# Idealized ML and Its Separation Logic

Neelakantan R. Krishnaswami
Carnegie Mellon University
neelk@cs.cmu.edu

Lars Birkedal
IT University of Denmark
birkedal@itu.dk

Jonathan Aldrich
Carnegie Mellon University
aldrich@cs.cmu.edu

John C. Reynolds
Carnegie Mellon University
jcr@cs.cmu.edu

## Abstract

Separation logic is an extension of Hoare logic which permits reasoning about low-level imperative programs that use shared mutable heap structure. In this work, we present a version of separation logic that permits effective, modular reasoning about typed, higher-order functional programs that use aliased mutable heap data, including pointers to code. Furthermore, we show how to use predicates in higher-order separation logic to modularly and abstractly specify the sharing behavior of programs.

## 1. Introduction

Separation logic [13, 30] is an extension of Hoare logic [12] originally developed to simplify the proofs of pointer programs. Conventional Hoare logic faced severe difficulties reasoning about pointer programs because of the problem of aliasing. Reasoning about aliasing is problematic because the specification must track whether or not any two pointer variables refer to the same data structure, so that variables are correctly updated when a data structure is modified. Therefore, the number of interference conditions (i.e., an assertion of the form $x = y$ or $x \neq y$) grows quadratically in the number of variables in the program, making verification of nontrivial routines infeasible.

O'Hearn and Reynolds resolved this difficulty by extending the logic with two new connectives, the separating conjunction $*$ and the separating implication $-\!\!*$ . The intuitive reading of an assertion $P * Q$ is that $P$ holds in one part of the heap, and $Q$ holds in a disjoint region of the heap. As a result, the interference conditions become implicit – in an assertion like $(x \hookrightarrow 5)*(y \hookrightarrow 5)$, meaning that $x$ is a pointer pointing to the value 5, and separately $y$ points to 5, we do not need to explicitly assert that $x$ and $y$ do not alias. Likewise, the reading of $P -\!\!* Q$ is that we have a heap in which $Q$ would hold, if we added additional disjoint storage in which $P$ holds.

Initially, work on separation logic focused on low level pointer programs, featuring explicit memory allocation and deallocation, pointer arithmetic, and heap structures made of integers. Since then, separation logic has been been used to verify a variety of complex pointer programs, including a copying garbage collector [6], the Schorr-Waite algorithm [32], and other graph algorithms [8]. More recently, separation logic has been extended in several directions, to shared variable concurrency [19, 20], static modules for first and higher-order languages [23, 7], a core subset of Java [25], and, very recently, to a language with higher-order store [28]

The contributions of this paper are as follows.

- We describe a version of separation logic designed to permit *reasoning about higher-order programs operating on higher-order store*. Our target language is an extension of the simply typed lambda calculus with products, sums, inductive types, references, and a monadic type constructor encapsulating heap effects (such as reading, writing, and allocating references). Our reference types are unrestricted, and in particular we allow pointers to code objects (such as functions or monadic computation terms).
- Furthermore, our assertion language is a *higher-order* separation logic [4], where quantifiers can range not just over program values, but also over propositions of separation logic. This permits us to *specify the behavior of an imperative module without revealing its concrete representation*, and so we can verify a module and the clients that use it independently.
- Finally, we can specify the behavior of the abstract heap predicates used in the abstract specification of a module using a novel idea of *static specifications*, which let us characterize the sharing behavior of a program as a tautology in separation logic.

We should like to stress that the combination of programming language features that we study here is interesting not only from the viewpoint of ML but also from the viewpoint of object-oriented programming, since we can represent some of the core challenging aspects of verification of object-oriented programs in our setting. Indeed, all of our examples in this paper are essentially standard object-oriented programs (although inheritance is not supported). We thus think of our present language and higher-order separation logic as a *core language for reasoning about imperative modules,* be it in ML-style or in OO-style.

The remainder of this paper is organized into three sections. First, we give several examples of how we can use higher-order separation logic to specify programs. Second, we give the semantics of this system, which stratifies into three layers, comprising the semantics of the programming language, the semantics of separation logic assertions, and finally the specification logic describing the behavior of programs. We conclude with a discussion of related work, and suggest some future directions.

## 2. Examples

In this section, we present three examples of specification and reasoning about shared mutable objects. The first example is of an iterator implementation in the style of Java [11]; the second is a specification of an implementation of a subject/observer pattern [11]; and the third is an example of two routines operating on a shared scratch space that sometimes ontains intermediate state. For the iterator, we describe an interface specification, sample implementation and sample client; for space reasons we give just the specifications for the other two examples.

The programming and assertion language is defined formally in the next section. To read the examples before looking at the formal definitions of the language, observe the following: We use ML notation for references, and thus write $!e$ for dereferencing a reference $e$. The square brackets [] are used to delineate commands and expressions. We write list for the type of *mutable* lists and we use LISP-style notation for operations on such mutable lists. We use ML-style notation for sequences used in assertions.

### 2.1 Iterators

A Java-style iterator interface works as follows. First, we have a mutable collection type. This type supports a number of operations, some of which, like `add`-ing an element to a collection will modify the collection, while others, like `empty`-ness checking, will not.

To access the elements of a collection, we create another mutable object called an iterator. This object has a method `next`, which returns a new element of the collection each time it is called, finally failing when there are no more elements within it.

However, both the collection and the iterator are imperative objects, so that correct usage of the iterator requires restrictions to ensure that the state of an iterator and its underlying collection remain in sync. Specifically, a client program:

- may create as many iterators on a single collection as they like,
- may freely call any methods on the collection that do not change the collection's state (such as `empty`)
- may freely call `next` on the iterators in any order, and
- may **NOT** call `next` on an iterator after calling `add` on the underlying collection.

The general idea is that an iterator maintains a pointer into some part of the collection during its traversal, and that updating the collection can cause the iterator's reference to point to an incorrect part of the collection.

We give a formal version of this English specification in Figure 1. The type of mutable linked lists is ref list (of nat), and an iterator object is a pointer to one of the tails of the list (i.e., type ref ref list).

To describe the heap behavior, we introduce a pair of existentially quantified predicates, $coll$ and $iter$. These predicates permit us to talk about the mutable state associated with collections and iterators, without revealing their concrete implementation. The predicate $coll(c, xs)$ asserts that the collection $c$ represents the abstract sequence $xs$, and the assertion $iter(i, c, xs, P)$ asserts that the iterator $i$ is an iterator over the collection $c$ with elements $xs$, and further that the collection $c$ is under some additional constraints $P$.

To read the specifications of the functions in the interface, read the formula $\langle P\rangle\, f(x)\, \langle a : \tau.\ Q\rangle$ as a Hoare triple, which says that beginning with a heap satisfying $P$, any terminating execution of $f(x)$ will produce a heap satisfying $Q$, provided the bound variable $a$ of type $\tau$ is taken to be the result returned by $f(x)$. For example, the specification $\langle \top\rangle$ `new_coll()` $\langle a :$ ref list$.\ coll(a, [])\rangle$ states that starting from any heap, calling `new_coll` will return a collection, stored in a modified heap, that represents the empty sequence.

The assertion $\langle coll(c, xs) \wedge P\rangle$ `new_iter`$(c)$ $\langle a.\ iter(a, c, xs, P)\rangle$ says that if we start with a collection $c$, then we can construct an iterator, and that if the state associated with the collection has any other invariants $P$, then the iterator will maintain them.

The `next` function has the specification

$$\langle iter(i, c, xs, P)\rangle\ \texttt{next}(i)\ \langle a : 1 + \mathsf{nat}.\ iter(i, c, xs, P)\rangle\,,$$

which says that if we have an iterator $i$, then `next`$(i)$ will give us an integer or signal a failure. In this spec, we do not model the behavior of the iterator in any further detail — the spec could easily be refined further, but that detail would not be relevant to the issue of reasoning about aliasing. The detail that is relevant is the fact that the iterator preserves the abstract collection state $P$, which is how we describe the fact that the iterator does not modify the underlying collection.

That said, a natural question is how we can create two iterators on the same collection, because the `new_iter` function transforms a $coll(c, xs)$ state to an $iter(i, c, xs, P)$ state, which means that the precondition to call `new_iter` no longer holds. This is where the *static specification* comes into play: The final invariant in the specification:

$$iter(i, c, xs, P) \supset \exists Q.\ (coll(c, xs) \wedge P \wedge Q) * \\ \forall R.\ (coll(c, xs) \wedge R \wedge Q) -\!\!* iter(i, c, xs, R)$$

is a single separation logic formula (hence the name "static specification") that describes how to recover a collection from an iterator state. It says that if we have an iterator state $iter(i, c, xs, P)$, then that state can be viewed as two disjoint pieces, one of which is the original collection (with the invariant $P$ maintained), and one piece that can be combined with the collection to restore the iterator.

The existential quantifier $Q$ enforces the property that the iterator becomes invalid if the underlying collection is mutated, because we must have $Q$ to recover the collection, and since this predicate is abstract, we can't mutate the collection and preserve its state $Q$. More concretely, $Q$ represents the invariant on which the iterator relies for its own correct operation.

The static specification makes fundamental use of the fact we have both standard implication and separating implication available in the same logic. We use implication to reason that the same piece of state can be viewed in multiple ways, and the separating implication to reason about one isolated part of the state.

We can see an example of how a client would make use of this specification in Figure 3. On line 1, we see that the precondition for our program is that the variable $c$ holds a collection. On line 4, we create an iterator $i_1$, consuming the collection to produce an iterator, as seen in the state on line 5. We now apply the static specification on line 6 to break the iterator state into two pieces, which lets us create a second iterator bound to $i_2$. (Notice that the existentially quantified assertion variable Q in the static specification becomes a fresh variable P on line 6 of the proof outline.) The program state on line 8 contains an iterator for $i_2$, and some state that will let us reconstruct $i_1$'s iterator. On line 9 we apply the static specification once more, to break out the collection state again, and this lets us call `empty` on line 10.

From line 11 to line 12, we use $(coll(c, xs) \wedge P \wedge Q) * \forall R.\ ((coll(c, xs) \wedge P \wedge R) -\!\!*\ iter(i_1, c, xs, R))$ to conclude $iter(i_1, c, xs, Q))$. Moreover, we apply the frame rule to maintain the starred invariant $\forall R.\ ((coll(c, xs) \wedge Q \wedge R) -\!\!* iter(i_2, c, xs, R))$. Thus we recover the precondition for calling `next`$(i_1)$ on line 13, and then on lines 14-16, we apply the static specification and combine the iterator state fragment for $i_2$, so that we can call `next`$(i_2)$ on line 17. On line 18 and 19, we once

$\exists$new_coll, empty, add, new_iter, next.
$\exists coll$ : (ref list $\times$ seq nat) $\Rightarrow$ prop.
$\exists iter$ : (ref ref list $\times$ ref list $\times$ seq nat $\times$ prop) $\Rightarrow$ prop.

$\langle \top \rangle$ new_coll() $\langle a$ : ref list. $coll(a, [])\rangle$ and

$\forall c, x, xs.\ \langle coll(c, xs)\rangle$ add$(c, x)$ $\langle a : 1.\ coll(c, x :: xs)\rangle$ and

$\forall P$ : prop, $c, xs.\ \langle coll(c, xs) \wedge P\rangle$
empty$(c)$
$\langle a$ : bool. $coll(c, xs) \wedge P \wedge a = (xs = [])\rangle$ and

$\forall c, xs, P.\ \langle coll(c, xs) \wedge P\rangle$
new_iter$(c)$
$\langle a$ : ref ref list. $iter(a, c, xs, P)\rangle$ and

$\forall i, c, xs, P.\ \langle iter(i, c, xs, P)\rangle$ next$(i)$ $\langle a : 1 + \mathsf{nat}.\ iter(i, c, xs, P)\rangle$ and

$\forall i, c, xs, P.\ \{iter(i, c, xs, P) \supset$
$\exists Q.\ (coll(c, xs) \wedge P \wedge Q) *$
$\forall R.\ (coll(c, xs) \wedge R \wedge Q) -\!\!* iter(i, c, xs, R)\}$

**Figure 1.** Iterator Specification

again use the static specification to disassemble the iterator and get back the collection.

