \cdot h'^1_1 \cdot h^2_1, h'_2 \cdot h'^1_2 \cdot h^2_2) \in \mathbf{safe}^{\Pi,\Lambda,A'_1\cup A_2,R}_{\tau_1\times\tau_2,\varepsilon_1\cup\varepsilon_2}\, w^\circ, w''$. This follows by induction hypothesis, since $n > 0$, if we can show the four assumptions. 6 and 7 hold due to downwards-closure and composition lemmas for pre- and post-condition, and the Λ part of the assumption above, and 8 we have verbatim as an assumption. This leaves us only 9 to show. To show it, notice that safety is both downwards closed in the step index, and closed under environment transitions and observe that we have $\mathbf{envtran}^{\Pi\cup\Lambda,\cdot,A_2,R}\, w, w''$ by composing the two world transitions above. This, in conjunction with the original assumption, suffices to finish this part of the proof.

The final case we consider is when $e^1_1, e^2_1 \in \mathcal{V}$ and $(\mathsf{par}\ e^1_1\ \mathsf{and}\ e^2_1 \mid g_1 \cdot h_1 \cdot h^1_1 \cdot h^2_1) \to^1_\emptyset (\langle e^1_1, e^2_1\rangle \mid g_1 \cdot h_1 \cdot h^1_1 \cdot h^2_1)$. In this case, we start with instantiating 8 and from the termination branch obtain:

- $(e^1_2 | g_2 \cdot h_2 \cdot h^1_2 \cdot h^2_2 \cdot f_2) \longmapsto^* (e'^1_2 | g'_2 \cdot h'_2 \cdot h'^1_2 \cdot h^2_2 \cdot f_2)$,
- $\mathbf{selftran}^{\Pi\cup\Lambda,\cdot,A_1,R}\, w', w''$,
- $\emptyset = (A_1 \cap \mathrm{dom}(w'')) \cup (\mathrm{dom}(w'') \setminus \mathrm{dom}(w'))$,
- $g_1 \cdot h_1 \cdot h^1_1 = g'_1 \cdot h'_1 \cdot h'^1_1$,
- $(j, e^1_1, e'^1_2) \in [\![\tau_1]\!]^R\,(w'')$,
- $(j, g_1 \cdot h_1, g_2 \cdot h_2, g'_1 \cdot h'_1, g'_2 \cdot h'_2) \in \mathbf{Q}^{\Pi\cup\Lambda,R}_{\varepsilon_1}\ w', w''$.

Now, with the new heaps, we can instantiate 9, and get:

- $(e^2_2 | g'_2 \cdot h'_2 \cdot h'^1_2 \cdot h^2_2 \cdot f_2) \longmapsto^* (e'^2_2 | g''_2 \cdot h''_2 \cdot h'^1_2 \cdot h'^2_2 \cdot f_2)$,
- $\mathbf{selftran}^{\Pi\cup\Lambda,\cdot,A_2,R}\, w'', w'''$,
- $\emptyset = (A_2 \cap \mathrm{dom}(w''')) \cup (\mathrm{dom}(w''') \setminus \mathrm{dom}(w''))$,
- $g'_1 \cdot h'_1 \cdot h^2_1 = g''_1 \cdot h''_1 \cdot h'^2_1 s$,
- $(j, e^1_2, e'^2_2) \in [\![\tau_2]\!]^R\,(w''')$,
- $(j, g'_1 \cdot h'_1, g'_2 \cdot h'_2, g''_1 \cdot h''_1, g''_2 \cdot h''_2) \in \mathbf{Q}^{\Pi\cup\Lambda,R}_{\varepsilon_2}\ w'', w'''$.

Now we can build the complete reduction for the right-hand side: $(\mathsf{par}\ e^1_2\ \mathsf{and}\ e^2_2 | g_2 \cdot h_2 \cdot h^1_2 \cdot h^2_2 \cdot f_2) \longmapsto^* (\langle e'^1_2, e'^2_2\rangle | g''_2 \cdot h''_2 \cdot h'^1_2 \cdot h'^2_2 \cdot f_2)$, and prove the rest of the required obligations: self-transition out of the composition of the two given, postcondition of public heap fragments by postcondition separation, composition and weakening, and the rest by simple manipulations. This leaves the final safety of two pairs to be proven. We unroll the definition again, taking $i \le j-1$, this time only considering the termination branch, since the left-hand side is a value, and do not do any reductions. This leaves us with proving two interesting obligations. The first one is $(i, \langle e^1_1, e^2_1, ,\rangle \langle e'^1_2, e'^2_2\rangle) \in [\![\tau_1 \times \tau_2]\!]^R\,(w^\dagger)$. However, since $w^\dagger \sqsupseteq w'''$, we can easily show this by definition of the denotation of product types, downwards- and future-world-closure, and the value assumptions obtained above. The other interesting obligation is showing the postcondition for the local, Λ part of the heap. This is done by using postcondition separation, composition and weakening on the postcondition properties obtained from assumptions and by assumption 7 itself. ◀

A.7 Proof of the Parallelization Theorem

The proof of the Parallelization Theorem is quite tricky. We sketch the idea of the proof in the next section, then proceed with the technical formulation of potential safety and Catch-up Lemma, as well as formal proofs, in the subsequent one.

A.7.1 Informal overview

The left-to-right direction of the theorem is not too hard, intuitively since sequential sequencing clearly can always be mimicked by parallel reduction. Thus, the hard part of the proof is to show the direction

$$\Pi \mid \Lambda \mid \Gamma \models \mathsf{par}\ e_1\ \mathsf{and}\ e_2 \lesssim_{\downarrow} \langle e_1, e_2\rangle : \tau_1 \times \tau_2, \varepsilon_1 \cup \varepsilon_2. \tag{10}$$

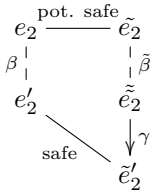
The challenge is that if we take a step of reduction on the left, then we cannot always immediately mimick it on the right, because if it is a step in (a derivative of) e_2 then we might not be ready to make it on the right, since we haven't yet started reducing in e_2 on the right (we only start reducing e_2 on the right when we are done with all the e_1 reductions).

We address this issue by formulating a notion of *potential safety* to relate (a derivative of) e_2 on the left with (a derivative of) e_2 on the right. Safety, and also potential safety, really involves configurations (expressions and heaps) rather than just expression), but to keep the overview, we will continue just using expression notation in the high-level sketch here. Potential safety of e_2 and $\tilde{e}_2$ then allows $\tilde{e}_2$ to *catch up* by doing some reductions to reach $\tilde{e}'_2$, and then e_2 and $\tilde{e}'_2$ should be safe (in the usual sense).

The idea of the proof of (10) is then to show (for suitable e_1, $\tilde{e}_1$, e_2, $\tilde{e}_2$ satisfying conditions corresponding to those in the parallelization theorem) that

$$\begin{array}{l}\mathrm{safe}(e_1, \tilde{e}_1) \wedge \text{potentially-safe}(e_2, \tilde{e}_2) \\ \quad \Longrightarrow\ \mathrm{safe}((\mathsf{par}\ e_1\ \mathsf{and}\ e_2), \langle \tilde{e}_1, \tilde{e}_2\rangle)).\end{array} \tag{11}$$

Now, for this to work, potential safety actually needs to allow for some transitions by e_1 and $\tilde{e}_1$ — and the context — but regulated by the overall assumptions regarding effects, etc., of the parallelization theorem. Thus potential safety of e_2 and $\tilde{e}_2$ actually says that for any good (in the sense that it is governed by overall assumptions regarding effects, etc.) pair of transitions, β and $\tilde{\beta}$, going from e_2 to e'_2 and $\tilde{e_2}$ to $\tilde{\tilde{e}_2}$ respectively, there is some sequence of catching-up reductions, call them γ, that reaches a $\tilde{e'}_2$ such that e'_2 and $\tilde{e}'_2$ are safe. Diagrammatically:



To show (11), we proceed by induction on the index used in the safety predicate. There are three cases to consider, corressponding to possible reductions of $\mathsf{par}\ e_1\ \mathsf{and}\ e_2$. The case where e_1 reduces is not too hard since we have defined potential safety to accomodate such reductions (the β's). The case where e_1 and e_2 are both values is not too hard either, intuitively since we can just run $\tilde{e_2}$ until we reach safety with e_2 at which point the proof becomes simple. The case where e_2 reduces, say to e'_2 by a reduction α, is more tricky. We proceed by showing potential safety of e'_2 and $\tilde{e}_2$ (argument outline follows below), and that

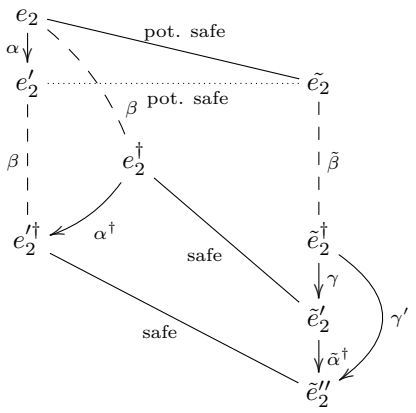


Figure 10 Proving parallelization: recovering potential safety after α transition.

e_1 and $\tilde{e_1}$ stay safe (this follows from the assumption on effects, etc.), which allows us to conclude by induction.

The argument for the potential safety of e_2' and $\tilde{e}_2$ is outlined in Figure 10 By assumption e_2 and $\tilde{e}_2$ are potentially safe, as depicted by the topmost arrow, and we need to show the potential safety depicted by the dashed arrow. Hence we consider any β and $\tilde{\beta}$ as shown, and we need to show that there exists a catching-up reduction γ' from $\tilde{e}_2^\dagger$ to $\tilde{e}_2''$. By properties of the instrumented operational semantics, we can commute α and β to get β and $\alpha^\dagger$, as shown in the diagram (essentially, this is Lemma 2). Hence by the assumption of potential safety of e_2 and $\tilde{e}_2$, we get safety of $e_2^\dagger$ and $\tilde{e}_2'$, as shown. Using this safety and the $\alpha^\dagger$ reduction, we get a reduction $\tilde{\alpha}^\dagger$ and the final required safety of $e'^\dagger_2$ and $\tilde{e}_2''$. Thus the required catching-up reduction γ' is γ followed by $\tilde{\alpha}^\dagger$. Note that the latter makes intuitive sense; we did a reduction on the left in e_2 and now we have correspondingly extended the catching-up reduction to be done on the right.

Formally, the Catch-up lemma in Appendix A.7 gives the precise formulation and proof of (11); item 3 in that lemma gives the precise technical formulation of potential safety.

A.7.2 Technical details

We start by stating and prooving the catch-up lemma mentioned in the explanation in the paper, that covers most of the proof of the theorem.

▶ **Lemma 17** (Catch-up). *Assuming that*

1. $(k, h_1, h_2) \in \mathbf{P}^{\Lambda,R}_{\varepsilon_1}\ w$,
2. $(k, \emptyset, \emptyset, e_1, \widetilde{e_1}, h_{11}, h_{12}) \in \mathbf{safe}^{\Pi\cup\Lambda,\cdot,A_1,R}_{\tau_1,\varepsilon_1}\ w^\circ, w$ *and*
3. *for any* $g_1, g_2, h_1^\dagger, h_2^\dagger, f_1, f_2, w', j \leq k$ *such that*
   - $\mathbf{envtran}^{\Pi\cup\Lambda,\cdot,A_2,R}\ w, w'$
   - $(g_1 \cdot h_1^\dagger, h_{21}, f_1, g_2 \cdot h_2^\dagger, \emptyset, f_2) \in \mathbf{splits}^{\Pi\cup\Lambda,\cdot,A_2,R}\ w'$
   - $(j, h_1, h_2, h_1^\dagger, h_2^\dagger) \in \mathbf{Q}^{\Lambda,R}_{\varepsilon_1}\ w, w'$

   there exist $h_1', h_2', g_1', g_2', h_{22}, h_{21}', \widetilde{e_1'}, w''$ *such that*
   - $\mathbf{selftran}^{\Pi\cup\Lambda,\cdot,A_2,R}\ w', w''$
   - $(\widetilde{e_2}|h_2^\dagger \cdot g_2 \cdot f_2) \longmapsto^* (\widetilde{e_2'}|h_2' \cdot g_2' \cdot h_{22} \cdot f_2)$
   - $h_1^\dagger \cdot g_1 \cdot h_{21} = h_1' \cdot g_1' \cdot h_{21}'$
   - $(k, g_1, g_2, g_1', g_2') \in \mathbf{Q}^{\Pi,R}_{\varepsilon_2}\ w', w''$

- $(k, h_1', h_2') \in \mathbf{P}^{\Lambda,R}_{\varepsilon_2}\ w''$
- $(j, h_1^\circ, h_2^\circ, h_1', h_2') \in \mathbf{Q}^{\Lambda,R}_{\varepsilon_1\cup\varepsilon_2}\ w^\circ, w''$
- $(k, \emptyset, \emptyset, e_2, \widetilde{e_2'}, h_{21}', h_{22}) \in \mathbf{safe}^{\Pi\cup\Lambda,\cdot,A_2',R}_{\tau_2,\varepsilon_2} w^\circ, w'',$

we have $(k, h_1^\circ, h_2^\circ, \mathsf{par}\ e_1\ \mathsf{and}\ e_2, \langle\widetilde{e_1}, \widetilde{e_2}\rangle, h_1\cdot h_{11}\cdot h_{21}, h_2\cdot h_{12}) \in \mathbf{safe}^{\Pi,\Lambda,A_1+A_2,R}_{\tau_1\times\tau_2,\varepsilon_1\cup\varepsilon_2} w^\circ, w$*, as long as* $\operatorname{rds}\varepsilon_1\cup\operatorname{wrs}\varepsilon_1\cup\operatorname{rds}\varepsilon_2\cup\operatorname{wrs}\varepsilon_2 \subseteq \Lambda$*,* $\operatorname{rds}\varepsilon_1\#\operatorname{wrs}\varepsilon_2$*,* $\operatorname{rds}\varepsilon_2\#(\operatorname{wrs}\varepsilon_1\cup\operatorname{als}\varepsilon_1)$*,* $\operatorname{wrs}\varepsilon_1\#\operatorname{wrs}\varepsilon_2$*,* $\operatorname{dom}(h_{i1}) \supseteq \operatorname{dom}_1^{A_i}(w)$*,* $\operatorname{dom}(h_{12}) \supseteq \operatorname{dom}_2^{A_1}(w)$ *and* $\operatorname{dom}_2^{A_2}(w) = \emptyset$*.*

Proof. The proof proceeds by well-founded induction on k. Take any $j \le k, g_1, g_2, f_1, f_2, w_1$ such that

- $\mathbf{envtran}^{\Pi,\Lambda,A_1+A_2,R} w, w_1$
- $(j, g_1, g_2) \in \mathbf{P}^{\Pi,R}_{\varepsilon_1\cup\varepsilon_2}\ w_1$
- $(g_1, h_1\cdot h_{11}\cdot h_{21}, f_1, g_2, h_2\cdot h_{12}, f_2) \in \mathbf{splits}^{\Pi,\Lambda,A_1+A_2,R} w_1$

There are two branches to consider. However, $\mathsf{par}\ e_1\ \mathsf{and}\ e_2$ can always reduce, so the termination branch is trivial. In the progress case, we proceed by case analysis on the reduction to get three subcases.

1. $(e_1 \mid g_1\cdot h_1\cdot h_{11}\cdot h_{21}\cdot f_1) \to^n_\mu (e_1' \mid h_1^\dagger)$

In this case we can use the assumption (2), which we instantiate with $j \le k, g_1\cdot h_1, g_2\cdot h_2, f_1\cdot h_{21}, f_2, w_1$. We need to show the prerequisites:

- $\mathbf{envtran}^{\Pi\cup\Lambda,\cdot,A_1,R} w, w_1$, which holds by env-transition weakening
- $(j+1, g_1\cdot h_1, g_2\cdot h_2) \in \mathbf{P}^{\Pi\cup\Lambda,R}_{\varepsilon_1}\ w_1$, which holds by assumptions, precondition composition, precondition weakening and future-world closure of precondition
- $(g_1\cdot h_1, h_{11}, f_1\cdot h_{21}, g_2\cdot h_2, h_{12}, f_2) \in \mathbf{splits}^{\Pi\cup\Lambda,\cdot,A_1,R} w_1$, which holds by massaging the assumption about splits.

From the progress branch we obtain the following properties:

- $(\widetilde{e_1}|g_2\cdot h_2\cdot h_{12}\cdot f_2) \longmapsto^* (\widetilde{e_1'}|g_2'\cdot h_2'\cdot h_{12}'\cdot f_2)$
- $\mathbf{selftran}^{\Pi\cup\Lambda,\cdot,A_1,R} w_1, w_2$
- $A_1' = (A_1 \cap \operatorname{dom}(w_2)) \cup (\operatorname{dom}(w_2) \setminus \operatorname{dom}(w_1))$
- $h_1^\dagger = g_1'\cdot h_1'\cdot h_{11}'\cdot f_1\cdot h_{21}$
- $\mu \in \mathbf{effs}^{A_1',R}_{\varepsilon_1,h_{11}'} w_2$
- $(g_1'\cdot h_1', h_{11}', f_1\cdot h_{21}, g_2'\cdot h_2', h_{12}', f_2) \in \mathbf{splits}^{\Pi\cup\Lambda,\cdot,A_1',R} w_2$
- $(j-n, g_1\cdot h_1, g_2\cdot h_2, g_1'\cdot h_1', g_2'\cdot h_2') \in \mathbf{Q}^{\Pi\cup\Lambda,R}_{\varepsilon_1}\ w_1, w_2$
- $(j-n, \emptyset, \emptyset, e_1', \widetilde{e_1'}, h_{11}', h_{12}') \in \mathbf{safe}^{\Pi\cup\Lambda,\cdot,A_1',R}_{\tau_1,\epsilon_1} w^\circ, w_2$

From these we can conclude that:

- $(\langle\widetilde{e_1}, \widetilde{e_2}\rangle|g_2\cdot h_2\cdot h_{12}\cdot f_2) \longmapsto^* (\langle\widetilde{e_1'}, \widetilde{e_2}\rangle|g_2'\cdot h_2'\cdot h_{12}'\cdot f_2)$ (recall pairs of expressions is syntactic sugar)
- $\mathbf{selftran}^{\Pi,\Lambda,A_1+A_2,R} w_1, w_2$, by self-transition strengthening
- $A_1'+A_2 = ((A_1+A_2)\cap\operatorname{dom}(w_2))\cup(\operatorname{dom}(w_2)\setminus\operatorname{dom}(w_1))$, by self-transition restrictions
- $(g_1', h_1'\cdot h_{11}'\cdot h_{21}, f_1, g_2', h_2'\cdot h_{12}', f_2) \in \mathbf{splits}^{\Pi,\Lambda,A_1'+A_2,R} w_2$, by massaging the splits assumption
- $\mu \in \mathbf{effs}^{A_1'+A_2,R}_{\varepsilon_1\cup\varepsilon_2,h_1'\cdot h_{11}'\cdot h_{21}} w_2$, by effs strengthening
- $(j-n, g_1, g_2, g_1', g_2') \in \mathbf{Q}^{\Pi,R}_{\varepsilon_1\cup\varepsilon_2}\ w_1, w_2$, by postcondition separation and postcondition weakening
- $(j-n, h_1', h_2') \in \mathbf{P}^{\Lambda,R}_{\varepsilon_1}\ w_2$, by postcondition separation and precondition composition
- $\mathbf{envtran}^{\Pi\cup\Lambda,\cdot,A_2,R} w, w_2$, by envtransition composition and relation of env-transition and self-transition
- $(j-n, h_1, h_2, h_1', h_2') \in \mathbf{Q}^{\Lambda,R}_{\varepsilon_1}\ w, w_2$, by postcondition separation

Since $0 < n \leq j$, we can now use the induction hypothesis to discharge the final safety obligation: we have already obtained preconditions (1) and (2), while (3) is closed under the world transition that we have made (see the final two properties above).

2. $(e_2 \mid g_1 \cdot h_1 \cdot h_{11} \cdot h_{21} \cdot f_1) \to_\mu^n (e_2' \mid h_1^\dagger)$

First, we need to establish that $h_1^\dagger = g_1 \cdot h_1' \cdot h_{21}' \cdot f_1 \cdot h_{11}$, $\mathrm{dom}(h_1') = \mathrm{dom}(h_1)$, $\mu \in \mathbf{effs}^{A,R}_{\varepsilon_2,h_{21}'}\, w$ and $(j-n, h_1', h_2) \in \mathbf{P}^{\Lambda,R}_{\varepsilon_1}\, w_1$. This can be obtained by using the assumption (3), as follows. Take $g_1, g_2, h_1, h_2, f_1 \cdot h_{11}, f_2 \cdot h_{12}, w_1, k \leq k$ as the universally quantified variables in (3). We need to show:

- $\mathbf{envtran}^{\Pi\cup\Lambda,\cdot,A_2,R}\, w_1, w_1$, which holds by reflexivity
- $(g_1 \cdot h_1, h_{21}, f_1 \cdot h_{11}, g_2 \cdot h_2, \emptyset, f_2 \cdot h_{12}) \in \mathbf{splits}^{\Pi\cup\Lambda,\cdot,A_2,R}\, w_1$, by massaging the splits we have assumed earlier
- $(k, h_1, h_2, h_1, h_2) \in \mathbf{Q}^{\Lambda,R}_{\varepsilon_1}\, w_1, w_1$, again, by reflexivity.

From this we obtain a bunch of properties, by unfolding safety. However, we are only interested in a few, namely that $h_1^\dagger = g_1^p \cdot h_1^l \cdot h_1^p \cdot f_1 \cdot h_{11}$, and $\mu \in \mathbf{effs}^{A_2^\dagger,R}_{\varepsilon_2,h_1^p}\, w^\dagger$ and $\mathbf{selftran}^{\Pi\cup\Lambda,\cdot,A_2,R}\, w_1, w^\dagger$. Let $h_1^\ddagger = g_1^p \cdot h_1^l \cdot h_1^p$.

Now, as witnesses for the progress case, we take $e_2, w_1, A_1 + A_2, h_1' \cdot h_{11} \cdot h_{21}', h_2 \cdot h_{12}, g_1, g_2$, where $h_1' = h_1^\ddagger \restriction_{\mathrm{dom}_1^{\mathrm{R}(\Lambda)}(w_1)}$ and $h_{21}' = h_1^\ddagger \restriction_{\mathrm{dom}_1(w_1)\setminus\mathrm{dom}_1^{\mathrm{R}(\Pi\cup\Lambda)}(w_1)}$. It suffices to show:

- $(\widetilde{e_2} | g_2 \cdot h_2 \cdot h_{12} \cdot f_2) \longmapsto^* (\widetilde{e_2} | g_2 \cdot h_2 \cdot h_{12} \cdot f_2)$ by reflexivity
- $\mathbf{selftran}^{\Pi,\Lambda,A_1+A_2,R}\, w_1, w_1$ by reflexivity
- $h_1^\dagger = g_1 \cdot h_1' \cdot h_{11} \cdot h_{21}' \cdot f_1$ by the restriction on actual effects (μ) we know g_1 didn't change, and we have an assumption to finish this off
- $(g_1, h_1' \cdot h_{11} \cdot h_{21}', f_1, g_2, h_2 \cdot h_{12}, f_2) \in \mathbf{splits}^{\Pi,\Lambda,A_1+A_2,R}\, w_1$ by assumption
- $\mu \in \mathbf{effs}^{A_2,R}_{\varepsilon_2,h_{21}'}\, w_1$, since any region killed between w_1 and $w^\dagger$ has to be in A_2, $\mathrm{dom}(h_1^p) \subseteq \mathrm{dom}(h_{21}')$, and we have an assumption for the other effects inside the world
- $(j-n, g_1, g_2, g_1, g_2) \in \mathbf{Q}^{\Pi,R}_{\varepsilon_1\cup\varepsilon_2}\, w_1, w_1$ by reflexivity,

so it remains to show that $(j-n, h_1^\circ, h_2^\circ, \mathsf{par}\ e_1\ \mathsf{and}\ e_2', \langle \widetilde{e_1}, \widetilde{e_2}\rangle, h_1' \cdot h_{11} \cdot h_{21}', h_2 \cdot h_{12}) \in \mathbf{safe}^{\Pi,\Lambda,A_1+A_2,R}_{\tau_1\times\tau_2,\varepsilon_1\cup\varepsilon_2}\, w^\circ, w_1$ holds. We can use induction hypothesis to prove it, provided we show the three assumptions. The first one ($(j-n, h_1', h_2) \in \mathbf{P}^{\Lambda,R}_{\varepsilon_1}\, w_1$) is simple, since μ is confined to ε_2, the second one ($(j-n, \emptyset, \emptyset, e_1, \widetilde{e_1}, h_{11}, h_{12}) \in \mathbf{safe}^{\Pi\cup\Lambda,\cdot,A_1,R}_{\tau_1,\varepsilon_1}\, w^\circ, w_1$) holds by env-transition closure and downwards closure of **safe**. This leaves us the last precondition to show. To this end, we take $h_1^\dagger, h_2^\dagger, g_1^\dagger, g_2^\dagger, f_1^\dagger, f_2\dagger, i \leq j-n$ and w_2, such that

- $\mathbf{envtran}^{\Pi\cup\Lambda,\cdot,A_2,R}\, w_1, w_2$
- $(i, h_1', h_2, h_1^\dagger, h_2^\dagger) \in \mathbf{Q}^{\Lambda,R}_{\varepsilon_1}\, w_1, w_2$
- $(g_1^\dagger \cdot h_1^\dagger, h_{21}', f_1^\dagger, g_2^\dagger \cdot h_2^\dagger, \emptyset, f_2^\dagger) \in \mathbf{splits}^{\Pi\cup\Lambda,\cdot,A_2,R}\, w_2$

Now we can instantiate the original assumption (3) with $h_1[h_1^\dagger/h_1'], h_2^\dagger, g_1^\dagger, g_2^\dagger, f_1^\dagger, f_2^\dagger, i \leq k$ and w_2, where the heap substitution notation means that any change (allocation or update) from h_1' to $h_1^\dagger$ should be replayed on h_1 (this is well-specified, since $\mathrm{dom}(h_1) = \mathrm{dom}(h_1')$). We need to prove the following:

- $\mathbf{envtran}^{\Pi\cup\Lambda,\cdot,A_2,R}\, w, w_2$, by envtran-comp and envtran-weaken
- $(i, h_1, h_2, h_1[h_1^\dagger/h_1'], h_2^\dagger) \in \mathbf{Q}^{\Lambda,R}_{\varepsilon_1}\, w, w_2$, by definition of heap substitution, and assumption
- $(g_1^\dagger \cdot h_1[h_1^\dagger/h_1'], h_{21}, f_1^\dagger, g_2^\dagger \cdot h_2^\dagger, \emptyset, f_2^\dagger) \in \mathbf{splits}^{\Pi\cup\Lambda,\cdot,A_2,R}\, w_2$ by assumption, given that $\mathrm{dom}(h_1^\dagger) = \mathrm{dom}(h_1[h_1^\dagger/h_1'])$

By proving these properties, we obtain the following:

- $(k, h_1^\ddagger, h_2^\ddagger) \in \mathbf{P}^{\Lambda,R}_{\varepsilon_2}\, w_3$
- $\mathbf{selftran}^{\Pi\cup\Lambda,\cdot,A_2,R}\, w_2, w_3$

- $(\widetilde{e_2}|h_2^\dagger \cdot g_2^\dagger \cdot f_2^\dagger) \longmapsto^* (\widetilde{e_2'}|h_2^\ddagger \cdot g_2^\ddagger \cdot h_{22} \cdot f_2^\ddagger)$
- $h_1[h_1^\dagger/h_1'] \cdot g_1^\dagger \cdot h_{21} = h_1^\ddagger \cdot g_1^\ddagger \cdot h_{21}^\ddagger$
- $(k, g_1^\dagger, g_2^\dagger, g_1^\ddagger, g_2^\ddagger) \in \mathbf{Q}^{\Pi,R}_{\varepsilon_2}\ w_2, w_3$
- $(i, h_1^\circ, h_2^\circ, h_1^\ddagger, h_2^\ddagger) \in \mathbf{Q}^{\Lambda,R}_{\varepsilon_1 \cup \varepsilon_2}\ w^\circ, w_3$
- $(g_1^\ddagger \cdot h_1^\ddagger, h_{21}^\ddagger, f_1^\dagger, g_2^\ddagger \cdot h_2^\ddagger, h_{22}, f_2^\dagger) \in \mathbf{splits}^{\Pi \cup \Lambda, \cdot, A_2', R}\ w_3$
- $(k, \emptyset, \emptyset, e_2, e_2', h_{21}^\ddagger, h_{22}) \in \mathbf{safe}^{\Pi \cup \Lambda, \cdot, A_2', R}_{\tau_2, \varepsilon_2}\ w^\circ, w_3$

Recall that $\mu \in \mathbf{effs}^{A_2, R}_{\varepsilon_2, h_{21}'}\ w_1$, and $(e_2 \mid g_1 \cdot h_1 \cdot h_{11} \cdot h_{21} \cdot f_1) \to^n_\mu (e_2' \mid g_1 \cdot h_1' \cdot h_{21}' \cdot f_1 \cdot h_{11})$. Hence, we can use Lemma 2 to get

$$(e_2 \mid g_1^\dagger \cdot h_1[h_1^\dagger/h_1'] \cdot h_{21} \cdot f_1^\dagger) \to^n_\mu (e_2' \mid g_1^\dagger \cdot h_1^\dagger \cdot h_{21}' \cdot f_1^\dagger),$$

which allows us to instantiate the safety predicate above. Note that this argument is precisely the commutation of α and β in the explanation in Section 4.1 (cf. Figure 10). We need to show:

- $\mathbf{envtran}^{\Pi \cup \Lambda, \cdot, A_2', R}\ w_3, w_3$, by reflexivity
- $(j, g_1^\ddagger, g_2^\ddagger) \in \mathbf{P}^{\Pi,R}_{\varepsilon_2}\ w_3$, trivial, since $\mathrm{rds}\, \varepsilon_2 \# \Pi$
- $(j, h_1^\ddagger, h_2^\ddagger) \in \mathbf{P}^{\Lambda,R}_{\varepsilon_2}\ w_3$, by assumption
- $(g_1^\ddagger \cdot h_1^\ddagger, h_{21}^\ddagger, f_1 \cdot h_{11} \cdot f_1^\ddagger, g_2^\ddagger \cdot h_2^\ddagger, h_{22}, f_2^\dagger) \in \mathbf{splits}^{\Pi \cup \Lambda, \cdot, A_2', R}\ w_3$, by massaging the assumption.

These facts, along with the reduction step, give us the following:

- $\mathbf{selftran}^{\Pi \cup \Lambda, \cdot, A_2', R}\ w_3, w_4$
- $(\widetilde{e_2'}|h_2^\ddagger \cdot g_2^\ddagger \cdot h_{22} \cdot f_2^\dagger) \longmapsto^* (\widetilde{e_2''}|g_2^f \cdot h_2^f \cdot h_{22}^f \cdot f_2^\dagger)$
- $g_1^\dagger \cdot h_1^\dagger \cdot h_{21}' = g_1^f \cdot h_1^f \cdot h_{21}^f$
- $(g_1^f \cdot h_1^f, h_{21}^f, f_1^\dagger, g_2^f \cdot h_2^f, h_{22}^f, f_2^\dagger) \in \mathbf{splits}^{\Pi \cup \Lambda, \cdot, A_2'', R}\ w_4$
- $(j - n, g_1^\ddagger \cdot h_1^\ddagger, g_2^\ddagger \cdot h_2^\ddagger, g_1^f \cdot h_1^f, g_2^f \cdot h_2^f) \in \mathbf{Q}^{\Pi \cup \Lambda, R}_{\varepsilon_2}\ w_3, w_4$
- $(j - n, \emptyset, \emptyset, e_2', \widetilde{e_2''}, h_{21}^f, h_{22}^f) \in \mathbf{safe}^{\Pi \cup \Lambda, \cdot, A_2'', R}_{\tau_2, \varepsilon_2}\ w^\circ, w_4$

Now we can show the remaining goals:

- $\mathbf{selftran}^{\Pi \cup \Lambda, \cdot, A_2, R}\ w_2, w_4$, by self-transition composition
- $(\widetilde{e_2}|g_2^\dagger \cdot h_2^\dagger \cdot f_2^\dagger) \longmapsto^* (\widetilde{e_2''}|g_2^f \cdot h_2^f \cdot h_{22}^f \cdot f_2^f)$, by transitivity
- $h_1^\dagger \cdot g_1^\dagger \cdot h_{21}' = g_1^f \cdot h_1^f \cdot h_{21}^f$, by assumption
- $(j - n, g_1^\dagger, g_2^\dagger, g_1^f, g_2^f) \in \mathbf{Q}^{\Pi,R}_{\varepsilon_2}\ w_2, w_4$, by postcondition composition
- $(j - n, h_1^f, h_2^f) \in \mathbf{P}^{\Lambda,R}_{\varepsilon_2}\ w_4$, by precondition composition
- $(i, h_1^\circ, h_2^\circ, h_1^f, h_2^f) \in \mathbf{Q}^{\Lambda,R}_{\varepsilon_1 \cup \varepsilon_2}\ w^\circ, w_4$, by postcondition composition
- $(j - n, g_1^\ddagger \cdot h_1^\ddagger, g_2^\ddagger \cdot h_2^\ddagger, g_1^f \cdot h_1^f, g_2^f \cdot h_2^f) \in \mathbf{Q}^{\Pi \cup \Lambda, R}_{\varepsilon_2}\ w_3, w_4$, by assumption
- $(j - n, \emptyset, \emptyset, e_2', \widetilde{e_2''}, h_{21}^f, h_{22}^f) \in \mathbf{safe}^{\Pi \cup \Lambda, \cdot, A_2'', R}_{\tau_2, \varepsilon_2}\ w^\circ, w_4$, by assumption,

which ends this case.

