

Verifying Higher Order Imperative Programs with Higher Order Separation Logic

Neelakantan Krishnaswami
neelk+@cs.cmu.edu

February 13, 2008

Abstract

In this thesis proposal document, I claim I will be able to modularly verify interesting higher-order imperative programs written in a high level language, using a higher order separation logic.

I discuss what modular verification means, and discuss the case studies I will do to examine how it can be supported using the logic we have developed. Finally, I give a summary of the formal system I have developed so far. This system combines three methods of reasoning about programs: first, a system of equational reasoning using β and η rules familiar to functional programmers; second, a separation logic-based Hoare logic for reasoning about primitive imperative commands; and third, a specification logic in the style of LCF and Reynolds's work on Idealized Algol.

1 Motivation

It is more difficult to reason about programs that use aliasing than ones that do not use mutable shared data. It is more difficult to reason about programs that use higher order features than first order programs. Put both together, and the problems become both challenging and interesting for formal verification.

The challenge arises from the fact that combining aliased mutable state with higher order programming yields a language that is more difficult than the sum of its parts. Techniques to reason about purely functional programs, which extensively use higher-order methods but eschew mutable state, are an extremely well-developed branch of programming language theory, with a long and successful history.

Historically, techniques to reason about mutable state have lagged behind, but some years ago O'Hearn and Reynolds proposed separation logic [26], which has proven extremely successful at enabling reasoning about even such intricate imperative programs as garbage collectors [3]. However, separation logic has historically focused on low-level programming languages that lack a strong type discipline and allow the use of techniques such as treating pointers as integers.

Most high level languages that allow the use of state are also prone to aliasing errors, since (with a few exceptions) type systems do not track interference properties. Even languages such as Haskell, which isolate all side-effects within a particular family of monadic types, still face the problem that reasoning about monadic code remains as difficult as ever. Mutation and state have been imprisoned, but they have not been rehabilitated.

If this combination only posed challenges, but had no applications, then it would be only of theoretical interest. However, it is possible to use it to simply and naturally express many common kinds of programs. For example, graphical user interfaces are typically structured as families of callbacks that operate over the shared data structures representing the interface and program. These callbacks are structured using the subject-observer pattern, which use collections of callbacks to implicitly synchronize mutable data structures across a program.

2 Proposed Thesis

The thesis I want to demonstrate is: “The modular formal verification of interesting higher-order imperative programs written in high level languages is natural and feasible using higher-order separation logic.”

3 Higher Order Separation Logic

The main tool I will use to prove programs correct is a higher order separation logic for a simple higher-order imperative programming language. The details of the language and the specification logic are given at the end of the proposal document, but I will say a few words about them here to set the stage for what follows.

The programming language I consider is a simply-typed monadic functional language, in Pfenning-Davies [21] style. It can be considered an instance of Moggi’s [17] monadic metalanguage – all heap effects are encapsulated within the monadic type constructor.

The basic form of specification is the Hoare triple over an imperative computation $\{P\}c\{a : \tau. Q\}$. Here, P is the precondition, c is the computation that we are specifying, and Q is the post-condition. The binder $a : \tau$ names the return value of the computation, so that we can mention it in the post-condition.

The assertion logic for the pre- and post-conditions is higher order separation logic [2]. This is a substructural logic that extends ordinary logic with three spatial connectives, which enable reasoning about the aliasing behavior of data. These connectives are the points-to assertion $e \mapsto e'$, which asserts that the reference e has e' as its contents. Second, we have the separating conjunction $p * q$, which says that the program state can be divided into two disjoint regions, one of which is described by p , and the other of which is described by q . This built-in disjointness is the key to a compact treatment of aliasing. Finally, we

have the “magic wand” $p \multimap q$, which is the separating implication adjoint to the separating conjunction. Intuitively, it means that if a disjoint heap described by p were joined to the current heap, then the whole combined heap would be described by q .

The Hoare triples themselves form the atomic propositions of a multi-sorted first-order intuitionistic logic of specifications, in the style of Reynolds’ specification logic for Algol. The quantifiers range over the sorts of the assertion logic, so that we can universally and existentially quantify over assertions and expressions. Implication over triples lets us specify the behavior of higher-order functions – we give its behavior as a conditional dependent on its higher-order arguments satisfying some specification.

Below, we give an example of a specification for a higher-order function encoding a for loop:

```

1    $\exists \text{repeat} : \mathbb{N} \rightarrow \circ 1 \rightarrow \circ 1.$ 
2    $\forall \text{body} : \circ 1.$ 
3    $\forall P : \mathbb{N} \Rightarrow \text{prop}.$ 
4      $(\forall i : \mathbb{N}. \{P(i)\} \text{body} \{a : 1. P(i + 1)\})$ 
5      $\text{implies}$ 
6      $(\forall j, k : \mathbb{N}. \{P(j)\} \text{repeat } k \text{ body} \{a : 1. P(j + k)\})$ 

```

In the first line, we assert the existence of a two argument function **repeat**, which takes an integer and a monadic computation as arguments, and returns a new monadic computation. In the next two lines, we quantify over all computations **body** and predicates P on \mathbb{N} . Then, we say that if **body** takes the precondition state $P(i)$ to the postcondition $P(i + 1)$, then **repeat** k **body** will take $P(j)$ to $P(j + k)$. Observe that we use implication over specifications to constrain the behavior of the argument to **repeat**.

4 Modular Verification

In order for a verification methodology to scale up to even modestly- sized programs, it must be modular. There are three informal senses in which I use the word “modular”, each of which is supported by different features of our logic.

1. First, we should be compositional – we should be able to verify programs a module at a time. That is, we should be able to give the specification of a library, and prove both that implementations satisfy the specification without knowing anything about the clients that use it, and likewise prove the correctness of client programs without knowing anything about the implementation beyond what is promised by the specification.

For example, we should be able to specify a hash table without revealing to a client whether the hash table implementation uses double hashing or

linear chaining. Thus, a client verified against the specification will work when joined with a module that uses either implementation strategy.

We will address this problem by making use of the fact that the presence of existential specifications in our specification logic permits us to use the Mitchell-Plotkin encoding of modules as existentials. Just as we can use existentially-quantified program variables to hide the implementation of functions in a module, we can use existentially-quantified heap predicates to hide the exact details of a module's invariants from a client [2].

2. Related to this is a modular treatment of aliasing. Ordinary Hoare logic becomes non-modular when asked to treat mutable, shared state, because we must explicitly specify the aliasing or non-aliasing of every variable and mutable data structure in our pre- and post-conditions. Besides the quadratic dependence on the size of the program, the relevant set of variables grows whenever one subprogram is embedded in a larger one.

Separation logic resolves this problem in two steps. First, the spatial connectives, such as the separating conjunction $P * Q$, implicitly states these non-aliasing conditions, which shrinks the size of the program assertions. Then, these connectives permits the statement of the frame rule, a feature which we carry forward in our logic. This rule permits local reasoning, in the sense that first, a program's pre- and post-conditions only need to refer to the state that the program actually reads or writes, and second, we can assume that any state the program doesn't look at remains untouched. Thus, subprograms with disjoint invariants do not need to mention one another's invariants.

3. Finally, it is important to ensure that the abstractions we introduce compose. Benton[?] described a situation where he was able to prove that a memory allocator and the rest of the program interacted only through a particular interface, but when he tried to divide the rest of the program into further modules, he encountered difficulties, because it was unclear how to share the resources and responsibility for upholding the contract with the allocator.

4.1 Existential Modularity

The specification logic I give is a first order logic of Hoare triples. This gives us a simple technique for separating implementation and client – namely, we can use existentially quantified specifications to create the boundary between a module and its client. We can verify a client program by proving it under the *hypothesis* that it has access to a program satisfying the existentially quantified specification, and we can verify the module implementation by proving that it *witnesses* the truth of the specification. Then, linking the two together is merely an application of the cut rule of logic.

As described above, this is essentially an application of Mitchell and Plotkin's identification of data abstraction and existential types. It's worth noting at this

point that we do not have existential types in our programming language (yet?). Instead, we are abstracting over the *heap*. As a concrete example, consider the specification of a counter module.

```

1    $\exists \text{counter} : \text{ref } \mathbb{N} \times \mathbb{N} \Rightarrow \text{prop}$ 
2    $\exists \text{create} : \mathbf{1} \rightarrow \bigcirc \text{ref } \mathbb{N}$ 
3    $\exists \text{next} : \text{ref } \mathbb{N} \rightarrow \bigcirc \mathbb{N}$ 
4    $\{\text{emp}\} \text{run } \text{create}() \{a : \text{ref } \mathbb{N}. \text{counter}(a, 0)\}$ 
5   and
6    $\forall c, n. \{\text{counter}(c, n)\} \text{run } \text{next}(c) \{a : \mathbb{N}. a = n \wedge \text{counter}(c, n + 1)\}$ 

```

In line 1, we assert the existence of a $\text{counter}(x, n)$ predicate, which asserts that x is a counter with current value n . In lines 2 and 3, we assert the existence of **create** and **next** functions, which create a new counter object, and increment a given counter, respectively. The specification for **create** is given in line 4, where we assert that calling **create** returns a new counter, and the specification of **next** is given in line 6, where we assert that calling **next**(c) returns the current value of the counter, and increments the value of the counter state.

Here is one possible implementation for the existential witnesses:

$$\begin{aligned} \text{counter} &\equiv \lambda(r, n). (r \mapsto n) \\ \text{create} &\equiv \lambda(). [\text{new}_{\mathbb{N}} 0] \\ \text{next} &\equiv \lambda r. [\text{letv } n = [!r] \text{ in letv } () = [r := n + 1] \text{ in } n] \end{aligned}$$

In this implementation, the counter is represented by an integer reference to the value of the counter. The predicate $\text{counter}(r, n)$ is defined as the function that returns the assertion $r \mapsto n$. **create** simply allocates a new reference, and **next** reads the value of the reference, increments it, and returns the old value.

Below, is another possible implementation.

$$\begin{aligned} \text{counter} &\equiv \lambda(r, n). (r \mapsto 2n + 7) \\ \text{create} &\equiv \lambda(). [\text{new}_{\mathbb{N}} 7] \\ \text{next} &\equiv \lambda r. [\text{letv } n = [!r] \text{ in letv } () = [r := n + 2] \text{ in } (n - 7)/2] \end{aligned}$$

In this implementation, we also use an integer reference. However, the contents of the reference are a function of the visible state: if the abstract counter state is n , the reference points to $2n + 7$.

As an aside, note that a well-specified client program cannot distinguish these two implementations, even though the reference type is a concrete (and not abstract) type, because of the resource semantics of separation logic. Merely knowing the counter module's implementation type does not let us access the reference, because we must have a points-to assertion $e \mapsto e'$ to dereference the reference e .

4.2 Imperative Modularity

4.2.1 Separation and Modularity

In the previous example, we gave a specification of an imperative counter, and used existentially quantified predicates to hide the implementation of the counter from client programs. However, since this is an imperative program, the identity of each individual counter object matters – incrementing one counter will leave other, distinct counters unchanged. We want to exploit this noninterference to shrink the size of our program proofs, which we do via the *frame rule* of separation logic, which says for triples:

$$\frac{\Gamma; \Sigma \vdash \{P\}c\{a : \tau. Q\} \text{ ok}}{\Gamma; \Sigma \vdash \{P * R\}c\{a : \tau. Q * R\} \text{ ok}} \text{FRAMER}$$

The reading of this rule is that if a program c takes a heap described by a precondition P to a heap described by a postcondition Q , then c will also take a heap in $P * R$ to a heap in $Q * R$. That is, any disjoint state the program c doesn't touch will be unchanged by its execution. To see this in action, consider the following client program, proven in an environment where the counter existential has been unpacked (that is, an environment with *counter*, **create**, and **next** free, and with the appropriate specifications for the two functions).

```

1   {emp}
2   letv  $c_1 = \text{create}()$  in
3   { $\text{counter}(c_1, 0)$ }
4   letv  $c_2 = \text{create}()$  in
5   { $\text{counter}(c_1, 0) * \text{counter}(c_2, 0)$ }
6   run next( $c_1$ )
7   { $a : \mathbb{N}. la = 0 \wedge \text{counter}(c_1, 1) * \text{counter}(c_2, 0)$ }

```

On line 1, we assume that the initial heap is empty. On line 2, we create a new counter bound to c_1 , which updates the post-condition on line 3 according to our specification of **create** (which is line 4 of the counter specification above). On line 4, we **create** another counter c_2 .

This time, note that the precondition state is $\text{counter}(c_1, 0)$, rather than **emp**, which is what the precondition asks for **create** asks for. What justifies using this rule is the fact that we can use the frame rule to add $\text{counter}(c_1, 0)$ onto both the pre- and post-condition of **create**. That is, it is harmless to ignore $\text{counter}(c_1, 0)$, since it is not touched by invocations of **create**. So we can conclude the postcondition in line 5 is $\text{counter}(c_1, 0) * \text{counter}(c_2, 0)$ – we have two distinct counters. In line 6, we call **next** on c_1 . Once again, the precondition in line five does not match the precondition of **next** (in line 6 of the previous counter specification), because we have an extraneous $\text{counter}(c_2, 0)$. And just as before, we can use the frame rule to conclude that we can frame this onto both the pre- and post-conditions, giving the post-condition line 7 of our annotated program.

4.2.2 Hiding State

While separating modules and clients via existentials and using separation to distinguish distinct objects is obviously useful, for imperative programs this is not entirely a sufficient mechanism on its own.

In a purely functional program, all communication between an implementation and a client of a module is mediated by the signature of the module. The addition of state makes it possible to create an extra channel of communication through the heap, and in our formalism this is captured by adding pre- and post-conditions to the specifications of imperative actions.

However, suppose we have a client program with specification C , which makes use of a particular module implementation with a specification S . Furthermore, suppose that *both* the client program implementing C , and the module implementing the S interface, rely upon yet another module with interface F . As an example, imagine an F that has a purely functional interface, but which makes use of an imperative cache for efficiency reasons (for example, to implement memoization). Now, even though there is aliased state between C and I , they cannot interfere with one another through it.

Therefore, a properly modular specification *should not require* S to mention the use of the caching module. Otherwise we will be forced to mention more and more extraneous details as we layer modules on top of one another, with the total size of the specifications growing with the size of the reachable module graph.