Next, we want to call add$(c, x)$ on line 21. However, the rule for add does not allow us to preserve an arbitrary invariant over the collection's state (i.e., there is no frame rule for ordinary conjunction in our system since it would be unsound). Therefore, on line 20 we forget the collection invariants.

Now, calling add$(c, x)$ on line 21 gives us a state in which $coll(c, x :: xs)$ holds. As a result, we can no longer apply the separating implication law to get a full iterator state, and we can't call next on either $i_1$ or $i_2$ any longer.

So the Hoare triples and static specifications put us in a situation where we can create multiple iterators, and can freely call methods on the collection which don't change its state, but which also enforce the property that there can be no calls to next after modifying the collection – and the client was able to do this without knowing anything about the internal heap structure of the collection.

Finally, in Figure 2, we give example implementation for this specification. The implementations are the witnesses to the existential type: for the abstract program variables we give monadic programs (which manipulate imperative linked lists in the obvious way), and for the existentially quantified predicates, we give recursive functions that compute predicates from their arguments – i.e., the inductive predicate characterizing the heap structure. The $iter$ predicate, for example, is an assertion stating that the iterator points to an interior pointer of the linked list, and that the predicate variable $P$ is preserved for the whole list.

The static specification is a tautology given the definitions of the predicates. In particular, the heap described by the separating implication is the predicate $i \hookrightarrow c'$, while the predicate $Q$ is witnessed by $seg(c, c', xs_1) * coll(c', xs_2) \wedge xs = xs_1 \cdot xs_2$.

In this example, and indeed in the whole paper, we focus on being able to abstractly specify and reason about the imperative aspects of modules. Of course, one would also like the Iterator specification to be abstract in the implementation types used for collections and iterators (here ref list), i.e., to have existential quantification over *types* to model abstract data types. We do not foresee any problems in extending the system presented in this paper to include existential types and leave it for future work.

### 2.2 The Subject/Observer Pattern

We also give an example specification for the subject/observer pattern as an example of how to specify the use of higher-order

new_coll () $\equiv$ [new$_{\mathsf{list}}$nil]

add$(c, x)$ $\equiv$ [letv $[cell] = [!c]$ in
letv $[t] = [\mathsf{new}_{\mathsf{list}} cell]$ in
$c :=$ cons$(x, t)$]

empty$(c)$ $\equiv$ [letv $[cell] = [!c]$ in
listcase$(cell, \mathsf{true}, (h, t).\ \mathsf{false})$]

new_iter$(c)$ $\equiv$ [new$_{\mathsf{ref\ list}}(c)$]

next$(i)$ $\equiv$ [letv $[c] = [!i]$ in
letv $[cell] = [!c]$ in
letv $[ans]$ =listcase$(cell, [\mathsf{None}],$
$(h, t).$ [letv $[\_] = [i := t]$ in
Some $h]$) in
$ans]$

$coll(c, x :: xs) \equiv \exists c'.\ c \hookrightarrow \mathsf{cons}(x, c') * coll(c', xs)$
$coll(c, []) \equiv c \hookrightarrow \mathsf{nil}$

$seg(c, c', x :: xs) \equiv \exists c''.\ c \hookrightarrow \mathsf{cons}(x, c'') * seg(c'', c', xs)$
$seg(c, c', []) \equiv c = c'$

$iter(i, c, xs, P) \equiv \exists c', xs_1, xs_2.$
$(P \wedge (seg(c, c', xs_1)\ * coll(c', xs_2))) *$
$i \hookrightarrow c' \wedge$
$xs = xs_1 \cdot xs_2$

**Figure 2.** Iterator Implementation

state. In the subject/observer pattern, a number of observer objects of type $\tau_o$ each register for updates from a subject object of type $\tau_s$ by calling listen and passing a a notify function, which can update the observer's state. The subject can then broadcast a message to all of the observers attached to it, updating their states according to the message.

In this specification, we existentially quantify over the listen and broadcast functions (implemented by the subject), and introduce two predicates $sub$ and $obs$ to represent the state of the subject and individual observers.

The predicate $sub$ takes as arguments the subject whose state it describes, the list of observers it will notify (seq $\tau_o$), and a proposition variable $P$ tracking the current hidden state of the subject. The predicate $obs$ takes as arguments the observer object whose state it describes, and an integer representing the portion of the subject state being listened to. We also introduce two auxilliary predicates, $observers$ and $observers\_at$, which say that a sequence of observers exist, and that a sequence of observers all exist with the same value.

Note that we *universally* quantify over the $obs$ predicate. This ensures that a subject implementation must work for any possible implementation of $obs$. Conversely, a client program (which supplies the observed objects) will have the subject implementation as an existential hypothesis. That is, client code can assume that an instantiation of our subject implementation exists, but can make no assumptions about how it is implemented. In this way, we ensure that the state of the subject and the state of the observer are held abstract from one another.

The first clause of the specification is simply a function to create a new_subject, and the second is a triple asserting that broadcast on an empty list of observers has no visible effect.

The third clause, as an implication over triples, is the most complex part of this specification. It asserts that if a notify function updates the state of an observer $o$, and if broadcast correctly updates the state of a list of observers $os$, then calling listen$(s, o)$ will attach the observer $o$ to the list of observers, and

1 $\{coll(c, xs)\}$
2 letv $[b] = \texttt{empty}(c)$ in
3 $\{coll(c, xs)\}$
4 letv $[i_1] = \texttt{new\_iter}(c)$ in
5 $\{iter(i_1, c, xs, \top)\}$
6 $\{(coll(c, xs) \wedge P) * \forall R.\ (coll(c, xs) \wedge P \wedge R) \mathrel{-\!\!*} iter(i_1, c, xs, R)\}$
7 letv $[i_2] = \texttt{new\_iter}(c)$ in
8 $\{iter(i_2, c, xs, P) * \forall R.\ (coll(c, xs) \wedge P \wedge R) \mathrel{-\!\!*} iter(i_1, c, xs, R)\}$
9 $\{(coll(c, xs) \wedge P \wedge Q) *$
$\forall R.\ ((coll(c, xs) \wedge P \wedge R) \mathrel{-\!\!*} iter(i_1, c, xs, R)) *$
$\forall R.\ ((coll(c, xs) \wedge Q \wedge R) \mathrel{-\!\!*} iter(i_2, c, xs, R))\}$
10 letv $[b'] = \texttt{empty}(c)$ in
11 $\{(coll(c, xs) \wedge P \wedge Q)*$
$\forall R.\ ((coll(c, xs) \wedge P \wedge R) \mathrel{-\!\!*} iter(i_1, c, xs, R)) *$
$\forall R.\ ((coll(c, xs) \wedge Q \wedge R) \mathrel{-\!\!*} iter(i_2, c, xs, R))\}$
12 $\{iter(i_1, c, xs, Q))*$
$\forall R.\ ((coll(c, xs) \wedge Q \wedge R) \mathrel{-\!\!*} iter(i_2, c, xs, R))\}$
13 letv $[v] = \texttt{next}(i_1)$ in
14 $\{iter(i_1, c, xs, Q))*$
$\forall R.\ ((coll(c, xs) \wedge Q \wedge R) \mathrel{-\!\!*} iter(i_2, c, xs, R))\}$
15 $\{(coll(c, xs) \wedge P \wedge Q)*$
$\forall R.\ ((coll(c, xs) \wedge P \wedge R) \mathrel{-\!\!*} iter(i_1, c, xs, R)) *$
$\forall R.\ ((coll(c, xs) \wedge Q \wedge R) \mathrel{-\!\!*} iter(i_2, c, xs, R))\}$
16 $\{iter(i_2, c, xs, P))*$
$\forall R.\ ((coll(c, xs) \wedge P \wedge R) \mathrel{-\!\!*} iter(i_2, c, xs, R))\}$
17 letv $[v] = \texttt{next}(i_2)$ in
18 $\{iter(i_2, c, xs, P))*$
$\forall R.\ ((coll(c, xs) \wedge P \wedge R) \mathrel{-\!\!*} iter(i_2, c, xs, R))\}$
19 $\{(coll(c, xs) \wedge P \wedge Q)*$
$\forall R.\ ((coll(c, xs) \wedge P \wedge R) \mathrel{-\!\!*} iter(i_1, c, xs, R)) *$
$\forall R.\ ((coll(c, xs) \wedge Q \wedge R) \mathrel{-\!\!*} iter(i_2, c, xs, R))\}$
20 $\{coll(c, xs)*$
$\forall R.\ ((coll(c, xs) \wedge P \wedge R) \mathrel{-\!\!*} iter(i_1, c, xs, R))*$
$\forall R.\ ((coll(c, xs) \wedge Q \wedge R) \mathrel{-\!\!*} iter(i_2, c, xs, R))\}$
21 letv $[\_] = \texttt{add}(c, x)$ in
22 $\{coll(c, x :: xs)*$
$\forall R.\ ((coll(c, xs) \wedge P \wedge R) \mathrel{-\!\!*} iter(i_1, c, xs, R)) *$
$\forall R.\ ((coll(c, xs) \wedge Q \wedge R) \mathrel{-\!\!*} iter(i_2, c, xs, R))\}$

**Figure 3.** Iterator Client

$\texttt{broadcast}(s, n)$ will update all of the listeners $o :: os$ with the new value $n$.

This is an example of the general pattern for specifying higher-order programs with our specification logic; to specify the behavior of a function that takes a function as an argument, we write a formula that specifies the behavior of the function argument as a hypothesis of the application argument. An order-$n$ function will need a specification of at least order $n$.

We also demonstrate another technique in this specification. After a call to `listen`, the effect of the `broadcast` function changes, and we use the existential variable $Q$ to tie these two functions together – we assert that calling listen changes the state of the subject from $P$ to $Q$, and then give a triple describing how `broadcast` behaves on a subject in state $Q$.

### 2.3 Shared Buffer

In this example, we demonstrate how our separation logic can support *abstract ownership transfer*, which is useful in, e.g., the specification of a library of sparse matrix routines that use a common buffer, and where it is unsafe to call one operation until a sequence of calls implementing another is complete (as indicated by a return value of true).

Our shared buffer specification is shown in Figure 5. For simplicity, we just include two routines $\texttt{op}_1$ and $\texttt{op}_2$ operating on a shared buffer, described by $buf$; we assume `init` initializes the system so that $\texttt{op}_1$ can be called. The routine $\texttt{op}_1$ operates on a

$\exists \texttt{new\_subject}, \texttt{listen}, \texttt{broadcast}.$
$\exists sub : \tau_s \times \mathsf{seq}\ \tau_o \times \mathsf{prop} \Rightarrow \mathsf{prop}.$
$\forall obs : \tau_o \times \mathbb{N} \Rightarrow \mathsf{prop}.$

$\langle \top \rangle\ \texttt{new\_subject}()\ \langle a : \tau_s.\ sub(a, [], \top) \rangle$ and

$\forall s, P.\ \langle sub(s, [], P) \rangle\ \texttt{broadcast}(s, n)\ \langle a : \mathbf{1}.\ sub(s, [], P) \rangle$ and

$\forall s, os, o, \texttt{notify}, P.$
$(\forall n.\ \langle obs(o, -) \rangle\ \texttt{notify}(n)\ \langle a : \mathbf{1}.\ obs(o, n) \rangle)$
and
$\forall n.\ \langle sub(s, os, P) * observers(os) \rangle$
$\texttt{broadcast}(s, n)$
$\langle a : \mathbf{1}.\ sub(s, os, P) * observers\_at(os, n) \rangle)$
implies
$(\exists Q.\ \langle sub(s, os, P) * observers(os) * obs(o, -) \rangle$
$\texttt{listen}(s, \texttt{notify})$
$\langle a : \mathbf{1}.\ sub(s, o :: os, Q) * observers(o :: os) \rangle$
and
$\langle sub(s, o :: os, Q) * observers(o :: os) \rangle$
$\texttt{broadcast}(s, n)$
$\langle a : \mathbf{1}.\ sub(s, o :: os, Q) * observers\_at(o :: os, n) \rangle)$ and

$observers([]) = \top$
$observers(o :: os) = obs(o, -) * observers(os)$

$observers\_at([], n) = \top$
$observers\_at(o :: os, n) = obs(o, n) * observers\_at(os, n)$

**Figure 4.** Subject Observer Specification

$\exists m_1 : \tau_1 \times \mathsf{bool} \Rightarrow \mathsf{prop}.$
$\exists m_2 : \tau_2 \times \mathsf{bool} \Rightarrow \mathsf{prop}.$
$\exists buf : \mathsf{prop}.$

$\langle \top \rangle\ \texttt{init}()\ \langle (x_1, x_2).\ m_1(x_1, \mathsf{false}) * (buf \mathrel{-\!\!*} m_2(x_2, \mathsf{true})) \rangle$ and

$\forall x_1, b.\ \langle m_1(x_1, b) \rangle\ \texttt{op}_1(x_1)\ \langle r : \mathsf{bool}.\ m_1(x_1, r) \rangle$ and

$\forall x_2, b.\ \langle m_2(x_2, b) \rangle\ \texttt{op}_2(x_2)\ \langle r : \mathsf{bool}.\ m_2(x_2, r) \rangle$ and

$\forall x_1.\ \{m_1(x_1, \mathsf{true}) \supset (buf * (buf \mathrel{-\!\!*} m_1(x_1, \mathsf{true})))\}$ and

$\forall x_2.\ \{m_2(x_2, \mathsf{true}) \supset (buf * (buf \mathrel{-\!\!*} m_2(x_2, \mathsf{true})))\}$

**Figure 5.** Shared Buffer Specification

state $m_1(x, b)$, consisting of the data associated with $x$ and the shared buffer $buf$, and likewise for $\texttt{op}_2$. The static specifications assert that only when $m_1(x, \mathsf{true})$ holds, can the state can be split into the buffer and the remaining data $buf \mathrel{-\!\!*} m_1(x_1, \mathsf{true})$, and similarly for $m_2$.