3. $(\mathsf{par}\ e_1\ \mathsf{and}\ e_2 \mid g_1 \cdot h_1 \cdot h_{11} \cdot h_{21} \cdot f_1) \to^1_\emptyset (\langle e_1, e_2 \rangle \mid g_1 \cdot h_1 \cdot h_{11} \cdot h_{21} \cdot f_1)$

This means in particular that $\mathrm{irr}(e_i | g_1 \cdot h_1 \cdot h_{11} \cdot h_{21} \cdot f_1)$. We only need to consider the case where $j \geq 1$. We start by using the assumption (2). We need to provide:

- $\mathbf{envtran}^{\Pi \cup \Lambda, \cdot, A_1, R}\ w, w_1$, by env-transition weakening
- $(j, g_1 \cdot h_1, g_2 \cdot h_2) \in \mathbf{P}^{\Pi \cup \Lambda, R}_{\varepsilon_1}\ w_1$, by precondition separation, assumption and (1)
- $(g_1 \cdot h_1, h_{11}, f_1 \cdot h_{21}, g_2 \cdot h_2, h_{12}, f_2) \in \mathbf{splits}^{\Pi \cup \Lambda, \cdot, A_1, R}\ w_1$, by massaging the assumption about **splits**.

From the termination case we can now get:

- $(\widetilde{e_1}|g_2 \cdot h_2 \cdot h_{12} \cdot f_2) \longmapsto^* (\widetilde{e_1'}|g_2' \cdot h_2' \cdot h_{12}' \cdot f_2)$
- $\mathbf{selftran}^{\Pi \cup \Lambda, \cdot, A_2, R}\ w_1, w_2$
- $g_1 \cdot h_1 \cdot h_{11} = g_1' \cdot h_1' \cdot h_{11}'$
- $(g_1' \cdot h_1', h_{11}', f_1 \cdot h_{21}, g_2' \cdot h_2', h_{12}', f_2) \in \mathbf{splits}^{\Pi \cup \Lambda, \cdot, \emptyset, R}\ w_2$

- $(j, g_1, g_2, g_1', g_2') \in \mathbf{Q}^{\Pi,R}_{\varepsilon_1}\ w_1, w_2$
- $(j, h_1, h_2, h_1', h_2') \in \mathbf{Q}^{\Lambda,R}_{\varepsilon_1}\ w_1, w_2$
- $(j, e_1, \widetilde{e_1'}) \in [\![\tau_1]\!]^R\, w_2$

Note that $\widetilde{e_1'}$ is irreducible, as it's a value. At this point we can use assumption (3). We need to show:

- $\mathbf{envtran}^{\Pi\cup\Lambda,\cdot,A_2,R}\, w, w_2$, by relation between self-transition and env-transition and env-transition weakening
- $(j, h_1, h_2, h_1', h_2') \in \mathbf{Q}^{\Lambda,R}_{\varepsilon_1}\ w, w_2$, by assumption
- $(g_1' \cdot h_1', h_{21}, f_1 \cdot h_{11}', g_2' \cdot h_2', \emptyset, f_2 \cdot h_{12}') \in \mathbf{splits}^{\Pi\cup\Lambda,\cdot,A_2,R}\, w_2$, by **splits** above.

By these facts we are able to obtain the following:

- $\mathbf{selftran}^{\Pi\cup\Lambda,\cdot,A_2,R}\, w_2, w_3$
- $(\widetilde{e_2}|g_2' \cdot h_2' \cdot f_2 \cdot h_{12}') \longmapsto^* (\widetilde{e_2'}|g_2'' \cdot h_2'' \cdot h_{22} \cdot f_2 \cdot h_{12}')$
- $h_1' \cdot g_1' \cdot h_{21} = h_1'' \cdot g_1'' \cdot h_{21}'$
- $(k, g_1', g_2', g_1'', g_2'') \in \mathbf{Q}^{\Pi,R}_{\varepsilon_2}\ w_2, w_3$
- $(k, h_1'', h_2'') \in \mathbf{P}^{\Lambda,R}_{\varepsilon_2}\ w_3$
- $(j, h_1^\circ, h_2^\circ, h_1'', h_2'') \in \mathbf{Q}^{\Lambda,R}_{\varepsilon_1\cup\varepsilon_2}\ w^\circ, w_3$
- $(k, \emptyset, \emptyset, e_2, \widetilde{e_2'}, h_{21}'', h_{22}) \in \mathbf{safe}^{\Pi\cup\Lambda,\cdot,A_2',R}_{\tau_2,\varepsilon_2}\, w^\circ, w_3$
- $(g_1'' \cdot h_1'', h_{21}'', f_1 \cdot h_{11}', g_2'' \cdot h_2'', h_{22}, f_2 \cdot h_{12}') \in \mathbf{splits}^{\Pi\cup\Lambda,\cdot,A_2',R}\, w_3$.

Finally, we can instantiate the safety predicate obtained above. We need to show:

- $\mathbf{envtran}^{\Pi\cup\Lambda,\cdot,A_2',R}\, w_3, w_3$, by reflexivity
- $(j, h_1'', h_2'') \in \mathbf{P}^{\Lambda,R}_{\varepsilon_2}\ w_3$, by assumption
- $(j, g_1'', g_2'') \in \mathbf{P}^{\Pi,R}_{\varepsilon_2}\ w_3$, trivial, since $\operatorname{rds}\varepsilon_2 \# \Pi$
- $(g_1'' \cdot h_1'', h_{21}'', f_1 \cdot h_{11}', g_2'' \cdot h_2'', h_{22}, f_2 \cdot h_{12}') \in \mathbf{splits}^{\Pi\cup\Lambda,\cdot,A_2',R}\, w_3$, by assumption.

From the termination branch (by irreducibility of e_2), we get:

- $\mathbf{selftran}^{\Pi\cup\Lambda,\cdot,A_2',R}\, w_3, w_4$
- $(\widetilde{e_2'}|g_2'' \cdot h_2'' \cdot h_{22} \cdot f_2 \cdot h_{12}') \longmapsto^* (\widetilde{e_2''}|g_2''' \cdot h_2''' \cdot h_{22}''' \cdot f_2 \cdot h_{12}')$
- $g_1'' \cdot h_1'' \cdot h_{21}'' = g_1''' \cdot h_1''' \cdot h_{21}'''$
- $(j, g_1'', g_2'', g_1''', g_2''') \in \mathbf{Q}^{\Pi,R}_{\varepsilon_2}\ w_3, w_4$
- $(j, h_1'', h_2'', h_1''', h_2''') \in \mathbf{Q}^{\Lambda,R}_{\varepsilon_2}\ w_3, w_4$
- $(j, e_2, \widetilde{e_2''}) \in [\![\tau_2]\!]^R\, w_4$
- $(g_1''' \cdot h_1''', h_{21}''', f_1 \cdot h_{11}', g_2''' \cdot h_2''', h_{22}''', f_2 \cdot h_{12}') \in \mathbf{splits}^{\Pi\cup\Lambda,\cdot,\emptyset,R}\, w_4$

At this point we can finally provide witnesses and prove the remaining properties:

- $\mathbf{selftran}^{\Pi,\Lambda,A_1+A_2,R}\, w_1, w_4$, by weakening and composition of previous self-transitions
- $(\langle\widetilde{e_1}, \widetilde{e_2}\rangle|g_2 \cdot h_2 \cdot h_{12} \cdot f_2) \longmapsto^* (\langle\widetilde{e_1'}, \widetilde{e_2''}\rangle|g_2''' \cdot h_2''' \cdot h_{22}''' \cdot h_{12}' \cdot f_2)$, by transitivity and irreducibility of $\widetilde{e_1'}$
- $\emptyset \in \mathbf{effs}^{\emptyset,R}_{\varepsilon_1\cup\varepsilon_2, h_1'''\cdot h_{11}'\cdot h_{21}'''}\, w_4$, which is trivially true
- $g_1 \cdot h_1 \cdot h_{11} \cdot h_{21} = g_1''' \cdot h_1''' \cdot h_1 1' \cdot h_{21}'''$, by congruence closure of equalities
- $(g_1''', h_1''' \cdot h_{21}''' \cdot h_{11}', f_1, g_2''', h_2''' \cdot h_{22}''' \cdot h_{12}', f_2) \in \mathbf{splits}^{\Pi,\Lambda,\emptyset,R}\, w_4$, by massaging the assumption above
- $(j-1, g_1, g_2, g_1''', g_2''') \in \mathbf{Q}^{\Pi,R}_{\varepsilon_1\cup\varepsilon_2}\ w_1, w_4$, by weakening and composition of postcondition assumptions

This leaves us with the final obligation to prove: $(j-1, h_1^\circ, h_2^\circ, \langle e_1, e_2\rangle, \langle\widetilde{e_1'}, \widetilde{e_2''}\rangle, h_1''' \cdot h_{11}' \cdot h_{21}''', h_2''' \cdot h_{12}' \cdot h_{22}''') \in \mathbf{safe}^{\Pi,\Lambda,\emptyset,R}_{\tau_1\times\tau_2,\varepsilon_1\cup\varepsilon_2}\, w^\circ, w_4$. To do this, we assume:

- $\mathbf{envtran}^{\Pi,\Lambda,\emptyset,R}\, w_4, w_5$
- $(i, g_1^\dagger, g_2^\dagger) \in \mathbf{P}^{\Pi,R}_{\varepsilon_1\cup\varepsilon_2}\ w_5$
- $(g_1^\dagger, h_1''' \cdot h_{11}' \cdot h_{21}''', f_1^\dagger, g_2^\dagger, h_2''' \cdot h_{12}' \cdot h_{22}''', f_2^\dagger) \in \mathbf{splits}^{\Pi,\Lambda,\emptyset,R}\, w_5$

Obviously, both expressions are irreducible, so we take the same values as witnesses. Most obligations are proved by reflexivity or assumption, the ones left are:

- $(i, \langle e_1, e_2\rangle, \langle \widetilde{e_1'}, \widetilde{e_2''}\rangle) \in [\![\tau_1 \times \tau_2]\!]^R w_5$, which holds by donwards- and future-world-closure of type denotation, and definition of product types, and
- $(i, h_1^\circ, h_2^\circ, h_1''', h_2''') \in \mathbf{Q}^{\Lambda,R}_{\varepsilon_1\cup\varepsilon_2}\ w^\circ, w_5$, which holds by weakening, composition and downwards closure of previous postcondition assumptions.

This ends the proof.

◀

▶ **Lemma 18** (Parallelization). *Assuming that*

1. $\Pi, \Lambda \mid\mid \Gamma \vdash e_1 : \tau_1, \varepsilon_1$,
2. $\Pi, \Lambda \mid\mid \Gamma \vdash e_2 : \tau_2, \varepsilon_2$,
3. $\operatorname{rds}\varepsilon_1 \cup \operatorname{wrs}\varepsilon_1 \cup \operatorname{rds}\varepsilon_2 \cup \operatorname{wrs}\varepsilon_2 \subseteq \Lambda$,
4. $\operatorname{rds}\varepsilon_1 \cap \operatorname{wrs}\varepsilon_2 = \operatorname{rds}\varepsilon_2 \cap (\operatorname{wrs}\varepsilon_1 \cup \operatorname{als}\varepsilon_1) = \operatorname{wrs}\varepsilon_1 \cap \operatorname{wrs}\varepsilon_2 = \emptyset$,

the following property holds:

$$\Pi \mid \Lambda \mid \Gamma \models \langle e_1, e_2\rangle \cong \ \mathsf{par}\ e_1\ \mathsf{and}\ e_2 : \tau_1 \times \tau_2, \varepsilon_1 \cup \varepsilon_2.$$

Proof. The proof consists of two parts, for two directions of contextual approximation. For both these direction we need assumptions about e_1 and e_2, that follow by the fundamental theorem of logical relations:

$$\Pi, \Lambda \mid\mid \Gamma \models e_1 \preceq e_1 : \tau_1, \varepsilon_1 \qquad\qquad \Pi, \Lambda \mid\mid \Gamma \models e_2 \preceq e_2 : \tau_2, \varepsilon_2$$

1. To show: $\Pi \mid \Lambda \mid \Gamma \models \langle e_1, e_2\rangle \preceq \mathsf{par}\ e_1\ \mathsf{and}\ e_2 : \tau_1 \times \tau_2, \varepsilon_1 \cup \varepsilon_2$.
   We start this part, by taking a_1 and a_2, the witnesses from the logical relations above, and setting $a = a_1 + a_2$ as the witness for the logical relation. Now we can assume the initial conditions:
   - $R(\mathrm{FRV}(\varepsilon_1 \cup \varepsilon_2)) \subseteq \mathrm{dom}(w)$
   - $A_1 + A_2 \# R(\Pi \cup \Lambda)$
   - $|A_i| \geq a_i$, for $i \in \{1, 2\}$
   - $\forall r \in A_1 + A_2.\ w(r) = \emptyset$
   - $(k, \gamma_1, \gamma_2) \in [\![\Gamma]\!]^R\ w$
   - $h_i^\circ \subseteq h_i$, for $i \in \{1, 2\}$
   - $(k, h_1^\circ, h_2^\circ) \in \mathbf{P}^{\Lambda,R}_{\varepsilon_1\cup\varepsilon_2}\ w$.

   What remains to be shown is safety: $(k, h_1^\circ, h_2^\circ, \langle e_1^1, e_2^1\rangle, \mathsf{par}\ e_1^2\ \mathsf{and}\ e_2^2, h_1, h_2) \in \mathbf{safe}^{\Pi,\Lambda,A_1+A_2,R}_{\tau_1\times\tau_2,\varepsilon_1\cup\varepsilon_2}\ w, w$, where $e_i^j = e_i[\gamma_j/\Gamma]$. We can also use the initial conditions above to obtain:

   $$(k, \emptyset, \emptyset, e_i^1, e_i^2, \emptyset, \emptyset) \in \mathbf{safe}^{\Pi\cup\Lambda,\cdot,A_i,R}_{\tau_i,\varepsilon_i}\ w, w.$$

   We proceed by well-founded induction, keeping the additional assumption that $(k, h_1 \restriction_{\mathrm{dom}_1^{\mathrm{R}(\Lambda)}(w)}, h_2 \restriction_{\mathrm{dom}_2^{\mathrm{R}(\Lambda)}(w)}) \in \mathbf{P}^{\Lambda,R}_{\varepsilon_1\cup\varepsilon_2}\ w$. After unrolling safety, there are three possibilities. Either e_i^1 are both irreducible, in which case we can recover the final reductions for e_i^2, replay those, and reduce the parallel composition to the pair of values related to the left-hand-side pair, or at least one of e_i^1 can reduce. In the latter cases, we can use the respective assumption to obtain the matching reduction, perform it, and use the induction hypothesis (with downwards- and envtran-closure ensuring that the other safety assumption matches and the precondition assumption fulfilled due to Λ being public). The details are very similar to the compatibility of parallel composition, and so are omitted.

2. To show: $\Pi \mid \Lambda \mid \Gamma \models \mathsf{par}\ e_1\ \mathsf{and}\ e_2 \preceq \langle e_1, e_2\rangle : \tau_1 \times \tau_2, \varepsilon_1 \cup \varepsilon_2$.
We proceed in the similar manner, by taking the witnesses, providing $a_1 + a_2$ as the witness for the whole computation, and obtaining the safety obligations: show that $(k, h_1^\circ, h_2^\circ, \mathsf{par}\ e_1^1\ \mathsf{and}\ e_2^1, \langle e_1^2, e_2^2\rangle, h_1, h_2) \in \mathbf{safe}^{\Pi,\Lambda,A_1+A_2,R}_{\tau_1\times\tau_2,\varepsilon_1\cup\varepsilon_2}\, w, w$ provided that $(k, \emptyset, \emptyset, e_i^1, e_i^2, \emptyset, \emptyset) \in \mathbf{safe}^{\Pi\cup\Lambda,\cdot,A_i,R}_{\tau_i,\varepsilon_i}\, w, w$, where $e_i^j = e_i[\gamma_j/\Gamma]$. However, to show it we use the catch-up lemma proved above. To do this, we need to show its three preconditions (we let $h_1^l = h_1 \restriction_{\mathrm{dom}_1^{\mathrm{R}(\Lambda)}(w)}$, $h_{11} = h_1 \setminus h_1^l$, $h_{12} = \emptyset$, and similar for h_2):

- $(k, h_1^l, h_2^l) \in \mathbf{P}^{\Lambda,R}_{\varepsilon_1}\, w$, which holds by pre-weakening,
- $(k, \emptyset, \emptyset, e_1^1, e_1^2, h_{11}, h_{12}) \in \mathbf{safe}^{\Pi\cup\Lambda,\cdot,A_1,R}_{\tau_1,\varepsilon_1}\, w, w$, which is easy to show since h_{1i} is just some random heap outside the control of the world (regions in A_1 are empty),
- and the final catching-up precondition. To show this property, take any $g_1, g_2, h_1^\dagger, h_2^\dagger, f_1, f_2, w', j \leq k$ such that:
  - $\mathbf{envtran}^{\Pi\cup\Lambda,\cdot,A_2,R}\, w, w'$
  - $(g_1 \cdot h_1^\dagger, \emptyset, f_1, g_2 \cdot h_2^\dagger, \emptyset, f_2) \in \mathbf{splits}^{\Pi\cup\Lambda,\cdot,A_2,R}\, w'$
  - $(j, h_1, h_2, h_1^\dagger, h_2^\dagger) \in \mathbf{Q}^{\Lambda,R}_{\varepsilon_1}\, w, w'$

  We take the witnesses to be the precise same things, so most of the obligations hold by reflexivity. The ones left are:
  - $(j, h_1^\circ, h_2^\circ, h_1^\dagger, h_2^\dagger) \in \mathbf{Q}^{\Lambda,R}_{\varepsilon_1\cup\varepsilon_2}\, w, w'$, which by post-weaken, since $h_i^\circ = h_i^l$,
  - $(k, \emptyset, \emptyset, e_2^1, e_2^2, \emptyset, \emptyset) \in \mathbf{safe}^{\Pi\cup\Lambda,\cdot,A_2,R}_{\tau_2,\varepsilon_2}\, w, w'$, which holds by envtran-closure of the safety assumption, and
  - $(k, h_1^\dagger, h_2^\dagger) \in \mathbf{P}^{\Lambda,R}_{\varepsilon_2}\, w'$, which holds because of disjointness condition on ε_1 and ε_2 — when cut down to the ε_2-read regions, $h_i^\dagger = h_i^\circ$, for which we have the right assumption.

This ends the proof.

◀

A Separation Logic for Fictional Sequential Consistency: Technical Appendix

Filip Sieczkowski
IT University of Copenhagen
fisi@itu.dk

Kasper Svendsen
Aarhus University
ksvendsen@cs.au.dk

Lars Birkedal
Aarhus University
birkedal@cs.au.dk

November 28, 2013

1 The Language

We build our logic for a simple, class-based programming language. For simplicity of both semantics and the logic, the language uses let-bindings and expressions, but it stays relatively low-level by ensuring that all the values are machine-word size. The values include variables, natural numbers, booleans, unit, object references (pointers), the null pointer and the special variable **this**. The expressions include values, let bindings, conditionals, constructor and method calls, field reads and writes, atomic compare-and-swap expression and a fork call. The syntax of values and expressions is shown in Figure 1. Note that for simplicity classes can have only one constructor each, and that it has to end with **this**, which ensures that constructors return the newly allocated value. Actual object references $\mathtt{o}$ cannot occur in the text of the program, but need to be values for the operational semantics to be able to reduce.

In Figure 2 we present the semantic domains: the interesting part is the machine state, which in addition to the heap also contains a pool of store buffers, one per thread, and the program state. The latter is only consistent if the thread pool contains the same threads that the store buffer pool contains — or if the program has already gone wrong.

Following the Views framework [1], the operational semantics is split into two components: a thread-local small-step semantics labeled with actions that occur during the step, and action semantics that defines the effect of the action on the machine state. Below, the possible actions are given, and in the Figure 3 we present the thread-local semantics. The interpretation of actions is shown in Figure 6.

$$\begin{array}{rcl} a \in \mathit{Act} & ::= & \varepsilon \mid \mathit{read}(t, o, f, v) \mid \mathit{write}(t, o, f, v) \mid \mathit{cas}(t, o, f, v_o, v_n, r) \mid \mathit{new}(t, C, o) \\ & \mid & \mathit{type}(t, o, T) \mid \mathit{fork}(t, o, T, t') \mid \lightning \end{array}$$

$$
\begin{array}{rll}
\mathsf{ClassDef} ::= & \textbf{class}\ \mathsf{C} = \{\overline{\mathsf{T}\ \mathsf{f}}; \mathsf{CDef}; \overline{\mathsf{MDef}}\} & \text{(Class definition)} \\
\mathsf{MDef} ::= & \mathsf{T}\ \mathsf{m}(\overline{\mathsf{T}\ \mathsf{x}}) = \mathsf{e} & \text{(Method definition)} \\
\mathsf{CDef} ::= & \mathsf{C}(\overline{\mathsf{T}\ \mathsf{x}}) = (\mathsf{e}; \textbf{this}) & \text{(Constructor definition)} \\
\mathsf{Val} \ni \mathsf{v} ::= & \mathsf{x} \mid \textbf{null} \mid \textbf{this} \mid \mathsf{o} \mid \mathsf{n} \mid \mathsf{b} \mid \textbf{()} & \text{(Values)} \\
\mathsf{Expr} \ni \mathsf{e} ::= & \mathsf{v} \mid \textbf{let}\ \mathsf{x} = \mathsf{e}_1\ \textbf{in}\ \mathsf{e}_2 \mid \textbf{if}\ \mathsf{v}\ \textbf{then}\ \mathsf{e}_1\ \textbf{else}\ \mathsf{e}_2 & \text{(Expressions)} \\
& \mid \textbf{new}\ \mathsf{C}(\overline{\mathsf{v}}) \mid \mathsf{v.f} \mid \mathsf{v.f} := \mathsf{v} \mid \mathsf{v.m}(\overline{\mathsf{v}}) \mid \textbf{CAS}(\mathsf{v}_1.\mathsf{f}, \mathsf{v}_2, \mathsf{v}_3) \mid \textbf{fork}(\mathsf{v.m}) & \\
\mathsf{E} ::= & [\,] \mid \textbf{let}\ \mathsf{x} = \mathsf{E}\ \textbf{in}\ \mathsf{e} & \text{(Evaluation Contexts)}
\end{array}
$$

Figure 1: The syntax of the programming language.

$$
\begin{array}{rl}
t \in \mathit{TId} & \text{(Thread Identifiers)} \\
o \in \mathit{OId} & \text{(Object Identifiers)} \\
v \in \mathit{Val} \stackrel{\text{def}}{=} \{\textbf{null}\} + \mathit{OId} + \mathbb{N} + \mathbf{2} + \mathbf{1} & \text{(Semantic Values)} \\
\mathit{OHeap} \stackrel{\text{def}}{=} \mathit{OId} \times \mathit{FName} \rightharpoonup_{fin} \mathit{Val} & \text{(Object Heaps)} \\
\mathit{THeap} \stackrel{\text{def}}{=} \mathit{OId} \rightharpoonup_{fin} \mathit{CName} & \text{(Type Heaps)} \\
h \in \mathit{Heap} \stackrel{\text{def}}{=} \mathit{OHeap} \times \mathit{THeap} & \text{(Heaps)} \\
s \in \mathit{SBuffer} \stackrel{\text{def}}{=} \mathrm{seq}\,(\mathit{OId} \times \mathit{FName} \times \mathit{Val}) & \text{(Store Buffers)} \\
T \in \mathit{TPool} \stackrel{\text{def}}{=} \mathit{TId} \rightharpoonup_{fin} \mathit{Expr} & \text{(Thread Pool)} \\
U \in \mathit{SPool} \stackrel{\text{def}}{=} \mathit{TId} \rightharpoonup_{fin} \mathit{SBuffer} & \text{(Store Buffer Pool)} \\
\mu \in \mathit{MSt} \stackrel{\text{def}}{=} \mathit{Heap} \times \mathit{SPool} + \{\lightning\} & \text{(Memory State)} \\
\mathit{PSt} \stackrel{\text{def}}{=} \{(\mu, T) \in \mathit{MSt} \times \mathit{TPool} \mid & \text{(Program State)} \\
\qquad \forall h \in \mathit{Heap}, U \in \mathit{SPool}.\ \mu = (h, U) \Rightarrow \mathrm{dom}\, U = \mathrm{dom}\, T &
\end{array}
$$

Figure 2: Semantic domains

Note that the action semantics uses two helper functions: lookup and flush. These are used to determine the result of reading a field from a thread's perspective, and to update the state of the heap with a contents of the buffer. These functions are defined in Figure 5.

$$\frac{}{(t, E[\textsf{let x} = \textsf{v in e}]) \xrightarrow{\varepsilon} \{(t, E[e[v/x]])\}}\ \textsc{Let}$$

$$\frac{}{(t, E[\textsf{if true then } \mathsf{e}_1 \textsf{ else } \mathsf{e}_2]) \xrightarrow{\varepsilon} \{(t, E[e_1])\}}\ \textsc{If-True}$$

$$\frac{}{(t, E[\textsf{if false then } \mathsf{e}_1 \textsf{ else } \mathsf{e}_2]) \xrightarrow{\varepsilon} \{(t, E[e_2])\}}\ \textsc{If-False}$$

$$\frac{}{(t, E[\textsf{CAS}(\mathsf{o.f}, \mathsf{v_o}, \mathsf{v_n})]) \xrightarrow{cas(t,o,f,v_o,v_n,r)} \{(t, E[r])\}}\ \text{CAS}$$

$$\frac{\textsf{ctorBody}(\mathsf{C}) = (\mathsf{C}(\overline{\mathsf{T}\ x}) = e)}{(t, E[\textsf{new C}(\bar{\mathsf{v}})]) \xrightarrow{new(t,C,o)} \{(t, E[e[o, \bar{v}/\textsf{this}, \bar{x}]])\}}\ \textsc{New}$$

$$\frac{}{(t, E[\mathsf{o.f}]) \xrightarrow{read(t,o,f,v)} \{(t, E[v])\}}\ \textsc{Read}$$

$$\frac{}{(t, E[\mathsf{o.f} := \mathsf{v}]) \xrightarrow{write(t,o,f,v)} \{(t, E[\textsf{()}])\}}\ \textsc{Write}$$

$$\frac{\textsf{body}(C, m) = (\mathsf{T}\ \mathsf{m}(\overline{\mathsf{T}\ x}) = e)}{(t, E[\mathsf{o.m}(\overline{\mathsf{v}})]) \xrightarrow{type(t,o,C)} \{(t, E[e[o, \overline{v}/\textsf{this}, \overline{x}]])\}}\ \textsc{Call}$$

$$\frac{\textsf{body}(C, m) = (\textsf{unit } m() = e) \qquad t \neq t'}{(t, E[\textsf{fork}(\mathsf{o.m})]) \xrightarrow{fork(t,o,C,t')} \{(t, E[\textsf{()}]), (t', e[o/\textsf{this}])\}}\ \textsc{Fork}$$

$$\frac{}{(t, e) \xrightarrow{flush(t)} \{(t, e)\}}\ \textsc{Flush}$$

Figure 3: Thread-local semantics — the non-fault cases.

Finally, the complete semantics proceeds by reducing one of the threads using the thread-local semantics, then interpreting the resulting action with the action semantics, and reducing to a memory state in the resultant set, as in Figure 4. Note how in some cases, notably read, this might require "guessing" the return value, and checking that the guess was right using the action semantics.

$$\frac{t \in \operatorname{dom} T \qquad (t, T(t)) \xrightarrow{a} T' \qquad \mu' \in [\![a]\!](\mu)}{(\mu, T) \to (\mu', (T - t) + T')}\ \textsc{Step}$$

Figure 4: Single step program evaluation

$$\text{lookup}(-.-,-,-) : \mathit{OId} \times \mathit{FName} \times \mathit{SBuffer} \times \mathit{Heap} \rightharpoonup \mathit{Val}$$

$$\text{lookup}(o.f,\alpha,h) \stackrel{\text{def}}{=} \begin{cases} \bot & \text{when } (o,f) \notin \text{dom}(h) \\ v & \text{when } h(o,f) = v \wedge \forall v' \in \mathit{Val}.\ (o,f,v') \notin \alpha \\ & \quad \text{or } \exists \beta,\gamma \in \mathit{SBuffer}.\ \alpha = \beta \cdot (o,f,v) \cdot \gamma \wedge \\ & \quad\quad \forall v' \in \mathit{Val}.\ (o,f,v') \notin \gamma \end{cases}$$

$$\text{flush}(-,-) : \mathit{Heap} \times \mathit{SBuffer} \to \mathit{Heap}$$

$$\text{flush}(h,\varepsilon) \stackrel{\text{def}}{=} h$$

$$\text{flush}(h,(o,f,v)\cdot\alpha) \stackrel{\text{def}}{=} \begin{cases} \text{flush}(h[(o,f) \mapsto v],\alpha) & \text{when } (o,f) \in \text{dom}(h) \\ \text{flush}(h,\alpha) & \text{otherwise} \end{cases}$$

Figure 5: Generic mathematical functions used throughout the report.

Finally, we show one of the simple lemmas about operational semantics, that we need in the process of building the logic. The decomposition lemma states that the only terminal program state that is not a fault is the one, in which all the store buffers are flushed, and all the threads have computed their respective values.

Lemma 1 (Decompose)**.** *For any consistent, non-error program state* $((h,U),T)$ *one of the following holds:*

- *For any thread* $t \in \text{dom}(T)$ *there exists a value* $v \in \mathit{Val}$ *such that* $T(t) = v$ *and* $U(t) = \varepsilon$
- *There exists a (possibly error) state* (μ',T') *such that* $(\mu,T) \to (\mu',T')$*.*

2 Model

The construction in this section follows closely the one presented in iCAP. As in that case, the non-recursive parts of the model are defined in the category of Sets, **Sets**, while the recursive ones — in the topos of trees, $\mathcal{S}$.

2.1 Local States, Shared States and the Treatment of Store Buffers

Local states consist of the usual, physical heap and the state transition capability map. Shared states consist of an abstract state and a labelled transition system for each shared region. For the buffered updates we use the physical store buffers defined in Section 1.

$$[\![\varepsilon]\!](h,U) = \{(h,U)\}$$

$$[\![\lightning]\!](h,U) = \{\lightning\}$$

$$[\![\mathit{read}(t,o,f,v)]\!](h,U) = \begin{cases} \{(h,U)\} & \text{if } (o,f) \in \operatorname{dom} h \text{ and } \operatorname{lookup}(o.f, U(t), h) = v \\ \emptyset & \text{if } t \notin \operatorname{dom} U \text{ or } (o,f) \in \operatorname{dom} h \text{ and } \operatorname{lookup}(o.f, U(t), h) \neq v \\ \{\lightning\} & \text{if } (o,f) \notin \operatorname{dom} h \end{cases}$$

$$[\![\mathit{write}(t,o,f,v)]\!](h,U) = \begin{cases} \{(h, U[t \mapsto U(t) \cdot (o,f,v)])\} & \text{if } (o,f) \in \operatorname{dom} h \text{ and } t \in \operatorname{dom} U \\ \{\lightning\} & \text{if } (o,f) \notin \operatorname{dom} h \\ \emptyset & \text{if } t \notin \operatorname{dom} U \end{cases}$$

$$[\![\mathit{new}(t,T,o)]\!](h,U) = \begin{cases} \{(h[(o,\overline{f}) \mapsto \mathsf{null}, o \mapsto T], U)\} & \text{if } o \notin \operatorname{dom} h \text{ and } \mathsf{fields}(T) = \overline{f} \\ \emptyset & \text{if } o \in \operatorname{dom} h \\ \{\lightning\} & \text{if } \mathsf{fields}(T) \text{ undefined} \end{cases}$$

$$[\![\mathit{cas}(t,o,f,v_o,v_n,r)]\!](h,U) = \begin{cases} \{(\operatorname{flush}(h, U(t) \cdot (o,f,v_n)), U[t \mapsto \varepsilon])\} & \text{if } (o,f) \in \operatorname{dom} h,\ r = \mathsf{true} \\ & \text{and } \operatorname{lookup}(o.f, U(t), h) = v_o \\ \{(h,U)\} & \text{if } (o,f) \in \operatorname{dom} h,\ r = \mathsf{false} \\ & \text{and } \operatorname{lookup}(o.f, U(t), h) \neq v_o \\ \{\lightning\} & \text{if } (o,f) \notin \operatorname{dom} h \\ \emptyset & \text{otherwise} \end{cases}$$

$$[\![\mathit{type}(t,o,C)]\!](h,U) = \begin{cases} \{(h,U)\} & \text{if } o \in \operatorname{dom}_t h \text{ and } h_t(o) = C \\ \emptyset & \text{if } o \in \operatorname{dom}_t h \text{ and } h_t(o) \neq C \\ \{\lightning\} & \text{if } o \notin \operatorname{dom}_t h \end{cases}$$

$$[\![\mathit{fork}(t,o,T,t')]\!](h,U) = \begin{cases} \{(h, U[t' \mapsto \varepsilon])\} & \text{if } o \in \operatorname{dom}_t h,\ h_t(o) = C \text{ and } t' \notin \operatorname{dom} U \\ \emptyset & \text{if } o \in \operatorname{dom}_t h \text{ and } t' \in \operatorname{dom} U \text{ or } h_t(o) \neq C \\ \{\lightning\} & \text{if } o \notin \operatorname{dom}_t h \end{cases}$$

$$[\![\mathit{flush}(t)]\!](h,U) = \begin{cases} \{(h[(o,f) \mapsto v], U[t \mapsto \alpha])\} & \text{if } U(t) = (o,f,v) \cdot \alpha \text{ and } (o,f) \in \operatorname{dom} h \\ \emptyset & \text{if } t \notin \operatorname{dom}(U),\ U(t) = \varepsilon, \text{ or } (o,f) \notin \operatorname{dom} h \end{cases}$$

$$[\![a]\!](\lightning) = \{\lightning\}$$

Figure 6: Action semantics

$$
\begin{aligned}
\mathit{Perm} &\stackrel{\text{def}}{=} \{q \in \mathbb{Q} \mid 0 \leq q \leq 1\} && \text{(Permissions)}\\
\mathit{Perm}^{+} &\stackrel{\text{def}}{=} \{q \in \mathit{Perm} \mid 0 < q\} && \text{(Non-zero permissions)}\\
\mathit{Cap} &\stackrel{\text{def}}{=} \{f \in (\mathit{RId} \times \mathit{AId}) \to \mathit{Perm} \mid \exists R \subseteq_{\mathit{fin}} \mathit{RId}. && \text{(Capabilities)}\\
&\qquad \forall r \in \mathit{RId} \setminus R.\ \forall \alpha \in \mathit{AId}.\ f(r,\alpha) = 0\}\\
l \in \mathit{LState} &\stackrel{\text{def}}{=} \mathit{Heap} \times \mathit{Cap} && \text{(Local States)}\\
\mathit{LTS} &\stackrel{\text{def}}{=} \mathit{AId} \to \mathcal{P}(\mathit{SId} \times \mathit{SId}) && \text{(Transition Systems)}\\
s \in \mathit{SState} &\stackrel{\text{def}}{=} \mathit{RId} \rightharpoonup_{\mathit{fin}} (\mathit{SId} \times \mathit{LTS}) && \text{(Shared States)}\\
\mathit{AState} &\stackrel{\text{def}}{=} \mathit{LState} \times \mathit{SState} \times \mathit{SPool}
\end{aligned}
$$

where PVal denotes the least set such that

$$\mathit{PVal} \cong \mathit{Val} + \mathit{Strings} + (\mathit{PVal} \times \mathit{PVal}).$$

We write $l.h$ and $l.c$ to refer to the appropriate components of the local state l, while $s(r).s$ and $s(r).p$ to refer to the state identifier and transition system components of region r in a shared state s.

