Enabling Concise and Modular Specifications in Separation Logic

Jonas B. Jensen

PhD dissertation
IT University of Copenhagen

March 2014

Abstract

Separation logic is an extension of Hoare logic for reasoning about programs that use pointers or references to potentially-shared data. The problem with such programs, both in formal proofs and in informal understanding of them, is to know not only what data they change but also to know what data they leave unmodified. Separation logic has one simple general answer to this, known as the *frame rule*; it intuitively says that all data that is disjoint from the minimal footprint of the program will be unmodified. This thesis is a collection of papers with the common theme of presenting new separation logics and examples of using these logics to verify challenging programs.

The article **Modular Verification of Linked Lists with Views via Separation Logic** reports on verification of a practical data structure with separation logic. The challenges identified in this work has served as motivation for later articles on verifying object-oriented programs and on giving specifications where the meaning of disjointness is different from physical heap disjointness in the implementation.

In **Verifying Object-Oriented Programs with Higher-Order Separation Logic in Coq**, we are concerned with giving modular specifications to programs that use object-oriented inheritance and dynamic dispatch. Dynamic dispatch is as powerful as general functional programming, but the typical usage patterns do not exploit that full generality. We designed a logic to handle the fully general case and presented design patterns that allow simple verification of simple programs.

Fictional Separation Logic allows multiple notions of disjointness to coexist in a verification framework, thereby extending the utility of the frame rule. Using techniques developed in the previous article, the exact meaning of disjointness can be made abstract, hiding the implementation from clients.

The article **High-Level Separation Logic for Low-Level Code** continues the theme of defining a powerful separation logic to give concise and intuitive specifications to challenging programs. In this case, the logic targets x86 machine code. Challenges here include unstructured control flow and the lack of basic facilities in the language such as memory allocation and procedure calls.

Finally, the chapter **Techniques for Model Construction in Separation Logic** surveys the mathematical techniques used to develop the previous separation logics and many other logics in the literature, concluding that most separation logic models fit into a common mathematical framework, and that building new separation logics within this framework provides practical benefits.

Acknowledgements

I am first of all grateful to my supervisors, Lars Birkedal and Peter Sestoft, for encouraging me to apply for a PhD and for guiding me through three exciting years of studies. Despite their busy schedules, they could always find time to sit down and discuss research and other matters with great focus and patience.

Nick Benton and Andrew Kennedy, my hosts during my internship at Microsoft Research Cambridge, also deserve much thanks. They let me work with a great degree of freedom and influence, and their hospitality made me feel equal parts friend and co-worker during my stay.

I worked in the PLS group at the IT University of Copenhagen, and I would like to thank all the faculty and students there for making it an excellent work environment. In particular, my colleagues on the ToMeSo project, Jesper Bengtson, Filip Sieczkowski, Hannes Mehnert, Jacob Thamsborg and Kasper Svendsen, have been great company both in an out of work, and we have had hour-long intense debates over big and small issues of any kind, be it research, politics or the right choice of text editor.

Finally, I am ever grateful to my wife Katja for all the love and support she has given me and for putting up with me over these three years of studies, especially the last hectic weeks before the deadline.

Contents

Chapter 1

Introduction

This introduction aims to describe the contents of this thesis, starting from a programmer's informal understanding of program correctness and developing the terminology needed to appreciate the motivation for and contribution of each of the articles and manuscripts that make up this thesis.

References and discussions of related work are left out here, but they can be found in each of the subsequent chapters.

1.1 The problem: pointer aliasing

In virtually all modern programming languages, both informal understanding and formal verification of programs are hampered by the complexities of *pointer aliasing*. The following two-line Java program demonstrates the problem.

$$\mathsf{x.f} = 1;$$
$$\mathsf{y.f} = 2;$$

After the second assignment, the field x.f might have either the value 1 or 2, depending on whether $\mathsf{x} \neq \mathsf{y}$ or $\mathsf{x} = \mathsf{y}$ respectively. We say that x and y *alias* when $\mathsf{x} = \mathsf{y}$; i.e., the two variables point to the same memory location.

This thesis is concerned with program verification, where the goal would typically be to determine the values of x.f and y.f after the program has executed. The aliasing problem affects other types of program analysis as well; for instance, an optimising compiler might want to know if the first assignment can be discarded, because $\mathsf{x} = \mathsf{y}$, or if the assignments can be reordered or run in parallel, because $\mathsf{x} \neq \mathsf{y}$.

To determine whether x and y alias, we could try to look higher up in the program. Perhaps the last assignment to either x or y was

$$\mathsf{y} = \mathsf{new\ MyClass}();$$

In this case, we would know for sure that they do not alias. But if x and y were both method arguments, we would need to examine all potential callers

of the current method, then perhaps all potential callers of *those* methods, and so on. The problem is undecidable, and the optimising compiler would be right to give up at this point and assume that both $\mathsf{x} = \mathsf{y}$ and $\mathsf{x} \neq \mathsf{y}$ are possible; but in a program verification task, the intended specification may only be provable if we can prove, e.g., $\mathsf{x} \neq \mathsf{y}$.

Examining all potential callers of a method introduces another problem central to this thesis: lack of *modularity*. Modularity, in this context, means the ability to specify and verify each part of a whole program in isolation such that the specifications can be composed to imply the specification of the whole program. For a public method in a shared library, modularity is essential because all the potential callers have not even been written yet, so examining them all is impossible rather than simply infeasible.

If correctness of the library method relies on the parameters not aliasing, then we have to pass on this proof obligation to callers and require that they do not call the method with arguments that might alias. This can be as simple as a code comment:

$$/** \text{ Beware: behaviour is undefined if } \mathsf{x} = \mathsf{y} \ */ \tag{1.1}$$

This is called a *precondition* of the method. More formally, a precondition is a logical predicate on the machine state that must be satisfied before the method can be called. Analogously, a *postcondition* is a logical predicate on the machine state that is guaranteed to be satisfied after the method returns.

For more sophisticated data structures, code comments like the above will not suffice. Even the simplest of data structures, such as linked lists, will have internal pointers that cannot easily be named in a precondition – their names are meaningless to callers because they will be the names of private fields. The best we can do is perhaps to write

```
/**
 * Beware: behaviour is undefined if any pointer
 * reachable from arg1 equals any pointer reachable
 * from arg2, except for the shared data of type C
 * they both refer to.
 */
```
(1.2)

At this point, it should be clear that we need a better formal language for these assertions. Separation logic offers exactly this.

1.2 A solution: separation logic

Separation logic is a family of program logics that make it *easy* to specify when pointers do not alias and *possible* to specify when they do. Separation logic was invented around 2001 by Reynolds, O'Hearn and others.

Predicates on machine state in separation logic are called *assertions*, and the most basic of them is the *points-to* assertion, written $l \mapsto v$ for a location l and value v. It asserts that the memory cell at location l contains value v. Separation logic also features the *separating conjunction* operator on assertions: $P * Q$ holds of a state that can be split into *disjoint* substates such that P holds in one and Q in the other.

The requirement for disjointness in the definition of $*$ is what makes it different from standard conjunction, $\wedge$. Coming back to the above example where an update to $\mathsf{y.f}$ might overwrite $\mathsf{x.f}$, a separation-logic version of our informal precondition (1.1) could require the existence of values v_1 and v_2 such that

$$\mathsf{x.f} \mapsto v_1 * \mathsf{y.f} \mapsto v_2.$$

This would establish two things. First, both locations are accessible in memory and can be accessed without faulting – in Java, this means $\mathsf{x} \neq \mathsf{null}$ and $\mathsf{y} \neq \mathsf{null}$. Second, the locations cannot alias, since if $\mathsf{x.f}$ and $\mathsf{y.f}$ denoted the same location l, then no matter how the state is split into disjoint substates, only one of them can contain l.

Generalising the above, a separating conjunction of n points-to assertions, $l_1 \mapsto v_1 * \cdots * l_n \mapsto v_n$, encodes $\frac{n(n-1)}{2}$ inequality constraints, so it quickly becomes more compact than writing out all those constraints. But more importantly, we can now encode the more challenging method specification, (1.2), as a separation-logic assertion:

$$\exists c.\ \mathsf{MyStruct}(\mathsf{arg1}, c) * \mathsf{MyStruct}(\mathsf{arg2}, c)$$

We see a few new elements here. First, a value c is existentially quantified in the assertion. The language of assertions is a full-featured *logic*, so all the standard logical operators ($\exists, \forall, \vee, \wedge, \mathsf{True}, \mathsf{False}, \Rightarrow, \neg$) are included and behave as usual. Second, the predicate $\mathsf{MyStruct}$ denotes a parametrised assertion that describes the shape of the data structure in question. A particular style of using such predicates has been popularised by Parkinson and Bierman under the name *abstract predicates*. In this approach, only the implementation of the $\mathsf{MyStruct}$ class can see the definition of the $\mathsf{MyStruct}$ predicate. Public callers see it as an opaque token provided by the constructor and required by methods on the class, and this allows it to mention private fields without exposing this information to clients.

Separation logic is a Hoare logic, where both whole methods and individual commands are specified in terms of *Hoare triples*. A Hoare triple $\{P\}\ c\ \{Q\}$ asserts that command c has precondition P with postcondition Q. Here P and Q are assertions, and we can say that (P, Q) is a *specification* of c; a command may have multiple specifications.

The concise description of aliasing provided by separating conjunction also leads to a solution for the *frame problem* in programming languages with pointers. This is the problem of describing what part of the state has *not*

changed after executing a command. Separation logic has a simple, universal answer to this: all state that is disjoint from some valid precondition of c will remain unmodified and disjoint after executing c. Formally, this is captured by the frame rule:

$$\frac{\{P\}\ c\ \{Q\}}{\{P * R\}\ c\ \{Q * R\}} \tag{1.3}$$

There is typically a syntactic side condition on this rule, requiring that no named variables that may be modified in c are mentioned in R.

Reading from the top down, the frame rule intuitively says that if a command c has been shown to have specification (P, Q), then it also has specification $(P * R,\ Q * R)$. Reading instead from the bottom up, it says that if we want to verify that c has a specification with a repeated disjoint component R, then we may disregard R for the duration of that verification.

1.3 Overview of this thesis

For all the benefits of separation logic, it still only solves a restricted set of problems within a restricted set of languages. The articles in this thesis all aim to extend the scope where separation logic applies.

A common theme to the articles is to define separation logics that allow giving good specifications to programs that would otherwise be hard to specify. In my opinion, a "good" specification should be concise and clear so it is easy to see whether it expresses what is intended. It should also be *modular*, which in this setting means that it is robust against changes to both library and caller: changing the internals of the library must not require reverification of callers, and new callers must be able to use the library in its full generality without reverifying it to a new specification.

Chapters 2 and 4–5 investigate cases where disjointness of memory locations as discussed above is not the only useful notion of disjointness. The application domain of the program may have its own natural notion of disjointness, as in Chapter 2. If this notion satisfies certain properties, *fictional separation logic* as introduced in Chapter 4 can make $*$ coincide with the disjointness of the application domain. This leads to specifications that can be highly robust against implementation changes. Multiple notions of disjointness can coexist in a single program, and they can be made abstract to callers.

Finding the best design pattern for specifying certain classes of programs, as well as the right features in the logic to support these, is an open problem that will always remain open. In Chapter 3, we investigate this problem for the case of object-oriented programming. The dynamic dispatch facility of object-oriented languages is as powerful as general functional programming, but it is rarely used in its full generality. This observation led Parkinson and Bierman to propose a separation logic featuring *abstract predicate fam-*

ilies, which is a specification facility that mirrors how object-orientation is most often used. Chapter 3 attempts to improve on this by decoupling the separation logic from the design pattern. This enables a more general but simpler separation logic, and it makes it easy to investigate variations on the design pattern without re-doing the metatheory of the separation logic for each variation.

The theme of extending the scope of separation logic and the frame rule continues in Chapter 6, where we define a separation logic for x86 machine code. The typical formulation of the frame rule we saw in Equation (1.3) is in terms of Hoare triples, but Hoare triples specify programs that have exactly one entry point (where the precondition holds) and either a single exit point or at least a common postcondition for all the exit points. Machine code has none of that structure, and previous approaches have been to either have no frame rule or define highly complex triples with multiple pre- and postconditions, each associated with a location in the code. In Chapter 6, we do just the opposite: break down the triple into simpler building blocks that can be assembled to yield both the standard triple for structured code fragments, the highly complex triples proposed in other work, and also completely new triple-like formulas. The frame rule holds by construction for any triple-like formula. Like in Chapter 3, the aim is to provide a general and powerful logic that allows encoding of multiple design patterns.

Working to solve all the problems mentioned above produces another problem in itself: there is a tendency for separation logic metatheory to be recreated from scratch with slight modifications in every article rather than being reused in a formal sense. Chapter 7 attempts to survey and point out the techniques that are common to almost every separation-logic article. Hopefully, this will help future research to focus on the novel ingredients in the proposed logic rather than the standard ones. There is no indication at this point that separation-logic research is converging on a single logic that is both general and practical, but I hope that we can get closer to that goal by reusing techniques rather than reinventing them.

Notation and terminology is unfortunately not consistent across the chapters of this thesis since they are independent papers. In particular, the term *separation algebra* refers to gradually more general structures in Chapters 3, 4 and 7. Similarly, the treatment of program variables and *open terms* is essentially the same throughout the thesis, but details and notation gradually improve.

1.4 Details of publications

Here follows a list of publications I contributed to during my PhD studies and an account of what role I had in the research and writing phases. The

publications where I only played a small role are not included in this thesis.

1. **Modular Verification of Linked Lists with Views via Separation Logic.** Jonas Braband Jensen, Lars Birkedal and Peter Sestoft. In *Proceedings of FTfJP*, 2010. Extended and revised version in *Journal of Object Technology*, 2011. Included as Chapter 2 of this thesis.

   This article is a summary of my MSc thesis but rewritten from scratch during my PhD. I wrote Sections 3–8 and the appendices.

2. **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. Included as Chapter 3 of this thesis.

   This is the first article about what we later dubbed the *Charge!* platform. Charge! is a development in the *Coq* proof assistant that aims to be a sound platform for the verification of object-oriented programs in a subset of Java. It was developed jointly by Jesper Bengtson, Filip Sieczkowski and myself, and we all contributed to every aspect of it.

   In the writing of this article, my main responsibilities were Sections 2 and 5. Both sections are about specification patterns in higher-order separation logic for object-oriented programs.

   Compared to the published version of the article, this thesis corrects a typo in the definition of the *later* operator: $\{0\}$ is corrected to S, which gives a somewhat non-standard definition of *later*, but it is the one we used, and it is the one that corresponds to the LÖB rule printed underneath it.

3. **Fictional Separation Logic.** Jonas Braband Jensen and Lars Birkedal. In *Proceedings of ESOP*, 2012. Included as Chapters 4 and 5 of this thesis.

   I developed this theory under guidance from Lars, who steered it from being specific to object-orientation and rather overcomplicated to being general and simple.

   I wrote a majority of the article text and the entirety of the appendix.

4. **Charge! – A framework for higher-order separation logic in Coq.** Jesper Bengtson, Jonas Braband Jensen and Lars Birkedal. In *Proceedings of ITP*, 2012. Not included in this thesis.

   This is the second article about the Charge! platform that I contributed to. It focuses on automation using Coq *tactics*, and my contributions were minor. I wrote the Coq code behind the example in Section 3, and I wrote Section 2 about the formal handling of program variables.

5. **High-Level Separation Logic for Low-Level Code.** Jonas B. Jensen, Nick Benton and Andrew Kennedy. In *Proceedings of POPL*, 2013. Included as Chapter 6 of this thesis.

   This article reports on another another development featuring separation logic in Coq. This one belongs to my co-authors at Microsoft Research Cambridge, where I did a 3-month internship in the spring of 2012.

   I did the writing and the research behind the sections focused on separation logic: Sections 3–5.1, 5.3–5.5, 6.2–6.3 and 7–8.

6. **Coq: The world's best macro assembler?** Andrew Kennedy, Nick Benton, Jonas B. Jensen and Pierre-Evariste Dagand. In *Proceedings of PPDP*, 2013. Not included in this thesis.

   While the article above focuses on the program *verification* aspects of the MSR Cambridge Coq development, this follow-up article focuses on the program *generation* aspects. In particular, parts of the development works as a macro assembler with powerful support for embedded domain-specific languages.

   My role in the writing of this article was minor. My main contributions were to the infrastructure in the Coq development that allowed lexically-scoped labels to be verified and assembled.

Published by ETH Zurich, Chair of Software Engineering, © JOT 2010
Online at http://www.jot.fm.

Modular Verification of Linked Lists with Views via Separation Logic

Jonas Braband Jensen[a] Lars Birkedal[a] Peter Sestoft[a]

a. IT University of Copenhagen

Abstract We present a separation logic specification and verification of *linked lists with views*, a data structure from the C5 collection library for .NET. A *view* is a generalization of the well-known concept of an iterator. Linked lists with views form an interesting case study for verification since they allow mutation through multiple, possibly overlapping, views of the same underlying list. For modularity, we build on a fragment of higher-order separation logic and use abstract predicates to give a specification with respect to which clients can be proved correct. We introduce a novel mathematical model of lists with views, and formulate succinct modular abstract specifications of the operations on the data structure. To show that the concrete implementation realizes the specification, we use fractional permissions in a novel way to capture the sharing of data between views and their underlying list.

We conclude by suggesting directions for future research that arose from conducting this case study.

Keywords Separation logic, formal verification, modularity

1 Introduction

Separation logic [Rey02] is a generalization of Hoare logic better suited for reasoning about heap data in imperative programming. In particular, the logic's separating conjunction connective directly supports reasoning about situations where heap-allocated data can be separated into non-overlapping regions. The challenging applications of separation logic are therefore those involving partially overlapping data structures. List iterators provide one example of such structures, and they have been studied extensively in connection with separation logic [KAB$^+$09, BRZ07, HH08].

Here we investigate the *linked list with views* (LLWV) data structure from the C5 library [KS06] of collections for the .NET framework. Where an iterator can be thought of as marking a current *position* in a list, a view more generally marks a list *segment*, so an iterator is just a special case of a view (of length zero). A list may have multiple views, the views may overlap, and modifications to the underlying list show through the views and vice versa; for more details, see Section 2.

Hence views provide a much more powerful mechanism than iterators but also pose new challenges for verification. In particular, the co-dependencies between a list and its views are typically implemented by cyclic pointer structures, and there is no "obvious" mathematical model of a linked list with views. We find that this makes the data structure a challenging and compelling case study for specification and verification with separation logic and related approaches.

1.1 Related Work

Hoare pioneered the proof method of relating a concrete (object-oriented) implementation to an abstract (functional, mathematical) implementation [Hoa72]; we use the same technique here. We also use the concepts of precondition and postcondition, which are due to Dijkstra [Dij76] and which form the basis for Meyer's design-by-contract methodology [Mey92]. However, we do not use class invariants, because they appear to fall short when, as in the case of linked lists with views, there is no hierarchical "ownership" relation among the objects making up a data structure [Par07]. Hence our formalization is not immediately expressible in contemporary frameworks such as the Java Modeling Language [CKLP06] or .NET Code Contracts [FBL10].

The formalization and proof of lists with views, presented in this paper, uses separation logic and has many similarities with separation logic formalizations of iterators. The iterators from the Java standard library seem to be the most popular objects of study [KAB+09, Par05, HH08, BRZ07]. In contrast to the list views discussed here, such iterators become invalid after structural modification to the underlying list, and so it becomes an important part of the specification to capture the protocol that constrains the permitted order of method calls.

Krishnaswami et al. [KAB+09] use higher-order separation logic to give an elegant specification of iterators. It allows multiple iterators at the same time, but iterators are read-only.

Parkinson [Par05] specifies iterators in first-order separation logic, instead using *counting permissions* to share the list between multiple iterators. Again, modification of the list through iterators is not considered.

Haack & Hurlin [HH08] use *fractional permissions* to give a specification that allows both multiple iterators and (limited) modification of the list through iterators. The techniques used to achieve this have similarities to what we present in Section 4.3.

In contrast to iterators, it is always well-defined how views behave after the underlying list is modified.

1.2 Significance for Object-Oriented Languages

The present work focuses on a particular aspect of object-oriented languages: *local update* (by assignment `x.f=e` to object fields) combined with *sharing* (by having multiple references `x` and `y` denoting the same object). This combination means that multiple surface "names" `x.f` and `y.f` denote the same updatable data structure, which makes object-oriented programs hard to reason about using the basically substitution-based approach of Hoare logic. This is not just a formal problem, but also a challenge to informal program understanding, as evidenced by the recent emphasis on the virtues of immutable data; see for instance Josh Bloch's admonishment "Minimize Mutability" [Blo08, Item 15].

In this paper we use separation logic to handle the combination of field update and sharing. We also assume the object-oriented virtue of *encapsulation*: a client

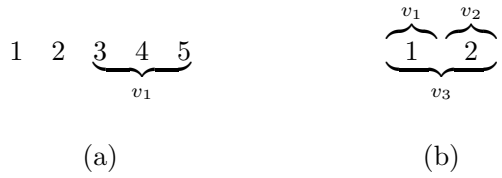

Figure 1 – (a) Linked list with one view. (b) Linked list with two 1-item views and one 2-item view.

cannot arbitrarily access the internals of objects on which it operates. This is of course essential for preservation of invariants and hence for correctness.

On the other hand, we do not address inheritance and virtual methods. Although these are important and challenging features, we believe they are rather orthogonal to the formalization here, and related work has devised one way in which to handle them formally in the context of separation logic [PB08].

1.3 Outline

Section 2 introduces linked lists with views as they are seen from the perspective of a client, and Section 3 describes how they were implemented in this case study.

We give our specification in Section 4 in a fragment of intuitionistic higher-order separation logic [BBTS05], using abstract predicates [PB05], such that clients can be verified without revealing information about the concrete implementation of the data structure. The overall idea in this approach is to use a predicate $\mathsf{L}(x, \alpha)$ that relates a data structure pointer x in the implementation to a mathematical object α that models the data structure abstractly. Partial-correctness specifications for each method f on x are then expressed in terms of this predicate; they are typically of the form $\{\mathsf{L}(x, \alpha)\}\, x.f(\ldots)\, \{\mathsf{L}(x, \alpha')\}$.

We present the concrete realization of the abstract predicates in Section 5, using fractional permissions.

Section 6 presents and discusses the alternative models we have considered.

An earlier version of the results in this paper was published at the FTfJP'10 workshop.

2 Linked Lists With Views

The linked list data structure is well known and is a standard example of separation logic specification and proof. Here we consider *linked lists with views*, a data structure designed as part of the C5 collection library [KS06] that provides several new verification challenges. A *view* is a window on a contiguous segment of a list; a list can have multiple, possibly overlapping, views; see Figure 1 for two examples. An update to a view affects the underlying list as well as overlapping views; and an update to the underlying list may affect multiple views. Finally, a view can be slid left and right along a list, and can be grown and shrunk. A list and its views are closely intertwined, and the update semantics means that there is no "obviously right" model in terms of standard mathematical structures such as sequences, trees and sets.

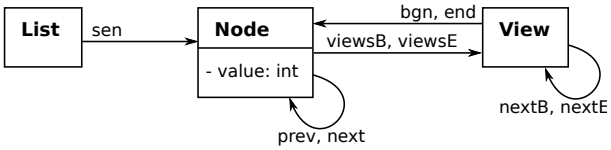

Figure 2 – Class diagram of the implementation.

But why consider the intricacies of these list views at all? Because views have several interesting applications. With views, one can give linked lists and array lists a single common interface, while avoiding the explicit manipulation of internal linked list nodes and hence raising questions of the list's structural integrity, yet provide efficient item access in linked lists, via views instead of item indices.

In fact, a zero-item view is a cursor that points *between* (or before or after) list items, and there are $n + 1$ distinct zero-item views on an n-item list; whereas a one-item view is a cursor that points *at* a list item, and there are n distinct one-item views on an n-item list. Just for this reason, views may be beneficial even for array list algorithms where it is often unclear whether an index i is meant to point *before*, *at*, or *after* the i'th item.

Moreover, a view implements the same interface and supports the same operations as linked lists and array lists, so "sort this particular list segment" can be decomposed into "create a view comprising this particular list segment" and "sort the view". This orthogonality considerably reduces the number of operations that the list interface must exhibit: a single "search view" operation replaces "search entire list", "search list starting at index i", "search list starting at index i and ending at item $i + n$". Furthermore, views (and lists) can be looked at "backwards" so "search view" actually represents $3 \cdot 2 = 6$ different search functions. The same holds for other kinds of list traversal, clearing, shuffling, and so on. Thus views lead to a considerably leaner and more regular list library design.

Apart from the use as between-item and at-item cursors, and to achieve orthogonality of list operations, our updatable slidable views enable elegant implementation of some algorithms such as Graham's point elimination scan when computing a 2D convex hull [KS06, section 11.3]; here three-item views are called for.

The actual C5 data structures are generic, or parametrically polymorphic, in the item type. In this paper we assume for simplicity that list items are just integers, but our proofs do not rely on this fact, and the specification and verification can be extended using higher-order verification techniques for generics [SBP10].

3 Implementation

A class diagram of the implementation data structures is shown in Figure 2. An LLWV has class List; it uses a circular doubly-linked list of Node objects internally to hold the list items. Each node n has a field value that holds the item value; fields prev and next for the doubly-linked list representation; and fields viewsB and viewsE that hold references to two singly-linked lists of View objects: a list of those views that begin just after n and a list of those views that end just before n.

It is an invariant of the data structure that if v.bgn points to n, then n.viewsB points to a list with next-pointers nextB in which v occurs exactly once; and similarly for end/viewsE/nextE.

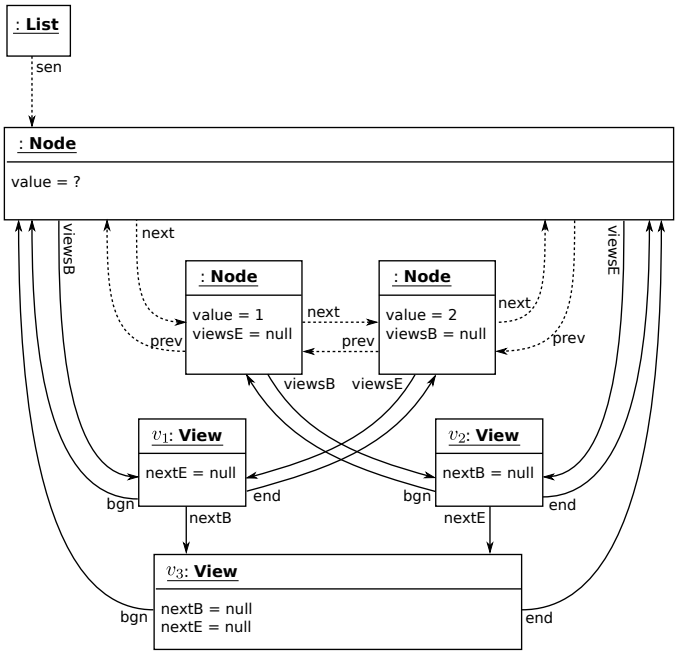

Figure 3 – Object diagram showing the heap layout of the example LLWV from Figure 1(b), with two items and three partially-overlapping views. Dashed arrows make up the list of items, while solid arrows are pointers maintained to support views.

There is a *sentinel* node, whose value we ignore, at the beginning and end of the list. In fact, a single Node object can be used as both start sentinel and end sentinel since the sets of fields used in these two roles are disjoint.

As an example of such a data structure, Figure 3 depicts a possible heap representation of the LLWV from Figure 1(b).

The actual code we have verified is not the original C# code from the C5 library but a Java implementation that has been written from scratch for verification purposes. It captures the essence of what makes LLWV interesting to verify without containing all the bells and whistles that would make it pleasant to use in an engineering context. The most important differences are discussed in Section 5.1.

4 Abstract Specification

This section presents a mathematical model of LLWV and specifications of its methods using abstract predicates. Verification of clients will rely only on these, not on the actual implementation of LLWV.

4.1 Model

Our specifications will revolve around a predicate $\mathsf{L}(l, \alpha)$ that relates a LLWV l (a pointer in the implementation) to a model "bex-list" α. A *bex-list* describes the list items along with all views defined on the list. It seems necessary to join these objects together in one monolithic model because the behaviour of views is defined such that a list and its views can affect each other to a great extent.

The actual definition of the predicate $\mathsf{L}(l, \alpha)$ is shown in Section 5 and is used only when proving an implementation correct. The definition is hidden from clients to prevent them from depending on implementation details [PB05].

Let View be the class of views. Then bex-lists α, β, γ are defined as follows, where "::" denotes list construction:

$$\begin{aligned} \alpha, \beta, \gamma ::= \epsilon \mid \mathsf{B}\, b :: \alpha \mid \mathsf{E}\, e :: \alpha \mid \mathsf{X}\, x :: \alpha \\ b, e \underset{\mathsf{fin}}{\subset} \text{pointers to View} \\ x \in \mathbb{Z} \end{aligned}$$

Intuitively, $\mathsf{B}\, b$ means that the views in set b begin at this position in the list. Similarly, $\mathsf{E}\, e$ means that the views in e end at this position in the list, and $\mathsf{X}\, x$ means that the item x is stored at this position in the list. Such a list element $\mathsf{B}\, b$, $\mathsf{E}\, e$ or $\mathsf{X}\, x$ is called a *bex*.

We will always want the bexes to appear in the order $\mathsf{B}, \mathsf{E}, \mathsf{X}, \mathsf{B}, \mathsf{E}, \mathsf{X}, \ldots$. To enforce this, we define a predicate $\mathsf{ord}(\alpha, t, t')$, where $t, t' \in \{\mathsf{B}, \mathsf{E}, \mathsf{X}\}$, expressing that α is an ordered bex-list starting with a bex of constructor t and ending just before a bex of constructor t'. Formally, let ord be the least predicate satisfying

$$\begin{aligned} &\mathsf{ord}(\epsilon, t, t) \\ &\mathsf{ord}(\mathsf{B}\, b :: \alpha, \mathsf{B}, t) \Longleftarrow \mathsf{ord}(\alpha, \mathsf{E}, t) \\ &\mathsf{ord}(\mathsf{E}\, e :: \alpha, \mathsf{E}, t) \Longleftarrow \mathsf{ord}(\alpha, \mathsf{X}, t) \\ &\mathsf{ord}(\mathsf{X}\, x :: \alpha, \mathsf{X}, t) \Longleftarrow \mathsf{ord}(\alpha, \mathsf{B}, t) \end{aligned}$$

Note that we here used the symbols $\mathsf{B}, \mathsf{E}, \mathsf{X}$ both as (unary) constructors and as (nullary) tags.

Concatenation is defined as usual for cons-based lists and is written $\alpha\beta$. It can be shown by induction on α that

$$\exists t'.\ \mathsf{ord}(\alpha, t, t') \wedge \mathsf{ord}(\beta, t', t'') \iff \mathsf{ord}(\alpha\beta, t, t'')$$

An empty LLWV is modelled by a bex-list $\mathsf{B}\, b :: \mathsf{E}\, e :: \epsilon$, abbreviated *be*. A singleton LLWV is modelled by a bex-list $b_1 e_1 x_1 b_2 e_2$. In general, a LLWV is modelled by an ordered bex-list that begins with a B and ends with an E (i.e. just before an X). We call such lists *well-formed*. We will also need the notion of the length of a bex-list α, written $|\alpha|$. In summary,

$$\begin{aligned} \mathsf{wf}(\alpha) &\triangleq \mathsf{ord}(\alpha, \mathsf{B}, \mathsf{X}) \\ |\alpha| &\triangleq \text{number of X's in } \alpha \end{aligned}$$

The bex-list may not seem like the most intuitive or obvious construction, but it will turn out that specification of the public methods on linked lists with views becomes very simple when using it. Some other models we tried before choosing the bex-list model are discussed in Section 6.

4.2 Operations on Lists

The List class has methods for the list operations one would expect: insertion, removal, subscripting, size query, etc. We now discuss the specifications of the most important operations; Figure 4 gives a summary of all the specifications.

A simple and typical specification is that of $\mathsf{setValue}(i, x')$, which replaces the item at index $i \geq 0$ in list l by x':

$$\{\, \mathsf{L}(l, \alpha x \gamma) \,\}\ l.\mathsf{setValue}(|\alpha|, x')\ \{\, \mathsf{L}(l, \alpha x' \gamma) \,\}$$

This specification says that, provided list l is described by the bex-list $\alpha x \gamma$ before the call, then after the call $l.\mathsf{setValue}(|\alpha|, x')$, list l is described by the bex-list $\alpha x' \gamma$. That is, all list items and views remain the same, except that the item at index $i = |\alpha|$ has been replaced by x'.

Note how part of the precondition is made implicit by restricting the first method argument to have the form $|\alpha|$, instead of an arbitary integer i. Together with the assertion $\mathsf{L}(l, \alpha x \gamma)$ about the shape of the list, this restriction ensures that the item index $|\alpha|$ is legal for the list.

The client should not know the exact definition of L, but it is part of the specification that $\mathsf{L}(l, \alpha)$ implies $\mathsf{wf}(\alpha)$.

It is an important detail that pre- and postconditions are both expressed in terms of equations in bex-lists and their lengths. This makes it easy for sequential client code to establish that the postcondition of one call implies the precondition of the next.

Removal and insertion can also be defined just in terms of bex-list equalities and operations on finite sets. Here, the $\uplus$ operator is a partial version of set union $\cup$ that is defined only for disjoint sets.

$$\{\, \mathsf{L}(l, \alpha b' \, e x b \, e' \gamma) \,\}\ l.\mathsf{remove}(|\alpha|)\ \{\, \mathsf{L}(l, \alpha (b' \uplus b)(e' \uplus e) \gamma) \,\}$$

$$\{\, \mathsf{L}(l, \alpha b e \gamma) \,\}\ l.\mathsf{insert}(|\alpha|, x)\ \{\, \mathsf{L}(l, \alpha (\mathsf{B}(b \cap e)) e x (\mathsf{B}(b \setminus e)) (\mathsf{E}\emptyset) \gamma) \,\}$$

Note that they both preserve well-formedness (and therefore ordering) of the bex-list. The complicated-looking postcondition of insert captures exactly the rules of how views that begin or end around the point of insertion are affected [KS06, Jen10].

4.3 Operations on Views

A new view is created on a list l by calling $l.\mathsf{view}$, specified in Figure 4. We use the special variable *ret* for the return value and use the notation b^v to mean the partial operation $b \cup \{v\}$ where $v \notin b$.

There is an abstract predicate $\mathsf{V}(v, \alpha)$ that is like L for most purposes; it says that view v is described by the bex-list α. As with L, clients are guaranteed that $\mathsf{V}(v, \alpha)$ implies $\mathsf{wf}(\alpha)$. As stated in Figure 4, the methods that work on both lists and views have identical specifications except that l and L are replaced by v and V in the case of views. For example, the specification of setValue on a view would be

$$\{\, \mathsf{V}(v, \alpha x \gamma) \,\}\ v.\mathsf{setValue}(|\alpha|, x')\ \{\, \mathsf{V}(v, \alpha x' \gamma) \,\}$$

The V predicate is not given directly in any method postcondition. Instead, the client is given a guarantee that the following implication is valid:

$$\mathsf{L}(l, \alpha b^v \beta e^v \gamma) \implies \mathsf{V}(v, b \beta e) \ast \forall b', \beta', e'.\ \Big[\mathsf{V}(v, b' \beta' e') \mathrel{-\!\!*} \mathsf{L}(l, \alpha b'^v \beta' e'^v \gamma)\Big] \quad (1)$$

In words, this expresses that a heap containing a LLWV l with a view v can be separated into two parts, say, h and h'. Heap h satisfies the V predicate and can

$$\{\,\mathsf{true}\,\}\ \mathsf{new\ List}()\ \{\,\mathsf{L}(ret, (\mathsf{B}\,\emptyset)\,(\mathsf{E}\,\emptyset))\,\}$$

The following methods are also available on views, with the same specification except that l and L are replaced by v and V.

$\{\,\mathsf{L}(l,\alpha)\,\}$	$l.\mathsf{count}()$	$\{\,\mathsf{L}(l,\alpha) \wedge ret = \lvert\alpha\rvert\,\}$
$\{\,\mathsf{L}(l,\alpha x\gamma)\,\}$	$l.\mathsf{getValue}(\lvert\alpha\rvert)$	$\{\,\mathsf{L}(l,\alpha x\gamma) \wedge ret = x\,\}$
$\{\,\mathsf{L}(l,\alpha x\gamma)\,\}$	$l.\mathsf{setValue}(\lvert\alpha\rvert, x')$	$\{\,\mathsf{L}(l,\alpha x'\gamma)\,\}$
$\{\,\mathsf{L}(l,\alpha b\beta e\gamma)\,\}$	$l.\mathsf{view}(\lvert\alpha\rvert, \lvert\beta\rvert)$	$\{\,\mathsf{L}(l,\alpha b^{ret}\beta e^{ret}\gamma)\,\}$
$\{\,\mathsf{L}(l,\alpha b'\,ex b\,e'\gamma)\,\}$	$l.\mathsf{remove}(\lvert\alpha\rvert)$	$\{\,\mathsf{L}(l,\alpha(b' \uplus b)(e' \uplus e)\gamma)\,\}$
$\{\,\mathsf{L}(l,\alpha be\gamma)\,\}$	$l.\mathsf{insert}(\lvert\alpha\rvert, x)$	$\{\,\mathsf{L}(l,\alpha(\mathsf{B}\,(b \cap e))ex(\mathsf{B}\,(b \setminus e))(\mathsf{E}\,\emptyset)\gamma)\,\}$

Methods specific to views:

$$\{\,\mathsf{L}(l,\alpha b^v\beta e^v\gamma) \wedge \alpha b\beta e\gamma = \alpha' b'\beta' e'\gamma'\,\}$$
$$v.\mathsf{slide}(\lvert\alpha'\rvert - \lvert\alpha\rvert,\ \lvert\beta'\rvert)$$
$$\{\,\mathsf{L}(l,\alpha' b'^v\beta' e'^v\gamma')\,\}$$

$$\{\,\mathsf{L}(l,\alpha b^v\beta e^v\gamma) \wedge \alpha b\beta e\gamma = \alpha' b'\beta' e'\gamma' \wedge \lvert\beta'\rvert = \lvert\beta\rvert\,\}$$
$$v.\mathsf{slide}(\lvert\alpha'\rvert - \lvert\alpha\rvert)$$
$$\{\,\mathsf{L}(l,\alpha' b'^v\beta' e'^v\gamma')\,\}$$

$\{\,\mathsf{L}(l,\alpha b^v\beta e^v\gamma)\,\}$	$v.\mathsf{dispose}()$	$\{\,\mathsf{L}(l,\alpha b\beta e\gamma)\,\}$
$\{\,\mathsf{V}(u,\alpha b^v\beta e^v\gamma)\,\}$	$v.\mathsf{dispose}()$	$\{\,\mathsf{V}(u,\alpha b\beta e\gamma)\,\}$
$\{\,\mathsf{L}(l,\alpha e^v\gamma)\,\}$	$v.\mathsf{atEnd}(l)$	$\{\,\mathsf{L}(l,\alpha e^v\gamma) \wedge ret = (\gamma = \epsilon)\,\}$
$\{\,\mathsf{L}(l,\alpha b^u\beta e^v\gamma)\,\}$	$u.\mathsf{span}(v)$	$\{\,\mathsf{L}(l,\alpha b^{u,ret}\beta e^{v,ret}\gamma)\,\}$
$\{\,\mathsf{L}(l,b\alpha e^v\gamma)\,\}$	$l.\mathsf{span}(v)$	$\{\,\mathsf{L}(l,b^{ret}\alpha e^{v,ret}\gamma)\,\}$
$\{\,\mathsf{L}(l,\alpha b^v\gamma e)\,\}$	$v.\mathsf{span}(l)$	$\{\,\mathsf{L}(l,\alpha b^{v,ret}\gamma e^{ret}\,\}$

Guarantees about predicates:

$$\mathsf{L}(l,\alpha) \implies \mathsf{wf}(\alpha), \qquad \mathsf{V}(v,\alpha) \implies \mathsf{wf}(\alpha),$$
$$\mathsf{L}(l,\alpha b^v\beta e^v\gamma) \implies \mathsf{V}(v,b\beta e) * \forall b',\beta',e'.\ \left[\mathsf{V}(v,b'\beta' e') \mathrel{-\!\!*} \mathsf{L}(l,\alpha b'^v\beta' e'^v\gamma)\right]$$

Figure 4 – Summary of specifications. The notation b^v means $b \cup \{v\}$ where $v \notin b$.

thus be used for calling the various methods on views, such as setValue in Section 4.2, leading to a V-assertion for the same v but with a different bex-list. Heap h' satisfies that given any such modified bex-list $b'\beta'e'$, if h' is extended with a heap in which view v is described by $b'\beta'e'$, then this extended heap describes the original list l except that the sublist delimited by view v has been modified.

The verifier, i.e. the person or heuristic attempting to verify the program, can use (1) to convert from L to V at any convenient time. To get back to L again, he can use the *separating modus ponens* rule: $P * (P \mathrel{-\!\!*} Q) \implies Q$. This is not too different from how the frame rule is used in separation logic in general; in fact, the following specification-logic rule follows from (1).

$$\frac{\{\,\mathsf{V}(v, b\beta e)\,\}\ c\ \{\,\mathsf{V}(v, b'\beta'e')\,\}}{\{\,\mathsf{L}(l, \alpha b^v \beta e^v \gamma)\,\}\ c\ \{\,\mathsf{L}(l, \alpha b'^v \beta' e'^v \gamma)\,\}} \tag{2}$$

In words, if command c changes a view v from $b\beta e$ to $b'\beta'e'$, then if v is a view on some underlying list l described by $\alpha b^v \beta e^v \gamma$, then c will also change list l to $\alpha b'^v \beta' e'^v \gamma$. In particular, the list "tails" α and γ are unaffected by c.

Hence (2) reads as a kind of frame rule, where α and γ constitute the frame that is disregarded while verifying c. Like the frame rule, its application happens at the discretion of the verifier rather than being driven by the program.

Note that (1) is more general than (2) since the conversions between L and V do not have to follow the nesting discipline of a tree in (1).

5 Verification of Implementation

We saw the implementation of the LLWV data structure in Section 3 and the specifications and guarantees involving the L and V predicates in Section 4. To tie these together, we must give the definitions of L and V. The I predicate (I as in Items) will be a key ingredient in this.

We define L as asserting the existence of a sentinel node n_s, which marks both the beginning and ending of the list:

$$\mathsf{L}(l, \alpha) \triangleq \mathsf{wf}(\alpha) * \exists n_\mathrm{s}.\ l.\mathsf{sen} \mapsto n_\mathrm{s} * \mathsf{I}(n_\mathrm{s}, n_\mathrm{s}, \alpha)$$

For an ordered bex-list α and nodes n and n', $\mathsf{I}(\alpha, n, n')$ asserts what must hold of a heap that spans α between n and n'. It does so by a case analysis on whether α is empty or starts with B, E or X. It is a convenient property of the bex-list model that no indirection is needed here: the bex-list as seen by the client corresponds so closely with the heap layout that I can be syntax-directed on α.

Another convenient property is that I admits an excellent correspondence between separation on the heap and concatenation of bex-lists:

$$\mathsf{I}(n, n'', \alpha\beta) \iff \exists n'.\ \mathsf{I}(n, n', \alpha) * \mathsf{I}(n', n'', \beta) \tag{3}$$

The definition of I and all predicates required by it is shown in Figure 5.

One might easily be tempted to define $\mathsf{V}(v, \alpha)$ as

$$\mathsf{wf}(\alpha) * \exists b, \beta, e.\ \alpha = b\beta e * \exists n_\mathrm{b}, n_\mathrm{e}.\ \mathsf{I}(n_\mathrm{b}, n_\mathrm{e}, b^v \beta e^v) \tag{4}$$

Expanding the I predicate will lead to the assertions $v.\mathsf{bgn} \mapsto n_\mathrm{b} * v.\mathsf{end} \mapsto n_\mathrm{e}$ needed by methods on the view as a starting points for accessing its items.

$$
\begin{aligned}
\mathsf{L}(l,\alpha) &\triangleq \mathsf{wf}(\alpha) * \exists n_\mathrm{s}.\ l.\mathsf{sen} \mapsto n_\mathrm{s} * \mathsf{I}(n_\mathrm{s}, n_\mathrm{s}, \alpha) \\
\mathsf{I}(n,n,\epsilon) &\triangleq \mathsf{true} \\
\mathsf{I}(n,n'',\mathsf{B}\,b :: \alpha) &\triangleq \mathsf{I}_\mathsf{B}(n,b) * \mathsf{I}(n,n'',\alpha) \\
\mathsf{I}(n,n'',\mathsf{E}\,e :: \alpha) &\triangleq \exists n'.\ \mathsf{N}(n,n') * \mathsf{I}_\mathsf{E}(n',e) * \mathsf{I}(n',n'',\alpha) \\
\mathsf{I}(n,n'',\mathsf{X}\,x :: \alpha) &\triangleq \mathsf{I}_\mathsf{X}(n,x) * \mathsf{I}(n,n'',\alpha) \\
\mathsf{N}(n,n') &\triangleq n.\mathsf{next} \mapsto n' * n'.\mathsf{prev} \mapsto n \\
\mathsf{I}_\mathsf{B}(n,b) &\triangleq \mathsf{BList}(n,b) * \circledast_{v \in b}\, v.\mathsf{bgn} \mapsto n \\
\mathsf{I}_\mathsf{E}(n,e) &\triangleq \mathsf{EList}(n,e) * \circledast_{v \in e}\, v.\mathsf{end} \mapsto n \\
\mathsf{I}_\mathsf{X}(n,x) &\triangleq n.\mathsf{value} \mapsto x \\
\mathsf{BList}(n,b) &\triangleq \exists v_\mathrm{h}.\ n.\mathsf{viewsB} \mapsto v_\mathrm{h} * \mathsf{BSeg}(v_\mathrm{h}, \mathsf{null}, b) \\
\mathsf{BSeg}(v_\mathrm{t}, v_\mathrm{t}, \emptyset) &\triangleq \mathsf{true} \\
\mathsf{BSeg}(v_1, v_\mathrm{t}, b^v) &\triangleq v_1 \in b^v * \exists v_2.\ v_1.\mathsf{nextB} \mapsto v_2 * \mathsf{BSeg}(v_2, v_\mathrm{t}, b^v \setminus \{v_1\})
\end{aligned}
$$

EList, ESeg are defined like BList, BSeg.

Figure 5 – Definition of L and related predicates. The $\circledast$ operator is *iterated separating conjunction* [Rey02].

But for such a definition of V, Equation (1) will not hold. The issue is that it permits the asserter of V to write to bgn and end, allowing him to "unhook" the view from its underlying list and place it somewhere else in memory. The challenge is to somehow ensure that the sentinel nodes have not changed when it is time to convert the V predicate back to L.

One way to solve this problem could be to provide clients with the weaker but often sufficient Equation (2) instead of (1). That could be be proved by induction over the command c, showing that c will not modify $v.\mathsf{bgn}$ or $v.\mathsf{end}$ because any write to these fields would either be denied because the fields are private, or it would happen through one of the methods of View, whose specifications would have to be strengthened to guarantee that they do not modify to those fields either.

Clearly, this approach is problematic. It requires reasoning about field access modifiers in the logic, which has not been formalized in the separation logics found in the literature to date. It also provides a weaker guarantee to the client, and it lacks modularity because we cannot add or change methods without invalidating the proof.

To find a definition of V that validates (1), look back to the original issue: the asserter of V must be able to read the bgn and end fields but not write them. A popular approach to expressing this is to amend the assertion logic with *fractional permissions* [BCOP05, BRZ07, HH08], a technique borrowed from concurrent programming that turns out to be useful for sequential programs as well [BRZ07, HH08].

In separation logic with fractional permissions, the *points-to* assertion $x.f \mapsto a$ is extended to read $x.f \overset{z}{\mapsto} a$, where $0 < z \leq 1$. A permission $z = 1$ gives read/write access to the field $x.f$, while any smaller permission gives read-only access. The assertion logic is then defined such that the following is valid: $x.f \overset{z}{\mapsto} a * x.f \overset{z'}{\mapsto} a \iff x.f \overset{z+z'}{\longmapsto} a$.

Now we can give a definition of V that works by modifying (4) to take away half

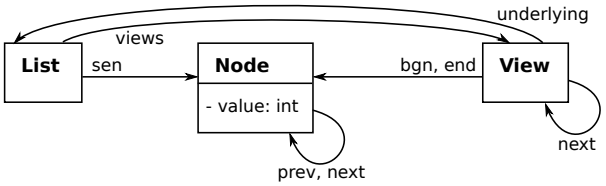

Figure 6 – A class diagram that, compared to Figure 2, more closely resembles the data structure found in the original C5 library [KS06].

of the permissions to the bgn and end fields:

$$\mathsf{V}(v,\alpha) \triangleq \mathsf{wf}(\alpha) * \exists b,\beta,e.\ \alpha = b\beta e\ * \exists n_\mathrm{b}, n_\mathrm{e}. \\ \left[v.\mathsf{bgn} \overset{0.5}{\mapsto} n_\mathrm{b} * v.\mathsf{end} \overset{0.5}{\mapsto} n_\mathrm{e} \mathrel{-\!\!\circledast} \mathsf{I}(n_\mathrm{b}, n_\mathrm{e}, b^v \beta e^v)\right]$$

Here, septraction [CPV07] ($-\!\!\circledast$) is used to "subtract" the permissions on its left side from those on its right side.

Septraction is not essential for this definition, but it does make it much more elegant than it would have been otherwise. Since we are using an intuitionistic separation logic, the standard definitions of septraction developed for classical versions of the logic do not work. In Appendix A, we describe a definition of septraction that works in intuitionistic separation logic.

With the above definition of V, Equation (1) can be proved valid – see Appendix B. It is an important point that the client does not need to know that fractional permissions are being used behind the scenes; clients may reason entirely as if there were no fractions.

5.1 Discussion

It turns out that (1) and (2) can be generalized to treat more than one view. For example, given a list with two non-overlapping views v_1 and v_2, one can mutate those views independently ($\mathsf{V}(v_1,\beta_1) * \mathsf{V}(v_2,\beta_2)$) and later establish that L holds for their underlying list and a suitably-modified bex-list.

With the implementation we have discussed so far, this generalization is straightforward to prove; see Appendix B. The data structure used in the original C5 code is different, however: it has a *global* list of views for the whole LLWV rather than for each node. This is illustrated in Figure 6. The two implementations are observationally indistinguishable through their public interface, but the implementation with a global view-list does not seem to allow defining a V-predicate that admits the assertion $\mathsf{V}(v_1,\beta_1) * \mathsf{V}(v_2,\beta_2)$ for non-overlapping views.

This is because operations on v_1 and v_2 such as item insertion and removal will have to traverse the same global list of views; operations to create and dispose views are even going to modify this list, so it cannot just be shared read-only using fractional permissions.

Since a separation logic assertion describes concrete heap contents, the validity of assertions such as (1) depends only on the choice of data structure in the implementation, not on the code in methods. On the other hand, in object-oriented languages there is a difference between (a) objects that cannot interfere with each other in any observable way and (b) objects that have disjoint heap footprints. Clearly, (b) implies (a), but the converse does not hold because of encapsulation: interference can be

Method	S	I	V	F
+List::init	✓	✓		
+l.count	✓	✓		
+l.getValue	✓	✓	✓	
+l.isEmpty	✓	✓		
+l.insert	✓	✓	✓[2]	
+l.insertFirst		✓		
+l.insertLast		✓		
+l.remove	✓	✓	✓	
+l.removeFirst		✓		
+l.removeLast		✓		
+l.setValue	✓			
+l.span	✓			✓
+l.view	✓	✓	✓	
−l.getNode	✓	✓	✓	
−l.insertNode	✓	✓		✓
−View::init/0	✓	✓		
−View::init/2	✓	✓	✓[2]	
+v.atEnd	✓			
+v.count	✓	✓		
+v.dispose	✓			
+v.getValue	✓	✓		✓
+v.isEmpty	✓	✓		
+v.insert	✓	✓		✓

Method	S	I	V	F
+v.insertFirst		✓		
+v.insertLast		✓		
+v.remove	✓	✓		✓
+v.removeFirst		✓		
+v.removeLast		✓		
+v.setValue	✓			✓
+v.slide/1	✓	✓		✓
+v.slide/2	✓	✓		✓
+v.span	✓			✓
+v.view	✓	✓		✓
−v.getNode	✓	✓		✓
−v.insertNode	✓	✓		✓
−Node::init	✓	✓		
−n.addViewB	✓	✓		✓
−n.addViewE	✓	✓		
−n.insertAfter	✓	✓		✓
−n.moveViewsToB	✓	✓	✓	
−n.moveViewsToE/1	✓	✓	✓[1]	
−n.moveViewsToE/2	✓	✓		✓
−n.remove	✓	✓	✓	
−n.removeViewB	✓	✓		
−n.removeViewE	✓	✓		

Notes:
1: Has been verified in the sense that its twin method with b's instead of e's has been verified.
2: The body of this method has been verified, but it contains calls to unverified methods.

Table 1 – Overview of what methods have been specified (S), implemented (I), verified (V), and which would make non-trivial future work (F). Public methods are prefixed with +, private methods are prefixed with −, and overloaded methods are suffixed with /n to refer to the n-argument version.

observed only through a data structure's public methods, whereas the actual sharing in the heap is described by its (private) fields. In separation logic as we use it here, assertions about objects that are "observationally separate" cannot be separated by the $*$ connective, although this connective is critical for reasoning.

Therefore, a guideline for writing code to be modularly verified with separation logic is to design data structures such that observationally separate parts of the data structure are also disjoint on the heap, whenever possible.

Finding a good way to lift this limitation is likely to be crucial in reasoning about real-world code. The ideas presented in the recent work of Dinsdale-Young et al. [DYDG$^+$10] look promising in this respect. The intuition seems to be that access to shared data is governed by a protocol, and this protocol can be as simple as requiring read-only access or perhaps as complex as required to solve this problem.

5.2 Method bodies

So far we have focused on method specifications and heap layout. To have a complete verified library of LLWV, one of course needs to write the Java code implementing each method and prove that it satisfies its specification in a dialect of separation logic that has been shown sound with respect to the programming language semantics. To get a feeling for whether the specifications are practically useful, sample client code should be verified.

Table 1 summarises which methods were specified, implemented and verified, including several private methods in class Node that we do not discuss here [Jen10]. The main accomplishment is that List::remove and everything it calls has been verified. Two sample clients have been verified: an implementation of Bubble Sort, sliding views of length 2 across the list, has been verified for safety, and a toy example that uses views to duplicate and increment every list item has been verified for correctness.

Method body proofs were done by hand and are presented as code interleaved with assertions in [Jen10]. Bex-lists and the most often used lemmas about them were formalised in the Coq proof assistant.

6 Alternative models

We chose the model and specifications in Section 4 with the following goals in mind.

- The model should admit short and clear specifications: it should be easy to see whether the intended meaning of an operation is expressed by its specification.
- Sequential composition should be straightforward: postconditions and preconditions should have the same form to make it easy to show that one operation's postcondition implies the next operation's precondition.
- The model should admit local reasoning: effects that are local in the implementation should also look local in the model.
- The specification should highlight the similarity between lists and views. Lists and views can be used interchangeably in many situations, so the reasoning in those cases should also be the same.

We believe that the bex-list model and the corresponding specifications achieve these goals. For comparison, we will here discuss some alternatives we considered before settling on bex-lists.

6.1 First Attempt

A straightforward way of modelling lists with views is to separate the model α into three components (L, B, E): a traditional cons-based list L of items, and maps B and E assigning to each view the offset in L where the view begins and ends. Formally,

$$\alpha = (L, B, E) \in \mathsf{int\ list} \times (\mathsf{View} \xrightarrow[\mathsf{fin}]{} \mathbb{N}) \times (\mathsf{View} \xrightarrow[\mathsf{fin}]{} \mathbb{N})$$

However, models that involve indices seem to lead to specifications that fail with respect to all four goals listed above. For example, the following attempt to specify remove, where $(\cdot)$ is list concatenation, leads to a postcondition that requires the

verifier to reason about subtraction.

$$
\begin{array}{l}
\{\, \mathsf{L}(l, (L \cdot x{::}L', B, E)) \,\} \\
l.\mathsf{remove}(|L|) \\
\{\, \exists B', E'.\ \mathsf{L}(l, (L \cdot L', B', E')) \wedge \\
\quad B' = \{[v \mapsto \text{if } j \le |L| \text{ then } j \text{ else } j-1] \mid [v \mapsto j] \in B\} \wedge \\
\quad E' = \{[v \mapsto \text{if } j \le |L| \text{ then } j \text{ else } j-1] \mid [v \mapsto j] \in E\} \,\}
\end{array}
$$

The specification also lacks locality: it is not clear from the specification that the update only affects the immediate vicinity of x since apparently all indices above x are decremented; however this reflects a property of the model, not the implementation.