The language I am proposing is able to properly describe this situation, thanks to our ability to use the frame rule. Concretely, suppose we have the following interface for our third module.

$$\forall n : \mathbb{N}. \{cache\} \mathbf{fib}(n) \{a : \mathbb{N}. cache \wedge a = fibonacci(n)\}$$

Here, we can imagine \mathbf{fib} produces fibonacci numbers using the naive exponential time recursive implementation, but adds caching to “optimize” it to linear time. The $cache$ predicate appears in the pre- and post-conditions of \mathbf{fib} , so at first it would seem that a C and S that used this function would have to mention that they require the heap associated with $cache$.

However, note that $cache$ is the same in both the pre- and post-condition. This means that we can prove C and I using a specification

$$F \equiv \{\mathbf{emp}\} \mathbf{fib}(n) \{a : \mathbb{N}. \mathbf{emp} \wedge a = fibonacci(n)\}$$

and then we can frame $cache$ on at the very end, after completing the proofs of C and I . Concretely, the proof tree looks like this:

$$\frac{\frac{\vdots}{\vdash F \otimes cache} \quad \frac{\frac{F \vdash S \quad F, S \vdash C}{F \vdash C}}{F \otimes cache \vdash C \otimes cache} \text{FRAME}}{\vdash C \otimes cache} \text{CUT}$$

Here, I write $R \otimes p$ to suggest that p is framed onto the triples in the specification R . Note that in the subderivation above the use of the Frame rule, we do not need to carry around *cache* in our specification of `fib`, which means that when we introduce S , we will not need to mention *cache* in its specification.

Simple caching schemes are relatively easy to hide – all we need is to make use of the Frame rule [19, 4] to remove the cache state from a specification. A stronger test of our ability to hide irrelevant detail from specifications will be if we can hide Liskov and Wing’s *monotonic constraints* [15].

Monotonic invariants are program invariants that hold monotonically throughout a program’s execution. Suppose we have a partial order (S, \sqsubseteq) and a predicate P , such that if $s' \sqsubseteq s$ then $P(s') \supset P(s)$. Now, if we then ensure that any triples that mention P are of the form $\{P(s)\}c\{a : \tau. \exists s'. P(s') \wedge s' \sqsubseteq s\}$, then we know that no calls to this module can break our invariant, since they can only make P “more true”. So it follows that this, like a cache, is potentially information we should not necessarily be forced to track in pre- and post-conditions.

This property is used in several verification methodologies to prove programs with potentially unknown calls [14], but using it for information hiding purposes may be novel.

Related to monotonic effects are *commutative effects* [16]. These are side-effects in which the order they happen in does not affect correctness. For example, a gensym operation is a commutative effect, since all we care about is the fact that two different invocations return distinct symbols.

Finally, it would be interesting to see whether and how we can encode rely-guarantee arguments [11, 29] to reasoning about imperative programs. Rely-guarantee reasoning works by defining a symmetric relation on the shared state of a program, and showing that the shared state in the pre- and post-conditions of each sub-program fall into this relation. It was originally invented to reason about concurrent programs, which can interleave the execution of a module and its client. Vafeiadis and Parkinson [27] have integrated this technique with separation logic to reason about concurrent imperative programs, and Bierhof has observed that it is also useful for verifying sequential programs [?]. Intuitively, it seems that modularity seems to induce an abstract form of concurrency, since we wish to deliberately abstract away from how different modules manipulate some shared state.

5 Interesting Higher-Order Imperative Programs

Validating my thesis means that I have to demonstrate that my system works on interesting imperative programs. So what do I mean with the word “interesting”? There are several senses in which I use it:

- One kind of interesting program are programs that pose particular difficulties for verification, such as “Landin’s knot”, which implements recursion by backpatching a function reference. These kinds of procedures are typi-

cally small, and may not be found in actual software very often¹, but help demonstrate the capabilities and limits of a verification methodology.

- A second kind of interesting program are example programs that capture a particular stylized patterns of use that arise frequently in practice. Design patterns [9] are a good example of this kind of program – they represent what might be called engineering wisdom, and formalizing these patterns can help demonstrate that our verification methodology can accommodate typical good design, and moreover its formal version is not overly cumbersome to use.
- A third kind of interesting program are big programs. Even if a verification methodology works successfully on small examples, it remains to be seen whether it will work on larger examples, because in a larger example we will have to show that the different patterns of use can coexist peacefully, and that they compose gracefully.

Also, I only mean “big” relative to the size of typical examples (which are usually in the tens of lines of code). I will be happy with verifying a program that is on the order of high hundreds to low thousands of lines of code. This is, of course, very small by industrial standards, but it is large enough to learn whether or not the different pieces of this methodology fit together.

- A fourth kind of interesting programs are strongly higher-order programs that nonetheless use state. For example, consider parsing combinators [?] that use state to track the parser state, or higher-order functions that abstract traversal orders (e.g., general depth- or breadth-first-search combinators) and use state to track visited nodes.

5.0.3 Design Patterns

I will begin by analyzing individual design patterns, giving a specification, an implementation, and a small client program for each of the design patterns with a substantially stateful part.

The intuition here is that 1) object-oriented software is the biggest repository of higher-order imperative code out there, and 2) the Gang of Four design patterns are well-respected engineering wisdom about how to manage such designs. So if I can show that I can formalize that practice, I’ll have taken a big step towards showing I can formally capture the sorts of reasoning that software engineers use to manage such programs.

Furthermore, there’s also a *negative* component here. I want to figure out what patterns of reasoning are difficult to capture in my framework, and systematically analyzing the design patterns should help me search the design space.

¹Recently, I found a natural use of this technique in a real program, so I do not mean “not very often” as a polite way of saying “never”.

One further thing that's worth noting is that a design pattern *isn't* a particular formula of specification logic. Any specification we write represent particular balances between generality (that is, more possible implementations) and specificity (that is, more invariants for a client to use). For example, suppose we have an iterator specification which supports deletion. For some collection types (such as stretchable arrays), deleting an element invalidates only those iterators that are further along the array than the one that did the deletion. If we put this fact into the specification, then clients can use this fact to not invalidate some iterators. But the price is that this specification rules out collection implementations such as binary trees, where any deletion will invalidate all other iterators.²

Furthermore, each relevant design pattern will allow me to analyze an important class of modularly designed programs that use higher order state. We can take each of the major features of our logic – for example, higher-order frame rules, existentials, separation – and study whether and how each feature is necessary in order to prove the correctness of a program in a clean, elegant style.

The necessity of a feature can be explored by seeing what becomes more difficult when we try to prove a program *without* using that feature.³

- *Flyweight*

The basic idea here is similar to memoization: cache the construction of (immutable) objects, and on an object request only return a new one if a satisfactory one is not in the cache. The cache, of course, is a global mutable data structure that *we don't want to see* in the proof of any clients. This makes it a good use case for the frame rule, and the issues that arise here will be a good test of whether we can actually scale proofs.

- *Memento*

This pattern is commonly used to support undo. It's basically an opaque token representing an object's old state, and the object has a method that the client can use (with the memento) to reset the state. We can implement this with higher-order state – the token can be a command the user can invoke, and there are some nice sequencing issues in when you can invoke the undo operation or not. We should also look at multilevel undo, and compare this to an internal undo operation.

- *Chain of Responsibility*

The Chain of Responsibility pattern is a pattern for composing commands from other commands. Rather than building a monolithic command, we break it up into processing units that take a message and handle the parts of the update they can, delegating the rest to the rest of the chain. This

²This means that any spec will be criticized by both those who think it is too liberal, and those who think it is too restrictive. I think this is a feature, not a bug – making assumptions explicit in a way that permits detailed argument is a big win!

³It is also possible we may discover that there are as-yet missing features in our system.

doesn't have to be a strict chain, so you can have a "tree of responsibility" as well.

There's probably a very slick specification of this in terms of state transformers, and I think there might be a nice ownership transfer idiom lurking in here that's worth unearthing.

- *State*

This is a pattern in which an object is made from other objects which provide abstract state – it does not know the precise implementation of its components. In fact, it's okay for the implementation of the concrete state to *change* as the program runs. Verifying this example is primarily useful as a diagnostic test; of course we can handle it, but even the slightest hint of ugliness in our development indicates trouble in the logic.

- *Composite*

At first, I thought that the Composite pattern didn't have any interesting issues, but then I realized I was wrong, because we are often composing *stateful* objects. For example, consider an iterator library (such as the `itertools` library in Python) which constructs new iterators out of existing ones.

Concretely, suppose we have `PairIterator` that takes two iterators and returns a new one that produces pairs of values, iterating over the two arguments in lockstep. This seems straightforward, until you think that this implies certain sharing constraints – in particular, the arguments have to be two *different* iterators.

So we need to be able to say something about sharing here. This may be easiest if we use multiple specifications for the same function (for example, if the two iterators come from the same underlying collection, or not).

- *Proxy*

The Proxy pattern does not have very much complexity in its implementation; it's essentially just a (possibly-imperative) wrapper function – its ML type could be `proxy : (a -> b) -> (a -> b)`. However, analyzing it will still offer some interesting evidence for or against my thesis. That's because at heart a Proxy transforms the specification of its argument, and we can examine how hard it is to model that.

Note that state can play an important in implementing a real Proxy, because it provides us with a "back channel" for communication. For example, a security Proxy that checks to see whether the owner of a piece of data is authorized before invoking a method might have a data structure it consults for that authorization information.

The Decorator pattern is similar to this, only we can add methods to our input in addition to transforming old ones.

- *Strategy*

In a language with first-class functions, there's not much to the Strategy pattern – we just need a reference to a function. But the very simplicity of this pattern means that we ought to be able to prove some fun programs, like showing the correctness of a program that updates its strategy as it runs.

- *Facade*

The Facade pattern takes some existing modules and presents a nice abstraction of them to the programmer. The way I conceptualize it, we want to compose imperative modules in a way that turns their state into a single new predicate. This is pretty straightforward in easy cases, but I'm interested in finding out if there's a way to make this *hard* I'd like to stress-test our ability to abstract over state.

In particular, the test of whether this works will be seeing how big the predicates grow. If I composing two modules with n -place predicates describing their heap yields a Facade with a $2n$ -place predicate, it's pretty clear that information hiding isn't genuinely happening. This contrasts with the Flyweight, where the test of modularity is seeing how many starred subformulas show up in the pre- and post-conditions.

I have already looked at two other patterns, the Iterator [12] and the Subject-Observer [13] pattern in detail.

5.0.4 Iterator Pattern

An *iterator* is essentially an imperative stream that enumerates the elements of a collection. This permits client programs to enumerate the elements of a collection without necessarily having to know the details of the collection's implementation. In the Java library specification [10], iterators are required to obey the following aliasing discipline:

- From a given collection, we can create as many iterators as we like.
- Each iterator can be advanced independently of all of the others.
- While there are active iterators on a collection, methods that do not modify that collection may be freely called.
- If a collection method is invoked that modifies the collection (for example, by adding or deleting an element), then all existing iterators become invalid and may not be invoked again.
- An iterator may support deletion or other modification operations on a collection. If an iterator modifies a collection, that particular iterator remains valid, but all other iterators are invalidated.

The standard library collections attempt to dynamically check conformance to these rules, raising a `ConcurrentModificationError` if it detects a violation of this protocol, but it does not promise to catch all violations.

It is worth considering what this specification looks like in our separation logic formalism. Below, we give a signature for a collection and its iterator. The types τ_c and τ_i are the collection and iterator types, respectively. (I am using them as abbreviations, rather than as true type variables.)

```

1   $\exists coll : \tau_c \times seq \times \mathbf{prop} \Rightarrow \mathbf{prop}$ .
2   $\exists iter : \tau_i \times \tau_c \times \mathbf{prop} \Rightarrow \mathbf{prop}$ .
3   $\exists newcoll : 1 \rightarrow \bigcirc \tau_c$ .
4   $\exists empty : \tau_c \rightarrow \bigcirc \mathbf{bool}$ .
5   $\exists add : \tau_c \times \mathbb{N} \rightarrow \bigcirc 1$ .
6   $\exists newiter : \tau_c \rightarrow \bigcirc \tau_i$ .
7   $\exists next : \tau_i \rightarrow \bigcirc(\mathbf{option} \ \mathbb{N})$ .
8   $\{\top\} \mathbf{run} \ \mathbf{newcoll}() \{a : \tau_c. \exists P. coll(a, \epsilon, P)\}$ 
9    and
10  $\forall c, xs, P. \{coll(c, xs, P)\} \mathbf{run} \ \mathbf{empty}(c) \{a : \mathbf{bool}. coll(c, xs, P) \wedge a = (|xs| = 0)\}$ 
11   and
12  $\forall c, x, xs, P. \{coll(c, xs, P)\} \mathbf{run} \ \mathbf{add}(c, x) \{a : 1. \exists Q. coll(c, x \cdot xs, Q)\}$ 
13   and
14  $\forall c, xs, P. \{coll(c, xs, P)\} \mathbf{run} \ \mathbf{newiter}(c) \{a : \tau_i. coll(c, xs, P) * iter(a, c, xs, P)\}$ 
15   and
16  $\forall i, c, xs, P.$ 
    $\{coll(c, xs, P) * iter(i, c, xs, P)\}$ 
    $\mathbf{run} \ \mathbf{next}(i)$ 
    $\{a : \mathbf{option} \ \mathbb{N}. coll(c, xs, P) * iter(i, c, xs, P)\}$ 
17   and
18  $\forall i, c, xs, P.$ 
    $\{coll(x, xs, P) * iter(i, c, xs, P)\}$ 
    $\mathbf{run} \ \mathbf{delete}(i)$ 
    $\{a : 1. \exists ys, Q. coll(c, ys, Q) * iter(i, c, ys, Q)\}$ 

```

This module is written as an existential specification, and provides two predicates, *coll* (representing the state of the collection), and *iter* (representing the state of the iterator). An collection state $coll(c, xs, P)$ means that we have a collection object c , representing the sequence xs , in an *abstract state* P . As we will see, the abstract state represents the “current state” of the collection. An iterator predicate $iter(i, c, xs, P)$ means that we have an iterator i over a collection c containing xs , and that the iterator i is valid when c is in the abstract state P .