Local State Composition

$$
\begin{aligned}
\bullet_{\mathit{LState}} = \{(l_1, l_2, l_r) \mid\ & \mathrm{dom}(l_1.h) \cap \mathrm{dom}(l_2.h) = \emptyset \wedge (\forall t \in \Delta(\mathit{RId} \times \mathit{AId}).\ l_1.c(t) + l_2.c(t) \leq 1)\\
& \wedge\ l_r.h = l_1.h \cup l_2.h \wedge\ \ l_r.c = l_1.c + l_2.c\},
\end{aligned}
$$

where $+$ on finite maps denotes pointwise addition.

The relation is functional, so it induces a partial function

$$\bullet_{\mathit{LState}} : \Delta(\mathit{LState}) \times \Delta(\mathit{LState}) \rightharpoonup \Delta(\mathit{LState}) \in \mathcal{S}$$

Lemma 2. $\forall l_1, l_2, l_3 \in \Delta(\mathit{LState}).\ \triangleright (l_1 = l_2 \bullet_{\mathit{LState}} l_3) \Rightarrow (l_1 = l_2 \bullet_{\mathit{LState}} l_3 \vee \triangleright \bot)$

Proof. Using the theory of upwards-closure from the iCAP technical report [2] □

Action Allowed

$$
\begin{aligned}
\mathrm{act}_\circ &: \mathit{LState} \times \mathit{RId} \to \mathcal{P}(\mathit{AId}) \in \mathbf{Sets}\\
\mathrm{act}_\circ(l, r) &\stackrel{\text{def}}{=} \{\alpha \in \mathit{AId} \mid l.c(r, \alpha) < 1\}
\end{aligned}
$$

Update Allowed

$$
\begin{aligned}
\mathrm{upd}_\circ &: \mathit{LState} \times \mathit{RId} \times \mathit{LTS} \to \mathcal{P}(\mathit{SId} \times \mathit{SId}) \in \mathbf{Sets}\\
\mathrm{upd}_\circ(l, r, p) &\stackrel{\text{def}}{=} \{(s_1, s_2) \in \mathit{SId} \times \mathit{SId} \mid \exists \alpha \in \mathrm{act}_\circ(l, r).\ (s_1, s_2) \in p(\alpha)\}
\end{aligned}
$$

2.2 Interference Relation

In our setup, interference is more complicated then in plain iCAP, due to the existance of store buffered updates. While in iCAP the interference concerned only the shared regions and transitions they are allowed to make, we also have to consider the three kinds of actions the environment can take with respect to the buffers: the allocation of a new store buffer, a buffered update written into an existing one, or a flushing of an update from store buffer to the heap. The interference is thus defined as follows.

$$
\begin{aligned}
(l, U_1)\ R_n\ (l, U_2) \text{ iff } & \mathrm{dom}(U_1) \subseteq \mathrm{dom}(U_2) \wedge \forall t \in \mathrm{dom}(U_2) \setminus \mathrm{dom}(U_1).\ U_2(t) = \varepsilon \wedge \\
& \forall t \in \mathrm{dom}(U_1).\ U_2(t) = U_1(t) \\
(l, U_1)\ R_w\ (l, U_2) \text{ iff } & \mathrm{dom}(U_1) = \mathrm{dom}(U_2) \wedge \\
& \exists t \in \mathrm{dom}(U_2), o \in \mathit{OId}, f \in \mathit{FName}, v \in \mathit{Val}.\ (o, f) \notin \mathrm{dom}(l.h) \wedge \\
& U_2(t) = U_1(t) \cdot (o, f, v) \wedge \forall t' \in \mathrm{dom}(U_2).\ t \neq t' \Rightarrow U_1(t) = U_2(t) \\
(l_1, U_1)\ R_f\ (l_2, U_2) \text{ iff } & \mathrm{dom}(U_1) = \mathrm{dom}(U_2) \wedge \\
& \exists t \in \mathrm{dom}(U_1), o \in \mathit{OId}, f \in \mathit{FName}, v \in \mathit{Val}.\ U_1(t) = (o, f, v) \cdot U_2(t) \\
& l_2 = \mathrm{flush}(l_1, (o, f, v)) \wedge \forall t' \in \mathrm{dom}(U_2).\ t \neq t' \Rightarrow U_1(t) = U_2(t) \\
R_{sb} \stackrel{\text{def}}{=} & (R_n \cup R_w \cup R_f)^* \\
(l_1, s_1)\ R_r^A\ (l_2, s_2) \text{ iff } & U_1 = U_2 \wedge l_1 \leq l_2 \wedge \mathrm{dom}(s_1) \subseteq \mathrm{dom}(s_2) \wedge \\
& \forall r \in \mathrm{dom}(s_1).\ (s_1(r).s, s_2(r).s) \in (\bigcup_{\alpha \in A, l_1.c(r,\alpha) < 1} S(r).t(\alpha))^* \\
(l_1, s_1, U_1)\ R'_A (l_2, s_2, U_2) \text{ iff } & ((l_1, s_1)\ R_r^A\ (l_2, s_2) \wedge U_1 = U_2) \vee \\
& ((l_1, U_1)\ R_{sb}\ (l_2, U_2) \wedge (s_1 \leq s_2)) \\
R_A \stackrel{\text{def}}{=} & (R'_A)^*
\end{aligned}
$$

In the above R^* denotes the reflexive, transitive closure of R. We use R as a shorthand for R_{AId}.

2.3 Region Interpretations

We follow the iCAP treatment to give the concrete interpretation to the abstract shared states in the shared regions — and thus define the type of region interpretations as a guarded recursive type in the topos of trees, $\mathcal{S}$. We take RIntr to be an object in $\mathcal{S}$ that satisfies the isomorphism

$$i : \mathit{RIntr} \cong \blacktriangleright((\Delta(\mathit{SId}) \times (\Delta(\mathit{RId}) \rightharpoonup_{fin} \mathit{RIntr})) \rightarrow_{mon} \mathcal{P}^{\uparrow}(\Delta(\mathit{AState}))),$$

where $\Delta(\mathit{AState})$ is upwards-closed with respect to the interference relation R, and the ordering on action models, $\mathit{AMod} \stackrel{\text{def}}{=} \Delta(\mathit{RId}) \rightharpoonup_{fin} \mathit{RIntr}$, is given as

$$\zeta_1 \leq \zeta_2 \stackrel{\text{def}}{=} \mathrm{dom}(\zeta_1) \subseteq \mathrm{dom}(\zeta_2) \wedge \forall r \in \mathrm{dom}(\zeta_1).\ \zeta_1(r) = \zeta_2(r).$$

The ordering on $\Delta(SId)$ is discrete.

Now we can define the maps between $\Delta(SId) \times AMod \to_{mon} \mathcal{P}^{\uparrow}(\Delta(AState))$ and $RIntr$ as follows:

$$\begin{aligned}
&lam : (\Delta(SId) \times AMod \to_{mon} \mathcal{P}^{\uparrow}(\Delta(AState))) \to RIntr \\
&lam \stackrel{\text{def}}{=} i^{-1} \circ next \\
&app : RIntr \to (\Delta(SId) \times AMod \to_{mon} \mathcal{P}^{\uparrow}(\Delta(AState))) \\
&app \stackrel{\text{def}}{=} \lambda x : RIntr.\ \lambda(s,\zeta) : \Delta(SId) \times (AMod).\ \lambda a : \Delta(AState). \\
&\qquad succ(J(J(i(x))(next(s), next(\zeta)))(next(a)))
\end{aligned}$$

Lemma 3. $app \circ lam = \triangleright$

2.4 Instrumented States

Instrumented states consist of a concrete local state and store buffer pool, an abstract state transition system for each shared region, and concrete interpretation of each abstract shared state:

$$\mathcal{M} \stackrel{\text{def}}{=} \Delta(LState) \times \Delta(SState) \times \Delta(SPool) \times AMod.$$

We use $m.l$, $m.s$, $m.U$ and $m.a$ to refer to the appropriate components of the instrumented state $m : \mathcal{M}$.

We also define a subset of $\mathcal{M}$ that omits the store buffers, which we will sometimes use:

$$\mathcal{H} \stackrel{\text{def}}{=} \Delta(LState) \times \Delta(SState) \times AMod.$$

We use analogous projections whenever needed.

Propositions We define two spaces of propositions, $Prop_{TSO}$, which denotes the general assertions, and $Prop_{SC}$, for assertions that do not mention the store buffers.

$$\begin{aligned}
Prop_{TSO} &\stackrel{\text{def}}{=} \mathcal{P}^{\uparrow}(\mathcal{M}) \\
Prop_{SC} &\stackrel{\text{def}}{=} \mathcal{P}^{\uparrow}(\mathcal{H}),
\end{aligned}$$

where the orderings are pointwise extension orderings.

We define an interpretation of $Prop_{SC}$ in $Prop_{TSO}$ as the propositions that have a "locally flushed" subset:

$$\begin{aligned}
\text{lfd} &\subseteq LState \times SPool \in \mathbf{Sets} \\
\text{lfd}(l, U) &\text{ iff } \forall t \in TId.\ \forall (o,f) \in \text{dom}(l).\ \forall v \in Val.\ (o,f,v) \notin U(t) \\
\ulcorner - \urcorner &: Prop_{SC} \to Prop_{TSO} \in \mathcal{S} \\
\ulcorner p \urcorner &\stackrel{\text{def}}{=} \{(l,s,U,\zeta) \in \mathcal{M} \mid \exists l' \leq l.\ (l',s,\zeta) \in p \wedge \text{lfd}(l',U)\}
\end{aligned}$$

Interpreted in this way $Prop_{SC}$ defines a subset of well-behaved propositions that do not depend on the state of the buffers.

Erasure

$$\lfloor (s,\zeta) \rfloor_r \stackrel{\text{def}}{=} \{(l,U) \in \mathit{LState} \times \mathit{SPool} \mid (l,s,U) \in \mathit{app}(\zeta(r))(s(r).s,\zeta)\}$$

$$\begin{aligned}\lfloor (l,s,U,\zeta) \rfloor_A \stackrel{\text{def}}{=} \{(h,U) \in \mathit{Heap} \times \mathit{SPool} \mid \\ \exists l' \in \mathit{LState}, sr \in \mathrm{dom}(s) \cap A \to \mathit{LState}. \\ h = l'.h \wedge l' = l \bullet \Pi_{r \in \mathrm{dom}(s) \cap A} sr(r) \wedge \\ \forall r \in \mathrm{dom}(s) \cap A.\ (sr(r),U) \in \lfloor (s,\zeta) \rfloor_r\}\end{aligned}$$

As usual, we use $\lfloor m \rfloor$ as a shorthand for $\lfloor m \rfloor_{\mathit{RId}}$.

Composition

$$\bullet_{\mathcal{M}} \stackrel{\text{def}}{=} \bullet_{\mathit{LState}} \times \bullet_{=} \times \bullet_{=} \times \bullet_{=}$$

2.5 Logical Connectives

Given below are the definitions for connectives in Prop_{TSO}. The definitions for Prop_{SC} are analogous.

$$\begin{aligned}\mathit{emp} &\stackrel{\text{def}}{=} \mathcal{M} \\ p * q &\stackrel{\text{def}}{=} \{m \in \mathcal{M} \mid \exists m_1, m_2 \in \mathcal{M}.\ m = m_1 \bullet m_2 \wedge m_1 \in p \wedge m_2 \in q\} \\ p \Rightarrow q &\stackrel{\text{def}}{=} \{m \in \mathcal{M} \mid \forall m' \geq m.\ m' \in p \Rightarrow m' \in q\} \\ \exists_\tau(p) &\stackrel{\text{def}}{=} \{m \in \mathcal{M} \mid \exists x : \tau.\ m \in p(x)\} \\ \forall_\tau(p) &\stackrel{\text{def}}{=} \{m \in \mathcal{M} \mid \forall x : \tau.\ m \in p(x)\}\end{aligned}$$

Lemma 4. *Let X be a total and inhabited object in $\mathcal{S}$. Then*

$$\forall p \in X \to \mathit{Prop}_{TSO}.\ \triangleright \exists_X(p) \iff \exists_X(\lambda x.\ \triangleright p(x))$$

Proof. Let $m \in \mathcal{M}$ such that $m \in \triangleright \exists_X(p)$. Then $\triangleright(m \in \exists_X(p))$ and thus

$$\triangleright(\exists x : X.\ m \in p(x)).$$

By totality and inhabitance, $\exists x : X.\ \triangleright\,(m \in p(x))$ and thus $m \in \exists_X(\lambda x.\ \triangleright\, p(x))$.

Likewise, let $m \in \mathcal{M}$ such that $m \in \exists_X(\lambda x.\ \triangleright\, p(x))$ then $\exists x : X.\ \triangleright\,(m \in p(x))$ and thus $\triangleright(\exists x : X.\ m \in p(x))$. □

Properties of the Embedding In order to use the $\ulcorner - \urcorner$ embedding to define the two separation logics, we need it to satisfy certain properties. These are formalized in the next few lemmas.

Lemma 5. *Embedding preserves the entailment relation, i.e., for any $p, q \in \mathit{Prop}_{SC}$,*

$$(p \subseteq q) \iff (\ulcorner p \urcorner \subseteq \ulcorner q \urcorner).$$

Proof (⇒). Assume $p \subseteq q$, and take any $(l, s, U, \zeta) \in \ulcorner p \urcorner$. By definition, this gives us $l' \leq l$, such that $(l', s, \zeta) \in p$ and $\mathrm{lfd}(l', U)$. By assumption, we have $(l', s, \zeta) \in q$, so we can take l' as a witness, and the properties hold trivially. □

Proof (⇐). Assume $\ulcorner p \urcorner \subseteq \ulcorner q \urcorner$, and take $(l, s, \zeta) \in p$. This means $(l, s, \emptyset, \zeta) \in \ulcorner p \urcorner$, since $\mathrm{lfd}(l, \emptyset)$ holds trivially, and so we get $(l, s, \emptyset, \zeta) \in \ulcorner q \urcorner$. From this, we obtain $l' \leq l$ such that $(l', s, \zeta) \in q$, but since q is upwards closed, $(l, s, \zeta) \in q$ holds. □

Lemma 6. *For any $p \in \mathit{Prop}_{TSO}$ and any $m \in p$ there exists a local state $l' \leq m.l$ such that $\mathrm{lfd}(l', m.U)$ and l' is the maximal state with this property (wrt the extension ordering).*

Proof. Take any $p \in \mathit{Prop}_{TSO}$ and any $(l, s, U, \zeta) \in p$. Let s be the set of all the locations in the store-buffer pool U: formally,

$$(o, f) \in s \text{ iff } \exists t \in \mathrm{dom}(U).\ \exists v \in \mathit{Val}.\ (o, f, v) \in U(t).$$

Take the state $l' = l\ |_{\mathrm{dom}(l) \setminus s}$ as the witness. Both $l' \leq l$ and $\mathrm{lfd}(l', U)$ clearly hold, so it suffices to show that this is the biggest state with this property. To this end, take any $l'' \leq l'$ such that $\mathrm{lfd}(l'', U)$. We need to show that $l'' \leq l'$. To this end, take any $(o, f) \in \mathrm{dom}(l'')$. Since both local states are smaller than l under extension ordering, it suffices to show that $(o, f) \in \mathrm{dom}(l')$ – the values (o, f) gets mapped to will have to agree. Clearly, we have $(o, f) \in \mathrm{dom}(l)$, since $l'' \leq l$, so it suffices to show that $(o, f) \notin s$. To this end, assume $(o, f) \in s$. This gives us $t \in \mathrm{dom}(U)$ and $v \in \mathit{Val}$ such that $(o, f, v) \in U(t)$. However, since $\mathrm{lfd}(l'', U)$, we know that $(o, f, v) \notin U(t)$, which ends the proof. □

Lemma 7. *Embedding preserves limits, i.e., for any τ and $p : \tau \to \mathit{Prop}_{SC}$,*

$$\forall_\tau(\ulcorner p \urcorner) = \ulcorner \forall_\tau(p) \urcorner.$$

Proof (⊆). Take any $(l, s, U, \zeta) \in \forall_\tau(\ulcorner p \urcorner)$. By definition of universal quantifier, this means that $\forall x : \tau.\ (l, s, U, \zeta) \in \ulcorner p(x) \urcorner$. By Lemma 6, we get l', the greatest local state such that $l' \leq m.l$ and $\mathrm{lfd}(l', U)$. Now it suffices to show that $(l', s, \zeta) \in \forall_\tau(p)$. Take any $x : \tau$. From the assumption, we get $l'' \leq l$ such that $\mathrm{lfd}(l'', U)$ and $(l'', s, \zeta) \in p(x)$. However by the universal property of l', we know that $l'' \leq l'$, and so, by upwards closure $(l', s, \zeta) \in p$. □

Proof (⊇). Take any $(l, s, U, \zeta) \in \ulcorner \forall_\tau(p) \urcorner$. By definition, this gives us $l' \leq l$ such that $\mathrm{lfd}(l', U)$ and $\forall x : \tau.\ (l', s, \zeta) \in p(x)$. Taking any $x : \tau$, and picking l' as the witness trivially finishes the proof. □

Lemma 8. *Embedding preserves colimits, i.e., for any τ and $p : \tau \to \mathit{Prop}_{SC}$,*

$$\exists_\tau(\ulcorner p \urcorner) = \ulcorner \exists_\tau(p) \urcorner.$$

Proof (⊆). Take any $(l, s, U, \zeta) \in \exists_\tau(\ulcorner p \urcorner)$. Unrolling the definitions, this gives us $x : \tau$ and $l' \leq l$ such that $\mathrm{lfd}(l', U)$ and $(l', s, \zeta) \in p(x)$. Taking l' and x as witnesses provides all the properties we need to finish the proof. □

Proof ($\supseteq$). Take any $(l, s, U, \zeta) \in \ulcorner \exists_\tau(p) \urcorner$. Unrolling the definitions gives us $l' \leq l$ such that $\mathrm{lfd}(l', U)$ and $x : \tau$ such that $(l', s, \zeta) \in p(x)$. Taking x and l' as witnesses provides all the needed properties. □

Lemma 9. *Embedding preserves separating conjunctions, i.e., for any $p, q \in Prop_{SC}$,*

$$\ulcorner p \urcorner * \ulcorner q \urcorner = \ulcorner p * q \urcorner.$$

Proof ($\subseteq$). Take any $(l, s, U, \zeta) \in \ulcorner p \urcorner * \ulcorner q \urcorner$. This means there exist $l_1, l_2 \in \Delta(LState)$ such that $l_1 \bullet l_2 = l$, $(l_1, s, U, \zeta) \in \ulcorner p \urcorner$ and $(l_2, s, U, \zeta) \in \ulcorner q \urcorner$. By definition of the embedding, this gets us $l_1' \leq l_1$ and $l_2' \leq l_2$ such that $(l_1', s, \zeta) \in p$, $(l_2', s, \zeta) \in q$, $\mathrm{lfd}(l_1', U)$ and $\mathrm{lfd}(l_2', U)$. Pick $l_1' \bullet l_2' \leq l$ as a witness. Obviously, $\mathrm{lfd}(l_1' \bullet l_2', U)$, and we can split the state as l_1' and l_2' to show that $(l_1', s, \zeta) \in p$ and $(l_2', s, \zeta) \in q$. □

Proof ($\supseteq$). Take any $(l, s, U, \zeta) \in \ulcorner p * q \urcorner$. This means that there exists $l' \leq l$, l_1 and l_2 such that $\mathrm{lfd}(l', U)$, $l' = l_1 \bullet l_2$, $(l_1, s, \zeta) \in p$ and $(l_2, s, \zeta) \in q$. Pick as witnesses l_1 and $l'' = l \mid_{\mathrm{dom}(l) \setminus \mathrm{dom}(l_1)}$. These states compose to give l, so we need to show $(l_1, s, U, \zeta) \in \ulcorner p \urcorner$ and $(l'', s, U, \zeta) \in \ulcorner q \urcorner$. For the first of these, pick l_1 as the witness. Since $l_1 \leq l'$, we have $\mathrm{lfd}(l_1, U)$, and we have $(l_1, s, \zeta) \in p$ by assumption. For the second goal, pick l_2 as the witness. Since l_2 is compatible with l_1, we get $l_2 \leq l''$, and since $l_2 \leq l'$ — $\mathrm{lfd}(l_2, U)$. By assumption we also have $(l_2, s, \zeta) \in q$. □

Validity

$$\mathrm{valid}(p) \stackrel{\text{def}}{=} \forall m \in \mathcal{M}.\ m \in p$$

Region Assertion and Interpretation

$$\begin{aligned} region &: \mathcal{P}(\Delta(SId)) \times \Delta(LTS) \times \Delta(RId) \to Prop_{SC} \in \mathcal{S} \\ region(X, p, r) &\stackrel{\text{def}}{=} \{(l, s, \zeta) \in \mathcal{H} \mid s(r).s \in X \wedge s(r).p = p\} \\ rintr &: (\Delta(SId) \to Prop_{TSO}) \times \Delta(RId) \to Prop_{SC} \in \mathcal{S} \\ rintr(I, r) &\stackrel{\text{def}}{=} \{(l, s, \zeta) \in \mathcal{H} \mid r \in \mathrm{dom}(\zeta) \wedge \forall x \in \Delta(SId), \zeta' \geq \zeta. \\ &\qquad app(\zeta(r))(x, \zeta') = \triangleright(\lambda(l, s, U).I(x)(l, s, U, \zeta'))\} \end{aligned}$$

Note that although technically these assertions are in $Prop_{SC}$, we will often omit the $\ulcorner - \urcorner$ intepretation when using them, since they do not depend at all on the local state, so one can always pick the empty local state as the appropriate witness.

Action Permission

$$[\alpha]^r_\pi \stackrel{\text{def}}{=} \{(l, s, \zeta) \in \mathcal{H} \mid \pi \leq l.c(r, \alpha)\}$$

2.6 Stability

$$\underset{A}{\text{stable}}(p) \stackrel{\text{def}}{=} R_A(p) \subseteq p$$
$$\text{bstable}(p) \stackrel{\text{def}}{=} R_{sb}(p) \subseteq p$$
$$\underset{A}{\text{rstable}}(p) \stackrel{\text{def}}{=} R_r^A(p) \subseteq p$$

As usual, we use stable as a shorthand for stable_{RId}

Lemma 10. $\forall p \in \mathit{Prop}_{TSO}.\ \text{stable}(p) \iff \text{bstable}(p) \wedge \mathit{rstable}(p)$

Lemma 11. $R(\mathit{emp}) \subseteq \mathit{emp} \wedge \forall p, q \in \mathit{Prop}_{TSO}.\ R(p * q) \subseteq R(p) * R(q)$

Lemma 12. $\forall A \in \mathcal{P}(\Delta(\mathit{RId})), p \in \mathit{Prop}_{TSO}.\ \text{stable}_A(p) \Rightarrow \text{stable}_A(\triangleright p)$

Proof. Assume $m_1 \in \triangleright p$ and $m_1\ R_A\ m_2$. By assumption, $R_A(p) \subseteq p$, and thus, by monotonicity of $\triangleright$,

$$\triangleright(\forall m_1, m_2 \in \mathcal{M}.\ m_1 \in p \wedge m_1\ R_A\ m_2 \Rightarrow m_2 \in p).$$

Thus, since $\mathcal{M}$ is total and inhabited,

$$\forall m_1, m_2 \in \mathcal{M}.\ m_1 \in \triangleright p \wedge \triangleright(m_1\ R_A\ m_2) \Rightarrow m_2 \in \triangleright p,$$

and so $m_2 \in \triangleright p$. □

2.7 View Shift

$$p \sqsubseteq_A q \stackrel{\text{def}}{=} \forall r \in \mathit{Prop}_{TSO}.\ \underset{A}{\text{stable}}(r) \Rightarrow \lfloor p * r \rfloor_A \subseteq \lfloor q * r \rfloor_A$$

Again, we write $\sqsubseteq$ as a shorthand for $\sqsubseteq_{RId}$.

Lemma 13. $\forall p, q, r \in \mathit{Prop}_{TSO}.\ \text{stable}(r) \wedge p \sqsubseteq_A q \Rightarrow p * r \sqsubseteq_A q * r$

Lemma 14. $\forall p_1, p_2, q_1, q_2 \in \mathit{Prop}_{TSO}.\ p_1 \sqsubseteq q_1 \wedge p_2 \sqsubseteq q_2 \Rightarrow p_1 * p_2 \sqsubseteq q_1 * q_2$

2.8 Atomic Satisfaction

$$\begin{aligned} a\ \text{sat}_A\ \{p\}\,\{q\} \stackrel{\text{def}}{=} \forall & r \in \mathit{Prop}_{TSO}, m \in \mathcal{M}, h \in \mathit{Heap}, U \in \mathit{SPool}. \\ & m \in p * \triangleright r \wedge (h, U) \in \lfloor m \rfloor_A \wedge \underset{A}{\text{stable}}(r) \Rightarrow \\ & (\triangleright\, \lightning \notin [\![a]\!](h, U)) \wedge \\ & \forall (h', U') \in [\![a]\!](h, U).\ \exists m' \in \mathcal{M}.\ \triangleright\,(m' \in q * r \wedge (h', U') \in \lfloor m' \rfloor_A) \end{aligned}$$

We write a sat $\{p\}\,\{q\}$ for $a\ \text{sat}_{RId}\ \{p\}\,\{q\}$, as usual.

Lemma 15.

$$\forall A \in \mathcal{P}(\Delta(RId)).\ \forall a \in \Delta(Act).\ \forall p, q \in \mathit{Prop}_{TSO}.\ \triangleright \bot \Rightarrow a \text{ sat}_A\ \{p\}\,\{q\}$$

Proof. Follows easily as $\triangleright\bot$ implies $\triangleright\lightning \notin [\![a]\!](h, U)$ and $\triangleright(m' \in q * r \wedge (h', U') \in \lfloor m' \rfloor_A)$ for any m', h' and U', and $\mathcal{M}$ is inhabited. □

Lemma 16 (Atomic-Shift)**.**

$$\begin{aligned}&\forall A \in \mathcal{P}(\Delta(RId)), p_1, p_2, q_1, q_2 \in \mathit{Prop}_{TSO}.\\ &\quad p_1 \sqsubseteq_A p_2 \wedge a \text{ sat}_A\ \{p_2\}\,\{q_2\} \wedge \triangleright(q_2 \sqsubseteq_A q_1) \Rightarrow a \text{ sat}_A\ \{p_1\}\,\{q_1\}\end{aligned}$$

Proof. Assume $r \in \mathit{Prop}_{TSO}$, $m \in \mathcal{M}$, $h, h' \in \mathit{Heap}$, and $U, U' \in \mathit{SPool}$ such that

$$\underset{A}{\text{stable}}(r) \qquad m \in p_1 * \triangleright r \qquad (h, U) \in \lfloor m \rfloor_A \qquad (h', U') \in [\![a]\!](h, U).$$

By Lemma 12 it follows that $\text{stable}_A(\triangleright r)$ and thus $(h, U) \in \lfloor p * \triangleright r \rfloor_A \subseteq \lfloor q * \triangleright r \rfloor_A$. Hence, there exists $m' \in \mathcal{M}$ such that $\triangleright(m' \in q_2 * r \wedge (h', U') \in \lfloor m' \rfloor_A)$, and so

$$\triangleright(m' \in \lfloor q_2 * r \rfloor_A \subseteq \lfloor q_1 * r \rfloor_A),$$

which ends the proof. □

Lemma 17 (Atomic-Frame)**.**

$$\begin{aligned}&\forall A \in \mathcal{P}(\Delta(RId)), p, q, r \in \mathit{Prop}_{TSO}.\\ &\quad \underset{A}{\text{stable}}(r) \wedge a \text{ sat}_A\ \{p\}\,\{q\} \Rightarrow a \text{ sat}_A\ \{p * \triangleright r\}\,\{q * r\}\end{aligned}$$

Proof. Assume $r' \in \mathit{Prop}_{TSO}$, $m \in \mathcal{M}$, $h, h' \in \mathit{Heap}$ and $U, U' \in \mathit{SPool}$ such that

$$\underset{A}{\text{stable}}(r') \qquad m \in p * \triangleright r * \triangleright r' \qquad (h, U) \in \lfloor m \rfloor_A \qquad (h', U') \in [\![a]\!](h, U).$$

This means that $m \in p * \triangleright(r * r')$. Since $\text{stable}_A(r * r')$, from definition of atomic satisfiability it follows that there exists an $m' \in \mathcal{M}$ such that

$$\triangleright(m' \in q * r * c \wedge (h', U') \in \lfloor m' \rfloor_A),$$

which ends the proof. □

Corollary 1.

$$\begin{aligned}&\forall A \in \mathcal{P}(\Delta(RId)), p, q, r \in \mathit{Prop}_{TSO}.\\ &\quad \underset{A}{\text{stable}}(r) \wedge a \text{ sat}_A\ \{p\}\,\{q\} \Rightarrow a \text{ sat}_A\ \{p * r\}\,\{q * r\}\end{aligned}$$

Lemma 18 (Id-Refl)**.** $\forall A \in \mathcal{P}(\Delta(RId)), p \in \mathit{Prop}_{TSO}.\ \varepsilon \text{ sat}_A\ \{p\}\,\{p\}$

Proof. Take a $r \in \mathit{Prop}_{TSO}$, $m \in \mathcal{M}$, $h \in \mathit{Heap}$ and $U \in \mathit{SPool}$ such that

$$\underset{A}{\mathrm{stable}}(r) \qquad m \in p * \triangleright r \qquad (h, U) \in \lfloor m \rfloor_A$$

Since $[\![\varepsilon]\!](h, U) = \{(h, U)\}$, so clearly $\lightning \notin [\![\varepsilon]\!](h, U)$. Now, it suffices to exhibit m' such that $(h, U) \in \lfloor m' \rfloor_A$ and $\triangleright(m' \in p * r)$, which follow simply by monotnicity of $\triangleright$ if we choose $m' = m$. □

Lemma 19 (Flush-Refl)**.** *For any $A \in \mathcal{P}(\Delta(\mathit{RId}))$, $p \in \mathit{Prop}_{TSO}$, $t \in \mathit{TId}$,*

$$\underset{A}{\mathrm{stable}}(p) \Rightarrow \mathit{flush}(t) \text{ sat}_A \ \{p\}\,\{p\}\,.$$

Proof. Take an $r \in \mathit{Prop}_{TSO}$, $m \in \mathcal{M}$, $h \in \mathit{Heap}$ and $U \in \mathit{SPool}$ such that

$$\underset{A}{\mathrm{stable}}(r) \qquad m \in p * \triangleright r \qquad (h, U) \in \lfloor m \rfloor_A.$$

Since flush never faults, we only need to consider the postcondition branch. Take any $(h', U') \in [\![\mathit{flush}(t)]\!](h, U)$. By definition this means that there exist $(o, f) \in \mathrm{dom}(h)$ and v such that $h' = h[(o, f) \mapsto v]$, $U(t) = (o, f, v) \cdot U'(t)$ and $U(t') = U'(t')$ for any $t' \neq t$. As a witness we give $m' = (\mathrm{flush}(m.l, (o, f, v)), m.s, U', m.\zeta)$. Since $(m.l, U)\ R_f\ (m'.l, U')$ and $\mathrm{stable}_A(p * \triangleright r)$, we easily have $\triangleright(m' \in p * r)$ by monotonicity of $\triangleright$. This means we only need to show that $(h', U') \in \lfloor m' \rfloor_A$. From $(h, U) \in \lfloor m \rfloor_A$ we get l_h and sr such that $h = l_h.h$, $l_h = m.l \bullet \Pi_{r \in \mathrm{dom}(m.s) \cap A} \mathit{sr}(r)$ and $(\mathit{sr}(r), U) \in \lfloor (m.s, m.\zeta) \rfloor_r$ for any $r \in \mathrm{dom}(m.s) \cap A$. Take $l'_h = \mathrm{flush}(l_h, (o, f, v))$ and $\mathit{sr}'(r) = \mathrm{flush}(\mathit{sr}(r), (o, f, v))$ as the witnesses. Since flush commutes over composition, we get

$$\begin{aligned} \mathrm{flush}(l_h, (o, f, v)) &= \mathrm{flush}(m.l \bullet \Pi_{r \in \mathrm{dom}(m.s) \cap A} \mathit{sr}(r), (o, f, v)) \\ &= m.l' \bullet \Pi_{r \in \mathrm{dom}(m.s) \cap A} \mathit{sr}'(r). \end{aligned}$$

Since, for every $r \in \mathrm{dom}(m.s) \cap A$ we also have $(\mathit{sr}(r), U)\ R_f\ (\mathit{sr}'(r), U')$ and the erasures of shared regions are closed under interference (since they are defined by the application of an action model), we get $(\mathit{sr}'(r), U') \in \lfloor (m.s, m.\zeta) \rfloor_r$ for each $r \in \mathrm{dom}(m.s) \cap A$, which ends the proof. □

2.9 Specification Embedding

$$\mathit{spec} : \Omega \to \mathit{Prop}_{TSO} \in \mathcal{S}$$
$$\mathit{spec}(s) \stackrel{\text{def}}{=} \{m \in \mathcal{M} \mid s\}$$

Lemma 20 (Sat-Spec-Later)**.**

$$\begin{aligned} &\forall a \in \mathit{Act}, p, q \in \mathit{Prop}_{TSO}, s \in \Omega. \\ &\quad a \text{ sat } \{p * \mathit{spec}(\triangleright s)\}\,\{q\} \Rightarrow a \text{ sat } \{p * \mathit{spec}(\triangleright s)\}\,\{q * \mathit{spec}(s)\} \end{aligned}$$

Proof. Take $r \in \mathit{Prop}_{TSO}$, $m \in \mathcal{M}$, $h, h' \in \mathit{Heap}$ and $U, U' \in \mathit{SPool}$ such that

$$\underset{A}{\mathrm{stable}}(r) \qquad m \in p * \mathit{spec}(\triangleright s) * \triangleright r \qquad (h, U) \in \lfloor m \rfloor_A \qquad (h', U') \in [\![a]\!](h, U).$$

By our assumption, there exists an $m' \in \mathcal{M}$, such that $\triangleright(m' \in q * r)$ and $\triangleright(h' \in \lfloor m' \rfloor)$. Hence, $\triangleright((m' \in q * r) \wedge s)$, and so $\triangleright(m' \in q * \mathit{spec}(s) * r)$. □

Lemma 21 (Later-Star)**.** $\triangleright(p * q) = (\triangleright p) * (\triangleright q)$

Proof (⊆). Assume $m \in \triangleright(p * q)$, and so $\triangleright(m \in p * q)$, and, unfolding the definition of $*$,

$$\triangleright(\exists m_1, m_2 \in \mathcal{M}.\ m = m_1 \bullet m_2 \wedge m_1 \in p \wedge m_2 \in q).$$

Now, unfolding the definition of $\bullet$, we get

$$\triangleright(\exists l_1, l_2 \in \Delta(\mathit{LState}).\ m.l = l_1 \bullet_{\mathit{LState}} l_2 \wedge (l_1, m.s, m.U, m.a) \in p \wedge (l_2, m.s, m.U, m.a) \in q).$$

This means we can take the witnesses l_1 and l_2, such that

$$\triangleright(m.l = l_1 \bullet_{\mathit{LState}} l_2) \qquad \triangleright((l_1, m.s, m.U, m.a) \in p) \qquad \triangleright((l_2, m.s, m.U, m.a) \in q).$$

From the first of these properties, by Lemma 2 we get $m.l = l_1 \bullet_{\mathit{LState}} l_2 \vee \triangleright\bot$. We inspect the two cases.