6.2 Second Attempt

The first attempt above can be used as a basis of something better, though. First, lift the length function of cons-based lists to work for the whole model $\alpha = (L, B, E)$, defining $|\alpha| \triangleq |L|$. Then define concatenation $\alpha_1 \cdot \alpha_2$ of models:

$$
\begin{array}{rl}
\alpha_1 \cdot \alpha_2 \triangleq & \big(L_1 \cdot L_2,\ \ B_1 \uplus (\mathsf{map}\ (+|L_1|)\ B_2), \\
& \qquad\qquad E_1 \uplus (\mathsf{map}\ (+|L_1|)\ E_2)\big) \\
& \text{where } (L_i, B_i, E_i) = \alpha_i \text{ for } i \in \{1, 2\} \\
& \text{and } \uplus \text{ is union of maps with disjoint domains.}
\end{array}
$$

Note that model concatenation $\cdot$ is associative.

Finally, observe that any model $\alpha = (L, B, E)$ can be written as a concatenation of four basic building blocks:

$\epsilon \triangleq (\mathsf{nil}, \{\}, \{\})$	Empty list
$\underline{x} \triangleq (x :: \mathsf{nil}, \{\}, \{\})$	Single-item list
$[_v \triangleq (\mathsf{nil}, \{[v \mapsto 0]\}, \{\})$	View begins
$]_v \triangleq (\mathsf{nil}, \{\}, \{[v \mapsto 0]\})$	View ends

With these ingredients, we can give concise specifications to most methods. In particular, remove looks much better than in the first attempt, and also more local and concise than when using bex-lists (Section 4.2):

$$\{\, \mathsf{L}(l, \alpha \cdot \underline{x} \cdot \gamma) \,\}\ l.\mathsf{remove}(|\alpha|)\ \{\, \mathsf{L}(l, \alpha \cdot \gamma) \,\}$$

Most other specifications resemble their bex-list cousins. For example,

$$
\begin{array}{l}
\{\, \mathsf{L}(l, \alpha \cdot \underline{x} \cdot \gamma) \,\}\ l.\mathsf{setValue}(|\alpha|, x')\ \{\, \mathsf{L}(l, \alpha \cdot \underline{x'} \cdot \gamma) \,\} \\
\{\, \mathsf{L}(l, \alpha \cdot \beta \cdot \gamma) \,\}\ l.\mathsf{view}(|\alpha|, |\beta|)\ \{\, \mathsf{L}(l, \alpha \cdot [_{ret} \cdot \beta \cdot]_{ret} \cdot \gamma) \,\}
\end{array}
$$

The drawbacks of this model are due to $\alpha \cdot \beta = \beta \cdot \alpha$ when $|\alpha| = |\beta| = 0$. This equality allows us to freely re-order views analogously to how the sets b and e in the bex-list model allow it. However, to correctly specify insert we need a restrictive variant $\circ$ of concatenation in the precondition:

$$\{\, \mathsf{L}(l, \alpha \circ \beta) \,\}\ l.\mathsf{insert}(|\alpha|, x)\ \{\, \mathsf{L}(l, \alpha \cdot \underline{x} \cdot \beta) \,\}$$

Using the normal concatenation operator $\cdot$ in place of $\circ$ in the precondition would allow the verifier to "choose" whether a view that begins or ends at the insertion point would end up to the left or right of the inserted item x. But since those two outcomes are observably different, such a specification would allow deriving a logical contradiction.

Instead, we define $\circ$ to be an (even more) partial variant of concatenation. Intuitively, a list that can be written $\alpha_1 \circ \alpha_2$ must have no views that end just before the first item in α_2, and any view that begins just after the last item of α_1 must be empty. Formally,

$$\begin{aligned}\alpha_1 \circ \alpha_2 \triangleq\ & \alpha_1 \cdot \alpha_2 \\ & \text{if } \forall v.\ [v \mapsto 0] \notin E_2 \\ & \text{and } \big([v \mapsto |L_1|] \in B_1 \Rightarrow [v \mapsto |L_1|] \in E_1\big) \\ & \text{where } (L_i, B_i, E_i) = \alpha_i \text{ for } i \in \{1, 2\}\end{aligned}$$

Thus, the specification of insert is only beautiful because it hides its complexity beneath the definition of $\circ$. The verifier would have to develop a theory to establish $\circ$ before every call to insert. But the approach is not very general since $\circ$ is specific to the semantics of the insert operation. If some other variant of insert were introduced, there would have to be another restricted concatenation operator with a corresponding theory.

The same problem of herding the views onto the desired side of a concatenation appears if we want to formulate a theorem such as (1), which is crucial for independent reasoning about views. It is also no longer possible to define L to be as syntax-directed on α as for bex-lists, which makes it harder to prove the implementation correct.

It was our desire to syntactically restrict where views may begin and end that made us abandon this model in favour of the bex-lists, which make explicit the grouping of all views that begin or end at each list position. The specifications that arose from the "second attempt" model remain more elegant and intuitive, though, and it would be interesting to investigate whether it could work well if the semantics of LLWV were changed.

7 Future work

The future work specific to the LLWV data structure includes:

- The semantics of inserting new items in a LLWV [KS06] can easily lead to surprises since nearby views can be affected [Jen10]. Thus, it should be investigated whether views, and insertion in particular, could be defined differently, and if so, whether something better than the bex-list model can be found.
- In verifying both the LLWV implementation and sample clients [Jen10], proofs often required solving equations in ordered bex-lists. The solutions were often intuitively simple but somewhat laborious to prove formally. It seems likely that a decision procedure could be developed for a useful fragment of these equations.
- As discussed in Section 2, the "list with views" abstraction can be applied to both linked lists and array lists. It remains future work to formally verify the array list case. Also, C5 has a variation of LLWV that uses *hash-indexes* to implement operations such as deciding whether a given item resides within a

given view in constant expected time. Verification of this would surely be a challenge.

- It might be advantageous to replace the C5 library's current implementation of views (mentioned in Section 5.1) with that presented in Section 3, which seems to have some algorithmic advantages. To support the extension to hash-indexed lists and views mentioned above, it would need to have distinct start and end sentinels, though.
- Could some of these ideas be applied to specifying powerful iterators such as java.util.ListIterator? Java's list iterators permit more modifications to the list than iterators studied elsewhere in the separation logic literature, though they are still less powerful than views.
- It would be interesting to give a specification of LLWV in other recent logics for shared mutable data structures, e.g., region logic [BNR08], and compare with the present formulation.

The remaining points concern improvements to the logic motivated by insights gained from this case study.

- Current dialects of separation logic do not take advantage of the guarantees offered by memory-safe languages such as Java and C#. As discussed in Section 5.1, separation logic works as if all fields were public; it would be interesting to integrate reasoning about field access modifiers into the logic.
- Fractional permissions proved useful here, but they seem to be a somewhat blunt instrument when used in a sequential setting. Their read-only guarantee can only be applied at the granularity of a field, so it is impossible to express invariants such as the least significant bit of a field being read-only.
- The original C5 implementation of LLWV employs the System.WeakReference class to let lists point to their views through references that are ignored by the garbage collector. Modelling such weak references in separation logic might be interesting future work.

8 Conclusion

Several things can be done when implementing a data structure to ease verification with separation logic. When modularity is desired, data should be laid out such that heap separation coincides with lack of observable interference. Modularity and local reasoning demand more features from the logic, such as existentially-quantified predicates and fractional permissions, but in return they lead to cleaner specifications.

The bex-list model was chosen over other candidates because it better satisfied the goals listed in Section 6. It seems that there is a balance between choosing a model that is easy to verify and one that is easy to work with for clients: with a model that directly mimics the heap layout, the implementation will be easier to prove correct, but clients are likely to find the model unnatural to work with. The bex-list aims to be a compromise between the two extremes.

Further discussion and subtleties can be found in [Jen10].

References

[BBTS05] B. Biering, L. Birkedal, and N. Torp-Smith. BI hyperdoctrines and higher-order separation logic. In *Proceedings of ESOP*, 2005.

[BCOP05] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Proceedings of POPL*, 2005.

[Blo08] Joshua Bloch. *Effective Java Programming Language Guide*. Addison-Wesley, second edition, 2008.

[BNR08] A. Banerjee, D. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *Proceedings of ECOOP*, 2008.

[BRZ07] John Boyland, William Retert, and Yang Zhao. Iterators can be independent "from" their collections. In *Proceedings of ECOOP*, 2007.

[CKLP06] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and Objects (FMCO) 2005. Lecture Notes in Computer Science, vol. 4111*, pages 342–363, 2006.

[CPV07] Cristiano Calcagno, Matthew J. Parkinson, and Viktor Vafeiadis. Modular safety checking for fine-grained concurrency. In *Proceedings of SAS*, 2007.

[Dij76] E.D. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[DYDG+10] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *Proceedings of ECOOP*, 2010.

[FBL10] Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In *ACM Symposium on Applied Computing, March 2010, Sierre, Switzerland*, 2010.

[HH08] Christian Haack and Clément Hurlin. Resource usage protocols for iterators. In *Proceedings of IWACO*, 2008.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[IO01] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. *SIGPLAN Not.*, 36(3):14–26, 2001.

[Jen10] Jonas B. Jensen. Specification and validation of data structures using separation logic. Master's thesis, Technical University of Denmark, February 2010.

[KAB+09] Neelakantan R. Krishnaswami, Jonathan Aldrich, Lars Birkedal, Kasper Svendsen, and Alexandre Buisse. Design patterns in separation logic. In *Proceedings of TLDI*, 2009.

[KS06] Niels Kokholm and Peter Sestoft. The C5 generic collection library for C# and CLI. Technical Report ITU-TR-2006-76, IT University of Copenhagen, 2006.

[Mey92] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, October 1992.

[Par05] Matthew Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, 2005.

[Par07] Matthew Parkinson. Class invariants: The end of the road? In *International Workshop on Aliasing, Confinement and Ownership in Object-Oriented programming (IWACO), Berlin, Germany*, 2007. At http://people.dsv.su.se/~tobias/iwaco/accepted.html.

[PB05] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *Proceedings of POPL*, 2005.

[PB08] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *Proceedings of POPL*, 2008.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of LICS*, 2002.

[Rey08] John C. Reynolds. An introduction to separation logic (preliminary draft). Course notes, October 2008.

[SBP10] Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. Verifying generics and delegates. In *Proceedings of ECOOP*, 2010.

A Septraction

In this appendix we define the septraction operator, $-\!\circledast$, which was used in the definition of V.

A.1 Formal set-up

Our heap model is identical to the model of heaps with fractional permissions in [BCOP05]:

$$\mathit{Heap} \triangleq \mathit{ObjId} \times \mathit{Field} \xrightarrow[\text{fin}]{} \mathit{Val} \times \{\pi \in \mathbb{Q} \mid 0 < \pi \leq 1\}$$

When $h_1, h_2 \in \mathit{Heap}$ we write $h_1 \# h_2$ to say that the composition of h_1 and h_2 is defined, and $h_1 \circ h_2$ denotes this composition. There is an ordering on heaps defined as $h_1 \sqsubseteq h$ iff $\exists h_2 \# h_1.\ h_1 \circ h_2 = h$.

The separation logic used throughout this article is intuitionistic, meaning that all formulas satisfy the *monotonicity condition*: they must continue to hold in any larger heap [IO01]. Thus, an assertion is a monotone function from heaps to Booleans, where the Booleans are ordered as $\mathsf{false} \sqsubseteq \mathsf{true}$.

A.2 The septraction operator

To get an intuitionistic septraction connective that has most of the desirable properties of its classical cousin [CPV07], we use the following definition.

Definition 1

$$(P -\!\circledast Q)\ h \triangleq \exists h_0 \sqsubseteq h.\ \exists h' \# h_0.\ \mathsf{Pr}\ P\ h' \wedge Q\ (h_0 \circ h') \qquad \text{where}$$
$$\mathsf{Pr}\ P\ h \triangleq P\ h \wedge \forall h' \sqsubseteq h.\ P\ h' \Rightarrow h' = h$$

■

Note that this definition satisfies the monotonicity condition since the same subheap h_0 continues to exist under any larger heap.

The operator $\mathsf{Pr} : (\mathit{Heap} \underset{\text{mon}}{\rightarrow} \mathbf{2}) \rightarrow (\mathit{Heap} \rightarrow \mathbf{2})$, where $\mathbf{2}$ denotes the Booleans, is Yang's *precising* operator (see the discussion of this operator in [Rey08]). In particular, $\mathsf{Pr}\ P\ h$ holds if h is a minimal heap such that $P\ h$.

To formulate the inference rules for septraction, we need the notion of a strongly supported assertion:

Definition 2 An assertion P is *strongly supported* if for any heap h, the set of subheaps of h that satisfy P is either empty or has a least element. ■

All the assertions in Figure 5 are strongly supported (modulo a few side conditions; see [Jen10] for details).

Lemma 1. *For strongly supported assertions P,*

$$\frac{Q \models P}{Q \models P * (P \mathrel{-\!\!\circledast} Q)} \qquad \textit{and} \qquad \frac{Q \models P * Q'}{P \mathrel{-\!\!\circledast} Q \models Q'}$$

where $\models$ denotes entailment among assertions; that is, $P \models Q$ iff for all h, $P\ h$ implies $Q\ h$.

Proof. Both proofs rely on the fact that if $P\ h$ then there exists $h_{\min} \sqsubseteq h$ such that $\mathsf{Pr}\ P\ h_{\min}$. □

A.3 Remark

An alternative to Definition 2 is the following:

Definition 3 An assertion P is *weakly supported* if for any heaps h_1, h_2 that are both subheaps of the same h and both satisfy P, there exists a heap h_{12} that is a subheap of both h_1 and h_2 and satisfies P. ■

Definitions 2 and 3 are usually equivalent and therefore used interchangeably [Rey08], but it turns out that they are not the same when fractional permissions are used. For example, the assertion $\exists z > 0.\ x.f \overset{z}{\mapsto} y$ is weakly but not strongly supported. This assertion is useful and natural since it represents a read-only points-to assertion that can be arbitrarily split across $*$.

B Conversions between lists and views

In this appendix we prove Equation (1) and its generalization to multiple views.

For brevity, we abbreviate a bex-list of the form $b\beta e$ as B and write B^v to mean $b^v \beta e^v$. Recall that the notation b^v means $b \cup \{v\}$ where $v \notin b$.

The following lemma captures the essence of why Equation (1) is valid.

Lemma 2. *If* $\mathsf{ord}(\alpha, \mathsf{B}, \mathsf{B})$ *and* $\mathsf{wf}(B)$*, then*

$$\mathsf{I}(n_1', n_2, \alpha B^v) \models \mathsf{V}(v, B) * \left[\mathsf{V}(v, B') \mathrel{-\!\!*} \mathsf{I}(n_1', n_2, \alpha B'^v)\right]$$

Proof. The left part expands to $\mathsf{I}(n_1', n_1, \alpha) * \mathsf{I}(n_1, n_2, B^v)$ for some n_1, and it is the right part of this conjunction that is interesting to us. By Lemma 1, that part entails $v.\mathsf{bgn} \overset{0.5}{\mapsto} n_1 * v.\mathsf{end} \overset{0.5}{\mapsto} n_2 * (v.\mathsf{bgn} \overset{0.5}{\mapsto} n_1 * v.\mathsf{end} \overset{0.5}{\mapsto} n_2 \mathrel{-\!\circledast} \mathsf{I}(n_1, n_2, B^v))$, where the side condition on the lemma follows from expanding the I predicate and its constituents. In that formula, the septraction part is just the definition of V, so we can contract that and get $v.\mathsf{bgn} \overset{0.5}{\mapsto} n_1 * v.\mathsf{end} \overset{0.5}{\mapsto} n_2 * \mathsf{V}(v, B)$.

If we can now show that

$$v.\mathsf{bgn} \overset{0.5}{\mapsto} n_1 * v.\mathsf{end} \overset{0.5}{\mapsto} n_2 \models \mathsf{V}(v, B') \mathrel{-\!\!*} \mathsf{I}(n_1, n_2, B'^v), \tag{5}$$

then we have altogether that

$$\mathsf{I}(n_1', n_2, \alpha B^v) \models \mathsf{I}(n_1', n_1, \alpha) * \mathsf{V}(v, B) * \big[\mathsf{V}(v, B') \mathrel{-\!\!*} \mathsf{I}(n_1, n_2, B'^v)\big],$$

and since $Q' * (P \mathrel{-\!\!*} Q) \models P \mathrel{-\!\!*} (Q * Q')$ we may then join the two I predicates by Equation (3) to finish the proof.

To show (5), first apply the fact that $Q \models P \mathrel{-\!\!*} (P * Q)$ to get $\mathsf{V}(v, B') \mathrel{-\!\!*} (\mathsf{V}(v, B') * v.\mathsf{bgn} \overset{0.5}{\mapsto} n_1 * v.\mathsf{end} \overset{0.5}{\mapsto} n_2)$. On the right of the separating implication, unfold the definition of V to replace $\mathsf{V}(v, B')$ with $v.\mathsf{bgn} \overset{0.5}{\mapsto} n_\mathrm{b} * v.\mathsf{end} \overset{0.5}{\mapsto} n_\mathrm{e} \mathrel{-\!\circledast} \mathsf{I}(n_\mathrm{b}, n_\mathrm{e}, B'^v)$ for new existentials n_b and n_e. Since there is an assertion I', too long to write out here, such that $\mathsf{I}(n_\mathrm{b}, n_\mathrm{e}, B'^v) \equiv I' * v.\mathsf{bgn} \mapsto n_\mathrm{b} * v.\mathsf{end} \mapsto n_\mathrm{e}$, and we can split the permissions on $v.\mathsf{bgn}$ and $v.\mathsf{end}$ in two halves, by the second half of Lemma 1 we can get $I' * v.\mathsf{bgn} \overset{0.5}{\mapsto} n_\mathrm{b} * v.\mathsf{end} \overset{0.5}{\mapsto} n_\mathrm{e}$. Recall that this is still in separating conjunction with $v.\mathsf{bgn} \overset{0.5}{\mapsto} n_1 * v.\mathsf{end} \overset{0.5}{\mapsto} n_2$, which lets us conclude that $n_1 = n_\mathrm{b}$ and $n_2 = n_\mathrm{e}$. Now we can join the split permissions and contract the I predicate again to get $\mathsf{I}(n_1, n_2, B'^v)$, which is what we wanted. □

In the following, let $i \geq 1$ and $m \geq 0$. The $\circledast$ operator binds tighter than $*$.

Lemma 3. *If* $\mathsf{ord}(\alpha_i, \mathsf{B}, \mathsf{B})$ *and* $\mathsf{wf}(B_i)$ *for all* i*, then*

$$\mathop{\circledast}_{i \leq m} \mathsf{I}(n_i, n_{i+1}, \alpha_i B_i^{v_i}) \models \mathop{\circledast}_{i \leq m} \mathsf{V}(v_i, B_i) * \big[\mathop{\circledast}_{i \leq m} \mathsf{V}(v_i, B_i') \mathrel{-\!\!*} \mathop{\circledast}_{i \leq m} \mathsf{I}(n_i, n_{i+1}, \alpha_i B_i'^{v_i})\big]$$

Proof. By induction on m. The base case is trivial. For the inductive case, let us first abbreviate the above formula to read

$$I(\leq m) \models V(\leq m) * [V'(\leq m) \mathrel{-\!\!*} I'(\leq m)]$$

Thus, we start out with $I(\leq m)$, which entails $I(< m) * I(m)$, and applying the induction hypothesis to the left part gives us $\big(V(< m) * [V'(< m) \mathrel{-\!\!*} I'(< m)]\big) * I(m)$. For the remaining part, since Lemma 2 gives us that

$$I(m) \models V(m) * [V'(m) \mathrel{-\!\!*} I'(m)],$$

then the whole entails $I(\leq m) \models V(\leq m) * [V'(\leq m) \mathrel{-\!\!*} I'(\leq m)]$ due to the fact that $(P \mathrel{-\!\!*} Q) * (P' \mathrel{-\!\!*} Q') \models (P * P') \mathrel{-\!\!*} (Q * Q')$. □

Theorem 1.

$$\mathsf{L}(l, \alpha_1 B_1^{v_1} \cdots \alpha_m B_m^{v_m} \gamma) \models \mathop{\circledast}_{i \leq m} \mathsf{V}(v_i, B_i) * \big[\mathop{\circledast}_{i \leq m} \mathsf{V}(v_i, B_i') \mathrel{-\!\!*} \mathsf{L}(l, \alpha_1 B_1'^{v_1} \cdots \alpha_m B_m'^{v_m} \gamma)\big]$$

Proof. By Lemma 3 and Equation (3) □

Corollary 1. *Equation* (1) *is valid; i.e.,*

$$\mathsf{L}(l, \alpha b^v \beta e^v \gamma) \implies \mathsf{V}(v, b\beta e) * \big[\mathsf{V}(v, b'\beta' e') \mathrel{-\!\!*} \mathsf{L}(l, \alpha b'^v \beta' e'^v \gamma)\big]$$

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 $C\sharp$. 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 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, $UPred(heap)$ 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).

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

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:

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

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 $(s, h_1, \tilde{c}) \not\rightsquigarrow^n \mathbf{err}$ 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].

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

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.

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

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:

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

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:

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

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]).

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

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 $v.f \mapsto v' \triangleq \{(h, n) \mid h \sqsupseteq [(v, f) \mapsto v']\}$.

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

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

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

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

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

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

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

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:

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

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

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

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

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

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.

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

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.

Fictional Separation Logic

Jonas Braband Jensen and Lars Birkedal

IT University of Copenhagen

Abstract. Separation logic formalizes the idea of local reasoning for heap-manipulating programs via the frame rule and the separating conjunction $P * Q$, which describes states that can be split into *separate* parts, with one satisfying P and the other satisfying Q. In standard separation logic, separation means physical separation. In this paper, we introduce *fictional separation logic*, which includes more general forms of fictional separating conjunctions $P * Q$, where $*$ does not require physical separation, but may also be used in situations where the memory resources described by P and Q overlap. We demonstrate, via a range of examples, how fictional separation logic can be used to reason locally and modularly about mutable abstract data types, possibly implemented using sophisticated sharing. Fictional separation logic is defined on top of standard separation logic, and both the meta-theory and the application of the logic is much simpler than earlier related approaches.

Keywords: Separation Logic, Local Reasoning, Modularity.

1 Introduction

Separation logic is a kind of Hoare logic for *local* reasoning about programs with shared mutable state. Locality is achieved by use of the $*$ connective and the frame rule:

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}$$

Recall that in standard separation logic, $P * R$ is satisfied by a heap if it can be split into two *separate* (disjoint) parts satisfying P and R respectively. The frame rule expresses that if command C is well-specified with precondition P and postcondition Q, then C will preserve any disjoint invariant R, intuitively (and formally in standard models) because of physical heap *separation*.

In many situations, however, physical separation is too strong a requirement – we would like to be able to reason locally using $*$-connectives and frame rules in situations where we do not have physical separation, but where we do have some form of logical or *fictional separation*[1]. The key idea is that fictional separation should allow us to reason separately about updates to shared resources, as long as the updates follow some kind of discipline to guarantee that updates to one

[1] The term "fictional separation" is derived from the phrase "fiction of disjointness", which, to the best of our knowledge, was introduced by Philippa Gardner [8].

side of the $*$ do not affect the truth of the other side. Permission accounting models [5, 4, 10] provide a familiar simple instance of this idea: they allow us to reason separately about shared heaps as long as we do not update but only read those heaps. In recent work on separation logic for concurrency [7, 11] and for abstraction [8, 9], it is possible to describe more elaborate patterns of sharing. We return to this when we discuss related work in Section 7.

In this paper we introduce *fictional separation logic* and demonstrate, via examples, how the logic can be used to reason locally and modularly about mutable abstract data types, possibly implemented using sophisticated sharing.

Before turning to the technical presentation, we consider a simple example.

Example: Bit Pair. Consider a small library for manipulating pointers to bit pairs. It has a constructor, destructor and some accessors that conform to the following specification in standard higher-order separation logic [2]:

$$\begin{aligned}
&\exists B_1, B_2 : \mathit{loc} \times \mathit{bool} \to \mathcal{P}(\mathit{heap}).\\
&\quad \{\mathit{emp}\}\ \mathsf{bp_new}()\ \{B_1(\mathsf{ret}, \mathit{false}) * B_2(\mathsf{ret}, \mathit{false})\} \wedge\\
&\quad \{B_1(\mathsf{p}, _) * B_2(\mathsf{p}, _)\}\ \mathsf{bp_free}(\mathsf{p})\ \{\mathit{emp}\} \wedge\\
&\quad \forall i \in \{1, 2\}.\ (\forall b.\ \{B_i(\mathsf{p}, b)\}\ \mathsf{bp_get}i(\mathsf{p})\ \{B_i(\mathsf{p}, b) \wedge \mathsf{ret} = b\}) \wedge\\
&\qquad\qquad \{B_i(\mathsf{p}, _)\}\ \mathsf{bp_set}i(\mathsf{p}, \mathsf{b})\ \{B_i(\mathsf{p}, \mathsf{b})\}.
\end{aligned}$$

Note the use of existential quantification over representation predicates B_1 and B_2; they correspond to what Parkinson and Bierman call *abstract predicates* [18]. The special variable ret in postconditions denotes the return value. As usual, the underscore is used for an existentially-quantified variable.

Implementing this naïvely and verifying the implementation is straightforward in standard separation logic. Simply let the constructor allocate two consecutive heap cells and let the accessors dereference either their p parameter or $\mathsf{p} + 1$. For the verification, instantiate $B_i(p, b)$ to $(p + (i - 1)) \mapsto b$.

But this implementation uses at least twice as much heap space as necessary. The least we could do is to allocate only one (integer) heap cell and store the pair of bits in its least significant bits. A possible implementation is the following, where / denotes integer division, and % denotes modulo:

```
bp_new() { p := alloc 1; [p] := 0; return p }
bp_free(p) { free p }
bp_get1(p) { x := [p]; return x % 2 }
bp_get2(p) { x := [p]; return x / 2 }
bp_set1(p,b) { x := [p]; [p] := b + x/2*2 }
bp_set2(p,b) { x := [p]; [p] := 2*b + x%2 }
```

The original specification is unfortunately unprovable for this implementation, even though the two implementations have completely identical behaviour when observed by a client that cannot inspect their internal memory.

The problem is that the abstract module specification is not sufficiently abstract since it requires that the constructor creates two heap chunks that are

physically disjoint. In other words, the abstract module specification reveals patterns of sharing or, as is the case here, lack of sharing, that really ought to be internal to the module implementation. Moving to a heap model with bit-level separation will not solve the essence of this problem. Indeed, a third implementation could store the two bits in an integer that is divisible by 3 when B_1 is true and divisible by 5 when B_2 is true. In this case, the fictional separation comes from arithmetic properties of the integers.

In fictional separation logic, we existentially quantify not only over representation predicates B_1 and B_2, but also over the choice of separation algebra, Σ, and an interpretation map I (explained below). The abstract module specification then looks like this:

$$\begin{aligned}
&\exists \Sigma : \mathit{sepalg}.\ \exists I : \Sigma \searrow \mathit{heap}.\ \exists B_1, B_2 : \mathit{loc} \times \mathit{bool} \to \mathcal{P}(\Sigma).\\
&\quad I.\ \{\mathit{emp}\}\ \mathsf{bp_new}()\ \{B_1(\mathsf{ret}, \mathit{false}) * B_2(\mathsf{ret}, \mathit{false})\} \wedge\\
&\quad I.\ \{B_1(\mathsf{p}, _) * B_2(\mathsf{p}, _)\}\ \mathsf{bp_free}(\mathsf{p})\ \{\mathit{emp}\} \wedge\\
&\quad \forall i \in \{1, 2\}.\ (\forall b.\ I.\ \{B_i(\mathsf{p}, b)\}\ \mathsf{bp_get}i(\mathsf{p})\ \{B_i(\mathsf{p}, b) \wedge \mathsf{ret} = b\}) \wedge\\
&\qquad\qquad I.\ \{B_i(\mathsf{p}, _)\}\ \mathsf{bp_set}i(\mathsf{p}, \mathsf{b})\ \{B_i(\mathsf{p}, \mathsf{b})\}.
\end{aligned}$$

The intention is that the interpretation map I should explain how elements of the separation algebra Σ are represented by predicates on physical heaps.

Note that the Hoare triples are now prefixed by I – we refer to such a predicate $I.\ \{P\}\ C\ \{Q\}$ as an *indirect Hoare triple*. The intention is that I records (1) which separation algebra P and Q should be interpreted over, and (2) how P and Q are translated into physical heap predicates, such that the triple meaningfully corresponds to a suitably translated triple in standard separation logic.

This module specification does not reveal information about sharing or lack of sharing, because Σ and I are abstract, i.e., existentially quantified. Client code can now be verified relative to this abstract module specification and since, as we will show, fictional separation logic supports the standard proof rules (and some additional rules), the verification of client code is as easy as it is in standard separation logic. We return to this example in Section 3.2 and show how both implementations of bit pairs satisfy the above abstract specification. We will consider an example of client code verification in Section 4.1.

Outline. The remainder of this paper is organized as follows. We first present some formal preliminaries in Section 2 and then go on to present four sections on fictional separation logic. In each of these sections, we first describe some theory and then present examples that demonstrate how to use the theory in program verification. Basic fictional separation logic and the indirect triple are defined in Section 3. In Section 4 we define separating products of interpretations, which allow clients to use several modules at the same time, and in Section 5 we define a notion of indirect entailment and show how to use it to define fractional permissions within fictional separation logic. We discuss how to stack several levels of abstraction in Section 6, and we conclude and discuss related work in Section 7. To focus on the core ideas, we present fictional separation logic for

a simple sequential imperative programming language with procedures, but it should be clear that the ideas are applicable to richer programming languages.

Proofs and further examples can be found in the online appendix [14].

2 Formal Preliminaries

2.1 Abstract Assertion Logic

The meaning of separation logic assertions is often parametrized on a *separation algebra* (SA) [6], which is an abstraction of the heap model. There are several competing definitions of separation algebra in the literature [6, 10, 12]; we use the one from [12]:[2]

Definition 1. *A* separation algebra *is a partial commutative monoid* $(\Sigma, \circ, 0)$. *We write* $\sigma \doteq \sigma_1 \circ \sigma_2$ *when the* $\sigma_1 \circ \sigma_2$ *is defined and has value* σ.

Given a separation algebra $(\Sigma, \circ, 0)$, the powerset $\mathcal{P}(\Sigma)$ forms a complete boolean BI algebra, i.e., a model of the assertion language of classical separation logic, where the connectives are defined in the standard way [6]:

$$\top \triangleq \Sigma \qquad \bot \triangleq \emptyset$$
$$P \wedge Q \triangleq P \cap Q \qquad P \vee Q \triangleq P \cup Q$$
$$\forall x : A.\ P(x) \triangleq \bigcap_{x:A} P(x) \qquad \exists x : A.\ P(x) \triangleq \bigcup_{x:A} P(x)$$
$$P \Rightarrow Q \triangleq \{\sigma \mid \sigma \in P \Rightarrow \sigma \in Q\} \qquad \mathit{emp} \triangleq \{0\}$$
$$P * Q \triangleq \{\sigma \mid \exists \sigma_1, \sigma_2.\ \sigma \doteq \sigma_1 \circ \sigma_2 \wedge \sigma_1 \in P \wedge \sigma_2 \in Q\}$$
$$P \mathrel{-\!\!*} Q \triangleq \{\sigma_2 \mid \forall \sigma_1.\ \forall \sigma \doteq \sigma_1 \circ \sigma_2.\ \sigma_1 \in P \Rightarrow \sigma \in Q\}$$

As usual, entailment is defined as $P \vdash Q \triangleq P \subseteq Q$. We refer to the elements of $\mathcal{P}(\Sigma)$ as (semantic) assertions.

2.2 Programming Language

The logic we will introduce in the next section is mostly independent of the underlying programming language, but we will fix a particular language here for clarity. It is a simple imperative language in the style of [20], extended with simple procedures:

$$\begin{aligned} C ::= {} & x := e \mid [e] := e \mid x := [e] \mid x := \mathsf{alloc}\ e \mid \mathsf{free}\ e \\ & \mid C; C \mid \mathsf{if}\ e\ \mathsf{then}\ C\ \mathsf{else}\ C \mid \mathsf{while}\ e\ \mathsf{do}\ C \mid \mathsf{call}\ x := f(\bar{e}) \end{aligned}$$

[2] The original definition of SA [6] also required *cancellativity*: that if $\sigma' \doteq \sigma \circ \sigma_1$ and $\sigma' \doteq \sigma \circ \sigma_2$ then $\sigma_1 = \sigma_2$. This is too restrictive for our purposes, so we do not include it in the general definition of a SA.

The commands are, respectively, assignment, heap write, heap read, allocation, deallocation, sequencing, conditional, loop and function call. The argument to alloc specifies how many consecutive heap locations should be allocated.

There is no module system at the language level. When we talk about a module in this paper, it simply refers to a collection of functions.

The operational semantics of the language is defined in a standard way, using the following memory model:

$$\begin{array}{rlrl} C : \mathit{cmd} & \text{(see above)} & v : \mathit{val} & \triangleq \mathit{loc} \uplus \{\mathsf{null}\} \uplus \mathbb{Z} \uplus \{\mathsf{true}, \mathsf{false}\} \\ x, y : \mathit{var} & \triangleq \mathit{string} & s : \mathit{stack} & \triangleq \mathit{var} \to \mathit{val} \\ f : \mathit{func\_name} & \triangleq \mathit{string} & e : \mathit{expr} & \triangleq \mathit{stack} \to \mathit{val} \\ l : \mathit{loc} & \triangleq \mathbb{N} & h : \mathit{heap} & \triangleq \mathit{loc} \xrightarrow[\mathsf{fin}]{} \mathit{val} \\ \multicolumn{4}{c}{\mathit{program} \triangleq \mathit{func\_name} \xrightarrow[\mathsf{fin}]{} \mathit{var}^* \times \mathit{cmd} \times \mathit{expr}} \end{array}$$

Verification always takes place in an implicit global context of type *program* that maps each function name to a parameter list, function body and return expression. The only type of syntactic entities in this paper is *cmd*. Assertions, specifications, inference rules, and even programming language expressions, are semantic. If desired, a syntactic system could be built on top of this, but it would serve no purpose in this paper.

As usual, *heap* is a separation algebra with composition being the union of disjoint maps and the identity being the empty map. In addition to the connectives from Section 2.1, the separation algebra of heaps also has the *points-to* assertion: $l \mapsto v \triangleq \{[l \mapsto v]\}$. We make this more precise in Section 2.4.

2.3 Specification Logic

A specification $S : \mathit{spec}$ is a logical proposition about the program under consideration. The specification logic has the connectives $(\top, \bot, \wedge, \vee, \forall, \exists, \Rightarrow)$ as operators on *spec* and entailment $(\vdash)$ as a relation on *spec*. These interact according to the standard rules of intuitionistic logic.

We assume that there is a definition of the Hoare triple $\{P\}\ C\ \{Q\} : \mathit{spec}$. Intuitively, if $S \vdash \{P\}\ C\ \{Q\}$, then under the assumptions of S, if the command C runs in a state satisfying P, it will not fault, and if it terminates, the resulting state satisfies Q. The Hoare triple is assumed to satisfy the standard structural and command-specific rules of separation logic [20].

The definition of *spec* and the Hoare triple, as well as the proofs that they satisfy the rules of separation logic, are standard and not important here. See, e.g., [1] for a definition of *spec* that allows for (mutually) recursive procedures and is formalized in Coq.

The assertions P, Q used in the Hoare triple are of type $\mathit{asn}(\mathit{heap})$, where $\mathit{asn}(\Sigma) \triangleq \mathit{stack} \to \mathcal{P}(\Sigma)$. Connectives and rules for $\mathcal{P}(\Sigma)$ can be lifted pointwise to $\mathit{asn}(\Sigma)$, so we will conflate the two in the following.

2.4 Constructing Separation Algebras

In this subsection we record some simple ways of constructing separation algebras, which will be useful in the following.

Given a set A and a SA $(\Sigma, \circ, 0)$, we write $A \xrightarrow{\text{fin}} \Sigma$ for the set of total maps $f : A \to \Sigma$ for which only a finite number of values $a : A$ have $f(a) \neq 0$. That is, f has finite support. The set $A \xrightarrow{\text{fin}} \Sigma$ is itself a SA with composition being pointwise and only defined when the composition in Σ is defined at every point.

We let $[a \mapsto \sigma]$ be the map in $A \xrightarrow{\text{fin}} \Sigma$ which maps a to σ and every other element to 0. Observe that $[a \mapsto 0]$ is the constant 0 map.

For $f : A \xrightarrow{\text{fin}} \Sigma$, define $\mathit{supp}(f) = \{a \mid f(a) \neq 0\}$.

A *permission algebra* (PA) [6] is a partial commutative semigroup; i.e., it is like a SA but may not have a unit element. The product of two PAs (SAs) is also a PA (SA); composition is pointwise, and it is defined only when defined on both components. Any set A can be seen as the *empty PA* ($A_\emptyset$) by letting the composition be undefined for all operands. Moreover, any set A can be seen as the *equality PA* ($A_=$) by letting the composition have $x \circ x \doteq x$ for all x and making it undefined for non-equal operands. Finally, any PA Π can be made into a SA $\Pi_\bot$ by adding a unit element.

In this terminology, the SA of heaps is $\mathit{heap} = \mathit{loc} \xrightarrow{\text{fin}} \mathit{val}_{\emptyset\bot}$.

3 Fictional Separation Logic

The basic idea of fictional separation logic is that assertions are not just expressed in a single separation algebra, chosen in advance to match the programming language, but instead each module may define its own domain-specific SA. Each such SA is interpreted into another SA and eventually to the SA of heaps. Given separation algebras $(\Sigma, \circ_\Sigma, 0_\Sigma)$ and $(\Sigma', \circ_{\Sigma'}, 0_{\Sigma'})$, an *interpretation* I is of type

$$\Sigma \searrow \Sigma' \triangleq \{I : \Sigma \to \mathcal{P}(\Sigma') \mid I(0_\Sigma) = \{0_{\Sigma'}\}\}.$$

The side condition is not strictly necessary but will ease presentation later.[3]

The logic revolves around the *indirect triple*, defined as

$$I.\ \{P\}\ C\ \{Q\} \triangleq \forall \phi.\ \{\exists \sigma \in P.\ I(\sigma \circ \phi)\}\ C\ \{\exists \sigma \in Q.\ I(\sigma \circ \phi)\}.$$

Here I is an interpretation map of type $\Sigma \searrow \mathit{heap}$, and $P, Q : \mathit{asn}(\Sigma)$, for the same SA Σ. The triple and the all-quantifier on the right-hand side are the ones from the standard specification logic (Section 2.3).

As mentioned in Section 2.3, we implicitly lift operators and constants from $\mathcal{P}(\Sigma)$ into $\mathit{asn}(\Sigma)$. In the definition above, the $(\in)$ operator has been lifted in this way for brevity. Following usual practice, there is also an implicit assumption that the partial composition is well-defined. Written out in full detail, the precondition on the right hand side above is the following element of $\mathit{asn}(\mathit{heap})$:

$$\lambda s : \mathit{stack}.\ \{h \mid \exists \sigma \in P(s).\ \exists \sigma' \doteq \sigma \circ \phi.\ h \in I(\sigma')\}.$$

[3] It simplifies the rule CreateL from Figure 1.

The postcondition is similar, only with Q instead of P.

The quantification over all possible abstract frames ϕ bakes the frame rule into the indirect triple definition, much as in [3], except that here the frame is in a more abstract SA.

The standard specification logic structural rules of Consequence, Exists and Frame hold for the indirect triple. For brevity we just show the frame rule:

$$\frac{\mathit{modifies}(C) \cap \mathit{fv}(R) = \emptyset \qquad S \vdash I.\ \{P\}\ C\ \{Q\}}{S \vdash I.\ \{P * R\}\ C\ \{Q * R\}}\ \textsc{Frame}$$

Here, $\mathit{modifies}(C)$ is the set of program variables possibly assigned to by C [20]. The usual rules for control flow commands (if, while, call and (;)) also hold. Proofs and a discussion of the conjunction rule are in the online appendix [14].

The unit interpretation on Σ is simply $1_\Sigma \triangleq \lambda\sigma : \Sigma.\ \{\sigma\}$; it is used to relate the standard separation logic triple to the indirect triple, as expressed by the following rule:[4]

$$\frac{S \vdash 1_{heap}.\ \{P\}\ C\ \{Q\}}{S \vdash \{P\}\ C\ \{Q\}}\ \textsc{Basic}$$

We typically drop the subscript on 1_Σ since it can be inferred from the context.

3.1 Proof patterns

In this subsection we include a couple of rules and lemmas that are often useful for reasoning about examples in fictional separation logic.

In practice, pre- and postconditions are often singletons, possibly conjoined with a *pure assertion*, i.e., one that either contains every σ or no σ. The following rule relates that special case to standard separation logic. The validity of this rule follows easily from the definition of the indirect triple.

$$\frac{p, q \text{ pure} \qquad S \vdash \forall\phi.\ \{I(\sigma \circ \phi) \wedge p\}\ C\ \{I(\sigma' \circ \phi) \wedge q\}}{S \vdash I.\ \{\{\sigma\} \wedge p\}\ C\ \{\{\sigma'\} \wedge q\}}\ \textsc{Enter1}$$

The name of this rule, like all other rules in this paper, suggests reading it from the bottom up; i.e., given a proof obligation matching its conclusion, "enter" the abstract scope by exchanging the conclusion for its premise.

We will see in examples that interpretation functions often follow a particular pattern. The following lemma records useful facts about this pattern. It uses the iterated separating conjunction [20] operator ($\forall_*$), defined as

$$\forall_*\, a \in \{a_1, \ldots, a_n\}.\ P(a) \triangleq P(a_1) * \ldots * P(a_n).$$

Lemma 1. *If $I : (A \xrightarrow{\text{fin}} \Sigma) \searrow heap$ with $I(f) = \forall_*\, a \in supp(f).\ P(a, f(a))$, then*

a. $I(f) * I(g) \dashv\vdash I(f \circ g)$ *if* $supp(f) \cap supp(g) = \emptyset$.
b. $I(f) * I(g) \vdash I(f \circ g)$ *if* $\forall a.\ (P(a, \_) * P(a, \_) \vdash \bot)$.
c. If p, q are pure, then the following rule is valid.

$$\frac{S \vdash \forall\phi.\ \{I([a \mapsto \sigma \circ \phi]) \wedge p\}\ C\ \{I([a \mapsto \sigma' \circ \phi]) \wedge q\}}{S \vdash I.\ \{\{[a \mapsto \sigma]\} \wedge p\}\ C\ \{\{[a \mapsto \sigma']\} \wedge q\}}$$

[4] Double lines mean that the rule can be used both from top to bottom and vice-versa.

3.2 Example: Bit Pair

We now return to the example of bit pairs from Section 1 and explain how to prove that the implementation with sharing meets the abstract module specification. (It is obvious how the naïve implementation meets the specification: choose I to be 1_{heap} and apply BASIC.)

Choose the witnesses of the existentials as follows (we convert freely between *bool* and $\{0,1\}$).

$$\begin{aligned} \Sigma &= loc \xrightarrow{\text{fin}} (\{1,2\} \xrightarrow{\text{fin}} bool_{\emptyset\perp}) \\ I(f) &= \forall_* p \in supp(f).\ supp(f(p)) = \{1,2\} \wedge p \mapsto (f(p)(1) + 2 \cdot f(p)(2)) \\ B_i(p,b) &= \{[p \mapsto [i \mapsto b]]\} \end{aligned}$$

Composition and unit in Σ follows from the constructions in Section 2.4. This resembles the SA used for object-oriented languages, where each object may have several fields.

The intuition behind the choice of I is that by requiring the support at each point to be the full set, i.e., $\{1,2\}$, we can control what we expect to find in the frame ϕ. For a function such as bp_get1, this means that since $B_1(\mathsf{p}, b)$ is assumed in the precondition, we are sure to find $B_2(\mathsf{p}, b')$ in the frame, for some b'. Showing that the frame is preserved then amounts to showing that the frame also contains $B_2(\mathsf{p}, b')$ after executing the function body – and showing that any other $p' \neq \mathsf{p}$ mentioned in the frame is unaffected, but we will see below that Lemma 1 on pointwise interpretation functions makes that easy.

We now have to show that the implementation of each function matches its specification with this choice of witnesses for the existentials. This is straightforward, because every function in this example has a specification that matches a combination of ENTER1 and Lemma 1. These rules reduce the proof obligations to statements in standard separation logic.

First note that the following *saturation lemma* holds for the interpretation map for bit pairs:

$$I([\_ \mapsto [i \mapsto b] \circ \phi]) \vdash \exists b'.\ \phi = [3{-}i \mapsto b'].$$

We present here the proof of bp_get1. Let $C = (\mathsf{x} := [\mathsf{p}])$, i.e., the body of bp_get1.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall b_2.\ \{\mathsf{p} \mapsto (b + 2 \cdot b_2)\}\ C\ \{\mathsf{p} \mapsto (b + 2 \cdot b_2) \wedge \mathsf{x}\%2 = b\}}\ \text{(trivial)}}{\forall b_2.\ \{I([\mathsf{p} \mapsto [1{\mapsto}b, 2{\mapsto}b_2]])\}\ C\ \{I([\mathsf{p} \mapsto [1{\mapsto}b, 2{\mapsto}b_2]]) \wedge \mathsf{x}\%2 = b\}}\ \text{(definition)}}{\forall \phi.\ \{I([\mathsf{p} \mapsto [1{\mapsto}b] \circ \phi])\}\ C\ \{I([\mathsf{p} \mapsto [1{\mapsto}b] \circ \phi]) \wedge \mathsf{x}\%2 = b\}}\ \text{Saturation}}{I.\ \{\{[\mathsf{p}{\mapsto}[1{\mapsto}b]]\}\}\ C\ \{\{[\mathsf{p}{\mapsto}[1{\mapsto}b]]\} \wedge \mathsf{x}\%2 = b\}}\ \text{Lemma 1c}}{I.\ \{B_1(\mathsf{p}, b)\}\ C\ \{B_1(\mathsf{p}, b) \wedge \mathsf{x}\%2 = b\}}\ \text{(definition)}$$

Notice that the overhead of showing that the abstract specification is met is fairly small and straightforward.

3.3 Example: Monotonic Counter

A monotonic counter is an integer stored in the heap with operations for reading it and incrementing it but not for decrementing it. The implementation could look like this:

$$\begin{array}{l}\mathsf{mc\_new()}\ \{\ \mathsf{c} := \mathsf{alloc}\ 1;\ [\mathsf{c}] := 0;\ \mathsf{return}\ \mathsf{c}\ \}\\ \mathsf{mc\_read(c)}\ \{\ \mathsf{x} := [\mathsf{c}];\ \mathsf{return}\ \mathsf{x}\ \}\\ \mathsf{mc\_inc(c)}\ \{\ \mathsf{x} := [\mathsf{c}];\ [\mathsf{c}] := \mathsf{x}{+}1\ \}\end{array}$$

Reasoning about monotonic counters was posed as a verification challenge by Pilkiewicz and Pottier [19]. They discussed the challenge in a type-and-capability system, so the presentation is somewhat different than for separation logic, but the idea is the same. The counter should have a representation predicate $MC(c, i)$ that can be freely duplicated; i.e., $MC(c, i) \vdash MC(c, i) * MC(c, i)$. It should be possible to frame out one of the copies while the other copy is used to call the increment function; when the first copy is later framed back in, it can soundly be used to call the read function since its postcondition only guarantees that the returned value is *at least* the value from the representation predicate.

The specification in fictional separation logic looks like this:

$$\begin{array}{l}\exists \Sigma : \mathit{sepalg}.\ \exists I : \Sigma \setminus \mathit{heap}.\ \exists MC : \mathit{loc} \times \mathbb{Z} \to \mathcal{P}(\Sigma).\\ \quad (\forall c, j.\ \forall i \le j.\ (MC(c, j) \dashv\vdash MC(c, j) * MC(c, i))) \wedge\\ \quad I.\ \{\mathit{emp}\}\ \mathsf{mc\_new()}\ \{MC(\mathsf{ret}, 0)\} \wedge\\ \quad (\forall i.\ I.\ \{MC(\mathsf{c}, i)\}\ \mathsf{mc\_read(c)}\ \{MC(\mathsf{c}, i) \wedge \mathsf{ret} \ge i\}) \wedge\\ \quad (\forall i.\ I.\ \{MC(\mathsf{c}, i)\}\ \mathsf{mc\_inc(c)}\ \{MC(\mathsf{c}, i + 1)\}).\end{array}$$

The fact about MC has several corollaries that are useful for clients:

$$\begin{array}{rl} MC(c, i) & \dashv\vdash MC(c, i) * MC(c, i)\\ i \le j \wedge MC(c, j) & \vdash MC(c, i) * \top\\ MC(c, i) * MC(c, j) & \vdash MC(c, \mathit{max}(i, j))\end{array}$$

Compared to the solution by Pilkiewicz and Pottier, this solution has several advantages. Our solution can be specified and verified without changing the implementation to account for limitations in the verification system [19, end of Sect. 4]. Moreover, it can be verified in the simple system of fictional separation logic, whereas there exists no soundness proof yet for the very complicated type system used by Pilkiewicz and Pottier.

To verify our specification against the implementation shown above, choose the existentials as follows:

$$\begin{array}{rl} \Sigma = \mathit{loc} \xrightarrow{\text{fin}} \mathbb{Z}_\perp & \text{where composition in } \mathbb{Z} \text{ is } \mathit{max}\\ I(f) = \forall_*\, c \in \mathit{supp}(f).\ \exists j \ge f(c).\ c \mapsto j & \\ MC(c, i) = \{[c \mapsto i]\} & \end{array}$$

The property about MC is straightforward to verify in the assertion logic. Verification of the three functions is shown in the online appendix [14].

$$\frac{S \vdash I * J.\ \{P^{\mathsf{L}}\}\ C\ \{Q^{\mathsf{L}}\}}{S \vdash I.\ \{P\}\ C\ \{Q\}}\ \textsc{CreateL} \qquad \frac{S \vdash I.\ \{P\}\ C\ \{Q\}}{S \vdash I * J.\ \{P^{\mathsf{L}}\}\ C\ \{Q^{\mathsf{L}}\}}\ \textsc{ForgetL}$$

$$\frac{S \vdash I * J.\ \{P^{\mathsf{L}}\}\ C\ \{Q \times \top\}}{S \vdash I * 1.\ \{P^{\mathsf{L}}\}\ C\ \{Q \times \top\}}\ \textsc{LeakL} \qquad \frac{p \text{ pure}}{(p \wedge P) \times Q \dashv\vdash p \wedge (P \times Q)}\ \textsc{Prod-Pure}$$

Fig. 1. Selected inference rules for separating products, using notation $P^{\mathsf{L}} \triangleq P \times \mathit{emp}$

There are some limitations in the specification. There can be no function to deallocate a counter because its representation predicate can be freely shared. The absence of deallocation means that this specification is more suited for a garbage-collected language. Also, the specification does not guarantee that consecutive calls to mc_read will return the same value; it would be valid to implement mc_read such that it has the side effect of incrementing the counter. These limitations are also present in the specification by Pilkiewicz and Pottier.

4 Clients and Separating Products

To allow clients of multiple libraries to know about more than one separation algebra and interpretation function, we introduce *separating products* of interpretations.

Given interpretations $I_1 : \Sigma_1 \setminus\!\!\setminus \Sigma$ and $I_2 : \Sigma_2 \setminus\!\!\setminus \Sigma$, their separating product $I_1 * I_2$ has type $\Sigma_1 \times \Sigma_2 \setminus\!\!\setminus \Sigma$ and is defined as

$$I_1 * I_2 \triangleq \lambda(\sigma_1, \sigma_2).\ I_1(\sigma_1) * I_2(\sigma_2).$$

Figure 1 shows a collection of inference rules about separating products. At the bottom of a proof tree, just above application of Basic, a client should use CreateL for each module that will be used. In that rule, $P^{\mathsf{L}} \triangleq P \times \mathit{emp}$, where $(\times)$ is simply the Cartesian product lifted into *asn*. To write that out, $P_1 \times P_2 \triangleq \lambda s.\ \{(\sigma_1, \sigma_2) \mid \sigma_1 \in P_1(s) \wedge \sigma_2 \in P_2(s)\}$.

The CreateL rule requires the command C to clean up the state abstracted by J completely; i.e., that state must satisfy *emp* in the postcondition. When this is not possible, for example in the Monotonic Counters example, we can instead use the LeakL rule.

Before calling a library function, a client will, as usual, have to frame out irrelevant facts. There, it can be useful to know that $(P * Q)^{\mathsf{L}} \dashv\vdash P^{\mathsf{L}} * Q^{\mathsf{L}}$ and that $P \times Q \dashv\vdash P^{\mathsf{L}} * Q^{\mathsf{R}}$, where $P^{\mathsf{R}} \triangleq \mathit{emp} \times P$. After applying the frame rule, the client can then ignore the irrelevant separation algebras using the ForgetL rule, which is just the CreateL rule inverted.

Pure assertions can move in and out of products as described by the Prod-Pure rule. There are of course rules CreateR, ForgetR and LeakR symmetric to the ones in Figure 1, and further structural rules can be defined to handle commutativity and associativity with separating products.

4.1 Example: Client of Two Modules

Assume we have a client program C that uses both the bit pair and the monotonic counter modules, and we want to show that it has precondition *emp* and postcondition $\top$. We suggest $\top$ in the postcondition because there is no deallocation function for monotonic counters as mentioned earlier.

The standard pattern for this is to assume the module specifications at the bottom of the tree and then move from the standard triple to the appropriate indirect triple by applying Basic once and then Create or Leak for each module, reading from the bottom up. Abbreviate the bit pair and monotonic counter specifications, minus the existentials, as S_{bp} and S_{mc} respectively. The bottom of the proof for C then looks like this.

$$\dfrac{\dfrac{\dfrac{\dfrac{S_{\text{bp}} \wedge S_{\text{mc}} \vdash I_{\text{bp}} * I_{\text{mc}}.\ \{emp \times emp\}\ C\ \{emp \times \top\}}{S_{\text{bp}} \wedge S_{\text{mc}} \vdash I_{\text{bp}} * 1.\ \{emp \times emp\}\ C\ \{emp \times \top\}}\ \textsc{LeakL}}{S_{\text{bp}} \wedge S_{\text{mc}} \vdash 1.\ \{emp\}\ C\ \{\top\}}\ \textsc{CreateR}}{S_{\text{bp}} \wedge S_{\text{mc}} \vdash \{emp\}\ C\ \{\top\}}\ \textsc{Basic}}{(\exists \Sigma, I_{\text{bp}}, B_1, B_2.\ S_{\text{bp}}) \wedge (\exists \Sigma, I_{\text{mc}}, MC.\ S_{\text{mc}}) \vdash \{emp\}\ C\ \{\top\}}\ \exists\text{L}$$

The bottom proof step applies the standard existential-left rule from sequent calculus twice.

If C uses the heap directly, not just through the two modules, it should apply Create once more to get the interpretation $I_{\text{bp}} * I_{\text{mc}} * 1$ on the indirect triple.

Further up in the proof tree, there will eventually be a point where it is necessary to call a function belonging to one of the modules, e.g., the bit pairs. The following pattern is used to ignore irrelevant modules during the call.

$$\dfrac{\dfrac{\dfrac{S \vdash I_{\text{bp}}.\ \{P\}\ \mathsf{call}\ f\ \{Q\}}{S \vdash I_{\text{bp}} * I_{\text{mc}}.\ \{P^{\mathsf{L}}\}\ \mathsf{call}\ f\ \{Q^{\mathsf{L}}\}}\ \textsc{ForgetL}}{S \vdash I_{\text{bp}} * I_{\text{mc}}.\ \{P^{\mathsf{L}} * {R_1}^{\mathsf{L}} * {R_2}^{\mathsf{R}}\}\ \mathsf{call}\ f\ \{Q^{\mathsf{L}} * {R_1}^{\mathsf{L}} * {R_2}^{\mathsf{R}}\}}\ \textsc{Frame}}{S \vdash I_{\text{bp}} * I_{\text{mc}}.\ \{(P * R_1) \times R_2\}\ \mathsf{call}\ f\ \{(Q * R_1) \times R_2\}}\ \textsc{Consequence}$$

Note that this kind of reasoning will not be so explicit in practice; a simple tool can easily elide these steps.