The static specifications permit us to transfer ownership of $buf$, from $m_1$ to $m_2$ and vice versa, in precisely the states when the boolean field is true, and not otherwise, without knowing anything about the internal representation of the buffer.

## 3. Semantics

### 3.1 The Monadic Language

The programming language we work with is a monadic call-by-value lambda calculus. Following [10, 9], we stratify the language into two syntactic categories, pure *expressions* and effectful *computations*, which are described in Figure 6 (the $\omega$ used in the definition of contexts ranges over assertion types, to be defined in Figure 11 below).

Expressions are the purely functional fragment of the language, where evaluation always terminates and corresponds to the beta-rule of the lambda calculus. We type expressions using the typing judgement $\Gamma \vdash e : \tau$, described in Figure 7. Expressions include the usual introduction and elimination forms for sum, product, and function types. Additionally, we have reference types and monadic types.

The type $\mathsf{ref}\ \tau$ classifies references that point to a value of type $\tau$. Departing slightly from the usual presentation, pointer values take on the form $l_\tau$ — that is, a pointer is tagged with the type of the value it points to. This simplifies the semantics of the separation logic assertions, which will be described in the next section. A computation $c$ that produces a value of type $\tau$, can be suspended and turned into a first-class value $[c]$ of the monadic type $\bigcirc\tau$. A value of monadic type is a frozen computation and does not evaluate any further, which keeps side-effects from infecting the pure part of the language.

Neither reference values nor frozen computations have any elimination rules in the expression language, which ensures that no expression can have a side-effect. We make this syntactically apparent in the dynamic semantics of expressions, given in Figure 9, by simply leaving out the store altogether from its reduction relation ($e \rightsquigarrow e'$).

We have a judgement $\Gamma \vdash c \div \tau$, given in Figure 8, which characterizes well-formed computations. A computation is either an expression (which becomes a computation without side-effects); reading ($!e$), writing ($e := e'$), or creating ($\mathsf{new}_\tau e$) a reference; or sequencing two computations via monadic sequencing with the form $\mathsf{letv}\ [x] = e\ \mathsf{in}\ c$.[1] Sequencing takes an expression of type $\bigcirc\tau$, evaluates it to a value $[c']$, and then executes $c'$ and passes the result (a value plus side-effects) to $c$.

As an example, consider the following program, which is a computation of type `nat`.

$\mathsf{letv}\ [r] = [\mathsf{new}_{\mathsf{nat}}\mathsf{z}]\ \mathsf{in}$
$\mathsf{letv}\ [dummy] = [r := \mathsf{s}\ \mathsf{s}\ \mathsf{z}]\ \mathsf{in}$
$\mathsf{letv}\ [n] = [!r]\ \mathsf{in}$
$\quad \mathsf{s}\ n$

Here, we have a computation which creates a new pointer, pointing to zero, and then updates it to point to two, and then dereferences the pointer and returns the successor of the dereferenced value, for a final result of 3. We freeze basic commands and turn them into monadic values when we put them in brackets (e.g., $[!r]$), and then we can run them in sequential order using the monadic let-binding construct.

Although we have no explicit operator for term-level recursion, nontermination is possible in the computation language, because we have higher-order store – that is, pointers to functions. We can code an imperative fixed point if we update a function pointer so that a function body contains a pointer to itself (this is Landin's technique of "tying a knot in the heap" to construct a recursive function; see [15] for a concrete example). Thus the language includes non-termination. In this paper, however, we will not deal with reasoning about recursion.

We also have a judgement $\sigma\ \mathsf{ok}$ in Figure 8 to characterize well-typed heaps. Unlike the usual presentation of adding references to the lambda calculus [26], this judgement explicitly does *not* check whether or not there are dangling pointers in the heap – it permits dangling pointers as long as the pointers are themselves well-typed.

The reduction relation for commands, $\langle c\,;\ \sigma\rangle \rightsquigarrow \langle c'\,;\ \sigma'\rangle$, is given in Figure 10 and takes a *program state*, i.e., a pair of a computation and a heap, into another program state. If evaluation would read or write a pointer not in the heap, then we transition into the abort state $\langle c\,;\ \sigma\rangle \rightsquigarrow \mathbf{abort}$.

The decision to explicitly model what would happen with dangling pointers ends up substantially simplifying the semantics of the new connectives of the assertion language of separation logic, as we will see in the sequel. Informally, we use separation logic to reason about heap fragments, and the abort state lets the reduction relation "tell us" when a partial heap did not contain enough data for evaluation to proceed.

Also, one standard choice still worth drawing attention to is that dynamic allocation via $\mathsf{new}_\tau v$ is nondeterministic. The evaluation rule for allocation promises to return a new pointer not in the domain of the heap, but does not say what value that pointer will take on. This is important, because it is necessary in order for the *frame property* discussed below to hold.

The metatheory of this language is fairly standard. The only wrinkle is that we have to prove soundness twice, one for each of the two judgements for expressions and computations. We have the usual proof of type soundness for the expression language via progress and type preservation lemmas.

PROPOSITION 1 (Expression Progress). *If $\cdot \vdash e : \tau$, then either $e$ is a value or there exists an $e'$ such that $e \rightsquigarrow e'$.*

PROPOSITION 2 (Expression Subject Reduction). *If $\cdot \vdash e : \tau$ and $e \rightsquigarrow e'$, then $\cdot \vdash e' : \tau$.*

These are proved by structural induction on typing derivations and evaluation derivations, respectively.

Additionally, we also show that expressions always reduce to a value. We take $e \rightsquigarrow^* v$ to be the transitive closure of the one-step evaluation relation.

PROPOSITION 3 (Termination). *If $\cdot \vdash e : \tau$, then there exists a $v$ such that $e \rightsquigarrow^* v$.*

We prove this using a straightforward logical relations argument. The only interesting case is for `nat`, where we need a nested induction to show termination. Once we have soundness for our expression language, we can use it to prove soundness for computation terms.

PROPOSITION 4 (Partial Computation Progress). *If $\cdot \vdash c \div \tau$ and $\sigma\ \mathsf{ok}$, then either $c$ is a value $v$, or there exists a $c'$ and $\sigma'$ such that $\langle c\,;\ \sigma\rangle \rightsquigarrow \langle c'\,;\ \sigma'\rangle$, or $\langle c\,;\ \sigma\rangle \rightsquigarrow \mathbf{abort}$.*

PROPOSITION 5 (Computation Subject Reduction). *If $\cdot \vdash c \div \tau$ and $\sigma\ \mathsf{ok}$ and $\langle c\,;\ \sigma\rangle \rightsquigarrow \langle c'\,;\ \sigma'\rangle$, then $\cdot \vdash c' \div \tau$ and $\sigma' ok$.*

The progress lemma includes the possibility of the computation aborting if it tries to access a dangling pointer. We can restore the full safety of the conventional type-safety theorem if we introduce the notion of a *closed* state . We say a state $\langle c;\ \sigma\rangle$ is *closed* if all of the pointers in $c$ and in each of the values in the range of $\sigma$ are members of the domain of $\sigma$.

PROPOSITION 6 (Full Computation Progress). *If $\cdot \vdash c \div \tau$, $\sigma\ \mathsf{ok}$, and $\langle c;\ \sigma\rangle$ is closed, then either $c$ is a value $v$, or there exists a $c'$ and $\sigma'$ such that $\langle c\,;\ \sigma\rangle \rightsquigarrow \langle c'\,;\ \sigma'\rangle$ and $\langle c'\,;\ \sigma'\rangle$ is closed.*

Finally, we give an equality judgement for our programming language. This is beta-eta equality at function and product types and beta equality for sums, lists and natural numbers. The equality judgement for monadic terms is restricted so that no unrestricted computation or heap effects are needed; only the monad laws are included. The equality judgment is described in Figure 12, although the structural congruence rules and the rules for sums are elided for space reasons. This judgment can be extended to heaps, as well, as in Figure 13. Note that Propositions 1–3 together with the following

[1] This is equivalent to the `bind` operation in Haskell.

| | | | |
|---|---|---|---|
| Program Types | $\tau$ | ::= | $\mathbf{1} \mid \tau \times \tau' \mid \tau + \tau' \mid \tau \to \tau' \mid \mathsf{ref}\ \tau \mid \bigcirc \tau$ |
| | | $\mid$ | $\mathsf{nat} \mid \mathsf{list}$ |
| Expressions | $e$ | ::= | $() \mid (e_1, e_2) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e$ |
| | | $\mid$ | $\mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathsf{case}(e,\ x'.\ e',\ x''.\ e'')$ |
| | | $\mid$ | $\lambda x : \tau.\ e \mid x \mid e_1\ e_2 \mid l_\tau \mid [c]$ |
| | | $\mid$ | $\mathsf{z} \mid \mathsf{s}\ e \mid \mathsf{iter}_{\mathsf{nat}}(e, e_z, x.\ e_s)$ |
| | | $\mid$ | $\mathsf{nil} \mid \mathsf{cons}(e, e') \mid \mathsf{listcase}(e, e_1, (h, t).\ e_2)$ |
| Computations | $c$ | ::= | $e \mid \mathsf{letv}\ [x] = e\ \mathsf{in}\ c \mid !e \mid e := e'$ |
| | | $\mid$ | $\mathsf{new}_\tau e$ |
| Values | $v$ | ::= | $() \mid (v_1, v_2) \mid \mathsf{inl}\ e \mid \mathsf{inr}\ e$ |
| | | $\mid$ | $\mathsf{z} \mid \mathsf{s}\ v \mid \mathsf{nil} \mid \mathsf{cons}(v, l_{\mathsf{list}})$ |
| | | $\mid$ | $\lambda x : \tau.\ e \mid l_\tau \mid [c]$ |
| Contexts | $\Gamma$ | ::= | $\cdot \mid \Gamma, x : \tau \mid \Gamma, x : \omega$ |
| Heaps | $\sigma$ | ::= | $\cdot \mid \sigma, l_\tau : v$ |

**Figure 6.** The Basic Syntactic Categories

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash \lambda x : \tau'.\ e : \tau' \to \tau} \qquad \frac{\Gamma \vdash e : \tau' \to \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\ e' : \tau}$$

$$\frac{}{\Gamma \vdash \mathsf{nil} : \mathsf{list}} \qquad \frac{\Gamma \vdash e : \mathsf{nat} \quad \Gamma \vdash e' : \mathsf{ref\ list}}{\Gamma \vdash \mathsf{cons}(e, e') : \mathsf{list}}$$

$$\frac{\Gamma \vdash e : \mathsf{list} \quad \Gamma \vdash e_1 : \tau \quad \Gamma, h : \mathsf{nat}, t : \mathsf{ref\ list} \vdash e_2 : \tau}{\Gamma \vdash \mathsf{listcase}(e, e_1, (h, t).\ e_2) : \tau}$$

$$\frac{}{\Gamma \vdash \mathsf{z} : \mathsf{nat}} \qquad \frac{\Gamma \vdash e : \mathsf{nat}}{\Gamma \vdash \mathsf{s}\ e : \mathsf{nat}}$$

$$\frac{\Gamma \vdash e : \mathsf{nat} \quad \Gamma \vdash e_z : \tau \quad \Gamma, x : \tau \vdash e_s : \tau}{\Gamma \vdash \mathsf{iter}_{\mathsf{nat}}(e, e_z, x.\ e_s) : \tau}$$

$$\frac{}{\Gamma \vdash l_\tau : \mathsf{ref}\ \tau} \qquad \frac{\Gamma \vdash c \div \tau}{\Gamma \vdash [c] : \bigcirc \tau}$$

**Figure 7.** Expression Typing

Proposition 7 express that expressions modulo equality provide an *adequate* model of the operational semantics (if $\vdash\ e\ :\ \mathbf{1} + \mathbf{1}$ and $\vdash\ e \equiv \mathsf{inl}\ () : \mathbf{1}$, then $e$ evaluates to $\mathsf{inl}\ ()$). The equational theory can be derived from a standard denotational semantics for the programming language (omitted here); the latter can also be used to show that the equational theory is consistent.