Assume, first, that $m.l = l_1 \bullet_{\mathit{LState}} l_2$. Then, $m = (l_1, m.s, m.U, m.a) \bullet (l_2, m.s, m.U, m.a)$, and so $m \in (\triangleright p) * (\triangleright q)$.

In the other case, assume $\triangleright\bot$. Now, take the splitting of $m = m \bullet (\varepsilon, m.s, m.U, m.a)$. Since $\triangleright\bot$, $\triangleright((\varepsilon, m.s, m.U, m.a) \in q)$, and so $m \in (\triangleright p) * (\triangleright q)$. □

Proof (⊇). Follows by monotonicity of $\triangleright$. □

2.10 Thread Safety

We define safety by guarded recursion:

$$\begin{aligned} &\mathrm{safe} : \mathcal{P}((\mathit{TId} \times \mathit{Expr}) \times \mathit{Prop}_{TSO} \times (\mathit{Val} \to \mathit{Prop}_{TSO})) \in \mathcal{S} \\ &\mathrm{safe} \stackrel{\mathrm{def}}{=} \mathrm{fix}\, f.\ \widehat{\mathrm{safe}}(f), \end{aligned}$$

where

$$\begin{aligned} \widehat{\mathrm{safe}}(f)((t, e), p, q) \stackrel{\mathrm{def}}{=} &(\exists v \in \mathit{Val}.\ e = v \wedge p \sqsubseteq q(v)) \vee \\ &\forall T \in \Delta(\mathit{TPool}), a \in \Delta(\mathit{Act}).\ (t, e) \xrightarrow{a} T \Rightarrow \\ &\exists p' : \mathrm{dom}(T) \to \mathit{Prop}_{TSO}.\ t \in \mathrm{dom}(T) \wedge \\ &(\forall t \in \mathrm{dom}(T).\ \triangleright\, \mathrm{stable}(p'(t))) \wedge \\ &a \text{ sat } \{p\} \left\{ \circledast_{t \in \mathrm{dom}(T)} p'(t) \right\} \wedge \\ &\triangleright\, f((t, T(t)), p'(t), q) \wedge \\ &\triangleright\, \forall t' \in \mathrm{dom}(T - t).\ f(t', p'(t'), \lambda v.\ \top) \end{aligned}$$

Lemma 22 (Stable-Closed)**.**

$$\forall X \in \mathcal{P}(\Delta(SId)), p \in \Delta(LTS), r \in \Delta(RId).$$
$$(\forall \alpha \in AId.\ \alpha \neq \alpha_i \Rightarrow p(\alpha)(X) \subseteq X) \Rightarrow \text{stable}(region(X, p, r) * \circledast_i [\alpha_i]_1^r)$$

Lemma 23 (ViewShift-Weaken)**.**

$$\forall A, B \in \mathcal{P}(\Delta(RId)), p, q \in Prop_{TSO}.\ A \subseteq B \land p \sqsubseteq_A q \Rightarrow p \sqsubseteq_B q$$

Lemma 24 (ViewShift-Trans)**.**

$$\forall p, q, r \in Prop_{TSO}, A \in \mathcal{P}(\Delta(RId)).\ p \sqsubseteq_A q \land q \sqsubseteq_A r \Rightarrow p \sqsubseteq_A r$$

Lemma 25 (RegInterp-WF)**.**

$$\forall p \in \Delta(SId) \to Prop_{TSO}.\ \text{stable}(p) \Rightarrow$$
$$(\lambda(x, \zeta).\ \{(l, s, U) \mid (l, s, U, \zeta) \in p(x)\} \in \Delta(SId) \times AMod \to_{mon} \mathcal{P}^{\uparrow}(AState))$$

Proof (Monotonicity). Take $x \in \Delta(SId)$, $\zeta_1, \zeta_2 \in AMod$ such that $\zeta_1 \leq \zeta_2$. By upwards-closure of p, for any $l \in LState$, $s \in SState$ and $U \in SPool$ such that $(l, s, U, \zeta_1) \in p(x)$, $(l, s, U, \zeta_2) \in p(x)$ also holds. □

Proof (Upwards-closure). Take $x \in \Delta(SId)$, $\zeta \in AMod$, $l_1, l_2 \in LState$, $s_1, s_2 \in SState$ and $U_1, U_2 \in SPool$ such that $(l_1, s_1, U_1, \zeta) \in p(x)$ and $(l_1, s_1, U_1)\ R\ (l_2, s_2, U_2)$. By stability of p it follows that $(l_2, s_2, U_2, \zeta) \in p(x)$. □

Lemma 26 (Safe-Sat)**.** *For any thread $t \in \Delta(TId)$, closed expression $e \in \Delta(Expr)$, $p \in Prop_{TSO}$ and $q \in \Delta(Val) \to Prop_{TSO}$ we have*

$$(\forall a \in \Delta(Act).\ \forall v \in \Delta(Val).\ (t, e) \xrightarrow{a} \{(t, v)\} \Rightarrow a \text{ sat } \{p\}\,\{q(v)\}) \land$$
$$\text{stable}(p) \land (\forall v \in \Delta(Val).\ \text{stable}(q(v))) \land \text{atomic}(e)$$
$$\Rightarrow \text{safe}((t, e), p, q).$$

Proof. The proof proceeds by Löb induction, which we will need to handle the flushing of the store buffer that can occur before the atomic command is actually executed.

Unfolding the definition of safety, we find that, since e is atomic, we should take the "progress" case. Take any $T \in TPool$ and $a \in Act$ such that $(t, e) \xrightarrow{a} T$. Since e is atomic, we only need to consider the cases when a is a *read*, *write*, *cas* or $flush$ action. We proceed by case analysis.

First, let us consider the flushing. By definition of the semantics, we know that $T = \{(t, e)\}$. We take the map p' to be the map given by $t \mapsto p$. Thus, stability of p' follows by stability of p, atomic satisfaction of the $flush(t)$ action – by Lemma 19, and since we did not fork any threads we only need to show that $\triangleright \text{safe}((t, e), p, q)$ – which follows from our Löb inductive hypothesis.

In the other three cases, we have to consider proper reductions. In all of these cases T has the same shape: we have $T = \{(t, v)\}$ for some value v. Thus, we can take the map p'

to be given by $t \mapsto q(v)$. Then, the stability of p' follows by stability of q, and the atomic satisfaction by the lemma's assumption. Since we do not allocate any new threads, this only leaves us with showing that $\triangleright \mathrm{safe}((t, v), q(v), q)$, which holds trivially by taking the "value" branch of the safety and showing that $\triangleright(q(v) \sqsubseteq q(v))$ by monotonicity of $\triangleright$ and reflexivity of the view-shift. □

Lemma 27 (Region-Alloc)**.**

$$\begin{aligned}
&\forall F \in \mathcal{P}(\Delta(\mathit{RId})), T \in \Delta(\mathit{LTS}), X \in \mathcal{P}(\Delta(\mathit{SId})), x \in X.\\
&\forall I \in \Delta(\mathit{RId}) \to \Delta(\mathit{SId}) \to \mathit{Prop}_{\mathit{TSO}}, P \in \mathit{Prop}_{\mathit{TSO}}, A, B \in \mathcal{P}(\Delta(\mathit{AId}))\\
&\quad (\forall r \in \Delta(\mathit{RId}), s \in \Delta(\mathit{SId}).\ \mathrm{stable}(I(r)(s))) \land F \ \mathit{infinite} \land A, B \ \mathit{finite} \land\\
&\quad \mathrm{valid}(\forall n \in F.\ P * \circledast_{\alpha \in A}[\alpha]_1^n \Rightarrow \triangleright I(n)(x)) \land A \cap B = \emptyset\\
&\qquad \Rightarrow P \sqsubseteq_F \exists n \in F.\ \mathit{region}(X, T, n) * \mathit{rintr}(I(n), n) * \circledast_{\alpha \in B}[\alpha]_1^n
\end{aligned}$$

Proof. By the definition of the view shift, we take any $r \in \mathit{Prop}_{\mathit{TSO}}$, $m \in \mathcal{M}$, $h \in \mathit{Heap}$, $U \in \mathit{SPool}$ such that $m \in P * r$ and $(h, U) \in \lfloor m \rfloor_F$. Hence, there have to exist $l_1, l_2, l_c \in \mathit{LState}$, $s \in \mathit{SState}$, $\zeta \in \mathit{AMod}$ and $sr : \mathit{dom}(s) \cap F \to \mathit{LState}$ such that

$$\begin{aligned}
&h = l_c.h \qquad l_c = l_1 \bullet_{\mathit{LState}} l_2 \bullet \Pi_{r \in \mathrm{dom}(s) \cap F} sr(r) \qquad m = (l_1 \bullet_{\mathit{LState}} l_2, s, U, \zeta)\\
&\forall r \in \mathrm{dom}(s) \cap F.\ (sr(r), U) \in \lfloor (s, \zeta) \rfloor_r \qquad (l_1, s, U, \zeta) \in P \qquad (l_2, s, U, \zeta) \in r
\end{aligned}$$

Pick an $n \in U$ such that $\forall \alpha \in \mathit{AId}.\ l_c.c(n, \alpha) = 0$ and $n \notin \mathrm{dom}(s) \cup \mathrm{dom}(\zeta)$. Now, let $s' = s[n \mapsto (x, T)]$, $\zeta' = \zeta[n \mapsto \mathit{lam}(\lambda(y, \zeta).\ \{(l, s, U) \in \mathit{AState} \mid (l, s, U, \zeta) \in I(n)(y)\})]$. Note that ζ' is well-defined, by virtue of Lemma 25. We have $s \leq s'$ and $\zeta \leq \zeta'$, and so $(l_1, s', U, \zeta') \in P$ and $(l_2, s', U, \zeta') \in r$. Furthermore, $((\varepsilon, [(n, A) \mapsto 1]), s', U, \zeta') \in \circledast_{\alpha \in A}[\alpha]_1^n$, and so, by the validity assumption,

$$(l_1 \bullet_{\mathit{LState}} (\varepsilon, [(n, A) \mapsto 1]), s', U, \zeta') \in \triangleright I(n)(x).$$

Now, clearly $(\varepsilon, s', U, \zeta') \in \mathit{region}(X, T, n)$. Furthermore, for any $\zeta'' \geq \zeta'$ and $y \in \mathit{SId}$,

$$\begin{aligned}
\mathit{app}(\zeta'(n))(y, \zeta'') =& \mathit{app}(\mathit{lam}(\lambda(y, \zeta).\ \{(l, s, U) \mid (l, s, U, \zeta) \in I(n)(y)\}))(y, \zeta'')\\
=& (\triangleright(\lambda(y, \zeta).\ \{(l, s, U) \mid (l, s, U, \zeta) \in I(n)(y)\}))(y, \zeta'')\\
=& \{(l, s, U) \mid (l, s, U, \zeta'') \in \triangleright I(n)(y)\}.
\end{aligned}$$

Thus, $(\varepsilon, s', U, \zeta'') \in \mathit{rintr}(I, n)$ also holds. Hence, we have

$$(l_2 \bullet_{\mathit{LState}} (\varepsilon, [(n, B) \mapsto 1]), s', U, \zeta') \in \mathit{region}(X, T, n) * \mathit{rintr}(I, n) * (\circledast_{\alpha \in B}[\alpha]_1^n) * r.$$

Lastly, we also get $(h, U) \in \lfloor (l_2 \bullet_{\mathit{LState}} (\varepsilon, [(n, B) \mapsto 1]), s', U, \zeta') \rfloor_F$, which ends the proof. □

Lemma 28.

$$\forall A \in \mathcal{P}(\Delta(RId)), X, Y \in \mathcal{P}(\Delta(SId)), T \in \Delta(LTS), n \in \Delta(RId).$$
$$\forall I \in \Delta(SId) \to Prop_{TSO}, p, q \in Prop_{TSO}, \alpha \in \Delta(AId), \pi \in \Delta(Perm^+), f : X \to Y.$$
$$\text{stable}(p) \wedge n \in A \wedge (\forall x \in X.\ (x, f(x)) \in T(\alpha) \vee f(x) = x) \wedge$$
$$(\forall x \in X.\ p * (\triangleright I(x)) * [\alpha]^n_\pi \sqsubseteq_{A \setminus \{n\}} q * \triangleright I(f(x)))$$
$$\Rightarrow (region(X, T, n) * rintr(I, n) * p * [\alpha]^n_\pi \sqsubseteq_A region(Y, T, n) * q)$$

Proof. Take $r \in Prop_{TSO}$, $m \in \mathcal{M}$, $h \in Heap$ and $U \in SPool$ such that

$$\underset{A}{\text{stable}}(r) \qquad m \in region(X, T, n) * rintr(I, n) * p * [\alpha]^n_\pi * r \qquad (h, U) \in \lfloor m \rfloor_A.$$

This means, there exist $l_1, l_2, l_3, l_4, l_5 \in LState$, $s \in SState$, $U' \in SPool$ and $\zeta \in AMod$ such that

$$(l_1, s, U', \zeta) \in region(X, T, n) \qquad (l_2, s, U', \zeta) \in rintr(I, n)$$
$$(l_3, s, U', \zeta) \in p \qquad (l_4, s, U', \zeta) \in [\alpha]^n_\pi \qquad (l_5, s, U', \zeta) \in r$$
$$m = (l_1 \bullet_{LState} l_2 \bullet_{LState} l_3 \bullet_{LState} l_4 \bullet_{LState} l_5, s, U', \zeta).$$

Let $l = m.l$. By unfolding $(h, U) \in \lfloor m \rfloor_A$ we get that $U = U'$, and $l_c \in LState$ and $sr : \text{dom}(s) \cap A \to LState$ such that

$$h = l_c.h \qquad l_c = l \bullet_{LState} \Pi_{r \in \text{dom}(s) \cap A} sr(r) \qquad \forall r \in \text{dom}(s) \cap A.\ (sr(r), U) \in \lfloor (s, \zeta) \rfloor_r.$$

Since $(l_1, s, U, \zeta) \in region(X, T, n)$, we know $n \in \text{dom}(s)$, $s(n).s \in X$ and $s(n).p = T$ hold. Since $n \in \text{dom}(s) \cap A$, we get $sr(n) \in \lfloor (s, \zeta) \rfloor_n$ and $\pi_2(sr(n)) = U$, and so $(sr(n), s, U) \in app(\zeta(n))(s(n).s, \zeta)$. Unfolding $(l_2, s, U, \zeta) \in rintr(I, n)$, we get that

$$\forall x \in \Delta(SId), \zeta' \geq \zeta.\ app(\zeta(n))(x, \zeta') = \triangleright(\lambda(l, s, U).\ I(x)(l, s, U, \zeta')).$$

From this, instantiating with $s(n).s$ and ζ, we get that $\triangleright(I(s(n).s)(sr(n), s, U, \zeta))$. By assumption, we know that $(s(n).s, f(s(n).s)) \in T(\alpha)$ (in the case where $f(s(n).s) = s(n).s$ it suffices to use the assumed view-shift). Furthermore, since $(l_4, s, \zeta) \in [\alpha]^n_\pi$ it follows that $\pi \leq l_4.c(n, \alpha)$ and thus $l_3.c(n, \alpha) < 1$, $sr(n).c(n, \alpha) < 1$ and $l_5.c(n, \alpha) < 1$. Thus,

$$(l_3, s, U)\ R_A\ (l_3, s', U) \qquad (l_5, s, U)\ R_A\ (l_5, s', U) \qquad (sr(n), s, U)\ R_A\ (sr(n), s', U),$$

where $s' = s[n \mapsto (f(s(n).s), s(n).p)]$. Hence, by stability of p, $\triangleright I(s(n).s)$ and r, it follows that

$$(l_3, s', U, \zeta) \in p \quad (sr(n), s', U, \zeta) \in \triangleright I(s(n).s) \quad (l_4, s', U, \zeta) \in [\alpha]^n_\pi \quad (l_5, s', U, \zeta) \in r.$$

Furthermore, for every $r \in \text{dom}(s) \cap (A \setminus \{n\})$, $sr(r).c(n, \alpha) < 1$, so

$$(sr(r), s, U)\ R_A\ (sr(r), s', U),$$

and so, by stability of region interpretations,

$$\forall r \in \mathrm{dom}(s') \cap (A \setminus \{n\}).\ (sr(r), U) \in \lfloor (s', \zeta) \rfloor_r.$$

Thus, we get that

$$(l \bullet_{LState} \pi_1(sr(n)), s', U, \zeta) \in \\ region(\{f(s(n).s)\}, T, n) * (\triangleright I(s(n).s)) * [\alpha]^n_\pi * p * r * \{m \in \mathcal{M} \mid \zeta \leq m.a\}$$

and

$$(h, U) \in \lfloor (l \bullet_{LState} sr(n), s', U, \zeta) \rfloor_{A \setminus \{n\}}.$$

Now we can use the assumed view-shift, from which it follows that there exists an $m' \in \mathcal{M}$ such that

$$m' \in q * (\triangleright I(f(s(n).s))) * r * region(\{f(s(n).s)\}, T, n) * \{m \in \mathcal{M} \mid \zeta \leq m.a\} \\ (h, U) \in \lfloor m' \rfloor_{A \setminus \{n\}}.$$

Thus, there exist $l'_1, l'_2, l'_3, l'_4, l'_5 \in LState$, $s'' \in SState$ and $\zeta' \in AMod$ such that

$$m' = (l'_1 \bullet_{LState} l'_2 \bullet_{LState} l'_3 \bullet_{LState} l'_4 \bullet_{LState} l'_5, s'', U, \zeta') \\ (l'_1, s'', U, \zeta') \in q \qquad (l'_2, s'', U, \zeta') \in \triangleright I(f(s(n).s)) \qquad (l'_3, s'', U, \zeta') \in r \\ (l'_4, s'', U, \zeta') \in region(\{f(s(n).s)\}, T, n) \qquad (l'_5, s'', U, \zeta') \in \{m \in \mathcal{M} \mid \zeta \leq m.a\}.$$

Hence, $s''(n).s = f(s(n).s) = s'(n).s$. From $(h, U) \in \lfloor m' \rfloor_{A \setminus \{n\}}$, it follows that tere exist $l'_c \in LState$ and $sr' : \mathrm{dom}(s'') \cap (A \setminus \{n\}) \to LState$ such that

$$l'_c = l'_1 \bullet_{LState} l'_2 \bullet_{LState} l'_3 \bullet_{LState} l'_4 \bullet_{LState} l'_5 \bullet_{LState} \circledast_{r \in \mathrm{dom}(s'') \cap (A \setminus \{n\})} \pi_1(sr'(r)) \\ h = l'_c.h \qquad \forall r \in \mathrm{dom}(s'') \cap (A \setminus \{n\}).\ (sr'(r), U) \in \lfloor (s'', \zeta') \rfloor_r.$$

From $(l'_5, s'', U, \zeta') \in \{m \in \mathcal{M} \mid \zeta \leq m.a\}$ it follows that $\zeta \leq \zeta'$, and thus

$$\forall x \in \Delta(SId).\ app(\zeta'(n))(x, \zeta') = \triangleright(\lambda(l, s, U).\ I(x)(l, s, U, \zeta')).$$

Hence, $(l'_2, s'', U) \in app(\zeta'(n))(f(s(n).s), \zeta')$, and so $(l'_2, U) \in \lfloor (s'', \zeta') \rfloor_n$. Thus, we have $(h, U) \in \lfloor region(Y, T, n) * q \rfloor_A$, which ends the proof. □

Lemma 29.

$\forall A \in \mathcal{P}(\Delta(RId)).\ \forall X \in \mathcal{P}(\Delta(SId)).\ \forall T \in \Delta(LTS).\ \forall n \in \Delta(RId).$
$\forall I \in \Delta(SId) \to Prop_{TSO}.\ \forall p, p' \in Prop_{TSO}.\ \forall q \in \Delta(SId) \to Prop_{TSO}.\ \forall B \in \mathcal{P}(\Delta(AId)).$
$\forall f : \Delta(SId) \to \mathcal{P}(\Delta(SId)).\ \forall g : \Delta(AId) \to \Delta(Perm^+).$

$$(\forall y \in \Delta(SId).\ \mathrm{stable}(q(y))) \wedge n \in A \wedge (\forall x \in X.\ \forall y \in f(x).\ (x, y) \in T(B)^*) \wedge \\ (\forall x \in X.\ a\ \mathrm{sat}_{A \setminus \{n\}}\ \{p * (\triangleright I(x))\} \left\{ \exists y \in f(x).\ q(y) * \circledast_{\alpha \in B} [\alpha]^n_{g(\alpha)} * \triangleright I(y) \right\}) \\ \Rightarrow a\ \mathrm{sat}_A\ \{region(X, T, I, n) * p\} \left\{ \exists y \in \Delta(SId).\ region(\{y\}, T, n) * \circledast_{\alpha \in B} [\alpha]^n_{g(\alpha)} * q(y) \right\}$$

Proof. Assume $r \in \mathit{Prop}_{TSO}$, $m \in \mathcal{M}$, $h, h' \in \Delta(\mathit{Heap})$ and $U, U' \in \mathit{SPool}$ such that

$$m \in \mathit{region}(X,T,I,n) * p * \triangleright r \quad (h,U) \in \lfloor m \rfloor_A \quad (h',U') \in [\![a]\!](h,U) \quad \underset{A}{\mathrm{stable}}(r).$$

Thus, there exist $l_1, l_2, l_3, l_4 \in \mathit{LState}$, $s \in \mathit{SState}$ and $\zeta \in \mathit{AMod}$ such that

$$(l_1, s, U, \zeta) \in \mathit{region}(X,T,n) \quad (l_2, s, U, \zeta) \in \mathit{rintr}(I,n) \quad (l_3, s, U, \zeta) \in p$$
$$(l_4, s, U, \zeta) \in \triangleright r \quad m = (l_1 \bullet l_2 \bullet l_3 \bullet l_4 \bullet l_5, s, U, \zeta).$$

Let $l = l_1 \bullet l_2 \bullet l_3 \bullet l_4$. Unfolding $(h,U) \in \lfloor m \rfloor_A$ we get an $l_c \in \mathit{LState}$ and $sr : (\mathrm{dom}(s) \cap A) \to \mathit{LState}$ such that

$$h = l_c.h \quad l_c = l \bullet \circledast_{r \in (\mathrm{dom}(s) \cap A)} sr(r) \quad \forall r \in (\mathrm{dom}(s) \cap A).\ (sr(r), U) \in \lfloor (s, \zeta) \rfloor_r.$$

Since $(l_1, s, U, \zeta) \in \mathit{region}(X,T,n)$, it follows that $n \in \mathrm{dom}(s)$, $s(n).s \in X$ and $s(n).p = T$. Since we also know that $n \in A$, it follows that $(sr(n), U) \in \lfloor (s,\zeta) \rfloor_n$, and so $(sr(n), s, U) \in \mathit{app}(\zeta(n))(s(n).s, \zeta)$. Unfolding $(l_2, s, U, \zeta) \in \mathit{rintr}(I,n)$ it follows that

$$\forall x \in \Delta(\mathit{SId}).\ \forall \zeta' \geq \zeta.\ \mathit{app}(\zeta(n))(x, \zeta') = \triangleright(\lambda(l,s,U).\ I(x)(l,s,U,\zeta')).$$

Thus, in particular, $\mathit{app}(\zeta(n))(s(n).s, \zeta) = \triangleright(\lambda(l,s,U).\ I(s(n).s)(l,s,U,\zeta))$, and so

$$\triangleright I(s(n).s)(sr(n), s, U, \zeta).$$

The high-level plan is now to use the assumption with a slightly modified state to account for opening of the region, and afterwards use the state obtained from the assumption to build a final instrumented state. To this end, we instantiate the assumption with a frame $r' = r * \mathit{region}(\{s(n).s\}, T, n) * \{m \in \mathcal{M} \mid m.\zeta \geq \zeta\}$ and an instrumented state $\bar{m} = (l \bullet sr(n), s, U, \zeta)$. The erasure $(h,U) \in \lfloor \bar{m} \rfloor_{A \setminus \{n\}}$ holds easily, since $n \notin A \setminus \{n\}$ but $sr(n)$ is in local state. We also know that $(l_3, s, U, \zeta) \in p$, $(sr(n), s, U, \zeta) \in \triangleright(I(s(n).s))$ and $(l_4, s, U, \zeta) \mathit{in} \triangleright r$, so it suffices to show that

$$(l_1 \bullet l_2, s, U, \zeta) \in \triangleright(\mathit{region}(\{s(n).s\}, T, n) * \{m \in \mathcal{M} \mid m.\zeta \geq \zeta\}),$$

which holds by monotonicity of later and the definition. The frame is also stable, since we cannot change the state of region n.

This means we get a state $\bar{m}'$ such that $(h', U') \in \triangleright \lfloor \bar{m}' \rfloor_{A \setminus \{n\}}$ and

$$\bar{m}' \in \triangleright(\exists y \in \mathsf{f}(x) * q(y) * \circledast_{\alpha \in B} [\alpha]^n_{g(\alpha)} * \triangleright I(y) * r').$$

Thus, we can take a $y \in \Delta(\mathit{SId})$ such that $\triangleright(y \in f(x))$, and local states l_1', l_2', l_3' and l_4', such that

$$\bar{m}' = (l_1' \bullet l_2' \bullet l_3' \bullet l_4', s', U', \zeta')$$
$$(l_1', s', U', \zeta') \in \triangleright q(y) \quad (l_2', s', U', \zeta') \in \triangleright \circledast_{\alpha \in B} [\alpha]^n_{g(\alpha)} \quad \triangleright ((l_3', s', U', \zeta') \in \triangleright I(y))$$
$$(l_4', s', U', \zeta') \in \triangleright r \quad \triangleright (s'(n).s = s(n).s) \quad \triangleright (s'(n).p = T) \quad \triangleright (\zeta' \geq \zeta).$$

Now, we can take $m' = (l'_1 \bullet l'_2 \bullet l'_4, s'[n \mapsto (y,T)], U', \zeta')$ as the witness. We need to show that $\triangleright((h',U') \in \lfloor m' \rfloor_A)$ and $m' \in \triangleright(\exists y \in \Delta(SId).\ region(\{y\},T,n) * q(y) * \circledast_{\alpha \in B}[\alpha]^n_{g(\alpha)} * r)$. For the latter, we take y as the witness and split the local state four ways into ε, l'_1, l'_2 and l'_4. The region assertion holds by definition, and the assertion about permissions by the earlier property, since it doesn't depend on the shared state. That leaves us with r and $q(y)$. Both these assertions are stable, so it suffices to show that x and y are in the interference relation from their perspective. To show this, notice that the permissions in B suffice to transition from x to y – and these permissions are outside both l'_1 and l'_4. Thus, we can get $(l'_1, s'[n \mapsto (y,T)], U', \zeta') \in \triangleright q(y)$ and $(l'_4, s'[n \mapsto (y,T)], U', \zeta') \in \triangleright r$.

This leaves us with proving that $(h',U') \in \lfloor m' \rfloor_A$. To this end, take $\bar{l}$ and $\bar{sr} : \mathrm{dom}(s') \cap (A \setminus \{n\}) \to LState$ – the witnesses of $(h',U') \in \lfloor \bar{m'} \rfloor_{A \setminus \{n\}}$. We know that

$$h' = \bar{l}.h \quad \bar{l} = l'_1 \bullet l'_2 \bullet l'_3 \bullet l'_4 \bullet \Pi_{r \in \mathrm{dom}(s) \cap (A \setminus \{n\})} \bar{sr}(r)$$
$$\forall r \in \mathrm{dom}(s) \cap (A \setminus \{n\}).\ \triangleright((\bar{sr}(r), U') \in \lfloor (s', \zeta') \rfloor_r)$$

Take as witnesses $l' = \bar{l}$ and $sr' = \bar{sr}[n \mapsto l'_3]$. Clearly sr' has the appropriate domain, and the first two conditions hold. For the final one, take any $r \in \mathrm{dom}(s) \cap A$. We have two cases to consider: either $r = n$, or $r \in \mathrm{dom}(s) \cap (A \setminus \{n\})$. In the latter, we can use the same argument as for r and $q(y)$ to show that the update of shared state is an allowed interference for the region r. In the former, we need to show that $\triangleright(l'_3, U') \in \lfloor (s'[n \mapsto (y,T)], \zeta') \rfloor_r$. Unrolling the definition, it suffices to show that $\triangleright(l'_3, s'[n \mapsto (y,T)], U') \in app(\zeta'(r))(y,\zeta')$. However, since $\triangleright(\zeta' \geq \zeta)$, by an earlier equation that followed from region interpretation, it suffices to show that $\triangleright(l'_3, s'[n \mapsto (y,T)], U', \zeta') \in \triangleright I(y)$, which follows by the same stability argument as the other assertions. □

Lemma 30. *For any $p \in Prop_{SC}$, $(l,s,U,\zeta) \in \ulcorner p \urcorner$, $(l',U') \in R_{sb}(l,U)$, $(l',s,U',\zeta) \in \ulcorner p \urcorner$.*

Proof. To show the lemma holds, it suffices to show p is closed under each of R_n, R_w and R_f. These proofs are presented in the following paragraphs.

(R_n) Take any $(l,s,U,\zeta) \in \ulcorner p \urcorner$ and $(l',U') \in R_n(l,U)$. By definition of R_n we have that $l = l'$, $\forall t \in \mathrm{dom}(U') \setminus \mathrm{dom}(U).\ U'(t) = \varepsilon$ and $\forall t \in \mathrm{dom}(U).\ U'(t) = U(t)$. From definition of $\ulcorner - \urcorner$ we get a state $l_1 \leq l$ such that $(l_1,s,\zeta) \in p$ and

$$\forall t \in \mathrm{dom}(U).\ \forall (o,f) \in \mathrm{dom}(l_1).\ \forall v \in Val.\ (o,f,v) \notin U(t).$$

We take the same l_1 as a witness, and so we only need to show that $(o,f,v) \notin U'(t)$ for any $t \in \mathrm{dom}(U')$, $(o,f) \in \mathrm{dom}(l_1)$ and $v \in Val$. If $t \in \mathrm{dom}(U)$, the property holds, since in that case $U(t) = U(t')$ and we know it held for U. If $t \notin \mathrm{dom}(U)$, on the other hand, we know $U'(t) = \varepsilon$, so in particular $(o,f,v) \notin U'(t)$.

(R_w) Take any $(l,s,U,\zeta) \in \ulcorner p \urcorner$ and $(l',U') \in R_w(l,U)$. By definition of R_w we know that $l = l'$, $\mathrm{dom}(U) = \mathrm{dom}(U')$ and we get $t \in \mathrm{dom}(U)$, $o \in OId$, $f \in FName$, and

$v \in \mathit{Val}$ such that $(o, f) \notin \mathrm{dom}(l)$, $U'(t) = U(t) \cdot (o, f, v)$ and $U'(t') = U(t')$ for any $t' \neq t$. From the definition of $\ulcorner - \urcorner$ we get a local state $l_1 \leq l$ such that $(l_1, s, \zeta) \in p$ and

$$\forall t \in \mathrm{dom}(U).\ \forall (o, f) \in \mathrm{dom}(l_1).\ \forall v \in \mathit{Val}.\ (o, f, v) \notin U(t).$$

Again, we take l_1 as the witness, which means we only need to prove $(o', f', v') \notin U(t')$ for any $(o', f') \in \mathrm{dom}(l_1)$, $t' \in \mathrm{dom}(U)$. Clearly, it suffices to check that this holds for the newly added update, (o, f, v) in thread t, since the rest of the contents of U' is inherited from U. However, we know that $(o, f) \notin \mathrm{dom}(l)$, so clearly $(o, f) \notin \mathrm{dom}(l_1)$, so the property holds.