In this section we considered a generic client; see the online appendix [14] for a concrete example client using bit pairs and monotonic counters.

4.2 Example: Wrapper

This example demonstrates how a module can extend the abstraction of another module by using a separating product. We will see that this example gives a compelling argument against solving the fiction-of-disjointness problem by letting the client carry around an explicit but opaque frame as done in [16, Chapter 5].

Consider first the following specification of a collection data structure.

$$
\begin{aligned}
S_{\mathrm{Coll}}(\Sigma : \mathit{sepalg}, I : \Sigma \setminus \mathit{heap}, \mathit{Coll} : \mathit{loc} \times \mathcal{P}_{\mathsf{fin}}(\mathit{val}) \to \mathcal{P}(\Sigma)) \triangleq \forall c, V.\\
&(\mathit{Coll}(c, \_) * \mathit{Coll}(c, \_) \vdash \bot) \wedge\\
&I.\ \{\mathit{emp}\}\ \mathsf{coll\_new}()\ \{\mathit{Coll}(\mathsf{ret}, \emptyset)\} \wedge\\
&I.\ \{\mathit{Coll}(\mathsf{c}, V)\}\ \mathsf{coll\_free}(\mathsf{c})\ \{\mathit{emp}\} \wedge\\
&I.\ \{\mathit{Coll}(\mathsf{c}, V)\}\ \mathsf{coll\_clone}(\mathsf{c})\ \{\mathit{Coll}(\mathsf{c}, V) * \mathit{Coll}(\mathsf{ret}, V)\} \wedge\\
&I.\ \{\mathit{Coll}(\mathsf{c}, V)\}\ \mathsf{coll\_contains}(\mathsf{c}, \mathsf{v})\ \{\mathit{Coll}(\mathsf{c}, V) \wedge \mathsf{ret} = (\mathsf{v} \in V)\} \wedge\\
&I.\ \{\mathit{Coll}(\mathsf{c}, V)\}\ \mathsf{coll\_add}(\mathsf{c}, \mathsf{v})\ \{\mathit{Coll}(\mathsf{c}, V \cup \{\mathsf{v}\})\} \wedge\\
&I.\ \{\mathit{Coll}(\mathsf{c}, V)\}\ \mathsf{coll\_remove}(\mathsf{c}, \mathsf{v})\ \{\mathit{Coll}(\mathsf{c}, V \setminus \{\mathsf{v}\})\}.
\end{aligned}
$$

This is a standard specification of a finite collection, except for the coll_clone function. This function could be implemented by simply copying the contents of the collection to a new data structure; in standard separation logic, that would be the only possible implementation because of the $(*)$ in the postcondition.

In fictional separation logic, it might also be implemented by using *copy on write* – the reference-counting technique in which the contents are initially shared between two collections and only copied when the need arises because one of them is modified [17]. The purpose of including coll_clone here is to have a reason why this library should be specified with fictional separation logic.

Consider now a wrapper module of indirect references to collections. The implementation could be this:

```
wcoll_new(c) { w := alloc 1; [w] := c; return w }
wcoll_contains(w,v) { c := [w]; return coll_contains(c,v) } ...
```

Functions wcoll_add, wcoll_remove and wcoll_free would be implemented analogously to wcoll_contains, forwarding the call. A more useful wrapper module would, of course, add some functionality, such as caching the last query to wcoll_contains or counting the number of calls to wcoll_add, but the essence remains the same.

We can give the following specification to this code.

$$
\begin{aligned}
&\forall \Sigma, I, \mathit{Coll}.\ S_{\mathrm{Coll}}(\Sigma, I, \mathit{Coll}) \Rightarrow\\
&\quad \exists\, \mathit{WColl} : \mathit{loc} \times \mathcal{P}_{\mathsf{fin}}(\mathit{val}) \to \mathcal{P}(\mathit{heap} \times \Sigma).\ \forall V.\\
&\quad\ 1 * I.\ \{\mathit{Coll}(\mathsf{c}, V)^{\mathsf{R}}\}\ \mathsf{wcoll\_new}(\mathsf{c})\ \{\mathit{WColl}(\mathsf{ret}, V)\} \wedge\\
&\quad\ 1 * I.\ \{\mathit{WColl}(\mathsf{w}, V)\}\ \mathsf{wcoll\_contains}(\mathsf{w}, \mathsf{v})\ \{\mathit{WColl}(\mathsf{w}, V) \wedge \mathsf{ret} = (\mathsf{v} \in V)\} \wedge \ldots
\end{aligned}
$$

Observe that this is an example of one specification depending on another, by being universal in the parameters of the S_{Coll} specification from above. (See [1] for more general cases of this design pattern, in standard higher-order separation logic.) For the example implementation above, the proof of the specification should instantiate the existential as $\mathit{WColl}(w, V) = \exists c.\ w \mapsto c \times \mathit{Coll}(c, V)$. As a side remark, this specification could be made more abstract, so that it would

reveal less implementation detail, by hiding the use of the 1-interpretation behind an existential.

The specification of the constructor, wcoll_new, is an example of ownership transfer: ownership of memory described by an abstract predicate ($Coll(\mathsf{c}, V)$) is transferred from the caller to the module. The specification intentionally does not reveal whether the transfer happened simply by storing a pointer, as in our example implementation, or whether the constructor allocated a new collection (or another container data structure), manually copied the contents of the given collection to that, and then freed the given collection.

For comparison, to mimic this in standard separation logic, one could take inspiration from Krishnaswami's design pattern [16, Chapter 5] and let the representation predicate of the collection module describe all the collections that may share data; i.e., $H : \mathcal{P}_{\mathsf{fin}}(loc \times \mathcal{P}_{\mathsf{fin}}(val)) \to \mathcal{P}(heap)$. The constructor specification would then be along the lines of the following, where $\uplus$ denotes union of sets of tuples with disjoint first components, and $Coll'(c, V) \triangleq \{(c, V)\}$.

$$\{H(Coll'(\mathsf{c}, V) \uplus \phi)\}\ \mathsf{wcoll\_new}(\mathsf{c})\ \{\exists\sigma.\ H(\sigma \uplus \phi) * WColl'(\mathsf{ret}, \sigma, V)\}.$$

This specification allows the same implementation freedom as the fictional separation logic version does, and the caller is guaranteed that the abstract frame ϕ is preserved. But it is a completely undesirable specification in practice because the $WColl'$ predicate can never be detached from the H predicate that it may share data with. This means that all the accessor functions must have both $WColl'$ and H in their pre- and postconditions. Even worse, clients need to keep track of the opaque σ that links the two together.

5 Indirect Entailment

There is no restriction that a physical heap can only be in the image of a single abstract σ. Therefore we can sometimes change abstract pre- and postconditions in a more powerful way than what the rule of consequence allows; we present an application of this idea in the next subsection. First, we define *indirect entailment*:

$$P \models_I Q \triangleq \forall\phi.\ \big((\exists\sigma \in P.\ I(\sigma \circ \phi)) \vdash (\exists\sigma \in Q.\ I(\sigma \circ \phi))\big).$$

We can now state the *indirect rule of consequence*:

$$\frac{P \models_I P' \qquad S \vdash I.\ \{P'\}\ C\ \{Q'\} \qquad Q' \models_I Q}{S \vdash I.\ \{P\}\ C\ \{Q\}}\ \text{ROC-Indirect}$$

Its correctness is immediate from the definitions.

The definition of indirect entailment is quite similar to the indirect triple. In fact, if $I : \Sigma \setminus heap$ for some Σ, then $P \models_I Q$ if and only if $\vdash I.\ \{P\}\ skip\ \{Q\}$.

For any I, the relation $(\models_I)$ is reflexive and transitive and is a superrelation of $(\vdash)$. Judgements $P \models_I Q$ can also be studied as a kind of degenerate assertion logic; in that case, the standard natural-deduction introduction and elimination rules for $(\top, \bot, \vee, \exists)$ hold, and so do $(\Rightarrow)$-introduction and $(\wedge, \forall)$-elimination. It

is also possible to reason locally on both sides of a separating conjunction, and the existential-left rule holds; i.e.,

$$\frac{P \models_I P' \qquad Q \models_I Q'}{P * Q \models_I P' * Q'} \qquad\qquad \frac{\forall x.\ (P(x) \models_I Q)}{\exists x.\ P(x) \models_I Q}$$

We discuss more rules for $(\models_I)$ in the online appendix [14].

5.1 Example: Fractional Permissions

Permission accounting [5, 4, 10] is a solution to simple sharing problems where just read-only data is shared. The points-to predicate is generalized to carry a permission, so $l \overset{z}{\mapsto} v$ denotes a z-permission to access heap location l. If z is a read-only permission, then there are no write permissions to l available to others, and therefore its value stays v. If z is a write permission, then there are no other read or write permissions for l available to others.

A write permission can be split across the $*$ into several read-only permissions. If it is known that all read-only permissions have been accounted for, then they can be re-assembled into a write permission. Permissions are clearly useful for sharing data read-only between threads in concurrent programs, but it also has uses in a sequential setting [13, 15].

We will now show how fractional permissions, a particular permission accounting scheme, can be encoded in fictional separation logic. This allows us to use fractional permissions where we need it, without having fractional permissions in the base logic! A permission z is a rational number satisfying $0 < z \leq 1$. The write permission is 1, and all smaller numbers are read-only permissions. We will define the assertion $l \overset{z}{\mapsto} v$ such that the splitting and joining of permissions can be described by the following inference rule.

$$\frac{}{l \overset{z_1}{\mapsto} v_1 * l \overset{z_2}{\mapsto} v_2 \dashv\vdash v_1 = v_2 \wedge z_1 + z_2 \leq 1 \wedge l \overset{z_1+z_2}{\longmapsto} v_1} \textsc{Fractions}$$

We first define the SA of heaps with fractional permissions as usual [4]:

$$\begin{aligned} \Sigma_{\text{fp}} &\triangleq \mathit{Ptr} \overset{\text{fin}}{\rightharpoonup} (\mathit{Val}_= \times \mathit{Perm})_\perp, \text{ where} \\ \mathit{Perm} &\triangleq \{z : \mathbb{Q} \mid 0 < z \leq 1\} \\ z_1 \dot{+} z_2 &\triangleq \begin{cases} z_1 + z_2 & \text{if } z_1 + z_2 \leq 1 \\ \text{undefined} & \text{otherwise} \end{cases} \end{aligned}$$

Since $(\mathit{Perm}, \dot{+})$ is a permission algebra, Σ_{fp} is a separation algebra (see Section 2.4). We define the fractional points-to predicate by $l \overset{z}{\mapsto} v \triangleq \{[l \mapsto (v, z)]\}$ and can then easily verify the Fractions rule.

To make use of all this, we define an interpretation $I_{\text{fp}} : \Sigma_{\text{fp}} \searrow \mathit{heap}$. The idea is the same as for the interpretation function in the bit pair example (Section 3.2): assume we have the full knowledge (permission) for each described heap location.

$$I_{\text{fp}}(f) \triangleq \forall_* l \in \mathit{supp}(f).\ \exists v.\ f(l) = (v, 1) \wedge l \mapsto v.$$

We can now prove a specification of the heap read command for fractional permissions. For clarity, let us just consider the case where the variable name being assigned to is fresh (formally we treat free variables as in [1]):

$$\frac{x \notin fv(e, e')}{\vdash I_{\text{fp}}.\ \{e \overset{z}{\mapsto} e'\}\ x := [e]\ \{e \overset{z}{\mapsto} e' \wedge x = e'\}}$$

Let us sketch the proof of this rule. We first expand the definition of the fractional points-to predicate in the conclusion:

$$\vdash I_{\text{fp}}.\ \{\{[e \mapsto (e', z)]\}\}\ x := [e]\ \{\{[e \mapsto (e', z)]\} \wedge x = e'\}.$$

By applying Lemma 1c, we can reduce this to the proof obligation

$$\vdash \forall \phi.\ \{I_{\text{fp}}([e \mapsto (e', z) \circ \phi])\}\ x := [e]\ \{I_{\text{fp}}([e \mapsto (e', z) \circ \phi]) \wedge x = e'\},$$

which is now a statement in standard separation logic that can be discharged using a saturation lemma like in Section 3.2. Intuitively, I_{fp} in the precondition gives us the points-to predicate needed for applying the standard read rule. The postcondition requires us to prove that I_{fp} holds for the same parameter, which is easy since the heap did not change.

We can also prove the write and allocation rules using the above approach, but we will instead show how to use indirect entailment to get even simpler proofs. The following indirect bi-entailment expresses that having the full permission to a location ($z = 1$) is the same as having a standard points-to predicate for it:

$$\frac{}{(l \overset{1}{\mapsto} v)^{\mathsf{L}} \mathrel{=\!|\!\models}_{I_{\text{fp}} * 1} (l \mapsto v)^{\mathsf{R}}}$$

With this lemma, proved in the online appendix [14], the write and allocation rules follow almost immediately from their standard separation logic versions. For instance, the fractional write rule is derived as follows.

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\vdash 1.\ \{e \mapsto \_\}\ [e] := e'\ \{e \mapsto e'\}}\ \textsc{Basic, Write-std}}{\vdash I_{\text{fp}} * 1.\ \{(e \mapsto \_)^{\mathsf{R}}\}\ [e] := e'\ \{(e \mapsto e')^{\mathsf{R}}\}}\ \textsc{ForgetR}}{\vdash I_{\text{fp}} * 1.\ \{(e \overset{1}{\mapsto} \_)^{\mathsf{L}}\}\ [e] := e'\ \{(e \overset{1}{\mapsto} e')^{\mathsf{L}}\}}\ \text{ROC-}\textsc{Indirect}}{\vdash I_{\text{fp}}.\ \{e \overset{1}{\mapsto} \_\}\ [e] := e'\ \{e \overset{1}{\mapsto} e'\}}\ \textsc{CreateL}$$

6 Stacking

Intuitively, fictional separation logic allows us to pretend we are working in a high-level memory model Σ if we show how to interpret that high-level memory model down to *heap*. It is then natural to investigate whether we can stack an even higher-level memory model Σ' on top of that construction and interpret Σ' down to Σ. Of course, this should generalize to arbitrary levels of stacking.

In this section, we present a theory of stacking that allows this while interacting well with the features introduced in previous sections and not being

a burden on the logic when not in use. It is important to stress that it is in many cases possible for one module to depend on and extend the abstraction of another module *without* using stacking; c.f. the wrapper example in Section 4.2.

The most basic way to combine two interpretations is to compose them as relations. Given interpretations $I : \Sigma_1 \setminus \Sigma_2$ and $J : \Sigma_2 \setminus \Sigma_3$, their relational composition $(I\,;J)$ has type $\Sigma_1 \setminus \Sigma_3$ and is defined as

$$I\,;J \triangleq \lambda\sigma_1.\ \exists\sigma_2' \in I(\sigma_1).\ J(\sigma_2').$$

That is, $\sigma_3 \in (I\,;J)(\sigma_1)$ if and only if $\exists\sigma_2' \in I(\sigma_1).\ \sigma_3 \in J(\sigma_2')$.

We can show the following rule for working with relational composition:

$$\frac{S \vdash \forall\phi.\ J.\ \{\exists\sigma \in P.\ I(\sigma \circ \phi)\}\ C\ \{\exists\sigma \in Q.\ I(\sigma \circ \phi)\}}{S \vdash (I\,;J).\ \{P\}\ C\ \{Q\}}\ \textsc{RelComp}$$

Reading the rule from the bottom up, RelComp allows peeling off the top layer of a multi-layered interpretation, making its frame explicit. This is desirable when verifying an implementation that extends upon the J-interpretation using I. Perhaps J is opaque at the point where this rule is applied.

Relational composition masks the J-interpretation to the outside; in particular, it masks the frame in Σ_2 that goes into J, which means that the converse of RelComp does not hold. In many situations, including the next example, we want to give specifications that expose both the Σ_1-algebra and the Σ_2-algebra in order to be useful to clients that do not have their data exclusively in Σ_1. This discussion motivates our definition of the *stacking* composition. Given interpretations $I : \Sigma_1 \setminus \Sigma_2$ and $J : \Sigma_2 \setminus \Sigma_3$, their stacking $I \succ J$ has type $\Sigma_1 \times \Sigma_2 \setminus \Sigma_3$ and is defined as

$$I \succ J \triangleq (I * 1)\,;J.$$

With this definition, we now get a generalization of RelComp in the form of the following rule, which holds in both directions:

$$\frac{S \vdash \forall\phi.\ J.\ \{\exists\sigma \in P_1.\ I(\sigma \circ \phi) * P_2\}\ C\ \{\exists\sigma \in Q_1.\ I(\sigma \circ \phi) * Q_2\}}{S \vdash I \succ J.\ \{P_1 \times P_2\}\ C\ \{Q_1 \times Q_2\}}\ \textsc{StackComp}$$

The special case of this rule where $P_2 = Q_2 = \mathit{emp}$ is similar to RelComp. The special case where $P_1 = Q_1 = \mathit{emp}$ leads to a rule that is more like a stacking version of Forget and Create (Section 4).

There is a generalization of the Enter1 rule to stacking:

$$\frac{S \vdash \forall\phi.\ J.\ \{I(\sigma \circ \phi) * P\}\ C\ \{I(\sigma' \circ \phi) * Q\}}{S \vdash I \succ J.\ \{\{\sigma\} \times P\}\ C\ \{\{\sigma'\} \times Q\}}\ \textsc{Enter1Stack}$$

This rule is simply a special case of StackComp. Notice that $(I \succ 1) = (I * 1)$, so these inference rules can also be applied to separating products in some cases.

A module may use stacking internally but hide that fact if the stacking does not need to be visible to its clients. This can be achieved by collapsing the stacking composition to a relational composition by the following rule:

$$\frac{S \vdash I \succ J.\ \{P^{\mathsf{L}}\}\ C\ \{Q^{\mathsf{L}}\}}{S \vdash (I\,;J).\ \{P\}\ C\ \{Q\}}\ \textsc{StackRel}$$

6.1 Example: Abstract Fractional Permissions

We saw the example of fractional permissions in Section 5.1, where the points-to predicate was extended to carry a permission. With stacking, we can use essentially the same construction to extend the *Coll* predicate from Section 4.2 to carry a permission. Just like the heap-read command could execute with any partial permission, while heap write required full permission, we can prove that coll_clone and coll_contains can execute with any partial permission, while the other functions require full permission.

Formally, we can prove the validity of the following specification.

$$
\begin{array}{l}
\forall \Sigma, I, \mathit{Coll}.\ S_{\mathrm{Coll}}(\Sigma, I, \mathit{Coll}) \Rightarrow \\
\quad \exists \Sigma' : \mathit{sepalg}.\ \exists I' : \Sigma' \searrow \Sigma. \\
\quad \exists \mathit{FColl} : \mathit{Perm} \times \mathit{loc} \times \mathcal{P}_{\mathsf{fin}}(\mathit{val}) \to \mathcal{P}(\Sigma').\ \forall V, V', c, z, z'. \\
\qquad (\mathit{FColl}^{z}(c, V) * \mathit{FColl}^{z'}(c, V') \dashv\vdash V = V' \wedge z + z' \leq 1 \wedge \mathit{FColl}^{z+z'}(c, V)) \wedge \\
\qquad (\mathit{FColl}^{1}(c, V)^{\mathsf{L}} \mathrel{=\!\!|\!\models_{I' \succ I}} \mathit{Coll}(c, V)^{\mathsf{R}}) \wedge \\
\qquad I' \succ I.\ \{\mathit{FColl}^{z}(\mathsf{c}, V)^{\mathsf{L}}\}\ \mathsf{coll\_contains}(\mathsf{c}, \mathsf{v})\ \{\mathit{FColl}^{z}(\mathsf{c}, V)^{\mathsf{L}} \wedge \mathsf{ret} = (\mathsf{v} \in V)\} \wedge \\
\qquad I' \succ I.\ \{\mathit{FColl}^{z}(\mathsf{c}, V)^{\mathsf{L}}\}\ \mathsf{coll\_clone}(\mathsf{c})\ \{\mathit{FColl}^{z}(\mathsf{c}, V)^{\mathsf{L}} * \mathit{FColl}^{1}(\mathsf{ret}, V)^{\mathsf{L}}\}.
\end{array}
$$

The elements of the specification are all the same as for the standard fractional permissions in Section 5.1. It is written such that the stacking is revealed to clients, allowing them to use the fractional and non-fractional collections together and convert between them using the indirect bi-entailment in the specification. There is thus no need for specifying fractional versions of the remaining functions since the indirect bi-entailment allows reusing the original specifications.

Notice that we can define *FColl* and prove fractional versions of all the functions without knowing their implementation or how *Coll*, I or Σ are defined. In particular, the implementations of coll_clone and coll_contains are allowed to modify the underlying heap, but they still appear read-only through the indirect specification.

If it is not necessary for fractional and non-fractional collections to coexist and share footprints from the perspective of clients, the StackRel rule could be used to hide the stacking in this specification. Then the specification can be made to look as simple as the original specification in Section 4.2 by hiding $(I' ; I)$ behind an existential.

We can verify the specification by choosing the existentials as follows:

$$
\begin{array}{rl}
\Sigma' = \mathit{loc} \overset{\mathsf{fin}}{\Rightarrow} (\mathcal{P}_{\mathsf{fin}}(\mathit{val})_{=} \times \mathit{Perm})_{\perp} \\
I'(f) = \forall_{*}\, c \in \mathit{supp}(f).\ \exists V.\ f(c) = (V, 1) \wedge \mathit{Coll}(c, V) \\
\mathit{FColl}^{z}(c, V) = \{[c \mapsto (V, z)]\}
\end{array}
$$

This is very similar to the original fractional permissions example, and the proofs are also similar.

7 Discussion and Related Work

Simplicity has been a major goal for this theory, particularly in three places: (1) clients of a module that uses fictional separation internally should be able to reason with the same ease as in standard separation logic; (2) the overhead in verifying an implementation with fictional separation should be minimal; and (3) correctness of the meta-theory should be easy to prove. The three goals are listed in order of importance since they represent tasks to be carried out by a decreasing number of people.

We believe that the first simplicity goal has been achieved in most situations, though clients of multiple modules with complex stacking patterns may benefit from tool support for composing the interpretations. The second goal has been achieved in the sense that it is easy to verify examples like those presented in this paper and, moreover, the separation algebras needed can be assembled from standard constructions. The third goal has been reached through judicious choice of definitions, especially by defining the indirect triple in terms of the standard triple – it has been possible to conduct all meta-theoretical proofs without unfolding the definition of the standard triple [14]. Because we work directly in the semantics of the logic, it should be natural to encode this theory in the Coq proof assistant, extending our existing Coq formalization of separation logic [1].

A major inspiration for fictional separation logic has been the design pattern used by Krishnaswami [16, Chapter 5] for specifying data structures with fictional disjointness in standard separation logic. The technique is to define a per-module custom separation logic (not separation *algebra*) and let the client manage the abstract frame, which is explicitly present in every function specification. Fictional separation logic makes the essential part of this design pattern formal, allowing the abstract frame to be managed implicitly by the indirect triple and enabling a general and comprehensive theory on these custom separation logics, instead of scattering the theory across modules on an ad-hoc basis. See also the discussion in Section 4.2. We ignore Krishnaswami's concept of a *ramification operator* since it would make the resulting logic too different from separation logic.

The work on *locality-breaking transformations* (LBT) for context logic, a kind of non-commutative separation logic, by Dinsdale-Young, Gardner and Wheelhouse [8, 9] can also be seen as a formalization of Krishnaswami's design pattern, though they were developed independently. LBT is in the field of program refinement, which means that not only are pre- and postconditions of a triple transformed across abstraction layers, like in fictional separation logic, but the command is also transformed. Despite that difference, the intuition and proof obligations are similar to fictional separation logic: verifying a module implementation involves showing that all atomic operations preserve an abstract frame from a per-module context algebra. Reasoning in LBT is fundamentally in two stages, though: a client program and proof are always created at the high level and are subsequently transformed to the low level outside the logic. In fictional separation logic, moving between the levels is done within the logic itself, and the

separation algebras are first-class entities in the logic. Hence, as we have seen, we can take advantage of all the features in the specification logic, e.g., to hide the definition of a separation algebra behind an existential quantifier. The soundness proofs of the meta-theory in [9] are much more complicated than ours, despite their much less expressive specification logic; it appears to be caused by their proof-theoretic approach to soundness as opposed to our semantic approach.

In terms of what examples can be encoded, fictional separation logic is quite close to the *concurrent abstract predicates* (CAP) framework [7, 11] restricted to sequential programs. CAP has been developed for reasoning about concurrent programs in which several threads may work on the same shared memory; when restricting attention to sequential programs, CAP thus allows to specify and reason about modules that are implemented using sharing. The CAP approach, seen from our perspective, is to fix one particular separation algebra for all modules, which is sufficiently powerful to handle most cases of sharing. The algebra is specialized to each module by giving a per-module protocol definition, with access to the various stages in the protocol controlled by permission accounting. These explicit protocols, describing what atomic modifications may be performed on shared memory regions, give verification tasks a completely different flavour and intuition compared to fictional separation logic. In a sequential setting, the two systems are therefore very different solutions to the same problem. Concurrent abstract predicates is fundamentally rooted in a concurrent setting, though, which complicates the proof system. In particular, program verification requires showing *stability* of all intermediate pre- and postconditions in a proof.

Future work includes extending fictional separation logic to richer programming languages. Our preliminary investigations suggest that it is straightforward to extend the logic to a language with function pointers, by using the idea of nested triples [21] to specify such pointers. In fictional separation logic we will, of course, use *indirect* nested triples. To make it possible to call a function f with a function argument that uses an interpretation stacked on top of f's own interpretation, one can specify both f and its argument through a stacking of interpretations.

We are also interested in extending fictional separation logic to a concurrent language in order to find out whether it can retain its simplicity.

Acknowledgements. We would like to thank Jesper Bengtson, Thomas Dinsdale-Young, Filip Sieczkowski, Kasper Svendsen, Peter Sestoft and Jacob Thamsborg for helpful feedback and discussions.

References

1. Bengtson, J., Jensen, J.B., Sieczkowski, F., Birkedal, L.: Verifying object-oriented programs with higher-order separation logic in Coq. In: Proceedings of ITP (2011)
2. Biering, B., Birkedal, L., Torp-Smith, N.: BI-hyperdoctrines, higher-order separation logic, and abstraction. ACM Transactions on Programming Languages and Systems 29(5) (2007)

3. Birkedal, L., Torp-Smith, N., Yang, H.: Semantics of separation-logic typing and higher-order frame rules for algol-like languages. Logical Methods in Computer Science 2(5:1) (Aug 2006)
4. Bornat, R., Calcagno, C., O'Hearn, P.W., Parkinson, M.J.: Permission accounting in separation logic. In: Proceedings of POPL. pp. 259–270 (2005)
5. Boyland, J.: Checking interference with fractional permissions. In: Proceedings of SAS (2003)
6. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: Proceedings of LICS (2007)
7. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M., Vafeiadis, V.: Concurrent abstract predicates. In: Proceedings of ECOOP (2010)
8. Dinsdale-Young, T., Gardner, P., Wheelhouse, M.: Abstraction and refinement for local reasoning. In: Proceedings of VSTTE (2010)
9. Dinsdale-Young, T., Gardner, P., Wheelhouse, M.: Abstraction and refinement for local reasoning (February 2011), journal submission
10. Dockins, R., Hobor, A., Appel, A.W.: A fresh look at separation algebras and share accounting. In: Proceedings of APLAS (2009)
11. Dodds, M., Jagannathan, S., Parkinson, M.J.: Modular reasoning for deterministic parallelism. In: Proceedings of POPL (2011)
12. Gotsman, A., Berdine, J., Cook, B.: Precision and the conjunction rule in concurrent separation logic. In: Proceedings of MFPS (2011)
13. Haack, C., Hurlin, C.: Resource usage protocols for iterators. In: Proceedings of IWACO (2008)
14. Jensen, J.B., Birkedal, L.: Fictional separation logic: Appendix (2011), included as the next chapter in this thesis
15. Jensen, J.B., Birkedal, L., Sestoft, P.: Modular verification of linked lists with views via separation logic. Journal of Object Technology 10, 2:1–20 (2011)
16. Krishnaswami, N.R.: Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic. Ph.D. thesis, Carnegie Mellon University (2011)
17. Meyers, S.: More Effective C++: 35 New Ways to Improve Your Programs and Designs. Addison-Wesley Professional, first edn. (1996)
18. Parkinson, M.J., Bierman, G.M.: Separation logic and abstraction. In: Proceedings of POPL. pp. 247–258 (2005)
19. Pilkiewicz, A., Pottier, F.: The essence of monotonic state. In: Proceedings of TLDI (2011)
20. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings of LICS. pp. 55–74 (2002)
21. Schwinghammer, J., Birkedal, L., Reus, B., Yang, H.: Nested Hoare triples and frame rules for higher-order store. In: Proceedings of CSL (2009)

Fictional Separation Logic: Appendix

Jonas B. Jensen Lars Birkedal

October 2011
(updated September 2013)

Contents

A-1 Additional Lemmas

$$\frac{}{\{\sigma \circ \sigma'\} \dashv\vdash \{\sigma\} * \{\sigma'\}} \textsc{Homomorphism}$$

The following definition will be used in this appendix for the sake of brevity.

Definition A-1. Given $I : \Sigma \searrow \Sigma'$ and $\phi : \Sigma$, the function $I^\phi : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma')$ is a lifting of I to predicates. It is defined as $I^\phi(P) \triangleq \exists \sigma \in P.\ I(\sigma \circ \phi).\diamond$

The definition above can be used to giver shorter definitions of indirect triple and indirect entailment:

$$I.\ \{P\}\ C\ \{Q\} = \forall \phi.\ \{I^\phi(P)\}\ C\ \{I^\phi(Q)\}$$
$$P \models_I Q = \forall \phi.\ (I^\phi(P) \vdash I^\phi(Q))$$

These definitions are equivalent to the original ones in the main text. Notice that $h \in I^\phi(P)$ iff $\exists \sigma \in P.\ h \in I(\sigma \circ \phi)$.

A-1.1 Specification Logic Rules

Lemma A-1. *If $e : expr$ and $I : \Sigma \searrow \Sigma'$, then $I^\phi(P \wedge e) \dashv\vdash I^\phi(P) \wedge e$.*

Note first how to read the above lemma. We have $e : expr = stack \to val$. When such an expression appears in the place of an assertion in $asn(\Sigma)$, it should be read as being implicitly injected by the function

$$inj_\Sigma(e) \triangleq \lambda s : stack.\ \{\sigma : \Sigma \mid e(s) = \mathsf{true}\}.$$

This is standard in Hoare logics, but we are explicit about it here because the e above appears injected into two different assertion logics: $asn(\Sigma)$ on the left and $asn(\Sigma')$ on the right.

Proof (of Lemma A-1).

$$\begin{array}{ll}
I^\phi(P \wedge e) & \dashv\vdash \\
\exists \sigma \in (P \wedge inj_\Sigma(e)).\ I(\sigma \circ \phi) & \dashv\vdash \\
\exists \sigma.\ \sigma \in P \wedge \sigma \in inj_\Sigma(e) \wedge I(\sigma \circ \phi) & \dashv\vdash \\
\exists \sigma.\ \sigma \in P \wedge e = \mathsf{true} \wedge I(\sigma \circ \phi) & \dashv\vdash \\
(\exists \sigma.\ \sigma \in P \wedge I(\sigma \circ \phi)) \wedge e = \mathsf{true} & \dashv\vdash \\
(\exists \sigma \in P.\ I(\sigma \circ \phi)) \wedge inj_{\Sigma'}(e) & \dashv\vdash \\
I^\phi(P) \wedge e & \square
\end{array}$$

In the following proofs, we omit $(S \vdash)$ on each line.

Proof of control flow commands.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{I.\ \{P \wedge e\}\ C\ \{P\}}{\forall \phi.\ \{I^\phi(P \wedge e)\}\ C\ \{I^\phi(P)\}}\ \text{(definition)}}{\forall \phi.\ \{I^\phi(P) \wedge e\}\ C\ \{I^\phi(P)\}}\ \text{Lemma A-1}}{\forall \phi.\ \{I^\phi(P)\}\ \mathsf{while}\ e\ \mathsf{do}\ C\ \{I^\phi(P) \wedge \neg e\}}\ \textsc{While-std}}{\forall \phi.\ \{I^\phi(P)\}\ \mathsf{while}\ e\ \mathsf{do}\ C\ \{I^\phi(P \wedge \neg e)\}}\ \text{Lemma A-1}}{I.\ \{P\}\ \mathsf{while}\ e\ \mathsf{do}\ C\ \{P \wedge \neg e\}}\ \text{(definition)}$$

Proofs of (if, call and $(;)$) have the same structure.

Proof of assign.

$$\dfrac{\dfrac{\dfrac{}{\forall \phi.\ \{I^\phi(Q)[e/x]\}\ x := e\ \{I^\phi(Q)\}}\ \textsc{Assign-std}}{\forall \phi.\ \{I^\phi(Q[e/x])\}\ x := e\ \{I^\phi(Q)\}}\ \text{(substitution on lifting)}}{I.\ \{Q[e/x]\}\ x := e\ \{Q\}}\ \text{(definition)}$$

Proof of the rule of consequence. This proof works simply by expanding the definition of the indirect triple and appealing to the standard rule of consequence:

$$\dfrac{P \vdash P' \qquad \dfrac{I.\ \{P'\}\ C\ \{Q'\}}{\forall \phi.\ \{\exists \sigma \in P'.\ I(\sigma \circ \phi)\}\ C\ \{\exists \sigma \in Q'.\ I(\sigma \circ \phi)\}} \qquad Q' \vdash Q}{\dfrac{\forall \phi.\ \{\exists \sigma \in P.\ I(\sigma \circ \phi)\}\ C\ \{\exists \sigma \in Q.\ I(\sigma \circ \phi)\}}{I.\ \{P\}\ C\ \{Q\}}}$$

Proof of the existential rule.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall x : A.\ I.\ \{P(x)\}\ C\ \{Q\}}{\forall x : A.\ \forall \phi.\ \{I^\phi(P(x))\}\ C\ \{I^\phi(Q)\}}\ \text{(definition)}}{\forall \phi.\ \forall x : A.\ \{I^\phi(P(x))\}\ C\ \{I^\phi(Q)\}}\ \textsc{Exists-std}}{\forall \phi.\ \{\exists x : A.\ I^\phi(P(x))\}\ C\ \{I^\phi(Q)\}}\ \text{(definition)}}{\forall \phi.\ \{\exists x : A.\ \exists \sigma \in P(x).\ I(\sigma \circ \phi)\}\ C\ \{I^\phi(Q)\}}}{\forall \phi.\ \{\exists \sigma.\ \exists x : A.\ \sigma \in P(x) \wedge I(\sigma \circ \phi)\}\ C\ \{I^\phi(Q)\}}\ \text{(def. of } (\exists) \text{ in } \mathit{asn})}{\forall \phi.\ \{\exists \sigma.\ \sigma \in (\exists x \in A.\ P(x)) \wedge I(\sigma \circ \phi)\}\ C\ \{I^\phi(Q)\}}\ \text{(definition)}}{\forall \phi.\ \{I^\phi(\exists x : A.\ P(x))\}\ C\ \{I^\phi(Q)\}}\ \text{(definition)}}{I.\ \{\exists x : A.\ P(x)\}\ C\ \{Q\}}\ \text{(definition)}$$

This rule is often presented in a more symmetric-looking style, where the conclusion has an existential in both the pre- and postcondition. The symmetric version of the rule can be derived from this one by the rule of consequence.

Proof of the frame rule.

$$\dfrac{\dfrac{mods(C) \mathrel{\#} fv(R)}{\forall \sigma_2.\ mods(C) \mathrel{\#} fv(\sigma_2 \in R)} \qquad \dfrac{\dfrac{I.\ \{P\}\ C\ \{Q\}}{\forall \phi.\ \{I^{\phi}(P)\}\ C\ \{I^{\phi}(Q)\}}\ (\text{definition})}{\forall \phi', \sigma_2.\ \{I^{\sigma_2 \circ \phi'}(P)\}\ C\ \{I^{\sigma_2 \circ \phi'}(Q)\}}\ (\text{specialize})}{\forall \phi', \sigma_2.\ \{\sigma_2 \in R \wedge I^{\sigma_2 \circ \phi'}(P)\}\ C\ \{\sigma_2 \in R \wedge I^{\sigma_2 \circ \phi'}(Q)\}}\ \textsc{Frame-std}$$

$$\dfrac{\forall \phi', \sigma_2.\ \{\sigma_2 \in R \wedge I^{\sigma_2 \circ \phi'}(P)\}\ C\ \{\sigma_2 \in R \wedge I^{\sigma_2 \circ \phi'}(Q)\}}{\forall \phi', \sigma_2.\ \left\{\begin{array}{l}\sigma_2 \in R \wedge \exists \sigma_1 \in P.\\ I(\sigma_1 \circ \sigma_2 \circ \phi')\end{array}\right\}\ C\ \left\{\begin{array}{l}\sigma_2 \in R \wedge \exists \sigma_1 \in Q.\\ I(\sigma_1 \circ \sigma_2 \circ \phi')\end{array}\right\}}\ (\text{definition})$$

$$\dfrac{}{\forall \phi'.\ \left\{\begin{array}{l}\exists \sigma_2 \in R.\ \exists \sigma_1 \in P.\\ I(\sigma_1 \circ \sigma_2 \circ \phi')\end{array}\right\}\ C\ \left\{\begin{array}{l}\exists \sigma_2 \in R.\ \exists \sigma_1 \in Q.\\ I(\sigma_1 \circ \sigma_2 \circ \phi')\end{array}\right\}}\ \textsc{Exists-std}$$

$$\dfrac{}{\forall \phi'.\ \{\exists \sigma \in (P * R).\ I(\sigma \circ \phi')\}\ C\ \{\exists \sigma \in (Q * R).\ I(\sigma \circ \phi')\}}\ (\text{definition})$$

$$\dfrac{}{I.\ \{P * R\}\ C\ \{Q * R\}}\ (\text{definition})$$

Notice that in the place where we appeal to the standard frame rule, we are framing out a *pure* fact conjoined with a non-pure fact. The frame rule applies here because $p \wedge Q \dashv\vdash (p \wedge emp) * Q$ when p is pure.

Proof of Basic.

$$\begin{array}{c} 1.\ \{P\}\ C\ \{Q\} \\ \hline\hline \forall \phi.\ \{\exists \sigma \in P.\ 1(\sigma \circ \phi)\}\ C\ \{\exists \sigma \in Q.\ 1(\sigma \circ \phi)\} \quad (\text{definition}) \\ \hline\hline \forall \phi.\ \{\exists \sigma \in P.\ \{\sigma \circ \phi\}\}\ C\ \{\exists \sigma \in Q.\ \{\sigma \circ \phi\}\} \quad (\text{definition}) \\ \hline\hline \forall \phi.\ \{\exists \sigma \in P.\ \{\sigma\} * \{\phi\}\}\ C\ \{\exists \sigma \in Q.\ \{\sigma\} * \{\phi\}\} \quad \textsc{Homomorphism} \\ \hline\hline \{\exists \sigma \in P.\ \{\sigma\}\}\ C\ \{\exists \sigma \in Q.\ \{\sigma\}\} \quad \star \\ \hline\hline \{P\}\ C\ \{Q\} \end{array}$$

The proof step labelled ($\star$) uses the frame rule in the bottom-to-top direction and specializes ϕ to 0 in the top-to-bottom direction.

A-1.2 Separating Products

$$\begin{array}{c} S \vdash \forall \phi.\ I * 1.\ \{P \times J(\sigma \circ \phi)\}\ C\ \{Q \times J(\sigma' \circ \phi)\} \\ \hline\hline S \vdash I * J.\ \{P \times \{\sigma\}\}\ C\ \{Q \times \{\sigma'\}\} \end{array}\ \textsc{Product1}$$

$$\dfrac{}{(P * Q) \times R \dashv\vdash P^{\mathsf{L}} * (Q {\times} R)} \qquad \dfrac{}{P \times Q \mathrel{=\!\!\mid\!\models_{1_\Sigma * 1_\Sigma}} (P * Q)^{\mathsf{L}}}$$

On assertions, $\times$ distributes over $\forall$ and $\exists$ (inhabited types only). Nothing happens at $\top$, but $\bot {\times} R \dashv\vdash \bot$. Implication is $(P \Rightarrow Q){\times}R \vdash P{\times}R \Rightarrow Q{\times}R$.

A-1.3 Pointwise Interpretations

Proof of Lemma 1.

a. The side condition ensures that $supp(f \circ g) = supp(f) \uplus supp(g)$, so the iterated separating conjunction can be split up in those two parts.

b. The side condition and the separating conjunction together entail that the supports of f and g are disjoint, and then case (a) applies.

c. We will use the fact that for any $f : X \xrightarrow{\text{fin}} \Sigma$, $x : X$ and $\sigma : \Sigma$, we have $[x \mapsto \sigma] \circ f = [x \mapsto \sigma \circ f(x)] \circ f[x \mapsto 0]$, where $f[x \mapsto 0]$ denotes the function that maps x to 0 and any other x' to $f(x')$. Note that both the composition in Σ and in $X \xrightarrow{\text{fin}} \Sigma$ are written as $(\circ)$. We derive the lemma as follows, where $(S \vdash)$ is omitted on each line.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall \phi.\ \{I([x \mapsto \sigma \circ \phi]) \wedge p\}\ C\ \{I([x \mapsto \sigma' \circ \phi]) \wedge q\}}{\forall f.\ \{I([x \mapsto \sigma \circ f(x)]) \wedge p\}\ C\ \{I([x \mapsto \sigma' \circ f(x)]) \wedge q\}}\ \text{(renaming)}}{\forall f.\ \left\{\begin{matrix} I([x \mapsto \sigma \circ f(x)]) * \\ I(f[x \mapsto 0]) \wedge p \end{matrix}\right\} C \left\{\begin{matrix} I([x \mapsto \sigma' \circ f(x)]) * \\ I(f[x \mapsto 0]) \wedge q \end{matrix}\right\}}\ \textsc{Frame}}{\forall f.\ \left\{\begin{matrix} I([x \mapsto \sigma \circ f(x)] \circ \\ f[x \mapsto 0]) \wedge p \end{matrix}\right\} C \left\{\begin{matrix} I([x \mapsto \sigma' \circ f(x)] \circ \\ f[x \mapsto 0]) \wedge q \end{matrix}\right\}}\ \text{Lemma 1a}}{\dfrac{\forall f.\ \{I([x \mapsto \sigma] \circ f) \wedge p\}\ C\ \{I([x \mapsto \sigma'] \circ f) \wedge q\}}{I.\ \{\{[x \mapsto \sigma]\} \wedge p\}\ C\ \{\{[x \mapsto \sigma']\} \wedge q\}}\ \textsc{Enter1}}$$

A-2 Details of Examples

Turnstiles omitted in these proofs. The comments on proof trees assume they are read from the bottom up.

A-2.1 Bit Pair

Saturation lemma: $I([\_ \mapsto [i \mapsto b] \circ a]) \vdash \exists b'.\ a = [3 - i \mapsto b']$.

Proof of bp_new. Let $C = (\mathsf{p} := \mathsf{alloc}\ 1;\ [\mathsf{p}] := 0)$, i.e., the body of bp_new.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall \phi.\ \{emp\}\ C\ \{\mathsf{p} \mapsto 0\}}\ \text{(trivial)}}{\forall \phi.\ \{emp\}\ C\ \{\{1,2\} = \{1,2\} \wedge \mathsf{p} \mapsto (0 + 2 \cdot 0)\}}\ \text{(simplify)}}{\forall \phi.\ \{emp\}\ C\ \{I([\mathsf{p} \mapsto [1 \mapsto \mathit{false}, 2 \mapsto \mathit{false}]])\}}\ \text{(definition)}}{\forall \phi.\ \{I(\phi)\}\ C\ \{I([\mathsf{p} \mapsto [1 \mapsto \mathit{false}, 2 \mapsto \mathit{false}]]) * I(\phi)\}}\ \textsc{Frame}}{\forall \phi.\ \{I(\phi)\}\ C\ \{I([\mathsf{p} \mapsto [1 \mapsto \mathit{false}, 2 \mapsto \mathit{false}]] \circ \phi)\}}\ \text{Lemma 1b}}{I.\ \{\{0\}\}\ C\ \{\{[\mathsf{p} \mapsto [1 \mapsto \mathit{false}, 2 \mapsto \mathit{false}]]\}\}}\ \textsc{Enter1}}{I.\ \{emp\}\ C\ \{B_1(\mathsf{p}, \mathit{false}) * B_2(\mathsf{p}, \mathit{false})\}}\ \text{(simplify)}$$

Proof of bp_free. Let $C = (\mathsf{free\ p})$, i.e., the body of bp_free.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\mathsf{p}\mapsto \_\}\ C\ \{emp\}}\ \text{(trivial)}}{\{\{1,2\}=\{1,2\}\wedge \mathsf{p}\mapsto(b_1+2\cdot b_2)\}\ C\ \{emp\}}\ \text{(simplify)}}{\{I([\mathsf{p}\mapsto[1{\mapsto}b_1,2{\mapsto}b_2]])\}\ C\ \{emp\}}\ \text{(definition)}}{\forall\phi.\ \{I([\mathsf{p}\mapsto[1{\mapsto}b_1,2{\mapsto}b_2]])*I(\phi)\}\ C\ \{I(\phi)\}}\ \textsc{Frame}}{\forall\phi.\ \{I([\mathsf{p}\mapsto[1{\mapsto}b_1,2{\mapsto}b_2]]\circ\phi)\}\ C\ \{I(\phi)\}}\ \text{Lemma 1a}}{I.\ \{\{[\mathsf{p}\mapsto[1{\mapsto}b_1,2{\mapsto}b_2]]\}\}\ C\ \{\{0\}\}}\ \textsc{Enter1}}{I.\ \{B_1(\mathsf{p},b_1)*B_2(\mathsf{p},b_2)\}\ C\ \{emp\}}\ \text{(simplify)}$$

Proof of bp_get1. Let $C = (\mathsf{x} := [\mathsf{p}])$, i.e., the body of bp_get1.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall b_2.\ \{\mathsf{p}\mapsto(b+2\cdot b_2)\}\ C\ \{\mathsf{p}\mapsto(b+2\cdot b_2)\wedge \mathsf{x}\%2=b\}}\ \text{(trivial)}}{\forall b_2.\ \{I([\mathsf{p}\mapsto[1{\mapsto}b,2{\mapsto}b_2]])\}\ C\ \{I([\mathsf{p}\mapsto[1{\mapsto}b,2{\mapsto}b_2]])\wedge \mathsf{x}\%2=b\}}\ \text{(definition)}}{\forall a.\ \{I([\mathsf{p}\mapsto[1{\mapsto}b]\circ a])\}\ C\ \{I([\mathsf{p}\mapsto[1{\mapsto}b]\circ a])\wedge \mathsf{x}\%2=b\}}\ \text{Saturation}}{I.\ \{\{[\mathsf{p}{\mapsto}[1{\mapsto}b]]\}\}\ C\ \{\{[\mathsf{p}{\mapsto}[1{\mapsto}b]]\}\wedge \mathsf{x}\%2=b\}}\ \text{Lemma 1c}}{I.\ \{B_1(\mathsf{p},b)\}\ C\ \{B_1(\mathsf{p},b)\wedge \mathsf{x}\%2=b\}}\ \text{(definition)}$$

Proof of bp_set1. Let $C = (\mathsf{x} := [\mathsf{p}];\ [\mathsf{p}] := \mathsf{b} + \mathsf{x}/2{*}2)$, i.e., the body of bp_set1.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall b_2.\ (\mathsf{p}\mapsto \mathsf{b}+(b'+2\cdot b_2)/2\cdot 2\ \vdash\ \mathsf{p}\mapsto \mathsf{b}+2\cdot b_2)}\ \text{(arithmetic)}}{\forall b_2.\ \{\mathsf{p}\mapsto \_\wedge \mathsf{x}=b'+2\cdot b_2\}\ [\mathsf{p}]:=\mathsf{b}+\mathsf{x}/2{*}2\ \{\mathsf{p}\mapsto \mathsf{b}+2\cdot b_2\}}\ \textsc{Write}}{\forall b_2.\ \{\mathsf{p}\mapsto b'+2\cdot b_2\}\ C\ \{\mathsf{p}\mapsto \mathsf{b}+2\cdot b_2\}}\ \textsc{Seq},\textsc{Read}}{\forall b_2.\ \{I([\mathsf{p}\mapsto[1{\mapsto}b',2{\mapsto}b_2]])\}\ C\ \{I([\mathsf{p}\mapsto[1{\mapsto}\mathsf{b},2{\mapsto}b_2]])\}}\ \text{(definition)}}{\forall a.\ \{I([\mathsf{p}\mapsto[1{\mapsto}b']\circ a])\}\ C\ \{I([\mathsf{p}\mapsto[1{\mapsto}\mathsf{b}]\circ a])\}}\ \text{Saturation}}{I.\ \{\{[\mathsf{p}{\mapsto}[1{\mapsto}b']]\}\}\ C\ \{\{[\mathsf{p}{\mapsto}[1{\mapsto}\mathsf{b}]]\}\}}\ \text{Lemma 1c}}{I.\ \{B_1(\mathsf{p},b')\}\ C\ \{B_1(\mathsf{p},\mathsf{b})\}}\ \text{(definition)}$$

A-2.2 Fractional Permissions

Saturation lemma: $I_{\mathrm{fp}}([\_\mapsto(v,z)\circ a])\vdash a=(v,1-z)$.

Proof of the indirect entailment. Read the following from the bottom up and consider all free variables to be universally quantified on every line.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\exists v'.\ (v,1) = (v',1) \land l \mapsto v' \dashv\vdash l \mapsto v}\ \text{(trivial)}}{I_{\mathrm{fp}}([l \mapsto (v,1)]) \dashv\vdash l \mapsto v}\ \text{(definition)}}{I_{\mathrm{fp}}([l \mapsto (v,1)]) * I_{\mathrm{fp}}(\phi_1) * \{\phi_2\} \dashv\vdash l \mapsto v * \{\phi_2\} * I_{\mathrm{fp}}(\phi_1)}\ \text{(cancellation)}}{I_{\mathrm{fp}}([l \mapsto (v,1)] \circ \phi_1) * \{\phi_2\} \dashv\vdash l \mapsto v * \{\phi_2\} * I_{\mathrm{fp}}(\phi_1)}\ \text{Lemma 1a}}{\exists a \in (l \overset{1}{\mapsto} v).\ I_{\mathrm{fp}}(a \circ \phi_1) * \{\phi_2\} \dashv\vdash \exists h \in (l \mapsto v).\ \{h \circ \phi_2\} * I_{\mathrm{fp}}(\phi_1)}\ \text{(simplify)}}{(l \overset{1}{\mapsto} v)^{\mathsf{L}} \Rrightarrow\!\models_{I_{\mathrm{fp}} * 1} (l \mapsto v)^{\mathsf{R}}}\ \text{(definition)}$$

Proof of read.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x \notin fv(e,e')}}{\{e \mapsto e'\}\ x := [e]\ \{e \mapsto e' \land x = e'\}}\ \textsc{Read-std}}{\left\{\begin{array}{c}\exists v.\ (e',1) = (v,1) \land \\ e \mapsto v\end{array}\right\} x := [e] \left\{\begin{array}{c}\exists v.\ (e',1) = (v,1) \land \\ e \mapsto v \land x = e'\end{array}\right\}}\ \text{(simplify)}}{\{I_{\mathrm{fp}}([e \mapsto (e',1)])\}\ x := [e]\ \{I_{\mathrm{fp}}([e \mapsto (e',1)]) \land x = e'\}}\ \text{(definition)}}{\forall a.\ (\{I_{\mathrm{fp}}([e \mapsto (e',z) \circ a])\}\ x := [e]\ \{I_{\mathrm{fp}}([e \mapsto (e',z) \circ a]) \land x = e'\})}\ \text{Saturation}}{I_{\mathrm{fp}}.\ \{\{[e \mapsto (e',z)]\}\}\ x := [e]\ \{\{[e \mapsto (e',z)]\} \land x = e'\}}\ \text{Lemma 1c}}{I_{\mathrm{fp}}.\ \{e \overset{z}{\mapsto} e'\}\ x := [e]\ \{e \overset{z}{\mapsto} e' \land x = e'\}}\ \text{(definition)}$$

A-2.3 Monotonic Counter

Proof of mc_new**.** Let $C = (\mathsf{c} := \mathsf{alloc}\ 1;\ [\mathsf{c}] := 0)$, i.e., the body of mc_new.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{emp\}\ C\ \{\mathsf{c} \mapsto 0\}}\ \text{(trivial)}}{\{emp\}\ C\ \{\exists j \geq 0.\ \mathsf{c} \mapsto j\}}\ \text{(instantiate)}}{\{emp\}\ C\ \{I([\mathsf{c} \mapsto 0])\}}\ \text{(definition)}}{\forall \phi.\ \{I(\phi)\}\ C\ \{I([\mathsf{c} \mapsto 0]) * I(\phi)\}}\ \textsc{Frame}}{\forall \phi.\ \{I(\phi)\}\ C\ \{I([\mathsf{c} \mapsto 0] \circ \phi)\}}\ \text{Lemma 1b}}{I.\ \{\{0\}\}\ C\ \{\{[\mathsf{c} \mapsto 0]\}\}}\ \textsc{Enter1}}{I.\ \{emp\}\ C\ \{MC(\mathsf{c}, 0)\}}\ \text{(definition)}$$

Proof of mc_read. Let $C = (\mathsf{x} := [\mathsf{c}])$, i.e., the body of mc_read.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall i'.\, \forall j \geq max(i,i').\, (\mathsf{x} = j \vdash \mathsf{x} \geq i)}\,(\text{arithmetic})}{\forall i'.\, \forall j \geq max(i,i').\, \{\mathsf{c} \mapsto j\}\ C\ \{\mathsf{c} \mapsto j \wedge \mathsf{x} \geq i\}}\,\textsc{Read}}{\forall i'.\, \forall j \geq max(i,i').\, \{\mathsf{c} \mapsto j\}\ C\ \{\exists j \geq max(i,i').\, \mathsf{c} \mapsto j \wedge \mathsf{x} \geq i\}}\,(\text{instantiate})}{\forall i'.\, \{\exists j \geq max(i,i').\, \mathsf{c} \mapsto j\}\ C\ \{\exists j \geq max(i,i').\, \mathsf{c} \mapsto j \wedge \mathsf{x} \geq i\}}\,\textsc{Exists}}{I.\, \{\{[\mathsf{c} \mapsto i]\}\}\ C\ \{\{[\mathsf{c} \mapsto i]\} \wedge \mathsf{x} \geq i\}}\,\text{Lemma 1c}}{I.\, \{MC(\mathsf{c}, i)\}\ C\ \{MC(\mathsf{c}, i) \wedge \mathsf{x} \geq i\}}\,(\text{definition})$$

Proof of mc_inc. Let $C = (\mathsf{x} := [\mathsf{c}];\ [\mathsf{c}] := \mathsf{x}{+}1)$, i.e., the body of mc_inc.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall j.\, \{\mathsf{c} \mapsto j\}\ C\ \{\mathsf{c} \mapsto j+1\}}\,(\text{trivial})}{\forall i'.\, \forall j \geq max(i,i').\, \{\mathsf{c} \mapsto j\}\ C\ \{\mathsf{c} \mapsto j+1\}}\,(\text{weakening})}{\forall i'.\, \forall j \geq max(i,i').\, \{\mathsf{c} \mapsto j\}\ C\ \{\exists j \geq max(i+1,i').\, \mathsf{c} \mapsto j\}}\,(\text{instantiate})}{\forall i'.\, \{\exists j \geq max(i,i').\, \mathsf{c} \mapsto j\}\ C\ \{\exists j \geq max(i+1,i').\, \mathsf{c} \mapsto j\}}\,\textsc{Exists}}{I.\, \{\{[\mathsf{c} \mapsto i]\}\}\ C\ \{\{[\mathsf{c} \mapsto i+1]\}\}}\,\text{Lemma 1c}}{I.\, \{MC(\mathsf{c}, i)\}\ C\ \{MC(\mathsf{c}, i+1)\}}\,(\text{definition})$$

where $(\star)$ is a proof of the arithmetic fact that

$$j \geq max(i,i') \vdash j+1 \geq max(i+1,i')$$

A-2.4 Abstract Fractional Permissions

Proof of clone(c)**.**

$$\dfrac{\dfrac{\dfrac{I.\, \{Coll(\mathsf{c}, V)\}\ \ \{Coll(\mathsf{c}, V) * Coll(\mathsf{ret}, V)\}}{I.\, \{I'([\mathsf{c} \mapsto (V,1)])\}\ \ \{I'([\mathsf{c} \mapsto (V,1)]) * Coll(\mathsf{ret}, V)\}}\,(\text{definition})}{\forall \phi.\, I.\, \{I'([\mathsf{c} \mapsto (V,z)] \circ \phi)\}\ \ \{I'([\mathsf{c} \mapsto (V,z)] \circ \phi) * Coll(\mathsf{ret}, V)\}}\,\text{Saturation etc.}}{I' \succ I.\, \{\{[\mathsf{c} \mapsto (V,z)]\}^{\mathsf{L}}\}\ \ \{\{[\mathsf{c} \mapsto (V,z)]\} \times Coll(\mathsf{ret}, V)\}}\,\textsc{Enter1Stack}}{I' \succ I.\, \{FColl^z(\mathsf{c}, V)^{\mathsf{L}}\}\ \ \{FColl^z(\mathsf{c}, V)^{\mathsf{L}} * FColl^1(\mathsf{ret}, V)^{\mathsf{L}}\}}\,\text{ROC-}\textsc{Indirect}$$

A-3 Conjunction Rule and Friends

The conjunction rule is most often written as

$$\dfrac{S \vdash I.\, \{P_1\}\ C\ \{Q_1\} \qquad S \vdash I.\, \{P_2\}\ C\ \{Q_2\}}{S \vdash I.\, \{P_1 \wedge P_2\}\ C\ \{Q_1 \wedge Q_2\}}$$

Despite not being used much in separation logic, the rule has received a lot of attention in the literature, e.g., [O'H07, DYGW11, GBC11], perhaps because the circumstances under which it holds are theoretically interesting. We shall here investigate restrictions on interpretations I that are sufficient for the conjunction rule or related rules to hold in fictional separation logic.