PROPOSITION 7 (Evaluation Preserves Equality). *If* $\cdot \vdash e : \tau$ *and* $e \rightsquigarrow e'$, *then* $\cdot \vdash e \equiv e' : \tau$.

PROPOSITION 8 (Computation Preserves Equality). *If* $\cdot\ \vdash\ c_1\ \equiv\ c_2 \div \tau$, *and* $\vdash \sigma_1 \equiv \sigma_2$, *then*

1. *if* $\langle c_1\ ;\ \sigma_1 \rangle\ \rightsquigarrow\ \langle c_1';\ \sigma_1' \rangle$, *then* $\langle c_2;\ \sigma_2 \rangle\ \rightsquigarrow^*\ \langle c_2';\ \sigma_2' \rangle$ *and* $\cdot \vdash c_1' \equiv c_2' \div \tau$ *and* $\vdash \sigma_1' \equiv \sigma_2'$,
2. *if* $\langle c_1\ ;\ \sigma_1 \rangle \rightsquigarrow \mathbf{abort}$, *then* $\langle c_2;\ \sigma_2 \rangle \rightsquigarrow^* \mathbf{abort}$

$$\boxed{\begin{array}{l}\Gamma \vdash c \div \tau \\ \\ \vdash \sigma\ \mathsf{ok}\end{array}}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e \div \tau} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{new}_\tau e \div \mathsf{ref}\ \tau}$$

$$\frac{\Gamma \vdash e : \mathsf{ref}\ \tau}{\Gamma \vdash !e \div \tau} \qquad \frac{\Gamma \vdash e_1 : \mathsf{ref}\ \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 \div \mathbf{1}}$$

$$\frac{\Gamma \vdash e : \bigcirc \tau' \quad \Gamma, x : \tau' \vdash c \div \tau}{\Gamma \vdash \mathsf{letv}\ [x] = e\ \mathsf{in}\ c \div \tau}$$

$$\frac{}{\vdash \cdot\ \mathsf{ok}} \qquad \frac{\vdash \sigma\ \mathsf{ok} \quad \cdot \vdash v : \tau}{\vdash \sigma, l_\tau : v\ \mathsf{ok}}$$

**Figure 8.** Computation and Heap Typing

$$\boxed{e \rightsquigarrow e'}$$

$$\frac{e_1 \rightsquigarrow e_1'}{(e_1, e_2) \rightsquigarrow (e_1', e_2)} \qquad \frac{e_2 \rightsquigarrow e_2'}{(v, e_2) \rightsquigarrow (v, e_2')}$$

$$\frac{e \rightsquigarrow e'}{\mathsf{fst}\ e \rightsquigarrow \mathsf{fst}\ e'} \qquad \frac{e \rightsquigarrow e'}{\mathsf{snd}\ e \rightsquigarrow \mathsf{snd}\ e'}$$

$$\frac{}{\mathsf{fst}\ (v_1, v_2) \rightsquigarrow v_1} \qquad \frac{}{\mathsf{snd}\ (v_1, v_2) \rightsquigarrow v_2}$$

$$\frac{e_h \rightsquigarrow e_h'}{\mathsf{cons}(e_h, e_t) \rightsquigarrow \mathsf{cons}(e_h', e_t)} \qquad \frac{e_t \rightsquigarrow e_t'}{\mathsf{cons}(v, e_t) \rightsquigarrow \mathsf{cons}(v, e_t')}$$

$$\frac{e_1 \rightsquigarrow e_1'}{e_1\ e_2 \rightsquigarrow e_1'\ e_2} \qquad \frac{e_2 \rightsquigarrow e_2'}{v\ e_2 \rightsquigarrow v\ e_2'}$$

$$\frac{}{(\lambda x : \tau.\ e) v \rightsquigarrow [v/x]e} \qquad \frac{e \rightsquigarrow e'}{\mathsf{s}\ e \rightsquigarrow \mathsf{s}\ e'}$$

$$\frac{}{\mathsf{iter}_{\mathsf{nat}}(0, e_z, x.\ e_n) \rightsquigarrow e_z}$$

$$\frac{}{\mathsf{iter}_{\mathsf{nat}}(\mathsf{s}\ v, e_z, x.\ e_n) \rightsquigarrow [\mathsf{iter}_{\mathsf{nat}}(v, e_z, x.\ e_n)/x]e_n}$$

$$\frac{e \rightsquigarrow e'}{\mathsf{iter}_{\mathsf{nat}}(e, e_z, x.\ e_n) \rightsquigarrow \mathsf{iter}_{\mathsf{nat}}(e', e_z, x.\ e_n)}$$

$$\frac{e \rightsquigarrow e'}{\mathsf{listcase}(e, e_1, (h, t).\ e_2) \rightsquigarrow \mathsf{listcase}(e', e_1, (h, t).\ e_2)}$$

$$\frac{}{\mathsf{listcase}(\mathsf{nil}, e_1, (h, t).\ e_2) \rightsquigarrow e_1}$$

$$\frac{}{\mathsf{listcase}(\mathsf{cons}(v_h, l_{\mathsf{list}}), e_1, (h, t).\ e_2) \rightsquigarrow [v_h/h][l_{\mathsf{list}}/t]e_2}$$

**Figure 9.** Small-step Dynamic Semantics of Expressions

$$\boxed{\begin{array}{c}\langle c;\ \sigma\rangle \rightsquigarrow \langle c';\ \sigma'\rangle \\ \\ \langle c;\ \sigma\rangle \rightsquigarrow \mathbf{abort}\end{array}}$$

$$\frac{e \rightsquigarrow e'}{\langle \mathsf{letv}\ [x] = e\ \mathsf{in}\ c;\ \sigma\rangle \rightsquigarrow \langle \mathsf{letv}\ [x] = e'\ \mathsf{in}\ c;\ \sigma\rangle}$$

$$\frac{\langle c_1;\ \sigma\rangle \rightsquigarrow \langle c_1';\ \sigma'\rangle}{\langle \mathsf{letv}\ [x] = [c_1]\ \mathsf{in}\ c;\ \sigma\rangle \rightsquigarrow \langle \mathsf{letv}\ [x] = [c_1']\ \mathsf{in}\ c;\ \sigma'\rangle}$$

$$\frac{}{\langle \mathsf{letv}\ [x] = [v]\ \mathsf{in}\ c;\ \sigma\rangle \rightsquigarrow \langle [v/x]c;\ \sigma\rangle}$$

$$\frac{e \rightsquigarrow e'}{\langle e;\ \sigma\rangle \rightsquigarrow \langle e';\ \sigma\rangle}$$

$$\frac{e \rightsquigarrow e'}{\langle !e;\ \sigma\rangle \rightsquigarrow \langle !e';\ \sigma\rangle} \qquad \frac{e \rightsquigarrow e'}{\langle \mathsf{new}_\tau e;\ \sigma\rangle \rightsquigarrow \langle \mathsf{new}_\tau e';\ \sigma\rangle}$$

$$\frac{l_\tau : v \in \sigma}{\langle !l_\tau;\ \sigma\rangle \rightsquigarrow \langle v;\ \sigma\rangle} \qquad \frac{l_\tau \notin \mathsf{dom}(\sigma) \quad \sigma' = \sigma, l_\tau : v}{\langle \mathsf{new}_\tau v;\ \sigma\rangle \rightsquigarrow \langle l_\tau;\ \sigma'\rangle}$$

$$\frac{l_\tau \in \mathsf{dom}(\sigma) \quad \sigma' = [\sigma | l_\tau : v]}{\langle l_\tau := v;\ \sigma\rangle \rightsquigarrow \langle ();\ \sigma'\rangle} \qquad \frac{e_2 \rightsquigarrow e_2'}{\langle l_\tau := e_2;\ \sigma\rangle \rightsquigarrow \langle l_\tau := e_2';\ \sigma\rangle}$$

$$\frac{e_1 \rightsquigarrow e_1'}{\langle e_1 := e_2;\ \sigma\rangle \rightsquigarrow \langle e_1' := e_2;\ \sigma\rangle}$$

$$\frac{l_\tau \notin \mathsf{dom}(\sigma)}{\langle l_\tau := v;\ \sigma\rangle \rightsquigarrow \mathbf{abort}} \qquad \frac{l_\tau \notin \mathsf{dom}(\sigma)}{\langle !l_\tau;\ \sigma\rangle \rightsquigarrow \mathbf{abort}}$$

$$\frac{\langle c_1;\ \sigma\rangle \rightsquigarrow \mathbf{abort}}{\langle \mathsf{letv}\ [x] = [c_1]\ \mathsf{in}\ c;\ \sigma\rangle \rightsquigarrow \mathbf{abort}}$$

**Figure 10.** Dynamic Semantics of Computations

## 3.2 The Assertion Language

### 3.2.1 The Syntax of Assertions

The syntax of our assertion language is given in Figure 11. The types we can quantify over are the ordinary types of our programming language $\tau$, and additionally we introduce a type of propositions $\mathsf{prop}$, and a type constructor $\Rightarrow$. We stratify the assertion language so that program expressions can appear in assertions, but assertions cannot appear in program terms. This stratification permits us to give the assertion language a denotational, set-theoretic semantics (particularly for the function space $\Rightarrow$), while letting us continue to reason about the programming language in operational terms.

The term language is given by the ordinary term constructors of the programming language, extended to range over the propositional constants such as $\top$, $\bot$, $p * p'$ and so on. The fact that inductive types are available in the pure sublanguage means that we can define inductive predicates by iteration on some data. For example, the $observers$ function in Figure 4 could be defined as:

$$\lambda os : \mathsf{seq}\ \tau_o.\ \mathtt{iterseq}\ {}_{\tau_o}(os,\ \top,\ (o,t).\ \exists m : \mathbb{N}.obs(o,m) * t)$$

In the examples, we gave an inductive definition by cases, but this is sugar for a definition internal to our assertion language.

### 3.2.2 The Semantics of Assertions

To give the meaning of a proposition, we will make use of a *BI-algebra*[21, 27], which is a structure modelling logical entailment in the logic of bunched implications. A BI algebra is a lattice with Heyting algebra structure (operations $\top$, $\bot$, $\wedge$, $\vee$, and $\rightarrow$) that model the intuitionistic connectives of BI, and closed monoidal operations ($I$, $*$, and $-\!\!*$ ) to model the separating conjunction and separating implication.