(R_f) Take any $(l, s, U, \zeta) \in \ulcorner p \urcorner$ and $(l', U') \in R_w(l, U)$. By definition of R_w we know that $\mathrm{dom}(U) = \mathrm{dom}(U')$ and we get $t \in \mathrm{dom}(U)$, $o \in \mathit{OId}$, $f \in \mathit{FName}$ and $v \in \mathit{Val}$ such that $l' = \mathrm{flush}(l, (o, f, v))$, $U(t) = (o, f, v) \cdot U'(t)$ and $U(t') = U'(t')$ for any $t' \neq t$. As before, from the definition of $\ulcorner - \urcorner$ we get an $l_1 \leq l$ such that $(l_1, s, \zeta) \in p$ and there are no updates to the domain of l_1. From the latter property, we know that $(o, f) \notin \mathrm{dom}(l_1)$, and so $l_1 \leq l'$. This allows us to take l_1 as the witness, and in this case both properties hold trivially, since the updates in U' are a subset of those in U. □

2.11 Thread-pool Evaluation

$$\mathit{eval} : \Delta(\mathit{MSt}) \times \Delta(\mathit{TPool}) \times \mathcal{P}(\Delta(\mathit{MSt}) \times \Delta(\mathit{TPool})) \to \Omega$$
$$\mathit{eval}(\mu, T, q) \stackrel{\mathrm{def}}{=} (\mathrm{irr}(\mu, T) \wedge (\mu, T) \in q) \ \vee (\forall T', \mu'.\ (\mu, T) \to (\mu', T') \Rightarrow \triangleright \mathit{eval}(\mu', T', q))$$

Lemma 31 (Safe-Eval)**.** *For any thead pool $(\mu, T) \in \Delta(\mathit{PSt})$ and families of assertions $p : \mathrm{dom}(T) \to \mathit{Prop}_{\mathit{TSO}}$, $q : \mathrm{dom}(T) \to \mathit{Val} \to \mathit{Prop}_{\mathit{TSO}}$, if*

$$\forall t \in \mathrm{dom}(T).\ \mathrm{stable}(p(t)) \qquad \forall t \in \mathrm{dom}(T).\ \forall v \in \mathit{Val}.\ \mathrm{stable}(q(t)(v)))$$
$$\forall t \in \mathrm{dom}(T).\ \mathrm{safe}((t, T(t)), p(t), q(t)) \qquad \mu \in \lfloor \circledast_{t \in \mathrm{dom}(T)} p(t) \rfloor$$

then

$$\mathit{eval}\left(\mu, T, \lambda(\mu', T').\ \mu' \in \lfloor \circledast_{t \in \mathrm{dom}(T)} q(t)(T'(t)) \rfloor\right).$$

Proof. By Löb induction. Since we know μ is an erasure of a state, we know there exist h and U such that $\mu = (h, U)$ and $\mathrm{dom}(U) = \mathrm{dom}(T)$. By Lemma 1, there are two cases: either we are in a terminal state, or there exist a possible reduction. We proceed by inspecting these two.

Let us first consider the terminal case. By Lemma 1, we know that for any thread $t \in \mathrm{dom}(T)$, $T(t)$ is a value, and $U(t) = \varepsilon$. Thus, $\mathrm{irr}(\mu, T)$ holds, and so we pick the left branch. We are left with showing that $(h, U) \in \lfloor \circledast_{t \in \mathrm{dom}(T)} q(t)(T(t)) \rfloor$. However, since we know that no thread can proceed, we also learn, from the safety assumption, that for any thread we have $p(t) \sqsubseteq q(t)(T(t))$. This shows us how to proceed: we should successively apply the view-shifts for each consecutive thread, while treating the separating conjunctions of the threads already shifted and these yet to be shifted as a

frame. Formally we need to weaken our assumption to say that for any thread pools T_1 and T_2 such that $T_1 \uplus T_2 = T$ if $(h, U) \in \lfloor \circledast_{t \in \mathrm{dom}(T_1)} q(t)(T_1(t)) * \circledast_{t \in \mathrm{dom}(T_2)} p(t) \rfloor$, then $(h, U) \in \lfloor \circledast_{t \in \mathrm{dom}(T)} q(t)(T(t)) \rfloor$, and proceed by induction on the size of T_2. In the base case T_2 is empty, so $T_1 = T$, and the lemma holds trivially. Otherwise, we know that $T_2 = (t, v) \uplus T_2'$ for some T_2', t and v. Thus, since all of p and q are stable, we can apply the view-shift for thread t: we know that $(h, U) \in \lfloor p(t) * \circledast_{t \in \mathrm{dom}(T_1)} q(t)(T_1(t)) * \circledast_{t \in \mathrm{dom}(T_2')} p(t) \rfloor$, so we get that $(h, U) \in \lfloor q(t)(v) * \circledast_{t \in \mathrm{dom}(T_1)} q(t)(T_1(t)) * \circledast_{t \in \mathrm{dom}(T_2')} p(t) \rfloor$. Now, to use the induction hypothesis for the smaller T_2' we only need to take $T_1' = T_1 \uplus (t, v)$. Thus, this part of the proof is finished.

Let us now consider the case where a reduction could occur. The decomposition lemma gives us a state (μ', T') such that $((h, U), T) \to (\mu', T')$. By definition of single step evaluation, this means that there is a thread $t \in \mathrm{dom}(T)$, an action $a \in Act$, and a thread pool $T' \in TPool$, such that $(t, T(t)) \xrightarrow{a} T''$, $\mu' \in [\![a]\!](\mu)$ and $T' = (T - t) \uplus T''$. We will consider the case when $a = \mathit{flush}(t)$ separately, since this can occur even if $T(t)$ is a value; all the other actions will follow from the safety of thread t.

Let us assume $a = \mathit{flush}(t)$. In this case we can take the instrumented state m that mediates the erasure and use the Lemma 19 to get a state $m' \in \triangleright \circledast_{t \in \mathrm{dom}(T)} p(t)$ such that $\mu' \in \triangleright \lfloor m' \rfloor$. The remaining obligation is easily discharged by induction hypothesis and monotonicity of $\triangleright$.

Now, let us consider the case when a is not a flush action. In this case, we know that $T(t)$ can not be a value, so from the assumption of safety of t we get $p' : \mathrm{dom}(T'') \to \mathit{Prop}_{TSO}$ and the following facts:

$$t \in \mathrm{dom}(T'') \quad \forall t' \in \mathrm{dom}(T''). \ \triangleright \mathrm{stable}(p'(t')) \quad a \text{ sat } \{p(t)\} \left\{ \circledast_{t' \in \mathrm{dom}(T'')} p'(t') \right\}$$
$$\triangleright \mathrm{safe}((t, T''(t)), p'(t), q(t)) \quad \forall t' \in \mathrm{dom}(T'' - t). \ \mathrm{safe}((t', T''(t')), p'(t'), \lambda\_.\top).$$

As in the previous case, we can now take the mediating instrumented state m and use the atomic satisfaction, taking $\circledast_{t' \in \mathrm{dom}(T - t)} p(t')$ as a frame, to obtain a state m' such that

$$\mu' \in \triangleright \lfloor m' \rfloor \qquad m' \in \triangleright \circledast_{t' \in \mathrm{dom}(T - t)} p(t) * \circledast_{t' \in \mathrm{dom}(T'')} p'(t)$$

This, along with the previously obtained safety and stability properties allows us to use the induction hypothesis to conclude that

$$\triangleright \mathit{eval}(\mu', T', \lambda(\mu'', T''). \ \mu'' \in \lfloor \circledast_{t \in \mathrm{dom}(T)} q(t)(T''(t)) * \circledast_{t \in \mathrm{dom}(T'' - t)} \top \rfloor.$$

However, since we know that $p * \top \subseteq \top$ for any assertion p, this gives us the required obligation that

$$\triangleright \mathit{eval}(\mu', T', \lambda(\mu'', T''). \ \mu'' \in \lfloor \circledast_{t \in \mathrm{dom}(T)} q(t)(T''(t)) \rfloor,$$

which ends the proof. □

Additional logical connectives

$$\ulcorner p \ \mathbf{in}\ t \urcorner \stackrel{\text{def}}{=} \{(l,s,U,\zeta) \in \mathcal{M} \mid (\text{flush}(l, U(t)), s, U[t \mapsto \varepsilon], \zeta) \in \ulcorner p \urcorner\}$$

$$p\,\mathcal{U}_t\,q \stackrel{\text{def}}{=} \{(l,s,U,\zeta) \in \mathcal{M} \mid \exists \alpha, \beta \in \Delta(\mathit{SBuffer}).\ \exists o \in \Delta(\mathit{OId}).\ \exists f \in \Delta(\mathit{FName}).\ \exists v \in \Delta(\mathit{Val}).$$
$$U(t) = \alpha \cdot (o,f,v) \cdot \beta \wedge (l,s,U[t \mapsto \alpha],\zeta) \in p \wedge$$
$$(\text{flush}(l, \alpha \cdot (o,f,v)), s, U[t \mapsto \beta], \zeta) \in q\}$$

$$\underset{A}{\text{cl}}(p) \stackrel{\text{def}}{=} \{m \in \mathcal{M} \mid \exists m' \in p.\ m'\ (\leq \cup R_A)^*\ m\}$$

Semantic read and write judgments

$$p \vdash_{\text{w}} o.f \mapsto v, q \stackrel{\text{def}}{=} \forall t \in \mathit{TId}.\ \forall (l,s,U,\zeta) \in p(t).\ t \in \text{dom}(U) \Rightarrow \triangleright \text{lookup}(o.f, U(t), l) = v \wedge$$
$$\triangleright\,(\text{flush}(l, U(t)), s, U[t \mapsto \varepsilon], \zeta) \in \ulcorner q(t) * o.f \mapsto v \urcorner$$

$$p \vdash^{\text{m}}_{\text{r}} o.f \mapsto v \stackrel{\text{def}}{=} \forall t \in \mathit{TId}.\ \forall m \in p(t).\ t \in \text{dom}(U) \Rightarrow \triangleright \text{lookup}(o.f, m.U(t), m.l) = v$$

$$p \vdash^{\text{o}}_{\text{r}} o.f \mapsto v \stackrel{\text{def}}{=} \forall t, t' \in \mathit{TId}.\ \forall m \in p(t').\ t \neq t' \wedge t \in \text{dom}(U) \Rightarrow$$
$$\triangleright\,\text{lookup}(o.f, m.U(t), m.l) = v$$

3 Logic

The specification logic is given by the specification entailment judgment,

$$\Gamma \mid \Phi \vdash \mathsf{S},$$

where S is a specification Γ — a logical variable context, and Φ — a specification context. The specification logic extends a standard higher-order intuitionistic logic.Similarly, both of the assertion logics, SC and TSO, are given by the assertion logic entailment judgments, of the form

$$\Gamma; \Delta \mid \Phi \mid \mathsf{P} \vdash_L \mathsf{Q},$$

where L ranges over $\{SC, TSO\}$, P and Q are the assertions of the appropriate logic, Γ is the logical variable context, Δ — the program variable context, and Φ — the specification context. The SC logic is a standard higher order assertion logic, while the TSO logic extends the higher-order intuitionistic separation logic with several additional proof rules for the TSO connectives.

3.1 TSO Assertion Logic

Listed below are some of the proof rules of the TSO assertion logic. For the sake of clarity, we do not present the introduction and elimination rules for the standard connectives of higher-order intuitionistic separation logic here.

$$\frac{\Gamma;\Delta \mid \Phi \vdash \mathsf{stable}(\mathsf{P}) \wedge \mathsf{stable}(\mathsf{Q})}{\Gamma;\Delta \mid \Phi \mid (\!|\mathsf{P}\ \mathcal{U}_t\ \mathsf{Q}|\!) \vdash_{TSO} (\mathsf{P}\ \mathcal{U}_t\ \mathsf{Q}) \vee \mathsf{Q}}\ \textsc{Stab-}\mathcal{U}$$

$$\frac{}{\Gamma;\Delta \mid \Phi \mid \mathsf{P} \vdash_{TSO} (\!|\mathsf{P}|\!)}\ \textsc{Stab-I} \qquad \frac{\Gamma;\Delta \mid \Phi \vdash \mathsf{stable}(\mathsf{P})}{\Gamma;\Delta \mid \Phi \mid (\!|\mathsf{P}|\!) \vdash_{TSO} \mathsf{P}}\ \textsc{Stab-S}$$

$$\frac{}{\Gamma;\Delta \mid \Phi \mid \ulcorner \mathsf{P} \urcorner \vdash_{TSO} \ulcorner \mathsf{P}\ \mathbf{in}\ \mathsf{t} \urcorner}\ \textsc{W-HInterp} \qquad \frac{}{\Gamma;\Delta \mid \mathsf{P}\ \mathcal{U}_t\ \mathsf{Q} \vdash_{TSO} \ulcorner \mathsf{Q}\ \mathbf{in}\ \mathsf{t} \urcorner}\ \textsc{W-}\mathcal{U}$$

$$\frac{\Gamma;\Delta \mid \Phi \mid \mathsf{P}_1 \vdash_{TSO} \mathsf{P}_2 \qquad \Gamma;\Delta \mid \Phi \mid \mathsf{Q}_1 \vdash_{TSO} \mathsf{Q}_2}{\Gamma;\Delta \mid \Phi \mid \mathsf{P}_1\ \mathcal{U}_t\ \mathsf{Q}_1 \vdash_{TSO} \mathsf{P}_2\ \mathcal{U}_t\ \mathsf{Q}_2}\ \textsc{Cons-}\mathcal{U}$$

$$\frac{\Gamma;\Delta \mid \Phi \vdash \mathsf{b{-}stable}(\mathsf{P})}{\Gamma;\Delta \mid \Phi \mid \mathsf{P} * (\mathsf{Q}\ \mathcal{U}_t\ \mathsf{R}) \vdash_{TSO} \mathsf{Q}\ \mathcal{U}_t\ (\mathsf{P} * \mathsf{R})}\ *\text{-}\mathcal{U}$$

3.2 Syntactic Sugar

We provide some syntactic sugar for more convenient reasoning within the TSO logic, where the pre- and postconditions are parametrized with the current thread identifier.

$$\begin{aligned}
\overline{\mathsf{P}} &\equiv \lambda \mathsf{t} \in \mathsf{TId}.\ \ulcorner \mathsf{P}\ \mathbf{in}\ \mathsf{t} \urcorner \\
\mathsf{P}\ \mathcal{U}\ \mathsf{Q} &\equiv \lambda \mathsf{t} \in \mathsf{TId}.\ \mathsf{P}(\mathsf{t})\ \mathcal{U}_{\mathsf{t}}\ \mathsf{Q}(\mathsf{t}) \\
\mathsf{P}\ \overline{\mathcal{U}}\ \mathsf{Q} &\equiv \lambda \mathsf{t} \in \mathsf{TId}.\ \exists \mathsf{t}' \in \mathsf{TId}.\ \mathsf{t} \neq \mathsf{t}' * \mathsf{P}(\mathsf{t}')\ \mathcal{U}_{\mathsf{t}'}\ \mathsf{Q}(\mathsf{t}') \\
\mathsf{stable}(\mathsf{P}) &\equiv \mathsf{r{-}stable}(\mathsf{P}) \wedge \mathsf{b{-}stable}(\mathsf{P})
\end{aligned}$$

3.3 Read and Write Rules

These judgments determine what values can be read from or written to the local state. The write judgment also provides the value of the field (for use of the compare-and-swap rule), as well as information about the state that will become flushed at the time the write action reaches the heap. The read judgment comes in two forms, dependent on whether the assertion currently inspected seen from the perspective of the thread which performs the read or some other thread's.

The forms of these judgments are $\Gamma;\Delta \mid \mathsf{P} \vdash^{s}_{\mathrm{r}} \mathsf{x.f} \mapsto \mathsf{v}$ and $\Gamma;\Delta \mid \mathsf{P} \vdash_{\mathrm{w}} \mathsf{x.f} \mapsto \mathsf{v}, \mathsf{Q}$ respectively, where s, ranging over the set $\{\mathrm{m}, \mathrm{o}\}$, is the flag. Note that Q above is an $\mathsf{Prop}_{\mathsf{sc}}$.

$$\frac{}{\Gamma;\Delta \mid \triangleright\overline{\mathsf{x.f} \mapsto \mathsf{v}} \vdash^{\mathrm{m}}_{\mathrm{r}} \mathsf{x.f} \mapsto \mathsf{v}}\ \text{R-SELF} \qquad \frac{}{\Gamma;\Delta \mid \triangleright\ulcorner \mathsf{x.f} \mapsto \mathsf{v}\urcorner \vdash^{\mathrm{o}}_{\mathrm{r}} \mathsf{x.f} \mapsto \mathsf{v}}\ \text{R-OTHER}$$

$$\frac{\Gamma;\Delta \mid \mathsf{Q} \vdash^{\mathrm{m}}_{\mathrm{r}} \mathsf{x.f} \mapsto \mathsf{v}}{\Gamma;\Delta \mid \mathsf{P}\ \mathcal{U}\ \mathsf{Q} \vdash^{\mathrm{m}}_{\mathrm{r}} \mathsf{x.f} \mapsto \mathsf{v}}\ \text{R-}\mathcal{U}\text{-M} \qquad \frac{\Gamma;\Delta \mid \mathsf{P} \vdash^{\mathrm{o}}_{\mathrm{r}} \mathsf{x.f} \mapsto \mathsf{v}}{\Gamma;\Delta \mid \mathsf{P}\ \mathcal{U}\ \mathsf{Q} \vdash^{\mathrm{o}}_{\mathrm{r}} \mathsf{x.f} \mapsto \mathsf{v}}\ \text{R-}\mathcal{U}\text{-O}$$

$$\frac{\Gamma;\Delta \mid \mathsf{P} \vdash^{\mathrm{o}}_{\mathrm{r}} \mathsf{x.f} \mapsto \mathsf{v}}{\Gamma;\Delta \mid \mathsf{P}\ \overline{\mathcal{U}}\ \mathsf{Q} \vdash^{\mathrm{m}}_{\mathrm{r}} \mathsf{x.f} \mapsto \mathsf{v}}\ \text{R-}\overline{\mathcal{U}} \qquad \frac{\Gamma;\Delta \mid \cdot \mid \mathsf{P} \vdash \mathsf{Q} \qquad \Gamma;\Delta \mid \mathsf{Q} \vdash^{s}_{\mathrm{r}} \mathsf{x.f} \mapsto \mathsf{v}}{\Gamma;\Delta \mid \mathsf{P} \vdash^{s}_{\mathrm{r}} \mathsf{x.f} \mapsto \mathsf{v}}\ \text{R-CONS}$$

$$\frac{}{\Gamma;\Delta \mid \triangleright\overline{\mathsf{x.f} \mapsto \mathsf{v}} \vdash_{\mathrm{w}} \mathsf{x.f} \mapsto \mathsf{v}, \top}\ \text{W-AX} \qquad \frac{\Gamma;\Delta \mid \mathsf{P} \vdash_{\mathrm{w}} \mathsf{x.f} \mapsto \mathsf{v}, \mathsf{R}}{\Gamma;\Delta \mid \mathsf{P} * \triangleright\overline{\mathsf{Q}} \vdash_{\mathrm{w}} \mathsf{x.f} \mapsto \mathsf{v}, \mathsf{Q} * \mathsf{R}}\ \text{W-}*$$

$$\frac{\Gamma;\Delta \mid \mathsf{Q} \vdash_{\mathrm{w}} \mathsf{x.f} \mapsto \mathsf{v}, \mathsf{R}}{\Gamma;\Delta \mid \mathsf{P}\ \mathcal{U}\ \mathsf{Q} \vdash_{\mathrm{w}} \mathsf{x.f} \mapsto \mathsf{v}, \mathsf{R}}\ \text{W-}\mathcal{U} \qquad \frac{\Gamma;\Delta \mid \cdot \mid \mathsf{P} \vdash \mathsf{Q} \qquad \Gamma;\Delta \mid \mathsf{Q} \vdash_{\mathrm{w}} \mathsf{x.f} \mapsto \mathsf{v}, \mathsf{R}}{\Gamma;\Delta \mid \mathsf{P} \vdash_{\mathrm{w}} \mathsf{x.f} \mapsto \mathsf{v}, \mathsf{R}}\ \text{W-CONS}$$

3.4 Specification Logic

In the following we present the rules for reasoning within the specification logic. The introduction and elimination rules for standard higher-order intuitionistic logic are omitted, since they are completely standard.

3.4.1 Reasoning about Hoare triples

Atomic Command Rules

$$\frac{\Gamma;\Delta \mid \mathsf{P} \vdash^{\mathrm{m}}_{\mathrm{r}} \mathsf{x.f} \mapsto \mathsf{v}}{\Gamma;\Delta \mid \Phi \vdash \langle \mathsf{P} \rangle\ \mathsf{x.f}\ \langle \mathsf{r}.\ \mathsf{P} * \mathsf{r} = \mathsf{v} \rangle^{C}}\ \text{A-READ}$$

$$\frac{\Gamma;\Delta \mid \mathsf{P} \vdash_{\mathrm{w}} \mathsf{x.f} \mapsto \mathsf{v}', \mathsf{Q}}{\Gamma;\Delta \mid \Phi \vdash \langle \mathsf{P} \rangle\ \mathsf{x.f} := \mathsf{v}\ \langle \_.\ \mathsf{P}\ \mathcal{U}\ (\ulcorner \mathsf{Q} * \mathsf{x.f} \mapsto \mathsf{v}\urcorner) \rangle^{C}}\ \text{A-WRITE}$$

$$\frac{\Gamma;\Delta \mid \mathsf{P} \vdash^{\mathrm{m}}_{\mathrm{r}} \mathsf{x.f} \mapsto \mathsf{v} \qquad \mathsf{v} \neq \mathsf{v}_2}{\Gamma;\Delta \mid \Phi \vdash \langle \mathsf{P} \rangle\ \mathbf{CAS}(\mathsf{x.f}, \mathsf{v}_1, \mathsf{v}_2)\ \langle \mathsf{r}.\ \mathsf{r} = \mathbf{false} * P \rangle^{C}}\ \text{A-CAS-FALSE}$$

$$\frac{\Gamma;\Delta \mid \mathsf{P} \vdash_{\mathrm{w}} \mathsf{x.f} \mapsto \mathsf{v}, \mathsf{Q}}{\Gamma;\Delta \mid \Phi \vdash \langle \mathsf{P} \rangle\ \mathbf{CAS}(\mathsf{x.f}, \mathsf{v}', \mathsf{v})\ \langle \mathsf{r}.\ \mathsf{r} = \mathbf{true} * \ulcorner \mathsf{Q} * \mathsf{x.f} \mapsto \mathsf{v}'\urcorner \rangle^{C}}\ \text{A-CAS-TRUE}$$

$$\frac{\begin{array}{c}\Gamma, \Delta \mid \Phi \vdash \forall \mathsf{y}, \mathsf{r}.\ \mathsf{stable}(\mathsf{Q}(\mathsf{y}, \mathsf{r})) \\ \Gamma, \Delta \mid \Phi \vdash \mathsf{n} \in C \qquad \Gamma, \Delta \mid \Phi \vdash \forall \mathsf{x} \in X.\ \forall \mathsf{y} \in f(\mathsf{x}).\ (\mathsf{x}, \mathsf{y}) \in (T(A))^{*} \\ \Gamma \mid \Phi \vdash \forall \mathsf{x} \in X.\ (\Delta).\langle \mathsf{P} * \triangleright I(\mathsf{x}) \rangle\ \mathsf{e}\ \langle \mathsf{r}.\ \exists \mathsf{y} \in f(\mathsf{x}).\ \mathsf{Q}(\mathsf{y}, \mathsf{r}) * \circledast_{\alpha \in A}[\alpha]^{\mathsf{n}}_{g(\alpha)} * \triangleright I(\mathsf{y}) \rangle^{C \setminus \{\mathsf{n}\}}\end{array}}{\begin{array}{c}\Gamma \mid \Phi \vdash (\Delta).\langle \mathsf{P} * \mathsf{region}(X, T, I, \mathsf{n}) \rangle \\ \mathsf{e} \\ \langle \mathsf{r}.\ \exists \mathsf{y}.\ \mathsf{Q}(\mathsf{y}, \mathsf{r}) * \circledast_{\alpha \in A}[\alpha]^{\mathsf{n}}_{g(\alpha)} * \mathsf{region}(\{\mathsf{y}\}, T, I, \mathsf{n}) \rangle^{C}\end{array}}\ \text{A-REGION}$$

$$\frac{\Gamma \mid \Phi \vdash (\Delta).\langle \mathsf{P} \rangle\ \mathsf{e}\ \langle \mathsf{Q} \rangle^{C} \qquad \mathsf{atomic}(e)}{\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P}\}\ \mathsf{e}\ \{\mathsf{Q}\}}\ \text{A-START}$$

Structural Rules

$$\frac{\Gamma \mid \Phi \vdash \mathsf{P} \sqsubseteq^{A} \mathsf{Q} \qquad \Gamma \mid \Phi \vdash \mathsf{stable}(\mathsf{R})}{\Gamma \mid \Phi \vdash \mathsf{P} * \mathsf{R} \sqsubseteq^{A} \mathsf{Q} * \mathsf{R}}\ \textsc{V-Frame}$$

$$\frac{\Gamma \mid \Phi \vdash (\Delta).\langle \mathsf{P} \rangle\ \mathsf{e}\ \langle \mathsf{Q} \rangle \qquad \Gamma, \Delta \mid \Phi \vdash \mathsf{stable}(\mathsf{R})}{\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P} * \triangleright \mathsf{R}\}\ \mathsf{e}\ \{\mathsf{Q} * \mathsf{R}\}}\ \textsc{A-Frame}$$

$$\frac{\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P}\}\ \mathsf{e}\ \{\mathsf{Q}\} \qquad \Gamma, \Delta \mid \Phi \vdash \mathsf{stable}(\mathsf{R})}{\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P} * \mathsf{R}\}\ \mathsf{e}\ \{\mathsf{Q} * \mathsf{R}\}}\ \textsc{Frame}$$

$$\frac{\Gamma \mid \Phi \vdash \mathsf{P}_1 \sqsubseteq^{A} \mathsf{P}_2 \qquad \Gamma \mid \Phi \vdash (\Delta).\langle \mathsf{P}_2 \rangle\ \mathsf{e}\ \langle \mathsf{Q}_2 \rangle^{A} \qquad \Gamma \mid \Phi \vdash \mathsf{Q}_2 \sqsubseteq^{A} \mathsf{Q}_1}{\Gamma \mid \Phi \vdash (\Delta).\langle \mathsf{P}_1 \rangle\ \mathsf{e}\ \langle \mathsf{Q}_1 \rangle^{A}}\ \textsc{A-Cons}$$

$$\frac{\Gamma \mid \Phi \vdash \mathsf{P}_1 \sqsubseteq \mathsf{P}_2 \qquad \Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P}_2\}\ \mathsf{e}\ \{\mathsf{Q}_2\} \qquad \Gamma \mid \Phi \vdash \mathsf{Q}_2 \sqsubseteq \mathsf{Q}_1}{\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P}_1\}\ \mathsf{e}\ \{\mathsf{Q}_1\}}\ \textsc{Cons}$$

Other Rules for Commands

$$\frac{\begin{array}{c}\Gamma, \bar{\mathsf{y}}, \mathbf{this} \mid \Phi \vdash \mathsf{stable}(\mathsf{P}) \wedge \mathsf{stable}(\mathsf{Q}) \\ \Gamma \mid \Phi \vdash \triangleright(\mathsf{C}::\mathsf{m} : (\bar{\mathsf{y}}).\ \{\mathsf{P}\}\ \{\mathsf{Q}\}) \qquad \Gamma \vdash \mathsf{C} : \mathsf{Class}\end{array}}{\Gamma \mid \Phi(\Delta).\{\mathsf{P}[\bar{\mathsf{v}}/\bar{\mathsf{y}}, \mathsf{x}/\mathbf{this}] * \mathsf{x} : \mathsf{C}\}\ \mathsf{x}.\mathsf{m}(\bar{\mathsf{v}})\ \{\mathsf{Q}[\bar{\mathsf{v}}/\bar{\mathsf{y}}, \mathsf{x}/\mathbf{this}]\}}\ \textsc{Call}$$

$$\frac{\begin{array}{c}\Gamma, \mathbf{this} \mid \Phi \vdash \mathsf{stable}(\mathsf{P}) \wedge \mathsf{stable}(\mathsf{Q}) \\ \Gamma \mid \Phi \vdash \triangleright(\mathsf{C}::\mathsf{m} : (-).\ \{\ulcorner \mathsf{P} \urcorner\}\ \{\mathsf{Q}\}) \qquad \Gamma \vdash \mathsf{C} : \mathsf{Class}\end{array}}{\Gamma \mid \Phi \vdash (\Delta).\{\ulcorner \mathsf{P}[\mathsf{x}/\mathbf{this}] \urcorner * \mathsf{y} : \mathsf{C}\}\ \mathbf{fork}(\mathsf{y}.\mathsf{m})\ \{\top\}}\ \textsc{Fork}$$

$$\frac{\Gamma, \bar{\mathsf{x}} \mid \Phi \vdash \mathsf{stable}(\mathsf{P}) \wedge \mathsf{stable}(\mathsf{Q}) \qquad \Gamma \mid \Phi \vdash \triangleright(\mathsf{C} : (\bar{\mathsf{x}}).\ \{\mathsf{P}\}\ \{\mathsf{Q}\}) \qquad \Gamma \vdash \mathsf{C} : \mathsf{Class}}{\Gamma \mid \Phi(\Delta).\{\mathsf{P}[\bar{\mathsf{v}}/\bar{\mathsf{x}}]\}\ \mathbf{new}\ \mathsf{C}(\bar{\mathsf{v}})\ \{\mathsf{Q}[\bar{\mathsf{v}}/\bar{\mathsf{x}}]\}}\ \textsc{New}$$

$$\frac{\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P}\}\ \mathsf{e}_1\ \{\mathsf{r}.\ \mathsf{Q}(\mathsf{r})\} \qquad \Gamma \mid \Phi \vdash (\Delta, \mathsf{x}).\{\mathsf{Q}(\mathsf{x})\}\ \mathsf{e}_2\ \{\mathsf{r}.\ \mathsf{R}(\mathsf{r})\}}{\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P}\}\ \mathbf{let}\ \mathsf{x} = \mathsf{e}_1\ \mathbf{in}\ \mathsf{e}_2\ \{\mathsf{r}.\ \mathsf{R}(\mathsf{r})\}}\ \textsc{Bind}$$

$$\frac{\Gamma; - \vdash \mathsf{v} : \mathsf{Val}}{\Gamma \mid \Phi \vdash \{\top\}\ \mathsf{v}\ \{\mathsf{r}.\ \mathsf{r} = \mathsf{v}\}}\ \textsc{Val}$$

View-Shifts

$$\frac{\Gamma \mid \Phi \vdash \mathsf{P} \sqsubseteq^{A} \mathsf{Q} \qquad \Gamma \mid \Phi \vdash \mathsf{Q} \sqsubseteq^{A} \mathsf{R}}{\Gamma \mid \Phi \vdash \mathsf{Q} \sqsubseteq^{A} \mathsf{R}}\ \textsc{VTrans} \qquad \frac{\Gamma \mid \Phi \mid \mathsf{P} \vdash \mathsf{Q}}{\Gamma \mid \Phi \vdash \mathsf{P} \sqsubseteq^{A} \mathsf{Q}}\ \textsc{VImpl}$$

$$\frac{\begin{array}{c}\Gamma \mid \Phi \vdash x \in X \qquad \Gamma \mid \Phi \vdash \forall n \in C.\ \forall s.\ up(I(n)(s)) \\ \Gamma \mid \Phi \vdash A \text{ and } B \text{ are finite} \qquad \Gamma \mid \Phi \vdash C \text{ is infinite} \\ \Gamma \mid \Phi \vdash \forall n \in C.\ \mathsf{P} * \circledast_{\alpha \in A}[\alpha]_1^n \Rightarrow \triangleright I(n)(x) \\ \Gamma \mid \Phi \vdash \forall n \in C.\ \forall s.\ \mathsf{stable}(I(n)(s)) \qquad \Gamma \mid \Phi \vdash A \cap B = \emptyset\end{array}}{\Gamma \mid \Phi \vdash \mathsf{P} \sqsubseteq^{C} \exists n \in C.\ \mathsf{region}(X, T, I(n), n) * \circledast_{\alpha \in B}[\alpha]_1^n}\ \textsc{VAlloc}$$

$$\frac{\begin{array}{c}\Gamma \mid \Phi \vdash \mathrm{stable}(\mathsf{P}) \qquad \Gamma \mid \Phi \vdash \mathrm{stable}(\mathsf{Q}) \\ \Gamma \mid \Phi \vdash n \in A \qquad \Gamma \mid \Phi \vdash \forall x \in X.\ f(x) \in Y \\ \Gamma \mid \Phi \vdash \forall x \in X.\ (x, f(x)) \in T(\alpha) \vee f(x) = x \\ \Gamma \mid \Phi \vdash \forall x \in X.\ \mathsf{P} * \triangleright I(x) * [\alpha]^n_\pi \sqsubseteq^{A \setminus \{n\}} \mathsf{Q} * \triangleright I(f(x))\end{array}}{\Gamma \mid \Phi \vdash \mathsf{region}(X, T, I, n) * \mathsf{P} * [\alpha]^n_\pi \sqsubseteq^{A} \mathsf{region}(Y, T, I, n) * \mathsf{Q}}\ \textsc{VOpen}$$