We will work with the following reformulation of the conjunction rule, which is easily shown equivalent to the previous one

$$\frac{S \vdash I.\ \{P\}\ C\ \{Q_1\} \qquad S \vdash I.\ \{P\}\ C\ \{Q_2\}}{S \vdash I.\ \{P\}\ C\ \{Q_1 \wedge Q_2\}}\ \textsc{Conjunction}$$

Recall the definition of I^ϕ (Definition A-1 on page 74).

Lemma A-1. *If, for all ϕ, I^ϕ distributes over conjunctions, then the conjunction rule holds.*

Proof.

$$\dfrac{\dfrac{\dfrac{I.\ \{P\}\ C\ \{Q_1\}}{\forall\phi.\ \{I^\phi(P)\}\ C\ \{I^\phi(Q_1)\}} \quad \dfrac{I.\ \{P\}\ C\ \{Q_2\}}{\forall\phi.\ \{I^\phi(P)\}\ C\ \{I^\phi(Q_2)\}}\ \text{(definition)}}{\forall\phi.\ \{I^\phi(P)\}\ C\ \{I^\phi(Q_1) \wedge I^\phi(Q_2)\}}\ \textsc{Conjunction-std}}{\dfrac{\forall\phi.\ \{I^\phi(P)\}\ C\ \{I^\phi(Q_1 \wedge Q_2)\}}{I.\ \{P\}\ C\ \{Q_1 \wedge Q_2\}}\ \text{(definition)}}\ \text{(assumption)}$$

The Conjunction-std rule applied above is the one for standard triples, which is known to hold. □

Note that $I^\phi(Q_1 \wedge Q_2) \vdash I^\phi(Q_1) \wedge I^\phi(Q_2)$ always holds; whereas the converse direction does not always hold. With the following definitions, we can give necessary and sufficient conditions for when it holds.

Definition A-1. I is *relationally frame-injective* if when $h \in I(\sigma \circ \phi)$ and $h \in I(\sigma' \circ \phi)$ then $\sigma = \sigma'$. ◇

Lemma A-2. *I is relationally frame-injective iff, for all ϕ, I^ϕ distributes over conjunctions.*

Proof. From left to right, assume $h \in I^\phi(P)$ and $h \in I^\phi(Q)$. The goal is to prove $h \in I^\phi(P \wedge Q)$. From our assumptions, we get the existence of $\sigma \in P$ and $\sigma' \in Q$ such that $h \in I(\sigma \circ \phi)$ and $h \in I(\sigma' \circ \phi)$. From the last two facts, relational frame-injectivity implies $\sigma = \sigma'$, so we can use this value as our witness to prove the goal.

From right to left, assume $h \in I(\sigma \circ \phi)$ and $h \in I(\sigma' \circ \phi)$. The goal is to prove $\sigma = \sigma'$. Instantiate the distributivity lemma as $I^\phi(\{\sigma\}) \wedge I^\phi(\{\sigma'\}) \vdash I^\phi(\{\sigma\} \wedge \{\sigma'\})$ and specialize this to h. The right-hand side of that entailment implies that $\sigma = \sigma'$, and the left-hand side is exactly our assumption. □

Definition A-2.

- I is *relationally injective* if when $h \in I(\sigma)$ and $h \in I(\sigma')$ then $\sigma = \sigma'$.
- Σ is *cancellative* if when $\sigma' \doteq \sigma \circ \sigma_1$ and $\sigma' \doteq \sigma \circ \sigma_2$ then $\sigma_1 = \sigma_2$. ◇

Lemma A-3. *For $I : \Sigma \setminus \Sigma'$, if Σ is cancellative and I is relationally injective, then I is relationally frame-injective.*

A-3.1 The Recombination Rule

Among our examples, the conjunction rule holds for bit pair and fractional permissions but not for monotonic counters. We can generalize the rule, however. If for some connective $(\square)$ we have $I^\phi(Q_1) \wedge I^\phi(Q_2) \vdash I^\phi(Q_1 \square Q_2)$, for all ϕ, then the following holds

$$\frac{S \vdash I.\ \{P\}\ C\ \{Q_1\} \qquad S \vdash I.\ \{P\}\ C\ \{Q_2\}}{S \vdash I.\ \{P\}\ C\ \{Q_1 \square Q_2\}}\ \textsc{Recombination}$$

Proof. Analogous to the proof of Lemma A-1. □

Lemma A-4. *For the monotonic counters, the* Recombination *rule holds for* $(\square) = (*)$.

Proof. Recall the algebra and interpretation for monotonic counters:

$$\Sigma = loc \xrightarrow{\text{fin}} \mathbb{Z}_\perp \text{ where composition in } \mathbb{Z} \text{ is } max$$
$$I(f) = \forall_* c \in supp(f).\ \exists j \geq f(c).\ c \mapsto j$$

Notice a fact about I: If $h \in I(\sigma)$ and $h \in I(\sigma')$ then $h \in I(\sigma \circ \sigma')$.

To prove the prerequisite of Recombination, assume $h \in I^\phi(Q_1)$ and $h \in I^\phi(Q_2)$. The goal is to prove $h \in I^\phi(Q_1 * Q_2)$. From the two assumptions, we get the existence of $\sigma_1 \in Q_1$ and $\sigma_2 \in Q_2$ such that $h \in I(\sigma_1 \circ \phi)$ and $h \in I(\sigma_2 \circ \phi)$. To prove our goal, we pick the witness $(\sigma_1 \circ \sigma_2) \in (Q_1 * Q_2)$ and see that $h \in I((\sigma_1 \circ \phi) \circ (\sigma_2 \circ \phi)) = I(\sigma_1 \circ \sigma_2 \circ \phi)$ thanks to the fact about I and idempotence of $(\circ)$. □

A-3.2 Indirect Entailment Rules

Figure A-1 summarizes the usual BI logic rules and whether they are sound for indirect entailment. As we can see, some rules are missing. In particular, there is no $\wedge$-introduction rule. But such a rule is a direct consequence of the conjunction rule when I interprets into *heap*:

Lemma A-5. *If the conjunction rule holds for I or if I^ϕ distributes over conjunctions for all ϕ, then $\wedge$-introduction holds in the logic of $(\models_I)$.*

Proof. To derive it from the conjunction rule, choose $C = skip$ and see that the definition of an indirect triple collapses to that of an indirect entailment. The derivation from distributivity is analogous to the proof of Lemma A-1.□

Interestingly, the converses of this lemma do not hold. Before discussing a counterexample, let us consider an even stronger condition on I: the situation where $(\models_I) = (\vdash)$. This equality of relations is equivalent to having for all P and Q,

$$\frac{P \vdash Q}{P \models_I Q} \qquad \text{and} \qquad \frac{P \models_I Q}{P \vdash Q}$$

$$\frac{P \models_I Q \quad Q \models_I R}{P \models_I R} \qquad \frac{P \vdash Q}{P \models_I Q} \qquad \frac{\vdash I.\ \{P\}\ \mathit{skip}\ \{Q\}}{P \models_I Q}$$

$$\frac{\forall x.\ (P(x) \models_I Q)}{\exists x.\ P(x) \models_I Q} \quad \frac{P \wedge Q \models_I R}{P \models_I Q \Rightarrow R} \quad \frac{P \models_I Q \mathrel{-\!\!*} R}{P * Q \models_I R} \quad \frac{P \models_I P' \qquad Q \models_I Q'}{P * Q \models_I P' * Q'}$$

The following are derivable because standard entailment implies indirect entailment.

$$\frac{}{P \models_I P} \qquad \frac{}{P \models_I \top} \qquad \frac{}{\bot \models_I P} \qquad \frac{P \models_I Q \wedge R}{P \models_I Q} \qquad \frac{P \models_I \forall x.\ Q(x)}{P \models_I Q(y)}$$

$$\frac{}{P \dashv\models_I P * \mathit{emp}} \qquad \frac{}{P * Q \models_I Q * P} \qquad \frac{}{(P * Q) * R \models_I P * (Q * R)}$$

The following are derivable because disjunction is just a special case of $\exists$.

$$\frac{P \models_I Q}{P \models_I Q \vee R} \qquad \frac{P \models_I R \qquad Q \models_I R}{P \vee Q \models_I R}$$

Rules from BI logic that are unsound here:

$$\cancel{\frac{P \models_I Q \qquad P \models_I R}{P \models_I Q \wedge R}} \qquad \cancel{\frac{\forall x.\ (P \models_I Q(x))}{P \models_I \forall x.\ Q(x)}} \qquad \cancel{\frac{P \models_I Q \quad P \models_I Q \Rightarrow R}{P \models_I R}} \qquad \cancel{\frac{P * Q \models_I R}{P \models_I Q \mathrel{-\!\!*} R}}$$

Figure A-1: Inference rules for indirect entailment. Symmetric ones have been elided.

The former of these rules always holds, while the latter appears to be connected with the following notion.

Definition A-3. I is *completable* if for any σ there exists ϕ and h such that $h \in I(\sigma \circ \phi)$. $\diamond$

This is a fairly weak property, satisfied by the bit pair, fractional permissions and monotonic counter examples.

Lemma A-6. $(\models_I) = (\vdash)$ *if and only if* I *is completable and* $\wedge$*-introduction holds for* $(\models_I)$.

Proof. Assume first that the relations are equal. Completability follows from the special case that $\{\sigma\} \models_I \bot$ implies $\{\sigma\} \vdash \bot$. It is trivial that $\wedge$-introduction holds.

For the other direction of the lemma, assume $P \models_I Q$ and note that this is equivalent to $\forall \sigma \in P.\ (\{\sigma\} \models_I Q)$ by the existential rule from Figure A-1. We must show that if $\sigma \in P$ then $\sigma \in Q$. The premise and our assumption allow us to conclude that $\{\sigma\} \models_I Q$. Now instantiate the $\wedge$-introduction rule with the valid premises $\{\sigma\} \models_I \{\sigma\}$ and $\{\sigma\} \models_I Q$ to conclude that $\{\sigma\} \models_I \{\sigma\} \wedge Q$. Specialize this proposition with ϕ and h obtained by completing σ so it says that if $h \in I(\sigma \circ \phi)$ then $h \in I^{\phi}(\{\sigma\} \wedge Q)$. The premise of that is true by the construction of ϕ and h, and the conclusion implies that $\sigma \in Q$. $\square$

Even though $(\models_I) = (\vdash)$ seems like a very strong condition on I, it is not enough to guarantee that the conjunction rule holds, as demonstrated by the following counterexample.

Example A-1. Take the separation algebra $\Sigma = \{1, 2\}_{\emptyset\bot}$. Define $I : \Sigma \searrow$ *heap* by

$$\begin{aligned} I(0) &= emp \\ I(1) &= \{[0 \mapsto 0], [0 \mapsto 1]\} \\ I(2) &= \{[0 \mapsto 0], [0 \mapsto 2]\} \end{aligned}$$

The mapping of 0 serves only to satisfy the side condition on $(\searrow)$ and plays no interesting role here. The other mappings are chosen such that they are different but share a common element.

This common element means that I is not relationally injective. It also makes the conjunction rule fail since we have

$$\begin{aligned} &\vdash I.\ \{\{1\}\}\ [0] := 0\ \{\{1\}\} \text{ and} \\ &\vdash I.\ \{\{1\}\}\ [0] := 0\ \{\{2\}\} \text{ but not} \\ &\vdash I.\ \{\{1\}\}\ [0] := 0\ \{\{1\} \wedge \{2\}\} \end{aligned}$$

since the last postcondition implies false.

But this I *does* satisfy $(\models_I) = (\vdash)$. To show this, we must assume $P \models_I Q$ and prove $P \vdash Q$, i.e., $P \subseteq Q$. There is a finite number of cases to check, and we can reduce their number somewhat. The conclusion is automatically true if $P = \emptyset$ or $Q = \{0, 1, 2\}$ among several other cases. It is only when subset inclusion does not hold that we must show that indirect entailment also fails. Observe that for any I,

$$\frac{P \subseteq P' \qquad P \not\models_I Q \qquad Q' \subseteq Q}{P' \not\models_I Q'}$$

This means that if we can disprove the indirect entailment for Q being $\{0, 1\}$, $\{0, 2\}$ or $\{1, 2\}$, then we have disproved it for all values of Q except the one we are not interested in. Knowing Q to be, for example, $\{0, 1\}$, we only have to check the indirect entailment for $P = \{2\}$ since any $P' \not\subseteq Q$ will a superset of this value. So there are just three cases to prove.

- $\{0\} \not\models_I \{1, 2\}$. If the frame is 0, there is only one choice of heap on the left side, and it cannot be recovered on the right. If the frame is 1 or 2, it will not compose with any of the two choices on the right.
- $\{1\} \not\models_I \{0, 2\}$. The frame must be 0 to compose with 1. The heap is constrained on the left to be $[0 \mapsto 0]$ or $[0 \mapsto 1]$, but the heap $[0 \mapsto 1]$ does not exist in neither $I(0)$ nor $I(2)$.
- Symmetric to the previous case. ◇

A-4 Further Examples

A-4.1 Concrete Client of Two Modules

To show in more detail how a client can work with separating products, we will here discuss a concrete piece of client code. It is the same example as in Section 4.1 but with some code inserted to make it concrete.

Let I_{bp} and I_{mc} be the interpretations functions for bit pairs and monotonic counters respectively. Then consider the following function:

```
f(bp, ctr) {
  b := call bp_get1(bp);
  if b = 0 then
    call mc_inc(ctr);
  call bp_set1(bp, 1)
}
```

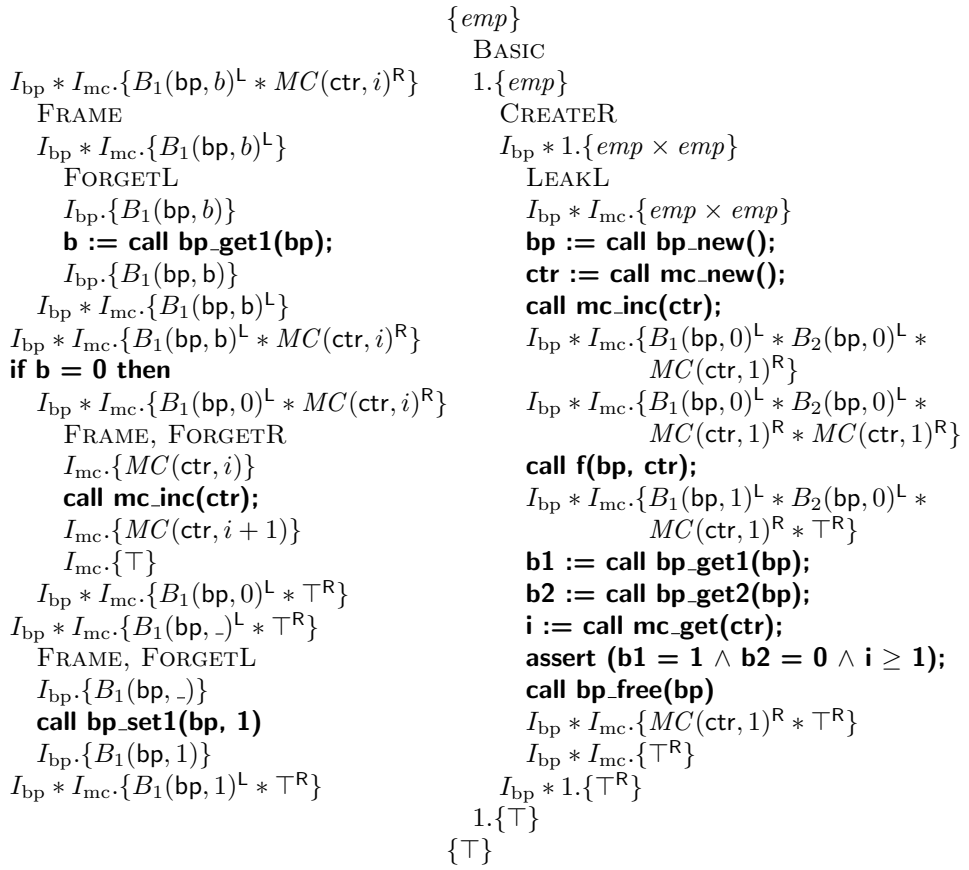

Figure A-2: Proof sketch for client code example

We can specify it as

$$I_{\mathrm{bp}} * I_{\mathrm{mc}}.\ \{B_1(\mathsf{bp}, b)^{\mathsf{L}} * MC(\mathsf{ctr}, i)^{\mathsf{R}}\}\ \mathsf{f}(\mathsf{bp}, \mathsf{ctr})\ \{B_1(\mathsf{bp}, 1)^{\mathsf{L}} * \top^{\mathsf{R}}\},$$

which is a fairly weak specification. Its verification is shown in the first column of Figure A-2.

A caller of this function might look as follows.

```
bp := call bp_new();
ctr := call mc_new();
call mc_inc(ctr);
call f(bp, ctr);
b1 := call bp_get1(bp);
b2 := call bp_get2(bp);
i := call mc_get(ctr);
assert (b1 = 1 and b2 = 0 and i >= 1);
call bp_free(bp)
```

If the above code is called C, then the second column in Figure A-2 shows a verification of $\{emp\}\ C\ \{\top\}$. The steps of using Frame and Forget around each function call as we saw in the left column are now elided just as use of Frame is traditionally left implicit in separation logic proof narrations. The conclusion here is that after the initial set-up of separating products to deal with multiple modules, verification with fictional separation logic is very similar to standard separation logic.

A-4.2 Weak-update type system

Most work on separation logic ignores the type system of the underlying programming language, if it has any. A notable exception is the work of Tan et al. [TSFC09], which presents a dialect of separation logic that mixes types and assertions. We will see in this section how to reconstruct their logic on top of fictional separation logic, saving a lot of effort compared to proving soundness directly with respect to the operational semantics of the language.

Types can be easier to work with than assertions in many cases where only memory safety is to be verified. This comes up, e.g., when using the foreign-function interface of a high-level language.

Recall that programming language values *val* is the disjoint union of integers, Booleans and locations. Assume the injections to *val* from Booleans and integers respectively are named val_{int} and val_{bool}. Define types as follows:

$$\tau : type ::= \mathsf{int} \mid \mathsf{bool} \mid \mathsf{ref}\ \tau.$$

Theorem A-1. *There exists a separation algebra Σ, a predicate $\langle v : \tau \rangle : \mathcal{P}(\Sigma)$, and an interpretation $I : \Sigma \setminus heap$ such that the rules in Figure A-3 are valid.*

Proof. Choose the existentials as follows.

$$\begin{aligned}
\Sigma &= loc \xrightarrow{\text{fin}} type_{=\bot} \\
\langle v : \tau \rangle &= \top * \mathsf{case}\ v, \tau\ \mathsf{of} \\
&\qquad \mid val_{\text{int}}(n), \mathsf{int} \Rightarrow emp \\
&\qquad \mid val_{\text{bool}}(b), \mathsf{bool} \Rightarrow emp \\
&\qquad \mid val_{\text{loc}}(l), \mathsf{ref}\ \tau \Rightarrow \{[l \mapsto \tau]\} \\
&\qquad \mid _, _ \Rightarrow \bot \\
I(\Psi) &= \forall_* l \in supp(\Psi).\ \exists v.\ l \mapsto v \wedge \Psi \in \langle v : \Psi(l) \rangle
\end{aligned}$$

Details of the proofs are in Section A-4.2.1. □

Figure A-3 contains the essential ingredients needed in a weak-update type system. Proving the rules is not trivial, but it is a very minimal theory,

$$\frac{}{\langle v : \mathsf{int}\rangle \dashv\vdash \exists n.\ v = val_{\mathrm{int}}(n)} \qquad \frac{}{\langle v : \mathsf{bool}\rangle \dashv\vdash \exists b.\ v = val_{\mathrm{bool}}(b)}$$

$$\frac{}{\langle e_1 : \mathsf{int}\rangle \wedge \langle e_2 : \mathsf{int}\rangle \vdash \langle e_1 + e_2 : \mathsf{int}\rangle} \quad \cdots \quad \frac{}{\langle e_1 : \mathsf{int}\rangle \wedge \langle e_2 : \mathsf{int}\rangle \vdash \langle e_1 \geq e_2 : \mathsf{bool}\rangle}$$

$$\frac{}{\vdash I.\ \{\langle e : \mathsf{ref}\ \tau\rangle\}\ x := [e]\ \{\langle x : \tau\rangle\}}\ \text{W-Read}$$

$$\frac{}{\vdash I.\ \{\langle e : \mathsf{ref}\ \tau\rangle * \langle e' : \tau\rangle\}\ [e] := e'\ \{\top\}}\ \text{W-Write}$$

$$\frac{}{\langle v : \tau\rangle \times l \mapsto v \models_{I*1} \langle l : \mathsf{ref}\ \tau\rangle \times emp}\ \text{W-s2w}$$

$$\frac{}{\langle v : \tau\rangle \dashv\vdash \langle v : \tau\rangle * \top} \qquad \frac{P : \mathcal{P}(\Sigma)}{P \vdash P * P}\ \text{W-Dupl.}$$

Figure A-3: The essentials of a type system for weak updates

free of distractions. Although it does not yet look like the system of Tan et al., or any other type system, we can derive such a presentation using standard building blocks from fictional separation logic.

We first define stack type environments, which conveniently can be modelled as separation algebras although this is not essential for the theory.

$$\begin{aligned} \Gamma : env &\triangleq var \overset{\text{fin}}{\Rightarrow} type_{=\bot} \\ [\![\Gamma]\!] &\triangleq \top * \forall_* x \in supp(\Gamma).\ \langle x : \Gamma(x)\rangle \\ e :_\Gamma \tau &\triangleq [\![\Gamma]\!] \vdash \langle e : \tau\rangle. \end{aligned}$$

As usual, we implicitly lift $\langle v : \tau\rangle : \mathcal{P}(\Sigma)$ to $\langle e : \tau\rangle : asn(\Sigma)$.

Theorem A-2. *The rules in Figure A-4 are valid.*

Proof. The only non-trivial rule is W-Hom-$\circ$. It is proved in Section A-4.2.2. □

We can now define a triple that combines the type system and standard separation logic:

$$\{\Gamma, P\}\ C\ \{\Gamma', P'\} \triangleq I * 1.\ \{[\![\Gamma]\!] \times P\}\ C\ \{[\![\Gamma']\!] \times P'\}.$$

This hybrid triple satisfies the rules in Figure A-5. The rules are similar but not identical to those of Tan et al. [TSFC09] – we have made a few

$$\frac{e_1 :_\Gamma \mathsf{int} \qquad e_2 :_\Gamma \mathsf{int}}{(e_1 + e_2) :_\Gamma \mathsf{int}} \qquad \cdots \qquad \frac{e_1 :_\Gamma \mathsf{int} \qquad e_2 :_\Gamma \mathsf{int}}{(e_1 \geq e_2) :_\Gamma \mathsf{bool}}$$

$$\frac{}{[\![\Gamma \circ \Gamma']\!] \vdash [\![\Gamma]\!]}\ \text{W-WEAKEN} \qquad \frac{}{[\![[x \mapsto \tau]]\!] \dashv\vdash \langle x : \tau \rangle}\ \text{W-W1}$$

$$\frac{}{[\![0]\!] \dashv\vdash \top}\ \text{W-HOM-0} \qquad \frac{}{[\![\Gamma \circ \Gamma']\!] \dashv\vdash [\![\Gamma]\!] * [\![\Gamma']\!]}\ \text{W-HOM-}\circ$$

Figure A-4: Rules for stack type environments

simplifications. The strong-heap rules are missing some side conditions that seem unnecessary because we do not require all variables to be mentioned in Γ. There is no Ψ-context on the rules because it is not necessary.

Theorem A-3. *The rules in Figure A-5 are valid.*

Proof. The proofs are easy because all the ingredients are already provided by the theory of separating products. The W-*' rules are proved from the rules in Figure A-3 by framing on the stack type environment, then extending them to the separating product with FORGETL. Similarly, the S-*' rules are proved from the standard separation logic rules by applying FORGETR, then framing on a stack environment. The structural and control-flow rules are just special cases of the standard rules we get automatically from the fictional separation logic framework. Details can be found in Section A-4.2.3.□

The purpose of this example was to show how much work can be saved by not building a theory from scratch but assembling most of it from the highly composable theories of separation algebras and separating products. This separates the essential core of the theory, Figure A-3, from the complete system. It also allows us to assemble the theories differently. Instead of combining the type system with a standard separation logic, we could combine it with a separation logic for fractional permissions or any other interpretation from this text.

A-4.2.1 Detailed proofs of Figure A-3.

Proof (of W-DUPL.*).* Follows from idempotence of composition in Σ. □

Lemma A-1. *For any $f : A \overset{\text{fin}}{\Rightarrow} \Sigma$, if $f(a) = \sigma$ then $f = f[a \mapsto 0] \circ [a \mapsto \sigma]$.*

Definition A-1. $I_{\Psi'}(\Psi) \triangleq \forall_* l \in \mathit{supp}(\Psi).\ \exists v.\ l \mapsto v \wedge \Psi' \in \langle v : \Psi(l) \rangle$. ◇

$$\frac{x \notin fv(e, e')}{\vdash \{\Gamma, e \mapsto e'\}\ x := [e]\ \{\Gamma[x{\mapsto}0], e \mapsto e' \wedge x = e'\}}\ \text{S-Read'}$$

$$\frac{}{\vdash \{\Gamma, e \mapsto \_\}\ [e] := e'\ \{\Gamma, e \mapsto e'\}}\ \text{S-Write'}$$

$$\frac{}{\vdash \{\Gamma, emp\}\ x := \mathsf{alloc}\ 1\ \{\Gamma[x{\mapsto}0], x \mapsto \_\}}\ \text{S-Alloc'}$$

$$\frac{}{\vdash \{\Gamma, e \mapsto \_\}\ \mathsf{free}\ e\ \{\Gamma, emp\}}\ \text{S-Free'}$$

$$\frac{}{\vdash \{\Gamma, Q[e/x]\}\ x := e\ \{\Gamma[x{\mapsto}0], Q\}}\ \text{S-Assign'}$$

$$\frac{\mathit{modifies}(C)\ \#\ fv(R) \qquad S \vdash \{\Gamma, P\}\ C\ \{\Gamma, Q\}}{S \vdash \{\Gamma, P * R\}\ C\ \{\Gamma, Q * R\}}\ \text{Frame'}$$

$$\frac{S \vdash \{\Gamma, P \wedge e\}\ C\ \{\Gamma, P\}}{S \vdash \{\Gamma, P\}\ \mathsf{while}\ e\ \mathsf{do}\ C\ \{\Gamma, P \wedge \neg e\}}\ \text{While'}$$

$$\frac{e :_\Gamma \tau}{\{\Gamma, x \mapsto e\}\ \mathit{skip}\ \{\Gamma[x \mapsto \mathsf{ref}\ \tau], emp\}}\ \text{s2w'}$$

$$\frac{e :_\Gamma \mathsf{ref}\ \tau}{\vdash \{\Gamma, emp\}\ x := [e]\ \{\Gamma[x \mapsto \tau], emp\}}\ \text{W-Read'}$$

$$\frac{e :_\Gamma \mathsf{ref}\ \tau \qquad e' :_\Gamma \tau}{\vdash \{\Gamma, emp\}\ [e] := e'\ \{\Gamma, emp\}}\ \text{W-Write'}$$

$$\frac{e :_\Gamma \tau}{\vdash \{\Gamma, emp\}\ x := \mathsf{alloc}\ 1; [x] := e\ \{\Gamma[x \mapsto \mathsf{ref}\ \tau], emp\}}\ \text{W-Alloc'}$$

$$\frac{e :_\Gamma \tau}{\vdash \{\Gamma, emp\}\ x := e\ \{\Gamma[x \mapsto \tau], emp\}}\ \text{W-Assign'}$$

Figure A-5: An adaptation of the inference rules by Tan et al. [TSFC09]. Rules for sequence, conditional, existential and consequence are not shown.

So we have $I(\Psi) \dashv\vdash I_\Psi(\Psi)$, and for any Ψ', the interpretation $I_{\Psi'}$ satisfies the condition for using Lemma 1.

Lemma A-2. *If* $\Psi \in \langle l : \mathsf{ref}\ \tau \rangle$, *then*

$$I(\Psi \circ \phi) \dashv\vdash (\exists v.\ l \mapsto v \wedge \Psi \circ \phi \in \langle v : \tau \rangle) * I_{\Psi\circ\phi}((\Psi \circ \phi)[l \mapsto 0]).$$

Proof. The assumption that $\Psi \in \langle l : \mathsf{ref}\ \tau \rangle$ implies that $(\Psi \circ \phi)(l) = \tau$, so Lemma A-1 is applicable. (We always know that $\Psi \circ \phi$ is defined since this fact is implied by both sides of the bi-entailment.) Now we can prove the lemma:

$$\begin{aligned}
I(\Psi \circ \phi) &\dashv\vdash && \text{(definition)} \\
I_{\Psi\circ\phi}(\Psi \circ \phi) &\dashv\vdash && \text{Lemma A-1} \\
I_{\Psi\circ\phi}([l \mapsto \tau] \circ (\Psi \circ \phi)[l \mapsto 0]) &\dashv\vdash && \text{Lemma 1a} \\
I_{\Psi\circ\phi}([l \mapsto \tau]) * I_{\Psi\circ\phi}((\Psi \circ \phi)[l \mapsto 0]) &\dashv\vdash && \text{(definition)} \\
(\exists v.\ l \mapsto v \wedge \Psi \circ \phi \in \langle v : \tau \rangle) * I_{\Psi\circ\phi}((\Psi \circ \phi)[l \mapsto 0]) & && \square
\end{aligned}$$

Proof (of W-Write*).*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \{e \mapsto \_\}\ [e] := e'\ \{e \mapsto e'\}}\ \textsc{Write-std}}{\vdash \forall \phi, \Psi.\ \left\{\begin{array}{l}\Psi \in (\langle e : \mathsf{ref}\ \tau \rangle * \langle e' : \tau \rangle) \wedge \\ e \mapsto \_ * I_{\Psi\circ\phi}((\Psi\circ\phi)[e \mapsto 0])\end{array}\right\}\ [e] := e'\ \left\{\begin{array}{l}\Psi \in (\langle e : \mathsf{ref}\ \tau \rangle * \langle e' : \tau \rangle) \wedge \\ e \mapsto e' * I_{\Psi\circ\phi}((\Psi\circ\phi)[e \mapsto 0])\end{array}\right\}}\ \textsc{Frame}}{\vdash \forall \phi, \Psi.\ \{\Psi \in (\langle e : \mathsf{ref}\ \tau \rangle * \langle e' : \tau \rangle) \wedge I(\Psi \circ \phi)\}\ [e] := e'\ \{I(\Psi \circ \phi)\}}\ \text{Lemma A-2}}{\vdash \forall \phi. \{\exists \Psi \in (\langle e : \mathsf{ref}\ \tau \rangle * \langle e' : \tau \rangle).\ I(\Psi \circ \phi)\}\ [e] := e'\ \{\exists \Psi.\ I(\Psi \circ \phi)\}}\ \textsc{Exists}}{\vdash I.\ \{\langle e : \mathsf{ref}\ \tau \rangle * \langle e' : \tau \rangle\}\ [e] := e'\ \{\top\}}\ \text{(definition)}$$

$\square$

Proof (of W-Read*).*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \forall \phi, \Psi, v.\ \left\{\begin{array}{l}\Psi \in \langle e : \mathsf{ref}\ \tau \rangle \wedge e \mapsto v \wedge \\ \Phi \circ \phi \in \langle v : \tau \rangle\ * \\ I_{\Psi\circ\phi}((\Psi\circ\phi)[e \mapsto 0])\end{array}\right\}\ x := [e]\ \left\{\begin{array}{l}x = v \wedge \exists l. \\ \Psi \in \langle l : \mathsf{ref}\ \tau \rangle \wedge l \mapsto v \wedge \\ \Phi \circ \phi \in \langle v : \tau \rangle\ * \\ I_{\Psi\circ\phi}((\Psi\circ\phi)[l \mapsto 0])\end{array}\right\}}\ \textsc{Read-std}}{\vdash \forall \phi. \forall \Psi. \{\Psi \in \langle e : \mathsf{ref}\ \tau \rangle \wedge I(\Psi \circ \phi)\}\ x := [e]\ \{\Psi \circ \phi \in \langle x : \tau \rangle \wedge I(\Psi \circ \phi)\}}\ \text{Lemma A-2}}{\vdash \forall \phi. \forall \Psi. \{\Psi \in \langle e : \mathsf{ref}\ \tau \rangle \wedge I(\Psi \circ \phi)\}\ x := [e]\ \{I^\phi \langle x : \tau \rangle\}}\ \text{(instantiate)}}{\vdash \forall \phi. \{\exists \Psi \in \langle e : \mathsf{ref}\ \tau \rangle.\ I(\Psi \circ \phi)\}\ x := [e]\ \{I^\phi \langle x : \tau \rangle\}}\ \textsc{Exists}}{\vdash I.\ \{\langle e : \mathsf{ref}\ \tau \rangle\}\ x := [e]\ \{\langle x : \tau \rangle\}}\ \text{(definition)}$$

Notice that the existential in the postcondition is instantiated not to Ψ but to $\Psi \circ \phi$. $\square$

Proof (of W-S2W*).*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall\phi,\Psi.\ (\Psi[l{\mapsto}\tau] \in \langle v:\tau\rangle \wedge I((\Psi[l{\mapsto}\tau]\circ\phi)[l{\mapsto}0]) * l\mapsto v \vdash I(\Psi[l{\mapsto}\tau]\circ\phi))}\ \text{Lemma A-2}}{\forall\phi,\Psi.\ (\Psi \in \langle v:\tau\rangle \wedge I(\Psi\circ\phi) * l\mapsto v \vdash I^\phi\langle l : \mathsf{ref}\ \tau\rangle)}\ (\star)}{\forall\phi.\ (I^\phi\langle v:\tau\rangle * l\mapsto v \vdash I^\phi\langle l:\mathsf{ref}\ \tau\rangle)}\ \exists\text{L}}{\forall\phi.\ (I^\phi\langle v:\tau\rangle * l\mapsto v \models_1 I^\phi\langle l:\mathsf{ref}\ \tau\rangle)}\ \text{(subrelation)}}{\langle v:\tau\rangle \times l\mapsto v \models_{I*1} \langle l:\mathsf{ref}\ \tau\rangle \times \mathit{emp}}\ \textsc{StackComp}$$

The names of the inference rules applied above are those from the indirect triple since the corresponding ones from indirect entailment don't have a name right now. The application of StackComp is justified because $I*1 = I \succ 1$ as mentioned in Section 6.

In the step labelled $(\star)$, we exploit that the left-hand side of the entailment implies that $l \notin \mathit{supp}(\Psi\circ\phi)$. Therefore, $\Psi\circ\phi = (\Psi[l{\mapsto}\tau]\circ\phi)[l{\mapsto}0]$. Also, $\Psi \in \langle v:\tau\rangle \vdash \Psi[l{\mapsto}\tau] \in \langle v:\tau\rangle$, and we can instantiate the existential on the right of the entailment to $\Psi[l\mapsto\tau]$ since this is sure to compose with ϕ. □

A-4.2.2 Detailed proofs of Figure A-4.

Proof (of expression typings). These are simply reformulations of the rules in Figure A-3. □

Proof (of W-Weaken*).* Corollary of W-Hom-∘ (proved below). □

Proof (of W-W1 *and* W-Hom-0*).* By definition. □

Lemma A-3. $\langle v:\tau\rangle * \langle v:\tau'\rangle \vdash \tau = \tau'$.

Proof. By case analysis on v,τ,τ'. The only non-trivial case is where $v : \mathit{loc}, \tau = \mathsf{ref}\ \tau_1, \tau' = \mathsf{ref}\ \tau_1'$, where we use the fact that $[v\mapsto\tau_1]\circ[v\mapsto\tau_1']$ is only defined when $\tau_1 = \tau_1'$, and therefore $\tau = \tau'$. □

It seems somewhat coincidental that the above lemma holds in our type system, and it should not be relied on too much. It is needed to show W-Hom-∘ from right to left.

Proof (of W-Hom-∘*).* Let us first show the lemma that $[\![[x\mapsto\tau]\circ\Gamma]\!] \dashv\vdash \langle x:\tau\rangle * [\![\Gamma]\!]$. If $x \notin \mathit{supp}(\Gamma)$, then Lemma 1a applies, and the lemma follows by the trivial W-W1. If instead $x \in \mathit{supp}(\Gamma)$, let us first show the entailment

from left to right. The composition being defined implies that $\Gamma(x) = \tau$.

$$
\begin{array}{rl}
[\![[x \mapsto \tau] \circ \Gamma]\!] \dashv\vdash & \text{Lemma A-1} \\
[\![[x \mapsto \tau] \circ [x \mapsto \tau] \circ \Gamma[x \mapsto 0]]\!] \dashv\vdash & \text{(idempotence)} \\
[\![[x \mapsto \tau] \circ \Gamma[x \mapsto 0]]\!] \dashv\vdash & \text{Lemma 1a} \\
[\![[x \mapsto \tau]]\!] * [\![\Gamma[x \mapsto 0]]\!] \dashv\vdash & \text{W-w1} \\
\langle x : \tau \rangle * [\![\Gamma[x \mapsto 0]]\!] \vdash & \text{W-Dupl.} \\
\langle x : \tau \rangle * \langle x : \tau \rangle * [\![\Gamma[x \mapsto 0]]\!] \dashv\vdash & \text{Lemma 1a} \\
\langle x : \tau \rangle * [\![[x \mapsto \tau] \circ \Gamma[x \mapsto 0]]\!] \dashv\vdash & \text{Lemma A-1} \\
\langle x : \tau \rangle * [\![\Gamma]\!]. &
\end{array}
$$

In the other direction, we have to appeal to Lemma A-3.

$$
\begin{array}{rl}
\langle x : \tau \rangle * [\![\Gamma]\!] \dashv\vdash & \text{Lemma A-1} \\
\langle x : \tau \rangle * [\![[x \mapsto \Gamma(x)] \circ \Gamma[x \mapsto 0]]\!] \dashv\vdash & \text{Lemma 1a} \\
\langle x : \tau \rangle * \langle x : \Gamma(x) \rangle * [\![\Gamma[x \mapsto 0]]\!] \vdash & \text{Lemma A-3} \\
\Gamma(x) = \tau \wedge \langle x : \tau \rangle * [\![\Gamma[x \mapsto 0]]\!] \dashv\vdash & \text{Lemma 1a} \\
\Gamma(x) = \tau \wedge [\![[x \mapsto \tau] \circ \Gamma[x \mapsto 0]]\!] \dashv\vdash & \text{(idempotence)} \\
\Gamma(x) = \tau \wedge [\![[x \mapsto \tau] \circ [x \mapsto \tau] \circ \Gamma[x \mapsto 0]]\!] \vdash & \text{Lemma A-1} \\
[\![[x \mapsto \tau] \circ \Gamma]\!]. &
\end{array}
$$

With this lemma done, we can now prove W-Hom-$\circ$ by induction over $|supp(\Gamma)|$. The base case holds by Lemma 1a. In the inductive case, we get the existence of x, τ, Γ_1 such that $\Gamma = [x \mapsto \tau] \circ \Gamma_1$ with $x \notin supp(\Gamma_1)$.

$$
\begin{array}{rl}
[\![[x \mapsto \tau] \circ \Gamma_1 \circ \Gamma']\!] \dashv\vdash & \text{(above lemma)} \\
\langle x : \tau \rangle * [\![\Gamma_1 \circ \Gamma']\!] \dashv\vdash & \text{(induction hyp.)} \\
\langle x : \tau \rangle * [\![\Gamma_1]\!] * [\![\Gamma']\!] \dashv\vdash & \text{Lemma 1a} \\
[\![[x \mapsto \tau] \circ \Gamma_1]\!] * [\![\Gamma']\!]. &
\end{array}
$$

□

A-4.2.3 Detailed proofs of Figure A-5.

Proof (of W-Write'*).*

$$
\dfrac{\dfrac{\dfrac{e :_\Gamma \mathsf{ref}\ \tau \qquad e' :_\Gamma \tau}{[\![\Gamma]\!] \vdash \langle e : \mathsf{ref}\ \tau \rangle \wedge \langle e' : \tau \rangle}}{[\![\Gamma]\!] \vdash \langle e : \mathsf{ref}\ \tau \rangle * \langle e' : \tau \rangle}\ (\star) \qquad \dfrac{\dfrac{}{\vdash I.\ \{\langle e : \mathsf{ref}\ \tau \rangle * \langle e' : \tau \rangle\}\ [e] := e'\ \{\top\}}\ \text{W-Write}}{\vdash I.\ \{\langle e : \mathsf{ref}\ \tau \rangle * \langle e' : \tau \rangle * [\![\Gamma]\!]\}\ [e] := e'\ \{\top * [\![\Gamma]\!]\}}\ \textsc{Frame}}{\dfrac{\vdash I.\ \{[\![\Gamma]\!]\}\ [e] := e'\ \{[\![\Gamma]\!]\}}{\vdash I * 1.\ \{[\![\Gamma]\!] \times emp\}\ [e] := e'\ \{[\![\Gamma]\!] \times emp\}}\ \textsc{ForgetL}}\ \text{ROC}
$$

The step labelled ($\star$) uses W-Dupl. □

Proof (of W-Read'*).*

$$\dfrac{\dfrac{e :_\Gamma \mathsf{ref}\ \tau}{[\![\Gamma]\!] \vdash \langle e : \mathsf{ref}\ \tau\rangle} \quad \dfrac{\dfrac{\dfrac{}{\vdash I.\ \{\langle e : \mathsf{ref}\ \tau\rangle\}\ x := [e]\ \{\langle x : \tau\rangle\}}\ \text{W-Read}}{\vdash I.\ \{\langle e : \mathsf{ref}\ \tau\rangle * [\![\Gamma[x \mapsto 0]]\!]\}\ x := [e]\ \{\langle x : \tau\rangle * [\![\Gamma[x \mapsto 0]]\!]\}}\ \text{Frame}}{\vdash I.\ \{\langle e : \mathsf{ref}\ \tau\rangle * [\![\Gamma]\!]\}\ x := [e]\ \{\langle x : \tau\rangle * [\![\Gamma[x \mapsto 0]]\!]\}}\ \text{W-Weaken}}{\vdash I.\ \{[\![\Gamma]\!]\}\ x := [e]\ \{[\![\Gamma[x \mapsto \tau]]\!]\}}\ \text{ROC}}{\vdash I * 1.\ \{[\![\Gamma]\!] \times emp\}\ x := [e]\ \{[\![\Gamma[x \mapsto \tau]]\!] \times emp\}}\ \text{ForgetL}$$

□

Proof (of s2w'*).*

$$\dfrac{e :_\Gamma \tau \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\langle v : \tau\rangle \times l \mapsto v \models_{I*1} \langle l : \mathsf{ref}\ \tau\rangle \times emp}\ \text{W-s2w}}{\langle e : \tau\rangle \times x \mapsto e \models_{I*1} \langle x : \mathsf{ref}\ \tau\rangle \times emp}\ \text{(add stack)}}{[\![\Gamma[x \mapsto 0]]\!] * \langle e : \tau\rangle \times x \mapsto e \models_{I*1} [\![\Gamma[x \mapsto 0]]\!] * \langle x : \mathsf{ref}\ \tau\rangle \times emp}\ \text{Frame}}{[\![\Gamma]\!] * \langle e : \tau\rangle \times x \mapsto e \models_{I*1} [\![\Gamma[x \mapsto 0]]\!] * \langle x : \mathsf{ref}\ \tau\rangle \times emp}\ \text{W-Weaken}}{[\![\Gamma]\!] \times x \mapsto e \models_{I*1} [\![\Gamma[x \mapsto \mathsf{ref}\ \tau]]\!] \times emp}\ \text{ROC}}{\vdash I * 1.\ \{[\![\Gamma]\!] \times x \mapsto e\}\ \mathit{skip}\ \{[\![\Gamma[x \mapsto \mathsf{ref}\ \tau]]\!] \times emp\}}\ \text{(same definition)}$$

□

Proof (of W-Alloc'*).*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash 1.\ \{emp\}\ x := \mathsf{alloc}\ 1; [x] := e\ \{x \mapsto e\}}\ \text{STD}}{\vdash I * 1.\ \{emp^{\mathsf{R}}\}\ x := \mathsf{alloc}\ 1; [x] := e\ \{(x \mapsto e)^{\mathsf{R}}\}}\ \text{ForgetR}}{\vdash I * 1.\ \{[\![\Gamma]\!] \times emp\}\ x := \mathsf{alloc}\ 1; [x] := e\ \{[\![\Gamma]\!] \times x \mapsto e\}}\ \text{Frame} \quad e :_\Gamma \tau}{\vdash I * 1.\ \{[\![\Gamma]\!] \times emp\}\ x := \mathsf{alloc}\ 1; [x] := e\ \{[\![\Gamma[x \mapsto \mathsf{ref}\ \tau]]\!] \times emp\}}\ \text{s2w'}$$

□

Proof (of W-Assign'*).*

$$\dfrac{\dfrac{e :_\Gamma \tau}{[\![\Gamma]\!] \vdash \langle e : \tau\rangle} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{I.\ \{\langle e : \tau\rangle\}\ x := e\ \{\langle x : \tau\rangle\}}\ \text{Assign}}{I.\ \{\langle e : \tau\rangle * [\![\Gamma[x \mapsto 0]]\!]\}\ x := e\ \{\langle x : \tau\rangle * [\![\Gamma[x \mapsto 0]]\!]\}}\ \text{Frame}}{I.\ \{\langle e : \tau\rangle * [\![\Gamma]\!]\}\ x := e\ \{\langle x : \tau\rangle * [\![\Gamma[x \mapsto 0]]\!]\}}\ \text{W-Weaken}}{I.\ \{[\![\Gamma]\!] * [\![\Gamma]\!]\}\ x := e\ \{\langle x : \tau\rangle * [\![\Gamma[x \mapsto 0]]\!]\}}\ \text{ROC}}{\vdash I.\ \{[\![\Gamma]\!]\}\ x := e\ \{[\![\Gamma[x \mapsto \tau]]\!]\}}\ \text{W-Dupl., Lemma 1a}$$

□

Proof (of S-Write' *and* S-Free'*).*

$$\dfrac{\dfrac{\dfrac{}{1.\ \{e \mapsto \_\}\ \mathsf{free}\ e\ \{emp\}}\ \text{Free}}{I * 1.\ \{emp \times e \mapsto \_\}\ \mathsf{free}\ e\ \{emp \times emp\}}\ \text{ForgetR}}{I * 1.\ \{[\![\Gamma]\!] \times e \mapsto \_\}\ \mathsf{free}\ e\ \{[\![\Gamma]\!] \times emp\}}\ \text{Frame}$$

This proves S-Free'. The proof of S-Write' is similar. □

Proof (of S-Read', S-Alloc' *and* S-Assign'*).*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{1.\ \{Q[e/x]\}\ x := e\ \{Q\}}\ \textsc{Assign}}{I * 1.\ \{emp \times Q[e/x]\}\ x := e\ \{emp \times Q\}}\ \textsc{ForgetR}}{I * 1.\ \{[\![\Gamma[x{\mapsto}0]]\!] \times Q[e/x]\}\ x := e\ \{[\![\Gamma[x{\mapsto}0]]\!] \times Q\}}\ \textsc{Frame}}{I * 1.\ \{[\![\Gamma]\!] \times Q[e/x]\}\ x := e\ \{[\![\Gamma[x{\mapsto}0]]\!] \times Q\}}\ \text{W-}\textsc{Weaken}$$

The other proofs are similar. □

Structural rules and control-flow rules follow directly from their general counterparts in fictional separation logic because of how connectives $(\wedge, \exists, *)$ distribute over the Cartesian product.

A-4.3 Fine-grained Collection

The example of a fine-grained collection is borrowed from the work on Concurrent Abstract Predicates (CAP) [DYDG$^+$10]. We reformulate it in fictional separation logic to allow comparison between this approach and the CAP approach. The two systems seem to allow very similar specifications to be exposed to clients even though the correctness proofs behind them look and feel completely different.

This example is also an opportunity to show how an existing specification can be refined in fictional separation logic. That is probably a good work flow in practice: give a correct and abstract specification of a module in standard separation logic, then show that it is refined by an indirect specification that hides sharing. These two steps can be carried out independently and concurrently as long as they agree on the intermediate specification.

In this case, we will assume a standard specification of a collection module: S_{standard} from Figure A-6. This specification is identical to the one assumed in [DYDG$^+$10], except that we have added creation and disposal functions to make the example complete and realistic.

The fine-grained specification is shown as S_{indirect} in Figure A-6. There is a lot to take in, but the idea is to have a representation predicate per (potential) member of the collection rather than one for the collection as a whole. This recognizes that if clients logically partition the set of program values *val* between them, then they can share access to the collection and ignore each others' interference since they will never query, add or remove the same values. For example, one client could promise to only access *even* numbers in a collection, while another client promised to only access *odd* numbers from the same collection. They would then access the collection with logically disjoint footprints, and the fine-grained specification would allow them to ignore each others' interference.

The intuitive reading of the four representation predicates is as follows. $In(c, v)$ means that value v is in (the collection pointed to by) c. $Out(c, v)$

$$
\begin{aligned}
S_{\text{standard}} \triangleq\ & \exists Coll : loc \times \mathcal{P}_{\mathsf{fin}}(val) \to \mathcal{P}(heap).\ \forall c, V. \\
& (Coll(c, \_) * Coll(c, \_) \vdash \bot) \wedge \\
& \{emp\}\ \mathsf{coll\_new}()\ \{Coll(\mathsf{ret}, \emptyset)\} \wedge \\
& \{Coll(\mathsf{c}, \_)\}\ \mathsf{coll\_free}(\mathsf{c})\ \{emp\} \wedge \\
& \{Coll(\mathsf{c}, V)\}\ \mathsf{coll\_contains}(\mathsf{c}, \mathsf{v})\ \{Coll(\mathsf{c}, V) \wedge \mathsf{ret} = (\mathsf{v} \in V)\} \wedge \\
& \{Coll(\mathsf{c}, V)\}\ \mathsf{coll\_add}(\mathsf{c}, \mathsf{v})\ \{Coll(\mathsf{c}, V \cup \{\mathsf{v}\})\} \wedge \\
& \{Coll(\mathsf{c}, V)\}\ \mathsf{coll\_remove}(\mathsf{c}, \mathsf{v})\ \{Coll(\mathsf{c}, V \setminus \{\mathsf{v}\})\}
\end{aligned}
$$

$$
\begin{aligned}
S_{\text{indirect}} \triangleq\ & \exists \Sigma : sepalg.\ \exists I : \Sigma \searrow heap. \\
& \exists In : loc \times val \to \mathcal{P}(\Sigma). \\
& \exists Outs : loc \times \mathcal{P}(val) \to \mathcal{P}(\Sigma). \\
& \mathsf{let}\ Out(c, v) := Outs(c, \{v\})\ \mathsf{in} \\
& \mathsf{let}\ Own(c, v) := In(c, v) \vee Out(c, v)\ \mathsf{in} \\
& (\forall c, v, V_1, V_2. \\
& \quad (Own(c, v) * Own(c, v) \vdash \bot) \wedge \\
& \quad (V_1 \# V_2 \Rightarrow (Outs(c, V_1 \cup V_2) \dashv\vdash Outs(c, V_1) * Outs(c, V_2))) \\
&) \wedge \\
& I.\ \{emp\}\ \mathsf{coll\_new}()\ \{Outs(\mathsf{ret}, val)\} \wedge \\
& I.\ \{\exists V.\ Outs(\mathsf{c}, val \setminus V) * \forall_* v \in V.\ In(\mathsf{c}, v)\}\ \mathsf{coll\_free}(\mathsf{c})\ \{emp\} \wedge \\
& I.\ \{In(\mathsf{c}, \mathsf{v})\}\ \mathsf{coll\_contains}(\mathsf{c}, \mathsf{v})\ \{In(\mathsf{c}, \mathsf{v}) \wedge \mathsf{ret} = true\} \wedge \\
& I.\ \{Out(\mathsf{c}, \mathsf{v})\}\ \mathsf{coll\_contains}(\mathsf{c}, \mathsf{v})\ \{Out(\mathsf{c}, \mathsf{v}) \wedge \mathsf{ret} = false\} \wedge \\
& I.\ \{Own(\mathsf{c}, \mathsf{v})\}\ \mathsf{coll\_add}(\mathsf{c}, \mathsf{v})\ \{In(\mathsf{c}, \mathsf{v})\} \wedge \\
& I.\ \{Own(\mathsf{c}, \mathsf{v})\}\ \mathsf{coll\_remove}(\mathsf{c}, \mathsf{v})\ \{Out(\mathsf{c}, \mathsf{v})\}
\end{aligned}
$$

Figure A-6: Standard (coarse-grained) and indirect (fine-grained) specifications of a collection module.

means that value v is not in c. $Outs(c, V)$ generalizes this to potentially infinite sets of values. $Own(c, v)$ is the permission to access the state of v being in c without knowing whether it is currently there. In all cases, the asserter of a predicate has exclusive access to the knowledge of whether v is in c.

We can now state the correctness theorem of the fine-grained specification.

Theorem A-4. *With reference to the two specifications defined in Figure A-6, in the context of any program,* $S_{\text{standard}} \vdash S_{\text{indirect}}$.

Proof. Instantiate the existential Σ to a separation algebra where each pointer to a collection maps to a pair: the values that are surely in the collection and the values that are surely not in the collection.

$$\begin{aligned}
\Sigma &= loc \xrightarrow{\text{fin}} \Sigma_{\text{inout}} \text{ where} \\
&\quad \Sigma_{\text{inout}} \triangleq \{(V_\in, V_\notin) : \mathcal{P}_{\text{fin}}(val) \times \mathcal{P}(val) \mid V_\in, V_\notin \text{ disjoint}\} \\
In(c, v) &= \{[c \mapsto (\{v\}, \emptyset)]\} \\
Outs(c, V) &= \{[c \mapsto (\emptyset, V)]\}
\end{aligned}$$

Composition in Σ_{inout} is pairwise disjoint union where possible:

$$(V_\in, V_\notin) \circ (V'_\in, V'_\notin) = \begin{cases} (V_\in \cup V'_\in, V_\notin \cup V'_\notin) & \text{if } V_\in, V_\notin, V'_\in, V'_\notin \text{ disjoint} \\ \text{undefined} & \text{otherwise} \end{cases}$$

The interpretation I is similar to previous examples: it requires all potential values to be accounted for, and then it asserts the *Coll* predicate for the values in the collection

$$I(f) = \forall_* c \in supp(f).\ (\pi_1 f(c) \cup \pi_2 f(c)) = val \wedge Coll(c, \pi_1 f(c))$$

Having chosen the existentials, we proceed to prove each conjunct of S_{indirect}.