First, we define an ordering on heaps such that one heap is above another if it is an extension of the other: $\sigma' \sqsubseteq \sigma$ if and only if $\exists \sigma''.\ \vdash \sigma \equiv \sigma', \sigma''$. Next, we take the set $H$ to be the set of well-typed heaps $H = \{\sigma \mid \vdash \sigma\ \mathsf{ok}\} / \equiv_\sigma$ quotiented by equality of heaps.[2]

We then define the lattice $\uparrow H$ to be the set of upward closed subsets of $H$, with the lattice ordering being subset inclusion.

$$\uparrow H = \left\{h \in \mathcal{P}(H) \mid \forall \sigma, \sigma'.\ \text{if } \sigma \in h \wedge \sigma \sqsubseteq \sigma' \text{ then } \sigma' \in h\right\}$$

Within this lattice, the lattice ordering defines logical entailment, and the BI-algebra operations are defined by:

$$\begin{aligned}
\top &= H \\
\bot &= \emptyset \\
h_1 \wedge h_2 &= h_1 \cap h_2 \\
h_1 \vee h_2 &= h_1 \cup h_2 \\
h_1 \supset h_2 &= \left\{\sigma \in H \mid \forall \sigma' \sqsupseteq \sigma.\ \text{if } \sigma' \in h_1 \text{ then } \sigma' \in h_2\right\} \\
I &= H \\
h_1 * h_2 &= \{\sigma \in H \mid \exists \sigma_1, \sigma_2.\ \ \sigma = \sigma_1 \uplus \sigma_2 \wedge \sigma_1 \# \sigma_2 \wedge \\
&\qquad\qquad \sigma_1 \in h_1 \wedge \sigma_2 \in h_2\} \\
h_1 -\!\!* h_2 &= \{\sigma \in H \mid \forall \sigma_1.\text{if } \sigma_1 \in h_1 \wedge \sigma_1 \# \sigma \text{ then } \sigma_1, \sigma \in h_2\}
\end{aligned}$$

Verifying that these definitions satisfy the equations of Heyting and monoidal structure is a routine calculation.

We will take the meaning a closed proposition to be an element of this lattice, which means that a proposition is interpreted as the set of heaps in which it is true. The restriction to downward-closed subsets of $H$ means that a proposition which is true at a heap will be true in all extensions of it, which is precisely what intuitionistic separation logic requires [13]. As a result of this requirement, truth $\top$ and the monoidal unit $I$ coincide, unlike in classical separation logic, where $I$ asserts the heap is empty.

Now, this structure is not sufficient to model full separation logic, because we can write propositions that include terms (such as $\Gamma \vdash e = e' : \mathsf{prop}$), and we do not yet have an interpretation of terms. To do this, we will make use of the notion of a *BI-hyperdoctrine*, originated by Biering, Birkedal and Torp-Smith [4]. A BI hyperdoctrine is a categorical structure which can interpret higher-order separation logic, but here we will sweep the abstract category theory under the carpet and work explicitly in a particular BI hyperdoctrine over sets and total functions.

Our interpretation consists of two parts. First, we give an intepretation for each of the types $\omega$. The type $\mathsf{prop}$ is interpreted as $\uparrow H$, and arrow ($\Rightarrow$) types are interpreted by the function space. The recursive type of sequences $\mathsf{seq}\ \omega$ is interpreted as the set of sequences of elements of $\omega$. The base types — which include all of the types of the programming language — are interpreted as the set of closed terms of the appropriate type, quotiented by equality.[3] The full definition can be seen in Figure 14.

[2] Hereafter we will abuse notation and write $\sigma \in H$ when we mean the equivalence class of heaps indexed by $\sigma$.

[3] Again, we will abuse notation and write $e \in [\![\omega]\!]$ to mean that the equivalence class identified by $e$ is a member of $[\![\omega]\!]$.

$$\begin{array}{llll} \text{Assertion Types} & \omega & ::= & \tau \mid \omega \Rightarrow \omega \mid \mathsf{seq}\ \omega \mid \mathsf{prop} \\ \text{Expressions} & p & ::= & e \mid \top \mid \bot \mid p \vee p' \mid p \wedge p' \\ & & \mid & p \supset p' \mid p * p' \mid p \mathrel{-\!\!*} p' \\ & & \mid & \forall x : \tau.\ p \mid \exists x : \tau.\ p \\ & & \mid & e =_\omega e' \mid e \hookrightarrow_\tau e' \\ & & \mid & \lambda x : \omega.\ p \mid p\ p' \mid x \\ & & \mid & [] \mid p :: ps \mid \mathsf{iter}_{\mathsf{seq}\ \omega}(p, p', (x, xs).\ p'') \\ & & \mid & \mathsf{iter}_{\mathsf{nat}}(e, p_z, x.\ p_s) \end{array}$$

$$\frac{\Gamma, x : \omega \vdash p : \mathsf{prop} \quad Q \in \{\forall, \exists\}}{\Gamma \vdash Qx : \omega.\ p : \mathsf{prop}} \qquad \frac{\mathbf{c} \in \{\top, \bot\}}{\Gamma \vdash \mathbf{c} : \mathsf{prop}}$$

$$\frac{\Gamma \vdash p_i : \mathsf{prop} \quad \mathbf{op} \in \{\wedge, \vee, \supset, \mathrel{-\!\!*}, *\}}{\Gamma \vdash p_1\ \mathbf{op}\ p_2 : \mathsf{prop}} \qquad \frac{\Gamma \vdash p : \omega \quad \Gamma \vdash p' : \omega}{\Gamma \vdash p =_\omega p' : \mathsf{prop}}$$

$$\frac{\Gamma \vdash e : \mathsf{ref}\ \tau \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e \hookrightarrow_\tau e' : \mathsf{prop}} \qquad \frac{}{\Gamma \vdash [] : \mathsf{seq}\ \omega}$$

$$\frac{\Gamma \vdash p : \omega \quad \Gamma \vdash p' : \mathsf{seq}\ \omega}{\Gamma \vdash p :: p' : \mathsf{seq}\ \omega} \qquad \frac{x : \omega \in \Gamma}{\Gamma \vdash x : \omega}$$

$$\frac{\Gamma \vdash p : \omega' \Rightarrow \omega \quad \Gamma \vdash p' : \omega'}{\Gamma \vdash p\ p' : \omega} \qquad \frac{\Gamma, x : \omega' \vdash p : \omega}{\Gamma \vdash \lambda x : \omega'.\ p : \omega' \Rightarrow \omega}$$

$$\frac{\Gamma \vdash p : \mathsf{seq}\ \omega' \quad \Gamma \vdash p_1 : \omega \quad \Gamma, h : \omega', t : \omega \vdash p_2 : \omega}{\Gamma \vdash \mathsf{iter}_{\mathsf{seq}\ \omega'}(p, p_1, (h, t).\ p_2) : \omega}$$

$$\frac{\Gamma \vdash e : \mathsf{nat} \quad \Gamma \vdash p_z : \omega \quad \Gamma, x : \omega \vdash p_s : \omega}{\Gamma \vdash \mathsf{iter}_{\mathsf{nat}}(e, p_z, x.\ p_s) : \omega}$$

**Figure 11.** Assertion Syntax

Next, we interpret terms in context (i.e., $\Gamma \vdash e : \omega$) as functions from the interpretation of the context to the interpretation of the expression $[\![\Gamma]\!] \to [\![\omega]\!]$, where a context $\Gamma = x_1 : \tau_1, \ldots x_n : \tau_n$ is interpreted by the set of substitutions $\gamma = [v_1/x_1, \ldots, v_n/x_n]$, with $v_1 \in [\![\tau_1]\!]$, $\ldots v_n \in [\![\tau_n]\!]$. The interpretation function is described in Figure 15. The definition makes use of the standard fold functions for natural numbers and sequences (primitive recursion).

Finally, we write logical entailment $p \rhd_\Gamma q$ to mean that $p$ entails $q$. That is, for all substitutions $\gamma$, if $\sigma \in [\![\Gamma \vdash p : \mathsf{prop}]\!]\gamma$ then $\sigma \in [\![\Gamma \vdash q : \mathsf{prop}]\!]\gamma$.

We verify that our semantics satisfies some basic properties:

PROPOSITION 9 (Substitution). *If* $\Gamma, x : \omega' \vdash p : \omega$, $\Gamma \vdash p' : \omega'$ *and* $\gamma \in [\![\Gamma]\!]$, *then*

$$[\![\Gamma \vdash [p'/x]p : \omega]\!]\gamma = [\![\Gamma, x : \omega' \vdash p : \omega]\!](\gamma, [\![\Gamma \vdash p' : \omega']\!]\gamma)$$

PROPOSITION 10 (Properties of Logical Entailment). *We have that*

$$\begin{array}{cc} \rhd_\Gamma I \supset T & \rhd_\Gamma T \supset I \\ \rhd_\Gamma p \supset p * I & \rhd_\Gamma p * I \supset p \\ (p * q) * r \rhd_\Gamma p * (q * r) & p * (q * r) \rhd_\Gamma (p * q) * r \\ p * q \rhd_\Gamma q * p & \end{array}$$

$$\frac{p \rhd_\Gamma q \quad r \rhd_\Gamma s}{p * r \rhd_\Gamma q * s} \qquad \frac{p * q \rhd_\Gamma r}{p \rhd_\Gamma q \mathrel{-\!\!*} r}$$

### 3.3 Semantics of Specifications

With a semantics for the assertion language in hand, we now consider how to construct a specification language from it. Our gen-

$$\boxed{\begin{array}{c} \Gamma \vdash e \equiv e' : \tau \\ \Gamma \vdash c \equiv c' \div \tau \end{array}}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e \equiv e : \tau} \qquad \frac{\Gamma \vdash e' \equiv e : \tau}{\Gamma \vdash e \equiv e' : \tau}$$

$$\frac{\Gamma \vdash e \equiv e' : \tau \quad \Gamma \vdash e' \equiv e'' : \tau}{\Gamma \vdash e \equiv e'' : \tau} \qquad \frac{}{\Gamma \vdash () \equiv () : \mathbf{1}}$$

$$\frac{\Gamma \vdash \lambda x : \tau'.\ e : \tau' \to \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash (\lambda x : \tau'.\ e)\ e' \equiv [e'/x]e : \tau} \qquad \frac{\Gamma \vdash e : \tau' \to \tau}{\Gamma \vdash e \equiv \lambda x : \tau'.\ e\ x : \tau' \to \tau}$$

$$\frac{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{fst}\ (e_1, e_2) \equiv e_1 : \tau_1} \qquad \frac{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{snd}\ (e_1, e_2) \equiv e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash (\mathsf{fst}\ e, \mathsf{snd}\ e) \equiv e : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e \equiv e' : \mathsf{nat}}{\Gamma \vdash \mathsf{s}\ e \equiv \mathsf{s}\ e' : \mathsf{nat}}$$

$$\frac{\Gamma \vdash \mathsf{iter}_{\mathsf{nat}}(\mathsf{z}, e_z, x.\ e_s) : \tau}{\Gamma \vdash \mathsf{iter}_{\mathsf{nat}}(\mathsf{z}, e_z, x.\ e_s) \equiv e_z : \tau} \qquad \frac{\Gamma \vdash c \equiv c' \div \tau}{\Gamma \vdash [c] \equiv [c'] : \bigcirc\tau}$$

$$\frac{\Gamma \vdash \mathsf{listcase}(\mathsf{nil}, e_1, (h, t).\ e_2) : \tau}{\Gamma \vdash \mathsf{listcase}(\mathsf{nil}, e_1, (h, t).\ e_2) \equiv e_1 : \tau}$$

$$\frac{\Gamma \vdash \mathsf{listcase}(\mathsf{cons}(e_h, e_t), e_1, (h, t).\ e_2) : \tau}{\Gamma \vdash \mathsf{listcase}(\mathsf{cons}(e_h, e_t), e_1, (h, t).\ e_2) \equiv [e_h/h][e_t/t]e_2 : \tau}$$

$$\frac{\Gamma \vdash \mathsf{iter}_{\mathsf{nat}}(\mathsf{s}\ e, e_z, x.\ e_s) : \tau}{\Gamma \vdash \mathsf{iter}_{\mathsf{nat}}(\mathsf{s}\ e, e_z, x.\ e_s) \equiv [\mathsf{iter}_{\mathsf{nat}}(e, e_z, x.\ e_s)/x]e_s : \tau}$$

$$\frac{\Gamma \vdash e \equiv e' : \tau}{\Gamma \vdash e \equiv e' \div \tau}$$

$$\frac{\Gamma \vdash c \div \tau}{\Gamma \vdash \mathsf{letv}\ [x] = [c]\ \mathsf{in}\ x \equiv c \div \tau}$$

$$\frac{\Gamma \vdash \mathsf{letv}\ [x] = [e]\ \mathsf{in}\ c' \div \tau}{\Gamma \vdash \mathsf{letv}\ [x] = [e]\ \mathsf{in}\ c' \equiv [e/x]c' \div \tau}$$

$$\frac{y \notin \mathrm{FV}(c'') \quad \Gamma \vdash \mathsf{letv}\ [x] = [\mathsf{letv}\ [y] = e\ \mathsf{in}\ c']\ \mathsf{in}\ c'' \div \tau}{\Gamma \vdash \begin{array}{l} \mathsf{letv}\ [x] = [\mathsf{letv}\ [y] = e\ \mathsf{in}\ c']\ \mathsf{in}\ c'' \equiv \\ \mathsf{letv}\ [y] = e\ \mathsf{in}\ \mathsf{letv}\ [x] = c'\ \mathsf{in}\ c'' \div \tau \end{array}}$$