Derived Rules – Hoare triples for the SC Logic

$$\frac{\Gamma; - \vdash \mathsf{v} : \mathsf{Val} \qquad \mathsf{x} \in \Delta}{\Gamma \mid \Phi \vdash (\Delta).[\mathsf{x.f} \mapsto \mathsf{v}]\ \mathsf{x.f}\ [\mathsf{r}.\ \mathsf{r} = \mathsf{v} * \mathsf{x.f} \mapsto \mathsf{v}]}\ \text{S-}\textsc{Read}$$

$$\frac{\Gamma; - \vdash \mathsf{v} : \mathsf{Val} \qquad \mathsf{x} \in \Delta}{\Gamma \mid \Phi \vdash (\Delta).[\mathsf{x.f} \mapsto \_]\ \mathsf{x.f} := \mathsf{v}\ [\mathsf{r}.\ \mathsf{x.f} \mapsto \mathsf{v}]}\ \text{S-}\textsc{Write}$$

$$\frac{\begin{array}{c}\Gamma, \bar{\mathsf{y}}, \mathbf{this}, \mathsf{r} \mid \Phi \vdash \mathsf{r}{-}\mathrm{stable}(\mathsf{P}) \wedge \mathsf{r}{-}\mathrm{stable}(\mathsf{Q}) \\ \Gamma \mid \Phi \vdash \triangleright(\mathsf{C::m} : (\bar{\mathsf{y}}).\ [\mathsf{P}]\ [\mathsf{r}.\ \mathsf{Q}]) \qquad \Gamma \vdash \mathsf{C} : \mathsf{Class}\end{array}}{\Gamma \mid \Phi(\Delta).[\mathsf{P}[\bar{\mathsf{v}}/\bar{\mathsf{y}}, \mathsf{x}/\mathbf{this}] * \mathsf{x} : \mathsf{C}]\ \mathsf{x.m}(\bar{\mathsf{v}})\ [\mathsf{r}.\ \mathsf{Q}[\bar{\mathsf{v}}/\bar{\mathsf{y}}, \mathsf{x}/\mathbf{this}]]}\ \text{S-}\textsc{Call}$$

$$\frac{\begin{array}{c}\Gamma, \bar{\mathsf{x}}, \mathsf{r} \mid \Phi \vdash \mathrm{stable}(\mathsf{P}) \wedge \mathrm{stable}(\mathsf{Q}) \\ \Gamma \mid \Phi \vdash \triangleright(\mathsf{C} : (\bar{\mathsf{x}}).\ [\mathsf{P}]\ [\mathsf{r}.\ \mathsf{Q}]) \qquad \Gamma \vdash \mathsf{C} : \mathsf{Class}\end{array}}{\Gamma \mid \Phi(\Delta).[\mathsf{P}[\bar{\mathsf{v}}/\bar{\mathsf{x}}]]\ \mathbf{new}\ \mathsf{C}(\bar{\mathsf{v}})\ [\mathsf{r}.\ \mathsf{Q}[\bar{\mathsf{v}}/\bar{\mathsf{x}}]]}\ \text{S-}\textsc{New}$$

$$\frac{\Gamma \mid \Phi \vdash (\Delta).[\mathsf{P}]\ \mathsf{e}\ [\mathsf{r}.\ \mathsf{Q}] \qquad \Gamma, \Delta \mid \Phi \vdash \mathsf{r}{-}\mathrm{stable}(\mathsf{R})}{\Gamma \mid \Phi \vdash (\Delta).[\mathsf{P} * \mathsf{R}]\ \mathsf{e}\ [\mathsf{r}.\ \mathsf{Q} * \mathsf{R}]}\ \text{S-}\textsc{Frame}$$

$$\frac{\Gamma; \Delta \mid \Phi \mid \mathsf{P}_1 \vdash_{SC} \mathsf{P}_2 \qquad \Gamma \mid \Phi \vdash (\Delta).[\mathsf{P}_2]\ \mathsf{e}\ [\mathsf{r}.\ \mathsf{Q}_2] \qquad \Gamma, \mathsf{r}; \Delta \mid \Phi \mid \mathsf{Q}_2 \vdash_{SC} \mathsf{Q}_1}{\Gamma \mid \Phi \vdash (\Delta).[\mathsf{P}_1]\ \mathsf{e}\ [\mathsf{r}.\ \mathsf{Q}_1]}\ \text{S-}\textsc{Cons}$$

$$\frac{\Gamma; \Delta \mid \Phi \mid \mathsf{P} \vdash \mathsf{v} = \mathbf{true} \qquad \Gamma \mid \Phi \vdash (\Delta).[\mathsf{P}]\ \mathsf{e}_1\ [\mathsf{r}.\ \mathsf{Q}]}{\Gamma \mid \Phi \vdash (\Delta).[\mathsf{P}]\ \mathbf{if}\ \mathsf{v}\ \mathbf{then}\ \mathsf{e}_1\ \mathbf{else}\ \mathsf{e}_2\ [\mathsf{r}.\ \mathsf{Q}]}\ \text{S-}\textsc{If-T}$$

$$\frac{\Gamma; \Delta \mid \Phi \mid \mathsf{P} \vdash \mathsf{v} = \mathbf{false} \qquad \Gamma \mid \Phi \vdash (\Delta).[\mathsf{P}]\ \mathsf{e}_2\ [\mathsf{r}.\ \mathsf{Q}]}{\Gamma \mid \Phi \vdash (\Delta).[\mathsf{P}]\ \mathbf{if}\ \mathsf{v}\ \mathbf{then}\ \mathsf{e}_1\ \mathbf{else}\ \mathsf{e}_2\ [\mathsf{r}.\ \mathsf{Q}]}\ \text{S-}\textsc{If-F}$$

$$\frac{\Gamma \mid \Phi \vdash (\Delta).[\mathsf{P}]\ \mathsf{e}_1\ [\mathsf{r}.\ \mathsf{Q}(\mathsf{r})] \qquad \Gamma \mid \Phi \vdash (\Delta, \mathsf{x}).[\mathsf{Q}(\mathsf{x})]\ \mathsf{e}_2\ [\mathsf{r}.\ \mathsf{R}(\mathsf{r})]}{\Gamma \mid \Phi \vdash (\Delta).[\mathsf{P}]\ \mathbf{let}\ \mathsf{x} = \mathsf{e}_1\ \mathbf{in}\ \mathsf{e}_2\ [\mathsf{r}.\ \mathsf{R}(\mathsf{r})]}\ \text{S-}\textsc{Bind}$$

$$\frac{\Gamma; - \vdash \mathsf{v} : \mathsf{Val}}{\Gamma \mid \Phi \vdash [\top]\ \mathsf{v}\ [\mathsf{r}.\ \mathsf{r} = \mathsf{v}]}\ \text{S-}\textsc{Val} \qquad \frac{\Gamma \mid \Phi \vdash (\Delta).\{\overline{\mathsf{P}}\}\ \mathsf{e}\ \{\overline{\mathsf{Q}}\}}{\Gamma \mid \Phi \vdash (\Delta).[\mathsf{P}]\ \mathsf{e}\ [\mathsf{r}.\ \mathsf{Q}(\mathsf{r})]}\ \text{S-}\textsc{Shift}$$

3.4.2 Later operator

In the following proof rules, the rules for distribution over quantifiers and binary connectives work for both SC- and TSO-level connectives (except for $\mathcal{U}$ operator, which is only defined on the TSO level), hence we omit the subscripts on the entailments.

$$\frac{\Gamma \mid \Phi, \triangleright \mathsf{S} \vdash \mathsf{S}}{\Gamma \mid \Phi \vdash \mathsf{S}}\ \textsc{SLob} \qquad \frac{}{\Gamma \mid \Phi \vdash \mathsf{S} \Rightarrow \triangleright \mathsf{S}}\ \textsc{SMono}$$

$$\frac{op \in \{\wedge, \vee, *, \mathcal{U}_t\}}{\Gamma; \Delta \mid \Phi \mid \triangleright(\mathsf{P}\ op\ \mathsf{Q}) \dashv\vdash (\triangleright \mathsf{P})\ op\ (\triangleright \mathsf{Q})}\ \textsc{LBin} \qquad \frac{E \in \{\ulcorner - \urcorner, \ulcorner - \textbf{ in } t \urcorner\}}{\Gamma; \Delta \mid \Phi \mid \triangleright(E(\mathsf{P})) \dashv\vdash E(\triangleright P)}\ \textsc{LEmb}$$

$$\frac{}{\Gamma; \Delta \mid \Phi \mid \triangleright(\mathsf{P} \Rightarrow \mathsf{Q}) \vdash (\triangleright \mathsf{P}) \Rightarrow (\triangleright \mathsf{Q})}\ \textsc{LImpl} \qquad \frac{}{\Gamma; \Delta \mid \Phi \mid \triangleright(\mathsf{P} \mathrel{-\!\!*} \mathsf{Q}) \vdash (\triangleright \mathsf{P}) \mathrel{-\!\!*} (\triangleright \mathsf{Q})}\ \textsc{LWand}$$

$$\frac{Q \in \{\forall, \exists\}}{\Gamma; \Delta \mid \Phi \mid \triangleright(Qx : \tau.\ \mathsf{P}(x)) \dashv\vdash Qx : \tau.\ \triangleright \mathsf{P}(x)}\ \textsc{LQuant}$$

3.4.3 Reasoning about stability

Stability consists of two components: stability under region interference ($\mathsf{r{-}stable}$) and stability under store-buffer interference ($\mathsf{b{-}stable}$). The first of these is defined for both level of assertions, the second only for $\mathsf{Prop}_{\mathsf{tso}}$. Note how both embeddings preserve stability under regions and *grant* stability under store-buffer interference, and that the explicit stabilization does indeed provide stability.

$$\frac{\Gamma \mid \Phi \vdash \forall \alpha \notin A.\ \forall x \in X.\ T(\alpha)(x) \subseteq X}{\Gamma \mid \Phi \vdash \mathsf{r{-}stable}(\mathsf{region}(X, T, n) * \circledast_{\alpha \in A} [\alpha]_1^n)}$$

$$\frac{\Gamma \vdash \mathsf{I} : \mathsf{AId} \to \mathcal{P}(\mathsf{SId} \times \mathsf{SId}) \qquad \Gamma \vdash \mathsf{n} : \mathsf{RId}}{\Gamma \mid \Phi \vdash \mathsf{r{-}stable}(\mathsf{rintr}(\mathsf{I}, \mathsf{n}))}$$

$$\frac{c \in \{\top, \bot, \mathsf{emp}\}}{\Gamma \mid \Phi \vdash \mathsf{stable}(c)} \qquad \frac{}{\Gamma \mid \Phi \vdash \mathsf{stable}(\mathsf{x.f} \mapsto \mathsf{v})} \qquad \frac{}{\Gamma \mid \Phi \vdash \mathsf{stable}(\mathsf{x} : \mathsf{C})}$$

$$\frac{}{\Gamma \mid \Phi \vdash \mathsf{stable}(\llparenthesis \mathsf{P} \rrparenthesis)} \qquad \frac{}{\Gamma \mid \Phi \vdash \mathsf{b{-}stable}(\ulcorner \mathsf{P} \urcorner)} \qquad \frac{}{\Gamma \mid \Phi \vdash \mathsf{b{-}stable}(\ulcorner \mathsf{P}\ \textbf{in}\ \mathsf{t} \urcorner)}$$

$$\frac{\Gamma \mid \Phi \vdash \mathsf{r{-}stable}(\mathsf{P})}{\Gamma \mid \Phi \vdash \mathsf{r{-}stable}(\ulcorner \mathsf{P} \urcorner)} \qquad \frac{\Gamma \mid \Phi \vdash \mathsf{r{-}stable}(\mathsf{P})}{\Gamma \mid \Phi \vdash \mathsf{r{-}stable}(\ulcorner \mathsf{P}\ \textbf{in}\ \mathsf{t} \urcorner)}$$

$$\frac{\Gamma \mid \Phi \vdash \mathsf{b{-}stable(P)} \qquad \Gamma \mid \Phi \vdash \mathsf{b{-}stable(Q)} \qquad op \in \{\wedge, \vee, *\}}{\Gamma \mid \Phi \vdash \mathsf{b{-}stable(P}\ op\ \mathsf{Q)}}$$

$$\frac{\Gamma \mid \Phi \vdash \forall \mathsf{x} : \tau.\ \mathsf{b{-}stable(P(x))}}{\Gamma \mid \Phi \vdash \mathsf{b{-}stable}(\forall \mathsf{x} : \tau.\ \mathsf{P(x)})} \qquad \frac{\Gamma \mid \Phi \vdash \forall \mathsf{x} : \tau.\ \mathsf{b{-}stable(P(x))}}{\Gamma \mid \Phi \vdash \mathsf{b{-}stable}(\exists \mathsf{x} : \tau.\ \mathsf{P(x)})}$$

$$\frac{\Gamma \mid \Phi \vdash \mathsf{r{-}stable(P)} \qquad \Gamma \mid \Phi \vdash \mathsf{r{-}stable(Q)} \qquad op \in \{\wedge, \vee, *\}}{\Gamma \mid \Phi \vdash \mathsf{r{-}stable(P}\ op\ \mathsf{Q)}}$$

$$\frac{\Gamma \mid \Phi \vdash \forall \mathsf{x} : \tau.\ \mathsf{r{-}stable(P(x))}}{\Gamma \mid \Phi \vdash \mathsf{r{-}stable}(\forall \mathsf{x} : \tau.\ \mathsf{P(x)})} \qquad \frac{\Gamma \mid \Phi \vdash \forall \mathsf{x} : \tau.\ \mathsf{r{-}stable(P(x))}}{\Gamma \mid \Phi \vdash \mathsf{r{-}stable}(\exists \mathsf{x} : \tau.\ \mathsf{P(x)})}$$

3.4.4 Method verification

To verify a method or a constructor, we verify its body:

$$\frac{\begin{array}{c}\Gamma; \Delta, \mathbf{this} \vdash \mathsf{P} : \mathsf{TId} \to \mathsf{Prop_{TSO}} \qquad \Gamma; \Delta, \mathbf{this} \vdash \mathsf{Q} : \mathsf{TId} \to \mathsf{Val} \to \mathsf{Prop_{TSO}} \\ \mathsf{body(C, m)} = (\mathsf{T\ m}(\Delta) = \mathsf{e}) \qquad \Gamma \mid \Phi \vdash (\Delta, \mathbf{this}).\{\mathsf{P} * \mathbf{this} : \mathsf{C}\}\ \mathsf{e}\ \{\mathsf{Q}\}\end{array}}{\Gamma \mid \Phi \vdash \mathsf{C{::}m} : (\Delta).\ \{\mathsf{P}\}\ \{\mathsf{Q}\}}$$

$$\frac{\begin{array}{c}\mathsf{ctorBody(C)} = (\mathsf{C}(\Delta) = \mathsf{e}) \\ \Gamma; \Delta \vdash \mathsf{P} : \mathsf{TId} \to \mathsf{Prop_{TSO}} \qquad \Gamma; \Delta \vdash \mathsf{Q} : \mathsf{TId} \to \mathsf{Val} \to \mathsf{Prop_{TSO}} \\ \Gamma \mid \Phi \vdash (\Delta, \mathbf{this}).\{\mathsf{P} * \circledast_{\mathsf{f} \in \mathsf{fields}(C)} \ulcorner \mathbf{this}.\mathsf{f} \mapsto \mathbf{null} \urcorner * \mathbf{this} : \mathsf{C}\}\ \mathsf{e}\ \{\mathsf{Q}\}\end{array}}{\Gamma \mid \Phi \vdash \mathsf{C} : (\Delta).\ \{\mathsf{P}\}\ \{\mathsf{Q}\}}$$

Note that constructors are syntactically required to return the value of the newly allocated object.

4 Interpretation

4.1 Types

$$\begin{aligned}
[\![\vdash 1 : \mathsf{Type}]\!] &= 1 \\
[\![\vdash \tau \to \sigma : \mathsf{Type}]\!] &= [\![\vdash \tau : \mathsf{Type}]\!] \to [\![\vdash \sigma : \mathsf{Type}]\!] \\
[\![\vdash \tau \times \sigma : \mathsf{Type}]\!] &= [\![\vdash \tau : \mathsf{Type}]\!] \times [\![\vdash \sigma : \mathsf{Type}]\!] \\
[\![\vdash \tau + \sigma : \mathsf{Type}]\!] &= [\![\vdash \tau : \mathsf{Type}]\!] + [\![\vdash \sigma : \mathsf{Type}]\!] \\
[\![\vdash \mathcal{P}(\tau) : \mathsf{Type}]\!] &= \mathcal{P}([\![\vdash \tau : \mathsf{Type}]\!]) \\
[\![\vdash \mathsf{Prop_{TSO}} : \mathsf{Type}]\!] &= \mathit{Prop}_{TSO} \\
[\![\vdash \mathsf{Prop_{SC}} : \mathsf{Type}]\!] &= \mathit{Prop}_{SC} \\
[\![\vdash \mathsf{Spec} : \mathsf{Type}]\!] &= \Omega \\
[\![\vdash \Delta(X) : \mathsf{Type}]\!] &= \Delta(X)
\end{aligned}$$

4.2 Contexts

$$\llbracket \Gamma, \mathsf{x} : \tau \rrbracket = \llbracket \Gamma \rrbracket \times \llbracket \vdash \tau : \mathsf{Type} \rrbracket \qquad \llbracket \varepsilon \rrbracket = 1$$
$$\llbracket \Delta, \mathsf{x} : \mathsf{Val} \rrbracket = \llbracket \Delta \rrbracket \times \llbracket \vdash \mathsf{Val} : \mathsf{Type} \rrbracket \qquad \llbracket \varepsilon \rrbracket = 1$$

4.3 Lambda calculus

$$\begin{aligned}
\llbracket \Gamma; \Delta \vdash \mathsf{x} : \tau \rrbracket(\gamma, \delta) &= \pi_{\mathsf{x}}(\vartheta) \\
\llbracket \Gamma; \Delta \vdash \mathsf{x} : \mathsf{Val} \rrbracket(\gamma, \delta) &= \pi_{\mathsf{x}}(\delta) \\
\llbracket \Gamma; \Delta \vdash \lambda \mathsf{x} : \tau.\ \mathsf{M} : \tau \to \sigma \rrbracket(\gamma, \delta) &= \lambda v \in \llbracket \vdash \tau : \mathsf{Type} \rrbracket.\ \llbracket \Gamma, \mathsf{x} : \tau; \Delta \vdash \mathsf{M} : \sigma \rrbracket((\vartheta, v), \theta) \\
\llbracket \Gamma; \Delta \vdash \mathsf{M}\ \mathsf{N} : \sigma \rrbracket(\gamma, \delta) &= (\llbracket \Gamma; \Delta \vdash \mathsf{M} : \tau \to \sigma \rrbracket(\gamma, \delta))(\llbracket \Gamma; \Delta \vdash \mathsf{N} : \tau \rrbracket(\gamma, \delta))
\end{aligned}$$

4.4 TSO Assertion Logic

$$\begin{aligned}
\llbracket \Gamma; \Delta \vdash \bot : \mathsf{Prop}_{\mathsf{TSO}} \rrbracket(\gamma, \delta) &= \emptyset \\
\llbracket \Gamma; \Delta \vdash \top : \mathsf{Prop}_{\mathsf{TSO}} \rrbracket(\gamma, \delta) &= \mathcal{M} \\
\llbracket \Gamma; \Delta \vdash \mathsf{P} \wedge \mathsf{Q} : \mathsf{Prop}_{\mathsf{TSO}} \rrbracket(\gamma, \delta) &= \llbracket \Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop}_{\mathsf{TSO}} \rrbracket(\gamma, \delta) \cap \llbracket \Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop}_{\mathsf{TSO}} \rrbracket(\gamma, \delta) \\
\llbracket \Gamma; \Delta \vdash \mathsf{P} \vee \mathsf{Q} : \mathsf{Prop}_{\mathsf{TSO}} \rrbracket(\gamma, \delta) &= \llbracket \Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop}_{\mathsf{TSO}} \rrbracket(\gamma, \delta) \cup \llbracket \Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop}_{\mathsf{TSO}} \rrbracket(\gamma, \delta) \\
\llbracket \Gamma; \Delta \vdash \mathsf{P} \Rightarrow \mathsf{Q} : \mathsf{Prop}_{\mathsf{TSO}} \rrbracket(\gamma, \delta) &= \{m \in \mathcal{M} \mid \forall n \geq m. \\
&\qquad n \in \llbracket \Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop}_{\mathsf{TSO}} \rrbracket(\gamma, \delta)\ \Rightarrow \\
&\qquad n \in \llbracket \Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop}_{\mathsf{TSO}} \rrbracket(\gamma, \delta)\} \\
\llbracket \Gamma; \Delta \vdash \forall \mathsf{x} : \tau.\ \mathsf{P} : \mathsf{Prop}_{\mathsf{TSO}} \rrbracket(\gamma, \delta) &= \bigcap_{v \in \llbracket \vdash \tau : \mathsf{Type} \rrbracket} \llbracket \Gamma, \mathsf{x} : \tau; \Delta \vdash \mathsf{P} : \mathsf{Prop}_{\mathsf{TSO}} \rrbracket((\gamma, v), \delta) \\
\llbracket \Gamma; \Delta \vdash \exists \mathsf{x} : \tau.\ \mathsf{P} : \mathsf{Prop}_{\mathsf{TSO}} \rrbracket(\gamma, \delta) &= \bigcup_{v \in \llbracket \vdash \tau : \mathsf{Type} \rrbracket} \llbracket \Gamma, \mathsf{x} : \tau; \Delta \vdash \mathsf{P} : \mathsf{Prop}_{\mathsf{TSO}} \rrbracket((\gamma, v), \delta)
\end{aligned}$$

$$\begin{aligned}
\llbracket \Gamma; \Delta \vdash \ulcorner \mathsf{P} \urcorner : \mathsf{Prop}_{\mathsf{TSO}} \rrbracket(\gamma, \delta) &\stackrel{\text{def}}{=} \text{let } p = \llbracket \Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop}_{\mathsf{SC}} \rrbracket(\gamma, \delta) \text{ in } \ulcorner p \urcorner \\
\llbracket \Gamma; \Delta \vdash \ulcorner \mathsf{P}\ \mathbf{in}\ t \urcorner : \mathsf{Prop}_{\mathsf{TSO}} \rrbracket(\gamma, \delta) &\stackrel{\text{def}}{=} \text{let } p = \llbracket \Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop}_{\mathsf{SC}} \rrbracket(\gamma, \delta) \text{ in } \ulcorner p\ \mathbf{in}\ t \urcorner \\
\llbracket \Gamma; \Delta \vdash \mathsf{P}\, \mathcal{U}_t\, \mathsf{Q} : \mathsf{Prop}_{\mathsf{TSO}} \rrbracket(\gamma, \delta) &\stackrel{\text{def}}{=} \text{let } p = \llbracket \Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop}_{\mathsf{TSO}} \rrbracket(\gamma, \delta) \text{ in} \\
&\qquad \text{let } q = \llbracket \Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop}_{\mathsf{TSO}} \rrbracket(\gamma, \delta) \text{ in} \\
&\qquad\quad p\, \mathcal{U}_t\, q \\
\llbracket \Gamma; \Delta \vdash (\!|P|\!) : \mathsf{Prop}_{\mathsf{TSO}} \rrbracket(\gamma, \delta) &\stackrel{\text{def}}{=} \text{let } p = \llbracket \Gamma; \Delta \vdash P : \mathsf{Prop}_{\mathsf{TSO}} \rrbracket(\gamma, \delta) \text{ in } \mathrm{cl}(p) \\
\llbracket \Gamma; \Delta \vdash \mathsf{emp} : \mathsf{Prop}_{\mathsf{TSO}} \rrbracket(\gamma, \delta) &= \mathcal{M} \\
\llbracket \Gamma; \Delta \vdash \mathsf{P} * \mathsf{Q} : \mathsf{Prop}_{\mathsf{TSO}} \rrbracket(\gamma, \delta) &= \llbracket \Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop}_{\mathsf{TSO}} \rrbracket(\gamma, \delta) * \llbracket \Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop}_{\mathsf{TSO}} \rrbracket(\gamma, \delta)
\end{aligned}$$

4.5 SC Assertion Logic

$$
\begin{aligned}
[\![\Gamma;\Delta \vdash \bot : \mathsf{Prop_{sc}}]\!](\gamma,\delta) &= \emptyset \\
[\![\Gamma;\Delta \vdash \top : \mathsf{Prop_{sc}}]\!](\gamma,\delta) &= \mathcal{H} \\
[\![\Gamma;\Delta \vdash \mathsf{P} \wedge \mathsf{Q} : \mathsf{Prop_{sc}}]\!](\gamma,\delta) &= [\![\Gamma;\Delta \vdash \mathsf{P} : \mathsf{Prop_{sc}}]\!](\gamma,\delta) \cap [\![\Gamma;\Delta \vdash \mathsf{Q} : \mathsf{Prop_{sc}}]\!](\gamma,\delta) \\
[\![\Gamma;\Delta \vdash \mathsf{P} \vee \mathsf{Q} : \mathsf{Prop_{sc}}]\!](\gamma,\delta) &= [\![\Gamma;\Delta \vdash \mathsf{P} : \mathsf{Prop_{sc}}]\!](\gamma,\delta) \cup [\![\Gamma;\Delta \vdash \mathsf{Q} : \mathsf{Prop_{sc}}]\!](\gamma,\delta) \\
[\![\Gamma;\Delta \vdash \mathsf{P} \Rightarrow \mathsf{Q} : \mathsf{Prop_{sc}}]\!](\gamma,\delta) &= \{m \in \mathcal{H} \mid \forall n \geq m. \\
&\qquad n \in [\![\Gamma;\Delta \vdash \mathsf{P} : \mathsf{Prop_{sc}}]\!](\gamma,\delta) \;\Rightarrow \\
&\qquad n \in [\![\Gamma;\Delta \vdash \mathsf{Q} : \mathsf{Prop_{sc}}]\!](\gamma,\delta)\} \\
[\![\Gamma;\Delta \vdash \forall \mathsf{x} : \tau.\ \mathsf{P} : \mathsf{Prop_{sc}}]\!](\gamma,\delta) &= \bigcap_{v \in [\![\vdash \tau : \mathsf{Type}]\!]} [\![\Gamma, \mathsf{x} : \tau; \Delta \vdash \mathsf{P} : \mathsf{Prop_{sc}}]\!]((\gamma, v), \delta) \\
[\![\Gamma;\Delta \vdash \exists \mathsf{x} : \tau.\ \mathsf{P} : \mathsf{Prop_{sc}}]\!](\gamma,\delta) &= \bigcup_{v \in [\![\vdash \tau : \mathsf{Type}]\!]} [\![\Gamma, \mathsf{x} : \tau; \Delta \vdash \mathsf{P} : \mathsf{Prop_{sc}}]\!]((\gamma, v), \delta) \\
[\![\Gamma;\Delta \vdash \mathsf{emp} : \mathsf{Prop_{sc}}]\!](\gamma,\delta) &= \mathcal{H} \\
[\![\Gamma;\Delta \vdash \mathsf{P} * \mathsf{Q} : \mathsf{Prop_{sc}}]\!](\gamma,\delta) &= [\![\Gamma;\Delta \vdash \mathsf{P} : \mathsf{Prop_{sc}}]\!](\gamma,\delta) * [\![\Gamma;\Delta \vdash \mathsf{Q} : \mathsf{Prop_{sc}}]\!](\gamma,\delta) \\
[\![\Gamma;\Delta \vdash \mathsf{x.f} \mapsto \mathsf{v} : \mathsf{Prop_{sc}}]\!](\gamma,\delta) &= \text{let } o = [\![\Gamma;\Delta \vdash \mathsf{x} : \mathsf{Val}]\!](\gamma,\delta) \text{ in} \\
&\quad \text{let } f = [\![\Gamma;\Delta \vdash \mathsf{f} : \mathsf{Field}]\!](\gamma,\delta) \text{ in} \\
&\quad \text{let } v = [\![\Gamma;\Delta \vdash \mathsf{v} : \mathsf{Val}]\!](\gamma,\delta) \text{ in} \\
&\qquad \{m \in \mathcal{H} \mid m.l(o,f) = v\}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\Gamma;\Delta \vdash \mathsf{region}(\mathsf{M},\mathsf{N},\mathsf{R}) : \mathsf{Prop_{sc}}]\!](\gamma,\delta) = \\
&\quad \{m \in \mathcal{M} \mid \exists T : \Delta(\mathit{AId} \to \mathcal{P}(\mathit{SId} \times \mathit{SId})). \\
&\qquad \text{let } M = [\![\Gamma;\Delta \vdash \mathsf{M} : \mathcal{P}(\mathsf{SId})]\!](\gamma,\delta) \text{ in} \\
&\qquad \text{let } R = [\![\Gamma;\Delta \vdash \mathsf{R} : \mathsf{RId}]\!](\gamma,\delta) \text{ in} \\
&\qquad\quad \phi(T) = [\![\Gamma;\Delta \vdash \mathsf{T} : \mathsf{AId} \to \mathcal{P}(\mathsf{SId} \times \mathsf{SId})]\!](\gamma,\delta) \wedge \\
&\qquad\quad\ m \in \mathit{region}(M,T,R)\}
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma;\Delta \vdash \mathsf{rintr}(\mathsf{I},\mathsf{R}) : \mathsf{Prop_{sc}}]\!](\gamma,\delta) &= \mathit{rintr}([\![\Gamma;\Delta \vdash \mathsf{I} : \mathsf{SId} \to \mathsf{Prop_{tso}}]\!](\gamma,\delta), [\![\Gamma;\Delta \vdash \mathsf{R} : \mathsf{RId}]\!](\gamma,\delta)) \\
[\![\Gamma;\Delta \vdash [\mathsf{A}]^{\mathsf{R}}_{\mathsf{P}} : \mathsf{Prop_{sc}}]\!](\gamma,\delta) &= \mathit{action}([\![\Gamma;\Delta \vdash \mathsf{A} : \mathsf{AId}]\!](\gamma,\delta), \\
&\qquad [\![\Gamma;\Delta \vdash \mathsf{R} : \mathsf{RId}]\!](\gamma,\delta), \\
&\qquad [\![\Gamma;\Delta \vdash \mathsf{P} : \mathsf{Perm}]\!](\gamma,\delta))
\end{aligned}
$$

where ϕ embeds predicates from our set-theoretic metalogic in the topos of trees. For more details, consult the iCAP technical report [2].

$$
\begin{aligned}
[\![\Gamma \vdash \bot : \mathsf{Spec}]\!](\gamma) &= \bot \\
[\![\Gamma \vdash \top : \mathsf{Spec}]\!](\gamma) &= \top \\
[\![\Gamma \vdash \mathsf{S} \wedge \mathsf{T} : \mathsf{Spec}]\!](\gamma) &= [\![\Gamma \vdash \mathsf{S} : \mathsf{Spec}]\!](\gamma) \wedge [\![\Gamma \vdash \mathsf{T} : \mathsf{Spec}]\!](\gamma) \\
[\![\Gamma \vdash \mathsf{S} \vee \mathsf{T} : \mathsf{Spec}]\!](\gamma) &= [\![\Gamma \vdash \mathsf{S} : \mathsf{Spec}]\!](\gamma) \vee [\![\Gamma \vdash \mathsf{T} : \mathsf{Spec}]\!](\gamma) \\
[\![\Gamma \vdash \mathsf{S} \Rightarrow \mathsf{T} : \mathsf{Spec}]\!](\gamma) &= [\![\Gamma \vdash \mathsf{S} : \mathsf{Spec}]\!](\gamma) \Rightarrow [\![\Gamma \vdash \mathsf{T} : \mathsf{Spec}]\!](\gamma)
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma \vdash \forall \mathsf{x} : \tau.\ \mathsf{S} : \mathsf{Spec}]\!](\gamma) &= \forall v \in [\![\vdash \tau : \mathsf{Type}]\!].\ [\![\Gamma, \mathsf{x} : \tau \vdash \mathsf{S} : \mathsf{Spec}]\!]((\gamma, v)) \\
[\![\Gamma \vdash \exists \mathsf{x} : \tau.\ \mathsf{S} : \mathsf{Spec}]\!](\gamma) &= \exists v \in [\![\vdash \tau : \mathsf{Type}]\!].\ [\![\Gamma, \mathsf{x} : \tau \vdash \mathsf{S} : \mathsf{Spec}]\!]((\gamma, v)) \\
[\![\Gamma \vdash \mathsf{M} =_\tau \mathsf{N} : \mathsf{Spec}]\!](\gamma) &= [\![\Gamma \vdash \mathsf{M} : \tau]\!](\gamma) = [\![\Gamma \vdash \mathsf{N} : \tau]\!](\gamma)
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma \vdash (\Delta).\{\mathsf{P}\}\ \mathsf{e}\ \{\mathsf{Q}\}]\!](\gamma) \stackrel{\text{def}}{=} \forall t \in \mathit{TId}.\ \forall \delta \in [\![\Delta]\!].\ & \\
\text{let } p = [\![\Gamma; \Delta \vdash \mathsf{P} : \mathsf{TId} \to \mathsf{Prop}_{\mathsf{TSO}}]\!](\gamma, \delta) \text{ in} & \\
\text{let } q = [\![\Gamma; \Delta \vdash \mathsf{Q} : \mathsf{TId} \to \mathsf{Val} \to \mathsf{Prop}_{\mathsf{TSO}}]\!](\gamma, \delta) \text{ in} & \\
\quad \mathrm{safe}((t, \delta(\mathsf{e})), p(t), q(t)) &
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma \vdash (\Delta).\langle \mathsf{P} \rangle\ \mathsf{e}\ \langle \mathsf{Q} \rangle^A]\!](\gamma) \stackrel{\text{def}}{=} \forall t \in \mathit{TId}.\ \forall \delta \in [\![\Delta]\!].\ \forall a \in \mathit{Act}.\ \forall v \in \mathit{Val}. & \\
\text{let } p = [\![\Gamma; \Delta \vdash \mathsf{P} : \mathsf{TId} \to \mathsf{Prop}_{\mathsf{TSO}}]\!](\gamma, \delta) \text{ in} & \\
\text{let } q = [\![\Gamma; \Delta \vdash \mathsf{Q} : \mathsf{TId} \to \mathsf{Val} \to \mathsf{Prop}_{\mathsf{TSO}}]\!](\gamma, \delta) \text{ in} & \\
\quad (t, \delta(e)) \xrightarrow{a} \{(t, v)\} \Rightarrow a\ \mathrm{sat}_A\ \{p(t)\}\,\{q(t)(v)\} &
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma \vdash \mathsf{r{-}stable(P)} : \mathsf{Spec}]\!](\gamma) &= \text{let } p = [\![\Gamma; \varepsilon \vdash \mathsf{P} : \mathsf{Prop}_{\mathsf{SC}}]\!](\gamma, \varepsilon) \text{ in } \mathrm{rstable}(p) \\
[\![\Gamma \vdash \mathsf{b{-}stable(P)} : \mathsf{Spec}]\!](\gamma) &= \text{let } p = [\![\Gamma; \varepsilon \vdash \mathsf{P} : \mathsf{Prop}_{\mathsf{TSO}}]\!](\gamma, \varepsilon) \text{ in } \mathrm{bstable}(p) \\
[\![\Gamma \vdash \mathsf{P} \sqsubseteq^{\mathsf{R}} \mathsf{Q} : \mathsf{Spec}]\!](\gamma) &= \text{let } p = [\![\Gamma; \varepsilon \vdash \mathsf{P} : \mathsf{Prop}_{\mathsf{TSO}}]\!](\gamma, \varepsilon) \text{ in} \\
&\quad \text{let } q = [\![\Gamma; \varepsilon \vdash \mathsf{Q} : \mathsf{Prop}_{\mathsf{TSO}}]\!](\gamma, \varepsilon) \text{ in} \\
&\quad \text{let } R = [\![\Gamma \vdash \mathsf{R} : \mathcal{P}(\mathsf{RId})]\!](\gamma) \text{ in} \\
&\quad\quad p \sqsubseteq_R q
\end{aligned}
$$