Proof of $Own(c, v) * Own(c, v) \vdash \bot$**.** Since *Own* is defined to be a disjunction, we can consider all four possible cases and see that our composition operation is undefined in all of them.

Proof of $Outs(c, V_1 \uplus V_2) \dashv\vdash Outs(c, V_1) * Outs(c, V_2)$**.** Expanding the definition of *Outs*, this is equivalent to

$$\{[c \mapsto (\emptyset, V_1 \uplus V_2)]\} = \{[c \mapsto (\emptyset, V_1)]\} * \{[c \mapsto (\emptyset, V_2)]\}$$

Expanding the definitions of separating conjunction and of composition on finite maps, this holds if $(\emptyset, V_1 \uplus V_2) = (\emptyset, V_1) \circ (\emptyset, V_2)$, which is true by definition of composition in our SA.

Proof of coll_new**.**

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{emp\}\ \mathsf{coll\_new}()\ \{Coll(\mathsf{ret},\emptyset)\}}\ \text{(assumption)}}{\{emp\}\ \mathsf{coll\_new}()\ \{\emptyset \cup val = val \wedge Coll(\mathsf{ret},\emptyset)\}}\ \text{(simplify)}}{\{emp\}\ \mathsf{coll\_new}()\ \{I([\mathsf{ret} \mapsto (\emptyset, val)])\}}\ \text{(definition)}}{\forall \phi.\ \{I(\phi)\}\ \mathsf{coll\_new}()\ \{I([\mathsf{ret} \mapsto (\emptyset, val)]) * I(\phi)\}}\ \textsc{Frame}}{\forall \phi.\ \{I(\phi)\}\ \mathsf{coll\_new}()\ \{I([\mathsf{ret} \mapsto (\emptyset, val)]) \circ \phi)\}}\ \text{Lemma 1b}}{I.\ \{\{0\}\}\ \mathsf{coll\_new}()\ \{\{[\mathsf{ret} \mapsto (\emptyset, val)]\}\}}\ \textsc{Enter1}}{I.\ \{emp\}\ \mathsf{coll\_new}()\ \{Outs(\mathsf{ret}, val)\}}\ \text{(definition)}$$

Proof of coll_free**.**

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall \phi.\ \{Coll(\mathsf{c}, V)\}\ \mathsf{coll\_free}(\mathsf{c})\ \{emp\}}\ \text{(assumption)}}{\forall \phi.\ \{V \cup (val \setminus V) = val \wedge Coll(\mathsf{c}, V)\}\ \mathsf{coll\_free}(\mathsf{c})\ \{emp\}}\ \text{(simplify)}}{\forall \phi.\ \{I([\mathsf{c} \mapsto (V, val \setminus V)])\}\ \mathsf{coll\_free}(\mathsf{c})\ \{emp\}}\ \text{(definition)}}{\forall \phi.\ \{I([\mathsf{c} \mapsto (V, val \setminus V)]) * I(\phi)\}\ \mathsf{coll\_free}(\mathsf{c})\ \{I(\phi)\}}\ \textsc{Frame}}{\forall \phi.\ \{I([\mathsf{c} \mapsto (V, val \setminus V)] \circ \phi)\}\ \mathsf{coll\_free}(\mathsf{c})\ \{I(\phi)\}}\ \text{Lemma 1a}}{I.\ \{\{[\mathsf{c} \mapsto (V, val \setminus V])\}\}\ \mathsf{coll\_free}(\mathsf{c})\ \{emp\}}\ \textsc{Enter1}}{I.\ \{\{[\mathsf{c} \mapsto (\emptyset, val \setminus V])\} * \{[\mathsf{c} \mapsto (V, \emptyset)]\}\}\ \mathsf{coll\_free}(\mathsf{c})\ \{emp\}}$$
$$I.\ \{Outs(\mathsf{c}, val \setminus V) * \forall_* v \in V.\ In(\mathsf{c}, v)\}\ \mathsf{coll\_free}(\mathsf{c})\ \{emp\}$$

At the application of Lemma 1a, its side condition of disjoint supports is satisfied since Σ was defined such that only the unit value composes with $(V, val \setminus V)$, so the pointer c cannot be in the support of the frame ϕ at that point.

Proof of coll_contains**, case** *In*. In this proof, we abbreviate coll_contains as cc.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall V_\in.\ \{Coll(\mathsf{c}, \{\mathsf{v}\} \uplus V_\in)\}\ \mathsf{cc}(\mathsf{c}, \mathsf{v})\ \{Coll(\mathsf{c}, \{\mathsf{v}\} \uplus V_\in) \wedge \mathsf{ret} = true\}}\ \text{(assumption)}}{\begin{array}{l}\forall V_\in, V_\notin.\ \{\mathsf{v}\} \uplus V_\in \uplus V_\notin = val \Rightarrow \\ \{Coll(\mathsf{c}, \{\mathsf{v}\} \uplus V_\in)\}\ \mathsf{cc}(\mathsf{c}, \mathsf{v})\ \{Coll(\mathsf{c}, \{\mathsf{v}\} \uplus V_\in) \wedge \mathsf{ret} = true\}\end{array}}\ \text{(weaken)}}{\begin{array}{l}\forall V_\in, V_\notin.\ \{\mathsf{v}\}, V_\in, V_\notin \text{ disjoint} \Rightarrow \{I([\mathsf{c} \mapsto (\{\mathsf{v}\} \uplus V_\in, V_\notin)])\} \\ \mathsf{cc}(\mathsf{c}, \mathsf{v})\ \{I([\mathsf{c} \mapsto (\{\mathsf{v}\} \uplus V_\in, V_\notin)]) \wedge \mathsf{ret} = true\}\end{array}}\ \text{(simplify)}}{I.\ \{[\mathsf{c} \mapsto (\{\mathsf{v}\}, \emptyset)]\}\ \mathsf{cc}(\mathsf{c}, \mathsf{v})\ \{[\mathsf{c} \mapsto (\{\mathsf{v}\}, \emptyset)] \wedge \mathsf{ret} = true\}}\ \text{Lemma 1c}}{I.\ \{In(\mathsf{c}, \mathsf{v})\}\ \mathsf{cc}(\mathsf{c}, \mathsf{v})\ \{In(\mathsf{c}, \mathsf{v}) \wedge \mathsf{ret} = true\}}\ \text{(definition)}$$

Proof of coll_contains**, case** *Out*. In this proof, we abbreviate coll_contains as cc.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall V_\in.\ \{\mathsf{v}\} \notin V_\in \Rightarrow \{\mathit{Coll}(\mathsf{c}, V_\in)\}\ \mathsf{cc}(\mathsf{c},\mathsf{v})\ \{\mathit{Coll}(\mathsf{c}, V_\in) \wedge \mathsf{ret} = \mathit{false}\}}\ \text{(assumption)}}{\forall V_\in, V_\notin.\ \{\mathsf{v}\} \uplus V_\in \uplus V_\notin = \mathit{val} \Rightarrow \{\mathit{Coll}(\mathsf{c}, V_\in)\}\ \mathsf{cc}(\mathsf{c},\mathsf{v})\ \{\mathit{Coll}(\mathsf{c}, V_\in) \wedge \mathsf{ret} = \mathit{false}\}}\ \text{(weaken)}}{\forall V_\in, V_\notin.\ \{\mathsf{v}\}, V_\in, V_\notin \text{ disjoint} \Rightarrow \{I([\mathsf{c} \mapsto (V_\in, \{\mathsf{v}\} \uplus V_\notin)])\}\ \mathsf{cc}(\mathsf{c},\mathsf{v})\ \{I([\mathsf{c} \mapsto (V_\in, \{\mathsf{v}\} \uplus V_\notin)]) \wedge \mathsf{ret} = \mathit{false}\}}\ \text{(simplify)}}{I.\ \{[\mathsf{c} \mapsto (\emptyset, \{\mathsf{v}\})]\}\ \mathsf{cc}(\mathsf{c},\mathsf{v})\ \{[\mathsf{c} \mapsto (\emptyset, \{\mathsf{v}\})] \wedge \mathsf{ret} = \mathit{false}\}}\ \text{Lemma 1c}}{I.\ \{\mathit{Out}(\mathsf{c},\mathsf{v})\}\ \mathsf{cc}(\mathsf{c},\mathsf{v})\ \{\mathit{Out}(\mathsf{c},\mathsf{v}) \wedge \mathsf{ret} = \mathit{false}\}}\ \text{(definition)}$$

The add and remove functions are similar, also using Lemma 1c. □

A-4.3.1 Comparison to Concurrent Abstract Predicates.

The only essential differences from the CAP specification [DYDG$^+$10] is our *Outs* predicate, which is needed to specify creation and disposal. We have specified *Out* in terms of *Outs*. It might be tempting to do it the other way around and let *Out* be the primitive predicate and define $\mathit{Outs}(c, V)$ as $\forall_* v \in V.\ \mathit{Out}(c, v)$, but this is impossible since iterated separating conjunction is only defined on finite domains. Defining it infinitely would require an infinite composition operator on the underlying separation algebra.

A-4.4 Permission Scaling

We can easily add a convenient feature to the assertion logic of fractional permissions from Section 5.1: *permission scaling*. The idea is to have an assertion $z \cdot P$ that loosely speaking multiplies every permission in P by z. Assuming we have an operation $z \cdot h$ that multiplies every permission in the fractional heap h by z, define

$$z \cdot P \triangleq \{z \cdot h \mid h \in P\}$$

A crucial difference between this definition and the intuition mentioned is that we have

$$\frac{}{z \cdot (l_1 \overset{1}{\mapsto} v_1 * l_2 \overset{1}{\mapsto} v_2) \vdash l_1 \neq l_2}$$

for any z, while a scaling on the individual points-to predicates only admits the weaker and much stranger rule

$$\frac{z > \frac{1}{2}}{l_1 \overset{z}{\mapsto} v_1 * l_2 \overset{z}{\mapsto} v_2 \vdash l_1 \neq l_2}$$

This leads to the problem noticed by Bornat et al. [BCOP05] that a tree predicate in which every points-to assertion is multiplied by some z actually describes a DAG if $z \leq \frac{1}{2}$.

With permission scaling, one just defines an ordinary tree predicate $\mathit{tree}(t, \tau)$, and a scaled tree is then $z \cdot \mathit{tree}(t, \tau)$. The latter is a tree, not a DAG, and it is unnecessary for the tree library to export lemmas about how the predicate can be fractionally split and joined since these follow from the general theory of permission scaling. Note that one does not get the tree property from Boyland's definition of permission scaling [Boy07] because he allows fractions greater than 1 in intermediate heaps.

The following assertion logic inference rules are valid.

$$\frac{p \text{ pure}}{z{\cdot}p \dashv\vdash p} \qquad \frac{}{l \overset{z \cdot z'}{\longmapsto} v \dashv\vdash z \cdot (l \overset{z'}{\mapsto} v)} \qquad \frac{}{z \cdot (P * Q) \vdash z{\cdot}P * z{\cdot}Q}$$

$$\frac{}{(z_1 \dot{+} z_2) \cdot P \vdash z_1{\cdot}P * z_2{\cdot}P} \qquad \frac{P \text{ precise}}{z_1{\cdot}P * z_2{\cdot}P \vdash (z_1 \dot{+} z_2) \cdot P}$$

$$\frac{P \vdash Q}{z{\cdot}P \vdash z{\cdot}Q} \qquad \frac{}{z \cdot \exists x.\ P(x) \dashv\vdash \exists x.\ z \cdot P(x)}$$

The reason for introducing permission scaling here is to show that it can be added to this logic at no cost. It is reasonable to require this feature to be supported when encoding fractional permissions in other systems, such as Concurrent Abstract Predicates [DYDG$^+$10].

A-4.5 Better Monotonic Counters

The monotonic counters example in Section 3.3 was designed to address the verification challenge posed in [PP11]. With a few changes, we can make it even better. First, we can strengthen the specification of mc_read so it becomes

$$\forall i.\ I.\ \{MC(\mathsf{c}, i)\}\ \mathsf{mc\_read}(\mathsf{c})\ \{MC(\mathsf{c}, \mathsf{ret}) \wedge \mathsf{ret} \geq i\}.$$

The postcondition previously had $MC(\mathsf{c}, i)$ instead of $MC(\mathsf{c}, \mathsf{ret})$. Second, we can remove the awkward $\top$ from the weakening corollary and instead show

$$i \leq j \wedge MC(c, j) \vdash MC(c, i).$$

To make this work, we only have to change the definition of MC, leaving the other existentials as they were:

$$\begin{aligned} \Sigma &= \mathit{loc} \overset{\text{fin}}{\Rightarrow} \mathbb{Z}_\perp \text{ where composition in } \mathbb{Z} \text{ is } \mathit{max} \\ I(f) &= \forall_* c \in \mathit{supp}(f).\ \exists k \geq f(c).\ c \mapsto k \\ MC(c, i) &= \exists j \geq i.\ \{[c \mapsto j]\} \end{aligned}$$

We then have to reverify all functions with the new definition of MC. The interesting case is mc_read.

Proof of mc_read. Let $C = (\mathsf{x} := [\mathsf{c}])$, i.e., the body of mc_read.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall k.\ \{\mathsf{c} \mapsto k\}\ \mathsf{x} := [\mathsf{c}]\ \{\mathsf{c} \mapsto k \wedge \mathsf{x} = k\}}\ \textsc{Read}}{\forall j \geq i.\ \forall \phi.\ \forall k \geq j \circ \phi.\ \{\mathsf{c} \mapsto k\}\ C\ \{\mathsf{c} \mapsto k \wedge \mathsf{x} = k\}}\ (\star)}{\forall j \geq i.\ \forall \phi.\ \forall k \geq j \circ \phi.\ \{\mathsf{c} \mapsto k\}\ C\ \{\mathsf{x} \geq \mathsf{x} \circ \phi \wedge \mathsf{c} \mapsto \mathsf{x} \wedge \mathsf{x} \geq i\}}\ \textsc{Consequence}}{\forall j \geq i.\ \forall \phi.\ \forall k \geq j \circ \phi.\ \{\mathsf{c} \mapsto k\}\ C\ \{\exists k \geq \mathsf{x} \circ \phi.\ \mathsf{c} \mapsto k \wedge \mathsf{x} \geq i\}}\ \text{(instantiate)}}{\forall j \geq i.\ \forall \phi.\ \{\exists k \geq j \circ \phi.\ \mathsf{c} \mapsto k\}\ C\ \{\exists k \geq \mathsf{x} \circ \phi.\ \mathsf{c} \mapsto k \wedge \mathsf{x} \geq i\}}\ \textsc{Exists}}{\forall j \geq i.\ \forall \phi.\ \{I([\mathsf{c} \mapsto j \circ \phi])\}\ C\ \{I([\mathsf{c} \mapsto \mathsf{x} \circ \phi]) \wedge \mathsf{x} \geq i\}}\ \text{(definition)}}{\forall j \geq i.\ I.\ \{\{[\mathsf{c} \mapsto j]\}\}\ C\ \{\{[\mathsf{c} \mapsto \mathsf{x}]\} \wedge \mathsf{x} \geq i\}}\ \text{Lemma 1c}}{\forall j \geq i.\ I.\ \{\{[\mathsf{c} \mapsto j]\}\}\ C\ \{\exists j \geq \mathsf{x}.\ \{[\mathsf{c} \mapsto j]\} \wedge \mathsf{x} \geq i\}}\ \text{(instantiate)}}{I.\ \{\exists j \geq i.\ \{[\mathsf{c} \mapsto j]\}\}\ C\ \{\exists j \geq \mathsf{x}.\ \{[\mathsf{c} \mapsto j]\} \wedge \mathsf{x} \geq i\}}\ \textsc{Exists}}{I.\ \{MC(\mathsf{c}, i)\}\ C\ \{MC(\mathsf{c}, \mathsf{x}) \wedge \mathsf{x} \geq i\}}\ \text{(definition)}$$

The proof tree labelled $(\star)$ above should be a proof of the arithmetic fact that if $j \geq i$ and $k \geq j \circ \phi$, then $\mathsf{c} \mapsto k \wedge \mathsf{x} = k \vdash \mathsf{x} \geq \mathsf{x} \circ \phi \wedge \mathsf{c} \mapsto \mathsf{x} \wedge \mathsf{x} \geq i$.

Note that while the above i, j, k belong to $\mathbb{Z}$, the frame ϕ belongs to $\mathbb{Z}_\perp$; i.e., the integers extended with a unit element. Since we made $\mathbb{Z}$ a permission algebra with composition *max*, composition in $\mathbb{Z}_\perp$ behaves like *max* lifted to also being defined for negative infinity, i.e., the unit of $\mathbb{Z}_\perp$. Therefore, the composition $(\circ)$ above is *max* lifted to treat the unit as negative infinity.

Proof of mc_new. Let $C = (\mathsf{c} := \mathsf{alloc}\ 1;\ [\mathsf{c}] := 0)$, i.e., the body of mc_new.

$$\dfrac{\dfrac{\dfrac{\vdots}{I.\ \{emp\}\ C\ \{\{[\mathsf{c} \mapsto 0]\}\}}}{I.\ \{emp\}\ C\ \{\exists j \geq 0.\ \{[\mathsf{c} \mapsto j]\}\}}\ \text{(instantiate)}}{I.\ \{emp\}\ C\ \{MC(\mathsf{c}, 0)\}}\ \text{(definition)}$$

The top part of the proof works like the original proof in Section A-2.3.

Proof of mc_inc. Let $C = (\mathsf{x} := [\mathsf{c}];\ [\mathsf{c}] := \mathsf{x}{+}1)$, i.e., the body of mc_inc.

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\forall j \geq i.\ I.\ \{\{[\mathsf{c} \mapsto j]\}\}\ C\ \{\{[\mathsf{c} \mapsto j+1]\}\}}}{\forall j \geq i.\ I.\ \{\{[\mathsf{c} \mapsto j]\}\}\ C\ \{\exists j \geq i+1.\ \{[\mathsf{c} \mapsto j]\}\}}\ \text{(instantiate)}}{I.\ \{\exists j \geq i.\ \{[\mathsf{c} \mapsto j]\}\}\ C\ \{\exists j \geq i+1.\ \{[\mathsf{c} \mapsto j]\}\}}\ \textsc{Exists}}{I.\ \{MC(\mathsf{c}, i)\}\ C\ \{MC(\mathsf{c}, i+1)\}}\ \text{(definition)}$$

The top part of the proof works like the original proof in Section A-2.3.

References

[BCOP05] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In *Proceedings of POPL*, pages 259–270, 2005.

[Boy07] John Boyland. Semantics of fractional permissions with nesting. Technical Report CS-07-01, University of Wisconsin-Milwaukee, Dept. of EE & CS, December 2007.

[DYDG+10] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *Proceedings of ECOOP*, 2010.

[DYGW11] Thomas Dinsdale-Young, Philippa Gardner, and Mark Wheelhouse. Abstraction and refinement for local reasoning, February 2011. Journal submission.

[GBC11] Alexey Gotsman, Josh Berdine, and Byron Cook. Precision and the conjunction rule in concurrent separation logic. In *Proceedings of MFPS*, 2011.

[O'H07] Peter W. O'Hearn. Resources, concurrency and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[PP11] Alexandre Pilkiewicz and François Pottier. The essence of monotonic state. In *Proceedings of TLDI*, 2011.

[TSFC09] Gang Tan, Zhong Shao, Xinyu Feng, and Hongxu Cai. Weak updates and separation logic. In *Proceedings of APLAS*, 2009.

High-Level Separation Logic for Low-Level Code

Jonas B. Jensen
IT University of Copenhagen
jobr@itu.dk

Nick Benton Andrew Kennedy
Microsoft Research Cambridge
{nick,akenn}@microsoft.com

Abstract

Separation logic is a powerful tool for reasoning about structured, imperative programs that manipulate pointers. However, its application to unstructured, lower-level languages such as assembly language or machine code remains challenging. In this paper we describe a separation logic tailored for this purpose that we have applied to x86 machine-code programs.

The logic is built from an assertion logic on machine states over which we construct a specification logic that encapsulates uses of frames and step indexing. The traditional notion of Hoare triple is not applicable directly to unstructured machine code, where code and data are mixed together and programs do not in general run to completion, so instead we adopt a continuation-passing style of specification with preconditions alone. Nevertheless, the range of primitives provided by the specification logic, which include a higher-order frame connective, a novel read-only frame connective, and a 'later' modality, support the definition of derived forms to support structured-programming-style reasoning for common cases, in which standard rules for Hoare triples are derived as lemmas. Furthermore, our encoding of scoped assembly-language labels lets us give definitions and proof rules for powerful assembly-language 'macros' such as while loops, conditionals and procedures.

We have applied the framework to a model of sequential x86 machine code built entirely within the Coq proof assistant, including tactic support based on computational reflection.

Categories and Subject Descriptors F.3.1 [*Logics and meanings of programs*]: Specifying and Verifying and Reasoning about Programs—Assertions, Invariants, Logics of programs, Mechanical verification, Pre- and post-conditions, Specification techniques; D.3.2 [*Programming Languages*]: Language Classifications—Macro and assembly languages; D.2.4 [*Software Engineering*]: Software / Program Verification—Correctness proofs, formal methods

General Terms Languages, theory, verification

Keywords Separation logic, machine code, proof assistants

1. Introduction

Formal verification is one of the most important techniques for building reliable computer systems. Research in software verification typically, and quite reasonably, concerns reasoning about the high-level programming languages with which most programmers work. But to build genuinely trustworthy systems, one really needs to verify the machine code that actually runs, whether it be hand-crafted or the output of a compiler. This is particularly important for establishing security properties, since failures of abstraction between the high and low-level models often lead to vulnerabilities (and because hand-crafted machine code is often found in security-critical places, such as kernels). A further motivation for verifying low-level code is that real systems are composed of components written in many different languages; machine code is the only truly universal lingua franca by which we can reason about properties of such compositions. Finally, experience shows that hand-written low-level programs are simply much harder to get right than higher-level ones, increasing the credibility gap between formal and informal verifications.

Verifying low-level, unstructured programs [18, 39], and compilers that produce them [24], both have a long history. And since such verifications are, like low-level programs themselves, extremely lengthy and error-prone, some form of mechanical assistance is absolutely crucial. This assistance often takes the form of automated decision procedures for first-order logic and various more specialized theories, combined by an SMT solver [21]. Here, however, our focus is on deductive Hoare-style verification of machine-code programs using an interactive proof assistant, in our case Coq. This requires more manual effort on the part of the user, but allows one to work with much richer mathematical models and specifications, which are particularly important for modularity. Both approaches to mechanization also have considerable history, with much pioneering work applying interactive provers to low-level code having been done with the Boyer-Moore prover in the 1980s [25, 41]. Recently, however, an exciting confluence of advances in foundational theory, program logics (most notably separation logic [35]) and the technology of proof assistants, together with increased interest in formal certification, have led to an explosion of work on mechanized verification of real (or at least, realistic) software, including compilers and operating systems. Although many of these formalizations do involve reasoning about machine code or assembly language programs, program logics for low-level code are generally much less satisfactory, and more ad hoc, than those for high-level languages.

The design of a high-level program logic tends to follow closely the structure and abstractions provided by the language. Commands in a while-language, for example, may be modelled as partial functions from stores to stores, which are only combined in certain very restricted ways. The classical Hoare triple, relating a predicate on inputs to a predicate on outputs, is a natural (indeed, inevitable) and generally satisfactory form of specification for such functions. Furthermore, the structured form of programs leads to particularly elegant, syntax-directed program logic rules for composing verifications. Machine code, by contrast, has almost nothing in the way of inherent structure or abstractions to guide one, supports chal-

lenging patterns of programming and also involves a host of messy complexities.

The messy complexities include large instruction sets with variable-length encodings, the need to work with bit-level operations and arithmetic mod 2^{32}, alignment, a plethora of flags, registers, addressing modes, and so on. These inevitably cause some pain, but are just the sort of thing proof assistants are good at checking precisely and, with a well-engineered formalization, removing some of the drudgery from.[1] There is, of course, complexity of a quite different order associated (at both high- and low-level) with concurrency – especially relaxed memory models on multiprocessors – which we do not address at all in this paper. Even in the sequential case, however, the lack of inherent structure in low-level code is a fundamental problem.

Machine code features unstructured control flow. A contiguous block of instructions potentially has many entry points and many exits, with the added complication that the same bytes may decode differently according to the entry point. Machine code is almost entirely untyped and higher-order, with no runtime tagging: any word in memory or a register may be treated as a scalar value, a pointer or a code pointer, and common coding patterns do make use of this flexibility: stealing bits in pointers, storing metadata at offsets from code pointers, computing branches, and so on. Finally, code and data live in the same heap, allowing code generation, self-modifying code and code examination, for example for interpreting instructions. The most basic abstractions, such as memory allocation or function calling, are not built in, but are conventions that must be specified, followed and verified at appropriate points. Furthermore, code that implements even the simplest of these abstractions, such as first-order function calls, uses features of machine code whose high-level analogues (higher-order, dynamically allocated local state) are challenging to reason about – and a subject of active research – even in very high-level languages such as ML.

Some logics, type systems and analyses for machine code deal with these complexities by imposing structure and restrictions on the code they deal with. For example, one can enforce a traditional basic block structure, hard-code memory allocation as a special pseudo-instruction or treat calling and a call-stack specially [5, 27, 31, 40]. Such techniques can work well for verifying code that looks like it came from a C compiler, but we would like something more generally applicable, able to verify smoothly higher-order code, systems code such as schedulers and allocators, and code that uses clever bit-level representation tricks. In previous work, for example on compiling a functional language to a rather idealized assembly language [7], one of us has proved useful results in Coq using a shallow embedding of step-indexed, separated predicates and relations, a notion of biorthogonality ('perping') for code pointers, explicit second-order quantification for framing, and a more-or-less ad hoc collection of lemmas for instructions, quantifier manipulation and entailment. Given sufficient effort, such an approach can undeniably be pushed through, but the proofs and specifications are very clumsy; although some of the connectives have respectable properties, there is certainly no sense that one is working in a well-structured program logic, with a well-behaved proof theory. Applying such a naive approach in the context of real machine code, with the above-mentioned messy complexities and in which we would clearly need to build numerous higher-level proof abstractions, seemed unlikely to work well.

Separation logics for higher-level languages, by contrast, do have a good proof theory. In particular, work on *higher-order* frame rules allows local reasoning about higher-order programs, allowing invariants to be framed onto commands in context by distributing them through the specifications of parameters [9]. A major goal of the work described here is to bring the power and concision of higher-order frame rules to reasoning about machine-code programs. At first sight, it may seem unclear how to incorporate even the first-order frame rule

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}$$

into a system for reasoning about machine code. Firstly, the frame rule is typically justified using a global property of commands with respect to a semantics defined over partial heaps: if a command executes without faulting in some heap, then it does so in any extension of that heap and moreover, if it terminates it preserves the extension. Partial heaps in the semantics model a built-in allocator, but, as in our previous work in low-level code [6, 7], we do not wish to define the ground semantics (with respect to which we interpret specifications) using partial heaps: whatever memory is in the machine is there all the time, and the allocator is just another piece of code to be specified and verified in our framework.

Secondly, the postcondition of a triple corresponds to the single exit point of a first-order command. Machine code fragments do not have single exits, or even a natural, local notion of terminating execution. We are ultimately concerned with the observable behaviour of whole programs, and do not wish to restrict ourselves to a form of specification that relies on the non-observable, intensional property of reaching a particular intermediate program counter value. We thus take our basic form of safety specifications to be one that only involves a precondition: execution from a given address in a state satisfying the precondition is safe. As Chlipala [16] observes, it is not obvious how to attach a frame soundly to such a specification. We address these problems by going beyond a shallow embedding of specifications of individual program points, to an embedding of a fully-fledged specification logic [23, 34], making the context within which code fragments are proved explicit, and with a semantics that captures (but a surface notation which hides) the way in which frames are preserved.

The specification logic allows one to work with subtle patterns of invariant preservation, but does not impose particular forms of specification. Rather, it provides building blocks from which more complex patterns, including Hoare triples, may be built. The rich, well-behaved theory of the core logic allows derived rules for new forms of specification to be expressed and proved concisely.

We have formalized our specification logic in the Coq proof assistant, and instantiated it for the particular case of a model for sequential x86 machine code. Our formalization also includes a range of reflective tactics for solving separation entailments and performing specification logic proofs at a high level of abstraction. This paper mainly discusses the logic in a machine-independent way, but we use the x86 instantiation for examples and motivation.

In summary, the contributions of this work include:

1. A separation logic for unstructured machine code that supports both first- and higher-order frame rules.
2. Accounts of specification-level connectives including framing, a 'read-only' frame, a 'later' modality and a full range of intuitionistic connectives, all with good logical properties.
3. Examples of higher-level patterns, such as Hoare triples, and associated proof rules being defined smoothly within the logic.
4. An certified assembler, supporting convenient macro definitions with internal label generation and natural derived proof rules, with examples including while-program constructs and procedure calling.

[1] Logics for high-level languages often ignore fixed-length arithmetic, even when that is what is provided by real implementations.

5. A semantics involving no instrumention or other modifications to the underlying machine model. Memory can be total, and step-counting and auxiliary variables happen only in the logic.
6. All this is formalized in Coq, with an instantiation for x86 machine code and tactic support for high-level proving. The formalization is available via the authors' web pages.

2. Machine model

Our separation logic is not tied to any particular machine architecture, but in order to illustrate its application we will be presenting examples from 32-bit x86, the architecture for which we have built a model in Coq.[2] In this section we present enough concrete detail of this model to support subsequent sections.

We have modelled a subset of the 32-bit x86 instruction set, considering sequential execution only, but treating memory, registers and flags in sufficient detail to obtain accurate specification of its behaviour.

Machine words and arithmetic We model n-bit machine words simply as n-nary tuples of boolean values, deploying an indexed type in Coq for the purpose. The x86 architecture makes various use of 8-bit (BYTE), 16-bit (WORD), 32-bit (DWORD) and 64-bit (QWORD) values, and so nat-dependent types in Coq are a boon to specification. Logical and arithmetic operations are defined directly in terms of bits, although to prove useful properties of arithmetic it proved handy to map words into arithmetic modulo 2^n, making use of the `ssreflect` library for algebraic identities [20].

Machine state The state of the machine is described by a triple of registers, flags and memory state:

$$\begin{aligned}\mathbb{S} = {} & (\mathsf{reg} \to \mathsf{DWORD}) \times \\ & (\mathsf{flag} \to \{\mathsf{true}, \mathsf{false}, \mathsf{undef}\}) \times \\ & (\mathsf{DWORD} \to (\mathsf{BYTE} \uplus \{\mathsf{unmapped}\}))\end{aligned}$$

Register state is a straightforward mapping from the x86's nine core registers (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP and EIP) to 32-bit values. The EIP instruction pointer register is the x86's program counter and points to the next instruction to be decoded by the processor. Rather than model the special EFLAGS as a monolithic register, we split it up into boolean-valued flags. The undef value represents the undefined state in which many instructions leave flags. Any dependence of execution on an undefined state, such as in a conditional branch instruction, is then treated as unspecified behaviour. Memory is modelled straightforwardly as 32-bit-addressable bytes, with the possibility that any byte might be missing or inaccessible. For now, we are not interested in finer distinctions such as read-only or no-execute, though it would be a simple matter to incorporate these notions. It is however important to note that the 'partiality' of memory has nothing to do with the partial states of separation logic; indeed we could choose to model and fully specify the x86 support for fault handling, using the very same logic.

Instructions Any machine model must of course include a datatype of instructions. The x86 instruction set is notoriously large and baroque but by careful subsetting and factoring our datatype is made reasonably concise. Figure 1 presents the instruction datatype, with only some names changed for the purposes of this paper.

Instruction decoding The x86 instruction format is also complex, being variable in length and not canonical: a single instruction can sometimes be encoded in many ways. We have implemented an instruction decoder as a partial function

$$\mathsf{decode} : \mathsf{DWORD} \times (\mathsf{DWORD} \to (\mathsf{BYTE} \uplus \{\mathsf{unmapped}\})) \rightharpoonup \mathsf{DWORD} \times \mathsf{Instr}$$

such that if $\mathsf{decode}(i, m) = (j, \iota)$ then memory m from address i up to but not including address j is defined and decodes to instruction ι. The decoder reads memory incrementally, returning an undefined value if memory is inaccessible or out of range, or if the contents do not describe an instruction in our chosen subset. (There is no need to specify explicitly a maximum instruction length.) Lifting decode to machine states, and threading the updating of the EIP register through, yields a partial function in $\mathbb{S} \rightharpoonup \mathbb{S} \times \mathsf{Instr}$.

Instruction execution Instruction execution is given by a partial function on states $\mathbb{S} \times \mathsf{Instr} \rightharpoonup \mathbb{S}$, which when composed with instruction decoding gives rise to a small-step transition function on machine states $\mathsf{step} : \mathbb{S} \rightharpoonup \mathbb{S}$. When this function is undefined, it means that either a fault occurred (such as the decoding of an illegal instruction or an access to unmapped memory), or behaviour is simply unspecified (such as branching on an undefined flag).

$\mathsf{Typeof}\ d = \text{if } d \text{ then } \mathsf{DWORD} \text{ else } \mathsf{BYTE}$

$\mathsf{Scale} = \mathsf{S}_1 \mid \mathsf{S}_2 \mid \mathsf{S}_4 \mid \mathsf{S}_8$

$\mathsf{MemSpec} = \mathsf{Reg} \times \mathsf{option}\ (\mathsf{NonSPReg} \times \mathsf{Scale}) \times \mathsf{DWORD}$

$\mathsf{RegMem} = \mathsf{RegMemR}\ (r{:}\mathsf{Reg}) \mid \mathsf{RegMemM}(ms{:}\mathsf{MemSpec})$

$\mathsf{RegImm}\ d = \mathsf{RegImmI}\ (c{:}\mathsf{Typeof}\ d) \mid \mathsf{RegImmR}\ (r{:}\mathsf{Reg})$

$\mathsf{Src} = \mathsf{SrcI}\ (c{:}\mathsf{DWORD}) \mid \mathsf{SrcM}\ (ms{:}\mathsf{MemSpec}) \mid \mathsf{SrcR}(r{:}\mathsf{Reg})$

$\mathsf{DstSrc}\ d =$
$\mid \mathsf{RR}\ (dst{:}\mathsf{Reg})\ (src{:}\mathsf{Reg})$
$\mid \mathsf{RM}\ (dst{:}\mathsf{Reg})\ (src{:}\mathsf{MemSpec})$
$\mid \mathsf{MR}\ (dst{:}\mathsf{MemSpec})\ (src{:}\mathsf{Reg})$
$\mid \mathsf{RI}\ (dst{:}\mathsf{Reg})\ (src{:}\mathsf{Typeof}\ d)$
$\mid \mathsf{MI}\ (dst{:}\mathsf{MemSpec})\ (src{:}\mathsf{Typeof}\ d)$

$\mathsf{BinOp} = \mathsf{ADC} \mid \mathsf{ADD} \mid \mathsf{AND} \mid \mathsf{CMP} \mid \mathsf{OR} \mid \mathsf{SBB} \mid \mathsf{SUB} \mid \mathsf{XOR}$

$\mathsf{UnaryOp} = \mathsf{INC} \mid \mathsf{DEC} \mid \mathsf{NOT} \mid \mathsf{NEG} \mid \mathsf{POP}$

$\mathsf{BitOp} = \mathsf{BT} \mid \mathsf{BTC} \mid \mathsf{BTR} \mid \mathsf{BTS}$

$\mathsf{ShiftOp} = \mathsf{ROL} \mid \mathsf{ROR} \mid \mathsf{RCL} \mid \mathsf{RCR} \mid \mathsf{SHL} \mid \mathsf{SHR} \mid \mathsf{SAL} \mid \mathsf{SAR}$

$\mathsf{Count} = \mathsf{ShiftCL} \mid \mathsf{ShiftImm}\ (b{:}\mathsf{BYTE})$

$\mathsf{Condition} = \mathsf{O} \mid \mathsf{B} \mid \mathsf{Z} \mid \mathsf{BE} \mid \mathsf{S} \mid \mathsf{P} \mid \mathsf{L} \mid \mathsf{LE}$

$\mathsf{Instr} =$
$\mid \mathsf{UOP}\ (d{:}\mathsf{bool})\ (op{:}\mathsf{UnaryOp})\ (dst{:}\mathsf{RegMem})$
$\mid \mathsf{BOP}\ d\ (op{:}\mathsf{BinOp})\ (ds{:}\mathsf{DstSrc}\ d)$
$\mid \mathsf{BITOP}\ (op{:}\mathsf{BitOp})\ (dst{:}\mathsf{RegMem}\ \mathsf{false})$
$\mid \mathsf{TEST}\ d\ (dst{:}\mathsf{RegMem})\ (src{:}\mathsf{RegImm}\ d)$
$\mid \mathsf{MOV}\ d\ (ds{:}\mathsf{DstSrc}\ d)$
$\mid \mathsf{SHIFTOP}\ (d{:}\mathsf{bool})\ (op{:}\mathsf{ShiftOp})\ (dst{:}\mathsf{RegMem})\ (c{:}\mathsf{Count})$
$\mid \mathsf{MUL}\ (src{:}\mathsf{RegMem})$
$\mid \mathsf{LEA}\ (reg{:}\mathsf{Reg})\ (src{:}\mathsf{RegMem})$
$\mid \mathsf{JCC}\ (cc{:}\mathsf{Condition})\ (dir{:}\mathsf{bool})\ (tgt{:}\mathsf{DWORD})$
$\mid \mathsf{PUSH}\ (src{:}\mathsf{Src})$
$\mid \mathsf{POP}\ (dst{:}\mathsf{RegMem})$
$\mid \mathsf{CALL}\ (src{:}\mathsf{Src}) \mid \mathsf{JMP}\ (src{:}\mathsf{Src})$
$\mid \mathsf{RET}\ (size{:}\mathsf{WORD})$
$\mid \mathsf{CLC} \mid \mathsf{STC} \mid \mathsf{CMC} \mid \mathsf{HLT}$

Figure 1. Instruction datatype

[2] There is no particular reason for choosing x86 over x64 or ARM.

3. Assertion logic

3.1 Partial states

Assertions in separation logic describe a subset, or 'footprint', of the machine state. For high-level imperative programs with dynamic allocation this footprint consists of a subset of the heap. Indeed a common idiom is to prove that some code starts or finishes with an 'empty' heap.

Here, there is no such thing: we have the whole machine at our disposal, and we must carve out our own abstractions such as heaps or stacks, so the footprint is simply that part of the state that we care about right now. We also find it useful to use separation in describing the manipulation of registers and flags, and so define *partial* states as follows, noting the resemblance to the definition of total states in Section 2.

$$\begin{aligned}\Sigma = {} & (\mathsf{reg} \rightharpoonup \mathsf{DWORD}) \times \\ & (\mathsf{flag} \rightharpoonup \{\mathsf{true}, \mathsf{false}, \mathsf{undef}\}) \times \\ & (\mathsf{DWORD} \rightharpoonup (\mathsf{BYTE} \uplus \{\mathsf{unmapped}\}))\end{aligned}$$

There is a partial binary operation $\uplus$ on elements of Σ, defined when its operands have disjoint domains on all three tuple components and yielding a tuple with the union of each of the maps. This makes $(\Sigma, \uplus)$ a *separation algebra* [15]; i.e., a partial commutative monoid.

3.2 Assertion logic

An assertion is a predicate on partial states:

$$\mathsf{asn} \triangleq \mathcal{P}(\Sigma)$$

Since $(\Sigma, \uplus)$ is a separation algebra, its powerset asn forms a complete boolean BI algebra, i.e., a model of the assertion language of classical separation logic, where the connectives are defined in the standard way [15]:

$$\begin{aligned}\forall x{:}T.\ P(x) &\triangleq \textstyle\bigcap_{x:T} P(x) \qquad \exists x{:}T.\ P(x) \triangleq \textstyle\bigcup_{x:T} P(x) \\ P \Rightarrow Q &\triangleq \{\sigma \mid \sigma \in P \Rightarrow \sigma \in Q\} \qquad \mathsf{emp} \triangleq \{([], [], [])\} \\ P * Q &\triangleq \{\sigma \mid \exists \sigma_1, \sigma_2.\ \sigma = \sigma_1 \uplus \sigma_2 \wedge \sigma_1 \in P \wedge \sigma_2 \in Q\} \\ P \mathrel{-\!\!*} Q &\triangleq \{\sigma_2 \mid \forall \sigma_1.\ \forall \sigma = \sigma_1 \uplus \sigma_2.\ \sigma_1 \in P \Rightarrow \sigma \in Q\}\end{aligned}$$

The propositional connectives $(\wedge, \top)$ and $(\vee, \bot)$ are just binary and nullary special cases of $\forall$ and $\exists$ respectively. As usual, entailment is defined as $P \vdash Q \triangleq P \subseteq Q$, and we write $\vdash P$ for $\top \vdash P$ and $P \equiv Q$ whenever $P \vdash Q$ and $Q \vdash P$.

There is a notion of points-to [35] for registers and for flags:

$$r \mapsto v \triangleq \{([r \mapsto v], [], [])\} \qquad f \mapsto b \triangleq \{([], [f \mapsto b], [])\}$$

The meaning of the points-to assertion for memory, $i..j \mapsto v$, depends on the type of v; this is done using a *type class* [37] in our Coq implementation. For BYTE and DWORD types, points-to means that memory from address i to j contains that value. In these cases, j is uniquely determined to be $i + 1$ or $i + 4$ respectively. For syntactic assembly instructions ι, it means that the memory at $i..j$ decodes to ι. In other words, $\mathsf{decode}(i, m) = (j, \iota)$ where m is the memory component of the state. Instruction encoding is not unique, so more than one byte sequence in memory may decode to the same ι. We write $i \mapsto v$ to mean $\exists j.\ i..j \mapsto v$.

Discussion. Another option would have been to let the registers and flags behave like the 'stack' in traditional separation logic [35] and not split them over $*$. This is the approach taken by Shao et al. [14, 31] and Chlipala [16], but it leads to the side condition on the frame rules that the program may not modify any registers mentioned by the frame. In a setting where programs have multiple entry and exit points and may be self-modifying, it is not even clear what that side condition means or how to check it, so we instead make registers and flags split across $*$, following Myreen et al. [29].

4. Specification logic

4.1 Safety

We extend the single-step partial function $\mathsf{step} : \mathbb{S} \rightharpoonup \mathbb{S}$ to a function $\mathsf{run} : \mathbb{N} \times \mathbb{S} \rightharpoonup \mathbb{S}$, where $\mathsf{run}(k, s)$ is the state that results from successful execution of k instructions starting from state s.

Unlike the high-level languages typically modelled with Hoare logics, a CPU has no natural notion of finishing a computation. It will run forever until it either faults or loses power[3]. This means that we cannot apply the standard Hoare-logic approach of describing a computation by a precondition and a postcondition since there is no meaningful time to check the postcondition.

Instead, specifications revolve around *safety*. We characterise the safe machine configurations as the set of pairs $(k, P) : \mathbb{N} \times \mathsf{asn}$ such that the machine will run for at least k steps without faulting if started from a state in P:

$$\mathsf{safe} \triangleq \{(k, P) \mid \forall \sigma \in P.\ \forall s \sqsupseteq \sigma.\ \exists s' : \mathbb{S}.\ \mathsf{run}(k, s) = s'\}$$

The relation $s \sqsupseteq \sigma$ states that all the mappings in σ are also found in s. This is how we connect the partial states found in assertions to the total states executed by the machine.

Example 1. It is safe to sit in a tight loop forever. That is,

$$\forall k, i.\ (k, (\mathsf{EIP} \mapsto i * i \mapsto \mathsf{jmp}\ i)) \in \mathsf{safe}$$

The EIP register is the instruction pointer, and $\mathsf{jmp}\ i$ is an unconditional jump to address i. The proof goes by induction on k. $\diamond$

The number k plays the role of a *step index* [2]. We are ultimately always interested in proving computations safe for an arbitrary number of steps, but exposing an intermediate step index gives us a value on which we can do induction.

As a running example, we will attempt to specify the unconditional jump instruction. We can show that for all i, a, k, R,

$$\begin{aligned}&(k, Q * R) \in \mathsf{safe} \Rightarrow (k + 1, P * R) \in \mathsf{safe} && \text{where} \\ &P = (\mathsf{EIP} \mapsto i * i \mapsto \mathsf{jmp}\ a) && \text{and} \\ &Q = (\mathsf{EIP} \mapsto a * i \mapsto \mathsf{jmp}\ a)\end{aligned}$$

In words, if you need to show that $P * R$ is a safe configuration for $k + 1$ steps, it suffices to show that $Q * R$ is safe for k steps. When a specification follows this pattern, we can think of P as a precondition, Q as a postcondition, and R as a frame.

The specification does not say that $Q * R$ will ever hold. Rather, it requires that if $Q * R$ does hold, then we are in a safe configuration. This can be seen as a CPS version of Hoare logic, which is appropriate for machine code since nothing ever returns or finishes at this level [5, 31, 38].

We will refine this specification in later examples as we develop constructions at higher levels of abstraction.

4.2 Specification logic

Reasoning directly about membership of safe is awkward since the step index and frame are explicit and visible even though their use always follows the same pattern. The solution is to instead consider safe as a formula in a *specification logic*. We define a specification to be a set of of (k, P)-pairs that is closed under decreasing k and under starring arbitrary assertions onto P:

$$\mathsf{spec} \triangleq \{S \subseteq \mathbb{N} \times \mathsf{asn} \mid \forall (k, P) \in S, k' \leq k, R.\ (k', P * R) \in S\}$$

[3] Even when it faults, it will typically reboot and so keep running, but this behaviour is outside of our model.

Intuitively, a specification $S : \mathsf{spec}$ describes how many steps the machine has to execute before it no longer holds and what frames the execution will preserve. This idea comes from the work of Birkedal, Torp-Smith and Yang on higher-order frame rules [8, 10], and spec is essentially a step-indexed version of Krishnaswami's specification logic model [23].

The definition of spec is such that $\mathsf{safe} \in \mathsf{spec}$. Furthermore, spec is a *complete Heyting algebra* and thus a model of intuitionistic logic. This gives us a notion of entailment ($\vdash \triangleq \subseteq$) and the logical connectives ($\forall, \exists, \wedge, \vee, \top, \bot, \Rightarrow$) with the expected rules. The definitions of the connectives follow a standard Kripke model:

$$\forall x{:}T.\ S(x) \triangleq \bigcap_{x:T} S(x) \qquad \exists x{:}T.\ S(x) \triangleq \bigcup_{x:T} S(x)$$
$$S \Rightarrow S' \triangleq \{(k, P) \mid \forall k' \le k.\ \forall R.\ (k', P * R) \in S \Rightarrow (k', P * R) \in S'\}$$

Again, the propositional connectives $(\wedge, \top)$ and $(\vee, \bot)$ are just binary and nullary special cases of $\forall$ and $\exists$ respectively.

Notice how the semantics of $\Rightarrow$ requires arbitrary frames to be preserved across the implication. This was not a choice we made – it is the only definition that makes $\Rightarrow$ be the right adjoint of $\wedge$, and it falls out of giving standard Kripke semantics.

We also get two new connectives: the *later* connective $\triangleright$ and the *frame* connective $\otimes$. We will define and discuss these in the next two subsections. They will enable us to state the specification of the jmp instruction from Section 4.1 more succinctly:

$$\triangleright\, \mathsf{safe} \otimes (\mathsf{EIP} \mapsto a * i \mapsto \mathsf{jmp}\ a) \vdash \mathsf{safe} \otimes (\mathsf{EIP} \mapsto i * i \mapsto \mathsf{jmp}\ a) \quad (1)$$

We can even factor out the duplicated part of the assertion and just write

$$\vdash (\triangleright\, \mathsf{safe} \otimes (\mathsf{EIP} \mapsto a) \Rightarrow \mathsf{safe} \otimes (\mathsf{EIP} \mapsto i)) \otimes i \mapsto \mathsf{jmp}\ a$$

or, informally, reading from right to left: 'given that i points to instruction jmp a, it is safe to execute with the instruction pointer set to i if it is later safe to execute with the instruction pointer set to a'.

4.3 Frame connective

Following the literature on higher-order frame rules [8, 10, 23], we define the *frame connective* $\otimes : \mathsf{spec} \times \mathsf{asn} \to \mathsf{spec}$ as

$$S \otimes R \triangleq \{(k, P) \mid (k, P * R) \in S\}$$

This is also known as the invariant extension connective [8] because an intuitive reading of $S \otimes R$ is that the computation described by S is allowed to additionally depend on and maintain the invariant R. Note that, by unpacking the definitions,

$$\vdash \mathsf{safe} \otimes P \quad \text{iff} \quad \forall k, R.\ (k, P * R) \in \mathsf{safe}$$

relating judgements in the specification logic with the safety of executions. Since we defined spec such that any S can be extended by any invariant, we can immediately prove the higher-order frame rule:

$$\frac{}{S \vdash S \otimes R}\ \textsc{Frame}$$

The frame connective distributes over all other connectives, including $\triangleright$ and itself. That means, for example, that

$$\frac{}{(S \Rightarrow S') \otimes R \ \equiv\ S \otimes R \Rightarrow S' \otimes R}\ \otimes\text{-}\Rightarrow$$

It also interacts with emp and $*$ as follows.

$$\frac{}{S \otimes \mathsf{emp} \ \equiv\ S}\ \otimes\text{-EMP}$$

$$\frac{}{S \otimes R_1 \otimes R_2 \ \equiv\ S \otimes (R_1 * R_2)}\ \otimes\text{-}*$$

Example 2. We can now start to see why Frame should be thought of as a frame rule. Assume we have proved for some P and Q that

$$\vdash \mathsf{safe} \otimes Q \Rightarrow \mathsf{safe} \otimes P.$$

Then by Frame, $\otimes$-$\Rightarrow$ and $\otimes$-$*$, we can derive

$$\vdash \mathsf{safe} \otimes (Q * R) \Rightarrow \mathsf{safe} \otimes (P * R).$$

Visually this looks like the standard frame rule, and it performs the same function: to extend both pre- and post-condition by an invariant. ◇

The formula $S \otimes R$ is covariant in S with respect to entailment, meaning that

$$\frac{S \vdash S'}{S \otimes R \vdash S' \otimes R}\ \otimes\text{-}\vdash$$

The variance in R is more complicated and will be discussed in Section 7.1.

Example 3. To illustrate informally how Frame generalises the standard first-order frame rule, consider a program in a high-level functional programming language $f_1 : (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$, whose specification is, for some particular P, Q and R,

$$\forall g.\ \{P * R\}\ g\ ()\ \{Q * R\} \Rightarrow \{P * R\}\ f_1(g)\ \{Q * R\}$$

That is, f_1 forwards the specification of g. Most likely, f_1 simply applies its argument to the unit value, but assume that it has been verified separately and we should not see its implementation.

If we have g_1 with specification $\{P\}\ g_1\ ()\ \{Q\}$, we cannot immediately apply $f_1(g_1)$ since the specification does not match what f_1 requires. However, we can apply the ordinary frame rule to deduce that g_1 also has the specification $\{P * R\}\ g_1\ ()\ \{Q * R\}$, and then we can call $f_1(g_1)$ if we are in a state satisfying $P * R$.

Now instead consider an f_2 with the specification

$$\forall g.\ \{P\}\ g\ ()\ \{Q\} \Rightarrow \{P\}\ f_2(g)\ \{Q\}$$

and a g_2 with specification $\{P * R\}\ g_2\ ()\ \{Q * R\}$. It is impossible with just the standard frame rule to call $f_2(g_2)$ since the specification of g_2 cannot be refined to match what is assumed by f_2. But with the higher-order frame rule, we can instead refine the specification of f_2 to be

$$\begin{aligned} &\forall g.\ (\{P\}\ g\ ()\ \{Q\} \Rightarrow \{P\}\ f_2(g)\ \{Q\}) \otimes R \equiv \\ &\forall g.\ \{P\}\ g\ ()\ \{Q\} \otimes R \Rightarrow \{P\}\ f_2(g)\ \{Q\} \otimes R \equiv \\ &\forall g.\ \{P * R\}\ g\ ()\ \{Q * R\} \Rightarrow \{P * R\}\ f_2(g)\ \{Q * R\} \end{aligned}$$

It is now compatible with our g_2.

Without the higher-order frame rule, we would have had to either re-verify the implementation of f_2 or generalise the original specification of f_2 to explicitly quantify over all possible frames that may be threaded through. The latter option is essentially what the definition of spec does, but this is invisible and implicit. ◇

Using $\otimes$ to give concise and modular specifications to higher-order functions is as important here as in any other separation logic, but that is not our main reason for including $\otimes$. We do it because it allows our logic to have a frame rule despite the program being unstructured and low-level. Chlipala [16] uses explicit second-order quantification in place of a frame rule, whilst Shao et al. [14, 31] have a frame rule that only applies to judgements in a very restrictive specification logic; in particular, it does not apply directly to specifications of function pointers.

4.4 Later connective

Just as we hide the explicit frames using $\otimes$, we hide the step indexes using the *later* connective, $\triangleright$. This is a trick pioneered by Nakano [30] that exploits the fact that we are never interested in the absolute number of steps but only that they are the same or differ

by exactly one between two specification formulas. We define

$$\triangleright S \triangleq \{(k, P) \mid \forall k' < k.\ (k', P) \in S\}$$

Because any $S : \mathsf{spec}$ is closed under decreasing steps, an equivalent definition is that $(0, P) \in \triangleright S$ for all P, and $(k+1, P) \in \triangleright S$ iff $(k, P) \in S$. The closure under decreasing steps is expressed logically as the rule

$$\frac{}{S \vdash \triangleright S}\ \triangleright\text{-WEAKEN}$$

As mentioned in Section 4.1, the purpose of step indexes is to serve as a handle for induction. We can phrase the induction principle on natural numbers using the following rule [3, 30], which is named for its similarity to a corresponding rule in Gödel-Löb logic [3].

$$\frac{\triangleright S \vdash S}{\vdash S}\ \text{LÖB}$$

The Löb rule is a reformulation of the strong induction principle for natural numbers: if $(\forall k' < k.\ P(k')) \Rightarrow P(k)$ for all k, then $P(n)$ holds for any n. It is a powerful rule in that it almost allows assuming the formula one wants to prove, except that the assumption may only be used after taking one step of computation.

Example 4. Recall the specification of a tight loop from Example 1. We can now express and prove that inside the specification logic in just two steps:

$$\dfrac{\dfrac{}{\triangleright\, \mathsf{safe} \otimes (\mathsf{EIP} \mapsto i * i \mapsto \mathsf{jmp}\ i)}\ (1)}{\vdash \mathsf{safe} \otimes (\mathsf{EIP} \mapsto i * i \mapsto \mathsf{jmp}\ i)}\ \text{LÖB}$$

$\diamond$

The $\triangleright$ connective distributes over every other connective we have mentioned except for $\bot$ and existential quantification over empty types.

Discussion. Like we saw in the rule for jmp (Equation (1)), every step of computation allows us to relax our remaining proof obligation by adding a $\triangleright$. For example, we could prove that

$$\vdash (\triangleright\triangleright \mathsf{safe} \otimes (\mathsf{EIP} \mapsto j) \Rightarrow \mathsf{safe} \otimes (\mathsf{EIP} \mapsto i)) \otimes i..j \mapsto (\mathsf{nop}; \mathsf{nop}) \quad (2)$$

where nop is the no-operation instruction. There are *two* $\triangleright$'s on the 'postcondition' of (2) because it takes two steps of computation to get there. It turns out, however, that it is never useful to have more than one $\triangleright$ applied to a specification since the purpose of step indexes is to do induction, and induction will always give us the necessary assumptions on the immediate predecessor of the number of interest.

Furthermore, we have found that $\triangleright$ is not necessary in code that only moves forward. Löb induction only makes sense when verifying loops, and a loop requires some form of backward jump unless we consider highly-contrived self-modifying code. Therefore, in practice, we would state (2) without any $\triangleright$-connective at all.