The specifications of the functions in this module are read as follows. In line 8, when we create a new collection with a call to `newcoll()`, we get a new, empty collection a as a return value, in some abstract state P . In line 10, when we check that a collection $coll(c, xs, P)$ is `empty`, we see that we get a boolean

return value, and that the abstract state of a collection remains in P . However, in line 12, when we call $\text{add}(c, x)$, the collection state goes from $\text{coll}(c, xs, P)$ to $\exists Q. \text{coll}(c, x \cdot xs, Q)$, potentially changing the abstract state.

We can create a new iterator on c with a call $\text{newiter}(c)$ on line 14, which takes a collection state $\text{coll}(c, xs, P)$ and, leaving the collection unchanged, returns a new iterator $\text{iter}(i, c, xs, P)$. When we want to advance the iterator, we call $\text{next}(i)$, whose spec is on line 16. This function returns either an element of the collection or a signal the collection is exhausted with a return value of type $\text{option } N$. The collection state remains $\text{coll}(c, xs, P) * \text{iter}(i, c, xs, P)$ in the pre- and post-conditions. Note that we are not precisely tracking the position of the iterator in this specification – I am giving a deliberately imprecise specification to focus on the aliasing behavior. In line 18, we give a $\text{delete}(i)$ operation, which deletes the element at the current cursor position. Again, we do not track this precisely, choosing to focus on the aliasing behavior – and we see that when we call $\text{delete}(i)$, the abstract state of the iterator and the collection changes in sync.

As an example of this specification’s use, consider the following fragment of a client program:

```

1   {coll(c, xs, P)}
2   letv i1 = newiter(c) in
3   {coll(c, xs, P) * iter(i1, c, xs, P)}
4   letv i2 = newiter(c) in
5   {coll(c, xs, P) * iter(i1, c, xs, P) * iter(i2, c, xs, P)}
6   letv x1 = next(i1) in
7   {coll(c, xs, P) * iter(i1, c, xs, P) * iter(i2, c, xs, P)}
8   letv b = empty(c) in
9   {coll(c, xs, P) * iter(i1, c, xs, P) * iter(i2, c, xs, P)}
10  letv x2 = next(i2) in
11  {coll(c, xs, P) * iter(i1, c, xs, P) * iter(i2, c, xs, P)}
12  letv dummy = delete(i1) in
13  {∃ys, Q. coll(c, ys, Q) * iter(i1, c, ys, Q) * iter(i2, c, xs, P)}
14  {coll(c, ys, Q) * iter(i1, c, ys, Q) * iter(i2, c, xs, P)}
15  letv x3 = next(i1) in
16  {coll(c, ys, Q) * iter(i1, c, ys, Q) * iter(i2, c, xs, P)}
...

```

In line 1, we have an initial state with a collection $\text{coll}(c, xs, P)$ in it, and in lines 2 and 4, we create two new iterators i_1 and i_2 , giving us a state $\text{coll}(c, xs, P) * \text{iter}(i_1, c, P) * \text{iter}(i_2, c, P)$ in line 5. Then, in lines 6, 8, and 10, we call next on i_1 and i_2 , and empty on c , all without changing the abstract state P of either the collection or the iterators.

In line 12, we call $\text{delete}(i_1)$, deleting an element from c . This puts c in a new state $\text{coll}(c, ys, Q)$, which i_1 alone shares – i_1 has the state $\text{iter}(i_1, c, ys, Q)$, but i_2 has the state $\text{iter}(i_2, c, xs, P)$. As a result, we are no longer able to call next on i_2 – we can only call it on i_1 (on line 15).

Now, let's consider the most important parts of a simple implementation of this specification. I will now take τ_c to be a simple mutable linked list type $\tau_c \equiv \text{ref list } \mathbb{N}$, and τ_i to be a reference into the list, of type $\text{ref ref list } \mathbb{N}$. We will need some auxiliary predicates, not exported from the module as well, in order to describe lists and list segments:

$$\begin{aligned}
list(c, \epsilon) &\equiv c \mapsto \text{Nil} \\
list(c, x \cdot xs) &\equiv \exists c'. c \mapsto \text{Cons}(x, c') * list(c', xs) \\
listseg(c, c', \epsilon) &\equiv \text{emp} \wedge c = c' \\
listseg(c, c', x \cdot xs) &\equiv \exists c''. c \mapsto \text{Cons}(x, c'') * listseg(c'', c', xs)
\end{aligned}$$

We can now give specifications for the *coll* and *iter* predicates.

$$\begin{aligned}
coll(c, xs, P) &\equiv list(c, xs) \wedge P \\
iter(i, c, xs, P) &\equiv \exists c'. i \mapsto c' * \\
&\quad [coll(c, xs, P) -* \\
&\quad \quad coll(c, xs, P) \wedge \\
&\quad \quad \exists ys, zs. (listseg(c, i, ys) * list(i, zs) \wedge xs = ys \cdot zs)]
\end{aligned}$$

So a collection is just a linked list, in a particular state P . An iterator is a reference to an interior node of a linked list, such that if we are given a particular collection, we can learn that the same state can be viewed as both the collection and a list segment and tail marked by the iterator. This is the reason we need the state variable P in *coll*, rather than just the *list* predicate: modifying the tail of a list can change the link structure while leaving the list predicate $list(c, xs)$ unchanged. Conjoining an arbitrary variable prohibits this, because we can't know what features of the state its truth relies on, and hence we cannot modify the collection.

To see how this works, suppose we have a heap h modelling $list(c, xs) \wedge P$. Now, one of legitimate instance of P could be a large starred sequence of points-to assertions $l_1 \mapsto v_1 * \dots * l_k \mapsto v_k$ that exactly describes h . Since P must be preserved in the postcondition, it follows that this state must undo any changes it makes to this heap during the action of the command.

The implementations of the collection and iterator functions are straightforward, and I will not give them here.

Now, let us consider which features of our logic were needed:

- *Existential Packages*: We needed to hide the implementations of the collection and iterator operations, as well as the predicates *iter* and *coll*. So this was a simple existentially quantified package.
- *Separation*: We needed separation to distinguish different collection and iterator instances. For the implementation, this proved essential. First, we needed the magic wand to represent the idea that the iterator was complete

only when it had access to its collection, and second, we enforced the mutation property by overlapping separating and ordinary conjunction.

- *Higher Order Logic*: In addition to using existentially quantified predicates to abstract over the heap, we used predicate variables to track changes to the internal state of collection objects, without revealing that internal state to external programs.

In this example, we just tracked whether (a part of) the data structure had been mutated or not, which is a fact other approaches (such as fractional permissions [5]) could track, but it would be interesting to see if we can extend this to tracking changes in a program's abstract state. For example, updating a cache involves changes to a data structure, but does not change the logical state of a program.

- *Frame Rules*: In this example, we only needed the first order frame rule. The higher order frame rules were not needed in this example.

So we made use of a fairly large portion of our program logic in order to give a specification which satisfies all of the conditions in the Java standard library specification of iterators. This is a fairly complex aliasing protocol, and it is a little surprising how simple the specification and client program verifications are. Most of the complexity lies in the iterator invariant, which an external client will never have to (or even be able to) examine.

5.1 The Subject Observer Pattern

In imperative programs, it is frequently the case that the invariant of some data structures depend on the state of some subject object. As the other object changes, it is necessary to update the observers, so that their invariants continue to hold. The subject-observer pattern [9] is a design pattern for ensuring that the invariants of a collection of *observer* objects remain synchronized with the state of a *subject* object.

The basic technique works as follows: each observer object registers an update function with the subject object, which stores the callbacks in a list. Whenever the subject changes, it will iterate over the list, calling the update functions one by one. This ensures that all of the registered observers are properly updated whenever the subject changes. We can give a specification of this as follows:

```

1    $\exists \text{sub} : \tau_s \times \mathbb{N} \times \text{seq} (\mathbb{N} \rightarrow \circ 1 \times \mathbb{N} \Rightarrow \text{prop}) \Rightarrow \text{prop}.$ 
2    $\exists \text{register} : \tau_s \times (\mathbb{N} \rightarrow \circ 1) \rightarrow \circ 1.$ 
3    $\exists \text{broadcast} : \tau_s \times \mathbb{N} \rightarrow \circ 1.$ 
4    $\forall f : \mathbb{N} \rightarrow 1, O : \mathbb{N} \Rightarrow \text{prop}.$ 
5      $(\forall m, n : \mathbb{N}. \{O(m)\} \text{run } f(n) \{a : 1. O(n)\})$ 
6     implies
7      $\forall s, x, os. \{ \text{sub}(s, x, os) \} \text{run } \text{register}(s, f) \{ a : 1. \text{sub}(s, x, (f, O) \cdot os) \}$ 
8     and
```

```

9    $\forall s, x, \vec{f}_i, \vec{O}_i, \vec{x}_i.$ 
10   $\{ \text{sub}(s, x, (f_i, O_i)) * O_1(x_1) * \dots * O_k(x_k) \}$ 
11   $\text{run broadcast}(s, y)$ 
12   $\{ a : 1. \text{sub}(s, y, (f_i, O_i)) * O_1(y) * \dots * O_k(y) \}$ 

```

In line 1, we assert the existence of the *sub* predicate, which is the state associated with a subject object. In the predicate $\text{sub}(s, n, os)$, s is the subject object, n is the value that observers are tracking (here, a natural number), and os is a sequence of pairs of functions and predicates. Each pair (f, O) in os is a state predicate O , with f a function that updates the state O .

In lines 2 and 3, we assert the existence of two functions **register** and **broadcast**. The function **register** (s, f) adds the function f to the sequence of callbacks stored in the subject, as specified in lines 4-7. Notice that we use specification-level implication to constrain the behavior of f – we are only allowed to believe that **register** works correctly, if we have an f and O , such that $\{O(m)\} \text{run } f(n) \{a : 1. O(n)\}$ holds. So if f , given n as an argument and $O(m)$ as a pre-condition, produces the post-condition $O(n)$, then we can conclude that registration works.

The specification of **broadcast** in lines 10-12 is fairly simple. If we have a subject state $\text{sub}(s, x, os)$, and the appropriate observer state $O_i(x_i)$ for each element of the sequence of listeners, then a call **broadcast** (s, y) will put both the subject and all of the observers in the state y .

Note that this particular specification captures a specific set of assumptions about how the subject-observer pattern should behave. In particular, this specification demands that the update functions passed to **register** are idempotent – even if they are called multiple times, they should yield the same result state. Also, it requires that the same observer value not be registered multiple times; if a user does do this, then it will be impossible to invoke **broadcast**, because it requires that all of the observer states are disjoint from one another. This separation also means that observers cannot communicate via the callback functions.

Variations on all of these restrictions are found in practice, and can be weakened or strengthened according to the needs of a particular application. But this illustrates the point that design patterns are categories of specification, and that we may need to vary the details of a specification to reflect those variations.

The primary interest of the subject observer pattern lies, as usual, in the definition of the *sub* invariant. We will take τ_s to be a pair of a reference to a natural number (the observed state), and a list of update functions, $\tau_s \equiv \text{ref } \mathbb{N} \times \text{list } (\mathbb{N} \rightarrow \bigcirc 1)$. Then, we can define the *sub* below:

$$\begin{aligned}
\text{Good}(f, O) &\equiv (\forall m, n : \mathbb{N}. \{O(m)\}\text{run } f(n)\{a : 1. O(n)\}) \text{ valid} \\
\text{GoodSeq}(\epsilon) &\equiv \top \\
\text{GoodSeq}((f, O) \cdot os) &\equiv \text{Good}(f, O) \wedge \text{GoodSeq}(os) \\
\text{sub}(s, n, os) &\equiv (\text{fst } s \mapsto n) * \text{list}(\text{snd } s, \text{map } \pi_2 os) \wedge \text{GoodSeq}(os)
\end{aligned}$$

The *Good* predicate makes use of another feature of our logic: the ability to embed specifications in assertions. The assertion $S \text{ valid}$ is an *assertion* that the specification S is true. (There is also a corresponding operator that embeds an assertion in the specification language.) This lets us make assertions about the behavior of computational values within our assertions. In this case, $\text{Good}(f, O)$ is an assertion that f satisfies the specification $\forall m, n. \{O(m)\}\text{run } f(n)\{a : 1. O(n)\}$. The $\text{GoodSeq}(os)$ predicate lifts *Good* to sequences of these pairs, so that every element of os satisfies the *Good* predicate.

The addition of the $S \text{ valid}$ assertions allows us to embed specifications into the assertion logic. In some cases, this lets us avoid the use of implication in some cases – any valid specification of the form $S \text{ implies } \{P\}c\{a : \tau. Q\}$ is equivalent to one of the form $\{P \wedge S \text{ valid}\}c\{a : \tau. Q\}$. However, I do not know if it is always possible to remove hypothetical specifications.

With this predicate, we can define $\text{sub}(s)$ very simply. The first component of s points to the tracked natural number, and the second component is a linked list of the functions in os .

- *Specification Logic:* As before, we to hide the implementations of the **register** and **broadcast** operations, as well as the *sub* predicate. So this was a simple existentially quantified package. However, this interface also featured the higher-order function **register**, so we also needed the ability to use specification-level implication to constrain the behavior of the functional argument.
- *Separation:* Here, we made use of separation to keep the different observer states disjoint from one another, and to ensure that within an implementation, the subject and observer states were all disjoint from one another. Overall, the spatial properties of the subject predicate was simpler than the iterator’s internal invariant.
- *Higher Order Logic:* In addition to using existentially quantified predicates to abstract over the heap, we needed to use sequences of predicates to track all of the invariants of the individual observer objects. This is more complex than in the case of the iterator – we also needed to use the modal operator that embeds specifications into the assertion language.
- *Frame Rules:* In this example, we only needed the first order frame rule. The higher order frame rules was not needed in this example. I think that in general, the higher-order frame rule will come into play only when there is some global state that needs to be abstracted out.

5.2 Larger Programs

Proving that I can prove the correctness of a wide variety of design patterns is nice, but design patterns are a means, not an end. The question remains whether these techniques can scale up to a larger program. There are two main reasons that a larger program might prove more difficult:

1. A *formula* of moderate size is much smaller than a *program* of moderate size; so we may see a larger gap between the structure of a program's invariants and the structure of its code in larger examples. I suspect this is a result of the fact that an interesting program will tend to use a sophisticated implementation to realize its specification. For example, an optimization pass for a compiler works because the code implements a fixed point iteration justified from various facts in lattice theory, but its actual implementation will typically be some kind of imperative worklist algorithm that pushes elements on and off a queue.
2. Secondly, we might discover that we can specify the components of a program (e.g., design patterns), but that composing them is difficult, because of unforeseen interactions between different aspects. For example, suppose we have two modules which each depend on a common (stateful) module. An important question is whether their specifications *necessarily* reveal that dependence in a way that causes specification blow-up. Or alternately, suppose we have a data structure that implements multiple design patterns. Under what conditions can we avoid (or not avoid) interactions between those different facets of an API?