4.7 Judgments

$$
\begin{aligned}
[\![\Gamma;\Delta \mid \mathsf{P} \vdash_w \mathsf{x.f} \mapsto \mathsf{v}, \mathsf{Q}]\!](\gamma,\delta) = {} & \text{let } p = [\![\Gamma;\Delta \vdash \mathsf{P} : \mathit{TId} \to \mathsf{Prop_{TSO}}]\!](\gamma,\delta) \text{ in} \\
& \text{let } q = [\![\Gamma;\Delta \vdash \mathsf{Q} : \mathsf{Prop_{TSO}}]\!](\gamma,\delta) \text{ in} \\
& \text{let } o = [\![\Gamma;\Delta \vdash \mathsf{x} : \mathsf{Val}]\!](\gamma,\delta) \text{ in} \\
& \text{let } v = [\![\Gamma;\Delta \vdash \mathsf{v} : \mathsf{Val}]\!](\gamma,\delta) \text{ in} \\
& p \vdash_{\mathrm{w}} o.\mathsf{f} \mapsto v, q \\
[\![\Gamma;\Delta \mid \mathsf{P} \vdash_r^m \mathsf{x.f} \mapsto \mathsf{v}]\!](\gamma,\delta) = {} & \text{let } p = [\![\Gamma;\Delta \vdash \mathsf{P} : \mathsf{Prop_{TSO}}]\!](\gamma,\delta) \text{ in} \\
& \text{let } o = [\![\Gamma;\Delta \vdash \mathsf{x} : \mathsf{Val}]\!](\gamma,\delta) \text{ in} \\
& \text{let } v = [\![\Gamma;\Delta \vdash \mathsf{v} : \mathsf{Val}]\!](\gamma,\delta) \text{ in} \\
& p \vdash_{\mathrm{r}}^{\mathrm{m}} o.\mathsf{f} \mapsto v \\
[\![\Gamma;\Delta \mid p \vdash_r^o \mathsf{x.f} \mapsto \mathsf{v}]\!](\gamma,\delta) = {} & \text{let } p = [\![\Gamma;\Delta \vdash \mathsf{P} : \mathsf{Prop_{TSO}}]\!](\gamma,\delta) \text{ in} \\
& \text{let } o = [\![\Gamma;\Delta \vdash \mathsf{x} : \mathsf{Val}]\!](\gamma,\delta) \text{ in} \\
& \text{let } v = [\![\Gamma;\Delta \vdash \mathsf{v} : \mathsf{Val}]\!](\gamma,\delta) \text{ in} \\
& p \vdash_{\mathrm{r}}^{\mathrm{o}} o.\mathsf{f} \mapsto v
\end{aligned}
$$

4.8 Embeddings and later operators

$$
\begin{aligned}
[\![\Gamma \vdash \mathsf{valid(P)} : \mathsf{Spec}]\!](\gamma) &= \mathrm{valid}([\![\Gamma; - \vdash \mathsf{P} : \mathsf{Prop_{TSO}}]\!](\gamma)) \\
[\![\Gamma \vdash \triangleright \mathsf{S} : \mathsf{Spec}]\!](\gamma) &= \triangleright([\![\Gamma \vdash \mathsf{S} : \mathsf{Spec}]\!](\gamma))
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma;\Delta \vdash \mathsf{spec(S)} : \mathsf{Prop_{sc}}]\!](\gamma,\delta) &= \mathit{spec}([\![\Gamma \vdash \mathsf{S} : \mathsf{Spec}]\!](\gamma)) \\
[\![\Gamma;\Delta \vdash \triangleright \mathsf{P} : \mathsf{Prop_{TSO}}]\!](\gamma,\delta) &= \triangleright([\![\Gamma;\Delta \vdash \mathsf{P} : \mathsf{Prop_{TSO}}]\!](\gamma,\delta)) \\
[\![\Gamma;\Delta \vdash \triangleright \mathsf{P} : \mathsf{Prop_{sc}}]\!](\gamma,\delta) &= \triangleright([\![\Gamma;\Delta \vdash \mathsf{P} : \mathsf{Prop_{sc}}]\!](\gamma,\delta))
\end{aligned}
$$

4.9 Entailment

$$
\begin{aligned}
&[\![\Gamma;\Delta \mid \Phi \mid \mathsf{P} \vdash_{TSO} \mathsf{Q}]\!] = \\
&\quad \forall \gamma \in [\![\Gamma]\!].\ \forall \delta \in [\![\Delta]\!].\ [\![\Phi]\!](\gamma) \Rightarrow [\![\Gamma;\Delta \vdash \mathsf{P} : \mathsf{Prop_{TSO}}]\!](\gamma,\delta) \subseteq [\![\Gamma;\Delta \vdash \mathsf{Q} : \mathsf{Prop_{TSO}}]\!](\gamma,\delta)
\end{aligned}
$$

$$
\begin{aligned}
&[\![\Gamma;\Delta \mid \Phi \mid \mathsf{P} \vdash_{SC} \mathsf{Q}]\!] = \\
&\quad \forall \gamma \in [\![\Gamma]\!].\ \forall \delta \in [\![\Delta]\!].\ [\![\Phi]\!](\gamma) \Rightarrow [\![\Gamma;\Delta \vdash \mathsf{P} : \mathsf{Prop_{sc}}]\!](\gamma,\delta) \subseteq [\![\Gamma;\Delta \vdash \mathsf{Q} : \mathsf{Prop_{sc}}]\!](\gamma,\delta)
\end{aligned}
$$

$$
[\![\Gamma \mid \mathsf{S}_1, ..., \mathsf{S}_n \vdash \mathsf{T}]\!] = \forall \gamma \in [\![\Gamma]\!].\ \left(\bigwedge_{i \in \{1,...,n\}} [\![\Gamma \vdash \mathsf{S}_i : \mathsf{Spec}]\!](\gamma) \right) \Rightarrow [\![\Gamma \vdash \mathsf{T} : \mathsf{Spec}]\!](\gamma)
$$

5 Soundness

Lemma 32.

$$\forall p \in \mathit{Prop}_{SC}.\, \triangleright \ulcorner p \urcorner = \ulcorner \triangleright p \urcorner$$

Proof ($\subseteq$). Assume $\triangleright(l, s, U, \zeta) \in \ulcorner p \urcorner$. Then by definition of the embedding,

$$\triangleright(\exists l' \in \Delta(\mathit{LState}).\, l' \leq l \wedge (l', s, U, \zeta) \in p \wedge \mathrm{lfd}(l', U))$$

Since later commutes over existentials over constant sets and conjunctions and $l' \leq l$ and lfd simply reduces to (universally quantified) equalities on constant sets, which are upwards-closed, the above is equivalent to:

$$\exists l' \in \Delta(\mathit{LState}).\, (l' \leq l \vee \triangleright\bot) \wedge (l', s, U, \zeta) \in \triangleright p \wedge (\mathrm{lfd}(l', U) \vee \triangleright\bot))$$

In case $\triangleright\bot$ then $(l, s, U, \zeta) \in \ulcorner \triangleright p \urcorner$ holds trivially, by taking l' to be the empty local state, which is trivially smaller than l. Otherwise, the conclusion follows directly from the assumptions. □

Proof ($\supseteq$). Follows easily by monotonicity of $\triangleright$. □

Lemma 33.

$$\forall p, q \in \mathit{Prop}_{TSO}.\ \triangleright\,(p\,\mathcal{U}_t\,q) = (\triangleright p)\,\mathcal{U}_t\,(\triangleright q)$$

Proof ($\subseteq$). Assume $(l, s, U, \zeta) \in \triangleright(p\,\mathcal{U}_t\,q)$. Hence, by the definition of $\mathcal{U}_t$

$$\begin{aligned}\triangleright(\ &\exists \alpha, \beta \in \Delta(\mathit{SBuffer}).\ \exists o \in \Delta(\mathit{OId}).\ \exists f \in \Delta(\mathit{FName}).\ \exists v \in \Delta(\mathit{Val}).\\ &U(t) = \alpha \cdot (o, f, v) \cdot \beta \wedge (l, s, U[t \mapsto \alpha], \zeta) \in p \wedge\\ &(\mathrm{flush}(l, \alpha \cdot (o, f, v)), s, U[t \mapsto \beta], \zeta) \in q)\end{aligned}$$

Since constant sets are trivially total and $\triangleright$ commutes over $\wedge$ this reduces to

$$\begin{aligned}&\exists \alpha, \beta \in \Delta(\mathit{SBuffer}).\ \exists o \in \Delta(\mathit{OId}).\ \exists f \in \Delta(\mathit{FName}).\ \exists v \in \Delta(\mathit{Val}).\\ &\quad \triangleright\,(U(t) = \alpha \cdot (o, f, v) \cdot \beta) \wedge (l, s, U[t \mapsto \alpha], \zeta) \in \triangleright p \wedge\\ &\quad (\mathrm{flush}(l, \alpha \cdot (o, f, v)), s, U[t \mapsto \beta], \zeta) \in \triangleright q\end{aligned}$$

Since equality on constant sets is upwards-closed, this is equivalent to

$$\begin{aligned}&\exists \alpha, \beta \in \Delta(\mathit{SBuffer}).\ \exists o \in \Delta(\mathit{OId}).\ \exists f \in \Delta(\mathit{FName}).\ \exists v \in \Delta(\mathit{Val}).\\ &\quad U(t) = \alpha \cdot (o, f, v) \cdot \beta \wedge (l, s, U[t \mapsto \alpha], \zeta) \in \triangleright p \wedge\\ &\quad (\mathrm{flush}(l, \alpha \cdot (o, f, v)), s, U[t \mapsto \beta], \zeta) \in \triangleright q\end{aligned}$$

Hence $(l, s, U, \zeta) \in (\triangleright p)\,\mathcal{U}_t\,(\triangleright q)$. □

Proof ($\supseteq$). Assume $(l, s, U, \zeta) \in (\triangleright p)\, \mathcal{U}_t\, (\triangleright q)$. Then

$$\begin{aligned}&\exists \alpha, \beta \in \Delta(\mathit{SBuffer}).\ \exists o \in \Delta(\mathit{OId}).\ \exists f \in \Delta(\mathit{FName}).\ \exists v \in \Delta(\mathit{Val}).\\ &\quad U(t) = \alpha \cdot (o, f, v) \cdot \beta \wedge (l, s, U[t \mapsto \alpha], \zeta) \in \triangleright p \wedge\\ &\quad (\mathrm{flush}(l, \alpha \cdot (o, f, v)), s, U[t \mapsto \beta], \zeta) \in \triangleright q\end{aligned}$$

by monotonicity of $\triangleright$ and by commuting $\triangleright$ over $\wedge$ and $\exists$, it follows that

$$\begin{aligned}&\triangleright(\ \exists \alpha, \beta \in \Delta(\mathit{SBuffer}).\ \exists o \in \Delta(\mathit{OId}).\ \exists f \in \Delta(\mathit{FName}).\ \exists v \in \Delta(\mathit{Val}).\\ &\quad U(t) = \alpha \cdot (o, f, v) \cdot \beta \wedge (l, s, U[t \mapsto \alpha], \zeta) \in p \wedge\\ &\quad (\mathrm{flush}(l, \alpha \cdot (o, f, v)), s, U[t \mapsto \beta], \zeta) \in q)\end{aligned}$$

Thus $(l, s, U, \zeta) \in \triangleright(p\, \mathcal{U}_t\, q)$. □

Lemma 34. *For any $A \in \mathcal{P}(\Delta(\mathit{RId}))$, $p \in \Delta(\mathit{TId}) \to \mathit{Prop}_{\mathit{TSO}}$, $f \in \Delta(\mathit{FName})$, $v_1, v_2 \in \Delta(\mathit{Val})$, $t \in \Delta(\mathit{TId})$ and $o \in \Delta(\mathit{OId})$,*

$$p \vdash^{\mathrm{m}}_{\mathrm{r}} o.f \mapsto v_1 \Rightarrow \mathit{read}(t, o, f, v_2)\ \mathrm{sat}_A\ \{p(t)\}\,\{p(t) * v_1 = v_2\}$$

Proof. Let $r \in \mathit{Prop}_{\mathit{TSO}}$, $m \in \mathcal{M}$, $h \in \Delta(\mathit{Heap})$, and $U \in \Delta(\mathit{SPool})$ such that

$$m \in p(t) * \triangleright r \qquad\qquad (h, U) \in \lfloor m \rfloor_A \qquad\qquad \operatorname*{stable}_A(r).$$

Hence, there exists $m_1, m_2 \in \mathcal{M}$ such that $m = m_1 \bullet m_2$, $m_1 \in p(t)$ and $m_2 \in \triangleright r$. If $t \notin \mathrm{dom}(U)$, $[\![\mathit{read}(t, o, f, v_2)]\!](h, U) = \emptyset$ and the conclusion holds vacuously. Assume, then, that $t \in \mathrm{dom}(U)$. Now, from the definition of $\vdash^{\mathrm{m}}_{\mathrm{r}}$ it follows that

$$\triangleright \mathrm{lookup}(o.f, m_1.U(t), m_1.l) = v_1.$$

Since equalities on terms of constant sets are upwards-closed it follows that

$$\mathrm{lookup}(o.f, m_1.U(t), m_1.l) = v_1 \vee \triangleright \bot$$

The $\triangleright\bot$ case follows trivially as the definition of atomic satisfaction only requires the post-condition to hold later. In the $\mathrm{lookup}(o.f, m_1.U(t), m_1.l) = v_1$ case, $o.f \in \mathrm{dom}\, h$, and so $\lightning \notin [\![\mathit{read}(t, o, f, v_2)]\!](h, U)$. Now we can take any $(h', U') \in [\![\mathit{read}(t, o, f, v_2)]\!](h, U)$. By definition, it follows that $h = h'$, $U = U'$, and $v_1 = v_2$. We pick m' to be m and thus have to show that

$$\triangleright(m \in p(t) * v_1 = v_1 * r) \qquad\qquad \triangleright((h, U) \in \lfloor m \rfloor_A)$$

Both follow easily from monotonicity of $\triangleright$ and the fact that $\varepsilon \in v_1 = v_1$. □

Lemma 35. *For any $A \in \mathcal{P}(\Delta(\mathit{RId}))$, $p \in \Delta(\mathit{TId}) \to \mathit{Prop}_{\mathit{TSO}}$, $q \in \Delta(\mathit{TId}) \to \mathit{Prop}_{\mathit{SC}}$, $f \in \Delta(\mathit{FName})$, $v_1, v_2 \in \Delta(\mathit{Val})$, $t \in \Delta(\mathit{TId})$ and $o \in \Delta(\mathit{OId})$,*

$$p \vdash_{\mathrm{w}} o.f \mapsto v_2, q \Rightarrow \mathit{write}(t, o, f, v_1)\ \mathrm{sat}_A\ \{p(t)\}\,\{p(t)\, \mathcal{U}_t\, (\ulcorner q(t) * o.f \mapsto v_1 \urcorner)\}$$

Proof. Let $r \in \mathit{Prop}_{TSO}$, $m \in \mathcal{M}$, $h \in \Delta(\mathit{Heap})$ and $U \in \Delta(\mathit{SPool})$ such that

$$m \in p(t) * \triangleright r \qquad (h, U) \in \lfloor m \rfloor_A \qquad \operatorname*{stable}_A(r)$$

Hence, there exists $m_1, m_2 \in \mathcal{M}$ such that $m = m_1 \bullet m_2$, $m_1 \in p(t)$ and $m_2 \in \triangleright r$. Let

$$m = (l, s, U, \zeta) \qquad m_1 = (l_1, s, U, \zeta) \qquad m_2 = (l_2, s, U, \zeta)$$

Analogously to the read case, we only need to consider the case when $t \in \mathrm{dom}(U)$. From the definition of $\vdash_{\mathrm{w}}$ it follows that

$$\triangleright \mathrm{lookup}(o.f, U(t), l_1) = v_2 \quad \triangleright(\mathrm{flush}(l_1, U(t)), s_1, U[t \mapsto \varepsilon], \zeta_1) \in \ulcorner q(t) * o.f \mapsto v_2 \urcorner.$$

Since equalities on constant sets are upwwards-closed, it follows that

$$\mathrm{lookup}(o.f, U(t), l_1) = v_2 \vee \triangleright \bot$$

As the $\triangleright\bot$ case follows easily from Lemma 15, assume $\mathrm{lookup}(o.f, U(t), l_1) = v_2$. Then $(o, f) \in \mathrm{dom}(h)$ and thus $\lightning \notin [\![\mathit{write}(t, o, f, v_1)]\!](h, U)$. Now, we can take any $(h', U') \in [\![\mathit{write}(t, o, f, v_1)]\!](h, U)$. By definition of action semantics, this means that $h' = h$ and $U' = U[t \mapsto U(t) \cdot (o, f, v_1)]$.

Take $m' = (l, s, U', \zeta)$. Since $(o, f) \in \mathrm{dom}(l_1)$, $\triangleright((l_2, s, U', \zeta) \in r)$ and $(h, U') \in \lfloor m' \rfloor_A$ follow by stability of r and interpretations of the shared regions. Hence, it suffices to show that $\triangleright(l_1, s, U', \zeta) \in p(t)\ \mathcal{U}_t\ (\ulcorner q(t) * o.f \mapsto v_1 \urcorner)$. To show this, we pick the final update in $U'(t)$, and thus need to show

$$\triangleright(l_1, s, U, \zeta) \in p(t) \quad \triangleright(\mathrm{flush}(l_1, U(t) \cdot (o, f, v_1)), s, U[t \mapsto \varepsilon], \zeta) \in \ulcorner q(t) * o.f \mapsto v_1 \urcorner$$

The first property follows from our assumptions and monotonicity of later; the second is more involved. Since the interpretation of Prop_{SC} distributes over separating conjunction, we know there exist $l_3, l_4 \in \Delta(\mathit{LState})$, such that $\triangleright l_3 \bullet l_4 = \mathrm{flush}(l_1, U(t))$, $\triangleright(l_3, s_1, U[t \mapsto \varepsilon], \zeta) \in \ulcorner q(t) \urcorner$ and $\triangleright(l_4, s_1, U[t \mapsto \varepsilon], \zeta) \in \ulcorner o.f \mapsto v_2 \urcorner$. We can now show, using the properties of flush, that

$$\mathrm{flush}(l_1, U(t) \cdot (o, f, v_1)) = \mathrm{flush}(l_3, (o, f, v_1)) \bullet \mathrm{flush}(l_4, (o, f, v_1)).$$

Since $\triangleright(o, f) \in \mathrm{dom}(l_4)$ has to hold, we can also conclude easily that

$$\triangleright(\mathrm{flush}(l_4, (o, f, v_1)), s, U[t \mapsto \varepsilon], \zeta) \in \ulcorner o.f \mapsto v_1 \urcorner.$$

However, since $\triangleright(o, f) \notin \mathrm{dom}(l_3)$, $\triangleright(l_3, U[t \mapsto \varepsilon])\ R_{sb}\ (\mathrm{flush}(l_3, (o, f, v_1)), U[t \mapsto \varepsilon])$, and by Lemma 30 and monotonicity of later, $\ulcorner q(t) \urcorner$ is stable under store-buffer interference later, which concludes the proof. □

Lemma 36. *For any $A \in \mathcal{P}(\Delta(\mathit{RId}))$, $p \in \Delta(\mathit{TId}) \to \mathit{Prop}_{TSO}$, $t \in \Delta(\mathit{TId})$, $o \in \Delta(\mathit{OId})$, $f \in \Delta(\mathit{FName})$, $v, v_o, v_n \in \Delta(\mathit{Val})$, $x \in \Delta(\mathbf{2})$,*

$$p \vdash^{\mathrm{m}}_{\mathrm{r}} o.f \mapsto v \wedge v \neq v_o \Rightarrow \mathit{cas}(t, o, f, v_o, v_n, x)\ \mathrm{sat}_A\ \{p(t)\}\,\{p(t) * x = \textbf{\textit{false}}\}$$

Proof. Let $r \in \mathit{Prop}_{TSO}$, $m \in \mathcal{M}$, $h \in \Delta(\mathit{Heap})$, and $U \in \Delta(\mathit{SPool})$ such that

$$m \in p(t) * \triangleright r \qquad (h, U) \in \lfloor m \rfloor_A \qquad \underset{A}{\mathrm{stable}}(r).$$

Hence, there exists $m_1, m_2 \in \mathcal{M}$ such that $m = m_1 \bullet m_2$, $m_1 \in p(t)$ and $m_2 \in \triangleright r$. As with read and write actions, if $t \notin \mathrm{dom}(U)$, the statement is vacuously true, since in that case $[\![\mathit{cas}(t, o, f, v_o, v_n, x)]\!] = \emptyset$. Thus, we know that $t \in \mathrm{dom}(U)$, and so, by definition of $\vdash_{\mathrm{r}}^{\mathrm{m}}$, we get $\triangleright \mathrm{lookup}(o.f, U(t), m_1.l) = v$. As equalities on constant sets are upwards-closed, it follows that

$$\mathrm{lookup}(o.f, U(t), m_1.l) = v \vee \triangleright \bot$$

As the conclusion follows trivially in the $\triangleright\bot$ case (by Lemma 15), assume

$$\mathrm{lookup}(o.f, U(t), m_1.l) = v.$$

This means that $(o, f) \in \mathrm{dom}(m_1.l) \subseteq \mathrm{dom}(h)$, and so $\lightning \notin [\![\mathit{cas}(t, o, f, v_o, v_n, x)]\!]$.

Take any $(h', U') \in [\![\mathit{cas}(t, o, f, v_o, v_n, x)]\!](h, U)$. Since $v \neq v_o$, there is only one non-vacuous case, in which we learn that $h' = h$, $U' = U$ and $x = \mathsf{false}$. Thus, we pick $m' = m$, so $(h, U) \in \lfloor m \rfloor_A$ holds by assumption. To show that

$$\triangleright(m \in p(t) * \mathsf{false} = \mathsf{false} * r),$$

we pick the same splitting of m, which gives us $m_2 \in \triangleright r$. Now we can split m_1 as $m_1 \bullet (\varepsilon, m.s, m.U, m.\zeta)$, and both branches follow easily by monotonicity of $\triangleright$. □

Lemma 37. *For any $A \in \mathcal{P}(\Delta(\mathit{RId}))$, $p \in \Delta(\mathit{TId}) \to \mathit{Prop}_{TSO}$, $q \in \Delta(\mathit{TId}) \to \mathit{Prop}_{SC}$, $t \in \Delta(\mathit{TId})$, $o \in \Delta(\mathit{OId})$, $f \in \Delta(\mathit{FName})$, $v_o, v_n \in \Delta(\mathit{Val})$, $x \in \Delta(\mathbf{2})$,*

$$p \vdash_{\mathrm{w}} o.f \mapsto v_o, q \Rightarrow \mathit{cas}(t, o, f, v_o, v_n, x) \text{ sat}_A \ \{p(t)\}\,\{\ulcorner q(t) * o.f \mapsto v_n \urcorner * x = \mathit{true}\}$$

Proof. Let $r \in \mathit{Prop}_{TSO}$, $m \in \mathcal{M}$, $h \in \Delta(\mathit{Heap})$ and $U \in \Delta(\mathit{SPool})$ such that

$$m \in p(t) * \triangleright r \qquad (h, U) \in \lfloor m \rfloor_A \qquad \underset{A}{\mathrm{stable}}(r)$$

Hence, there exists $m_1, m_2 \in \mathcal{M}$ such that $m = m_1 \bullet m_2$, $m_1 \in p(t)$ and $m_2 \in \triangleright r$. Let

$$m = (l, s, U, \zeta) \qquad m_1 = (l_1, s, U, \zeta) \qquad m_2 = (l_2, s, U, \zeta)$$

As in the other cases, if $t \notin \mathrm{dom}(U)$, the statement holds vacuously. In the case when $t \in \mathrm{dom}(U)$, by definition of $\vdash_{\mathrm{w}}$ we get $\triangleright \mathrm{lookup}(o.f, U(t), l_1) = v_o$ and

$$\triangleright(\mathrm{flush}(l_1, U(t)), s, U[t \mapsto \varepsilon], \zeta) \in \ulcorner q(t) * o.f \mapsto v_o \urcorner.$$

Since equalities on constant sets are upwards-closed it follows that

$$\mathrm{lookup}(o.f, U(t), l_1) = v_o \vee \triangleright \bot$$

As the conclusion follows trivially by Lemma 15 in the $\triangleright\bot$ case, assume $\mathrm{lookup}(o.f, U(t), l_1) = v_o$. Since $(o, f) \in \mathrm{dom}(l_1) \subseteq \mathrm{dom}(h)$, we can conclude that $\lightning \notin [\![cas(t, o, f, v_o, v_n, x)]\!]$, which leaves us with the second conjunct.

Take any $(h', U') \in [\![cas(t, o, f, v_o, v_n, x)]\!](h, U)$. The only non-vacuous case is the successful compare-and-swap action, in which case $x = \mathbf{true}$, $U' = U[t \mapsto \varepsilon]$, and $h' = \mathrm{flush}(h, U(t) \cdot (o, f, v_n))$. Pick $m' = (\mathrm{flush}(l, U(t) \cdot (o, f, v_n)), s, U[t \mapsto \varepsilon], \zeta)$. Since flushing distributes over $\bullet_{LState}$, we can easily show that

$$(\mathrm{flush}(l_2, U(t) \cdot (o, f, v_n)), s, U[t \mapsto \varepsilon], \zeta) \in \triangleright(r)$$

by using stability of the latter assertion. By using the same argument as in the case of write, and Lemma 30, we can also show

$$\triangleright(\mathrm{flush}(l_1, U(t) \cdot (o, f, v_n)), s, U[t \mapsto \varepsilon], \zeta) \in \ulcorner q(t) * o.f \mapsto v_n \urcorner * \mathbf{true} = \mathbf{true}.$$

What remains is showing that $(h', U') \in \lfloor m' \rfloor_A$. However, we can pick the state $l'_n = \mathrm{flush}(l', U(t) \cdot (o, f, v))$ and the map $sr_n(r) = (\mathrm{flush}(\pi_1(sr(r)), U(t) \cdot (o, f, v)), U')$, where l' and sr are the witnesses of $(h, U) \in \lfloor m \rfloor_A$. This leaves us with showing $sr_n(r) \in \lfloor (s, \zeta) \rfloor_r$ for $r \in \mathrm{dom}\, s \cap A$. However, since $(o, f) \notin \mathrm{dom}(\pi_1(sr(r)))$, we know that $sr(r)\ R_{sb}\ sr_n(r)$. Also, since $\lfloor (s, \zeta) \rfloor_r$ is defined as application of the action model, it's upwards-closed under interference, and so $sr_n(r) \in \lfloor (s, \zeta) \rfloor_r$. □

5.1 Read and Write Judgments

Lemma 38 (W-Ax-Sound)**.** *For any $o \in \Delta(OId)$, $f \in \Delta(FName)$, $v \in \Delta(Val)$*

$$(\lambda t \in TId.\ \triangleright \ulcorner o.f \mapsto v\ \mathbf{in}\ t \urcorner) \vdash_{\mathrm{w}} o.f \mapsto v, \top.$$

Proof. Take any $t \in \Delta(TId)$ and $\triangleright(l, s, U, \zeta) \in \ulcorner o.f \mapsto v\ \mathbf{in}\ t \urcorner$ such that $t \in \mathrm{dom}(U)$. By definition, this means that $\triangleright(\mathrm{flush}(l, U(t)), s, U[t \mapsto \varepsilon], \zeta) \in \ulcorner o.f \mapsto v \urcorner$. This implies both that

$$\triangleright(\mathrm{flush}(l, U(t)), s, U[t \mapsto \varepsilon], \zeta) \in \ulcorner \top * o.f \mapsto v \urcorner$$

and, by definition of points-to predicate, that $\triangleright(\mathrm{flush}(l, U(t))).h(x, f) = v$, which means that

$$\triangleright \mathrm{lookup}(x.f, U(t), l) = v,$$

which in turn ends the proof. □

Lemma 39 (W-*-Sound)**.** *For any $p \in \Delta(TId) \to Prop_{TSO}$, $q \in Prop_{SC}$, $r \in Prop_{SC}$, $o \in \Delta(OId)$, $f \in \Delta(FName)$, $v \in \Delta(Val)$*

$$p \vdash_{\mathrm{w}} o.f \mapsto v, r \Rightarrow (\lambda t \in TId.\ p(t) * \triangleright \ulcorner q\ \mathbf{in}\ t \urcorner) \vdash_{\mathrm{w}} o.f \mapsto v, q * r.$$

Proof. Take any $t \in \Delta(TId)$ and $m \in p(t) * \triangleright \ulcorner q\ \mathbf{in}\ t \urcorner$ such that $t \in \mathrm{dom}(m.U)$. By definition, there exist states m_1 and m_2 such that $m = m_1 \bullet m_2$, $m_1 \in p(t)$ and $m_2 \in \triangleright \ulcorner q\ \mathbf{in}\ t \urcorner$. Let $(l_1, s, U, \zeta) = m_1$ and $(l_2, s, U, \zeta) = m_2$. From the assumption we get that

$\triangleright\text{lookup}(o.f, U(t), l_1) = v$ and $\triangleright(\text{flush}(l_1, U(t)), s, U[t \mapsto \varepsilon], \zeta) \in \ulcorner r * o.f \mapsto v \urcorner$. Since $m.l = l_1 \bullet_{LState} l_2$, this means that $\triangleright\text{lookup}(o.f, m.U(t), m.l) = v$. For the remaining property, note that

$$\text{flush}(l_1 \bullet_{LState} l_2, \alpha) = \text{flush}(l_1, \alpha) \bullet_{LState} \text{flush}(l_2, \alpha),$$

so it suffices to show that $\triangleright(\text{flush}(l_2, U(t)), s, U[t \mapsto \varepsilon], \zeta) \in \ulcorner q \urcorner$, which follows directly from $m_2 \in \triangleright \ulcorner q \textbf{ in } t \urcorner$. □

Lemma 40 (W-$\mathcal{U}$-Sound)**.** *For any $p \in \Delta(TId) \to Prop_{TSO}$, $q \in \Delta(TId) \to Prop_{TSO}$, $r \in Prop_{SC}$, $o \in \Delta(OId)$, $f \in \Delta(FName)$, $v \in \Delta(Val)$,*

$$q \vdash_{\text{w}} o.f \mapsto v, r \Rightarrow (\lambda t \in TId.\ p(t)\ \mathcal{U}_t\ q(t)) \vdash_{\text{w}} o.f \mapsto v, r.$$

Proof. Take any $t \in \Delta(TId)$ and $(l, s, U, \zeta) \in p(t)\ \mathcal{U}_t\ q(t)$ such that $t \in \text{dom}(U)$. By definition of $\mathcal{U}_t$, this means that there exist α and β such that $U(t) = \alpha \cdot \beta$ and $(\text{flush}(l, \alpha), s, U[t \mapsto \beta], \zeta) \in q(t)$ (we fold the update from the definition into α, since it's not of interest in this proof). By instantiating the assumption with the latter, we get that $\triangleright\text{lookup}(o.f, \beta, \text{flush}(l, \alpha)) = v$ and $\triangleright(\text{flush}(\text{flush}(l, \alpha), \beta), s, U[t \mapsto \varepsilon], \zeta) \in \ulcorner r * o.f \mapsto v \urcorner$. By the properties of flush and lookup, this means that $\triangleright\text{lookup}(o.f, \alpha \cdot \beta, l) = v$ and

$$\triangleright(\text{flush}(l, \alpha \cdot \beta), s, U[t \mapsto \varepsilon], \zeta) \in \ulcorner r * o.f \mapsto v \urcorner,$$

which ends the proof. □

Lemma 41 (W-Cons-Sound)**.** *For any $p \in \Delta(TId) \to Prop_{TSO}$, $q \in \Delta(TId) \to Prop_{TSO}$, $r \in Prop_{SC}$, $o \in \Delta(OId)$, $f \in \Delta(FName)$, $v \in \Delta(Val)$,*

$$q \vdash_{\text{w}} o.f \mapsto v, r \wedge (\forall t \in TId.\ p(t) \subseteq q(t)) \Rightarrow p \vdash_{\text{w}} o.f \mapsto v, r.$$

Proof. Take any $t \in \Delta(TId)$ and $m \in p(t)$, such that $t \in \text{dom}(m.U)$. By the entailment assumption, this means $m \in q(t)$, so $\triangleright\text{lookup}(o.f, m.U(t), m.l) = v$ and

$$\triangleright(\text{flush}(m.l, m.U(t)), m.s, m.U[t \mapsto \varepsilon], \zeta) \in \ulcorner r * o.f \mapsto v \urcorner,$$

which ends the proof. □

Lemma 42 (R-Self-Sound)**.** *For any $o \in \Delta(OId)$, $f \in \Delta(FName)$, $v \in \Delta(Val)$,*

$$(\lambda t.\ \triangleright \ulcorner o.f \mapsto v \textbf{ in } t \urcorner) \vdash^{\text{m}}_{\text{r}} o.f \mapsto v.$$

Proof. Take any $t \in TId$ and $(l, s, U, \zeta) \in p(t)$ such that $t \in \text{dom}(U)$. We have

$$\triangleright(\text{flush}(l, U(t)), s, U[t \mapsto \varepsilon], \zeta) \in \ulcorner o.f \mapsto v \urcorner.$$

By definition, this means we have $\triangleright(l' \leq \text{flush}(l, U(t)))$ such that $\triangleright(l', s, \zeta) \in o.f \mapsto v$, and so $\triangleright o.f \in \text{dom}(l')$. From this, using properties of flush and lookup, we can conclude that $\triangleright\text{lookup}(o.f, U(t), l) = v$, which ends the proof. □