4.5 Read-only frame

The instruction rules we have discussed so far are too weak for some purposes. Recall the rule for jmp:

$$\vdash (\triangleright\, \mathsf{safe} \otimes (\mathsf{EIP} \mapsto a) \Rightarrow \mathsf{safe} \otimes (\mathsf{EIP} \mapsto i)) \otimes i \mapsto \mathsf{jmp}\ a$$

Because the meaning of $i \mapsto \mathsf{jmp}\ a$ is only that the memory starting at i decodes to jmp a, the rule would be satisfied in a semantics where the jmp instruction not only performed the jump but also replaced its own machine code in memory with a different byte sequence that also decoded to jmp a. This would be a problem for programs whose code needs to stay unmodified; e.g., to verify that the checksum of the code remains the same.

Our solution is to make this specification more precise by employing the *read-only frame* connective, defined as

$$S \oslash R \triangleq \forall \sigma \in R.\ S \otimes \{\sigma\}.$$

A more precise specification for jmp is then

$$\vdash (\triangleright\, \mathsf{safe} \otimes (\mathsf{EIP} \mapsto a) \Rightarrow \mathsf{safe} \otimes (\mathsf{EIP} \mapsto i)) \oslash i \mapsto \mathsf{jmp}\ a$$

Intuitively, $S \oslash R$ requires S not only to preserve the truth of R but to leave unmodified the underlying state fragment that made R true. The state may be changed temporarily, just as R might be broken, as long as it is restored at the end of the computation described by S.

Like for $\otimes$, there is a frame rule:

$$\frac{}{S \vdash S \oslash R}\ \text{FRAME-RO}$$

The $\oslash$ connective does not distribute over every other connective like $\otimes$ does, but it does distribute over $\forall, \wedge, \top, \otimes, \oslash, \triangleright$. It only distributes in one direction over $\exists$ and $\Rightarrow$:

$$\overline{(\exists a.\ S(a) \oslash R) \vdash (\exists a.\ S(a)) \oslash R}$$

$$\overline{(S \Rightarrow S') \oslash R \vdash S \oslash R \Rightarrow S' \oslash R}$$

The formula $S \oslash R$ is covariant in S and contravariant in R with respect to entailment, meaning that

$$\frac{S \vdash S' \qquad R' \vdash R}{S \oslash R \vdash S' \oslash R'}\ \oslash\text{-}\vdash$$

Another convenient property is that existential quantifiers can be moved in and out of the frame:

$$\overline{S \oslash (\exists x.\ R(x)) \ \equiv\ \forall x.\ S \oslash R(x)}$$

These last two properties of $\oslash$ about variance and commuting with existentials do not generally hold for $\otimes$. The cases where they hold are discussed in Sections 7.1 and 7.2. We explore further properties of $\oslash$ in Section 7.3.

Discussion. This connective is reminiscent of fractional permissions [12] but more coarse-grained and light-weight. We mention connections to other notions of weak ownership [22] in Section 9.

Our definition of $\oslash$ may not be the only good one, but we have examined three other candidate definitions and found that the one given above had the most convenient properties for our purposes. The candidates relate to each other as follows.

$$\begin{aligned} \forall R' \vdash R * \top.\ S \otimes R' \quad &\vdash \\ \forall R' \vdash R.\ S \otimes R' \quad &\vdash \\ \forall \sigma \in (R * \top).\ S \otimes \{\sigma\} \quad &\equiv \\ \forall \sigma \in R.\ S \otimes \{\sigma\} & \end{aligned}$$

5. High-level assembly code

5.1 Basic blocks

Using safe and the connectives discussed so far, we can specify code with multiple entry points and exit points, jumps to code pointers, self-modification, and so on. In practice, though, most code is much simpler. For code that behaves like a *basic block*, with control flow always coming in at the top and going out at the bottom, we can describe its behaviour with a Hoare triple, defined in the specification logic as:

$$\begin{aligned} \{P\}\ c\ \{Q\} \triangleq \forall i, j.\ (&\mathsf{safe} \otimes (\mathsf{EIP} \mapsto j * Q) \Rightarrow \\ &\mathsf{safe} \otimes (\mathsf{EIP} \mapsto i * P)) \oslash i..j \mapsto c \end{aligned}$$

Example 5. The instruction mov r, v (move literal v to register r) can be specified as $\vdash \{r?\}\ \mathsf{mov}\ r, v\ \{r \mapsto v\}$, where $r?$ is

shorthand for $\exists v.\ r \mapsto\!\triangleright\ v$. This is much more compact and readable than writing the specification in terms of safe. ◇

This triple satisfies the structural rules we expect from a Hoare triple in separation logic:

$$\frac{P \vdash P' \quad S \vdash \{P'\}\ c\ \{Q'\} \quad Q' \vdash Q}{S \vdash \{P\}\ c\ \{Q\}}$$

$$\frac{S \vdash \forall x.\ \{P(x)\}\ c\ \{Q\}}{S \vdash \{\exists x.\ P(x)\}\ c\ \{Q\}} \qquad \frac{S \vdash \{P\}\ c\ \{Q\}}{S \vdash \{P * R\}\ c\ \{Q * R\}}$$

The frame rule for the triple follows from FRAME and the fact that $\otimes$ distributes into the triple:

$$\frac{}{\{P\}\ c\ \{Q\} \otimes R\ \equiv\ \{P * R\}\ c\ \{Q * R\}}\ \otimes\text{-TRIPLE}$$

There is no rule of conjunction for the triple since this would be unsound in the presence of FRAME [8].

Discussion. This kind of triple is certainly not the only useful one. One could also adapt the position-indexed triples of Myreen and Gordon [29] to this setting, allowing use of the triple metaphor in specifying code with multiple entry and exit points. It would be a matter of taste whether this seemed more convenient to work with than reasoning directly in terms of safe.

The triple defined here can be thought of as encoding a very simple calling convention: inlining; i.e., concatenation of code. We envision defining triples for other conventions as needed and proving similar properties about them. See Section 5.5 for another example.

It is a valid question to ask why there is no $\triangleright$ on the postcondition part of the triple so it would read $\triangleright\ \mathsf{safe} \otimes (\mathsf{EIP} \mapsto\!\triangleright\ j * Q)$. It would give stronger specifications for single instructions like in Example 5, but as discussed in Section 4.4, it would also be unnecessary since control flow always moves forward in a triple. We will also see in Section 5.3 that there are useful values of c that take no computation steps.

5.2 Rules for x86 instructions

With a variety of logical building blocks in place, we can give appealingly simple rules for x86 instructions. These split into instructions that do not touch the instruction pointer, for which we can use the Hoare-triple form, and control flow instructions, for which we describe their effect on the instruction pointer explicitly.

Example 6. The following rule for 'add register indirect with offset' is a typical instance. Here d is a literal DWORD offset, and addition of two 32-bit values produces a pair (c, v) where v is the 32-bit (truncated) result, and c is the carry into bit 32.

$$\begin{array}{ll}\vdash & \{r_1 \mapsto\!\triangleright\ v_1 * \mathsf{OF}? * \mathsf{SF}? * \mathsf{ZF}? * \mathsf{CF}? * \mathsf{PF}?\} \\ & \mathsf{add}\ r_1, [r_2 + d] \\ & \{r_1 \mapsto\!\triangleright\ v * \mathsf{OF} \mapsto\!\triangleright\ \neg(\mathrm{msb}\ v_1 \oplus \mathrm{msb}\ v_2) \oplus \mathrm{msb}\ v \\ & \quad * \mathsf{SF} \mapsto\!\triangleright\ \mathrm{msb}\ v * \mathsf{ZF} \mapsto\!\triangleright\ (v = 0) * \mathsf{CF} \mapsto\!\triangleright\ c * \mathsf{PF} \mapsto\!\triangleright\ \mathrm{lsb}\ v\} \\ & \otimes\ (r_2 \mapsto\!\triangleright\ w * w + d \mapsto\!\triangleright\ v_2) \\ & \text{where } v_1 + v_2 = (c, v)\end{array}$$

The $\neg$ and $\oplus$ operators are boolean negation and xor respectively. The instruction affects flags OF, SF, ZF, CF and PF whose values initially are arbitrary ($F?$ is shorthand for $\exists f.\ F \mapsto\!\triangleright\ f$, where f may be undef). Notice the framing of invariant registers and memory. ◇

Example 7. For the jump-if-zero instruction, we specify two 'post-conditions', the first for when the branch is taken, and the second for when it isn't.

$$\begin{array}{l}\vdash (\triangleright\ \mathsf{safe} \otimes (b \wedge \mathsf{EIP} \mapsto\!\triangleright\ a * \mathsf{ZF} \mapsto\!\triangleright\ b) \wedge \\ \quad \mathsf{safe} \otimes (\neg b \wedge \mathsf{EIP} \mapsto\!\triangleright\ j * \mathsf{ZF} \mapsto\!\triangleright\ b) \\ \quad \Rightarrow \mathsf{safe} \otimes (\mathsf{EIP} \mapsto\!\triangleright\ i * \mathsf{ZF} \mapsto\!\triangleright\ b)) \\ \quad \oslash\ i..j \mapsto\!\triangleright\ \mathsf{jz}\ a\end{array}$$

Note the use of the *later* connective when the (possibly backwards) branch is taken. ◇

Our approach is to give a very general specification to each instruction and then on top of that provide convenience definitions for common cases. In a sense, our rules are therefore just a logical reformulation of the operational semantics, which may seem a bit unimpressive but turns out to be a strong platform on which to build higher-level layers of abstraction.

5.3 Instruction encoding and assembly language

We have implemented an encoder for syntactic instructions, and it has the property that

$$i..j \mapsto \mathsf{encode}(i, \iota) \vdash i..j \mapsto \iota$$

That is, if the memory at $i..j$ contains the sequence of bytes $\mathsf{encode}(i, \iota)$, then that memory will decode to the instruction ι. The instruction decoder referred to here is the same one that is part of the operational semantics for the machine. The encode function takes i as parameter because the encoding of x86 instructions is not generally position-independent.

This encoder is the main ingredient in our *assembler*: a certified and executable Coq function that takes a *program* as input and produces a list of bytes as output. A program is a value in the following inductive definition.

$$p ::= (\iota) \mid \mathsf{skip} \mid p; p \mid l\text{:} \mid \mathsf{LOCAL}\ l;\ p$$

That is, a program is essentially a list of instructions with label markers 'l:' interspersed. A label l may be declared local to program p with the $\mathsf{LOCAL}\ l;\ p$ construction. A label is simply a memory address; i.e., a 32-bit word, and it can therefore be used as an argument to jump instructions. The following is a closed program that loops forever.

$$\mathsf{LOCAL}\ l;\ l\text{:}\ \mathsf{jmp}\ l$$

The LOCAL constructor in our Coq implementation has type $(\mathsf{DWORD} \to \mathsf{program}) \to \mathsf{program}$, so writing $\mathsf{LOCAL}\ l;\ p$ is just syntactic sugar for $\mathsf{LOCAL}(\lambda l.\ p(l))$. The benefit of modelling label scopes with function spaces is that Coq handles all aspects of label naming transparently, including the necessary capture-avoidance and α-conversion. The downside is that it is not viable to statically rule out ill-formed programs, such as programs that place the same label more than once.

The assembler function, assemble, is partial and maps an address and a program to a sequence of bytes. It is undefined if the program is ill-formed. Where defined, it has the correctness property that

$$i..j \mapsto \mathsf{assemble}(i, p) \vdash i..j \mapsto p$$

Here, $i..j \mapsto p$ is defined recursively as follows.

$$\begin{array}{ll} i..j \mapsto (\iota) & \triangleq i..j \mapsto \iota \\ i..j \mapsto \mathsf{skip} & \triangleq i = j \wedge \mathsf{emp} \\ i..j \mapsto p_1; p_2 & \triangleq \exists i'.\ i..i' \mapsto p_1 * i'..j \mapsto p_2 \\ i..j \mapsto \mathsf{LOCAL}\ l;\ p & \triangleq \exists l.\ i..j \mapsto p(l) \\ i..j \mapsto l\text{:} & \triangleq i = j = l \wedge \mathsf{emp} \end{array}$$

Recall that the definition of triples $\{P\}\ c\ \{Q\}$ in Section 5.1 did not require c to have a particular type; the definition and its rules are valid for any c that can occur on the right of a points-to. Thus, we can put a program p in a triple, and it turns out that the following rules hold.

$$\frac{}{\vdash \{P\}\ \mathsf{skip}\ \{P\}} \qquad \frac{S \vdash \{P\}\ p_1\ \{Q\} \quad S \vdash \{Q\}\ p_2\ \{R\}}{S \vdash \{P\}\ p_1; p_2\ \{R\}}$$

$$\frac{S \vdash \{P\}\ \iota\ \{Q\}}{S \vdash \{P\}\ (\iota)\ \{Q\}} \qquad \frac{S \vdash \forall l.\ \{P\}\ p(l)\ \{Q\}}{S \vdash \{P\}\ \mathsf{LOCAL}\ l;\ p\ \{Q\}}$$

There is no useful rule for the case of l: in a triple.

Example 8. We cannot specify jmp with a triple in any useful way, but we can specify the special case of a tight loop shown above:

$$\vdash \{\mathsf{emp}\}\ \mathsf{LOCAL}\ l;\ l\colon \mathsf{jmp}\ l\ \{\bot\}$$

The proof is by first applying the triple rule for LOCAL, then unfolding the definition of the triple and applying the result of Example 4. ◇

5.4 Assembly macros

A useful assembly language has not only labels but also macros; i.e., parameterised definitions that expand to instruction sequences when invoked. We get macros almost for free since assembly programs are written and parsed inside Coq and can be intermixed with all the features of its term language. This includes let-bindings, fixpoint computations, custom syntax, coercions, overloading and other features of a modern dependently-typed programming language.

An example of a very useful macro is the following definition of $\mathsf{while}(p_1, t, b, p_2)$, where p_1 is a loop test, p_2 is a loop body, t encodes the combination of processor flags to be branched on, and b is a boolean that indicates whether the test should be inverted.

$$\begin{array}{rl}\mathsf{while}(p_1,t,b,p_2) \triangleq & \mathsf{LOCAL}\ l_1, l_2;\\ & \quad \mathsf{jmp}\ l_1;\\ & l_2\colon\ p_2;\\ & l_1\colon\ p_1;\\ & \quad \mathsf{jcc}\ t, b, l_2\end{array}$$

The jcc instruction is the general *conditional jump* on x86. We see here how LOCAL lets us declare labels that will be fresh for every invocation of the while macro. Real-world macro assemblers also have that functionality, although the scope is usually tied to the nearest named macro or global label. Our Coq notations for assembly syntax, including LOCAL, are chosen to be compatible with *MASM*, the Microsoft assembler.

Macros such as while give us the usual convenience of not having to write similar code many times. But even better, it lets us avoid writing similar proofs many times. If the body and test can be specified in terms of a triple, then the loop as a whole also has a triple specification:

$$\frac{\begin{array}{c}S \vdash \{P\}\ p_1\ \{\exists b'.\ I(b') * \mathsf{cond}(t,b')\}\\ S \vdash \{I(b) * \mathsf{cond}(t,b)\}\ p_2\ \{P\}\end{array}}{S \vdash \{P\}\ \mathsf{while}(p_1,t,b,p_2)\ \{I(\neg b) * \mathsf{cond}(t,\neg b)\}}\ \textsc{While}$$

Here, $\mathsf{cond}(t, b)$ translates t, of type Condition from Figure 1, to an assertion that tests the relevant flags. For example, $\mathsf{cond}(\mathsf{Z}, b) = \mathsf{ZF} \mapsto\!\triangleright\, b$, where ZF is the *zero flag*. There are two loop invariants, P and I, representing the state before and after executing the test p_1 since this may have side effects.

The proof of the WHILE rule involves $\triangleright$-operators and the LÖB rule, but these technicalities do not leak out into the rule statement.

With if and while macros and the sequence operator on programs, we have the building blocks to easily write and verify programs with structured control flow. These constructs also facilitate using our assembly language as the target of a verified compiler from a structured language, which is something we hope to investigate more in future work.

5.5 Procedure calls

The triple $\{P\}\ c\ \{Q\}$ encodes and abstracts the often-occurring programming pattern of structured control flow. Another crucial pattern to capture is procedure calls. We will here show the theory of a very simple calling convention [29]: store the return address in register EDX and jump to the procedure entry point. The following macro calls the procedure whose code is at address f.

$$\begin{array}{rl}\mathsf{call}\ f \triangleq & \mathsf{LOCAL}\ i_{\text{ret}};\\ & \quad \mathsf{mov}\ \mathsf{EDX}, i_{\text{ret}};\\ & \quad \mathsf{jmp}\ f;\\ & i_{\text{ret}}\colon\end{array}$$

The calling convention does not specify how to pass arguments or return values; this is instead part of individual procedure specifications. A more realistic calling convention would maintain a stack of arguments and return addresses to allow deep call hierarchies and reentrancy, but this would clutter our examples with arithmetic side conditions because the stack has to be finite [29].

The following definition describes the behaviour of a procedure starting at f with precondition P and postcondition Q.

$$\begin{array}{rl}f{\mapsto}\{P\}\{Q\} \triangleq \forall i_{\text{ret}}.\ & \mathsf{safe} \otimes (\mathsf{EIP} \mapsto i_{\text{ret}} * \mathsf{EDX?} * Q) \Rightarrow\\ & \mathsf{safe} \otimes (\mathsf{EIP} \mapsto f * \mathsf{EDX} \mapsto i_{\text{ret}} * P)\end{array}$$

Recall that EDX? is shorthand for $\exists v.\ \mathsf{EDX} \mapsto v$. This definition satisfies the usual rules for a triple-like formula, including

$$\frac{}{f{\mapsto}\{P\}\{Q\} \otimes R\ \equiv\ f{\mapsto}\{P * R\}\{Q * R\}}\ \otimes\text{-}\textsc{Proc}$$

In contrast with the triple defined in Section 5.1, this definition of a procedure specification does not mention the code stored at f. The code should be mentioned separately from its behaviour such that the footprint of the code covers both the caller and the callee.

The rule for calling a procedure looks fairly standard:

$$\frac{}{\triangleright f{\mapsto}\{P\}\{Q\} \vdash \{P\}\ \mathsf{call}\ f\ \{Q\} \otimes \mathsf{EDX?}}\ \textsc{Call}$$

It reveals that EDX is overwritten as part of the calling convention. The $\triangleright$ modality on the premise, together with LÖB, permits recursion [3].

Example 9. This is the first of three examples to illustrate independent verification of caller and callee. Consider the following definition of a program that calls some procedure at f twice:

$$p_{\text{caller}}(f) \triangleq \mathsf{call}\ f;\ \mathsf{call}\ f$$

If the intention with this program is to compose it with a procedure that satisfies

$$S_{\text{callee}}(f) \triangleq \forall a.\ f{\mapsto}\{\mathsf{EAX} \mapsto a\}\{\mathsf{EAX} \mapsto a + 2\},$$

then we can specify the caller as

$$S_{\text{callee}}(f) \vdash \{\mathsf{EAX} \mapsto a\}\ p_{\text{caller}}(f)\ \{\mathsf{EAX} \mapsto a + 4\} \otimes \mathsf{EDX?}$$

We can prove this specification directly from the program sequencing rule and CALL. No $\triangleright$ connective is put on the assumption since no recursion is intended. ◇

If a procedure body p is structured and returns at its very end, we can prove its specification through the following rule.

$$\frac{S \vdash \{P\}\ p\ \{Q\} \oslash \mathsf{EDX?}}{S \vdash f{\mapsto}\{P\}\{Q\} \oslash f \mapsto (p;\ \mathsf{jmp}\ \mathsf{EDX})}\ \textsc{Body}$$

In words, this means that calling f behaves as (P, Q) when in memory where the program $(p;\ \mathsf{jmp}\ \mathsf{EDX})$ is at address f, assuming we can prove the given triple, which is allowed to access EDX as long as it restores its value in the end.

Example 10. The following program almost satisfies S_{callee} as defined in Example 9.

$$p_{\text{callee}} \triangleq \mathsf{inc}\ \mathsf{EAX};\ \mathsf{inc}\ \mathsf{EAX};\ \mathsf{jmp}\ \mathsf{EDX}$$

We say *almost* because the inc instruction affects the status flags of the CPU as a side effect. The caller is not interested in the flags, but they have to be in the specification of p_{callee} since they

do get affected. Let flags be the assertion that all flags are of some (existential) value. Then we can prove

$$\vdash S_{\text{callee}}(f) \otimes \mathsf{flags} \oslash f \mapsto p_{\text{callee}}.$$

The proof is by applying BODY, whose conclusion matches the above specification after rewriting by $\otimes$-PROC. $\diamond$

The next example demonstrates how to compose a caller and a callee, even if the callee has a larger footprint than what the caller assumes. This shows how to execute the informal reasoning from Example 3 in our logic.

Example 11. We can now compose the implementations of the caller from Example 9 and the callee from Example 10 to obtain the following closed program. We arbitrarily choose to place the callee in memory before the caller.

$$p_{\text{main}}(\mathit{entry}) \triangleq \mathsf{LOCAL}\ f;\ f\colon p_{\text{callee}};\ \mathit{entry}\colon p_{\text{caller}}(f)$$

We can give the following specification to this program, which says that the code between entry and j will increment EAX by 4 and step on EDX and the flags.

$$\begin{array}{l}\vdash (\mathsf{safe} \otimes (\mathsf{EIP} \mapsto j * \mathsf{EAX} \mapsto a + 4) \Rightarrow \\ \quad \mathsf{safe} \otimes (\mathsf{EIP} \mapsto \mathit{entry} * \mathsf{EAX} \mapsto a) \\ \quad) \otimes (\mathsf{EDX}? * \mathsf{flags}) \oslash (i..j \mapsto p_{\text{main}}(\mathit{entry}))\end{array}$$

The crucial step in proving this specification is to satisfy the caller's assumption, S_{callee}, with the callee specification, which is essentially $S_{\text{callee}} \otimes \mathsf{flags}$. The former entails the latter, but here we would need the entailment to go the other way. Instead, we exploit that FRAME is a higher-order frame rule [9] and lets us frame an assertion on to the left and right side of an entailment simultaneously. This is allowed by the rules $\otimes$-$\vdash$ and $\oslash$-$\vdash$ from Sections 4.3 and 4.5. Abbreviating

$$S_{\text{caller}}(f) \triangleq \forall a.\ \{\mathsf{EAX} \mapsto a\}\ p_{\text{caller}}(f)\ \{\mathsf{EAX} \mapsto a + 4\} \otimes \mathsf{EDX}?,$$

we can derive

$$\dfrac{\dfrac{\dfrac{}{S_{\text{callee}}(f) \vdash S_{\text{caller}}(f)}\ \text{Ex. 9}}{S_{\text{callee}}(f) \otimes \mathsf{flags} \vdash S_{\text{caller}}(f) \otimes \mathsf{flags}}\ \otimes\text{-}\vdash}{S_{\text{callee}}(f) \otimes \mathsf{flags} \oslash f \mapsto p_{\text{callee}} \vdash S_{\text{caller}}(f) \otimes \mathsf{flags} \oslash f \mapsto p_{\text{callee}}}\ \oslash\text{-}\vdash$$

We know from Example 10 that $\vdash S_{\text{callee}}(f) \otimes \mathsf{flags} \oslash f \mapsto p_{\text{callee}}$, so by transitivity of $\vdash$ we conclude $\vdash S_{\text{caller}}(f) \otimes \mathsf{flags} \oslash f \mapsto p_{\text{callee}}$. From this, it is straightforward to derive our desired specification for p_{main}. $\diamond$

The preceding example showed how to use FRAME as a *second-order* [9], or, *hypothetical* [32] frame rule. The procedure involved was first-order at run-time, though. The following example involves a proper higher-order procedure; i.e., a procedure that takes a pointer to another procedure as argument.

Example 12. The simplest example of a higher-order procedure is 'apply', which in a functional programming language would be defined as

$$\mathsf{apply}(g, x) = g(x).$$

In our set-up, an apply procedure that takes its g argument in register EBX is implemented simply as

$$p_{\text{apply}} \triangleq \mathsf{jmp\ EBX}.$$

Its specification reflects how it forwards the behaviour (P, Q) of g:

$$\begin{array}{l}\vdash \big(g \mapsto \{P * \mathsf{EBX}?\}\{Q\} \Rightarrow \\ \quad f \mapsto \{P * \mathsf{EBX} \mapsto g\}\{Q\}\big) \oslash f \mapsto p_{\text{apply}}.\end{array} \quad \diamond$$

6. Practical verification

We have used our Coq development not only to build a machine model and to validate the logic developed in this paper; it is also an environment for building and verifying actual machine-code programs.

In this section we describe the Coq tactic support that we have developed for making machine code verification manageable, and present a slightly larger example of assembly language (seven instructions!) in order to give a flavour of the Coq proof of its correctness.

6.1 Example: memory allocation

We illustrate the use of the logic, rules, and Coq tactics with a slightly more challenging example: the specification of a memory allocator and its simplest possible realisation, the bumping of a pointer and checking it against a limit.

Its specification is as follows, parameterized by the number of bytes n to be allocated and an address fail to jump to on failure.

$$\begin{array}{l}\mathsf{allocSpec}(n, \mathit{fail}, \mathit{inv}, \mathit{code}) \triangleq \forall i, j. \\ \quad (\mathsf{safe} \otimes (\mathsf{EIP} \mapsto \mathit{fail} * \mathsf{EDI}?) \wedge \\ \quad\ \mathsf{safe} \otimes (\mathsf{EIP} \mapsto j * \exists a.\ (\mathsf{EDI} \mapsto a{+}n) * (a\ ..\ a{+}n \mapsto \_)) \Rightarrow \\ \quad\ \mathsf{safe} \otimes (\mathsf{EIP} \mapsto i * \mathsf{EDI}?)) \\ \quad \otimes (\mathsf{ESI}? * \mathsf{flags} * \mathit{inv}) \oslash (i..j \mapsto \mathit{code})\end{array}$$

The specification is framed by an assertion that register ESI is used as scratch storage, flags are updated arbitrarily, and an internal invariant inv is maintained. The latter might be the well-formedness of some representation of free lists, or in our trivial allocator, simply a pair of pointers.

The calling convention is 'inline', in other words, the allocator is just a macro consisting of assembly in code. In Section 6.5, we will wrap a slightly less trivial calling convention around it.

Control either drops through, if successful, or branches to address fail, if memory cannot be allocated. On success, the allocator leaves an address a in EDI that is just beyond the n bytes of memory that were allocated; on failure, EDI is trashed.

Perhaps surprisingly, even a bump-and-check implementation consists of seven instructions:

$$\begin{array}{l}\mathsf{allocImp}(\mathit{info}, n, \mathit{fail}) \triangleq \mathsf{mov\ ESI}, \mathit{info}; \\ \qquad\qquad\qquad\qquad\quad \mathsf{mov\ EDI}, [\mathsf{ESI}]; \\ \qquad\qquad\qquad\qquad\quad \mathsf{add\ EDI}, n; \\ \qquad\qquad\qquad\qquad\quad \mathsf{jc}\ \mathit{fail}; \\ \qquad\qquad\qquad\qquad\quad \mathsf{cmp}\ [\mathsf{ESI} + 4],\ \mathsf{EDI}; \\ \qquad\qquad\qquad\qquad\quad \mathsf{jc}\ \mathit{fail}; \\ \qquad\qquad\qquad\qquad\quad \mathsf{mov}\ [\mathsf{ESI}],\ \mathsf{EDI}.\end{array}$$

The implementation invariant inv is the following:

$$\begin{array}{l}\mathsf{inv}(\mathit{info}) \triangleq \exists \mathit{base}, \mathit{limit}. \\ \quad \mathit{info} \mapsto \mathit{base} * (\mathit{info} + 4 \mapsto \mathit{limit}) * (\mathit{base}\ ..\ \mathit{limit} \mapsto \_).\end{array}$$

In other words, at address info there is a pair of pointers base and limit that bound a piece of mapped memory.

6.2 Applying instruction rules

During a proof, we typically keep the goal in the form

$$S_{\text{ctx}} \vdash (S \Rightarrow \mathsf{safe} \otimes P) \oslash R.$$

The specifications discussed in this paper are easy to put into that form by applying distributivity rules for $\otimes$ and decurrying nested implications, and we have implemented a tactic to do this automatically. Typically, R describes the code to be executed, and P describes the instruction pointer and the remaining state that will go into proving the precondition of the next instruction.

We may use the full range of specification-logic rules on this goal, but eventually we will want to apply the lemma appropriate

for the code that EIP is pointing to in P. We assume that the lemma has the same form as the goal and apply a lemma through the following rule.

$$\frac{\begin{array}{l} S'_{\text{ctx}} \vdash (S' \Rightarrow \mathsf{safe} \otimes P') \oslash R' \\ S_{\text{ctx}} \vdash S'_{\text{ctx}} \\ P \ \vdash P' * R_{\text{P}} \\ R \ \vdash R' * \top \\ S_{\text{ctx}} \vdash (S \Rightarrow S' \otimes R_{\text{P}}) \oslash R \end{array}}{S_{\text{ctx}} \vdash (S \Rightarrow \mathsf{safe} \otimes P) \oslash R} \ \textsc{SpecApply}$$

The top premise is the lemma to be applied, and the bottom premise is the remaining proof obligation that describes the symbolic state after having applied the lemma. If the lemma is an instruction rule, the three middle premises correspond to satisfying its preconditions at the level of specifications, data memory and code memory respectively. The latter two can be dealt with by our entailment checker, described in the next subsection.

6.3 Assertion entailment solving

Much of the activity in a formal separation logic proof is proving entailment between assertions. This happens every time a precondition needs to be discharged, and if it is not automated, the proofs will drown in the details of fragile manual context manipulation and rewriting modulo associativity and commutativity.

Typically, we are given a description of the current state P and a precondition P', and we must show $P \vdash P' * R$ for our own choice of frame R, which represents all the left-over state that was not consumed by the precondition and can therefore be framed out. Our approach to this automation is similar to other separation-logic tools [4, 16]: if P and P' consist only of $*$, emp and atomic assertions, we iterate through the conjuncts of P', attempting to unify each with a conjunct found in P and let the two cancel out.

Typically, P' is full of holes corresponding to universally-quantified variables that have yet to be instantiated. The holes are represented in Coq as *unification variables*, which are identifiers that will receive a value upon being unified with a subformula from P. Several subformulas of P may unify, but typically only one choice will permit the entailment as a whole to be solved. For example, we may be proving

$$\mathsf{EAX} \mapsto i * j \mapsto 2 * i \mapsto 1 \vdash \mathsf{EAX} \mapsto U_1 * U_1 \mapsto U_2 * \top,$$

where U_1 and U_2 are unification variables of type DWORD. If our algorithm should attempt to unify the atom $U_1 \mapsto U_2$ with the atom $j \mapsto 2$, it will succeed, but the remaining proof obligation will be

$$\mathsf{EAX} \mapsto i * i \mapsto 1 \vdash \mathsf{EAX} \mapsto j * \top$$

The algorithm succeeds even if it did not solve the goal entirely, leaving the rest to be proved interactively, but in this case there is no solution for the remaining part of the goal.

Rather than try to support backtracking, which does not combine well with interactive proof, we make the algorithm greedy but predictable: subformulas of P' are unified from left to right. In our current example, this would first fix the choice of U_1 to be i, and the second conjunct of P' would therefore become $i \mapsto U_2$, which rules out the bad unification choice from before.

There is of course no guarantee that this always works, but we have found that it virtually always works in practice as long as preconditions are written with this left-to-right order in mind. This happens naturally since it is also more readable for humans who read from left to right. If the algorithm should still fail, it remains possible to manually instantiate the unification variables.

The entailment solving algorithm is implemented with a hybrid approach, where the unification is done by Coq's built-in higher-order unification engine, while the cancellation of identical terms is done with *proof by reflection* [17], which has good performance.

If an entailment has existential quantifiers on the left-hand side, we can apply the rule

$$\frac{\forall x.\ (\mathcal{C}[P(x)] \vdash Q)}{\mathcal{C}[\exists x.\ P(x)] \vdash Q}$$

where $\mathcal{C}$ is formula with a hole that contains only $*$-connectives in the path from the root to the hole. This lets us effectively move the quantified variable into the Coq variable context.

If an entailment has existential quantifiers on the right-hand side, we will eventually need to instantiate them with witnesses. This can be done with the rule

$$\frac{\exists x.\ (P \vdash \mathcal{C}[Q(x)])}{P \vdash \mathcal{C}[\exists x.\ Q(x)]}$$

We immediately apply the rule, but we instantiate x with a *unification variable*, which in practice defers instantiation until a unification forces it to happen as described above.

We have extended the tactic for moving quantifiers into the context so it also works on specification-logic entailments. For example, given the goal

$$S' \vdash \forall x_1.\ S \Rightarrow \mathsf{safe} \otimes (\exists x_3.\ P(x_1, x_3)) \oslash (\exists x_2.\ R(x_2)),$$

the extended tactic will introduce x_1, x_2 and x_3 into the Coq variable context and leave the new goal

$$S' \vdash S \Rightarrow \mathsf{safe} \otimes P(x_1, x_3) \oslash R(x_2).$$

The rules allowing x_2 and x_3 to be pulled out are given in Sections 4.5 and 7.2 respectively.

6.4 Proving the allocator

Correctness consists of proving the following, for any $info$, n, $fail$.

$$\vdash \mathsf{allocSpec}(n, fail, \mathsf{inv}(info), \mathsf{allocImp}(info, n, fail)).$$

Here is a fragment of the Coq proof script that deals with the second instruction in the implementation. (We make use of `ssreflect` extensions to standard Coq tactic notation [20].)

```
(* mov EDI, [ESI] *)
rewrite {2}/inv. specintros => base limit.
specapply MOV_RM0_rule.
- by ssimpl.
```

For this instruction, almost everything is handled automatically. The initial `rewrite` simply unfolds the invariant inv to expose the existential quantifiers. The custom tactic `specintros` pulls the existentially-quantified variables from deep within the goal to introduce them into the Coq context. The tactic '`specapply` l' will first normalise both the goal and lemma l to have the form required by the SPECAPPLY rule from Section 6.2. It will then invoke SPECAPPLY with l as the first premise. In this case, l is `MOV_RM0_rule`, the rule for instructions of the form mov $r_1, [r_2]$. The second and fourth premises are trivial, leaving only the precondition of the mov rule as a subgoal. This can be discharged by our `ssimpl` tactic, which implements entailment checking as described in Section 6.3.

In fact none of the instructions needs more than four lines of proof, and we hope to reduce this further through the use of additional lemma and tactic support once we have more experience with proving.

6.5 Wrapping the allocator

Having verified a component, such as the allocator, it is reasonably straightforward to use the logic to verify higher-level abstractions in a modular way. As an example, we show the wrapping of the allocator in a procedure for consing onto a list.

We start with the inductive 'list segment' assertion of separation logic (originally due to Burstall [13]):

$$\mathsf{listSeg}(p, e, vs) \triangleq \begin{cases} \exists q.\ (p \mapsto v) * (p{+}4 \mapsto q) * \mathsf{listSeg}(q, e, vs') & \text{if } vs = v :: vs' \\ p = e \wedge \mathsf{emp} & \text{otherwise} \end{cases}$$

Here, vs is a list of DWORDs and the assertion says that memory contains a linked list starting at p and ending at e with elements given by vs. A possible specification for our cons function is

$$\begin{array}{l} \mathsf{consSpec}(r_1, r_2, \mathit{info}, i, j, \mathit{code}) \triangleq \forall h, t, e, vs. \\ (i \mapsto \{r_1 \mapsto h * r_2 \mapsto t * \mathsf{listSeg}(t, e, vs) * \mathsf{EDI}?\} \\ \qquad \{r_1? * r_2? * ((\mathsf{EDI} \mapsto 0 * \mathsf{listSeg}(t, e, vs)) \vee \\ \qquad\qquad (\exists q.\ \mathsf{EDI} \mapsto q * \mathsf{listSeg}(q, e, h :: vs))\} \\ ) \otimes (\mathsf{ESI}? * \mathsf{flags} * \mathsf{inv}(\mathit{info})) \oslash (i..j \mapsto \mathit{code}) \end{array}$$

specifying a procedure that is passed a value h in r_1 and a pointer to a list starting at t in r_2. On return, EDI is either zero, and the original linked list is preserved, or EDI points to a linked list segment ending at e with h added as the new head element. An implementation is given by

$$\begin{array}{rl} \mathsf{cons}(r_1, r_2, \mathit{info}) \triangleq & \mathsf{LOCAL}\ \mathit{fail};\ \mathsf{LOCAL}\ \mathit{succeed}; \\ & \mathsf{allocImp}(\mathit{info}, 8, \mathit{fail}); \\ & \mathsf{sub}\ \mathsf{EDI}, 8; \\ & \mathsf{mov}\ [\mathsf{EDI}], r_1; \\ & \mathsf{mov}\ [\mathsf{EDI}+4], r_2; \\ & \mathsf{jmp}\ \mathit{succeed}; \\ & \mathit{fail}: \\ & \mathsf{mov}\ \mathsf{EDI}, 0; \\ & \mathit{succeed}: \\ & \mathsf{jmp}\ \mathsf{EDX} \end{array}$$

The proof that for any $r_1, r_2, \mathit{info}, i$ and j,

$$\vdash \mathsf{consSpec}(r_1, r_2, \mathit{info}, i, j, \mathsf{cons}(r_1, r_2, \mathit{info}))$$

is entirely modular, relying on the BODY rule and the previous result that allocImp meets allocSpec.

7. Properties of the frame connectives

We now return to the frame connective, $\otimes$, defined in Section 4.3. In previous literature on higher-order frame rules [8–10, 33], the R in $S \otimes R$ tends to be inert and does not interact with its environment until it has distributed inwards across all connectives and has been merged into the pre- and post-conditions of a triple. Only at that point will the rule of consequence and the existential rule for triples be used to interact with R.

Since we only see triples in certain special cases, as described in Section 5.1, we are interested in specification-level generalisations of the consequence and existential rules, just as FRAME is a specification-level generalisation of the frame rule for triples. The use of these generalised rules in practice is similar to their counterparts in ordinary Hoare logic.

7.1 Specification-level rule of consequence

The standard Hoare rule of consequence states that $\{P\}\ c\ \{Q\}$ is contravariant in P and covariant in Q with respect to entailment. Analogously, the generalisation we present here describes the variance of $S \otimes R$ in R. It turns out that $S \otimes R$ is not always covariant nor always contravariant in R; it can be either, depending on S. We encode this as two predicates on S:

$$\mathsf{frame}_+(S) \triangleq \forall P, Q.\ (P \vdash Q) \Rightarrow (S \otimes P \vdash S \otimes Q)$$
$$\mathsf{frame}_-(S) \triangleq \forall P, Q.\ (P \vdash Q) \Rightarrow (S \otimes Q \vdash S \otimes P)$$

These definitions directly give rise to our two specification-level rules of consequence:

$$\frac{\mathsf{frame}_+(S) \quad P \vdash Q}{S \otimes P \vdash S \otimes Q} \qquad \frac{\mathsf{frame}_-(S) \quad P \vdash Q}{S \otimes Q \vdash S \otimes P}$$

All we did so far was to switch the problem to proving $\mathsf{frame}_+(S)$ or $\mathsf{frame}_-(S)$ for particular S, but it turns out that there is a very schematic set of rules for this. Writing $f : (V_1, \ldots, V_n) \longrightarrow V$ to mean

$$\forall S_1, \ldots, S_n.\ \mathsf{frame}_{V_1}(S_1) \wedge \ldots \wedge \mathsf{frame}_{V_n}(S_n) \Rightarrow \mathsf{frame}_V(f(S_1, \ldots, S_n)),$$

we can tabulate the rules for various connectives concisely:

$$\begin{array}{r} \mathsf{safe} : - \\ \top, \bot : + \text{ and } - \\ \triangleright, \otimes, \oslash, \forall, \exists : + \longrightarrow + \text{ and } - \longrightarrow - \\ \wedge, \vee : (+, +) \longrightarrow + \text{ and } (-, -) \longrightarrow - \\ \Rightarrow : (-, +) \longrightarrow + \text{ and } (+, -) \longrightarrow - \end{array}$$

Notice that all the logical connectives preserve either covariance or contravariance of their operands (modulo the flip that happens for implication), but there is no way to combine the variances.

Example 13. For all P_1 and P_2, $\mathsf{frame}_-(\triangleright\mathsf{safe} \otimes P_1 \wedge \mathsf{safe} \otimes P_2)$.◇

Example 14. For all P, $\mathsf{frame}_+(\mathsf{safe} \otimes P \Rightarrow \bot)$. ◇

There are no definitions analogous to $\mathsf{frame}_+(S)$ and $\mathsf{frame}_-(S)$ for the read-only frame connective since $S \oslash R$ is always contravariant in R. But as we will see in Section 7.3, the frame family of predicates plays an important role for $\oslash$ too.

It is no coincidence that these rules for variance have not been studied in the previous literature. There is no rule for frame_+ or frame_- on Hoare triples, and in a logic where the only atomic specifications are the triple and $\top, \bot$, then any S that satisfies $\mathsf{frame}_+(S)$ or $\mathsf{frame}_-(S)$ is equivalent to either $\top$ or $\bot$.

7.2 Specification-level existential rule

The *existential rule* in Hoare logic allows moving an existential quantifier from the precondition of a Hoare triple out into the logical variable context. Just as we have generalised the frame and consequence rules, we can generalise the existential rule to work with other specifications than triples. Using the same approach as in Section 7.1, we define

$$\mathsf{frame}_\exists(S) \triangleq \forall P.\ \big((\forall x.\ S \otimes P(x)) \vdash S \otimes (\exists x.\ P(x))\big)$$

We can then state the specification-level existential rule as

$$\frac{\mathsf{frame}_\exists(S) \quad S' \vdash \forall x.\ S \otimes P(x)}{S' \vdash S \otimes (\exists x.\ P(x))}$$

The following rules, using the notation introduced in Section 7.1, let us schematically prove $\mathsf{frame}_\exists$.

$$\begin{array}{r} \mathsf{safe} : \exists \\ \triangleright, \otimes, \oslash, \forall : \exists \longrightarrow \exists \\ \wedge : (\exists, \exists) \longrightarrow \exists \\ \Rightarrow : (-, \exists) \longrightarrow \exists \end{array}$$

The natural converse of $\mathsf{frame}_\exists(S)$, with the entailment in the other direction, is equivalent to $\mathsf{frame}_-(S)$. This gives an intuitive justification of the rule for implication above.

Example 15. For all P, c, Q, we have $\mathsf{frame}_\exists(\{P\}\ c\ \{Q\})$. This is seen by unfolding the definition of the triple and applying the above rules. ◇

7.3 Further properties of read-only frame connective

The read-only frame and the frame connectives are interchangeable in certain cases:

1. For singleton frames: $S \oslash \{\sigma\} \equiv S \otimes \{\sigma\}$.
2. If $\mathsf{frame}_{\exists}(S)$ then $S \oslash R \vdash S \otimes R$.
3. If $\mathsf{frame}_{\_}(S)$ then $S \otimes R \vdash S \oslash R$.

Whereas two adjacent frame connectives can always be merged and split by the $\otimes$-$*$ rule, this is not always possible for the read-only frame connective:

1. If $\mathsf{frame}_{\exists}(S)$ then $S \oslash (R * R') \vdash S \oslash R \oslash R'$.
2. Unconditionally, $S \oslash R \oslash R' \vdash S \oslash (R * R')$.

8. Related work

This paper builds on previous work on higher-order frame rules, assembly-language verification and guarded recursion.

Separation logic for assembly code. Our work shares many goals with the work of Myreen et al. [28, 29]. They have built a separation logic for subsets of ARM and x86 in the HOL4 proof assistant. Their logic emphasises total correctness, but since assembly-language programs do not terminate, total correctness does not mean guaranteed termination as it usually would. Instead, a post-condition Q means that execution will eventually reach a machine state in Q. This makes specifications much more intensional than in our case, preventing, for example, relocating and patching (or interpreting) code in memory in an externally-unobservable way unless this has been somehow explicitly allowed by the specification.

The logic of Myreen et al. lacks labels in assembly programs, relying instead on explicit instruction address arithmetic. Their entire specification logic takes place in a generalised Hoare triple with multiple pre- and postconditions and offset transformer functions. This is general enough to support jumps, function calls and self-modifying code, but it remains a triple and is thus restricted to what can be expressed with preconditions, postconditions and code blocks.

The CAP family of logics from Shao et al. are also Hoare logics for low-level code, all verified in Coq. The family includes XCAP [31], GCAP [14], SCAP and ISCAP [40]. Unfortunately, neither of them is a generalisation of any of the others, so each has its strengths and weaknesses. All except GCAP and SCAP have high-level heap manipulation commands such as allocation or function calls built into the machine semantics.

All except GCAP have the program residing in a map from labels to instruction sequences, which is a high level of abstraction and cannot support treating code as data. As Myreen [28] and hopefully this paper have shown, it is not difficult to treat code as data and support function pointers if the logic is fundamentally set up for it. In contrast, GCAP supports it with some awkwardness by attempting to impose the map-from-labels abstraction on top of what is actually happening in the machine.

Affeldt et al [1] have formalized in Coq a separation logic for first-order MIPS assembly code, extending a simpler logic due to Saabas and Uustalu [36], and applied it to verifying provable security of implementations of cryptographic primitives.

Chlipala's Bedrock project [16] is also a Coq framework for verifying low-level code with separation logic. Like our framework, Bedrock has 'while' and 'if' macros and associated proof principles for common patterns of structured code. Chlipala emphasises automated verification, and the program logic is therefore not very expressive. There is no frame rule, so frames are instead passed around explicitly in procedure specifications. Chlipala explains the problem with defining a frame rule for programs with unstructured jumps; here we have demonstrated how this may be solved.

None of the logics discussed above feature a higher-order frame rule.

Higher-order frame rules. The frame rule was extended by O'Hearn et al. [32] to the *hypothetical frame rule*, which allowed framing invariants onto a context of procedure specifications in addition to the triple under consideration. This allowed greater modularity in separate verification of caller and callee, but it still required programs to have structured control flow.

The higher-order frame rule was proposed by Birkedal et al. and used in a separation-logic type system for a programming language with higher-order functions and ground store [8–10]. It has later been extended to languages with higher-order store and used by Krishnaswami [23] and by Pottier [33]. In all cases, it has been for high-level functional programming languages, whereas we have applied it to machine code. We believe we are the first to complement the higher-order frame rule with a higher-order rule of consequence and a higher-order existential rule (Sections 7.1 and 7.2).

Typed assembly language. The work of Appel et al. on typed assembly language and foundational proof-carrying code has demonstrated that step indexing [2] is a viable technique for describing the behaviour of low-level programs. The 'Very Modal Model' paper [3] popularised Nakano's *later* operator, which we also use here, and demonstrated its applicability to assembly code.

The work on typed assembly language focuses on safety of reference types coming from high-level languages and does not attempt to verify code for full functional correctness as we do.

9. Future work

The logic described in this paper will form the foundation for broader research on language-based security in verified systems software.

Although the focus of this paper is on the general design of a separation logic for machine code, and is thus largely parametric in the underlying machine model, one's confidence in the real world validity of verifications in the logic is undeniably limited by one's confidence in the accuracy of that model. Our x86 model was hand-constructed from reading the Intel manuals and, although small programs extracted from Coq seem to run as expected on real hardware, has not been subject to any systematic testing or verification. Indeed, there is one aspect of the Intel specification that we knowingly do not currently model, namely the presence of a code cache: instructions written to memory are, on post-Pentium processors, not guaranteed to be picked up by subsequent execution until a jump or other synchronizing instruction has occurred. We plan to treat the code cache following the approach of Myreen [28], which we expect to be unproblematic. More generally, however, we would like to work with a more trustworthy machine model; these have previously been obtained by extensive testing [19, 26] and semi-automated extraction from the text of reference manuals [11].

An important feature currently missing from our machine model is I/O. By adding this we would incorporate *observations* beyond the simple notion of 'safe' execution, but we believe that our framework is generic enough that the safe specification can be generalized to safety properties involving observable input and output transitions. We have not so far given any serious thought to how one might also prove liveness properties in a comparably extensional way, though that is clearly an interesting subject for future work. It would also be useful to extend our logic to deal with binary relations, rather than unary predicates, on machine states. Such an extension would allow us to verify information flow and abstraction properties [7].

We have already begun to experiment with verified compilation, building a tiny imperative language, its compiler, program logic and proof of correctness, all within Coq. It is straightforward to mix machine code with higher-level languages, as our logic provides a common framework for specifying their interaction at a suitably high level. We plan to develop a number of domain-specific 'little languages' within the same framework.

Low-level code often makes sophisticated use of low-level data structures whose 'ownership' properties cannot easily be captured by the default model of separation described here. We might instead employ 'fictional separation logic' [22]; it is interesting to note that even our use of partial states Σ to describe the machine state $\mathbb{S}$ in a more fine-grained way is reminiscent of fictional separation.

Acknowledgements. We would like to thank Lars Birkedal and Kasper Svendsen for many discussions on higher-order frame rules and their applications.

References

[1] R. Affeldt, D. Nowak, and K. Yamada. Certifying assembly with formal security proofs: the case of BBS. *Sci. Comput. Prog.*, 77(10-11), 2012.

[2] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 2001.

[3] 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.

[4] J. Bengtson, J. B. Jensen, and L. Birkedal. Charge! – a framework for higher-order separation logic in Coq. In *Proc. of ITP*, 2012.

[5] N. Benton. A typed, compositional logic for a stack-based abstract machine. In *APLAS*, volume 3780 of *LNCS*, 2005.

[6] N. Benton. Abstracting allocation: The new new thing. In *Computer Science Logic (CSL 2006)*, volume 4207 of *LNCS*, 2006.

[7] N. Benton and N. Tabareau. Compiling functional types to relational specifications for low level imperative code. In *TLDI*, 2009.

[8] L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules. In *Proc. of LICS*, 2005.

[9] L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. *Logical Methods in Computer Science*, 2006.

[10] L. Birkedal and H. Yang. Relational parametricity and separation logic. *Logical Methods in Computer Science*, 2008.

[11] F. Blanqui, C. Helmstetter, V. Joloboff, J.-F. Monin, and X. Shi. Designing a CPU model: from a pseudo-formal document to fast code. In *3rd Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO 2011)*, 2011.

[12] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In *Proceedings of POPL*, 2005.

[13] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7, 1972.

[14] H. Cai, Z. Shao, and A. Vaynberg. Certified self-modifying code. In *Proc. of PLDI*, 2007.

[15] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *Proc. of LICS*, 2007.

[16] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proc. of PLDI*, 2011.

[17] A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, to appear.

[18] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proc. of Symposia in Applied Mathematics*, Providence, Rhode Island, 1967. AMS.

[19] A. C. J. Fox and M. O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *1st International Conference on Interactive Theorem Proving (ITP 2010)*, volume 6172 of *LNCS*, 2010.

[20] G. Gonthier, A. Mahboubi, and E. Tassi. A small scale reflection extension for the Coq system. Technical Report 6455, INRIA, 2011.

[21] C. Hawblitzel and E. Petrank. Automated verification of practical garbage collectors. In *POPL*, 2009.

[22] J. B. Jensen and L. Birkedal. Fictional separation logic. In *Proc. of ESOP*, volume 7211 of *LNCS*. Springer, 2012.

[23] N. R. Krishnaswami. *Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic*. PhD thesis, Carnegie Mellon University, 2012.

[24] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Mathematical Aspects of Computer Science*, volume 19 of *Proc. of Symposia in Applied Mathematics*. AMS, 1967.

[25] J. Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5, 1989.

[26] G. Morrisett, G. Tan, J. Tassarotti, J.B. Tristan, and E. Gan. Rocksalt: Better, faster, stronger SFI for the x86. In *33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2012)*. ACM, 2012.

[27] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3), 1999.

[28] M. O. Myreen. Verified just-in-time compiler on x86. In *Proc. of POPL*, 2010.

[29] M. O. Myreen and M. J. C. Gordon. Hoare logic for realistically modelled machine code. In *Proc. of TACAS*, 2007.

[30] H. Nakano. A modality for recursion. In *Proc. of LICS*, 2000.

[31] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *Proc. of POPL*, 2006.

[32] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proc. of POPL*, 2004.

[33] F. Pottier. Hiding local state in direct style: a higher-order anti-frame rule. In *Proc. of LICS*, 2008.

[34] J. C. Reynolds. An introduction to specification logic. In *Logics of Programs*, 1983.

[35] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of LICS*, 2002.

[36] A. Saabas and T. Uustalu. A compositional natural semantics and Hoare logic for low-level languages. *Theor. Comput. Sci.*, 373(3), 2007.

[37] M. Sozeau and N. Oury. First-class type classes. In *Proc. of TPHOLs*, 2008.

[38] G. Tan and A. W. Appel. A compositional logic for control flow. In *7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2006)*, 2006.

[39] A. M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, 1949.

[40] X. Jiang W. Wang, Z. Shao and Y. Guo. A simple model for certifying assembly programs with first-class function pointers. In *Proc. of TASE*, 2011.

[41] W. D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5, 1989.

Techniques for model construction in separation logic

Jonas B. Jensen

September 2013

Contents

1 Introduction

Separation logic has been very successful at giving concise specifications and short proofs to pointer-manipulating programs. Unfortunately, the term *separation logic* covers a whole zoo of different theories. Almost every publication that applies separation logic starts by defining a new logic that is general enough to attack the problem at hand but not so general that it disturbs the presentation with orthogonal features.

This proliferation of theories has been identified as a problem several times [BBTS05, Par10], but that has not stopped the flow of new logics being created. Given that no logic proposed so far is a generalisation of all others, we should welcome more exploration and diversity for the foreseeable future.

For new researchers in the field, the best sources for building an intuitive as well as formal understanding of separation logic are still the original articles from 2001-2002 [Rey02, IO01]. Unfortunately, this leads many researchers to ignore some of the recent advances that lead to more general theories with simpler and more concise presentation and metatheory.

This chapter attempts to summarise those advances in the hope that it will allow new research on separation logic to start from the cutting edge as of 2013 instead of 2002. The reader is assumed to have some familiarity with separation logic. A good place to start would be the 2002 introduction by Reynolds [Rey02] or the 2012 tutorial by O'Hearn [O'H12]. More in-depth theoretical discussions are found in [IO01] and [Rey11], although these are still introductory texts.

The level of formal detail in this chapter will be quite high since it aims to be useful for readers trying to encode separation logic in a proof assistant or building a stand-alone tool.

1.1 Overview

The general recipe for making a separation logic, and the structure of this chapter, is as follows.

1. Choose a **programming language**. Small variations in the definition and semantics of the language can have great consequences for the ease of building a separation logic for it.

2. Design the **assertion logic**; i.e., the formulas that describe machine state. It typically includes separating conjunction and points-to formulas. Assertion-logic formulas are typically modelled as sets of machine state subject to some instrumentation and/or side conditions.

3. Design the **specification logic**; i.e., the formulas that describe computations. It typically includes Hoare triples, which refer to assertions

in their pre- and postconditions. In some higher-order settings, the assertion logic may also refer to specifications, in which case the definitions of these two logics may have to be mutually recursive or may coincide.

There is also an optional 0'th step: choose a **metalogic**; i.e., a mathematical framework in which to formulate the theory. The metalogic is usually implicitly chosen as "standard math as taught in school", but recent examples have shown that some separation logics become both simpler and more general when embedded in a metalogic that provides, for example, bound names [Pit01], dependent types [KBJD13] or step-indexing [SB13a, BMSS12].

1.2 What is a separation logic?

To limit the scope of this text, we impose some minimum requirements on what is considered a separation logic. We will say it must contain:

1. An logic of **assertions** P, Q, R that is a complete BI algebra; i.e., it satisfies the rules in Figure 2 (page 125) and Figure 4 (page 129).
2. A logic of **specifications** S that is a complete Heyting algebra; i.e., it satisfies the rules in Figure 2.
3. A **Hoare-triple** specification of the form $\{P\}\ c\ \{Q\}$ or a generalisation thereof. The Hoare triple must satisfy

$$\frac{P \vdash P' \quad S \vdash \{P'\}\ c\ \{Q'\} \quad Q' \vdash Q}{S \vdash \{P\}\ c\ \{Q\}}\ \textsc{Consequence}$$

$$\frac{S \vdash \forall x.\ \{P(x)\}\ c\ \{Q\}}{S \vdash \{\exists x.\ P(x)\}\ c\ \{Q\}}\ \textsc{Exists} \qquad \frac{S \vdash \{P\}\ c\ \{Q\}}{S \vdash \{P * R\}\ c\ \{Q * R\}}\ \textsc{Frame}$$

These requirements are somewhat imprecise, but they have to be, since they apply to logics that have yet to be invented.