**Figure 12.** Equational Theory

$$\boxed{\vdash \sigma \equiv \sigma'}$$

$$\frac{}{\vdash \cdot \equiv \cdot} \qquad \frac{\cdot \vdash v \equiv v' : \tau \quad \vdash \sigma_1, \sigma_2 \equiv \sigma_1', \sigma_2'}{\vdash \sigma_1, l_\tau : v, \sigma_2 \equiv \sigma_1', l_\tau : v', \sigma_2'}$$

**Figure 13.** Heap Equality

$$\begin{array}{rl} [\![\tau]\!] & = \{e \mid \cdot \vdash e : \tau\} / \equiv_\tau \\ [\![\omega_1 \Rightarrow \omega_2]\!] & = [\![\omega_2]\!]^{[\![\omega_1]\!]} \\ [\![\mathsf{seq}\ \omega]\!] & = [\![\omega]\!]^* \\ [\![\mathsf{prop}]\!] & = \uparrow H \end{array}$$

**Figure 14.** Semantics of Types

$$
\begin{aligned}
[\![\Gamma \vdash e : \tau]\!]\gamma &= \{e' \mid \forall x_i \in \mathsf{dom}(\Gamma),\ e_i \in [\gamma]x_i. \\
&\qquad e' = [\vec{e_i}/\vec{x_i}]e\}/\equiv_\tau \\
[\![\Gamma \vdash \lambda x : \omega'.\ e : \omega' \Rightarrow \omega]\!]\gamma &= \lambda v : [\![\omega']\!].\ [\![\Gamma, x : \omega' \vdash e : \omega]\!](\gamma, v/x) \\
[\![\Gamma \vdash e\ e' : \omega]\!]\gamma &= [\![\Gamma \vdash e : \omega' \to \omega]\!](\gamma)([\![\Gamma \vdash e' : \omega']\!](\gamma)) \\
[\![\Gamma \vdash x_i : \omega_i]\!]\gamma &= [\gamma]x_i \\
[\![\Gamma \vdash [] : \mathsf{seq}\ \tau]\!]\gamma &= \epsilon \\
[\![\Gamma \vdash e :: e' : \mathsf{seq}\ \tau]\!]\gamma &= ([\![\Gamma \vdash e : \tau]\!]\gamma) \cdot ([\![\Gamma \vdash e' : \mathsf{seq}\ \tau]\!]\gamma)
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma \vdash \top : \mathsf{prop}]\!]\gamma &= H \\
[\![\Gamma \vdash \bot : \mathsf{prop}]\!]\gamma &= \emptyset \\
[\![\Gamma \vdash p_1 \wedge p_2 : \mathsf{prop}]\!]\gamma &= [\![\Gamma \vdash p_1 : \mathsf{prop}]\!]\gamma \cap [\![\Gamma \vdash p_2 : \mathsf{prop}]\!]\gamma \\
[\![\Gamma \vdash p_1 \vee p_2 : \mathsf{prop}]\!]\gamma &= [\![\Gamma \vdash p_1 : \mathsf{prop}]\!]\gamma \cup [\![\Gamma \vdash p_2 : \mathsf{prop}]\!]\gamma \\
[\![\Gamma \vdash p_1 \supset p_2 : \mathsf{prop}]\!]\gamma &= [\![\Gamma \vdash p_1 : \mathsf{prop}]\!]\gamma \supset [\![\Gamma \vdash p_2 : \mathsf{prop}]\!]\gamma \\
[\![\Gamma \vdash p_1 * p_2 : \mathsf{prop}]\!]\gamma &= [\![\Gamma \vdash p_1 : \mathsf{prop}]\!]\gamma * [\![\Gamma \vdash p_2 : \mathsf{prop}]\!]\gamma \\
[\![\Gamma \vdash p_1 \mathrel{-\!\!*} p_2 : \mathsf{prop}]\!]\gamma &= [\![\Gamma \vdash p_1 : \mathsf{prop}]\!]\gamma \mathrel{-\!\!*} [\![\Gamma \vdash p_2 : \mathsf{prop}]\!]\gamma \\
[\![\Gamma \vdash \forall x : \tau.\ p : \mathsf{prop}]\!]\gamma &= \bigcap_{v \in [\![\tau]\!]} [\![\Gamma, x : \tau \vdash p : \mathsf{prop}]\!](\gamma, v/x) \\
[\![\Gamma \vdash \exists x : \tau.\ p : \mathsf{prop}]\!]\gamma &= \bigcup_{v \in [\![\tau]\!]} [\![\Gamma, x : \tau \vdash p : \mathsf{prop}]\!](\gamma, v/x) \\
[\![\Gamma \vdash p =_\omega p' : \mathsf{prop}]\!]\gamma &= [\![\Gamma \vdash p : \omega]\!]\gamma = [\![\Gamma \vdash p' : \omega]\!]\gamma \\
[\![\Gamma \vdash e \hookrightarrow_\tau e' : \mathsf{prop}]\!]\gamma &= \{h \in \uparrow H \mid \forall \sigma \in h.\ \exists l_\tau, v. \\
&\qquad l_\tau \in [\![\Gamma \vdash e : \mathsf{ref}\ \tau]\!]\gamma \wedge \\
&\qquad v \in [\![\Gamma \vdash e' : \tau]\!]\gamma \wedge \\
&\qquad l_\tau : v \in \sigma\}
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma \vdash \mathsf{iter}_{\mathsf{nat}}(e, p_z, x.\ p_s) : \omega]\!]\gamma = \mathrm{fold}_{\mathbb{N}} & \\
&[\![\Gamma \vdash p_z : \omega]\!]\gamma \\
&\lambda v : [\![\omega]\!]. \\
&\qquad [\![\Gamma, x : \omega \vdash p_s : \omega]\!](\gamma, v/x) \\
&[\![\Gamma \vdash e : \mathsf{nat}]\!]\gamma
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma \vdash \mathsf{iter}_{\mathsf{seq}\ \tau'}(e, e_1, (h, t).\ e_2) : \tau]\!]\gamma = & \\
\mathrm{fold}_{\mathsf{seq}\ \tau'}&[\![\Gamma \vdash e_1 : \tau]\!]\gamma \\
&\lambda v_h : [\![\tau']\!].\ \lambda v_t : [\![\tau]\!]. \\
&\qquad [\![\Gamma, h : \tau', t : \tau \vdash e_2 : \tau]\!](\gamma, v_h/h, v_t/t) \\
&[\![\Gamma \vdash e : \mathsf{seq}\ \tau']\!]\gamma
\end{aligned}
$$

**Figure 15.** Assertion Semantics

eral approach closely mirrors specification logic [29] for Algol, in which Reynolds took a Hoare logic for commands and integrated it with a call-by-name functional programming language. Our language is call-by-value, but the critical property we retain is that the full beta rule is a valid notion of equality (intuitively, the monadic type discipline ensures that function application won't have side effects).

To begin, we analyze Hoare triples and see what changes we will need to make. First, and most simply, a computation in our language will return a value in addition to having side effects. That is why the syntax for triples in our specification language is of the form $\{p\}\ \ c\ \ \{x : \tau.\ q\}$ – the $x$ is a binder for the value the computation will return, and should not shadow any variables in $p$ or $c$.

Secondly, it's not sufficient to give a semantics for triples as a relation between states, because free variables in a higher-order program might be bound to functions or suspended computations, which both contain code. Consider the following small program:

$\{\top\}$
$\mathsf{letv}\ [n] = fact(k)\ \mathsf{in}\ (k+1) \times n$
$\{a : \mathbb{N}.\ a = (k+1)!\}$

Whether this triple is true for this program depends on what value `fact` is bound to. So, we begin by giving the semantics of a triple as a function of its free variables. If $\Gamma \vdash \{p\}\ \ c\ \ \{x : \tau.\ q\} : \mathsf{spec}$, then we give meaning to the triple as a function of type-correct substitution $\gamma$:

$$
\begin{aligned}
[\![\Gamma \vdash \{p\}\ \ c\ \ \{x : \tau.\ q\} : \mathsf{spec}]\!]\gamma = & \\
&\forall \sigma, \sigma', v. \\
&\quad \text{if } \sigma \in [\![\Gamma \vdash p : \mathsf{prop}]\!]\gamma \text{ then} \\
&\qquad \langle [\gamma]c;\ \sigma\rangle \not\rightsquigarrow^* \mathbf{abort} \\
&\qquad \text{and if } \langle [\gamma]c;\ \sigma\rangle \rightsquigarrow^* \langle v;\ \sigma'\rangle \text{ then} \\
&\qquad\quad \sigma' \in [\![\Gamma, x : \tau \vdash q : \mathsf{prop}]\!](\gamma, v/x)
\end{aligned}
$$

Note that it is a partial correctness criterion.[4] Likewise, we give a semantics for triples over suspended monadic computations as follows:

$$
\begin{aligned}
[\![\Gamma \vdash \langle p\rangle\ e\ \langle x : \tau.\ q\rangle : \mathsf{spec}]\!]\gamma = & \\
&\forall \sigma, \sigma', c, v. \\
&\quad \text{if } \sigma \in [\![\Gamma \vdash p : \mathsf{prop}]\!]\gamma \\
&\qquad \text{and } [c] \in [\![\Gamma \vdash e : \textstyle\bigcirc\tau]\!]\gamma \text{ then} \\
&\qquad \langle c;\ \sigma\rangle \not\rightsquigarrow^* \mathbf{abort} \\
&\qquad \text{and if } \langle c;\ \sigma\rangle \rightsquigarrow^* \langle v;\ \sigma'\rangle \text{ then} \\
&\qquad\quad \sigma' \in [\![\Gamma, x : \tau \vdash q : \mathsf{prop}]\!](\gamma, v/x)
\end{aligned}
$$

This is essentially a duplicate of the previous semantic equation, but is needed for two reasons. First, a let-binding $\mathsf{letv}\ [x] = [c]\ \mathsf{in}\ c'$ in the monadic language will evaluate the expression of monadic type $[c]$ before substituting the value, so it is convenient to have a primitive triple that can be used directly with monadic expressions when giving an inference rule for sequencing. Secondly, all of the variables in our monadic language are expression variables; there are no variables that range over computations.

Finally, we define that a specification $\{p\}$ that just consists of an assertion $p$ is true when it holds for all heaps.

$$[\![\Gamma \vdash \{p\} : \mathsf{spec}]\!]\gamma = \forall \sigma.\ \sigma \in [\![\Gamma \vdash p : \mathsf{prop}]\!]\gamma$$

Once we have these basic triples, we can compose the triples themselves with logical connectives to construct more complex specifications, and inductively build up a meaning for the composite formulas:

$$
\begin{aligned}
[\![\Gamma \vdash S_1\ \mathsf{and}\ S_2 : \mathsf{spec}]\!]\gamma &= [\![\Gamma \vdash S_1 : \mathsf{spec}]\!]\gamma \text{ and } [\![\Gamma \vdash S_2 : \mathsf{spec}]\!]\gamma \\
[\![\Gamma \vdash S_1\ \mathsf{or}\ S_2 : \mathsf{spec}]\!]\gamma &= [\![\Gamma \vdash S_1 : \mathsf{spec}]\!]\gamma \text{ or } [\![\Gamma \vdash S_2 : \mathsf{spec}]\!]\gamma \\
[\![\Gamma \vdash S_1\ \mathsf{implies}\ S_2 : \mathsf{spec}]\!]\gamma &= \text{if } [\![\Gamma \vdash S_1 : \mathsf{spec}]\!]\gamma \text{ then } [\![\Gamma \vdash S_2 : \mathsf{spec}]\!]\gamma \\
[\![\Gamma \vdash \forall x : \omega.\ S : \mathsf{spec}]\!]\gamma &= \forall v \in [\![\omega]\!].\ [\![\Gamma, x : \omega \vdash S : \mathsf{spec}]\!](\gamma, v/x) \\
[\![\Gamma \vdash \exists x : \omega.\ S : \mathsf{spec}]\!]\gamma &= \exists v \in [\![\omega]\!].\ [\![\Gamma, x : \omega \vdash S : \mathsf{spec}]\!](\gamma, v/x)
\end{aligned}
$$

[4] Technically elements of $[\![\mathsf{prop}]\!]$ are sets of equivalence classes of heaps, but we disregard that to avoid cluttering this definition.