Lemma 43 (R-$\mathcal{U}$-M-Sound)**.** *For any* $p, q \in \Delta(\mathit{TId}) \to \mathit{Prop}_{\mathit{TSO}}$*,* $o \in \Delta(\mathit{OId})$*,* $f \in \Delta(\mathit{FName})$*,* $v \in \Delta(\mathit{Val})$*,*

$$q \vdash^{\mathrm{m}}_{\mathrm{r}} o.f \mapsto v \Rightarrow (\lambda t \in \mathit{TId}.\ p(t)\ \mathcal{U}_t\ q(t)) \vdash^{\mathrm{m}}_{\mathrm{r}} o.f \mapsto v.$$

Proof. Take any $t \in \Delta(\mathit{TId})$ and $(l, s, U, \zeta) \in p(t)\ \mathcal{U}_t\ q(t)$ such that $t \in \mathrm{dom}(U)$. By definition of $\mathcal{U}_t$ we have α and β such that $U(t) = \alpha \cdot \beta$ and

$$(\mathrm{flush}(l, \alpha), s, U[t \mapsto \beta], \zeta) \in q(t).$$

This allows us to use the assumption, from which we can conclude that

$$\triangleright\mathrm{lookup}(o.f, \beta, \mathrm{flush}(l, \alpha)) = v,$$

and so, using the properties of lookup and flush, $\triangleright\mathrm{lookup}(o.f, \alpha \cdot \beta, l) = v$, which ends the proof. □

Lemma 44 (R-$\overline{\mathcal{U}}$-Sound)**.** *For any* $p, q \in \Delta(\mathit{TId}) \to \mathit{Prop}_{\mathit{TSO}}$*,* $o \in \Delta(\mathit{OId})$*,* $f \in \Delta(\mathit{FName})$*,* $v \in \Delta(\mathit{Val})$*,*

$$p \vdash^{\mathrm{o}}_{\mathrm{r}} o.f \mapsto v \Rightarrow (\lambda t \in \mathit{TId}.\ \exists t' \in \Delta(\mathit{TId}).\ t \neq t' * p(t')\ \mathcal{U}_{t'}\ q(t')) \vdash^{\mathrm{m}}_{\mathrm{r}} o.f \mapsto v.$$

Proof. Take any $t \in \Delta(\mathit{TId})$ and $m \in \exists t' \in \Delta(\mathit{TId}).\ t \neq t' * p(t')\ \mathcal{U}_{t'}\ q(t')$ such that $t \in \mathrm{dom}(m.U)$. By definition, this gives us $t' \in \Delta(\mathit{TId})$ and states $(l_1, s, U, \zeta) \in t \neq t'$, $(l_2, s, U, \zeta) \in p(t')\ \mathcal{U}_{t'}\ q(t')$, such that $m = (l_1 \bullet_{\mathit{LState}} l_2, s, U, \zeta)$. By definition of $\mathcal{U}_t$ we get α and β such that $U(t') = \alpha \cdot \beta$ and $(l_2, s, U[t' \mapsto \alpha] \in p(t')$ (this time we fold the update into β). Now we can use the assumption, obtaining

$$\triangleright\mathrm{lookup}(o.f, U[t' \mapsto \alpha](t), l_2) = v,$$

from which it easily follows, since $t \neq t'$, that $\triangleright\mathrm{lookup}(o.f, U(t), l) = v$, which ends the proof. □

Lemma 45 (R-Cons-M-Sound)**.** *For any* $p, q \in \Delta(\mathit{TId}) \to \mathit{Prop}_{\mathit{TSO}}$*,* $o \in \Delta(\mathit{OId})$*,* $f \in \Delta(\mathit{FName})$*,* $v \in \Delta(\mathit{Val})$*,*

$$q \vdash^{\mathrm{m}}_{\mathrm{r}} o.f \mapsto v \wedge (\forall t.\ p(t) \subseteq q(t)) \Rightarrow p \vdash^{\mathrm{m}}_{\mathrm{r}} o.f \mapsto v.$$

Proof. Take any $t \in \Delta(\mathit{TId})$ and $m \in p(t)$ such that $t \in \mathrm{dom}(m.U)$. By the entailment assumption $m \in q(t)$, so from the other assumption we get $\triangleright\mathrm{lookup}(o.f, m.U(t), m.l) = v$, which ends the proof. □

Lemma 46 (R-Other-Sound)**.** *For any* $o \in \Delta(\mathit{OId})$*,* $f \in \Delta(\mathit{FName})$*,* $v \in \Delta(\mathit{Val})$*,*

$$\triangleright\ulcorner o.f \mapsto v \urcorner \vdash^{\mathrm{o}}_{\mathrm{r}} o.f \mapsto v.$$

Proof. Take any $t, t' \in \Delta(\mathit{TId})$ and $m \in \triangleright\ulcorner o.f \mapsto v\urcorner$ such that $t \neq t'$ and $t \in \mathrm{dom}(m.U)$. By definition of $\ulcorner - \urcorner$, we know that $\triangleright m.l(o,f) = v$ and

$$\triangleright(\forall t \in \mathrm{dom}(m.U).\ \forall v \in \mathit{Val}.\ (o,f,v) \notin m.U(t)).$$

This means that $\triangleright(o,f,v) \notin m.U(t)$ for any v, and so

$$\triangleright(\mathrm{lookup}(o.f, m.U(t), m.l) = m.l(o,f) = v).$$

□

Lemma 47 (R-$\mathcal{U}$-O-Sound)**.** *For any* $p, q \in \Delta(\mathit{TId}) \to \mathit{Prop}_{TSO}$, $o \in \Delta(\mathit{OId})$, $f \in \Delta(\mathit{FName})$, $v \in \Delta(\mathit{Val})$,

$$p \vdash^{\mathrm{o}}_{\mathrm{r}} o.f \mapsto v \Rightarrow (\lambda t \in \mathit{TId}.\ p(t)\ \mathcal{U}_t\ q(t)) \vdash^{\mathrm{o}}_{\mathrm{r}} o.f \mapsto v.$$

Proof. Take any $t, t' \in \Delta(\mathit{TId})$, $(l, s, U, \zeta) \in p(t')\ \mathcal{U}_t\ q(t')$ such that $t \neq t'$ and $t \in \mathrm{dom}(U)$. By definition of $\mathcal{U}_t$, we get α and β such that $U(t') = \alpha \cdot \beta$ and $(l, s, U[t' \mapsto \alpha], \zeta) \in p(t')$. Thus, we can use the assumption to conclude that $\triangleright\mathrm{lookup}(o.f, U[t' \mapsto \alpha](t), l) = v$ and, since $t \neq t'$, $\triangleright\mathrm{lookup}(o.f, U(t), l) = v$. □

Lemma 48 (R-Cons-O-Sound)**.** *For any* $p, q \in \Delta(\mathit{TId}) \to \mathit{Prop}_{TSO}$, $o \in \Delta(\mathit{OId})$, $f \in \Delta(\mathit{FName})$, $v \in \Delta(\mathit{Val})$,

$$q \vdash^{\mathrm{o}}_{\mathrm{r}} o.f \mapsto v \wedge (\forall t.\ p(t) \subseteq q(t)) \Rightarrow p \vdash^{\mathrm{o}}_{\mathrm{r}} o.f \mapsto v.$$

Proof. Take any $t, t' \in \Delta(\mathit{TId})$ and $m \in p(t')$ such that $t \neq t'$ and $t \in \mathrm{dom}(U)$. By the entailment assumption, this means $m \in q(t')$, so we can use the read-judgment assumption to conclude that $\triangleright\mathrm{lookup}(o.f, m.U(t), m.l) = v$. □

Theorem 1 (Soundness)**.** *If* $\Gamma \mid \Phi \vdash \mathsf{S}$ *then* $[\![\Gamma \mid \Phi \vdash \mathsf{S}]\!]$.

6 Verification of a spin-lock in the TSO logic

In this section we verify the spin-lock implementation using the TSO logic. We assume the reader has read the accompanying paper.

6.1 Specification

$$\begin{aligned}
&\exists \mathit{isLock}, \mathit{locked} : \mathsf{Prop_{sc}} \times \mathsf{Val} \to \mathsf{Prop_{sc}}.\\
&\quad \forall \mathsf{R} : \mathsf{Prop_{sc}}.\ \mathsf{stable}(\mathsf{R})\ \Rightarrow\\
&\qquad\quad \{\overline{\mathsf{R}}\}\ \mathsf{Lock}()\ \{\mathsf{r}.\ \mathit{isLock}(\mathsf{R}, \mathsf{r})\}\\
&\qquad \wedge\ \{\mathit{isLock}(\mathsf{R}, \mathbf{this})\}\ \mathsf{Lock.acquire}()\ \{\mathit{locked}(\mathsf{R}, \mathbf{this}) * \ulcorner \mathsf{R} \urcorner\}\\
&\qquad \wedge\ \{\mathit{locked}(\mathsf{R}, \mathbf{this}) * \overline{\mathsf{R}}\}\ \mathsf{Lock.release}()\ \{\top\}\\
&\qquad \wedge\ \ \mathsf{valid}(\forall \mathsf{x} : \mathsf{Val}.\ \mathit{isLock}(\mathsf{R}, \mathsf{x}) \Leftrightarrow \mathit{isLock}(\mathsf{R}, \mathsf{x}) * \mathit{isLock}(\mathsf{R}, \mathsf{x}))\\
&\qquad \wedge\ \forall \mathsf{x} : \mathsf{Val}.\ \mathsf{stable}(\mathit{isLock}(\mathsf{R}, \mathsf{x})) \wedge \mathsf{stable}(\mathit{locked}(\mathsf{R}, \mathsf{x}))
\end{aligned}$$

Formally, the *isLock* and *locked* assertions are also embedded using $\overline{-}$, however as mentioned earlier, region assertions are independent of the local state and we thus elide the embeddings to make the proofs more readable.

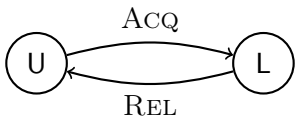
6.2 Predicate Definitions

$$isLock(\mathsf{R},\mathsf{x}) = \exists \mathsf{n} : \mathsf{RId}.\ [\textsc{Acq}]^{\mathsf{n}}_{\_} * \mathsf{region}(\{\mathsf{U},\mathsf{L}\}, \mathsf{T}_{lock}, I(\mathsf{x},\mathsf{R},\mathsf{n}), \mathsf{n})$$
$$locked(\mathsf{R},\mathsf{x}) = \exists \mathsf{n} : \mathsf{RId}.\ [\textsc{Acq}]^{\mathsf{n}}_{\_} * [\textsc{Rel}]^{\mathsf{n}}_{1} * \mathsf{region}(\{\mathsf{L}\}, \mathsf{T}_{lock}, I(\mathsf{x},\mathsf{R},\mathsf{n}), \mathsf{n})$$

where

$$I(\mathsf{x},\mathsf{R},\mathsf{n})(\mathsf{U}) = \exists \mathsf{t} : \mathsf{TId}.\ (\!|\ulcorner \mathsf{x.locked} \mapsto \mathsf{true} \urcorner\ \mathcal{U}_{\mathsf{t}}\ \ulcorner \mathsf{x.locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]^{n}_{1} \urcorner |\!)$$
$$I(\mathsf{x},\mathsf{R},\mathsf{n})(\mathsf{L}) = \ulcorner \mathsf{x.locked} \mapsto \mathsf{true} \urcorner$$

and T_{lock} refers to the following transition system



6.3 Proof outline

In this section we sketch proofs for each of the spin-lock methods in iCAP-TSO. Interestingly, the *structure* of the proof outlines differ in crucial ways from the corresponding proofs in iCAP (which assumes a strong memory model): In the case of acquire, even if the shared lock region is in the unlocked abstract state, it might still appear to be locked from the point of view of a client attempting to acquire the lock (if the lock was last released by a different thread, and release has not made its way to main memory yet), and the client will thus be unable to acquire the lock.

Constructor

```
class Lock {
  bool locked;

  Lock() =
{R̄ * ⌜this.locked ↦ null⌝}
  ⟨R̄ * ⌜this.locked ↦ null⌝⟩
    CAS(this.locked, false, null);
  ⟨(⌜R * this.locked ↦ null⌝)⟩
  ⟨region({U, L}, T_lock, I(this, R, n), n) * [Acq]^n_⟩
  ⟨isLock(R, this)⟩
{isLock(R, this)}
    this
{r. isLock(R, r)}
```

Acquire. The following is a proof sketch of the acquire method. As expected, most of the interesting reasoning concerns the atomic compare-and-swap expression that attempts to acquire the lock.

$\mathsf{acquire}() =$
$\{isLock(\mathsf{R}, \mathbf{this})\}$
$\{\mathsf{region}(\{\mathsf{U}, \mathsf{L}\}, \mathsf{T}_{lock}, I(\mathbf{this}, \mathsf{R}, \mathsf{n}), \mathsf{n}) * [\textsc{Acq}]^{\mathsf{n}}_{\_}\}$
  $\mathbf{let}\ \mathsf{y} = \mathsf{CAS}(\mathbf{this}.\mathsf{locked}, \mathbf{true}, \mathbf{false})\ \mathbf{in}$
$\{(\mathsf{y} = \mathsf{true} * \mathsf{region}(\{\mathsf{L}\}, \mathsf{T}_{lock}, I(\mathbf{this}, \mathsf{R}, \mathsf{n}), \mathsf{n}) * [\textsc{Acq}]^{\mathsf{n}}_{\_} * [\textsc{Rel}]^{\mathsf{n}}_{1} * \ulcorner \mathsf{R} \urcorner) \ \vee$
 $(\mathsf{y} = \mathsf{false} * \mathsf{region}(\{\mathsf{U}, \mathsf{L}\}, \mathsf{T}_{lock}, I(\mathbf{this}, \mathsf{R}, \mathsf{n}), \mathsf{n}) * [\textsc{Acq}]^{\mathsf{n}}_{\_})\}$
    $\mathbf{if}\ \mathsf{y}\ \mathbf{then}$
$\{\mathsf{region}(\{\mathsf{L}\}, \mathsf{T}_{lock}, I(\mathbf{this}, \mathsf{R}, \mathsf{n}), \mathsf{n}) * [\textsc{Acq}]^{n}_{\_} * [\textsc{Rel}]^{\mathsf{n}}_{1} * \ulcorner \mathsf{R} \urcorner\}$
$\{locked(\mathsf{R}, \mathbf{this}) * \ulcorner \mathsf{R} \urcorner\}$
      $()$
    $\mathbf{else}$
$\{\mathsf{region}(\{\mathsf{U}, \mathsf{L}\}, \mathsf{T}_{lock}, I(\mathbf{this}, \mathsf{R}, \mathsf{n}), \mathsf{n}) * [\textsc{Acq}]^{\mathsf{n}}_{\_}\}$
$\{isLock(\mathsf{R}, \mathbf{this})\}$
      $\mathsf{acquire}()$
$\{locked(\mathsf{R}, \mathbf{this}) * \ulcorner \mathsf{R} \urcorner\}$

To verify the atomic compare-and-swap we use the ATOMIC rule to open the shared lock region. Since the pre-condition asserts that the lock is either locked or unlocked ($region(\{\mathsf{U}, \mathsf{L}\}, ...)$), we get two proof obligations, corresponding to each case:

$\langle [\textsc{Acq}]^{\mathsf{n}}_{\_} * \triangleright I(\mathbf{this}, \mathsf{R}, \mathsf{n})(\mathsf{U})\rangle$
  $\mathsf{CAS}(\mathbf{this}.\mathsf{locked}, \mathbf{true}, \mathbf{false})$
$\langle \mathsf{r}.\ \exists \mathsf{y} \in \{\mathsf{U}, \mathsf{L}\}.\ [\textsc{Acq}]^{\mathsf{n}}_{\_} * \triangleright I(\mathbf{this}, \mathsf{R}, \mathsf{n})(\mathsf{y}) * Q(\mathsf{y}, \mathsf{r}, \mathsf{n})\rangle$

where

$$Q(\mathsf{y}, \mathsf{r}, \mathsf{n}) = (\mathsf{y} = \mathsf{L} * [\textsc{Rel}]^{\mathsf{n}}_{1} * \ulcorner \mathsf{R} \urcorner * \mathsf{r} = \mathsf{true}) \vee (\mathsf{y} = \mathsf{U} * \mathsf{r} = \mathsf{false})$$

and

$\langle [\textsc{Acq}]^{\mathsf{n}}_{\_} * \triangleright I(\mathbf{this}, \mathsf{R}, \mathsf{n})(\mathsf{L})\rangle$
  $\mathsf{CAS}(\mathbf{this}.\mathsf{locked}, \mathbf{true}, \mathbf{false})$
$\langle \mathsf{r}.\ [\textsc{Acq}]^{\mathsf{n}}_{\_} * \triangleright I(\mathbf{this}, \mathsf{R}, \mathsf{n})(\mathsf{L}) * \mathsf{r} = \mathsf{false}\rangle$

We start with the (easy) second proof obligation:

$\langle [\textsc{Acq}]^{\mathsf{n}}_{\_} * \triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{true} \urcorner\rangle$
  $\langle \triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{true} \urcorner\rangle$
  $\mathsf{CAS}(\mathbf{this}.\mathsf{locked}, \mathbf{true}, \mathbf{false})$
  $\langle \mathsf{r}.\ \triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{true} \urcorner * \mathsf{r} = \mathsf{false}\rangle$
$\langle \mathsf{r}.\ [\textsc{Acq}]^{\mathsf{n}}_{\_} * \triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{true} \urcorner * \mathsf{r} = \mathsf{false}\rangle$

By rule A-Cas-False it thus suffices to prove that

$$\triangleright\ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{true}\urcorner \vdash^{\mathrm{m}}_{\mathrm{r}} \mathsf{this}.\mathsf{locked} \mapsto \mathsf{true},$$

which follows easily by rules R-Self and R-Cons (to weaken $\ulcorner - \urcorner$ to $\overline{-}$).

To prove the first proof obligation, we first use Stab-$\mathcal{U}$ to do case analysis on whether or not the last release has made its way to main memory yet. Then we commute $\triangleright$ all the way into the pre-condition, before finally doing case analysis on whether (if the last release is still pending) the last release is in our store buffer or not.

$\langle [\textsc{Acq}]^{\mathsf{n}}_{\_} * \triangleright(\exists \mathsf{t} : \mathsf{TId}.\ (\!|\ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{true}\urcorner\, \mathcal{U}_{\mathsf{t}}\, \ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]^{\mathsf{n}}_1\urcorner\ |\!|))\rangle$
$\quad\langle \triangleright(\exists \mathsf{t} : \mathsf{TId}.\ (\!|\ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{true}\urcorner\, \mathcal{U}_{\mathsf{t}}\, \ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]^{\mathsf{n}}_1\urcorner\ |\!|))\rangle$
$\quad\langle \exists \mathsf{t} : \mathsf{TId}.\ \triangleright(\!|\ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{true}\urcorner\, \mathcal{U}_{\mathsf{t}}\, \ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]^{\mathsf{n}}_1\urcorner\ |\!|))\rangle$
$\quad\quad\langle \triangleright(\!|\ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{true}\urcorner\, \mathcal{U}_{\mathsf{t}}\, \ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]^{\mathsf{n}}_1\urcorner\ |\!|))\rangle$
$\quad\quad\langle \triangleright((\ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{true}\urcorner\, \mathcal{U}_{\mathsf{t}}\, (\!|\ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]^{\mathsf{n}}_1\urcorner\ |\!|)) \lor$
$\quad\quad\quad\quad (\!|\ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]^{\mathsf{n}}_1\urcorner\ |\!|))\rangle$
$\quad\quad\langle \triangleright((\ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{true}\urcorner\, \mathcal{U}_{\mathsf{t}}\, \ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]^{\mathsf{n}}_1\urcorner) \lor$
$\quad\quad\quad\quad \ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]^{\mathsf{n}}_1\urcorner)\rangle$
$\quad\quad\langle ((\triangleright\ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{true}\urcorner\, \mathcal{U}_{\mathsf{t}}\, \triangleright\ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{false}\urcorner * \triangleright\ulcorner\mathsf{R}\urcorner * \triangleright\ulcorner[\textsc{Rel}]^{\mathsf{n}}_1\urcorner) \lor$
$\quad\quad\quad\quad \triangleright\ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{false}\urcorner * \triangleright\ulcorner\mathsf{R}\urcorner * \triangleright\ulcorner[\textsc{Rel}]^{\mathsf{n}}_1\urcorner)\rangle$
$\quad\quad\langle ((\triangleright\ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{true}\urcorner\, \mathcal{U}\, \triangleright\ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{false}\urcorner * \triangleright\ulcorner\mathsf{R}\urcorner * \triangleright\ulcorner[\textsc{Rel}]^{\mathsf{n}}_1\urcorner) \lor$
$\quad\quad\quad\quad (\triangleright\ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{true}\urcorner\, \overline{\mathcal{U}}\, \triangleright\ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{false}\urcorner * \triangleright\ulcorner\mathsf{R}\urcorner * \triangleright\ulcorner[\textsc{Rel}]^{\mathsf{n}}_1\urcorner) \lor$
$\quad\quad\quad\quad\quad \triangleright\ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{false}\urcorner * \triangleright\ulcorner\mathsf{R}\urcorner * \triangleright\ulcorner[\textsc{Rel}]^{\mathsf{n}}_1\urcorner)\rangle$
$\quad\mathsf{CAS}(\mathsf{this}.\mathsf{locked}, \mathsf{true}, \mathsf{false})$
$\quad\langle \mathsf{r}.\ \exists \mathsf{y} \in \{\mathsf{U}, \mathsf{L}\}.\ \triangleright I(\mathsf{this}, \mathsf{R}, \mathsf{n})(\mathsf{y}) * Q(\mathsf{y}, \mathsf{r}, \mathsf{n})\rangle$
$\langle \mathsf{r}.\ [\textsc{Acq}]^{\mathsf{n}}_{\_} * \exists \mathsf{y} \in \{\mathsf{U}, \mathsf{L}\}.\ \triangleright I(\mathsf{this}, \mathsf{R}, \mathsf{n})(\mathsf{y}) * Q(\mathsf{y}, \mathsf{r}, \mathsf{n})\rangle$
$\langle \mathsf{r}.\ \exists \mathsf{y} \in \{\mathsf{U}, \mathsf{L}\}.\ [\textsc{Acq}]^{\mathsf{n}}_{\_} * \triangleright I(\mathsf{this}, \mathsf{R}, \mathsf{n})(\mathsf{y}) * Q(\mathsf{y}, \mathsf{r}, \mathsf{n})\rangle$

This leaves us with the following three proof obligations:

$\langle \triangleright\ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{true}\urcorner\, \mathcal{U}\, \triangleright\ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{false}\urcorner * \triangleright\ulcorner\mathsf{R}\urcorner * \triangleright\ulcorner[\textsc{Rel}]^{\mathsf{n}}_1\urcorner\rangle$
$\quad\mathsf{CAS}(\mathsf{this}.\mathsf{locked}, \mathsf{true}, \mathsf{false})$
$\langle \mathsf{r}.\ \exists \mathsf{y} \in \{\mathsf{U}, \mathsf{L}\}.\ \triangleright I(\mathsf{this}, \mathsf{R}, \mathsf{n})(\mathsf{y}) * Q(\mathsf{y}, \mathsf{r}, \mathsf{n})\rangle$

and

$\langle \triangleright\ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{true}\urcorner\, \overline{\mathcal{U}}\, \triangleright\ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{false}\urcorner * \triangleright\ulcorner\mathsf{R}\urcorner * \triangleright\ulcorner[\textsc{Rel}]^{\mathsf{n}}_1\urcorner)\rangle$
$\quad\mathsf{CAS}(\mathsf{this}.\mathsf{locked}, \mathsf{true}, \mathsf{false})$
$\langle \mathsf{r}.\ \exists \mathsf{y} \in \{\mathsf{U}, \mathsf{L}\}.\ \triangleright I(\mathsf{this}, \mathsf{R}, \mathsf{n})(\mathsf{y}) * Q(\mathsf{y}, \mathsf{r}, \mathsf{n})\rangle$

and

$\langle \triangleright\ulcorner\mathsf{this}.\mathsf{locked} \mapsto \mathsf{false}\urcorner * \triangleright\ulcorner\mathsf{R}\urcorner * \triangleright\ulcorner[\textsc{Rel}]^{n}_1\urcorner)\rangle$
$\quad\mathsf{CAS}(\mathsf{this}.\mathsf{locked}, \mathsf{true}, \mathsf{false})$
$\langle \mathsf{r}.\ \exists \mathsf{y} \in \{\mathsf{U}, \mathsf{L}\}.\ \triangleright I(\mathsf{this}, \mathsf{R}, \mathsf{n})(\mathsf{y}) * Q(\mathsf{y}, \mathsf{r}, \mathsf{n})\rangle$

corresponding to the three cases:

- that the last release is pending in our store buffer, or
- the last release is pending in some other thread's store buffer, or
- the last release has already made its way to main memory.

In the first and the last case, the lock is objectively *and* subjectively unlocked, and we can thus acquire the lock and transition to the locked state. However, in the second case, the lock is objectively unlocked, but subjectively locked, and we thus remain in the unlocked state. We thus strengthen the post-conditions of these three proof obligations as follows:

$\langle \triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{true} \urcorner \, \mathcal{U} \, \triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{false} \urcorner * \triangleright \ulcorner \mathsf{R} \urcorner * \triangleright \ulcorner [\textsc{Rel}]_1^{\mathsf{n}} \urcorner \rangle$
$\quad \mathsf{CAS}(\mathbf{this}.\mathsf{locked}, \mathbf{true}, \mathbf{false})$
$\langle \triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{true} \urcorner * [\textsc{Rel}]_1^{\mathsf{n}} * \ulcorner \mathsf{R} \urcorner * \mathsf{r} = \mathsf{true} \rangle$
$\langle \mathsf{r}.\ \triangleright I(\mathbf{this}, \mathsf{R}, \mathsf{n})(\mathsf{L}) * Q(\mathsf{L}, \mathsf{r}, \mathsf{n}) \rangle$

and

$\langle \triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{true} \urcorner \, \overline{\mathcal{U}} \, \triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{false} \urcorner * \triangleright \ulcorner \mathsf{R} \urcorner * \triangleright \ulcorner [\textsc{Rel}]_1^{\mathsf{n}} \urcorner) \rangle$
$\quad \mathsf{CAS}(\mathbf{this}.\mathsf{locked}, \mathbf{true}, \mathbf{false})$
$\langle \mathsf{r}.\ \triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{true} \urcorner \, \overline{\mathcal{U}} \, \triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{false} \urcorner * \triangleright \ulcorner \mathsf{R} \urcorner * \triangleright \ulcorner [\textsc{Rel}]_1^{\mathsf{n}} \urcorner) * \mathsf{r} = \mathsf{false} \rangle$
$\langle \mathsf{r}.\ \triangleright (\exists \mathsf{t} : \mathsf{TId}.\ (\!| \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{true} \urcorner \, \mathcal{U}_{\mathsf{t}} \, \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]_1^{\mathsf{n}} \urcorner |\!)) * \mathsf{r} = \mathsf{false} \rangle$
$\langle \mathsf{r}.\ \triangleright I(\mathbf{this}, \mathsf{R}, \mathsf{n})(\mathsf{U}) * Q(\mathsf{U}, \mathsf{r}, \mathsf{n}) \rangle$

and

$\langle \triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{false} \urcorner * \triangleright \ulcorner \mathsf{R} \urcorner * \triangleright \ulcorner [\textsc{Rel}]_1^{\mathsf{n}} \urcorner) \rangle$
$\quad \mathsf{CAS}(\mathbf{this}.\mathsf{locked}, \mathbf{true}, \mathbf{false})$
$\langle \mathsf{r}.\ \triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{true} \urcorner * [\textsc{Rel}]_1^{\mathsf{n}} * \ulcorner \mathsf{R} \urcorner * \mathsf{r} = \mathsf{true} \rangle$
$\langle \mathsf{r}.\ \triangleright I(\mathbf{this}, \mathsf{R}, \mathsf{n})(\mathsf{L}) * Q(\mathsf{L}, \mathsf{r}, \mathsf{n}) \rangle$

By rules A-CAS-False and A-CAS-True, these three proof obligations reduce to the following three read/write proof obligations:

$$\triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{true} \urcorner \, \mathcal{U} \, \triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{false} \urcorner * \triangleright \ulcorner \mathsf{R} \urcorner * \triangleright \ulcorner [\textsc{Rel}]_1^{\mathsf{n}} \urcorner \vdash_{\mathrm{w}} \mathbf{this}.\mathsf{locked} \mapsto \mathsf{false}, [\textsc{Rel}]_1^{\mathsf{n}} * \mathsf{R}$$

and

$$\triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{true} \urcorner \, \overline{\mathcal{U}} \, \triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{false} \urcorner * \triangleright \ulcorner \mathsf{R} \urcorner * \triangleright \ulcorner [\textsc{Rel}]_1^{\mathsf{n}} \urcorner \vdash_{\mathrm{r}}^{\mathrm{m}} \mathbf{this}.\mathsf{locked} \mapsto \mathsf{true}$$

and

$$\triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{false} \urcorner * \triangleright \ulcorner \mathsf{R} \urcorner * \triangleright \ulcorner [\textsc{Rel}]_1^{\mathsf{n}} \urcorner \vdash_{\mathrm{w}} \mathbf{this}.\mathsf{locked} \mapsto \mathsf{false}, [\textsc{Rel}]_1^{\mathsf{n}} * \mathsf{R}$$

In the first and last case we can use rule W-Cons to weaken $\ulcorner - \urcorner$ to $\overline{-}$ and W-* to frame off R and [Rel] and remove the laters in front of these. It thus suffices to prove:

$$\triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{true} \urcorner \, \mathcal{U} \, \triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{false} \urcorner \vdash_{\mathrm{w}} \mathbf{this}.\mathsf{locked} \mapsto \mathsf{false}, \top$$

$$\triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{true} \urcorner \, \overline{\mathcal{U}} \, \triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{false} \urcorner \vdash_{\mathrm{r}}^{\mathrm{m}} \mathbf{this}.\mathsf{locked} \mapsto \mathsf{true}$$

$$\triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{false} \urcorner \vdash_{\mathrm{w}} \mathbf{this}.\mathsf{locked} \mapsto \mathsf{false}, \top$$

The last proof obligation follows easily from W-Ax and W-Cons. By rules W-$\mathcal{U}$ and R-$\overline{\mathcal{U}}$, the first two obligations reduce to:

$$\triangleright\ulcorner\mathbf{this}.\mathsf{locked} \mapsto \mathsf{false}\urcorner \vdash_{\mathrm{w}} \mathbf{this}.\mathsf{locked} \mapsto \mathsf{false}, \top$$
$$\triangleright\ulcorner\mathbf{this}.\mathsf{locked} \mapsto \mathsf{true}\urcorner \vdash^{\mathrm{o}}_{\mathrm{r}} \mathbf{this}.\mathsf{locked} \mapsto \mathsf{true}$$

These follow easily using rules W-Ax, W-Cons and R-Other, R-Cons, respectively.

Release. Below we sketch the proof outline of release:

$\mathsf{release}() =$
$\{\mathit{locked}(\mathsf{R}, \mathbf{this}) * \overline{\mathsf{R}}\}$
$\{\mathsf{region}(\{\mathsf{L}\}, \mathsf{T}_{\mathit{lock}}, I(\mathbf{this}, \mathsf{R}, \mathsf{n}), \mathsf{n}) * [\textsc{Acq}]^{\mathsf{n}}_{\_} * [\textsc{Rel}]^{\mathsf{n}}_{1} * \overline{\mathsf{R}}\}$
  $\{\triangleright\ulcorner\mathbf{this}.\mathsf{locked} \mapsto \mathsf{true}\urcorner * [\textsc{Acq}]^{\mathsf{n}}_{\_} * [\textsc{Rel}]^{\mathsf{n}}_{1} * \overline{\mathsf{R}}\}$
    $\{\triangleright\ulcorner\mathbf{this}.\mathsf{locked} \mapsto \mathsf{true}\urcorner * [\textsc{Rel}]^{\mathsf{n}}_{1} * \overline{\mathsf{R}}\}$
  $\mathbf{this}.\mathsf{locked} := \mathbf{false}$
    $\{(\triangleright\ulcorner\mathbf{this}.\mathsf{locked} \mapsto \mathsf{true}\urcorner * [\textsc{Rel}]^{\mathsf{n}}_{1} * \overline{\mathsf{R}})\ \mathcal{U}\ ([\textsc{Rel}]^{\mathsf{n}}_{1} * \ulcorner\mathsf{R}\urcorner * \ulcorner\mathbf{this}.\mathsf{locked} \mapsto \mathsf{false}\urcorner)\}$
    $\{\triangleright(\exists \mathsf{t} : \mathsf{TId}.\ (\!|\ulcorner\mathbf{this}.\mathsf{locked} \mapsto \mathsf{true}\urcorner\ \mathcal{U}_{\mathsf{t}}\ \ulcorner\mathbf{this}.\mathsf{locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]^{\mathsf{n}}_{1}\urcorner|\!))\}$
  $\{\triangleright(\exists \mathsf{t} : \mathsf{TId}.\ (\!|\ulcorner\mathbf{this}.\mathsf{locked} \mapsto \mathsf{true}\urcorner\ \mathcal{U}_{\mathsf{t}}\ \ulcorner\mathbf{this}.\mathsf{locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]^{\mathsf{n}}_{1}\urcorner|\!)) * [\textsc{Acq}]^{\mathsf{n}}_{\_}\}$
  $\{\triangleright I(\mathbf{this}, \mathsf{R}, \mathsf{n})(\mathsf{U}) * [\textsc{Acq}]^{\mathsf{n}}_{\_}\}$
$\{\mathsf{region}(\{\mathsf{U}, \mathsf{L}\}, \mathsf{T}_{\mathit{lock}}, I(\mathbf{this}, \mathsf{R}, \mathsf{n}), \mathsf{n}) * [\textsc{Acq}]^{\mathsf{n}}_{\_}\}$
$\{\mathit{isLock}(\mathsf{R}, \mathbf{this})\}$

The innermost proof obligation follows by A-Write.

References

[1] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: Compositional Reasoning for Concurrent Programs. In *Proceedings of POPL*, 2013.

[2] K. Svendsen and L. Birkedal. Impredicative Concurrent Abstract Predicates. Technical report, 2013. Available at `www.kasv.dk/articles/icap-tr.pdf`.