The inference rules required of the triple ensure that it admits the *narration* style of writing a Hoare-logic proof [Rey11, WDP13], where commands are interleaved with assertions. Figure 1 shows an example proof narration that uses all three rules.

1.3 Contributions

The main contribution of this chapter is to survey a portion of the first twelve years of separation-logic literature in a common mathematical framework. There is too much published literature to mention it all, so the focus is on techniques that are needed everywhere rather than the very latest developments in specific areas such as concurrency.

$\{\mathsf{head} \neq \mathit{nil} \wedge \mathit{list}(\mathsf{head})\}$ (Consequence)
$\{\exists n.\ \mathsf{head} \mapsto n * \mathit{list}(n)\}$ (Exists)
$\{\mathsf{head} \mapsto n * \mathit{list}(n)\}$ (Frame)
  $\{\mathsf{head} \mapsto n\}$
    next := [head];
  $\{\mathsf{head} \mapsto n \wedge \mathsf{next} = n\}$
$\{(\mathsf{next} = n \wedge \mathsf{head} \mapsto n) * \mathit{list}(n)\}$ (Consequence)
$\{\mathsf{head} \mapsto \mathsf{next} * \mathit{list}(\mathsf{next})\}$ (Frame)
  $\{\mathsf{head} \mapsto \mathsf{next}\}$
    free(head);
  $\{\mathit{emp}\}$
$\{\mathit{emp} * \mathit{list}(\mathsf{next})\}$ (Consequence)
$\{\mathit{list}(\mathsf{next})\}$

Figure 1: Narration-style proof of a two-line example program.

The chapter follows the same structure as most other articles that define a programming language and build a separation logic for it, but along the way we discuss alternatives and variations at every point. A particularly thorough treatment is given to *separation algebras*, since their theory is highly scattered across the literature and often stated in a much less general form than it could be.

Despite its length, this text is far from being self-contained. The reader is encouraged to follow literature references for more details and discussions and for fully worked-out examples.

The conclusion is that making a full-featured higher-order separation logic is not difficult. With the well-known but underused techniques of *shallow embedding* and *separation algebras*, satisfying the axioms of Figures 2 and 4 can be done by satisfying another set of axioms that is simpler and relates more directly to the memory model of a programming language.

2 Semantics of the programming language

A separation logic is typically formulated in the context of one particular programming language, and it is left as an exercise to the reader to port it to other languages. This is rarely a hurdle, but it certainly makes comparisons more difficult. In those articles that abstract away from the details of the programming language [COY07, BJSB11, DYBG$^+$13], the added generality

is typically considered a central contribution.

The choice of programming language and semantics forms part of the statement of the soundness theorem, so it should be as uncontroversial as possible. It is often so uncontroversial that it is not even written out in full. There are a few choices to make, though, beginning with the style of semantics:

Operational. Some flavour of operational semantics is typically the preferred choice. A typical **big-step** semantics is an inductively-defined predicate of the form $\sigma, c \rightsquigarrow \sigma'$ that holds when command c from state σ may terminate successfully in state σ'. A typical **small-step** semantics is an inductively-defined predicate of the form $\sigma, c \rightsquigarrow \sigma', c'$ that holds when command c may take a single step from state σ, leaving a residual command c' and a modified state σ'. Both types of operational semantics often have an additional form $\sigma, c \rightsquigarrow \mathsf{fail}$ that holds when c from state σ may fail, either eventually (for big-step semantics) or in one step (for small-step semantics).

The word "may" above refers to non-determinism of the semantics, which typically arises from memory allocation and concurrency.

Denotational. It is possible to build a separation logic over a denotational semantics, but this is quite rarely seen. Examples include the work of Varming and Birkedal [VB08], Brookes's soundness proof of concurrent separation logic [Bro07] and Hoare Type Theory [PBNM08].

Axiomatic. An axiomatic semantics can be thought of as a program logic without a soundness proof, which sounds like a bad thing. But if this underlying program logic has already been proved sound elsewhere, then an axiomatic semantics can be an effective shortcut to a simpler metatheory. It also demonstrates that the underlying program logic is sufficiently expressive when a high-level program logic can be layered on top of it.

Examples of axiomatic semantics for separation logics are rare, but we found it very useful in [JB12], where the soundness of *fictional* separation logic is proved relative to the soundness of *standard* separation logic. Another example is [DYGW10], where a high-level program and its specification are translated to a low-level program and a low-level specification.

The rest of this section will discuss *operational* semantics only.

Authors of separation logics are often guilty of defining the programming-language semantics in ways that are perhaps unnatural for the language but make it easier to prove the separation-logic metatheory. It is then understood, either formally or informally, that this **instrumented semantics** is **adequate** with respect to a semantics that would be more natural for the

language; i.e., any specification proved to hold in the instrumented semantics also holds in the original semantics.

For instance, it is typical to instrument the semantics so it is well-defined how commands behave in partial states – i.e., states where any location might be missing. This might be done even when modelling a machine where all memory is present all the time [JBK13, Myr10], or where the type system of the language would ordinarily guarantee that there are no dangling pointers [Par05, BJSB11]. Commands executed from a too small partial state should then fail. When this is defined just right, we can prove:

Safety monotonicity. If command c cannot fail in state σ, then it cannot fail in any extension of σ either.

Frame property. In a big-step semantics, the frame property holds if whenever a configuration (σ_0, c) cannot fail and $(\sigma_0 \cdot \sigma_1), c \rightsquigarrow \sigma'$ then there exists σ'_0 such that $\sigma_0, c \rightsquigarrow \sigma'_0$ and $\sigma' = \sigma'_0 \cdot \sigma_1$. Here, $\cdot$ is composition of disjoint states – see Section 3.3.

When these two properties hold, the frame rule follows easily [YO02].

The properties above surprisingly sensitive to language features. Safety monotonicity will fail if there is a command to query whether some memory location is mapped [YO02]. The frame property will even fail if the memory allocator is deterministic [YO02]. It is possible, though, to have the frame rule without having the frame property – see Section 4.1.1.

2.1 Modelling failure

Failure – i.e., the program crashing – is traditionally a crucial part of separation logic. Hoare triples $\{P\}\ c\ \{Q\}$ are always interpreted to imply that (σ, c) cannot fail when σ satisfies P; thus, there must be a possibility of failure in the semantics, or this property of triples would hold vacuously. As mentioned above, failure can be modelled with a transition $\sigma, c \rightsquigarrow \mathsf{fail}$ in the operational semantics, and it typically happens when dereferencing an invalid pointer.

One must consider whether the semantics permits the distinction between three important program behaviours: **successful termination**, **failure** and **non-termination**. One or both of the latter two behaviours might correspond to the semantics getting **stuck**; i.e., not allowing further reductions. We say that a configuration (σ, c) is stuck[1] when there is no x, not even fail, such that $\sigma, c \rightsquigarrow x$.

In a big-step semantics, stuckness should correspond to (guaranteed) non-termination, and therefore we need the fail value to model actual failure if we want to distinguish the two. But this means that a rule must be added to trigger and propagate failure in every place it might occur. In the version

[1] In a small-step semantics, we might further require $c \neq \mathsf{skip}$.

of the Charge! platform described in [BJSB11], there are about as many rules for failure as there are rules for success, and forgetting to add a failure rule renders the model of the programming language unsound. This is because it makes failure look like non-termination, and a partial-correctness program logic cannot distinguish non-termination from success.

A possible fix for this problem is to not have failure rules but instead a set of coinductive rules for when a configuration (σ, c) may diverge [LG09]. Then failure is defined as a configuration that is stuck but cannot diverge, and omission of rules leads to incompleteness instead of unsoundness.

In a small-step semantics, we can often design a system where stuckness corresponds to failure, avoiding the need for a special fail configuration. This increases confidence in the semantics and cuts the number of rules approximately in half for the reasons mentioned above. It works because stuckness of a subcommand tends to propagate to compound commands: the sequence command crash; c will be stuck after one step because the crash command is stuck. Unfortunately, this is sensitive to language features; in particular, it will not work in a language with a parallel operator: the command crash || loop_forever is never stuck. In such a language, it is therefore necessary to add an explicit fail configuration [Vaf11], leading to the same problems as in a big-step semantics.

It is worth mentioning that the Views framework [DYBG$^+$13] quite elegantly avoids explicit propagation of failure through control-flow commands. It is a small-step semantics, but instead of a failure *configuration* (replacing (σ, c)), there is a failure *state* (replacing σ). See [SB13b] for a generalisation of the approach that also works for procedure calls and fork-parallelism.

2.2 Modelling concurrency

Separation logic for concurrent languages is an important and active area of research, and there is certainly room for further exploration. Important choices to be made in the semantics include the following.

- The basic concurrency primitive can be either a **parallel operator** (c_1 || c_2) or a **fork command** (fork c or fork f for a function name f). The fork command is more similar to real-world programming languages, but the parallel operator is sometimes easier to model. This is because it is often less expressive – in particular, it is often impossible to write a program that executes n commands concurrently, where n is only determined at run time. This becomes possible to do in continuation-passing style if the language has recursive procedures.

  In a language without procedures and where concurrency comes from the parallel operator, it is possible to syntactically see which threads are active at which point in the program. This can be used to simplify

proofs [GBC11], but that technique does not scale directly to realistic languages.

- Communication between threads must also be built into the language at some level. Common solutions include static **locks**, dynamically-allocated locks, a **compare-and-swap** command, and an **atomic** command modifier. Some of these primitives can be derived from each other in a more or less practical way.

- Local (stack) variables can be shared between threads or not. Even though real-world programming languages rarely share mutable local variables between threads, this is often allowed in the semantics of toy languages, and then races must be ruled out at the logic level [O'H07].

  Sharing of mutable local variables becomes more complicated if concurrency comes from a fork command or if they might also be captured in lambda-expressions [SBP10].

- Most separation logics published so far have been for toy languages with **sequentially-consistent** memory, meaning that all threads agree on the value of shared memory at all times. Actual programming languages and multi-core hardware have weaker memory models, and **weak-memory** separation logics have only recently started to emerge [FFS10, WB11, VN13].

3 Assertion logic

Assertions are the formulas P, Q occurring in the pre- and postconditions of Hoare triples $\{P\}\ c\ \{Q\}$. They are essentially predicates on machine state, and the distinguishing feature of separation logic is that they can contain *separating conjunction* and related operators. In this section, we will develop the theory necessary to make all this formal.

3.1 Heyting algebras

The assertion logic must first of all be a *logic*. I choose to define a logic as a **complete Heyting algebra**:

Definition 3.1. A **complete Heyting algebra** is a type equipped with operators $(\top, \bot, \wedge, \vee, \forall, \exists, \Rightarrow)$ and a binary **entailment** relation $\vdash$, satisfying the axioms of Figure 2. Read the horizontal lines in the figure as implication in the metalogic. $\diamond$

$$\frac{}{P \vdash P}\ \vdash\text{-Refl} \qquad \frac{P \vdash Q \quad Q \vdash R}{P \vdash R}\ \vdash\text{-Trans} \qquad \frac{P \vdash Q \quad Q \vdash P}{P = Q}\ \vdash\text{-AnSy}$$

$$\frac{}{P \vdash \top}\ \top\text{-R} \qquad \frac{}{\bot \vdash P}\ \bot\text{-L}$$

$$\frac{P \vdash Q_1 \quad P \vdash Q_2}{P \vdash Q_1 \wedge Q_2}\ \wedge\text{-R} \qquad \frac{P_1 \vdash Q}{P_1 \wedge P_2 \vdash Q}\ \wedge\text{-L1} \qquad \frac{P_2 \vdash Q}{P_1 \wedge P_2 \vdash Q}\ \wedge\text{-L2}$$

$$\frac{P_1 \vdash Q \quad P_2 \vdash Q}{P_1 \vee P_2 \vdash Q}\ \vee\text{-L} \qquad \frac{P \vdash Q_1}{P \vdash Q_1 \vee Q_2}\ \vee\text{-R1} \qquad \frac{P \vdash Q_2}{P \vdash Q_1 \vee Q_2}\ \vee\text{-R2}$$

$$\frac{P \vdash Q \Rightarrow R}{P \wedge Q \vdash R}\ \wedge\text{-Adjoint} \qquad \frac{P \wedge Q \vdash R}{P \vdash Q \Rightarrow R}\ \Rightarrow\text{-Adjoint}$$

$$\frac{\forall x : T.\ (P \vdash Q(x))}{P \vdash \forall x : T.\ Q(x)}\ \forall\text{-R} \qquad \frac{P(t) \vdash Q}{\forall x : T.\ P(x) \vdash Q}\ \forall\text{-L}$$

$$\frac{\forall x : T.\ (P(x) \vdash Q)}{\exists x : T.\ P(x) \vdash Q}\ \exists\text{-L} \qquad \frac{P \vdash Q(t)}{P \vdash \exists x : T.\ Q(x)}\ \exists\text{-R}$$

Figure 2: Axioms of a complete Heyting algebra.

For convenience, we define the following abbreviations.

$\vdash P \triangleq \top \vdash P$	pronounced "P is **valid**"
$P \equiv Q \triangleq P = Q$	but with low precedence, like $\vdash$
$\neg P \triangleq P \Rightarrow \bot$	

The axioms in Figure 2 are one of many equivalent presentations. Like a sequent calculus, most rules are presented as left-rules and right-rules for each operator. However, it is not a standard sequent calculus. Notice the following details.

- There is a single hypothesis on the left of the turnstile rather than a comma-separated list of hypotheses. This is the norm in separation logic because it is otherwise not clear whether the comma would denote ordinary conjunction or separating conjunction. For an alternative that is more suitable for proof theory, see [OP99].
- The rules for implication ($\wedge$-Adjoint and $\Rightarrow$-Adjoint) do not follow the pattern of left-rules and right-rules. They are instead presented as an **adjunction**, or **Galois connection**: for any P, the functor $- \wedge P$ is the left adjoint of $P \Rightarrow -$. This presentation simplifies the proof system when there is only one hypothesis on the left of the

turnstile. It also highlights the fact, coming from the general theory of adjunctions, that the implication operator is uniquely determined by the conjunction operator and vice versa.

- The rules in Figure 2 are written out quite verbosely such that they look like a logic and can be practically applied as such. A more common definition of complete Heyting algebra would characterise entailment $\vdash$ as a partial order with least upper bounds $(\bot, \vee, \exists)$, greatest lower bounds $(\top, \wedge, \forall)$ and an exponential $(\Rightarrow)$.

Most logics in the separation-logic literature are less general than a complete Heyting algebra, either because they are missing some operators or because the domain of quantification is restricted. However, there is rarely a reason for this lack of generality, other than a perceived gain in simplicity. We will see in Section 3.3 how to define an assertion logic such that it is a complete Heyting algebra by construction.

3.1.1 Shallow embedding

Definition 3.1 does not go through the traditional indirection of defining a syntax for formulas and a denotation function from syntactic formulas to semantic assertions. We have only the semantic assertions, and operators such as $\wedge$ are merely infix functions on those.

This approach is sometimes known in the literature as a **shallow embedding** [WN04], "working directly in the semantics" or "the extensional approach" [Nip02]. It is used by the majority of separation-logic formalisations inside proof assistants [AB07, TKN07, CSV07, VB08, McC09, Myr10, BJSB11, JBK13, AM13] since it eliminates a lot of tedious work – the kind of work that tends to be dismissed as "routine" in informal mathematics but cannot be ignored when every detail has to be machine-checked. In particular, a shallow embedding eliminates the need for contexts of **logical variables** and their types, accounts of free logical variables, or capture-avoiding substitutions and notions of fresh names. It does not save us from proving tedious results about **program variables**, though; see Section 3.4.

The quantifiers in Figure 2 are annotated with a domain T, which we sometimes omit when it is clear from the context. Because we have a shallow embedding, T ranges over the types of the metalogic, and the formula under the quantifier is a metalogic function from T to assertions. Notice that the universal quantifiers in the premise of rules $\forall$-R and $\exists$-L belong to the metalogic rather than the complete Heyting algebra. Notice also that it is possible for T itself to be the type of assertions, which means that we have defined a higher-order logic.

The opposite of shallow embedding is a **deep embedding**, where formulas and inference rules are syntactic objects that can be manipulated independently of their semantic **models**, of which there can be many. A

common motivation for deep embeddings is to study the rules of the logic independently from its models. But notice that we can still do this since Definition 3.1 characterises complete Heyting algebras in the abstract, separately from describing any particular such algebra. Shallow embedding is not a new way to study logic, but it is rarely used with separation logic outside proof assistants. Compare this with other mathematical fields, where it is standard, for example, to study group theory independently of particular groups, and nobody would propose to use syntactic formulas for this.

With all this said, it is justifiable to use a deep embedding when the desire is to limit expressiveness of the logic deliberately [Nip02]. This can be used for stating decidability or completeness results or when documenting a software tool that manipulates this logic and thus needs to represent it symbolically.

3.1.2 Injecting metalogic propositions

The quantifiers in a shallow embedding allow us to mention data of arbitrary type in formulas. For this to be useful, we also need to inject propositions from the metalogic that describe this data. In particular, we will need an injection $\langle p \rangle$ of metalogic propositions p into assertions.

The alternative to such an injection would be to recreate the necessary mathematical theories inside the assertion logic; i.e., equality, induction, recursion, etc. While this is certainly possible, it could end up being more work than the separation logic itself.

Fortunately, we can define a $\langle p \rangle$ for any Heyting algebra by exploiting the existential quantifier and metalogic subtyping[2]:

$$\langle p \rangle \triangleq \exists x : \{x : \mathit{unit} \mid p\}.\ \top$$

The injection is covariant with respect to entailment, and it satisfies practical left- and right-rules:

$$\frac{p \Rrightarrow q}{\langle p \rangle \vdash \langle q \rangle}\ \langle\rangle\text{-}\vdash \qquad \frac{p \Rrightarrow (\vdash Q)}{\langle p \rangle \vdash Q}\ \langle\rangle\text{-L} \qquad \frac{q}{P \vdash \langle q \rangle}\ \langle\rangle\text{-R}$$

We now have the theory of equality in our complete Heyting algebra for free, simply by lifting it from the metalogic. For instance, we can prove

$$P(x) \wedge \langle x = y \rangle \vdash P(y)$$

The corresponding entailment in a deep embedding would be written as $P \wedge x = y \vdash P[y/x]$. The deep-embedding concepts of free variables and substitutions are modelled here with function application.

[2] The exact definition will vary depending on the metalogic. The *unit* type can be replaced with any other non-empty type. In Coq, we can simply write $\langle p \rangle \triangleq \exists x : p.\ \top$.

$$\top \triangleq T \qquad P \wedge Q \triangleq P \cap Q \qquad \forall x : T.\ P(x) \triangleq \bigcap_{x:T} P(x)$$
$$\bot \triangleq \emptyset \qquad P \vee Q \triangleq P \cup Q \qquad \exists x : T.\ P(x) \triangleq \bigcup_{x:T} P(x)$$
$$P \vdash Q \triangleq P \subseteq Q \qquad P \Rightarrow Q \triangleq \{t \mid \forall t' \geq t.\ t' \in P \Rightarrow t' \in Q\}$$

Figure 3: Kripke definition of a complete Heyting algebra when assertions are subsets of T, upwards closed under a preorder $\leq$.

3.1.3 Kripke models

Complete Heyting algebras are convenient because they correspond to a familiar notion of logic. They are also convenient because they are easy to construct from a type T and a **preorder** on T; i.e., a binary relation that is reflexive and transitive:

Proposition 3.1. *Given a preordered type $(T, \leq)$, the powerset $\mathcal{P}_{\leq}(T)$ is a complete Heyting algebra, where*

$$\mathcal{P}_{\leq}(T) \triangleq \{P : \mathcal{P}(T) \mid \forall t \in P.\ \forall t' \geq t.\ t' \in P\}$$

and the operators of $\mathcal{P}_{\leq}(T)$ are defined as in Figure 3.

The definitions in Figure 3 are known as a **Kripke model**; the interesting part of it is the definition of implication, which explicitly ensures closure under $\leq$, whereas that closure holds directly for all other operators. The injection from the metalogic to the assertion logic that was discussed in Section 3.1.2 can be defined as $\langle p \rangle = \{t \mid p\}$.

More generally, we can construct a Heyting algebra from an existing one as follows.

Proposition 3.2. *Given a complete Heyting algebra A and a preordered type $(T, \leq)$, the space of monotonic functions $T \to_{\leq} A$ is also a complete Heyting algebra, where*

$$T \to_{\leq} A \triangleq \{f : T \to A \mid \forall t, t'.\ t \leq t' \Rightarrow (f(t) \vdash f(t'))\}$$

and the operators of $T \to_{\leq} A$ are defined in terms of the operators on A:

$$\begin{aligned}
P \oplus Q &\triangleq \lambda t.\ P(t) \oplus Q(t) && \text{for } \oplus \in \{\wedge, \vee\} \\
\mathrm{P} &\triangleq \lambda t.\ \mathrm{P} && \text{for } \mathrm{P} \in \{\top, \bot\} \\
\kappa x.P(x) &\triangleq \lambda t.\ \kappa x.P(x)(t) && \text{for } \kappa \in \{\forall, \exists\} \\
P \Rightarrow Q &\triangleq \lambda t.\ \forall t' \geq t.\ P(t') \Rightarrow Q(t') \\
P \vdash Q &\triangleq \forall t.\ (P(t) \vdash Q(t))
\end{aligned}$$

$$\frac{}{(P * Q) * R \vdash P * (Q * R)}\ *\text{-Assoc} \qquad \frac{}{P * Q \vdash Q * P}\ *\text{-Comm}$$

$$\frac{}{P * emp \equiv P}\ *\text{-}emp \qquad \frac{P \vdash Q}{P * R \vdash Q * R}\ *\text{-}\vdash$$

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

Figure 4: Additional axioms of a BI algebra

Proof. See [BJ], Lemma ILPre_ILogic. □

We will use these constructions to define specification logics, and we will use generalised forms of them to define assertion logics.

Note that the **law of the excluded middle**, i.e. $\vdash P \vee \neg P$ for all P, does not follow from the axioms of a complete Heyting algebra, and it is invalid in the models constructed here unless the preorder is also symmetric; i.e., an equivalence relation.

3.2 BI algebras

Defining complete Heyting algebras only got us half way to separation-logic assertions. We still need an account of the operators that make separation logic special: separating conjunction ($*$), separating implication ($\mathrel{-\!\!*}$), and *emp*. Note that *emp* is sometimes written I in the literature.

Definition 3.2. A **complete BI algebra** [Pym02] is a complete Heyting algebra with additional operators $(*, \mathrel{-\!\!*}, emp)$ satisfying the axioms in Figure 4. The **precedence** of operators used in this text will be, in decreasing order:

$$= \quad * \quad \wedge \quad \vee \quad \mathrel{-\!\!*} \quad \Rightarrow \quad \forall \ \exists \quad \vdash \quad \equiv$$ ◇

The axioms in Figure 4 are intentionally minimal. Rules for associativity and commutativity with $\equiv$ instead of $\vdash$ are derivable. It can also be derived that $*$ is covariant in both arguments; i.e.,

$$\frac{P \vdash P' \quad Q \vdash Q'}{P * Q \vdash P' * Q'}$$

When I required in Section 1.2 that the assertion logic must be a complete BI algebra, it was not only because this gives us the rules that make separation logic intuitive to work with. It is also because it is easy to satisfy these rules. We will see in Section 3.3 how a complete BI algebra arises

naturally from the memory model of a typical programming language or machine.

The operators $-\!\!*$ and $\Rightarrow$ are often omitted from presentations when they are not needed for the examples at hand. But even then, they play an important role in metatheory: they witness that $*$ and $\wedge$ commute with existential quantification as explained in the following proposition.

Proposition 3.3. *In a complete Heyting algebra with an operator $*$ satisfying rule $*$-$\vdash$, it holds that $(\exists x{:}\,T.\ P(x)) * Q \vdash \exists x{:}\,T.\ P(x) * Q$ if and only if there is an operator $-\!\!*$ satisfying the rules* $*$-Adjoint *and* $-\!\!*$-Adjoint.

This proposition is a direct consequence of the *adjoint functor theorem* in category theory, but it is worth seeing the proof written out for the specific case of separation logic. Notice that the left-to-right direction, which is probably the surprising one, is only provable because the assertion logic is higher order, and we can quantify over a type T that represents a set of assertions.

Proof. ($\Leftarrow$) By the following proof tree.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall x.\ (Q(x) * P \vdash Q(x) * P)}\,\vdash\text{-Refl}}{\forall x.\ (Q(x) * P \vdash \exists x.\ Q(x) * P)}\,\exists\text{-R}}{\forall x.\ (Q(x) \vdash P -\!\!* \exists x.\ Q(x) * P)}\,-\!\!*\text{-Adjoint}}{\exists x.\ Q(x) \vdash P -\!\!* \exists x.\ Q(x) * P}\,\exists\text{-L}}{(\exists x.\ Q(x)) * P \vdash \exists x.\ Q(x) * P}\,*\text{-Adjoint}$$

($\Rightarrow$) Define $Q -\!\!* R$ as $\exists P' : \{P' \mid P' * Q \vdash R\}.\ P'$. We first prove $-\!\!*$-Adjoint, which holds without appealing to the assumption we made:

$$\dfrac{\dfrac{\dfrac{}{P \vdash P}\,\vdash\text{-Refl} \qquad \dfrac{P * Q \vdash R}{P : \{P' \mid P' * Q \vdash R\}}\,\text{(metalogic)}}{P \vdash \exists P' : \{P' \mid P' * Q \vdash R\}.\ P'}\,\exists\text{-R with } P}{P \vdash Q -\!\!* R}\,\text{Def.}$$

To prove $*$-Adjoint, we appeal to our assumption under the name $*$-$\exists$.

$$\dfrac{\dfrac{P \vdash Q -\!\!* R}{P * Q \vdash (Q -\!\!* R) * Q}\,*\text{-}\vdash \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall P'.\ (P' * Q \vdash R) \Rightarrow (P' * Q \vdash R)}}{\forall P' : \{P' \mid P' * Q \vdash R\}.\ (P' * Q \vdash R)}\,\text{(metalogic)}}{\exists P' : \{P' \mid P' * Q \vdash R\}.\ P' * Q \vdash R}\,\exists\text{-L}}{(\exists P' : \{P' \mid P' * Q \vdash R\}.\ P') * Q \vdash R}\,*\text{-}\exists}{(Q -\!\!* R) * Q \vdash R}\,\text{Def.}}{P * Q \vdash R}\,\vdash\text{-Trans}$$

□

By analogy with Proposition 3.3, we can also show that the $\Rightarrow$-operator exists if and only if the $\wedge$-operator commutes with existentials.

3.2.1 Classical and intuitionistic logics

There is an important special case of BI that is relevant for separation logic: **Boolean BI** (**BBI**). BBI is obtained by adding the **law of the excluded middle** ($\vdash P \lor \neg P$ for all P) to the axioms of Figure 2, which makes Figure 2 describe a **complete Boolean algebra** – a special case of a complete Heyting algebra. Adding the axioms of Figure 4 as well, one obtains a **complete BBI algebra**.

Another important dialect is **affine BI** [GMP05]. This is obtained by adding **weakening of** $*$ to BI, meaning that $P * Q \vdash P$, or equivalently, $emp \equiv \top$. It is the preferred way to define a separation logic for a garbage-collected language, where it lets us "logically forget" the resource Q with the expectation that it will be garbage-collected some time after (or even before! [Rey00, HDV11]) logically forgetting it.

In the separation-logic literature [IO01], BBI is typically known as **classical separation logic**, and affine BI is known as **intuitionistic separation logic**. This is unlike the more established terminology from philosophical logic, where propositions that are provable in intuitionistic logic are also expected to be provable in classical logic. To add to the confusion, there also exists *classical BI* [BC10], which is something else entirely. To avoid these clashes of terminology, I will prefer the terms *Boolean* and *affine* over *classical* and *intuitionistic* in the remainder of this chapter.

A BI algebra that is both Boolean and affine collapses in the sense that $P * Q \equiv P \land Q$ [BK10]. On the other hand, as we will see in Section 3.3.4, there exist useful separation logics that are neither Boolean nor affine.

3.2.2 Design freedom

It might seem like there is a great deal of freedom when designing a logic that satisfies the axioms in Figures 2 and 4, but many operators are uniquely determined by others. In fact, all operators of a complete BI algebra are uniquely determined when $\vdash$ and $*$ are chosen.

The left- and right-rules of Figure 2 uniquely identify the operators $(\forall, \exists, \land, \lor, \top, \bot)$. For example, if another operator $\barwedge$ satisfies the axioms for $\land$ in Figure 2, it follows that $P \barwedge Q \equiv P \land Q$ for all P and Q.

It follows from Figure 4 that $(*, emp)$ is a monoid, so its unit emp is unique. This means that once the operator $*$ is defined, there is no freedom left to choose the operator emp.

Similarly, the proof rules for $-\!\!*$ and $\Rightarrow$ essentially say that $(P -\!\!* -)$ is the right adjoint of $(- * P)$ and that $(P \Rightarrow -)$ is the right adjoint of $(- \land P)$. Since adjoints are unique, there is no freedom in choosing the operators $(-\!\!*, \Rightarrow)$ once $(*, \land)$ have been defined. In fact, we can define $-\!\!*$ and $\Rightarrow$ in terms of other operators:

Proposition 3.4. *In a complete BI algebra asn, for all* $Q, R : asn$,

1. $Q \mathrel{-\!\!*} R \equiv \exists P : \{P \mid P * Q \vdash R\}.\ P$
2. $Q \Rightarrow R \equiv \exists P : \{P \mid P \wedge Q \vdash R\}.\ P$

Proof (of 1.). Define $Q \mathrel{-\!\!*_\exists} R \triangleq \exists P : \{P \mid P * Q \vdash R\}.\ P$. The proof of Proposition 3.3 shows that $\mathrel{-\!\!*_\exists}$ follows the same two axioms as $\mathrel{-\!\!*}$. It follows that $(P \mathrel{-\!\!*_\exists} Q \vdash P \mathrel{-\!\!*} Q)$ if $((P \mathrel{-\!\!*_\exists} Q) * P \vdash Q)$ if $(P \mathrel{-\!\!*_\exists} Q \vdash P \mathrel{-\!\!*_\exists} Q)$ if true. The converse is similar, which gives the necessary bientailment. □

We can also define $(\wedge, \top)$ and $(\vee, \bot)$ in terms of $\forall$ and $\exists$ respectively:

Proposition 3.5.

1. *Assume P, Q : asn, where asn is a complete Heyting algebra. Let $f_2 : \{true, false\} \to asn$ be the function that maps true to P and false to Q. Then*

$$P \vee Q \equiv \exists b.\ f_2(b)$$
$$P \wedge Q \equiv \forall b.\ f_2(b)$$

2. *Let $f_0 : \emptyset \to asn$ be the unique function with that type. Then, in a complete Heyting algebra,*

$$\bot \equiv \exists x.\ f_0(x)$$
$$\top \equiv \forall x.\ f_0(x)$$

Proof. See the Coq code accompanying [JBK13], Section ILogicEquiv. □

3.3 Separation algebras

3.3.1 Heaps

For a typical toy programming language, the type of heaps is defined as $heap = loc \xrightarrow{\text{fin}} val$, where *loc* is the type of heap **locations** (e.g., the natural numbers), *val* is the type of **values** that can be stored in the heap (e.g., integers and locations), and $\xrightarrow{\text{fin}}$ is the space of partial functions with finite domain. It is convenient to let *loc* be an infinite set and let all *heap*s have finite domain because this guarantees that allocations can always succeed – there are always infinitely many free locations in the heap.

The standard way to build a complete BI algebra from this type is to define a **composition** operation: $(\cdot) : heap \times heap \rightharpoonup heap$. The composition $h_1 \cdot h_2$ is defined when the domains of h_1 and h_2 are disjoint, in which case it takes the union of the two partial functions; i.e.,

$$(h_1 \cdot h_2)(l) = \begin{cases} h_1(l) & \text{if } l \in dom(h_1) \setminus dom(h_2) \\ h_2(l) & \text{if } l \in dom(h_2) \setminus dom(h_1) \\ \text{undefined} & \text{otherwise} \end{cases}$$

The assertion logic can then be defined as $asn = \mathcal{P}(heap)$, which is a complete BBI algebra with the following operators.

$$\exists x : T.\ P(x) = \bigcup_{x:T} P(x) \tag{1}$$

$$\forall x : T.\ P(x) = \bigcap_{x:T} P(x) \tag{2}$$

$$P * Q = \{h_1 \cdot h_2 \mid h_1 \in P \wedge h_2 \in Q\} \tag{3}$$

To make this useful, we define a points-to operator as $l \mapsto v = \{[l \mapsto v]\}$, where $[l \mapsto v]$ is the singleton map that only maps l to v.

Although we could in principle reason directly about heaps and their composition [NVB10], it is typically considered easier to work in terms of the total operator $*$ than the partial operator $\cdot$. The $*$ operator also generalises better than $\cdot$, as we will see in Section 3.3.4.

Affine assertion logic. The assertion logic defined in Equations (1–3) is Boolean in the sense described in Section 3.2.1. More generally, for any set X, its powerset $\mathcal{P}(X)$ is a complete Boolean algebra with the quantifiers defined as in (1) and (2). To build an affine logic instead of a Boolean one, we define assertions to be closed under the **extension ordering** of heaps, defined as $h \sqsubseteq h'$ when h' has all the same mappings as h (and possibly more).

$$h \sqsubseteq h' \triangleq \exists h_0.\ h' = h \cdot h_0$$
$$asn = \mathcal{P}_{\sqsubseteq}(heap)$$

With the same definitions of $(\exists, \forall, *)$ as in Equations (1–3), this forms a complete affine BI algebra [IO01].

3.3.2 Motivations for generalisation

Other programming languages might define *heap* differently. For example, an object-oriented language [BJSB11] might have $heap = loc \times field \xrightarrow{\text{fin}} val$, where *field* is the set of field names; i.e., strings. A machine language for a 32-bit machine [JBK13] might have $heap = [0..2^{32}) \xrightarrow{\text{fin}} [0..2^8)$.

Furthermore, memory models are often **instrumented**, either at the level of operational semantics or at the logical level. An important example of such instrumentation is **fractional permissions** [BCOP05, Boy03], where a heap cell contains not only a value but also a rational number in $perm = \{q : \mathbb{Q} \mid 0 < q \leq 1\}$, where permission 1 is a read-write permission, and any smaller number is a read-only permission. Then we write $heap = loc \xrightarrow{\text{fin}} val \times perm$ and let heap composition $(\cdot)$ be defined even when there is overlap in the heap domains as long as the overlapping locations agree on values, and their permissions sum up to at most 1.

Very elaborately-instrumented heaps can be found in the work on **concurrent abstract predicates** [DYDG+10, SBP13, SB13a]. These "heaps" can contain fractions, named regions, relations on (simpler) heaps, state machines, step-indexes [AM01] and ghost state.

There is thus clearly a need to construct complete BI algebras from the powersets of very elaborate structures. The good news is that there is a theory for doing exactly that in two steps: first, prove that the structure in question is a **separation algebra**. Then invoke a theorem that says that the powerset of any separation algebra, possibly closed under a suitable preorder, is a complete BI algebra. Proving that something is a separation algebra is often a very syntax-directed activity, so this approach greatly reduces the amount of work to be carried out when building an assertion logic.

We will first look at how to build a complete BI algebra from a separation algebra, and then we will look at how to build a separation algebra.

3.3.3 Definitions of separation algebra

The claim from Section 3.3.1 that the powerset of $loc \xrightarrow{\text{fin}} val$ is a complete BI algebra can be proved entirely based on the abstract properties of the composition operator $(\cdot)$: it forms a **partial commutative monoid**. Being a monoid means that composition is associative, which lifts to powersets and lets us prove associativity of the $*$ operator as we defined it in Equation (3). It also means that there is a unit element 0, and we can define $emp = \{0\}$ and prove that it is the unit of $*$. Commutativity lifts to powersets as well. The remaining rules in Figure 4 follow from (3) without appealing to the monoid properties.

The term **separation algebra** is used in the literature to describe structures of this kind. Unfortunately, there is little agreement on what a separation algebra is precisely. Some authors [JB12, GBC11] define it as a partial commutative monoid $(\Sigma, \cdot, 0)$ with a carrier set Σ, a partial binary operation $(\cdot)$ and a unit 0 for that operation, but the following variations, and more, exist.

- The original definition [COY07] required **cancellativity**, meaning that if $a_1 \cdot a$ and $a_2 \cdot a$ are defined and equal, then $a_1 = a_2$. This property is not important for constructing a complete BI algebra, but it can be important for validating the **conjunction rule**, which says that $\{P\}\ c\ \{Q_1\}$ and $\{P\}\ c\ \{Q_2\}$ implies $\{P\}\ c\ \{Q_1 \wedge Q_2\}$ [JB11, GBC11]. It is still common for definitions of separation algebras to include cancellativity [Tue09, DHA09, BJSB11, BK10], but it is less common that they make active use of that axiom.

- Newer definitions [DHA09, BK10, DYBG+13] allow **multiple units**, where the intuition is that every element is associated with exactly one

unit, but there does not have to be one unit that is compatible with every element. This effectively partitions the separation algebra into equivalence classes; one for each unit. This situation arises naturally when taking the disjoint union of two single-unit separation algebras – then there will be two units [DHA09].

- Pottier [Pot13] does not require there to be units but instead requires that there is a **core** for every element. Ordering elements by the extension order, the core of a is meant to be the largest **duplicable** element less than a, where an element is duplicable if it composes with itself to yield itself. This definition does not generalise partial commutative monoids; it is something different.

- Working with a partial monoid can be awkward. Asserting definedness of composition can be overly verbose when done explicitly [NVB10] and potentially ambiguous when done implicitly [JB12]. Some authors have addressed this by requiring a **total** (i.e., ordinary) commutative monoid and adding an absorbing element to represent undefinedness, either always [GMP05] or when necessary [KTDG12].

- Other authors have gone in the opposite direction [GMP05, GLW06, Pot13] and generalised the composition to have type $\Sigma \times \Sigma \to \mathcal{P}(\Sigma)$. This is called a **non-deterministic monoid** or, in the equivalent presentation of a composition with type $\mathcal{P}(\Sigma \times \Sigma \times \Sigma)$, a **relational monoid**.

- Dockins et al. [DHA09] proposed several more axioms that limit the class of separation algebras to those that resemble heaps in various senses.

In very recent work, Brotherston and Villard [BV13] propose a definition that generalises all of the above, except possibly Pottier's "core" concept:

Definition 3.3. A **separation algebra** is a triple $(\Sigma, \cdot, U)$ where $(\cdot) : \Sigma \times \Sigma \to \mathcal{P}(\Sigma)$ and $U \subseteq \Sigma$, and the following holds

1. Commutativity: $a \in a_1 \cdot a_2 \Rightarrow a \in a_2 \cdot a_1$
2. Assoc.: $a_{12} \in a_1 \cdot a_2 \wedge a_{123} \in a_{12} \cdot a_3 \Rightarrow \exists a_{23} \in a_2 \cdot a_3.\ a_{123} \in a_1 \cdot a_{23}$
3. Existence of unit: $\forall a.\ \exists u \in U.\ a \in u \cdot a$
4. Minimality of unit: $u \in U \wedge a' \in u \cdot a \Rightarrow a = a'$ $\diamond$

This definition is identical to the one in the Views framework [DYBG+13] except that composition here is non-deterministic rather than partial. We will use this definition in the rest of this section and show several ways to construct a complete BI algebra from it.

The four axioms of Definition 3.3 may not look like a natural or obvious definition, but consider a lifting of $\cdot$ to sets $A \subseteq \Sigma$:

$$A_1 \cdot A_2 \triangleq \{a \mid \exists a_1 \in A_1.\ \exists a_2 \in A_2.\ a \in a_1 \cdot a_2\}$$

The axioms of Definition 3.3 are then equivalent to this lifted $\cdot$ being commutative and associative with unit U.

3.3.4 Upwards-closed assertions

All assertion logics I have encountered in the literature are essentially modelled as the powerset of some separation algebra Σ, upwards closed under some preorder $\leq$. That is, assertions are of type

$$\mathcal{P}_{\leq}(\Sigma) \triangleq \{P : \mathcal{P}(\Sigma) \mid \forall a \in P.\ \forall a' \geq a.\ a' \in P\}$$

The Heyting part of the logic is then a standard Kripke semantics as defined in Figure 3 (page 128).

Typical choices of the preorder are

- Equality ($=$), in which case $\mathcal{P}_{\leq} = \mathcal{P}(\Sigma)$, and the law of the excluded middle holds in the logic.
- Some other equivalence relation ($\equiv$), in which case the law of the excluded middle also holds.
- The **extension ordering** on the separation algebra ($\sqsubseteq$). We encountered the extension ordering for heaps in Section 3.3.1, and it can be generalised to arbitrary separation algebras as

  $$a \sqsubseteq a' \triangleq \exists a_0.\ a' \in a \cdot a_0$$

  As we will see below, this leads to an affine assertion logic.
- An **interference relation** [DYDG$^+$10, DYBG$^+$13] that describes how other threads may modify the state described by assertions. This enables local reasoning in a concurrent setting, at the cost of precision of the assertions.

  As a simple example [SBP13], a memory location could be tagged as containing a **monotonic counter**, which can be increased or read by any thread at any time. The interference relation is then chosen to allow for such counters to go up but not down, which means that assertions can only express that the counter has *at least* value n but not *exactly* value n.

  Another example is to let $\leq$ model the actions of a garbage collector, such as deallocating and moving objects in memory [HDV11]. Assertions closed under such a relation would be guaranteed immune to garbage collection.

While the Heyting part of $\mathcal{P}_{\leq}(\Sigma)$ is always as in Figure 3, there are at least two different ways to obtain the BI part, depending on how the separation algebra interacts with the preorder. The first is adapted from [DYBG+13]:

Proposition 3.6. *If $(\Sigma, \cdot, U)$ is a separation algebra and $\leq$ is a preorder on Σ satisfying the following two conditions*

1. *The unit set is closed under the preorder; i.e., $\forall u \in U.\, \forall a \geq u.\, a \in U$.*
2. *$\forall a_1, a_2.\ \forall a \in a_1 \cdot a_2.\ \forall a' \geq a.\ \exists a'_1 \geq a_1.\ \exists a'_2 \geq a_2.\ a' \in a'_1 \cdot a'_2$; intuitively, the operands of $\cdot$ can be transported upwards along $\leq$ to follow the result.*

then a complete BI algebra is formed by $\mathcal{P}_{\leq}(\Sigma)$ with the operators defined as in Figure 3 and

$$\begin{aligned} emp &= U \\ P * Q &= \{a \mid \exists a_1 \in P.\, \exists a_2 \in Q.\, a \in a_1 \cdot a_2\} \\ P \mathrel{-\!\!*} Q &= \{a_2 \mid \forall a'_2 \geq a_2.\, \forall a_1 \in P.\, a_1 \cdot a'_2 \subseteq Q\} \end{aligned}$$

Proof. See [BJ], Section BIViews. That proof is actually of a slightly more general fact, analogous to how Proposition 3.2 generalises Proposition 3.1.□

If the conditions for Proposition 3.6 are not satisfied[3], then the following proposition might apply instead. It is adapted from [GMP02, POY04] and generalised from its original setting of partial commutative monoids to our setting of more general separation algebras.

Proposition 3.7. *If $(\Sigma, \cdot, U)$ is a separation algebra and $\leq$ is a preorder on Σ satisfying the following condition*

1. *$\forall a'_1, a'_2.\ \forall a' \in a'_1 \cdot a'_2.\ \forall a_1 \leq a'_1.\ \forall a_2 \leq a'_2.\ \exists a \leq a'.\ a \in a_1 \cdot a_2$; intuitively, the result of $\cdot$ can be transported downwards along $\leq$ to follow the operands.*

then a complete BI algebra is formed by $\mathcal{P}_{\leq}(\Sigma)$ with the operators defined as in Figure 3 and

$$\begin{aligned} emp &= \{a' \mid \exists u \in U.\, u \leq a'\} \\ P * Q &= \{a' \mid \exists a_1 \in P.\, \exists a_2 \in Q.\, \exists a \in a_1 \cdot a_2.\, a \leq a'\} \\ P \mathrel{-\!\!*} Q &= \{a_2 \mid \forall a_1 \in P.\, a_1 \cdot a_2 \subseteq Q\} \end{aligned}$$

Proof. See [BJ], Section BISepRel. □

[3] for instance, the extension ordering does not satisfy the first condition

The conditions of neither Proposition 3.6 nor Proposition 3.7 generalise the conditions of the other, so perhaps a unifying theorem is still waiting to be discovered. Notice that Proposition 3.6 gives a simple and standard definition of $*$ but a more involved definition of $-\!\!*$, while in Proposition 3.7 it is the other way around.

Proposition 3.7 has the following corollaries for special cases of $\leq$.

Corollary 3.1. *If $(\Sigma, \cdot, U)$ is a separation algebra, then a complete* Boolean *BI algebra is formed by $\mathcal{P}(\Sigma)$ with the operators defined as in Figure 3 and*

$$\begin{aligned} \mathit{emp} &\equiv U \\ P * Q &\equiv \{a \mid \exists a_1 \in P.\ \exists a_2 \in Q.\ a \in a_1 \cdot a_2\} \\ P -\!\!* Q &\equiv \{a_2 \mid \forall a_1 \in P.\ \forall a \in a_1 \cdot a_2.\ a \in Q\} \\ P \Rightarrow Q &\equiv \{a \mid a \in P \Rightarrow a \in Q\} \end{aligned}$$

Corollary 3.2. *If $(\Sigma, \cdot, U)$ is a separation algebra with extension ordering $\sqsubseteq$, then a complete* affine *BI algebra is formed by $\mathcal{P}_{\sqsubseteq}(\Sigma)$ with the operators defined as in Figure 3 and*

$$\begin{aligned} \mathit{emp} &\equiv \top \\ P * Q &\equiv \{a \mid \exists a_1 \in P.\ \exists a_2 \in Q.\ a \in a_1 \cdot a_2\} \\ P -\!\!* Q &\equiv \{a_2 \mid \forall a_1 \in P.\ \forall a \in a_1 \cdot a_2.\ a \in Q\} \end{aligned}$$

In logics modelled over $\mathcal{P}_{\leq}(\Sigma)$, primitive assertions such as points-to can typically be defined in terms of the injection $\cdot^{\uparrow} : \Sigma \to \mathcal{P}_{\leq}(\Sigma)$, defined as $a^{\uparrow} \triangleq \{a' \mid a' \geq a\}$. In words, $a^{\uparrow}$ is the smallest set in $\mathcal{P}_{\leq}(\Sigma)$ that includes a.

3.3.5 Constructions

In recent work on separation logic and related formalisms [JB12, KTDG12, LWN13, DYGW10], each module of the program can have its own separation algebra, so the task of verifying a module includes coming up with a separation algebra suitable for it and checking the conditions in Definition 3.3. While this is already much simpler than proving that something is a complete BI algebra, we can make it even simpler still, because separation algebras are very compositional. The following proposition is adapted from [DHA09] and [JB12].

Proposition 3.8. *Given separation algebras $(\Sigma_1, \cdot_1, U_1)$ and $(\Sigma_2, \cdot_2, U_2)$ and an arbitrary type T,*

1. *The* ***product*** *$\Sigma_1 \times \Sigma_2$ is also a separation algebra with unit $U_1 \times U_2$ and composition $(a_1, a_2) \cdot (b_1, b_2) \triangleq (a_1 \cdot_1 b_1) \times (a_2 \cdot_2 b_2)$.*

2. *The* ***tagged union*** $\Sigma_1 + \Sigma_2 \triangleq (\{1\} \times \Sigma_1) \cup (\{2\} \times \Sigma_2)$ *is also a separation algebra with unit* $U_1 + U_2$ *and composition as the smallest relation satisfying* $(i, a) \cdot (i, b) = \{i\} \times (a \cdot_i b)$ *for* $i \in \{1, 2\}$.

3. *The set* T *can be viewed as a* ***discrete separation algebra*** T_{discr} *if we define the units as* $U \triangleq T$ *and composition as the smallest relation satisfying* $t \cdot t = \{t\}$.

4. *The space* $T \overset{\text{fin}}{\Rightarrow} \Sigma_1$ *of* ***finitely-supported functions*** *is a separation algebra. Being finitely supported means that only a finite number of values from the domain are mapped to non-unit values. The units are the functions mapping everything to some unit, and composition is pointwise:*

$$U \triangleq \{f \mid \forall t.\ f(t) \in U_1\}$$
$$f \cdot g \triangleq \{h \mid \forall t.\ h(t) \in f(t) \cdot_1 g(t)\}$$

Proof. See [BJ] for items 1,2,4. See [DHA09] for item 3. □

The space of finitely-supported functions ($\overset{\text{fin}}{\Rightarrow}$) has good composition properties. In particular, it allows currying, so the set $(A \times B) \overset{\text{fin}}{\Rightarrow} \Sigma$ is isomorphic to the set $A \overset{\text{fin}}{\Rightarrow} (B \overset{\text{fin}}{\Rightarrow} \Sigma)$. This is in contrast to the space of finite partial functions ($\overset{\text{fin}}{\rightharpoonup}$), which does not have this isomorphism [Par05] since the curried form allows distinguishing between the values [] (the empty map) and $[a \mapsto []]$ (a singleton map that maps a to the empty map). We found that proofs of deallocation in fictional separation logic [JB12, JB11] became much simpler when using ($\overset{\text{fin}}{\Rightarrow}$) instead of ($\overset{\text{fin}}{\rightharpoonup}$).

We will in practice need more constructions than those given in Proposition 3.8. In fictional separation logic [JB12], we found it useful to revive the concept of a **permission algebra** [COY07], which is like a separation algebra but without units:

Definition 3.4. A **permission algebra** is a pair $(\Pi, \cdot)$ where $(\cdot) : \Pi \times \Pi \to \mathcal{P}(\Pi)$, and the following holds

1. Commutativity: $a \in a_1 \cdot a_2 \Rightarrow a \in a_2 \cdot a_1$
2. Assoc.: $a_{12} \in a_1 \cdot a_2 \wedge a_{123} \in a_{12} \cdot a_3 \Rightarrow \exists a_{23} \in a_2 \cdot a_3.\ a_{123} \in a_1 \cdot a_{23}$ ⋄

A similar but more restrictive definition is given in [Hob11] and used for the same purpose: to serve as an intermediate structure when composing a separation algebra.

Proposition 3.9.

1. *Permission algebras form products and tagged unions just like separation algebras do.*

2. *A permission algebra Π together with a fresh unit element 0 can be viewed as a separation algebra $(\Pi)_0 \triangleq \Pi \cup \{0\}$ with units $\{0\}$ and composition defined as in Π for non-unit elements and as $0 \cdot a = a \cdot 0 = \{a\}$ in other cases.*

3. *Any set T can be viewed as an **equality permission algebra** $T_=$ if we define composition as the smallest relation satisfying $t \cdot t = \{t\}$.*

4. *Any set T can be viewed as an **empty permission algebra** $T_\emptyset$ if we define composition as $t \cdot t' = \emptyset$.*

With these constructions, we can now redefine the heaps from Section 3.3.1 as $heap \triangleq loc \overset{\text{fin}}{\Rightarrow} (val_\emptyset)_0$. We have described the same separation algebra $(heap, \cdot, [])$ as before, but this time there is nothing further to define or prove. The composition operation and its properties follow syntactically from Propositions 3.8 and 3.9, and the fact that $\mathcal{P}(heap)$ forms a complete BBI algebra follows from Corollary 3.1.

We can also define heaps with permissions [BCOP05, Hob11] for any permission algebra Π as $heap_\Pi \triangleq loc \overset{\text{fin}}{\Rightarrow} (val_= \times \Pi)_0$. For further examples, see [JB12, JB11].

As already mentioned, the study of separation algebras is still at an early stage, and the constructions presented here could soon be superseded by better ones. Not all definitions of separation algebra support all the constructions; in particular, multiple units are needed to support tagged unions and discrete separation algebras [DHA09]. For most of the alternative definitions of separation algebras discussed in Section 3.3.3, none of the constructions have been verified. One exception is [DHA09], which proposes several specialisations of separation algebras and verifies that each one is preserved by all constructions.

3.3.6 Cyclic definitions

Advanced separation logics often feature instrumented heaps that can "store" assertions. Examples of such stored assertions include the invariant associated with a storable lock [GBC+07], the operations allowed on a shared resource [DYDG+10], or the precondition of a procedure stored in memory [NS06].

A representative example of this situation, inspired by models of storable locks, could be the following attempt to define heaps:

$$\Sigma = loc \overset{\text{fin}}{\Rightarrow} ((val \times asn)_\emptyset)_0$$

If $asn = \mathcal{P}_\leq(\Sigma)$ as usual, then this definition becomes cyclic, with Σ in a negative position:

$$\Sigma = loc \overset{\text{fin}}{\Rightarrow} ((val \times \mathcal{P}_\leq(\Sigma))_\emptyset)_0$$

There is no set-theoretic solution to this equation, so it cannot be used as a definition.

A comprehensive treatment of the techniques that apply here is beyond the scope of this text, but the solutions can roughly be grouped into three types, ordered here by increasing expressiveness of the resulting logic.

1. Store a syntactic assertion [VN13, DYDG+10] or token [GBC+07] instead of a semantic assertion. This can work well enough for first-order theories.

2. Use **step-indexing** or similar techniques [AMRV07, DHA09, BRS+11] to **guard** the recursive occurrence. This essentially creates an approximation of the recursively-defined heap up to $n+1$ recursive iterations, exploiting that a program that has only n steps of execution left will not have time to observe what lies beyond that depth in the heap when a heap dereference takes one step. Specification validity then means that the program is valid for arbitrary values of this n.

3. The separation logic can be developed in a metalogic that does not restrict recursive occurrences to being strictly positive in the traditional sense. The **topos of trees** [BMSS12] has recently been proposed for this purpose; it allows negative occurrences as long as they are guarded by a modal operator $\triangleright$. In the model of the topos of trees, this modal operator is explained in terms of step-indexing, so this technique is sound for essentially the same reason as item 2 above.

   See [SB13a] for a recent example of using the topos of trees as the metalogic of an impredicative concurrent separation logic.

Step-indexing in logical propositions, rather than types, are discussed in Section 4.2.2.

3.4 Program variables

We have so far discussed assertions quite abstractly, but ultimately they are of course used in pre-and postconditions of commands, and they must be able to describe the values of **program variables** as named in the source program. The exact technique will necessarily be specific to the programming language, but there are some common patterns and even some reusable theory just like there was for heaps.

Using a shallow embedding gave us typed *logical* variables practically for free, but there is no such shortcut for *program* variables. Fortunately, program variables still tend to be simpler to support than logical variables since program variables tend to have a more restricted binding structure.

When every formal detail has to be right – especially when working in a proof assistant – then there are many pitfalls in the encoding of program

variables. This section surveys the techniques that have been proposed for handling program variables in various programming languages. The goal is to make program variables behave much like logical variables, which is the tradition in Hoare logic, while still retaining all the benefits of a shallow embedding.

The semantics of a programming language tends to divide the state into a heap and a **stack** (i.e., stack frame). Shared mutable data lives on the heap, while the content of local variables lives on the stack. Some authors use the term **store** instead of stack. Stacks in While-like toy programming languages are typically modelled as $\mathit{stack} \triangleq \mathit{var} \to \mathit{val}$. This also suffices for modelling many realistic languages such as Java [PB05, BJSB11] or assembly [CSV07, Myr10, JBK13], where the type *var* is chosen as strings or register names respectively.

Other languages have more complex stacks, where a simple mapping from variables to values does not suffice. This tends to happen when the language enables access to the L-value of local variables, either with an explicit address-of operation as in the C programming language, or implicitly through variable capture [SBP10]. Complications may also arise in concurrent languages, where the stack becomes shared when threads fork. Variable scoping rules in JavaScript is a whole research topic in itself [GMS12].

The rest of this section assumes that (instrumented) machine states can be modelled as $\mathit{stack} \times \mathit{heap}$ for some definition of *heap*. Even separation logics for the C programming language adopt this model and simply disallow access to the address of local variables [AB07, TKN07, JSP12, AM13].

Using the constructions from Section 3.3.5, there are at least two useful ways to turn the whole machine state into a separation algebra.