Thus, we have turned triples into the atomic propositions of yet another logic. This means that we have a two-level logic, in which we have propositions of separation logic appearing in triples, and the triples themselves are propositions in another logic. This permits us to characterize the behavior of free variables in a specification. For example, revisiting the factorial example, we might write:

$$\begin{array}{l}(\forall m : \mathbb{N}.\ \langle \top \rangle\ fact(m)\ \langle a : \mathbb{N}.\ a = m! \rangle) \\ \mathsf{implies}\ \ \{\top\} \\ \qquad \mathsf{letv}\ [n] = fact(k)\ \mathsf{in}\ (k+1) \times n \\ \qquad \{a : \mathbb{N}.\ a = (k+1)!\}\end{array}$$

We can read this specification as saying, "If `fact` computes the factorial function, then the consequent will compute $(k+1)!$." However, we still have free variables in this spec, and we must ask under what circumstances we can use such a specification – are there implementations of `fact` or values of `k` for which the specification will be falsified?

The approach we will take towards answering this question is to come up with inference rules for deriving specifications which remain true in all type-correct substitutions. Such specifications are said to be *valid* (called "universal" by Reynolds [29]). For a well-formed specification $\Gamma \vdash S : \mathsf{spec}$, we say:

$$\Gamma \vdash S : \mathsf{spec}\ \text{is valid} \quad \text{iff} \quad \forall \gamma. [\![\Gamma \vdash S : \mathsf{spec}]\!]\gamma$$

In Figures 17 and 18, we give a collection of deduction rules for inferring valid specifications. The rules are written in a sequent style, but semantically a sequent of the form $\Gamma; \Delta \vdash S$, where $\Delta = S_1, \ldots S_n$, is interpreted as the specification $(S_1\ \mathsf{and}\ \ldots\ \mathsf{and}\ S_n)\ \mathsf{implies}\ S$ with free variables in $\Gamma$.

PROPOSITION 11 (Soundness of Specification Logic). *Every derivation $\Gamma; \Delta \vdash S$ using the rules in Figures 17 and 18 derives a valid specification.*

We prove this with an induction on the derivation. Most of the cases are routine, except for the base cases, the rule Frame, and the Substitution rule.

The base cases can all be proven with a straightforward appeal to the semantics, but are noteworthy because they are all "small" or "tight". In O'Hearn's terminology [22], they enable local reasoning because the preconditions and postconditions refer to no other pointers than the ones that are accessed. To prove the Frame rule, we must show that the operational semantics validates the *safety monotonicity* and *frame* properties [23].

PROPOSITION 12. *The safety monotonicity and frame properties are:*

1. *If $\cdot \vdash c \div \tau$ and $\vdash \sigma$ ok, then if $\langle c;\ \sigma\rangle \not\rightsquigarrow^*$ **abort**, then if $\sigma \sqsubseteq \sigma'$ and $\vdash \sigma'$ ok, then $\langle c;\ \sigma'\rangle \not\rightsquigarrow^*$ **abort**.*
2. *For all $\cdot \vdash c \div \tau$ and $\vdash \sigma_0, \sigma_1$ ok, if $\langle c;\ \sigma_0\rangle \not\rightsquigarrow^*$ **abort**, and $\langle c\,;\ \sigma_0, \sigma_1\rangle \rightsquigarrow \langle v;\ \sigma'\rangle$, then there exists a $\sigma_0'$ such that $\sigma' \equiv \sigma_0', \sigma_1$ and $\langle c\,;\ \sigma_0\rangle \rightsquigarrow \langle v;\ \sigma_0'\rangle$.*

Informally, safety monotonicity is the property that if a particular program and heap do not evaluate to an abort, then that program will not abort with any extension of the heap, and the frame property is the local reasoning property – a program will not modify any state outside of its footprint.

The Substitution rule arises from the basic substitution principle that if $\Gamma, x : \tau' \vdash e : \tau$ and $\Gamma \vdash e' : \tau'$, then $\Gamma \vdash [e'/x]e : \tau$, lifted first through the syntax of assertions and then lifted again to specifications. This rule is significant, because it is the combination of the frame rule and the substitution property that enables genuinely modular reasoning about imperative programs.

$$\begin{array}{rcl} S & ::= & \{p\}\ c\ \{x : \tau.\ q\} \mid \langle p\rangle\, e\, \langle x : \tau.\ q\rangle \mid \{p\} \\ & \mid & S\ \mathsf{and}\ S \mid S\ \mathsf{or}\ S' \mid S\ \mathsf{implies}\ S' \\ & \mid & \forall x : \tau.\ S \mid \exists x : \tau.\ S \\ \Delta & ::= & \cdot \mid \Delta, S \end{array}$$

$$\frac{\Gamma \vdash p : \mathsf{prop} \quad \Gamma \vdash c \div \tau \quad \Gamma, x : \tau \vdash q : \mathsf{prop}}{\Gamma \vdash \{p\}\ c\ \{x : \tau.\ q\} : \mathsf{spec}}$$

$$\frac{\Gamma \vdash p : \mathsf{prop} \quad \Gamma \vdash e : \bigcirc\tau \quad \Gamma, x : \tau \vdash q : \mathsf{prop}}{\Gamma \vdash \langle p\rangle\, e\, \langle x : \tau.\ q\rangle : \mathsf{spec}}$$

$$\frac{\Gamma \vdash S : \mathsf{spec} \quad \Gamma \vdash S' : \mathsf{spec} \quad \mathrm{op} \in \{\text{and, or, implies}\}}{\Gamma \vdash S\ \mathrm{op}\ S' : \mathsf{spec}}$$

$$\frac{\Gamma, x : \tau \vdash S : \mathsf{spec} \quad \mathrm{Q} \in \{\forall, \exists\}}{\Gamma \vdash \mathrm{Q}x : \tau.\ S : \mathsf{spec}} \qquad \frac{\Gamma \vdash p : \mathsf{prop}}{\Gamma \vdash \{p\} : \mathsf{spec}}$$

**Figure 16.** Well-Formedness of Specifications

As seen in the examples, we prove client programs that use an imperative module by taking a hypothetical specification of the form $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n; \Delta, S_i \vdash S_c$. Here, we take $S_i$ to be the signature of the module, naming the data and operations in its interface with the variables. $S_i$ does not have to mention any of the client's data – whenever $S_c$ uses an operation from $S_i$, it can use the Frame rule to assert that its data is untouched.

Then, we can derive a concrete implementation of that module, if we prove a program with a specification $\Gamma; \Delta \vdash \exists x_1 : \tau_1, \ldots, \exists x_n : \tau_n.\ S_i$, using the local reasoning property to consider only the module's data. Then, we can compose those pieces using the existential elimination rule.

The existential introduction and elimination rules are only provable because the programming language is consistent with the beta-reduction rule of the lambda calculus, and so permits us to freely substitute expressions for variables without changing the meaning of a program.

As a result, a crucial part of the success of this methodology arises from the clear separation of the language into pure and imperative parts via a monad. This is what lets us use the obvious substitution principle, which allows our simple method of combining specifications to work correctly.

Having a completely pure and total sublanguage also greatly simplifies the assertion language: there is no need for definedness predicates to assert that a term terminates, nor for nested specifications within assertions, to cope with possibly effectful expressions within an assertion. That is what permits us to stratify the specification language, which drastically simplifies its formulation.

In fact, we conjecture that when adapting this methodology to impure languages like SML or Java, it will be greatly helpful to use a Moggi-style monadic translation [17] to *create* such a stratification.

## 4. Related and Future Work

In spirit, this work is a direct descendent of Reynolds' work on specification logic for Idealized Algol [29]. It was the two observations that the full beta-rule is a valid reasoning principle in Idealized Algol, and that Algol's command type completely separates the imperative and functional parts of the language, that spawned the hypothesis that the same idea could apply to a monadic language. This combination turned out to be especially pleasant to work with, because the decision to remove assignable variables and put all aliasing into the heap, meant that we could simply dispense with the complex interference conditions of specification

$$\boxed{\Gamma; \Delta \vdash S}$$

$$\frac{\Gamma \vdash \mathsf{new}_\tau e \div \mathsf{ref}\ \tau}{\Gamma; \Delta \vdash \{\top\}\ \mathsf{new}_\tau e\ \{x : \mathsf{ref}\ \tau.\ x \hookrightarrow e\}}\ New$$

$$\frac{\Gamma \vdash !e \div \tau \quad \Gamma \vdash e' : \tau}{\Gamma; \Delta \vdash \{P \wedge e \hookrightarrow_\tau e'\}\ !e\ \{x : \tau.\ P \wedge e \hookrightarrow_\tau e' \wedge x = e'\}}\ Deref$$

$$\frac{\Gamma \vdash e := e'' \div \mathbf{1}}{\Gamma; \Delta \vdash \{e \hookrightarrow -\}\ e := e''\ \{x : \mathbf{1}.\ e \hookrightarrow e''\}}\ Assign$$

$$\frac{\begin{array}{l}\Gamma; \Delta \vdash \langle p \rangle\, e\, \langle x : \tau.\ q \rangle \\ \Gamma, x : \tau; \Delta \vdash \{q\}\ c\ \{y : \tau'.\ r\} \quad x \notin \mathrm{FV}(r, \Delta)\end{array}}{\Gamma; \Delta \vdash \{p\}\ \mathsf{letv}\ [x] = e\ \mathsf{in}\ c\ \{y : \tau'.\ r\}}\ Seq$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma; \Delta \vdash \{\top\}\ e\ \{x : \tau.\ x = e\}}\ Pure$$

$$\frac{\Gamma; \Delta \vdash \{p\}\ c\ \{x : \tau.\ q\}}{\Gamma; \Delta \vdash \langle p \rangle\, [c]\, \langle x : \tau.\ q \rangle}\ Monad$$

$$\frac{\Gamma; \Delta \vdash \{p\}\ c'\ \{x : \tau.\ q\} \quad \Gamma \vdash c \equiv c' \div \tau}{\Gamma; \Delta \vdash \{p\}\ c\ \{x : \tau.\ q\}}\ CompEq$$

$$\frac{\Gamma; \Delta \vdash \langle p \rangle\, e'\, \langle x : \tau.\ q \rangle \quad \Gamma \vdash e \equiv e' : \bigcirc \tau}{\Gamma; \Delta \vdash \langle p \rangle\, e\, \langle x : \tau.\ q \rangle}\ MonadEq$$

$$\frac{\Gamma; \Delta \vdash \{p\}\ c\ \{x : \tau.\ q\} \quad \Gamma \vdash r : \mathsf{prop}}{\Gamma; \Delta \vdash \{p * r\}\ c\ \{x : \tau.\ q * r\}}\ Frame1$$

$$\frac{\Gamma; \Delta \vdash \langle p \rangle\, e\, \langle x : \tau.\ q \rangle \quad \Gamma \vdash r : \mathsf{prop}}{\Gamma; \Delta \vdash \langle p * r \rangle\, e\, \langle x : \tau.\ q * r \rangle}\ Frame2$$

$$\frac{p \rhd_\Gamma p' \quad \Gamma; \Delta \vdash \{p'\}\ c\ \{x : \tau.\ q'\} \quad q' \rhd_{\Gamma, x:\tau} q}{\Gamma; \Delta \vdash \{p\}\ c\ \{x : \tau.\ q\}}\ Consequence$$

**Figure 17.** Valid Specification Inference Rules

logic, which described whether variables in two expressions might interfere with one another. Instead, we could use separation logic to reason about aliasing.