So to validate my thesis, I should look for a program that exhibits both of these features – we want a program with a high level specification divorced from the structure, and which is systematically constructed from smaller components. I am currently considering two possibilities for such a program.

5.2.1 A Graphical User Interface Library

The main possibility is to implement and prove correct a graphical user interface library. This is attractive for several reasons.

First, the structure of typical GUI code is quite awkward, in *precisely* a higher-order imperative way. A GUI program is structured as a set of callback functions which are executed by an event loop, which invokes each callback whenever an event of interest to it happens. All communication between callbacks happens via the heap. This is a bit like converting a program to continuation-passing style and then representing the continuation as heap data the programmer must manually modify. Such code is notoriously difficult to write and debug, and many patterns such as the model-view-controller pattern have been proposed to help tame the complexity.

The next question is what we might mean by proving the correctness of a GUI. We can only prove a program correct with respect to a specification, and

so we need a specification of a GUI. Here, I propose adapting a simplified version of the work of Courtney [6], on denotational semantics for GUI libraries. He gave a semantics in terms of synchronous dataflow, where each GUI component is a signal transformer that takes an input signal of events and transforms it into an output signal of window layouts.

Below, I give partial ML-style datatypes describing events and layouts, and then define two simple GUI widgets as signal transformers, in a purely functional style.

$$ST(A, B) = (\text{Time} \rightarrow A) \rightarrow (\text{Time} \rightarrow B)$$

$$GUI(A, B) = ST(A \times \text{Event}, B \times \text{Layout})$$

$$\begin{aligned} \text{Layout} &= \text{Label of String} \mid \text{Button of Unit} \mid \\ &\quad \text{Stack of Layout} * \text{Layout} \mid \dots \end{aligned}$$

$$\text{Event} = \text{Click of Unit} \mid \text{Key of Char} \mid \dots$$

$$\begin{aligned} \text{label} &: GUI(\text{String}, \text{Unit}) \\ &\equiv \lambda \text{input} : \text{Time} \rightarrow (\text{String} \times \text{Event}). \\ &\quad \lambda t : \text{Time}. \\ &\quad \text{let } (s, e) = \text{input}(t) \text{ in} \\ &\quad (\text{Label}(s), ()) \end{aligned}$$

$$\begin{aligned} \text{button} &: GUI(\text{Unit}, \text{Bool}) \\ &\equiv \lambda \text{input} : \text{Time} \rightarrow (\text{Unit} \times \text{Event}). \\ &\quad \lambda t : \text{Time}. \\ &\quad \text{let } (_, e) = \text{input}(t) \text{ in} \\ &\quad \text{case } e \text{ of} \\ &\quad \mid \text{Click}() \Rightarrow (\text{true}, \text{Button}()) \\ &\quad \mid - \quad \Rightarrow (\text{false}, \text{Button}()) \end{aligned}$$

$$\begin{aligned} \text{stack} &: GUI(A, B) \times GUI(X, Y) \rightarrow GUI(A \times X, B \times Y) \\ &\equiv \lambda (w_1, w_2) : GUI(A, B) \times GUI(X, Y). \\ &\quad \lambda \text{input} : \text{Time} \rightarrow A \times X. \\ &\quad \lambda t : \text{Time}. \\ &\quad \text{let } (b, l_1) = w_1 \text{ input } t \text{ in} \\ &\quad \text{let } (y, l_2) = w_2 \text{ input } t \text{ in} \\ &\quad ((b, y), \text{Stack}(l_1, l_2)) \end{aligned}$$

The `label` widget takes a `String` input signal and at each time step produces a `Label` value carrying that string. Likewise, the `button` widget produces a boolean output signal, registering whether the button has been clicked on this time step or not. The `stack` widget takes two widgets, and combines them into a new, larger widget which multiplexes the inputs and outputs.

We can also specify various operations to combine widgets and compositionally construct a GUI. For example, two signals $ST(A, B)$ and $ST(B, C)$ can

be composed into a result $ST(A, C)$, similarly to function composition. More exotically, we can take a signal transformer of type $ST(A \times X, B \times X)$ along with an element of X , and construct a feedback loop of type $ST(A, B)$.

This functional-dataflow specification style is quite different from an implementation, which would have a single, mutable layout that gets updated by a family of callbacks on each trip through the event loop. So this will let us test the claim that we can prove programs with a reasonable gap between the specification and the implementation.

Additionally, we would want to respect the modularity constraints within the implementation – the callbacks and the event loop should of course be oblivious to one another’s concrete implementations, and the callback code should be structured in proper MVC style. This will let us test the idea that we can prove a program modularly, out of modularly-proven sub-components.

The way we will do this is to assume an imperative module for drawing on the screen, and another for imperative I/O. The window state will be a module whose state is a predicate of the form $Window(w, layout)$, where the predicate and handle w is data representing the *layout*. Then, we’ll show that if the GUI state is a dataflow specification g at time t , then invoking the event loop will take the window state from time t to $t + 1$.

```
{GUIstate(g, t) * Window(w, g inputs t) * IO(io, t, inputs)}
eventloop()
{a : 1. GUIstate(g, t + 1) * Window(w, g inputs (t + 1)) * IO(io, t + 1, inputs)}
```

6 “Feasible”

6.1 Syntactic Issues

First, there’s a simple technical question of how we write down proofs in this formal system. Writing down proof trees is too cumbersome to actually use for two reasons. First, the contexts must be copied again and again in the premises, and second, the branching of the proof tree will get too wide to lay out.

This problem arises both in ordinary logic, and in Hoare logic. In ordinary logic, this problem is resolved with a Fitch style presentation of natural deduction [8]. The basic idea is to use scoping to implicitly build and track the context of hypotheses. A proof of a formula like $((X \supset B) \wedge (Y \supset B)) \supset (X \vee Y) \supset B$ could be written as:

```
1   Assume  $(X \supset B) \wedge (Y \supset B)$ 
2      $X \supset B$  by And-elimination on 1
3      $Y \supset B$  by And-elimination on 1
4     Assume  $X \vee Y$ .
5       Analyze 4:
6       If  $X$ , then
7          $B$  by Imp-elimination on 6, 2.
```

8 Otherwise Y :
9 B by Imp-elimination on 8, 3.
10 In either case, B
11 Thus $(X \vee Y) \supset B$
12 Thus $((X \supset B) \wedge (Y \supset B)) \supset (X \vee Y) \supset B$

Observe that in this style, we use scoping (in this example, marked with indentation) to implicitly construct and carry around the context, instead of explicitly passing around the context, as in the following proof tree:

$$\begin{array}{c}
\frac{X \vdash X \quad B \vdash B}{X \supset B, X \vdash B} \text{IMPL} \quad \frac{Y \vdash Y \quad B \vdash B}{Y \supset B, Y \vdash B} \text{IMPL} \\
\hline
(X \supset B), (Y \supset B), (X \vee Y) \vdash B \quad \text{ORL} \\
\hline
(X \supset B) \wedge (Y \supset B), (X \vee Y) \vdash B \quad \text{ANDL} \\
\hline
(X \supset B) \wedge (Y \supset B) \vdash (X \vee Y) \supset B \quad \text{IMPI} \\
\hline
\vdash (X \supset B) \wedge (Y \supset B) \supset (X \vee Y) \supset B \quad \text{IMPI}
\end{array}$$

Even in an example this simple, we have to discard unused assumptions in many of the steps, simply to fit the proof on the page.

Similarly, with Hoare logic we also have the question of how to write programs in a non-sequent style, in order to avoid redundantly repeating assertions and program fragments in the proof. The solution in this case is to write the program in conventional style, but to annotate each statement with its pre- and post-condition. The post-condition of one statement becomes the pre-condition of the next, so these assertions are not repeated and duplicated throughout the program. An annotated program might look like this:

```

{P}
letv x = e in
{Q}
{Q'}
letv y = e' in
{R}
c
{S}

```

which would correspond to the following proof tree:

$$\frac{\frac{\{P\}\text{run } e\{Q\} \quad Q \vdash Q'}{\{P\}\text{run } e\{Q'\}} \quad \frac{\{Q'\}\text{run } e'\{R\} \quad \{R\}c\{S\}}{\{Q'\}\text{letv } y = e' \text{ in } c\{S\}}}{\{P\}\text{letv } x = e \text{ in letv } y = e' \text{ in } c\{S\}}$$

Reconstructing a proof tree from an appropriately annotated program is generally a relatively straightforward task, because the proof tree follows the abstract syntax of the program, with extra nodes for the rule of consequence and the frame rule.

As the system I am proposing embeds Hoare logic within a system of natural deduction, I will need to design a syntax that combines the virtues of annotated programs and of scoped natural deduction. This is necessary to make program proofs manageable, both for reading and for writing. I expect that large portions of my actual thesis will be program proofs, and so having a clear syntax will be essential.

Additionally, I will need to give proofs of entailment within separation logic. One possibility is to adapt Beans’s [?] *ribbon proofs*, which are a generalization of Fitch-style natural deduction to handle the branching structure of bunched logic [?], the logic underlying separation logic. I have also developed a variant natural deduction that is sound with respect to separation logic, and does not use bunches – instead, it is a labelled deduction system [?] in which the labels are monoidal annotations that control when hypotheses can be safely used.

Finally, our system also supports a powerful notion of equality, and so we will have to give equality proofs to justify the equational reasoning of the functional parts of the language. I believe that it should be possible to adapt the Bird-Meertens, or “Squiggol”, style of equality reasoning [?] for this purpose. As is the case with specifications, it is not yet clear to me how to integrate this form of proof with Hoare-style proofs.

6.2 Checking Proofs

Once a proof is given, an obvious question is whether the proof is actually correct. Especially as the proof grows large relative to the program, it can easily be the case that a bug lurks in the proof, thereby giving false assurance.

I plan on addressing this issue by writing a proof checker for my system. My idea is to formalize all of the syntax and inference rules of my system in Twelf [?], and then to translate my program proofs into Twelf terms. Typechecking the Twelf term will ensure that all of the rules have been correctly applied, and that my purported proof actually forms a legitimate derivation in my system.

To be clear, I emphatically do not plan on doing any of the metatheory of the specification logic in Twelf. The semantics of the system and proofs of the soundness of all of the rules will be done by hand, on paper – the checker will only verify that a proof actually is a legitimate derivation in my system.

7 Related Work

7.1 Separation Logic

Separation logic [26] was introduced by O’Hearn and Reynolds in order to simplify the treatment of reasoning about aliasable, mutable data structures. The

central idea was to extend the assertion language with some spatial connectives – the separating conjunction $P * Q$, its adjoint implication $P \multimap Q$, and a points-to relation $e \mapsto e'$ – in order to track whether data structures aliased or not implicitly, rather than explicitly. Beyond the direct advantage in terms of smaller assertions, these new connectives also permitted the statement of the *frame rule*, which captures the idea that any state not touched by a command will remain unchanged. Together, this permits a flexible and modular form of reasoning about mutable state.

O’Hearn *et al* [19] added procedures to the language, and discovered the second-order frame rule, which allows hiding module-level data from a client program in a program proof. Birkedal and Yang [4] generalized the frame rule to arbitrary order, in the context of Idealized Algol. This language supports higher-order procedures, and still has the first-order heap of traditional separation logic.

Biering and Birkedal [2] investigated higher-order separation logic, a generalization of separation logic in which quantifiers are permitted to range over assertions as well as pure terms. As we have already seen, this is extremely useful for talking about abstract data types, since it allows us to talk about the state associated with an abstraction without revealing its concrete representation.

(Who invented the admissible closure trick? Nick? Lars? Hongseok?)

7.2 Specification Logic

Reynolds developed specification logic [25] in order to specify and prove programs written in Idealized Algol, a block-structured, call-by-name programming language with an integer store. In particular, Algol supported higher-order procedures, in which commands could be passed as arguments. The key idea that enabled specifying and reasoning about such procedures was to make Hoare triples into the atomic formulas of a logic, so that a higher-order command could be specified using an implication. (That is, if we have $f \text{ cmd}$, we could give a specification of the application that is conditional on the behavior of cmd – we can say “if cmd has a particular behavior, then $f \text{ cmd}$ will have this other behavior”.)

This is an idea I have carried forward in my system. It can be seen as a simplification of specification logic, taking advantage of the fact that idealized ML, unlike idealized Algol, does not have mutable variables. This means that all aliasing happens in the heap, where we can use separation logic to reason about its behavior. This allows us to dispense with all of the “good variable” conditions in specification logic. Secondly, we take advantage of our monadic language to place all side-effects, including non-termination, within the monad. Making the functional part of the language terminating means that we do not have to have well-foundedness assertions that ensure that any expressions occurring within assertions terminate.

7.3 Hoare Type Theory

Nanevski, Morrisett, and Birkedal [18] have developed a dependent type theory called Hoare Type Theory, whose central feature is the idea of refining a monadic type of computations with predicates describing its pre- and post-conditions.

The semantic structure of the system is similar to, but substantially more complex than, this work at a conceptual level – it might be said that Hoare Type Theory is what you get when you integrate the specification language into the type system via the addition of dependent types and polymorphism, or conversely that my system is what you get when the specifications are separated from the types and dependency erased.

As far as I can tell, the main difference between these two systems is a difference in intent. I intend to do program proofs directly in this logic, whereas Hoare Type Theory is intended to be the proof term language that a tactical theorem prover targets. As a result, they bias their design towards making automatic checking possible, whereas I bias the design towards making proofs short. For example, the definitional equality of HTT does not include eta-rules, whereas my equality does include all the extensionality principles.

7.4 Separation Logic for Java

Matthew Parkinson [20] has developed a version of separation logic for a large subset of Java. Since Java (like SML) allows imperative effects anywhere, his assertion language is restricted to field accesses and arithmetic, essentially the pure fragment of Java. His language also makes use of second order quantification, allowing information hiding by hiding data representations.