1. If we let *stack* be a discrete separation algebra, then the product $\mathit{stack}_{\text{discr}} \times \mathit{heap}$ is a separation algebra whose composition is defined by

   $$(s, h) \in (s_1, h_1) \cdot (s_2, h_2) \iff s = s_1 = s_2 \wedge h \in h_1 \cdot h_2$$

   With a standard construction to form a complete BI algebra from $\mathit{stack}_{\text{discr}} \times \mathit{heap}$, such as Corollary 3.1, we obtain the same definition of $*$ as in the vast majority of separation-logic texts:

   $$P * Q \equiv \{(s, h) \mid \exists h_1, h_2.\ h \in h_1 \cdot h_2 \wedge (s, h_1) \in P \wedge (s, h_2) \in Q\}$$

   A drawback of this approach is that it typically requires a syntactic side condition on the frame rule to say that variables free in the frame R must not be modified by the command c:

   $$\frac{\{P\}\ c\ \{Q\} \qquad \mathit{modifies}(c) \cap \mathit{fv}(R) = \emptyset}{\{P * R\}\ c\ \{Q * R\}}$$

A simple way to get rid of this side condition is to make local variables immutable [BTSY06], but this can of course be a major restriction of the programming language.

2. We can alternatively define stacks almost like heaps: $stack = var \xrightarrow{\text{fin}} (val_\emptyset)_0$. This approach is known as **variables as a resource**, and the original paper about this idea [PBC06] goes even further and adds fractional permissions *perm*, defining $stack = var \xrightarrow{\text{fin}} (val_= \times perm)_0$.

   With variables as a resource, we get a more aesthetically-pleasing frame rule because the syntactic side condition essentially becomes integrated into the definition of $*$.

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

   This is also formally better in cases where $\mathit{modifies}(c)$, the set of local variables potentially modified by c, is not easy to determine. This happens in languages where we cannot syntactically see from a program what variables might be modified, or when that over-approximation is too coarse [JBK13, MG07].

   The drawback is that the convenient similarity between program variables and logical variables is lost. Program variables have to be treated like heap locations using some type of points-to predicate, which complicates the rules for variable assignment and conditionals [PBC06] [DYBG+13, Definition 17].

3.4.1 Open terms and lifting

The two constructions above allow us to have program variables in assertions, but that was only half of the problem. We also need program variables in expressions, including logical expressions that are not part of the programming language. For instance, we might like to assert that $\mathsf{n} > \mathsf{m} + 1$, where n and m are written in a sans-serif font because they are program variables; i.e., symbols of type *var*.

If we are using variables as a resource, the example assertion above could be written as

$$\exists m.\ \mathsf{m} \mapsto m * \exists n.\ \mathsf{n} \mapsto n \wedge \langle n > m + 1 \rangle.$$

Notice first the distinction between the variable name m and its value m, which makes the formula somewhat verbose. Syntactic sugar has been proposed to reduce this somewhat [PBC06], but it comes at a price: expected identities such as $e_1 \neq e_2 \equiv \neg(e_1 = e_2)$ fail to hold. On the other hand, the verbosity is not much of a burden in assembly language, where the "program variables" are uninformative register names, and their values tend to be named differently from the registers that hold them [JBK13, MG07].

The rest of this section tries to follow the Hoare-logic tradition of referring to program variables from deep inside expressions. As a first step, our example assertion of $\mathsf{n} > \mathsf{m} + 1$ can be written formally as

$$\{(s, h) \mid s(\mathsf{n}) > s(\mathsf{m}) + 1\},$$

but this is undesirable as it exposes the stack s, which increases verbosity and looks quite different from standard presentations. With some amount of syntactic sugar, it can be made practical, though [McC09].

In Charge! [BJB12, BJ], a Coq formalisation of separation logic, we distinguish between *assertions* and *open assertions*, which I will denote here as

$$\mathit{asn} \triangleq \mathcal{P}(\mathit{heap}) \qquad \text{and} \qquad \mathit{open\ asn} \triangleq \mathit{stack} \to \mathit{asn}$$

respectively. The type *open asn* is a complete BI algebra, and it is in fact isomorphic[4] to $\mathcal{P}(\mathit{stack}_{\mathrm{discr}} \times \mathit{heap})$. In general:

Proposition 3.10. *Given a complete BI algebra A and a preordered type $(T, \leq)$, the space of monotonic functions $T \to_{\leq} A$ is a complete BI algebra, where*

$$T \to_{\leq} A \triangleq \{f : T \to A \mid \forall t, t'.\ t \leq t' \Rightarrow (f(t) \vdash f(t'))\}$$

and the operators of $T \to_{\leq} A$ are defined in terms of the operators on A:

$$\begin{aligned} P * Q &\triangleq \lambda t.\ P(t) * Q(t) \\ \mathit{emp} &\triangleq \lambda t.\ \mathit{emp} \\ P \mathrel{-\!\!*} Q &\triangleq \lambda t.\ \forall t' \geq t.\ P(t') \mathrel{-\!\!*} Q(t') \end{aligned}$$

The Heyting operators are defined as in Proposition 3.2 (page 128).

Proof. See [BJ], Lemma BILPreLogic. □

We can extend the definition of *open asn* to arbitrary types T:

$$\mathit{open}\ T \triangleq \mathit{stack} \to T$$

This allows us to give a uniform treatment of free variables and substitutions on **open terms** regardless of their type. We can **lift** functions to work on open terms, ranged over by o:

Definition 3.5. Given an n-ary function $f : (T_1 \times \cdots \times T_n) \to T$, define $\dot{f} : (\mathit{open}\ T_1 \times \cdots \times \mathit{open}\ T_n) \to \mathit{open}\ T$ as

$$\dot{f}(o_1, \ldots, o_n) \triangleq \lambda s.\ f(o_1(s), \ldots, o_n(s)).$$

We lift constants $t : T$ to $\dot{t} : \mathit{open}\ T$, by taking $n = 0$ above. Finally, we overload the same notation to mean something different for program variables $x : \mathit{var}$, defining $\dot{x} : \mathit{open\ val}$ as $\dot{x} \triangleq \lambda s.\ s(x)$. ◇

[4] They are isomorphic as complete BI algebras, meaning that the isomorphism preserves all operators.

Returning to our example assertion, we may now write it formally as

$$\dot{\mathsf{n}} \mathbin{\dot{>}} \dot{\mathsf{m}} \mathbin{\dot{+}} \dot{1}.$$

We can just as easily lift functions from the metalogic that do not also exist as programming-language expressions; for instance, given $fac : \mathbb{N} \to \mathbb{N}$, we can express that variable n holds the factorial of variable m as

$$\dot{\mathsf{n}} \doteq \dot{fac}(\dot{\mathsf{m}}).$$

As a final and important example, we can lift operators on types that have no representation in the programming language; e.g., *list cons* and recursively-defined *list predicates* [BJB12]. This lets us give a satisfying formal account of how we reason with arbitrary mathematics inside assertions, without implicitly assuming that the theories we need have been reconstructed from scratch within *open asn*.

While the benefits of *open* T presented thus far could be dismissed as being superficial, we found in [BJSB11][5] that the lifting concept was crucial for harnessing the abstraction and modularity benefits of *higher-order* separation logic. It is a standard pattern of specification in higher-order separation logic to quantify and parametrise over assertion-logic predicates [BBTS05, PB08, BJSB11]. This means that formulas tend to involve opaque predicates F, and eventually we will have to ask what are the free variables of, say, $F(e)$. One would hope that $fv(F(e)) \subseteq fv(e)$, but this depends on the definition of F. Examples of "undisciplined" F include

$$F(e) = e \doteq \dot{\mathsf{x}}$$
$$F(e) = e[\mathsf{y}/\mathsf{x}] \doteq \dot{0}$$
$$F(e) = \langle \mathsf{x} \in fv(e) \rangle$$

If there is only one type of assertions, $\mathcal{P}(stack \times heap)$, then it is difficult to statically rule out such undesired values in a shallow embedding. See the discussions in [App06], where side conditions about free variables and substitutions have to be carried around with predicates. In contrast, the lifting approach always gives us well-behaved free variables and substitutions. The following two subsections will define semantic notions of free variables and substitutions such that the following holds:

$$fv(\dot{f}(o_1, \ldots, o_n)) \subseteq fv(o_1) \cup \ldots \cup fv(o_n)$$
$$\dot{f}(o_1, \ldots, o_n)[\bar{e}/\bar{x}] = \dot{f}(o_1[\bar{e}/\bar{x}], \ldots, o_n[\bar{e}/\bar{x}])$$

For further examples and motivation, see [BJSB11, BJB12].

[5] In that paper, *open* T is called *sm* T, and lifting is written $\hat{f}$ instead of $\dot{f}$.

3.4.2 Free variables

Following Appel et al. [AB07], we can characterise free variables semantically by saying that x is free in $o : \mathit{open}\ T$ when a change to x can cause a change to o:

$$\mathit{fv}(o) \triangleq \{x \mid \exists s, v.\ o(s) \neq o(s[x := v])\}$$

An open term can have an infinite number of free variables; for instance, every variable is free in $\forall x : \mathit{var}.\ \dot{x} \doteq \dot{0}$. We might like to forbid such terms, but it can be hard to do without restricting expressiveness, and it turns out we do not need to. Compare this to *nominal logic* [Pit01], which is much more well-behaved. There, open terms have a finite number of free variables and elegant support for binders in the programming language, but the approach requires the entire metalogic to be replaced.

The definition of fv satisfies convenient rules for how it applies to pointwise-lifted functions and to variables:

$$\mathit{fv}(\dot{f}(o_1, \ldots, o_n)) \subseteq \mathit{fv}(o_1) \cup \ldots \cup \mathit{fv}(o_n)$$
$$\mathit{fv}(\dot{x}) \subseteq \{x\}$$

The property on $\dot{f}$ only holds with inclusion, not equality, since a variable may not be semantically free even if it occurs in an expression – for instance, $\mathit{fv}(\dot{\mathsf{x}} \mathbin{\dot{-}} \dot{\mathsf{x}}) = \emptyset$. The property on $\dot{x}$ of course holds with equality for any non-trivial choice of val.

3.4.3 Substitutions

It is standard, both in deep and shallow embeddings, to define a substitution $\rho : \mathit{subst}$ as a function from variables to expressions, which here means

$$\mathit{subst} \triangleq \mathit{var} \to \mathit{open}\ \mathit{val}$$

Expressions, ranged over by e, are of type $\mathit{open}\ \ \mathit{val}$ in their shallowly-embedded form. A simultaneous substitution of n distinct variables can be defined as

$$[e_1, \ldots, e_n \,/\, x_1, \ldots, x_n] \triangleq \lambda x.\ \begin{cases} e_i & \text{if } x = x_i \text{ for some } i \\ \dot{x} & \text{otherwise} \end{cases}$$

We can then define a semantic notion of applying a substitution to an open term. This is written here with postfix notation as per tradition, and it is defined in terms of applying a substitution to a stack.

$$o\rho \triangleq \lambda s.\ o(s\rho), \text{ where}$$
$$s\rho \triangleq \lambda x.\ \rho(x)(s)$$

From these definitions, we can prove as lemmas how substitution acts on pointwise-lifted functions and on variables.

$$(\dot{f}(o_1, \ldots, o_n))\rho = \dot{f}(o_1\rho, \ldots, o_n\rho)$$
$$(\dot{x})\rho = \rho(x)$$

Contrast this to how substitutions work in a deep embedding: the *lemmas* above about how ρ acts on $\dot{f}$ and $\dot{x}$ would be taken as the *definition* of substitution on syntactic terms, and our *definition* of $o\rho$ would instead be proved as a *lemma* [SBP09, Lemma 10] [Kri12, Lemma 28.2.a.ii].

3.4.4 Typed values

Separation logic is often applied to typed programming languages such as Java [Par05, BJSB11] or C [TKN07, AB07]. The typical approach is then to generalise the syntax and operational semantics to remove static types and let the separation logic enforce typing instead – a program logic generalises a simple type system, so it is a burden to have both. It is standard [Rey02, AB07, McC09, TSFC09, SBP10, BJSB11] to do as we have been doing since Section 3.3.1 and define a type *val* as a tagged union of integers, pointers, Booleans and any other types that can be stored in local variables or on the heap.

The question is then whether an arithmetic operator, such as $>$, should have type $\mathit{val} \times \mathit{val} \to \mathit{val}$ or $\mathit{int} \times \mathit{int} \to \mathit{bool}$. In the first approach, we can immediately write logic expressions such as $\dot{\mathsf{x}} \mathbin{\dot{>}} \dot{\mathsf{y}}$, and the types will match up. On the other hand, we need to have an answer for how to compare, say, a pointer with a Boolean, even though such a comparison could never happen in the original, typed, programming language. This issue extends to any other operator we want to lift from the metalogic.

In the other approach, where $(>) : \mathit{int} \times \mathit{int} \to \mathit{bool}$, we cannot write $\dot{\mathsf{x}} \mathbin{\dot{>}} \dot{\mathsf{y}}$, since $\dot{>}$ has type $\mathit{open\ int} \times \mathit{open\ int} \to \mathit{open\ bool}$ while $\dot{\mathsf{x}}$ and $\dot{\mathsf{y}}$ have type $\mathit{open\ val}$. One option is to read variables not as an untyped $\dot{x} : \mathit{open\ val}$ but as a typed $\mathit{intvar}(x) : \mathit{open\ int}$ etc. Then we can write $\mathit{intvar}(x) \mathbin{\dot{>}} \mathit{intvar}(y)$. The problem that a *value* could have an unexpected type is now replaced with the problem that a *variable* can have an unexpected type, and *intvar* will have to return a dummy value, such as 0, if it reads a non-*int*.

We have tried both approaches in Charge! [BJSB11, BJB12], and they both ended up littering specifications and proofs with distracting coercions in and out of *val*.

A third approach is to make expression evaluation partial [PBC06, AB07], but this has its own set of problems; for instance, expected identities such as $e_1 \neq e_2 \equiv \neg(e_1 = e_2)$ no longer hold [PBC06]; here, $(\neq, =)$ are partial expressions, and $\neg$ is from the assertion logic. Despite this, many authors model stacks as partial functions without explaining what happens when lookup fails.

It is worth looking at two separation logics that are not affected by these problems at all, even down to the last formal detail.

- In [JBK13, KBJD13], we define a separation logic for x86 machine code. Assertions are of type $\mathcal{P}(\Sigma)$, where

$$\begin{aligned}\Sigma = {} & (\mathit{register} \xrightarrow{\text{fin}} (\mathit{DWORD}_\emptyset)_0) \times \\ & (\mathit{flag} \xrightarrow{\text{fin}} (\mathit{bool}_\emptyset)_0) \times \\ & (\mathit{DWORD} \xrightarrow{\text{fin}} (\mathit{BYTE}_\emptyset)_0)\end{aligned}$$

  The three components in the product denote CPU registers, CPU flags and main memory respectively; types BYTE and DWORD denote bool^8 and bool^{32} respectively.

  The registers and flags together can be thought of as the local variables – the *stack*, in our current terminology – and this stack can store both DWORD and bool values. The separation-algebra annotations on Σ reveal that we are using the variables-as-a-resource approach, but this is not the essence of why types on the stack work out here. It works because there is a separate name space for the registers and flags; i.e., EAX is a register, and it is clear from its name only that it holds a DWORD and not a bool. This approach can also work in more conventional programming languages [CGZ05].

  The main memory can be thought of as the *heap* in our current terminology. Types on the heap work out for a completely different reason than for the stack. To let us store other things than BYTEs in memory, there is essentially a points-to predicate $\mapsto_T$ for every type T that has a defined decoding from byte sequences to T. Then $l \mapsto_T v$ holds when $v : T$, and the memory contents starting at l decodes to v. See also [TKN07, AM13] for related approaches with slightly different goals.

- Another approach is to not *replace* the type system with a program logic but instead *extend* the type system until it becomes as powerful as a program logic. Programming-language terms with side effects, such as heap write, are given a type describing those effects, such as $\{P\}\{Q\}$, where P and Q can refer to program variables. Logical entailment is encoded as subtyping.

  Examples include [BTSY06, NAMB07, Pot08, KTDG12]. Since this requires either a deep embedding or reverification of the whole metatheory [NMS+08, CMM+09], we lose the advantages gained from having a shallow embedding. Features such as higher kinds and dependent types must be re-created within the type system rather than borrowed from the metalogic. See also the discussion in Section 4.3.2.

The unproblematic logics mentioned above have one thing in common: they do not define a *val* type, but instead they keep all programming-language types explicit and separate.

The problem also seems to go away when using variables as a resource. Recall the example from Section 3.4.1, where we wrote

$$\exists m.\ \mathsf{m} \mapsto m * \exists n.\ \mathsf{n} \mapsto n \wedge \langle n > m + 1 \rangle.$$

In a setting where the injection from *int* to *val* is called *intval*, we can write this assertion more explicitly as

$$\exists m : \mathit{int}.\ \mathsf{m} \mapsto \mathit{intval}(m) * \exists n : \mathit{int}.\ \mathsf{n} \mapsto \mathit{intval}(n) \wedge \langle n > m + 1 \rangle.$$

This solves the problem and can be useful for any type T with an injective function $T \to \mathit{val}$. On the other hand, variables as a resource remains very verbose and does not look like standard Hoare logic.

The above pattern for getting typed program variables using a *stack* version of points-to predicate will work just as well for any standard *heap* points-to predicate. Since separation logic, with very few exceptions [SJP10, PS12], forbids direct heap references in expressions, we are already forced into this pattern of existential quantification and distinction between a heap location and its value, so this may as well be used to get stronger typing.

4 Specifications

The primitive unit of specification in separation logic is usually the **Hoare triple**. In the most basic form, in a shallow embedding, the triple is a predicate in the metalogic. Section 4.1 discusses definitions and inference rules for the triple based on this assumption.

Section 4.2 will demonstrate benefits and techniques for considering the triple instead as a formula of **specification logic** [Rey82], which allows us to give a logical account of the context in which a given triple holds.

4.1 Hoare triples

4.1.1 Definitions

The Hoare triple is where the assertion logic from Section 3 meets the operational semantics from Section 2. The Hoare triple for **partial correctness** is usually defined to mean, intuitively, "for any state satisfying the precondition, no execution from that state will crash, and any *terminating* execution from that state will result in a state satisfying the postcondition". In the most basic form, the triple is defined as

$$\{P\}\ c\ \{Q\}_1 \triangleq \forall \sigma \in P.\ \neg(\sigma, c \rightsquigarrow \mathsf{fail}) \wedge \forall \sigma'.\ \sigma, c \rightsquigarrow \sigma' \Rightarrow \sigma' \in Q$$

For partial correctness, we are content with ignoring divergence, but we will not ignore failure.

Contrast this to **total correctness**, where we additionally require the command to *terminate* from any state satisfying the precondition. If there is a relation $\sigma, c \rightsquigarrow \infty$ meaning that σ, c may diverge, then a basic definition of the Hoare triple for total correctness can be

$$[P]\ c\ [Q]_2 \triangleq \forall \sigma \in P.\ \neg(\sigma, c \rightsquigarrow \mathsf{fail}) \wedge \neg(\sigma, c \rightsquigarrow \infty) \\ \forall \sigma'.\ \sigma, c \rightsquigarrow \sigma' \Rightarrow \sigma' \in Q$$

The relation $\sigma, c \rightsquigarrow \infty$ is a simple coinductive definition for a small-step semantics, and it was discussed for big-step semantics in Section 2.1. Another approach is to describe the absence of failure and divergence together in one predicate [Nip02].

If the semantics is deterministic, then total correctness can be defined much more succinctly as

$$[P]\ c\ [Q]_3 \triangleq \forall \sigma \in P.\ \exists \sigma'.\ \sigma, c \rightsquigarrow \sigma' \wedge \sigma' \in Q$$

Total correctness is treated only in a minor portion of the separation-logic literature. It can be argued that there is no practical difference between a diverging program and one that terminates after a million years, but total correctness is nevertheless important for discovering bugs. See [Atk10] for an interesting take on amortised running-time analysis with separation logic.

The remainder of this chapter will discuss partial correctness only, but most concepts can be extended to total correctness.

The triples defined above all satisfy the **rule of consequence** and the **existential rule**:

$$\frac{P \vdash P' \quad \{P'\}\ c\ \{Q'\} \quad Q' \vdash Q}{\{P\}\ c\ \{Q\}} \qquad \frac{\forall x.\ \{P(x)\}\ c\ \{Q\}}{\{\exists x.\ P(x)\}\ c\ \{Q\}}$$

Whether they satisfy the frame rule, however, depends on whether the operational semantics satisfies the frame property and safety monotonicity[6], as discussed in Section 2. If these properties should not hold, we can still get the frame rule by instead defining the triple as follows:

$$\{P\}\ c\ \{Q\}_4 \triangleq \forall R.\ \{P * R\}\ c\ \{Q * R\}_1$$

With this definition, we are guaranteed to have the rule of consequence, the existential rule and the frame rule. The technique was first used [BTSY06] in the setting of a higher-order programming language, where it was not clear how to define the frame property, let alone prove it [RS06, BRSY08].

The new triple, $\{P\}\ c\ \{Q\}_4$, is sound with respect to $\{P\}\ c\ \{Q\}_1$ but may not be complete. For example, if the memory allocator is deterministic and always allocates the smallest free location [YO02], then we can no

[6] Safety monotonicity is not required for an affine assertion logic [BJSB11].

longer prove a triple that describes this fact; we can only prove the usual triple for allocation, where the new location is existentially quantified in the postcondition. This can be considered a shortcoming of the theory or a gain in abstraction, depending on viewpoint.

So far, we have implicitly assumed that assertions belong to a complete BI algebra of type $\mathcal{P}_{\leq}(\mathit{state})$ as constructed in Section 3.3.4, where *state* is the type of states in the operational semantics. But interesting separation logics often have some form of instrumentation, or annotations, in the assertions that is not present in the operational semantics. One example is **fractional permissions** [BCOP05], where each heap location has a permission value as well as a data value.

A powerful pattern for defining a triple in such cases has recently emerged [DYBG$^+$13, JB12]. A function $\mathit{reify} : \mathit{asn} \to \mathcal{P}(\mathit{state})$ is defined to translate from instrumented assertions to machine state. Then the triple can be defined as

$$\{P\}\ c\ \{Q\}_5 \triangleq \forall R : \mathit{asn}.\ \{\mathit{reify}(P * R)\}\ c\ \{\mathit{reify}(Q * R)\}_1$$

We still require *asn* to be a complete BI algebra, but $\mathcal{P}(\mathit{state})$ need not be. If we additionally require *reify* to preserve existentials[7] and to be covariant with respect to entailment[8], then the rules of consequence, existential and frame all hold for this triple. Those two properties always hold [JB11, DYBG$^+$13] if *asn* has been constructed from a separation algebra Σ like in Section 3.3.4 and the *reify* function has been lifted from some $f : \Sigma \to \mathcal{P}(\mathit{state})$ as

$$\mathit{reify}(P) = \bigcup_{a \in P} f(a)$$

4.1.2 Structural rules

An inference rule is informally called a **structural rule** when all Hoare triples in it have the same universally-quantified c as their command.

We have already discussed the rule of consequence, the existential rule and the frame rule:

$$\frac{P \vdash P' \quad \{P'\}\ c\ \{Q'\} \quad Q' \vdash Q}{\{P\}\ c\ \{Q\}}\ \textsc{Consequence}$$

$$\frac{\forall x.\ \{P(x)\}\ c\ \{Q\}}{\{\exists x.\ P(x)\}\ c\ \{Q\}}\ \textsc{Exists} \qquad \frac{\{P\}\ c\ \{Q\}}{\{P * R\}\ c\ \{Q * R\}}\ \textsc{Frame}$$

In Section 1.2, I attempted to motivate why these three rules are essential in a separation logic. Any Hoare triple that does not satisfy these rules should come with a good explanation of why not.

[7] Means that $\mathit{reify}(\exists x : T.\ P(x)) = \bigcup_{x:T} \mathit{reify}(P(x))$.
[8] Means that $P \vdash Q$ implies $\mathit{reify}(P) \subseteq \mathit{reify}(Q)$.

There are a few variations on the above rules. Many authors print the existential rule as

$$\frac{\forall x.\ \{P(x)\}\ c\ \{Q(x)\}}{\{\exists x.\ P(x)\}\ c\ \{\exists x.\ Q(x)\}}\ \textsc{Exists}'$$

This has a somewhat satisfying symmetry to it, but it is derivable from Exists and Consequence.

The **disjunction rule** and **vacuity rule** [Rey11] shown below are derivable from Exists because $\vee$ and $\bot$ can be seen as special cases of the existential quantifier as shown in Section 3.2.2.

$$\frac{\{P_1\}\ c\ \{Q\} \quad \{P_2\}\ c\ \{Q\}}{\{P_1 \vee P_2\}\ c\ \{Q\}}\ \textsc{Disjunction} \qquad \frac{}{\{\bot\}\ c\ \{Q\}}\ \textsc{Vacuity}$$

Like the existential rule, the disjunction rule often appears in the literature in a more symmetric but redundant form.

As discussed in Section 3.4, the frame rule typically comes with the side condition that $\mathit{modifies}(c) \cap \mathit{fv}(R) = \emptyset$. We have explored a variation of the frame rule in the Charge! platform [BJB12, BJ], where that side condition is replaced by a substitution:

$$\frac{\{P\}\ c\ \{Q\}}{\{P * R\}\ c\ \{Q * \exists \bar{v} : \mathit{val}.\ R[\bar{v}/\mathit{modifies}(c)]\}}\ \textsc{Frame}$$

The notation is meant to suggest that if $\mathit{modifies}(c) = \{x_1, \ldots, x_n\}$, then $\exists \bar{v} : \mathit{val}.\ R[\bar{v}/\mathit{modifies}(c)]$ means

$$\exists v_1, \ldots, v_n : \mathit{val}.\ R[v_1, \ldots, v_n\ /\ x_1, \ldots, x_n]$$

Essentially, instead of preventing the frame rule from being applied at all, we weaken it to the extent needed for it to hold. This rule is not formally stronger than the standard one, but it allowed us to develop a separation logic without the concept of free variables, which meant there was one less concept to build theory and automation for.

Finally, the **conjunction rule** is of some interest:

$$\frac{\{P\}\ c\ \{Q_1\} \quad \{P\}\ c\ \{Q_2\}}{\{P\}\ c\ \{Q_1 \wedge Q_2\}}\ \textsc{Conjunction}$$

This rule holds for simple definitions of the Hoare triple, such as $\{P\}\ c\ \{Q\}_1$ through $[P]\ c\ [Q]_3$ above, but it fails for many other definitions. Much has been written about what restrictions must be placed on the logic for the conjunction rule to hold [OYR04, BTSY06, O'H07, DYGW11, GBC11, JB11], but comparatively little has been written about why this rule is useful at all.

Some generalisations of the conjunction rule have been proposed. When it holds for binary conjunctions, then it typically also holds for universal quantification over a non-empty domain [Rey11, DYBG+13]:

$$\frac{\forall x : T.\ \{P\}\ c\ \{Q(x)\} \quad T \neq \emptyset}{\{P\}\ c\ \{\forall x : T.\ Q(x)\}}\ \textsc{Universal}$$

In fictional separation logic [JB11], the conjunction rule is generalised to the **recombination rule**, parametrised over a binary operator $\square$:

$$\frac{\{P\}\ c\ \{Q_1\} \quad \{P\}\ c\ \{Q_2\}}{\{P\}\ c\ \{Q_1 \square Q_2\}}\ \textsc{Recombination}$$

This rule holds for $\square = \wedge$ in some cases and for $\square = *$ in other cases [JB11].

4.2 Specification logic

4.2.1 Example: procedure map

The machine state of more realistic languages contains more than a stack and a heap. A language with procedures could, for example, have some form of map $m : M$ from procedure names to their implementation code. The map could in principle be added to the machine state alongside the stack and heap, so states would be $\sigma = (s, h, m)$. But when no command can modify m, at least in a big-step sense, it becomes redundant to have m repeated in the operational semantics on both sides of $\rightsquigarrow$. It is equally redundant in Hoare triples to repeat facts about the m-component in both pre- and postcondition.

The solution in big-step semantics is to separate the state σ that may change after running a command from the state m that may not. A big-step operational semantics is then a relation of the form $\sigma, c \rightsquigarrow_m \sigma'$, and a similar relation could be derived from a small-step semantics. The Hoare triple could be extended analogously, yielding a quadruple; e.g.,

$$m \Vdash \{P\}\ c\ \{Q\}_6 \triangleq \forall \sigma \in P.\ \neg(\sigma, c \rightsquigarrow \mathsf{fail}) \wedge \\ \forall \sigma'.\ \sigma, c \rightsquigarrow_m \sigma' \Rightarrow \sigma' \in Q$$

This definition is not very practical, though: it requires all quadruples to carry the m-parameter even though it is only relevant when c is a procedure call.

The solution is to define a logic *spec* of specifications [Rey82] in which, intuitively, the truth value of a formula is measured by how many m it holds for. The Hoare triple is a formula in this logic, defined along the lines of

$$\{P\}\ c\ \{Q\}_7 \triangleq \{m \mid m \Vdash \{P\}\ c\ \{Q\}_6\}$$

The triple now appears to have only three parameters – the m-parameter has been *hidden* just like the heap is hidden in the assertion logic. Defining

$$\mathit{spec} \triangleq \mathcal{P}_{\leq}(M)$$

for an appropriate preorder, *spec* is a complete Heyting algebra, defined with a Kripke model as in Proposition 3.1 (page 128).

It remains to choose the preorder. Since separation logic emphasises local and modular reasoning, it is beneficial to let the preorder be the extension ordering $\sqsubseteq$ on M. This ensures that any specification that holds for procedure map m also holds in any $m' \sqsupseteq m$, thus enabling **local reasoning for procedures** just as the frame rule enables it for heaps.

We must then show closure under $\sqsubseteq$ for all atomic formulas in *spec*, while the Kripke construction guarantees it for the logical connectives. If the big-step relation is closed under extension of the m-component, then we have $\{P\}\ c\ \{Q\}_7 \in \mathcal{P}_{\sqsubseteq}(M)$ as desired. Otherwise, a technique similar to that used in $\{P\}\ c\ \{Q\}_4$ applies, and we can define a $\{P\}\ c\ \{Q\}_8 \in \mathcal{P}_{\sqsubseteq}(M)$:

$$\{P\}\ c\ \{Q\}_8 \triangleq \{m \mid \forall m' \sqsupseteq m.\ m' \Vdash \{P\}\ c\ \{Q\}_6\}$$

We can additionally define a primitive specification $f(\bar{x}) \mapsto c$ to say that a procedure f has parameters $\bar{x}$ and body c. Despite the similarity to points-to for heaps, no one has yet found a good reason to separate specifications with $*$! With these ingredients, we can define a formula in $\mathcal{P}_{\sqsubseteq}(M)$ to assert that f has specification (P, Q):

$$f(\bar{x}) \mapsto \{P\}\{Q\} \triangleq \exists c.\ f(\bar{x}) \mapsto c \wedge \{P\}\ c\ \{Q\}$$

See [BJSB11, JBK13] for details and variations on the this formula.

Note that if local reasoning for procedures is *not* desired, and the procedure map is static and global, then a much simpler technique applies: make the whole theory parametric in this map [Nip02, PB05, BJSB11]. This is the most common approach, but it results in logics that are not formally modular because there is formally a different theory of program verification for each program! That is, *given* a program fragment (with its procedure map), one *obtains* a theory in which to verify this one fragment. When multiple fragments have been verified this way, each in their own theory, there is no explicit theorem that tells us that the specifications of all the fragments are guaranteed valid in the theory obtained for the composite program.

4.2.2 Other examples

Above, we saw just one example of what a specification logic can do. A rule of thumb is that any state that remains unchanged across commands belongs in the specification logic, and any other state belongs in the assertion logic.

The specification logics considered here are complete Heyting algebras, constructed using the techniques in Section 3.1.3.

Below are some additional applications of specification logic. The various ingredients in the model of specifications tend not to interfere, so a full-featured specification logic can be modelled as $\mathcal{P}_{\leq_1,\ldots,\leq_n}(T_1 \times \cdots \times T_n)$, generalising from the descriptions of $\mathcal{P}_{\leq_i}(T_i)$ below.

Aliasing in call-by-name. The original specification logic by Reynolds [Rey82] was used to describe procedure mappings, much as we saw in Section 4.2.1, but also to assert non-aliasing between procedure parameters in the call-by-name setting of ALGOL 60. Unlike heap aliasing, which can be changed by commands, parameter aliasing stays fixed throughout a scope and is thus a good candidate for describing in the specification logic.

Immutable variables. Languages such as C and Java allow declaring certain variables as *constant* or *immutable*. In ML-like languages, all variables are immutable. All such variables could be described in the specification logic rather than the assertion logic. I have not seen this done in practice, though.

Recursion. To aid in verifying recursive procedures, it has proved useful to add natural numbers to the specification logic to count either the depth of recursion [vO99, Nip02] or the number of execution steps remaining [AM01, AMRV07, BJSB11]. This is known as **step indexing**. In both cases, specifications are downwards-closed sets of natural numbers: $spec = \mathcal{P}_{\geq}(\mathbb{N})$. Intuitively, the truth value of a specification measures how many steps of execution (or depth of procedure calls) the specification will hold for; a valid specification holds for any number of steps (or any depth of procedure calls).

This model enables the definition of a **later-operator** [Nak00, AMRV07, DAB09, DAH08, JBK13] on specifications:

$$\triangleright S \triangleq \{k \mid \forall k' < k.\ k' \in S\}$$

Intuitively, $\triangleright S$ means that S will hold after one step of execution (or for recursive calls one level deeper). The rule for procedure calls then requires only $\triangleright(f(\bar{x}) \mapsto \{P\}\{Q\})$ in its assumptions, and we can get this assumption by applying the **Löb rule** with $S = f(\bar{x}) \mapsto \{P\}\{Q\}$:

$$\frac{\triangleright S \vdash S}{\vdash S}\ \textsc{Löb}$$

Counting recursion depth clearly belongs in the specification logic because the depth cannot have changed after executing some c. On the

other hand, counting steps of program execution can also be done usefully in the assertions, which allows for a $\triangleright$-operator in the assertion logic [BJSB11].

Recursive specifications. A specification can be defined recursively just like any other predicate, but this will always be subject to well-formedness restrictions, typically requiring the recursive occurrence to be in a **positive position** in the formula, or require a **well-founded** term to become smaller with each self-application. For instance, the existence of a specification S satisfying $S \equiv S \Rightarrow \bot$ would render the logic inconsistent[9].

Having step-indexes in the specifications, as sketched above, allows recursive definitions with a third type of well-formedness restriction: *contractiveness* [AM01, DAH08, BRS$^+$11]. This allows definitions in which the recursive occurrence can be anywhere, as long as it is syntactically under a $\triangleright$-operator. This allows defining an S such that $S \equiv (\triangleright S) \Rightarrow \bot$. This was a silly example, but useful examples can be found in [DAH08, BRS$^+$11].

A powerful alternative to this would be to use a metalogic that has a $\triangleright$-operator and then lift this into the separation logic. For an example, see the work on impredicative CAP [SB13a], which uses the *topos of trees* [BMSS12] as its metalogic.

Frames. Consider a small variation on $\{P\}\ c\ \{Q\}_4$:

$$\{P\}\ c\ \{Q\}_9 \triangleq \{R \mid \{P * R\}\ c\ \{Q * R\}_1\}$$

Then specifications are predicates on assertions, and the validity of a triple is intuitively measured by how many assertions can be framed on to it. This immediately gives us a **frame operator** [BTSY05, BTSY06, Kri12, JBK13], traditionally written $\otimes$, defined as

$$S \otimes R \triangleq \{P \mid (P * R) \in S\}$$

It satisfies the following identity, which allows us to write more concise specifications that do not repeat assertions between pre- and postcondition:

$$\{P\}\ c\ \{Q\}_9 \otimes R \equiv \{P * R\}\ c\ \{Q * R\}_9$$

As in the procedure-map example above, when we make *spec* a Kripke model, we can extend any useful closure property from atomic specifications to the full logic. Here, we expect that the frame rule holds for triples; i.e.,

$$\{P\}\ c\ \{Q\}_9 \vdash \{P\}\ c\ \{Q\}_9 \otimes R$$

[9] Exercise! First prove $S \vdash \bot$, which proves $\vdash S$, and together they prove $\vdash \bot$.

If so, we can extend the frame rule to all specifications by defining $spec = \mathcal{P}_{\sqsubseteq}(asn)$, where $\sqsubseteq$ is the extension ordering on the monoid $(asn, *, emp)$. This gives the following inference rule, called the **higher-order frame rule**.

$$\frac{}{S \vdash S \otimes R}$$

Under certain conditions, this rule allows framing invariants onto triples in negative positions of an entailment, whereas the ordinary frame rule only allows it for positive positions. Examples of its utility as a second-order frame rule are found in [OYR04, BTSY06, JBK13]; an example of using it as a third-order frame rule is in [BTSY06].

4.2.3 Structural rules in specification logic

The structural rules discussed in Section 4.1.2 can now typically[10] be entailments in the specification logic rather than the metalogic. For example, the existential rule becomes

$$\frac{}{\forall x.\ \{P(x)\}\ c\ \{Q\} \vdash \{\exists x.\ P(x)\}\ c\ \{Q\}}\ \textsc{Exists}$$

To get a presentation that looks more standard, it is customary to quantify over all specifications S and instead print the rule as

$$\frac{S \vdash \forall x.\ \{P(x)\}\ c\ \{Q\}}{S \vdash \{\exists x.\ P(x)\}\ c\ \{Q\}}\ \textsc{Exists}$$

If specifications can be usefully embedded in assertions, then the rule of consequence can also make use of this S; see the consequence rule in [PB08] for an example.

4.3 Alternative formulations

An assertion logic, as developed in Section 3, can also serve as ingredient in other theories than the specification logics defined above. A brief overview is given here.

4.3.1 Rigid specification logics

The specification logics described above are complete Heyting algebras and thus full higher-order logics. Less expressive and more disciplined logics have also been proposed. In particular, the logic of Parkinson and Bierman [PB08], extended by van Staden and Calcagno [vSC10], stands out as a specification logic that is expressive enough to specify most object-oriented code but requires specifications to follow a rigid structure that is essentially a mirror image of the class structure of an object-oriented program.

[10] I know of one exception to this: the higher-order frame rule in [SBRY11].

Several challenging specifications have been expressed in this system [PB08, DP08]. On the other hand, the extensions made in [vSC10] certainly extended the range of useful specifications that could be expressed even though the original system perhaps seemed powerful enough at first glance. It is likely that a third case study would expose the need for further extensions, and so on. Essentially, these rigid logics need almost all the features of a complete Heyting algebra: *auxiliary variables* are universal quantifiers, *abstract predicates* are existential quantifiers, *specification refinement* is entailment, *specification combination* is conjunction, and so on.

When the specification logic is a complete Heyting algebra from the beginning, there is more freedom to use it in new ways without requiring extensions. Rigid specification logics can then be built on top as needed, hopefully reducing the burden of the soundness proof for these.

Compare [KAB+09, BJSB11], where programs are specified in a separation logic in which both specifications and assertions are higher-order logics. The drawback is that specifications can be hard to understand when their structure does not follow a known pattern. They can be more or less verbose compared to a specification in a rigid framework, depending on whether that framework is a good fit for the code at hand.

Rigid logics seem to be a good fit to model stand-alone verification tools with extensive automation [vSC10, DP08, JSP12]. Full higher-order logics tend to be embedded in proof assistants such as Coq and HOL, where automation is guaranteed to be sound but runs orders of magnitude slower [CMM+09, Chl11, Chl13, McC09, Tue09, MSBS12, BJB12, JBK13].

4.3.2 Type systems

It is possible to formulate a separation logic as a type system. Type inference will of course be undecidable, and type checking will require annotations corresponding to proofs in a program logic.

In Hoare type theory [NMB08, NAMB07, PBNM08], a computation has a monadic type, similar to the IO monad in Haskell but with pre- and postcondition annotations. Ignoring variable contexts, the typing judgement $c : \{P\}\ x{:}A\ \{Q\}$ means that computation c has precondition P and returns a value of type A, bound as x in postcondition Q. Essentially, their types correspond to our *specifications*. Pre- and postconditions are predicates on the heap, almost exactly as we defined them in Section 3.

In the type system of Pottier [Pot08, SBP+12], as well as Krishnaswami et al. [KTDG12], their types essentially correspond to our *assertions*. Like in ML, a computation is a function, and the semantics has to be call-by-value to serialise side effects predictably. A function that writes to a heap cell has a dependent type along the lines of $\Pi l.\ \mathit{cap}(l) \times \mathit{val} \to \mathit{cap}(l)$, where $\mathit{cap}(l)$ is an abstract capability to access location l. The capability and the

function space are **linear**, and the capability is therefore returned again by the function; otherwise, it would be lost to the caller. It is understood that capability tokens will be compiled away, but they do have a representation in the term language. Capability types can be composed with separating conjunction, which makes these systems behave much like separation logic.

One thing to beware of when building a separation-logic type system is the handling of logical variables. Consider for instance a procedure that increments the value at a given heap location, specified in the style of Section 4.2.1:

$$\forall i.\ \mathsf{inc}(\mathsf{x}) \mapsto \{\mathsf{x} \mapsto i\}\{\mathsf{x} \mapsto i + 1\}$$

There is a clear distinction here between x, which has a run-time representation, and i, which exists purely in the specification. If the arrow and dependent-product types denote function spaces, as in the example with capabilities above, then there must be a different mechanism for quantifying over a logical variable. Nanevski et al. proposed "binary postconditions" [NVB10, NMS+08] for addressing this in Hoare type theory, but their solution restricts the scope of logical variables to a single triple. Another branch of Hoare type theory [CMM+09] proposed explicitly marking logical variables as such, but this required extending Coq with an axiom whose soundness has not been formally established. The type system in [BTSY06] does not include logical variables, and thus it cannot specify inc. The original system of Pottier [Pot08] had the same problem, but this was later addressed [PP11] by adding logical universal quantifiers and singleton types, which is also how [KTDG12] handles the problem.

5 Conclusion

At the time of writing this text, the ACM Digital Library lists 147 publications with the keyword "separation logic". It is well known that these have a lot in common underneath their cosmetic differences. Unfortunately, those commonalities are typically treated as *design patterns* to draw inspiration from when building a separation-logic theory from scratch, rather than a formally *reusable theory* that can be built upon.

In this text, I have presented a core of standard definitions and theorems in the hope that future texts on separation logic can take these for granted rather than recreate them. Those definitions lead to expressive higher-order logics without adding additional complexity over the first-order case.

In particular, a typical assertion logic should arise as the powerset of a separation algebra, closed under a preorder. The interesting contribution of future theories should be the choice of separation algebra and preorder, while turning this into a logic is standard. Similarly, specification logics are made easy since they arise from standard Kripke models, which automatically contain all the operators and quantifiers needed for abstract and

modular specifications. Finally, simple but fully formal treatment of program variables can be achieved using open terms, although this is not as widely applicable as the other theories mentioned.

At the same time, we have discussed the aspects that still remain patterns and cosmetics. Variations in the operational semantics and Hoare triple are necessarily language-specific, and there is rarely any formal reuse between theories, but there are still many design patterns to be borrowed.

Much more ought to be said about concurrent separation logic and models that use guarded recursion, but these areas are still very much in flux. Hopefully, general and reusable theories will eventually emerge from those lines of research.

Acknowledgements. I would like to thank the proof readers – Jesper Bengtson, Lars Birkedal, Aleš Bizjak and Marco Paviotti – for helpful feedback and discussions.

Index

References

[AB07] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step C Minor. In *Proceedings of TPHOLs*, 2007.

[AM01] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 2001.

[AM13] Reynald Affeldt and Nicolas Marti. Towards formal verification of TLS network packet processing written in C. In *Proceedings of PLPV*, 2013.

[AMRV07] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. In *Proceedings of POPL*, 2007.

[App06] Andrew W. Appel. Tactics for separation logic, Draft of January 2006. `http://www.cs.princeton.edu/~appel/papers/septacs.pdf`.

[Atk10] Robert Atkey. Amortised resource analysis with separation logic. *Programming Languages and Systems*, pages 85–103, 2010.

[BBTS05] B. Biering, L. Birkedal, and N. Torp-Smith. BI hyperdoctrines and higher-order separation logic. In *Proceedings of ESOP*, 2005.

[BC10] James Brotherston and Cristiano Calcagno. Classical BI: Its semantics and proof theory. *Logical Methods in Computer Science*, 6(3), 2010.

[BCOP05] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In *Proceedings of POPL*, 2005.

[BJ] Jesper Bengtson and Jonas B. Jensen. Charge! development version. `https://github.com/jesper-bengtson/Charge`.

[BJB12] Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. Charge! – a framework for higher-order separation logic in Coq. In *Proceedings of ITP*, 2012.

[BJSB11] Jesper Bengtson, Jonas Braband Jensen, Filip Sieczkowski, and Lars Birkedal. Verifying object-oriented programs with higher-order separation logic in Coq. In *Proceedings of ITP*, 2011.

[BK10] J. Brotherston and M. Kanovich. Undecidability of propositional separation logic and its neighbours. In *Proceedings of LICS*, 2010.

[BMSS12] L. Birkedal, R. Møgelberg, J. Schwinghammer, and K. Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Logical Methods in Computer Science*, 8(4), October 2012.

[Boy03] John Boyland. Checking interference with fractional permissions. In *Proceedings of SAS*, 2003.

[Bro07] Stephen Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1):227–270, 2007.

[BRS+11] 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.

[BRSY08] L. Birkedal, B. Reus, J. Schwinghammer, and H. Yang. A simple model of separation logic for higher-order store. In *Proceedings of ICALP*, 2008.

[BTSY05] L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules. In *Proceedings of LICS*, 2005.

[BTSY06] L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. *Logical Methods in Computer Science*, 2(5:1), August 2006.

[BV13] James Brotherston and Jules Villard. Parametric completeness for separation theories, Submitted, 2013.

[CGZ05] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Context logic and tree update. In *Proceedings of POPL*, 2005.

[Chl11] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of PLDI*, 2011.

[Chl13] Adam Chlipala. The Bedrock structured programming system. In *Proceedings of ICFP*, 2013.

[CMM+09] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *Proceedings of ICFP*, 2009.

[COY07] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *Proceedings of LICS*, 2007.

[CSV07] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified self-modifying code. In *Proceedings of PLDI*, 2007.

[DAB09] Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. In *Proceedings of LICS*, 2009.

[DAH08] Robert Dockins, Andrew W. Appel, and Aquinas Hobor. Multimodal separation logic for reasoning about operational semantics. *Electronic Notes in Theoretical Computer Science*, 218:5–20, 2008.

[DHA09] Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In *Proceedings of APLAS*, 2009.

[DP08] Dino Distefano and Matthew J. Parkinson. jstar: towards practical verification for java. In *Proceedings of OOPSLA*, 2008.

[DYBG+13] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. Views: Compositional reasoning for concurrent programs. In *Proceedings of POPL*, 2013.

[DYDG+10] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *Proceedings of ECOOP*, 2010.

[DYGW10] Thomas Dinsdale-Young, Philippa Gardner, and Mark Wheelhouse. Abstraction and refinement for local reasoning. In *Proceedings of VSTTE*, 2010.

[DYGW11] Thomas Dinsdale-Young, Philippa Gardner, and Mark Wheelhouse. Abstraction and refinement for local reasoning, February 2011. Journal submission.

[FFS10] Rodrigo Ferreira, Xinyu Feng, and Zhong Shao. Parameterized memory models and concurrent separation logic. In *Proceedings of ESOP*, 2010.

[GBC+07] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *Proceedings of APLAS*, 2007.

[GBC11] Alexey Gotsman, Josh Berdine, and Byron Cook. Precision and the conjunction rule in concurrent separation logic. In *Proceedings of MFPS*, 2011.

[GLW06] Didier Galmiche and Dominique Larchey-Wendling. Expressivity properties of boolean BI through relational models. In *Proceedings of FSTTCS*, 2006.

[GMP02] D. Galmiche, D. Méry, and D. Pym. Resource tableaux (extended abstract). In *Proceedings of CSL*, 2002.

[GMP05] D. Galmiche, D. Mery, and D. Pym. Semantics of BI and resource tableaux. *Mathematical Structures in Computer Science*, 15(6):1033–1088, 2005.

[GMS12] Philippa Gardner, Sergio Maffeis, and Gareth Smith. Towards a program logic for javascript. In *Proceedings of POPL*, 2012.

[HDV11] C-K Hur, Derek Dreyer, and Viktor Vafeiadis. Separation logic in the presence of garbage collection. In *Proceedings of LICS*, 2011.

[Hob11] Aquinas Hobor. Improving the compositionality of separation algebras, July 2011. Unpublished draft.

[IO01] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of POPL*, 2001.

[JB11] Jonas Braband Jensen and Lars Birkedal. Fictional separation logic: Appendix, 2011. `http://itu.dk/~jobr/research/fsl-appendix.pdf`.

[JB12] Jonas B. Jensen and Lars Birkedal. Fictional separation logic. In *Proceedings of ESOP*, 2012.

[JBK13] Jonas B. Jensen, Nick Benton, and Andrew Kennedy. High-level separation logic for low-level code. In *Proceedings of POPL*, 2013.

[JSP12] Bart Jacobs, Jan Smans, and Frank Piessens. The VeriFast program verifier: A tutorial, December 2012.

[KAB+09] Neelakantan R. Krishnaswami, Jonathan Aldrich, Lars Birkedal, Kasper Svendsen, and Alexandre Buisse. Design patterns in separation logic. In *In Proceedings of TLDI*, 2009.

[KBJD13] Andrew Kennedy, Nick Benton, Jonas B. Jensen, and Pierre-Evariste Dagand. Coq: The world's best macro assembler? In *Proceedings of PPDP*, 2013.

[Kri12] Neelakantan R. Krishnaswami. *Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic*. PhD thesis, Carnegie Mellon University, 2012.

[KTDG12] Neelakantan R. Krishnaswami, Aaron Turon, Derek Dreyer, and Deepak Garg. Superficially substructural types. In *Proceedings of ICFP*, 2012.

[LG09] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009.

[LWN13] Ruy Ley-Wild and Aleksandar Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *Proceedings of POPL*, 2013.

[McC09] Andrew McCreight. Practical tactics for separation logic. In *Proceedings of TPHOLs*, 2009.

[MG07] M. O. Myreen and M. J. C. Gordon. Hoare logic for realistically modelled machine code. In *Proceedings of TACAS*, 2007.

[MSBS12] Hannes Mehnert, Filip Sieczkowski, Lars Birkedal, and Peter Sestoft. Formalized verification of snapshotable trees: Separation and sharing. In *Proceedings of VSTTE*, 2012.

[Myr10] M. O. Myreen. Verified just-in-time compiler on x86. In *Proceedings of POPL*, 2010.

[Nak00] Hiroshi Nakano. A modality for recursion. In *Proceedings of LICS*, 2000.

[NAMB07] Aleksandar Nanevski, Amal Ahmed, Greg Morrisett, and Lars Birkedal. Abstract predicates and mutable ADTs in Hoare type theory. In *In Proceedings of ESOP*, 2007.

[Nip02] Tobias Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In *Proceedings of CSL*, 2002.

[NMB08] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Hoare Type Theory, Polymorphism and Separation. *Journal of Functional Programming*, 18(5–6):865–911, 2008.

[NMS+08] Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Reasoning with the awkward squad. In *Proceedings of ICFP*, 2008.

[NS06] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *Proceedings of POPL*, 2006.

[NVB10] Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. Structuring the verification of heap-manipulating programs. In *Proceedings of POPL*, 2010.

[O'H07] Peter W. O'Hearn. Resources, concurrency and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[O'H12] Peter W. O'Hearn. A primer on separation logic (and automatic program verification and analysis). *NATO Science for Peace and Security Series D*, 33:286–318, 2012.

[OP99] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[OYR04] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proceedings of POPL*, 2004.

[Par05] Matthew Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, November 2005.

[Par10] Matthew Parkinson. The next 700 separation logics. In *Proceedings of VSTTE*, 2010.

[PB05] Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In *Proceedings of POPL*, 2005.

[PB08] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *Proceedings of POPL*, 2008.

[PBC06] M. J. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in Hoare logic. In *Proceedings of LICS*, 2006.

[PBNM08] R. L. Petersen, L. Birkedal, A. Nanevski, and G. Morrisett. A realizability model of impredicative Hoare type theory. In *Proceedings of ESOP*, 2008.

[Pit01] A. M. Pitts. Nominal logic, a first order theory of names and binding. In *Proceedings of TACS*, 2001.

[Pot08] François Pottier. Hiding local state in direct style: a higher-order anti-frame rule. In *Proceedings of LICS*, 2008.

[Pot13] François Pottier. Syntactic soundness proof of a type-and-capability system with hidden state. *Journal of Functional Programming*, 23(1):38–144, January 2013.

[POY04] David J. Pym, Peter W. O'Hearn, and Hongseok Yang. Possible worlds and resources: the semantics of bi. *Theoretical Computer Science*, 315(1):257—-305, May 2004.

[PP11] Alexandre Pilkiewicz and François Pottier. The essence of monotonic state. In *Proceedings of TLDI*, 2011.

[PS12] Matthew Parkinson and Alexander J. Summers. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 2012.

[Pym02] D.J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*, volume 26 of *Applied Logic Series*. Kluwer Academic Publishers, 2002.

[Rey82] John C. Reynolds. Idealized Algol and its specification logic. *Tools and notions for program construction*, pages 121–161, 1982.

[Rey00] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. *Millennial Perspectives in Computer Science*, pages 303—321, 2000.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of LICS*, 2002.

[Rey11] John C. Reynolds. Introduction to separation logic, 2011. Course notes for 15-818A3 at Carnegie Mellon University.

[RS06] Bernhard Reus and Jan Schwinghammer. Separation logic for higher-order store. In *Proceedings of CSL*, 2006.

[SB13a] Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates, 2013. Submitted for publication.

[SB13b] Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates (technical appendix), 2013.

[SBP09] Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. Verifying generics and delegates (technical appendix). Technical report, IT University of Copenhagen, 2009. Available at `http://itu.dk/~kasv/generics-delegates-tr.pdf` or as part of Kasper Svendsen's PhD thesis.

[SBP10] K. Svendsen, L. Birkedal, and M.J. Parkinson. Verifying generics and delegates. In *Proceedings of ECOOP*, 2010.

[SBP+12] J. Schwinghammer, L. Birkedal, F. Pottier, B. Reus, K. Støvring, and H. Yang. A step-indexed Kripke model of hidden state. *Mathematical Structures in Computer Science*, 2012.

[SBP13] Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. Modular reasoning about separation for concurrent data structures. In *Proceedings of ESOP*, 2013.

[SBRY11] J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang. Nested Hoare triples and frame rules for higher-order store. *Logical Methods in Computer Science*, 7(3:21), July 2011.

[SJP10] Jan Smans, Bart Jacobs, and Frank Piessens. Heap-dependent expressions in separation logic. In *Proceedings of FMOODS*, 2010.

[TKN07] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *Proceedings of POPL*, 2007.

[TSFC09] Gang Tan, Zhong Shao, Xinyu Feng, and Hongxu Cai. Weak updates and separation logic. In *Proceedings of APLAS*, 2009.

[Tue09] Thomas Tuerk. A formalisation of Smallfoot in HOL. In *Proceedings of TPHOLs*, 2009.

[Vaf11] Viktor Vafeiadis. Concurrent separation logic and operational semantics. *Electron. Notes Theor. Comput. Sci.*, 276:335–351, September 2011.

[VB08] C. Varming and L. Birkedal. Higher-order separation logic in Isabelle/HOLCF. *Electr. Notes Theor. Comput. Sci.*, 218:371–389, 2008.

[VN13] Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *Proceedings of OOPSLA*, 2013.

[vO99] David von Oheimb. Hoare logic for mutual recursion and local variables. 1999.

[vSC10] Stephan van Staden and Cristiano Calcagno. Reasoning about multiple related abstractions with MultiStar. In *Proceedings of OOPSLA*, 2010.

[WB11] Ian Wehrman and Josh Berdine. A proposal for weak-memory local reasoning, Presented at the LOLA workshop, 2011.

[WDP13] John Wickerson, Mike Dodds, and Matthew Parkinson. Ribbon proofs for separation logic. In *Proceedings of ESOP*, 2013.

[WN04] Martin Wildmoser and Tobias Nipkow. Certifying machine code safety: Shallow versus deep embedding. In *Proceedings of TPHOLs*, 2004.

[YO02] Hongseok Yang and Peter W. O'Hearn. A semantic basis for local reasoning. In *Proceedings of FoSSaCS*, 2002.