Birkedal, Torp-Smith, and Yang [7] developed a version of separation logic for a version of Algol with immutable variables (as we also have here), where the heap is only a first-order map of integers (unlike here, where any value can be stored in the heap). This system doesn't distinguish the type system from the specification language: command types can contain preconditions and postconditions written in separation logic in a fashion similar to refinement types. Thus they only give semantics to well-specified programs. The assertion language they support is first-order (no quantification over assertions), but they support a very powerful kind of hypothetical frame rule in their language, extending the second-order frame rule of [23] to higher-order. Future work includes combining the higher-order frame rules of *loc. cit.* with the separation logic for higher-order store that we have presented here.

In [18], Nanevski et al. develop a Polymorphic Hoare Type Theory, which integrates the type system and the specification language, as in [7], but for an ML language with higher-order store and polymorphism. Nanevski et al. also make use of a monadic presentation of the language to separate the imperative and functional parts of the language. The assertion language of *loc. cit.* is first-order, which makes it difficult to express specifications of imperative modules such as those presented here in Section 2. It remains to be seen whether the pure type-theoretic approach in-

$$\boxed{\Gamma; \Delta \vdash S}$$

$$\frac{}{\Gamma; \Delta, S \vdash S}\ Hyp$$

$$\frac{\Gamma; \Delta \vdash S' \quad \Gamma; \Delta, S' \vdash S}{\Gamma; \Delta \vdash S}\ Cut$$

$$\frac{\Gamma, x : \omega; \Delta \vdash S \quad \Gamma \vdash p : \omega}{\Gamma; [p/x]\Delta \vdash [p/x]S}\ Substitution$$

$$\frac{\Gamma; \Delta \vdash S \quad \Gamma; \Delta \vdash S'}{\Gamma; \Delta \vdash S\ \mathsf{and}\ S'}\ AndIntro$$

$$\frac{\Gamma; \Delta, S_1\ \mathsf{and}\ S_2 \vdash S}{\Gamma; \Delta, S_1, S_2 \vdash S}\ AndElim$$

$$\frac{\Gamma; \Delta \vdash S_1 \quad \Gamma \vdash S_2 : \mathsf{spec}}{\Gamma; \Delta \vdash S_1\ \mathsf{or}\ S_2}\ OrIntro1$$

$$\frac{\Gamma; \Delta \vdash S_2 \quad \Gamma \vdash S_1 : \mathsf{spec}}{\Gamma; \Delta \vdash S_1\ \mathsf{or}\ S_2}\ OrIntro2$$

$$\frac{\Gamma; \Delta, S_1 \vdash S \quad \Gamma; \Delta, S_2 \vdash S}{\Gamma; \Delta, S_1\ \mathsf{or}\ S_2 \vdash S}\ OrElim$$

$$\frac{\Gamma; \Delta, S' \vdash S}{\Gamma; \Delta \vdash S'\ \mathsf{implies}\ S}\ ImpIntro$$

$$\frac{\Gamma; \Delta, S_1\ \mathsf{implies}\ S_2 \vdash S_1 \quad \Gamma; \Delta, S_1\ \mathsf{implies}\ S_2, S_2 \vdash S}{\Gamma; \Delta, S_1\ \mathsf{implies}\ S_2 \vdash S}\ ImpElim$$

$$\frac{\Gamma, x : \omega; \Delta \vdash S \quad x \notin \mathrm{FV}(\Delta)}{\Gamma; \Delta \vdash \forall x : \omega.\ S}\ UnivIntro$$

$$\frac{\Gamma, x : \omega; \Delta, \forall x : \omega.\ S', S' \vdash S \quad x \notin \mathrm{FV}(\Delta, S)}{\Gamma, \Delta, \forall x : \omega.\ S' \vdash S}\ UnivElim$$

$$\frac{\Gamma; \Delta \vdash [p/x]S \quad \Gamma \vdash p : \omega}{\Gamma; \Delta \vdash \exists x : \omega.\ S}\ ExistIntro$$

$$\frac{\Gamma, x : \omega; \Delta, \exists x : \omega.\ S', S' \vdash S \quad x \notin \mathrm{FV}(\Delta, S)}{\Gamma; \Delta, \exists x : \omega.\ S' \vdash S}\ ExistElim$$

$$\frac{\Gamma; \Delta \vdash [\mathsf{z}/x]S \quad \Gamma, x : \mathsf{nat}; \Delta, S \vdash [\mathsf{s}\ x/x]S \quad x \notin \mathrm{FV}(\Delta)}{\Gamma; \Delta \vdash \forall x : \mathsf{nat}.\ S}\ NatInd$$

$$\frac{\begin{array}{l}\Gamma; \Delta \vdash [\mathsf{nil}/x]S \\ \Gamma, h : \omega, x : \mathsf{seq}\ \omega; \Delta, S \vdash [\mathsf{cons}(h, x)/x]S \quad h, x \notin \mathrm{FV}(\Delta)\end{array}}{\Gamma; \Delta \vdash \forall x : \mathsf{seq}\ \omega.\ S}\ SeqInd$$

**Figure 18.** Universal Specifications, Continued

tegrating types and specifications is more suitable in practice than the more traditional approach taken here where types and specifications are kept separate. Future work further includes investigating how to extend [18] with a higher-order assertion language, and how to extend the present language with polymorphism.

In [31], Mandelbaum et al. develop a refinement type system to track imperative effects. Their system also distinguishes expressions and computations (though they don't internalize the distinction with a monad), and they use a fragment of linear logic to describe program effects. As a consequence, they cannot reason about sharing, though they do have tractable typechecking. The refinement type methodology bears some similarity to the way we layer separation logic over a typed base language, so this work may offer some guidance relating the current work to Hoare Type Theory.

Parkinson [24] has built a version of separation logic for a programming language embodying a core subset of Java. As in this work, he used an intuitionistic variant of separation logic. However, the lack of pure higher-order functions sharply limits the program expressions that can appear in assertions. Parkinson and Biermann introduces the notion of an abstract predicate, in which a kind of assertion variable is used to conceal the concrete predicate used to implement an object's state[25]. Our version of separation logic supports information hiding by means of existentially quantified assertions, as proposed in [5]. The latter paper also develops a specification logic using valid specifications for a simple first-order programming language.

Reus and Schwinghammer [28] describe an interesting denotational semantics of first-order separation logic for higher-order store. Since their language does not include pure higher-order functions in expressions and their assertion logic is only first-order, the resulting system is however not as expressive as the one presented here.

Berger, Honda and Yoshida recently described a new logic for analysing aliasing in imperative functional programs [3]. Like separation logic, they added connectives to a first-order logic, but their new additions are modal operators that can be interpreted as quantification over the possible content of a pointer. Also, they work in a setting where effects are not confined to a monad, and their operators require explicitly require tracking the write set of a procedure. They noted that one of the difficulties that hindered comparison with separation logic is that the languages the two approaches targetted were so different; we hope the current work will enable a better point of comparison.

Barnett and Naumann [2] propose to use a *friendship* system for verification of ownership-based invariants in object-oriented programs. Their method builds on the *Boogie* methodology, which combines ownership-based invariants [1] with auxiliary owner fields [16]. The friendship discipline supports modular verification of cooperating classes. The auxiliary fields used in the friendship discipline appear to correspond very closely to (some of) the arguments of our existentially quantified propositions, for example, the boolean arguments of the $m_1$ and $m_2$ predicates in our shared buffer example. We have used our higher-order separation logic to specify some of the examples from [2] (including the Master-Clock example with Reset) and it seems that our higher-order separation logic can express the invariants expressible in the friendship discipline. Indeed we conjecture that variants of our higher-order separation logic may be used as a general logical tool for specification and verification of imperative modules consisting of shared mutable objects. Future work includes testing this hypothesis more extensively, e.g., by verifying parts of the C5 collection library [14], and extending the higher-order separation logic to accomodate subtyping and other features from object-oriented programming.

## Acknowledgments

We gratefully acknowledge comments and suggestions on the exposition from Peter Sestoft.

Krishnaswami's and Reynolds' research was partially supported by National Science Foundation grants CCR-0204242, CCF-0541021. Krishnaswami's and Aldrich' research was partially supported by NASA cooperative agreement NNA05CS30A.

## References

[1] M. Barnett, R. DeLine, M. Fähndrich, K. Leino, and W. Schulte. Verification of object-orietned programs with invariants. In S. Eisenbach, G. Leavens, P. Müller, A. Poetzsch-Heffter, and E. Poll, editors, *Formal Techniques for Java-like Programs 2003*, July 2003.

[2] M. Barnett and D. Naumann. Friends need a bit more: Maintaining invariants over shared shate. In *Proceedings of Mathematics of Program Construction 2004*, 2004.

[3] M. Berger, K. Honda, and N. Yoshida. A logical analysis of aliasing in imperative higher-order functions. In *Proceedings of 10th ACM-SIGPLAN International Conference on Functional Programming (ICFP 05)*, 2005.

[4] B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines and higher order separation logic. In *Proc. of ESOP 2005: The European Symposium on Programming*, pages 233–247, Edinburgh, Scotland, April 2005.

[5] B. Biering, L. Birkedal, and N. Torp-Smith. Bi hyperdoctrines, higher-order separation logic, and abstraction. Technical Report ITU-TR-2005-69, IT University of Copenhagen, Copenhagen, Denmark, July 2005.

[6] L. Birkedal, N. Torp-Smith, and J. Reynolds. Local reasoning about a copying garbage collector. In *Proc. 31-st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, pages 220–231. ACM Press, January 2004.

[7] L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules. In *Proceedings of the 20th IEEE Symposium on Logic in Computer Science (LICS'05)*, pages 260–269, Chicago, IL, USA, June 2005.

[8] R. Bornat, C. Calcagno, and P. O'Hearn. Local reasoning, separation and aliasing. In *Proceedings of SPACE 2004*, Venice, Italy, January 2004.

[9] R. Davies and F. Pfenning. A judgemental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.

[10] A. Filinski. *Controlling Effects*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1996.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of reusable object-oriented software*. Addison-Wesley, 1995.

[12] C. A. R. Hoare. An axiomatic approach to computer programming. *Communications of the ACM*, 12(583):576–580, 1969.

[13] S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of the 28th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL'01)*, London, 2001.

[14] N. Kokholm and P. Sestoft. The c5 collection library for c# for cli. Technical Report ITU–TR–2006–76, IT University of Copenhagen, Jan. 2006. 252 pages.

[15] N. Krishnaswami. Separation logic for a higher-order typed language. Unpublished draft. Presented at SPACE 2006 workshop.

[16] K. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *ECOOP 2004—Object-oriented Programming, 18th European Conference, Proceedings*, volume 3086 of *LNCS*, jun 2004.

[17] E. Moggi. Notions of computation and monads. *Information and*

*Computation*, 93(1):55–92, 1991.

[18] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in hoare type theory. In *Proceedings of International Conference on Functional Programming 2006*, 2006. To Appear.

[19] P. O'Hearn. Resources, concurrency, and local reasoning. In *Proceedings of CONCUR 2004*, 2004.

[20] P. O'Hearn. A semantics for concurrent separation logic. In *Proceedings of CONCUR 2004*, 2004.

[21] P. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2), June 1999.

[22] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL 2001)*, Berlin, Germany, September 2001.

[23] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proceedings of the 31st ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL'04)*, pages 268–280, Venice, Italy, 2004.

[24] M. Parkinson. *Local Reasoning for Java*. PhD thesis, Cambridge University, 2005.

[25] M. Parkinson and G. Bierman. Separation logic and abstraction. In *Proceedings of the 32nd Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL'05)*, pages 247–258, Long Beach, CA, USA, January 2005.

[26] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[27] D. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*, volume 26 of *Applied Logics Series*. Kluwer, 2002.

[28] B. Reus and J. Schwinghammer. Separation logic for higher-order store. In *Proceedings of CSL 2006*, 2006.

[29] J. Reynolds. Idealized algol and its specification logic. In P. O'Hearn and R. Tennent, editors, *ALGOL like languages*, volume 1. Birkhäuser, 1997.

[30] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 55–74, Copenhagen, Denmark, July 2002. IEEE Press.

[31] R. H. Y. Mandelbaum, D. Walker. An effective theory of type refinements. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 213–225, Uppsala, Sweden, 2003.

[32] H. Yang. *Local Reasoning for Stateful Programs*. PhD thesis, University of Illinois, Urbana-Champaign, 2001.