A fascinating feature of Parkinson’s development is that *predicates* also engage in “dynamic dispatch”. It is possible to define a version of a predicate for each class, and the meaning of a predicate symbol depends on the class of its receiver argument. It would be very interesting to study how to simulate this behavior with a more conventional notion of predicate.

7.5 Yoshida, Honda, Berger

Berger, Honda, and Yoshida [1] have developed a program logic for reasoning about the total correctness of imperative functional programs with state. They developed this logic by considering a translation of stateful PCF into the pi-calculus, and then specializing Hennessey-Milner logic to the image of their translation. HM logic is a modal calculus for reasoning about labelled transition systems. This heritage is revealed in the “content quantification” operations of their system. These are families of modal operators (corresponding to box and diamond) which are indexed by the set of locations that a formula might dereference.

Their system also has a form of nesting of Hoare triples in assertions. This is essential, because they construct their logic essentially by translating the operational semantics of imperative PCF into the pi-calculus, and in this language

each subexpression returns a value and updates the store, and so they must be reasoned about in a sequential style. In my system, in contrast, we use a monadic type to explicitly sequence all imperative operations and to isolate them from pure computations. So at the price of a more complex model, we end up with a much richer equational theory.

The comparison is complicated by the fact that my logic is focused on partial correctness, and their logic on total correctness. In the context of traditional Hoare logic, this is not an enormously large difference, but with higher-order store matters become much more complex, since the question of how to compare two heaps (which may contain code!) is now quite tricky.

7.6 JML, Spec# and Boogie

Boogie is a verification methodology for object-oriented programs similar to JML. It achieves soundness and modular verification through the notion of *object invariant*, which adapts the idea of loop invariant to a whole object: the object invariant describes an invariant on the whole object's state, which is guaranteed to hold on entry and exit to each method in the object's class.

Complexities arise due to re-entrant calls (that is, when a method is called recursively on an object, when its invariant might not hold), and because of aliasing between objects (that is, an object O might be part of the representation of another object O' , and yet a third party might call a method on O). To address these issues, the ownership discipline is adopted which only permits objects to be seriously modified by their owners. This permits users to calculate sound frame conditions – that is, a user can deduce that certain formulas can be soundly added to both pre- and post-conditions via the rule of conjunction. (In separation logic, the use of the separating conjunction allows adding arbitrary formulas R with the separating conjunction.)

7.7 Linear Types and Typestates

There has been a long history of using substructural logics to control aliasing. In particular, linear logic has also been widely used to reason about aliased data structures. Mandelbaum, Walker and Harper [?] developed a system that distinguished between pure and computation terms, in a style similar to a monadic language (albeit computations were not internalized with a monad). Computation types were then refined, similarly to Hoare type theory, with a language of refinements that were drawn from linear logic. This system allowed checking that clients correctly obeyed the usage protocols of linear logic.

Linearity is a very restrictive discipline for state, since many uses of state involve aliasing. Boyland introduced fractional permissions [5] as a technique for systematically reasoning about shared data. The idea is that a linear resource could be broken up into fractions, and the fractions distributed to different parts of the program. As long as none of the fractions modified the data, there could be no interference, and tracking the fractions also meant that they could be recombined to regain a linear resource, and hence the right to modify it.

Bierhof and Aldrich [?] have integrated these two lines of research with work on object typestates [7], in order to design a flexible type system capable of tracking fractional aliases in the style of Boyland, and checking protocol conformance, in the style of the effective refinements work, and in addition can check implementation conformance as well.

However, all this work is focused on lightweight, automatically checkable properties, and as a result this cannot express full specifications as we can (though of course we must add the effort of doing a proof). It seems likely that many of these could be expressed as very stylized patterns of proof in higher-order separation logic, with the exception of anything having to do with permissions, which can guarantee that a piece of state is never changed, rather than the weaker fact we can prove, that a piece of code never makes a net change.

8 Proposed Work and Timeline

The proposed work in this section is a superset of what I actually will do for my thesis. I am suggesting more things than I will do, in order to give my committee some choices about what they would like me to focus my efforts on. First, I'll discuss the remaining theoretical work to be done, and then I'll revisit the possible empirical work.

There are four or five pieces of theory that I would like to work out.

Second, I would like to take a second look at admissibility of predicates. Right now, the meaning of a Hoare triple $\{P\}c\{a : \tau. Q\}$ is not quite that beginning c in a state P will leave us in a state Q should c terminate. Instead, it means that running c will leave us in a state that is in the *admissible closure* of Q . This is necessary to prove the partial correctness of the recursion rule, for example.

However, this treatment is not sufficient for all the programs we might wish to write. For example, Landin's knot (that is, implementing recursion via back-patching) seems to require a closure over the precondition as well⁴, and this makes me think that it would be interesting to take admissibility out of the definition of triples, and turn it into a proper modality of our logic, so that for any proposition P , there would also be a proposition $Adm(P)$, which denotes the closure of P . Then we could reason about admissibility explicitly, rather than keeping it implicit.

This might take one to two months to work out.

Third, I need to properly study the interaction of the two logics of my specification language. We have embedded assertions in specifications via the specification $\{P\}$, and we have embedded specifications in assertions via the assertion S valid. These languages have connectives that are similar, but not identical, and I need to study their interaction. This is quite subtle; for example, the assertion S_1 valid \vee S_2 valid entails the assertion $(S_1$ or $S_2)$ valid, but the converse entailment does not hold. I hope that this study will supply me with

⁴It also requires some other technical conditions I am omitting here.

a principled reason for choosing between classical and intuitionistic separation logic for the assertion language.

Likewise, the presence of both nested triples and specification logic add a certain degree of redundancy to the logic; a provable specification of the form S implies $\{P\}c\{a : \tau. Q\}$ is equivalent to the specification $\{S \text{ valid} \wedge P\}c\{a : \tau. Q\}$. However, while I do not believe it is always possible to “compile away” either the modality or the nested specifications (in either direction), neither do I have a proof of this fact yet, nor do I have pragmatic arguments about when to use either form.

I am planning on spending two months working on this. It could well be done sooner, but I hope this research will help settle an open design question: how to choose between classical and intuitionistic separation logic. I have used classical separation logic, simply because its semantics are slightly more familiar. However, both choices seem to work (indeed, an earlier version used intuitionistic separation logic) and I would like some reason to prefer one version over the other.

Fourth, I would like to add polymorphism to our programming language. I am particularly interested in adding existential types to the language, in order to add a simple form of modules to the programming language. This will let us examine how traditional modularity mechanisms such as type abstraction interact with abstraction over the heap. I have no plans, however, on extending the language with rules for reasoning via parametricity (“theorems for free”), as in Abadi-Plotkin logic [22, 28].

I estimate this could take one month to do. Petersen [?] has studied this question in the context of Hoare type theory, and Birkedal, Mogeburg and Petersen [?] have written a report summarizing the necessary theory, so the path seems relatively clear. It will involve redoing a fair amount of the theory – I will need to build PER models over an untyped version of this language, and while that does not seem difficult there will be many conditions that will need to be checked.

In addition to this theoretical work, I am also planning on evaluating the logic by doing a series of case studies.

First, I will study some small, complex programs to verify, chosen to stress various theoretical properties of my system. An example of this class of program is “Landin’s knot”, which implements recursion via mutating a function reference. This will force me to think carefully about issues of admissibility. Another example is the union-find algorithm, which is first-order, but whose specification can vary depending on whether its arguments are aliased or not. This will let me test how much more difficult the absence of the rule of conjunction makes program proof.

Second, I will specify and verify the interesting design patterns, which I have discussed above. This will let us see how this logic actually works when verifying popular patterns of modularization. I expect that this will take about two to three months to do, assuming it takes me about a week to figure out a good account of each pattern.

Furthermore, in addition to verifying each design pattern on its own, it will

also be interesting to study combinations of the design patterns. I don't think it is worth looking at all the $O(n^2)$ combinations, so I will consult some experienced programmers to find out which combinations are typical, and study those. I will also focus on the combinations that arise in a GUI library, to prepare for the major case study of my thesis. I would like to spend up to three months thinking about these issues. In particular, I would like to have a proof of the model-view-controller pattern in hand when I begin work on designing the graphical user interface library.

For example, many programs feature multiple instances of the subject-observer pattern – a spreadsheet cell might listen to each of its neighbors – and many interesting new difficulties arise. For example, the natural invariants for these structures is global over the ensemble of objects, and it is not immediately obvious how to state them in a style that allows us to best exploit separation (say, by temporarily peeling an object out of the global state and then returning it).

This is a prelude for the major case study, which will be the design and implementation of a graphical user interface library. I will design a widget library, and an event loop that uses it, in the traditional stateful callback style of most user interface toolkits. I will also want to try and write some simple client programs (e.g., a calculator), in a model-view-controller style, since this is a widely-used way of structuring these programs.

Such a proof of a model-view-controller architecture will represent another step forward in the complexity of sharing structures researchers have investigated with separation logic. The high water mark in this arena are the proofs of the correctness of garbage collection algorithms [?, ?], which simultaneously view the heap in multiple ways in order to prove that collection preserves the spanning tree of the heap. A model-view-controller system has the same degree of

I think this might take as much as four months.

In total, all of this adds up to fifteen months, plus perhaps another three months for writing.

9 Programming Language Syntax

The programming language we consider is a simply-typed, monadic, pure functional programming language.

First, we give the syntax of types and programs.

Type	$\tau ::= 0 \mid 1 \mid \mathbb{N} \mid \tau + \tau \mid \tau \times \tau \mid \tau \rightarrow \tau \mid \text{ref } \tau \mid \bigcirc \tau$
Pure terms	$e ::= () \mid \langle e_1, e_2 \rangle \mid \text{fst } e \mid \text{snd } e$ $\quad \mid \text{inl } e \mid \text{inr } e \mid \text{case}(e, x'. e', x''. e'')$ $\quad \mid x \mid e e' \mid \lambda x : \tau. e$ $\quad \mid z \mid \mathfrak{s}(e) \mid \text{iter}(e, e_z, x. e_s)$ $\quad \mid l_\tau \mid \text{fix}_\tau e \mid [e]$
Computations	$c ::= e \mid \text{letv } x = e \text{ in } c \mid \text{run } e$ $\quad \mid !e \mid e_1 := e_2 \mid \text{new}_\tau e$

The types of this programming language are a simply-typed, monadic functional language. We have the empty type 0, the unit type 1, pair types $\tau \times \tau'$, sum types $\tau + \tau'$, function space $\tau \rightarrow \tau'$, the natural number type \mathbb{N} , and the reference types $\text{ref } \tau$. All terms of these types are pure, in the sense that evaluating a term of this type has no side effects (does not read or write the store) and always terminates.

In addition, we also have a *monadic* type constructor $\bigcirc \tau$, which is the type of potentially side-effecting computations producing values of type τ . Side effects include both heap effects (such as reading, writing, or allocating a reference) and nontermination. This separation of effectful and effect-free code is very strict – most real functional languages, such as SML or Haskell admit at least nontermination as effects.

We maintain a very strong distinction between pure and impure code for two reasons. First, it allows us to validate very powerful equational reasoning principles for our language: we can validate the full β and η rules for each of the pure types. We think this can greatly simplify reasoning about even imperative programs, because we can relatively freely restructure our program to make it easier to follow the structure of a proof. Second, when expressions appear in assertions – that is, the pre- and post-conditions of Hoare triples – they must be pure. After all, what could an assertion that modifies the heap even mean? But being able to use full program expressions like function calls arithmetic in expressions makes it much easier to write modular specifications. Making the function type pure squares this circle.

The pure terms of the language are as expected. We have $()$ as the inhabitant of 1, pairs $\langle e, e' \rangle$, sums $\text{inl } e$ and $\text{inr } e$, natural numbers z and $\mathfrak{s}(e)$, and functions $\lambda x : \tau. e$. We also have the corresponding eliminations for each type. We have the projections, $\text{fst } e$ and $\text{snd } e$, for pairs; the case statement $\text{case}(e, x_1. e_1, x_2. e_2)$ for sums; the primitive iteration construct $\text{iter}(e, e_z, x. e_s)$ for natural numbers; and function application $e e'$. The only surprising member of this collection is primitive iteration – having well-founded iteration in the pure part of the language lets us simply define any arithmetic operations we might need.

We also have l_τ , which are the literal terms corresponding to reference loca-

tions. Note that they also carry their type with them – we use this device as a convenience, to relieve ourselves of having to carry a store typing around in our typing judgements. Finally, we also have the terms of monadic type $[c]$, which are *suspended computations*. They are blocks of imperative code, suspended so that we can treat them as pure values. We allow general recursion with a term-level fixed point operator $\text{fix}_x \tau c$. The type of fixed points must be at a type whose interpretation is a pointed domain – it is given by a grammar like $\delta ::= \bigcirc\tau \mid \tau \rightarrow \delta \mid \delta \times \delta$. This allows us to ensure that the fixed point actually exists.

The terms c of the computation language are as follows. First, we can embed any pure term e into the computations, and treat it as a computation that happens not to have any side effects. Second, we have the term $\text{run } e$, whose interpretation is that it takes a suspended computation and then executes it.

Third, we have sequential composition $\text{letv } x = e \text{ in } c$. This is also the proof term corresponding to the monadic bind operation, which lifts a term of type $A \rightarrow \bigcirc B$ to $\bigcirc A \rightarrow \bigcirc B$. The operational interpretation of this expression is to 1) evaluate e until it becomes some $[c']$, 2) then to evaluate c' and bind its result value to x , and then to run c in an environment enriched with x and the state produced by executing c' .

If we think of the monadic type $\bigcirc\tau$ as a generalization of Algol’s command type that returns a value in addition to having side-effects, then $\text{letv } x = e \text{ in } c$ can be thought of as a generalization of the sequential composition $c; c'$ to account for the possible return value.

Note that $\text{run } e$ is equivalent to $\text{letv } x = e \text{ in } x$. Both of these expressions evaluate e to get a suspended monadic computation, run that computation, and then return the result. While this is redundant, having both simplifies the specification logic a bit.

Next, there are the computation terms that actually allow us to perform side effects. For heap effects, we can allocate a new reference of type $\text{ref } \tau$ via $\text{new}_\tau e$, dereference a reference with the expression $!e$, and perform an assignment with the computation expression $e := e'$.

Below we give the typing rules for pure expressions.

$$\begin{array}{c}
\frac{}{\Gamma \vdash () : 1} \text{TUNIT} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \text{TPAIR} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{fst } e : \tau_1} \text{TFST} \\
\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{snd } e : \tau_2} \text{TSND} \quad \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \text{inl } e : \tau_1 + \tau_2} \text{TINL} \quad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \text{inr } e : \tau_1 + \tau_2} \text{TINR} \\
\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_2 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \text{case}(e, x_1. e_1, x_2. e_2) : \tau} \text{TCASE} \\
\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \text{TVAR} \quad \frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash \lambda x : \tau'. e : \tau' \rightarrow \tau} \text{TLAM} \\
\frac{\Gamma \vdash e : \tau' \rightarrow \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash e e' : \tau} \text{TAPP} \quad \frac{}{\Gamma \vdash z : \mathbb{N}} \text{TZERO} \quad \frac{\Gamma \vdash e : \mathbb{N}}{\Gamma \vdash \text{s}(e) : \mathbb{N}} \text{TSUCC} \\
\frac{\Gamma \vdash e : \mathbb{N} \quad \Gamma \vdash e_z : \tau \quad \Gamma, x : \tau \vdash e_s : \tau}{\Gamma \vdash \text{iter}(e, e_z, x. e_s) : \tau} \text{TITER} \quad \frac{}{\Gamma \vdash l_\tau : \text{ref } \tau} \text{TLOC} \\
\frac{\Gamma \vdash c \div \tau}{\Gamma \vdash [c] : \bigcirc \tau} \text{TCOMP} \quad \frac{\Gamma \vdash e : \tau \rightarrow \tau \quad \tau \text{ is a pointed type}}{\Gamma \vdash \text{fix}_\tau e : \tau} \text{TFIX}
\end{array}$$

Now, here are the typings for the impure computations.

$$\begin{array}{c}
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e \div \tau} \text{TPURE} \quad \frac{\Gamma \vdash e : \bigcirc \tau' \quad \Gamma, x : \tau' \vdash c \div \tau}{\Gamma \vdash \text{letv } x = e \text{ in } c \div \tau} \text{TLET} \\
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{new}_\tau e \div \text{ref } \tau} \text{TNEW} \quad \frac{\Gamma \vdash e : \text{ref } \tau}{\Gamma \vdash !e \div \tau} \text{TGET} \\
\frac{\Gamma \vdash e : \text{ref } \tau \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e := e' : 1} \text{TSET} \quad \frac{\Gamma \vdash e : \bigcirc \tau}{\Gamma \vdash \text{run } e \div \tau} \text{TRUN}
\end{array}$$

We also have a denotational semantics for the types and terms of this language. Because I want to present all of the syntactic rules of the language and specification logic together, in one continuous block, I will defer giving that semantics until after the syntax of everything is given.

10 Equality Theory on Terms

Once we have the denotational semantics, we can prove that various terms of the programming language have equal denotations, and this will let us give a

syntactic equality judgement for pure and impure computations. Because there are two syntactic categories, we also have two corresponding judgements. First, we have the judgement that $\Gamma \vdash e \equiv e' : \tau$, which says that two terms e and e' are equal at type τ in the context Γ . Second, we have the judgement $\Gamma \vdash c \equiv c' \div \tau$, which says that two computation terms c and c' are equal at type τ , in the context Γ .

For each type, we will give some congruence rules, some β rules, and some η rules. Because we have ensured that the pure language has no side-effects, we can show that our products are categorical products, our sums are categorical sums, our natural numbers form a natural number object, in the category of predomains and continuous functions. I want to emphasize that we have a fully extensional equality, even for sums and *natural numbers*.

For the monadic type, we prove equivalences corresponding to the monad laws – for example, that $\text{letv } x = [e] \text{ in } c \equiv [e/x]c$, and that $\text{letv } x = [c] \text{ in } x \equiv c$, and so on. In addition, we have the equivalence that $\text{letv } x = e \text{ in } x \equiv \text{run } e$, and at all pointed types, we equate $\text{fix}_\tau e$ with e ($\text{fix}_\tau e$).

I have not have gone to any particular effort to prove much equational theory about store allocations. Instead, we reason about store allocations is via Hoare logic on monadic terms. This lets us get away with using a fairly crude model of the heap, since we aren't asking our equational theory to validate many equations about it.

In addition, we have several structural rules. There are the usual rules of reflexivity, transitivity and symmetry that make this relation into an equivalence relation. We also show that substitution of equals into equals preserves equality.

$$\begin{array}{c}
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e \equiv e : \tau} \qquad \frac{\Gamma \vdash e \equiv e' : \tau \quad \Gamma \vdash e' \equiv e'' : \tau}{\Gamma \vdash e \equiv e'' : \tau} \qquad \frac{\Gamma \vdash e \equiv e' : \tau}{\Gamma \vdash e' \equiv e : \tau} \\
\\
\frac{\Gamma \vdash c \div \tau}{\Gamma \vdash c \equiv c \div \tau} \qquad \frac{\Gamma \vdash c \equiv c' \div \tau \quad \Gamma \vdash c' \equiv c'' \div \tau}{\Gamma \vdash c \equiv c'' \div \tau} \qquad \frac{\Gamma \vdash c \equiv c' \div \tau}{\Gamma \vdash c' \equiv c \div \tau} \\
\\
\frac{\Gamma, x : \tau \vdash e_1 \equiv e_2 : \tau' \quad \Gamma \vdash e'_1 \equiv e'_2 : \tau}{\Gamma \vdash [e'_1/x]e_1 \equiv [e'_2/x]e_2 : \tau'} \\
\\
\frac{\Gamma, x : \tau \vdash c_1 \equiv c_2 \div \tau' \quad \Gamma \vdash e'_1 \equiv e'_2 : \tau}{\Gamma \vdash [e'_1/x]c_1 \equiv [e'_2/x]c_2 \div \tau'}
\end{array}$$

$$\begin{array}{c}
\frac{\Gamma \vdash e : 1 \quad \Gamma \vdash e' : 1}{\Gamma \vdash e \equiv e' : 1} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{fst} \langle e_1, e_2 \rangle \equiv e_1 : \tau_1} \\
\\
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{snd} \langle e_1, e_2 \rangle \equiv e_2 : \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \langle \mathbf{fst} e, \mathbf{snd} e \rangle \equiv e : \tau_1 \times \tau_2} \\
\\
\frac{\Gamma \vdash e_1 \equiv e'_1 : \tau_1 \quad \Gamma \vdash e_2 \equiv e'_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle \equiv \langle e'_1, e'_2 \rangle : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e \equiv e' : \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{fst} e \equiv \mathbf{fst} e' : \tau_1} \\
\\
\frac{\Gamma \vdash e \equiv e' : \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{snd} e \equiv \mathbf{snd} e' : \tau_2} \qquad \frac{\Gamma \vdash e : 0 \quad \Gamma \vdash e' : 0}{\Gamma \vdash e \equiv e' : 0} \\
\\
\frac{\Gamma \vdash \mathbf{case}(\mathbf{inr} e, x_1. e_1, x_2. e_2) : \tau}{\Gamma \vdash \mathbf{case}(\mathbf{inr} e, x_1. e_1, x_2. e_2) \equiv [e/x]e_1 : \tau} \\
\\
\frac{\Gamma \vdash \mathbf{case}(\mathbf{inl} e, x_1. e_1, x_2. e_2) : \tau}{\Gamma \vdash \mathbf{case}(\mathbf{inl} e, x_1. e_1, x_2. e_2) \equiv [e/x]e_2 : \tau} \\
\\
\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x : \tau_1 + \tau_2 \vdash e' : \tau}{\Gamma \vdash [e/x]e' \equiv \mathbf{case}(e, y. [\mathbf{inl} y/x]e', y. [\mathbf{inr} y/x]e') : \tau} \\
\\
\frac{\Gamma \vdash e \equiv e' : \tau_1}{\Gamma \vdash \mathbf{inr} e \equiv \mathbf{inr} e' : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e \equiv e' : \tau_2}{\Gamma \vdash \mathbf{inl} e \equiv \mathbf{inl} e' : \tau_1 + \tau_2} \\
\\
\frac{\Gamma \vdash e \equiv e' : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 \equiv e'_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 \equiv e'_2 : \tau}{\Gamma \vdash \mathbf{case}(e, x_1. e_1, x_2. e_2) \equiv \mathbf{case}(e', x_1. e'_1, x_2. e'_2) : \tau} \\
\\
\frac{\Gamma \vdash \lambda x : \tau'. e : \tau' \rightarrow \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash (\lambda x : \tau'. e) e' \equiv [e'/x]e : \tau} \\
\\
\frac{\Gamma \vdash (\lambda x : \tau'. e x) \equiv (\lambda x : \tau'. e' x) : \tau' \rightarrow \tau}{\Gamma \vdash e \equiv e' : \tau' \rightarrow \tau} \\
\\
\frac{\Gamma, x : \tau' \vdash e \equiv e' : \tau}{\Gamma \vdash (\lambda x : \tau'. e) \equiv (\lambda x : \tau'. e') : \tau' \rightarrow \tau} \\
\\
\frac{\Gamma \vdash e_1 \equiv e'_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 \equiv e'_2 : \tau'}{\Gamma \vdash (e_1 e_2) \equiv (e'_1 e'_2) : \tau}
\end{array}$$

$$\begin{array}{c}
\frac{\Gamma \vdash \text{iter}(z, e_z, x : \tau. e_s) : \tau}{\Gamma \vdash \text{iter}(z, e_z, x : \tau. e_s) \equiv e_z : \tau} \\
\\
\frac{\Gamma \vdash \text{iter}(s(e), e_z, x : \tau. e_s) : \tau}{\Gamma \vdash \text{iter}(s(e), e_z, x. e_s) \equiv [\text{iter}(e, e_z, x. e_s)/x]e_s : \tau} \\
\\
\frac{\Gamma, x : \tau \vdash h : \tau' \quad \Gamma, y : \tau \vdash f : \tau \quad \Gamma, z : \tau' \vdash g : \tau' \quad \Gamma \vdash u : \tau \quad \Gamma \vdash e : \mathbb{N} \quad \Gamma, y : \tau \vdash [f/y]h \equiv [h/z]g : \tau'}{\Gamma \vdash [\text{iter}(e, u, y. f)/x]h \equiv \text{iter}(e, [u/x]h, z. g) : \tau'} \\
\\
\frac{\Gamma \vdash \text{fix}_\tau e : \tau}{\Gamma \vdash \text{fix}_\tau e \equiv e (\text{fix}_\tau e) : \tau} \qquad \frac{\Gamma \vdash c \equiv c' \div \tau}{\Gamma \vdash [c] \equiv [c'] : \bigcirc \tau} \\
\\
\frac{\Gamma \vdash \text{letv } x = [e] \text{ in } c \div \tau}{\Gamma \vdash \text{letv } x = [e] \text{ in } c \equiv [e/x]c \div \tau} \qquad \frac{\Gamma \vdash c \div \tau}{\Gamma \vdash c \equiv \text{letv } x = [c] \text{ in } x \div \tau} \\
\\
\frac{\Gamma \vdash \text{letv } y = [\text{letv } x = e \text{ in } c] \text{ in } c' \div \tau}{\Gamma \vdash \text{letv } y = [\text{letv } x = e \text{ in } c] \text{ in } c' \equiv \text{letv } x = e \text{ in } \text{letv } y = [c] \text{ in } c' \div \tau} \\
\\
\frac{\Gamma \vdash e \equiv e' : \tau}{\Gamma \vdash \text{new}_\tau e \equiv \text{new}_\tau e' \div \text{ref } \tau} \qquad \frac{\Gamma \vdash e \equiv e' : \text{ref } \tau}{\Gamma \vdash !e \equiv !e' \div \tau} \\
\\
\frac{\Gamma \vdash e_1 \equiv e'_1 : \text{ref } \tau \quad \Gamma \vdash e_2 \equiv e'_2 : \tau}{\Gamma \vdash (e_1 := e_2) \equiv (e'_1 := e'_2) \div 1} \qquad \frac{\Gamma \vdash e : \bigcirc \tau}{\Gamma \vdash \text{run } e \equiv (\text{letv } x = e \text{ in } x) \div \tau} \\
\\
\frac{\Gamma \vdash e \equiv e' : \bigcirc \tau}{\Gamma \vdash \text{run } e \equiv \text{run } e' \div \tau}
\end{array}$$

11 Syntax and Typing of Assertions and Specifications

The design of the specification language we use is inspired by Reynolds work on specification logic for Algol. Like specification logic, we have a two level logic, in which the top level logic is a first-order⁵ intuitionistic logic whose atomic formulas are Hoare triples. The logic used to specify the pre and postconditions in the Hoare triples is higher order separation logic. We give the syntax of these formulas below:

⁵The model can accommodate higher-order specifications, but it is not clear to what use such specifications can be put.

Formula Types	$\omega ::= \tau \mid \mathbf{prop} \mid \omega \Rightarrow \omega$
Formula Terms	$p ::= e \mid \top \mid p \wedge p' \mid p \supset p'$ $\quad \mid \perp \mid p \vee p' \mid \mathbf{emp} \mid p * p' \mid p -* p'$ $\quad \mid e \mapsto e' \mid p = p' \mid \exists x : \omega. p \mid \forall x : \omega. p$ $\quad \mid x \mid p p' \mid \lambda x : \omega. p$ $\quad \mid S \text{ valid}$
Formula Contexts	$\Delta ::= \cdot \mid \Delta, x : \omega$
Specifications	$S ::= \{P\}c\{a : \tau. Q\} \mid \{P\}$ $\quad \mid S \text{ and } S' \mid S \text{ implies } S' \mid S \text{ or } S'$ $\quad \mid \forall x : \omega. S \mid \exists x : \omega. S$
Specification Contexts	$\Sigma ::= \cdot \mid S, \Sigma$

The formulas of the separation logic are the terms of a multi-sorted higher order logic. The base sorts are program types τ , and the sort of propositions \mathbf{prop} . We can form higher types with the function space $\omega \Rightarrow \omega$.

The terms of the assertion language include all program terms e (which occur at the sorts τ), and the connectives of separation logic, all at sort \mathbf{prop} . These include the usual intuitionistic connectives of truth \top , conjunction $p \wedge q$, implication $p \supset q$, falsity \perp , and disjunction $p \vee q$, as well as equality $p = q$. This equality contains the denotational equality for program types τ , which will let us connect our equality judgement with our assertion language. (For example, if we can prove $e \equiv e'$, we should be able to believe the assertion $e = e'$ is true.)

Furthermore, we have the spatial connectives points-to $e \mapsto e'$, which denotes a heap consisting of a single reference e that has e' as its contents; the separating conjunction $p * q$, which asserts that p holds of one part of the heap and q holds for another, disjoint part; and the separating implication $p -* q$, the so-called “magic wand”. In addition, we have universal and existential quantification $\forall x : \omega. p$ and $\exists x : \omega. p$. Since the sort of the variable in these quantifiers ranges over all ω , this makes the assertion logic a higher-order logic.

Finally, we have a modal operator $S \text{ valid}$, which is an assertion that the specification S is a *valid* specification. This connects the two levels of our specification language, and is useful when, for example, we return a command as a result and wish to say something about its behavior.

The specification logic is a first-order intuitionistic logic built with two kinds of atomic propositions. The first kind are Hoare triples. They take the form $\{P\}c\{a : \tau. Q\}$, which mean that if the command c is run in a heap in a precondition state P , then if the command terminates, the heap will be described by the post-condition state Q . The binder $a : \tau$ lets us name the return value in the postcondition.

The second kind of atomic proposition we have are is the modal operator $\{P\}$, which is a specification that holds when the proposition P is a tautology of the assertion logic. Now, specifications and assertions can refer to each other.

The specification logic also has the usual propositional connectives conjunction S and S' , disjunction S or S' , and implication S implies S' . The first order quantifiers are $\forall x : \omega. S$ and $\exists x : \omega. S$. These are these are (multi-sorted) first order quantifiers at the level of the specification language, because the sorts ω only range over assertions and programming language types; they do not range over specifications. We could also add higher-order quantification at this level, but so far I have seen no particular need for it.

The well-typing judgement for assertion and specifications are mutually recursive, and consist of the two judgements $\Delta \vdash p : \omega$ and $\Delta \vdash S : \text{spec}$.

The typing rules for assertions sorts are the rules of the lambda calculus, plus lifting expressions up to the level of assertions and specifications. The only notable feature of these rules is the restriction operation $\Delta \downarrow$, which restricts Δ to only hypotheses of expression type τ .

$$\begin{array}{c}
\frac{\Delta \downarrow \vdash e : \tau}{\Delta \vdash e : \tau} \text{TTERM} \qquad \frac{}{\Delta \vdash \top : \text{prop}} \text{TTRUE} \\
\\
\frac{\Delta \vdash p_1 : \text{prop} \quad \Delta \vdash p_2 : \text{prop}}{\Delta \vdash p_1 \wedge p_2 : \text{prop}} \text{TAND} \\
\\
\frac{\Delta \vdash p_1 : \text{prop} \quad \Delta \vdash p_2 : \text{prop}}{\Delta \vdash p_1 \supset p_2 : \text{prop}} \text{TIMPLIES} \qquad \frac{}{\Delta \vdash \perp : \text{prop}} \text{TFALSE} \\
\\
\frac{\Delta \vdash p_1 : \text{prop} \quad \Delta \vdash p_2 : \text{prop}}{\Delta \vdash p_1 \vee p_2 : \text{prop}} \text{TOR} \qquad \frac{}{\Delta \vdash \text{emp} : \text{prop}} \text{TEMP} \\
\\
\frac{\Delta \vdash p_1 : \text{prop} \quad \Delta \vdash p_2 : \text{prop}}{\Delta \vdash p_1 * p_2 : \text{prop}} \text{TSTAR} \\
\\
\frac{\Delta \vdash p_1 : \text{prop} \quad \Delta \vdash p_2 : \text{prop}}{\Delta \vdash p_1 \multimap p_2 : \text{prop}} \text{TWAND} \\
\\
\frac{\Delta \downarrow \vdash e : \text{ref } \tau \quad \Delta \downarrow \vdash e' : \tau}{\Delta \vdash e \mapsto e' : \text{prop}} \text{TPOINTSTO} \qquad \frac{\Delta \vdash p : \omega \quad \Delta \vdash p' : \omega}{\Delta \vdash p =_{\omega} p' : \text{prop}} \text{TEQUAL} \\
\\
\frac{\Delta, x : \omega \vdash p : \text{prop}}{\Delta \vdash \forall x : \omega. p : \text{prop}} \text{TFORALL} \qquad \frac{\Delta, x : \omega \vdash p : \text{prop}}{\Delta \vdash \exists x : \omega. p : \text{prop}} \text{T EXISTS} \\
\\
\frac{x : \omega \in \Delta}{\Delta \vdash x : \omega} \text{TVARF} \qquad \frac{\Delta \vdash p : \omega' \Rightarrow \omega \quad \Delta \vdash p' : \omega'}{\Delta \vdash p p' : \omega} \text{TAPPF} \\
\\
\frac{\Delta, x : \omega' \vdash p : \omega}{\Delta \vdash \lambda x : \omega'. p : \omega' \Rightarrow \omega} \text{TLAMF} \qquad \frac{\Delta \vdash S : \text{spec}}{\Delta \vdash S \text{ valid} : \text{prop}} \text{TSPECVALID}
\end{array}$$

Then, we have the typing rules for specifications.

$$\begin{array}{c}
\frac{\Delta \vdash P : \text{prop} \quad \Delta \vdash c \div \tau \quad \Delta, a : \tau \vdash Q : \text{prop}}{\Delta \vdash \{P\}c\{a : \tau. Q\} : \text{spec}} \text{THOARE} \\
\\
\frac{\Delta \vdash P : \text{prop}}{\Delta \vdash \{P\} : \text{spec}} \text{T\tauAUT} \quad \frac{\Delta \vdash S : \text{spec} \quad \Delta \vdash S' : \text{spec}}{\Delta \vdash S \text{ and } S' : \text{spec}} \text{TSPECAND} \\
\\
\frac{\Delta \vdash S : \text{spec} \quad \Delta \vdash S' : \text{spec}}{\Delta \vdash S \text{ or } S' : \text{spec}} \text{TSPECOR} \\
\\
\frac{\Delta \vdash S : \text{spec} \quad \Delta \vdash S' : \text{spec}}{\Delta \vdash S \text{ implies } S' : \text{spec}} \text{TSPECIMP} \\
\\
\frac{\Delta, x : \omega \vdash S : \text{spec}}{\Delta \vdash \forall x : \omega. S : \text{spec}} \text{TSPECFORALL} \quad \frac{\Delta, x : \omega \vdash S : \text{spec}}{\Delta \vdash \exists x : \omega. S : \text{spec}} \text{TSPEC EXISTS}
\end{array}$$

Finally, we have the typing rules for specification contexts, $\Delta \vdash \Sigma : \text{context}$.

$$\begin{array}{c}
\frac{}{\Delta \vdash \cdot : \text{context}} \text{TSPECCTXNIL} \\
\\
\frac{\Delta \vdash S : \text{spec} \quad \Delta \vdash \Sigma : \text{context}}{\Delta \vdash S, \Sigma : \text{context}} \text{TSPECCTXCONS}
\end{array}$$

12 Entailment of Assertions and Specifications

Now that we know what well-formed assertions and specifications are, we can give the judgements for manipulating them. The first judgement, $\Delta \vdash p \equiv q : \omega$, extends the equality judgement to the level of sorts. Note that we judge two propositions to be equal when they are logically equivalent, and we allow inferring equality from a true propositional equality. This is a common pattern; each of the judgements – equality, assertional entailment, and specification entailment – has rules that depend on the other judgements.

$$\begin{array}{c}
\frac{\Delta \downarrow \vdash e \equiv e' : \tau}{\Delta \vdash e \equiv e' : \tau} \text{PEQTERM} \qquad \frac{\Delta \triangleright p \vdash q \quad \Delta \triangleright q \vdash p}{\Delta \vdash p \equiv q : \text{prop}} \text{PEQASSERT} \\
\\
\frac{\Delta \vdash \lambda x : \omega'. p : \omega' \Rightarrow \omega \quad \Delta \vdash q : \omega'}{\Delta \vdash (\lambda x : \omega'. p) q \equiv [q/x]p : \omega} \text{PEQBETA} \\
\\
\frac{\Delta \vdash p \equiv p' : \omega' \Rightarrow \omega}{\Delta \vdash \lambda x : \omega'. p x \equiv \lambda x : \omega'. p' x : \omega' \Rightarrow \omega} \\
\\
\frac{\Delta, x : \omega' \vdash p \equiv p' : \omega}{\Delta \vdash (\lambda x : \omega'. e) \equiv (\lambda x : \omega'. e') : \omega' \rightarrow \omega} \qquad \frac{\Delta \triangleright \top \vdash (p =_{\omega} p')}{\Delta \vdash p \equiv p' : \omega}
\end{array}$$

The second judgement, $\Delta \triangleright p \vdash q$, says that the assertion p entails the formula q for all assignments of the free variables in Δ . These rules are all sound with respect to the standard entailment relation for our separation logic. Since separation logic is not finitely axiomatizable, we cannot give a complete set of rules. Below, we give some rules corresponding to the associativity, commutativity and unit properties of the separating conjunction; the distributivity laws that the separating conjunction satisfies; the adjointness relationship between the separating conjunction and magic wand; and rules for how the equality and specification validity judgements let us introduce equality and specification hypotheses.

$$\begin{array}{c}
\frac{\Delta \vdash p : \text{prop} \quad \Delta \vdash q : \text{prop}}{\Delta \triangleright p * q \vdash q * p} \text{PCOMM} \quad \frac{\Delta \vdash p : \text{prop} \quad \Delta \vdash q : \text{prop}}{\Delta \triangleright p * (q * r) \vdash (p * q) * r} \text{PAssoc1} \\
\frac{\Delta \vdash p : \text{prop} \quad \Delta \vdash q : \text{prop}}{\Delta \triangleright (p * q) * r \vdash p * (q * r)} \text{PAssoc2} \quad \frac{\Delta \vdash p : \text{prop}}{\Delta \triangleright p * \text{emp} \vdash p} \text{PUNIT1} \\
\frac{\Delta \vdash p : \text{prop}}{\Delta \triangleright p \vdash p * \text{emp}} \text{PUNIT2} \\
\frac{\Delta \vdash p_1 : \text{prop} \quad \Delta \vdash p_2 : \text{prop} \quad \Delta \vdash q : \text{prop}}{\Delta \triangleright (p_1 \vee p_2) * q \vdash (p_1 * q) \vee (p_2 * q)} \text{PORDIST1} \\
\frac{\Delta \vdash p_1 : \text{prop} \quad \Delta \vdash p_2 : \text{prop} \quad \Delta \vdash q : \text{prop}}{\Delta \triangleright (p_1 * q) \vee (p_2 * q) \vdash (p_1 \vee p_2) * q} \text{PORDIST2} \\
\frac{\Delta \vdash p_1 : \text{prop} \quad \Delta \vdash p_2 : \text{prop} \quad \Delta \vdash q : \text{prop}}{\Delta \triangleright (p_1 \wedge p_2) * q \vdash (p_1 * q) \wedge (p_2 * q)} \text{PANDDIST} \\
\frac{\Delta \vdash \exists x : \omega. p : \text{prop} \quad \Delta \vdash q : \text{prop}}{\Delta \triangleright (\exists x : \omega. p) * q \vdash \exists x : \omega. (p * q)} \text{PEXISTSDIST1} \\
\frac{\Delta \vdash \exists x : \omega. p : \text{prop} \quad \Delta \vdash q : \text{prop}}{\Delta \triangleright \exists x : \omega. (p * q) \vdash (\exists x : \omega. p) * q} \text{PEXISTSDIST2} \\
\frac{\Delta \vdash \forall x : \omega. p : \text{prop} \quad \Delta \vdash q : \text{prop}}{\Delta \triangleright (\forall x : \omega. p) * q \vdash \forall x : \omega. (p * q)} \text{PforallDIST} \\
\frac{\Delta \triangleright p * q \vdash r}{\Delta \triangleright p \vdash q \neg * r} \text{PWANDADJ1} \quad \frac{\Delta \triangleright p \vdash q \neg * r}{\Delta \triangleright p * q \vdash r} \text{PWANDADJ2} \\
\frac{\Delta \triangleright p \vdash p' \quad \Delta \triangleright q \vdash q'}{\Delta \triangleright p * p' \vdash q * q'} \text{PMONOTONE} \quad \frac{\Delta; \cdot \vdash S \text{ ok} \quad \Delta \vdash p : \text{prop}}{\Delta \triangleright p \vdash p \wedge (S \text{ valid})} \text{PVALID} \\
\frac{\Delta \vdash p \equiv p' : \omega \quad \Delta \vdash q : \text{prop}}{\Delta \triangleright q \vdash q \wedge (p = p')} \text{PEQUAL}
\end{array}$$

(TODO: add the rules for distributing specificational and propositional connectives)

The third judgement, $\Delta; \Sigma \vdash S \text{ ok}$, says that the specification S is provable under the specification hypotheses in Σ , for all assignments to the free variables in Δ . This is the standard natural deduction calculus for first-order intuitionistic logic; we are slightly more explicit than usual because we give all the parameters

and their sorts in Δ .

$$\begin{array}{c}
\frac{\Delta; \Sigma \vdash S_1 \text{ ok} \quad \Delta; \Sigma \vdash S_2 \text{ ok}}{\Delta; \Sigma \vdash S_1 \text{ and } S_2 \text{ ok}} \text{SANDI} \\
\\
\frac{\Delta; \Sigma \vdash S_1 \text{ and } S_2 \text{ ok}}{\Delta; \Sigma \vdash S_1 \text{ ok}} \text{SANDE1} \qquad \frac{\Delta; \Sigma \vdash S_1 \text{ and } S_2 \text{ ok}}{\Delta; \Sigma \vdash S_2 \text{ ok}} \text{SANDE2} \\
\\
\frac{\Delta; \Sigma \vdash S_1 \text{ ok}}{\Delta; \Sigma \vdash S_2 \text{ or } S_2 \text{ ok}} \text{SORI1} \qquad \frac{\Delta; \Sigma \vdash S_2 \text{ ok}}{\Delta; \Sigma \vdash S_1 \text{ or } S_2 \text{ ok}} \text{SORI2} \\
\\
\frac{\Delta; \Sigma \vdash S_1 \text{ or } S_2 \text{ ok} \quad \Delta; \Sigma, S_1 \vdash S \text{ ok} \quad \Delta; \Sigma, S_2 \vdash S \text{ ok}}{\Delta; \Sigma \vdash S \text{ ok}} \text{SORE} \\
\\
\frac{\Delta; \Sigma, S_1 \vdash S_2 \text{ ok}}{\Delta; \Sigma \vdash S_1 \text{ implies } S_2 \text{ ok}} \text{SIMPI} \\
\\
\frac{\Delta; \Sigma \vdash S_1 \text{ implies } S_2 \text{ ok} \quad \Delta; \Sigma \vdash S_1 \text{ ok}}{\Delta; \Sigma \vdash S_2 \text{ ok}} \text{SIMPE} \\
\\
\frac{\Delta, x : \omega; \Sigma \vdash S \text{ ok} \quad x \notin \text{FV}(\Sigma)}{\Delta; \Sigma \vdash \forall x : \omega. S \text{ ok}} \text{SALLI} \\
\\
\frac{\Delta; \Sigma \vdash \forall x : \omega. S \text{ ok} \quad \Delta \vdash p : \omega}{\Delta; \Sigma \vdash [p/x]S \text{ ok}} \text{SALLE} \\
\\
\frac{\Delta; \Sigma \vdash [p/x]S \text{ ok} \quad \Delta \vdash p : \omega}{\Delta; \Sigma \vdash \exists x : \omega. S \text{ ok}} \text{SEXISTS1} \qquad \frac{\Delta \vdash \Sigma : \text{context} \quad S \in \Sigma}{\Delta; \Sigma \vdash S \text{ ok}} \text{TIDENT} \\
\\
\frac{\Delta; \Sigma \vdash S' \text{ ok} \quad \Delta; \Sigma, S' \vdash S \text{ ok}}{\Delta; \Sigma \vdash S \text{ ok}} \text{TCUT}
\end{array}$$

Beyond that, we must give rules for each of the atomic forms of the computation logic, to embed Hoare logic into the calculus. We give these rules in the “small footprint” style.

$$\begin{array}{c}
\frac{\Delta \vdash \Sigma : \text{context} \quad \Delta \vdash P : \text{prop} \quad \Delta \downarrow \vdash e : \tau}{\Delta; \Sigma \vdash \{P\}e\{a : \tau. P \wedge a = e\} \text{ ok}} \text{SPURE} \\
\\
\frac{\Delta \downarrow \vdash e : \tau \quad \Delta \vdash \Sigma : \text{context}}{\Delta; \Sigma \vdash \{\text{emp}\}\text{new}_\tau e\{a : \text{ref } \tau. a \mapsto e\} \text{ ok}} \text{SNEW} \\
\\
\frac{\Delta \downarrow \vdash e : \text{ref } \tau \quad \Delta \downarrow \vdash e' : \tau \quad \Delta \vdash p : \text{prop} \quad \Delta \vdash \Sigma : \text{context}}{\Delta; \Sigma \vdash \{p \wedge (e \mapsto e')\}!e\{a : \tau. p \wedge (e \mapsto e') \wedge a =_\tau e'\} \text{ ok}} \text{SREAD} \\
\\
\frac{\Delta \downarrow \vdash e : \text{ref } \tau \quad \Delta \downarrow \vdash e' : \tau \quad \Delta \downarrow \vdash e'' : \tau \quad \Delta \vdash \Sigma : \text{context}}{\Delta; \Sigma \vdash \{e \mapsto e'\}e := e''\{a : 1. e \mapsto e''\} \text{ ok}} \text{SWRITE} \\
\\
\frac{\Delta, a : \tau \vdash R : \text{prop} \quad \Delta; \Sigma \vdash \{P\}\text{run } e\{x : \tau'. Q\} \text{ ok} \quad \Delta, x : \tau'; \Sigma \vdash \{Q\}c\{a : \tau. R\} \text{ ok}}{\Delta; \Sigma \vdash \{P\}\text{letv } x = e \text{ in } c\{a : \tau. R\} \text{ ok}} \text{SSEQ} \\
\\
\frac{\Delta; \Sigma \vdash \{P\}c\{a : \tau. Q\} \text{ ok}}{\Delta; \Sigma \vdash \{P\}\text{run } [c]\{a : \tau. Q\} \text{ ok}} \text{SRUN} \\
\\
\frac{\Delta, f : \vec{\tau}_i \rightarrow \text{O}\tau; \Sigma, \forall \vec{x}_i : \vec{\tau}_i. \{P\}\text{run } f \vec{x}_i\{a : \tau. Q\} \vdash \forall \vec{x}_i : \vec{\tau}_i. \{P\}\text{run } e f \vec{x}_i\{a : \tau. Q\} \text{ ok}}{\Delta; \Sigma \vdash \forall \vec{x}_i : \vec{\tau}_i. \{P\}\text{run } (\text{fix}_{\vec{\tau}_i \rightarrow \text{O}\tau} e) \vec{x}_i\{a : \tau. Q\} \text{ ok}} \text{SFIX} \\
\\
\frac{\Delta \downarrow \vdash e : \tau_1 + \tau_2 \quad \Delta \vdash \Sigma : \text{context} \quad \Delta, x : \tau_1; \Sigma \vdash \{P \wedge \text{inl } x = e\}\text{run } e_1\{a : \tau. Q\} \text{ ok} \quad \Delta, y : \tau_2; \Sigma \vdash \{P \wedge \text{inr } y = e\}\text{run } e_2\{a : \tau. Q\} \text{ ok}}{\Delta; \Sigma \vdash \{P\}\text{run } \text{case}(e, x. e_1, y. e_2)\{a : \tau. Q\} \text{ ok}} \text{SCASE}
\end{array}$$

Next, we give some more rules to manipulate Hoare triples. We give the rule of consequence, a rule allowing substitution of equals for equals, and an induction rule for this logic.

$$\begin{array}{c}
\frac{\Delta \triangleright P \vdash P' \quad \Delta; \Sigma \vdash \{P'\}c\{a : \tau. Q'\} \text{ ok} \quad \Delta \triangleright Q \vdash Q'}{\Delta; \Sigma \vdash \{P\}c\{a : \tau. Q\} \text{ ok}} \text{SCONSEQUENCE} \\
\\
\frac{\Delta; \Sigma \vdash \{P\}c'\{a : \tau. Q\} \text{ ok} \quad \Delta \downarrow \vdash c' \equiv c \div \tau}{\Delta; \Sigma \vdash \{P\}c\{a : \tau. Q\} \text{ ok}} \text{SEQUAL} \\
\\
\frac{\Delta; \Sigma \vdash [z/x]S \text{ ok} \quad \Delta, x : \mathbb{N}; \Sigma, S \vdash [s(x)/x]S \text{ ok}}{\Delta; \Sigma \vdash \forall x : \mathbb{N}. S \text{ ok}} \text{SINDUCT}
\end{array}$$

Finally, we give some rules to manipulate assertion tautologies $\{P\}$. We give two introduction rules, some rules to move tautologies into and out of the

preconditions of Hoare triples, and a rule to move from a validity assertion to a valid specification.

$$\begin{array}{c}
\frac{\Delta \triangleright \top \vdash P \quad \Delta \vdash \Sigma : \text{context}}{\Delta; \Sigma \vdash \{P\} \text{ ok}} \text{STAUTI1} \\
\frac{\Delta; \Sigma \vdash \{P\} \text{ ok} \quad \Delta \triangleright P \vdash Q}{\Delta; \Sigma \vdash \{Q\} \text{ ok}} \text{STAUTI2} \\
\frac{\Delta; \Sigma \vdash \{R\} \text{ ok} \quad \Delta; \Sigma \vdash \{P \wedge R\}c\{a : \tau. Q\} \text{ ok}}{\Delta; \Sigma \vdash \{P\}c\{a : \tau. Q\} \text{ ok}} \text{STAUTE1} \\
\frac{\Delta \triangleright P \vdash R \quad \Delta; \Sigma, \{R\} \vdash \{P\}c\{a : \tau. Q\} \text{ ok}}{\Delta; \Sigma \vdash \{P\}c\{a : \tau. Q\} \text{ ok}} \text{STAUTE2} \\
\frac{\Delta; \Sigma \vdash \{S \text{ valid}\} \text{ ok}}{\Delta; \Sigma \vdash S \text{ ok}} \text{SVALID}
\end{array}$$

Finally, we give the frame rules of separation logic. We support higher order frame rules.

$$\begin{array}{c}
\frac{\Delta; \Sigma, \Sigma' \vdash S \text{ ok} \quad \Delta \vdash P : \text{prop}}{\Delta; \Sigma, \Sigma' \otimes P \vdash S \otimes P \text{ ok}} \text{SFRAMER} \\
\frac{\Delta; \Sigma, \Sigma \otimes P \vdash S \text{ ok} \quad \Delta \vdash P : \text{prop}}{\Delta; \Sigma, \Sigma \vdash S \text{ ok}} \text{SFRAMEL}
\end{array}$$

The notation $S \otimes P$ uniformly frames the assertion P to every Hoare triple occurring as a subformula of S , and $\Sigma \otimes P$ lifts that to sequences of specifications. The definition is given below:

$$\begin{array}{lcl}
\{Q\}c\{a : \tau. R\} \otimes P & = & \{P * Q\}c\{a : \tau. P * R\} \\
\{Q\} \otimes P & = & \{Q\} \\
(S_1 \text{ and } S_2) \otimes P & = & (S_1 \otimes P) \text{ and } (S_2 \otimes P) \\
(S_1 \text{ implies } S_2) \otimes P & = & (S_1 \otimes P) \text{ implies } (S_2 \otimes P) \\
(S_1 \text{ or } S_2) \otimes P & = & (S_1 \otimes P) \text{ or } (S_2 \otimes P) \\
(\forall x : \omega. S) \otimes P & = & \forall x : \omega. (S \otimes P) \\
(\exists x : \omega. S) \otimes P & = & \exists x : \omega. (S \otimes P) \\
\cdot \otimes P & = & \cdot \\
(S, \Sigma) \otimes P & = & (S \otimes P), (\Sigma \otimes P)
\end{array}$$

This style of presentation differs a bit from the higher order frame rules of Birkedal and Yang [4], where a subtyping judgement was used to describe these

rules. Here, there is no (explicit) subtyping, and instead we allow framing to happen between any two appropriately syntactically-related formulas. There could also be a third style, in which $- \otimes p$ is an explicit modality of the logic.

13 Bibliography

References

- [1] Martin Berger, Kohei Honda, and Nobuko Yoshida. A logical analysis of aliasing in imperative higher-order functions. *SIGPLAN Not.*, 40(9):280–293, 2005.
- [2] Bodil Biering, Lars Birkedal, and Noah Torp-Smith. BI-hyperdoctrines, higher-order separation logic, and abstraction. *ACM Transactions on Programming Languages and Systems*, 29(5):24, 2007.
- [3] Lars Birkedal, Noah Torp-Smith, and John C. Reynolds. Local reasoning about a copying garbage collector. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 220–231, New York, NY, USA, 2004. ACM.
- [4] Lars Birkedal, Noah Torp-Smith, and Hongseok Yang. Semantics of separation-logic typing and higher-order frame rules. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*, pages 260–269, June 2005.
- [5] John Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72, Berlin, Heidelberg, New York, 2003. Springer.
- [6] Antony Courtney. Functionally modeled user interfaces. In Joaquim A. Jorge, Nuno Jardim Nunes, and João Falcão e Cunha, editors, *DSV-IS*, volume 2844 of *Lecture Notes in Computer Science*, pages 107–123. Springer, 2003.
- [7] R. DeLine and M. Fahndrich. Typestates for objects, 2004.
- [8] F.B. Fitch. Natural deduction rules for obligation. *American Philosophical Quarterly*, 3:27–38, 1966.
- [9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.
- [10] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley Professional, 2000.

- [11] C.B. Jones. *Development Methods for Computer Programs Including a Notion of Interference*. PhD in Computer Science, University of Cambridge, June 1981.
- [12] Neelakantan R. Krishnaswami. Reasoning about iterators with separation logic. In *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, pages 83–86, New York, NY, USA, 2006. ACM.
- [13] Neelakantan R. Krishnaswami, Lars Birkedal, and Jonathan Aldrich. Modular verification of the subject-observer pattern via higher-order separation logic. In *FTfJP '07: The 9th Workshop on Formal Techniques for the Verification of Java-like Programs*, 2007.
- [14] K. Rustan M. Leino and Wolfram Schulte. Using history invariants to verify observers. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, pages 80–94. Springer, 2007.
- [15] Barbara H. Liskov and Jeannette M. Wing. Behavioural subtyping using invariants and constraints. pages 254–280, 2001.
- [16] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.
- [17] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
- [18] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in hoare type theory. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 62–73, New York, NY, USA, 2006. ACM.
- [19] Peter W. O’Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 268–280, New York, NY, USA, 2004. ACM.
- [20] Matthew Parkinson. *Local Reasoning for Java*. PhD in Computer Science, University of Cambridge, August 2005.
- [21] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.
- [22] Gordon Plotkin and Martín Abadi. A logic for parametric polymorphism. In M. Bezem and J. F. Groote, editors, *International Conference on Typed Lambda Calculi and Applications*, number 664, pages 361–375, Utrecht, The Netherlands, 1993. Springer-Verlag.

- [23] Bernhard Reus and Jan Schwinghammer. Separation logic for higher-order store. In Zoltán Ésik, editor, *CSL*, volume 4207 of *Lecture Notes in Computer Science*, pages 575–590. Springer, 2006.
- [24] Bernhard Reus and Thomas Streicher. About hoare logics for higher-order store. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *ICALP*, volume 3580 of *Lecture Notes in Computer Science*, pages 1337–1348. Springer, 2005.
- [25] John C. Reynolds. An introduction to specification logic. In *Proceedings of the Carnegie Mellon Workshop on Logic of Programs*, page 442, London, UK, 1984. Springer-Verlag.
- [26] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002.
- [27] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR '07: 18th International Conference on Concurrency Theory, ONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings*, Lecture Notes in Computer Science, pages 256–271. Springer, 2007.
- [28] Philip Wadler. Theorems for free! In *Proceedings 4th Int. Conf. on Funct. Prog. Languages and Computer Arch., FPCA'89, London, UK, 11-13 Sept 1989*, pages 347–359, New York, 1989. ACM Press.
- [29] Qiwen Xu, Willem-Paul de Roever, and Jifeng He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.