Separation Logic for a Higher-Order Typed Language

[Extended Abstract]

Neelakantan Krishnaswami
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
neelk+@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 create an extension of separation logic that permits effective, modular reasoning about typed, higher-order functional programs that use aliased mutable heap data, including pointers to code.

Categories and Subject Descriptors

D.3.0 [**Programming Languages**]: General; F.3.1 [**Logic and Meaning of Programs**]: Specifying and Verifying and Reasoning About Programs

General Terms

Separation Logic

Keywords

Separation logic, specification logic, imperative, functional, aliasing, monads, ML, call-by-value

1. INTRODUCTION

Traditionally, pointer programs – programs which make use of aliased, mutable data values – have been difficult to analyze, for two reasons. First, aliasing allows an assignment to a pointer to nonlocally affect other variables that contain the same pointer. This requires specifications to explicitly enumerate all the possible interference patterns between mutable variables and fields – and the number of possible interference clauses can rise quadratically with the number of such objects in a program. Second, the heap is a global data structure, and changes to it are visible to every part of a program, greatly complicating efforts at modular reasoning.

Separation logic, originally developed by John Reynolds and Peter O'Hearn [18, 10, 8], and elaborated by many others, is a powerful extension of Hoare logic for reasoning about pointer programs. To handle the difficulties described above, separation logic extends the assertion language of Hoare logic with *spatial* connectives. They are spatial in the sense that they permit formulas to range over just part of the heap: a spatial proposition $p * q$, written using the "separating conjunction", holds when p holds for one part of the heap, and q holds for the other part of it. Thus, aliasing between p and q is implicitly forbidden, removing the need to write tedious and extensive noninterference conditions. Separation further enables local reasoning about the heap: If one command in a program does its job in a heap fragment described by a proposition p, and another does its job in a fragment described by q, then we can safely combine the commands to work together in a heap described by $p * q$ — the separating conjunction's disjointness property means that the two subroutines cannot interfere with each other.

However, most of the work on separation logic has focused on low-level languages, which are strictly first-order and where features like the precise layout of data in memory and identity of pointers as integers are visible.

In this work, we define a specification logic for a a high-level language: a monadic lambda calculus in which pointers are a separate data type. Pointers may contain values of any type, including code (denoted by closures), such as functions and frozen monadic computations. Thus, our programming language resembles a subset of ML, except for the use of monadic form to make effects and their order of execution more explicit. Our program logic uses separation logic as the essential mechanism for reasoning about imperative computations, and simultaneously permits us to use the equational theory of the lambda calculus to reason about the functional aspect of programs.

In Section 2, we give the static and dynamic semantics of our language, and then in Section 3 define an assertion language with the separating conjunction and implication in it and give its semantics. Finally, in Section 4 we define a notion of Hoare triple for our language, and then build a specification logic on top of that, which uses triples (containing separation logic formulas) as its atomic formulas. This gives us a two-level logic in which we can relatively easily write program specifications.

2. THE MONADIC LANGUAGE

The programming language we work with is a monadic call-by-value lambda calculus. Following Davies and Pfenning [7], we stratify the language into two syntactic categories, *expressions* and *computations*, which are described in Figure 1. 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 2. Expressions include the usual introduction and elimination forms for sum, product, and function types. Additionally, we have reference types and monadic types.

The type ref τ classifies references that point to a value of type τ. Departing slightly from the usual presentation, pointer values take on the form l_τ – 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 τ, 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 5, by simply leaving out the store altogether from its reduction relation ($e \leadsto e'$).

We have a judgement $\Gamma \vdash c \div \tau$, given in Figure 3, 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 ($\text{new}_\tau e$) a reference; or sequencing two computations via monadic sequencing with the form letv $x = e$ 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`.

```
letv r = [new 5] in
letv dummy = [r := 17] in
letv n = [!r] in
  (n - 5)
```

Here, we have a computation which creates a new pointer, pointing to 5, and then updates it to point to 17, and then dereferences the pointer and returns the dereferenced value minus 5, for a final result of 12. 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 the sequential order using the monadic let-binding construct.

We also have a judgement σ ok in Figure 4 to characterize well-typed heaps. Unlike the usual presentation of adding references to the lambda calculus [13], 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, $\langle c;\ \sigma\rangle \leadsto \langle c';\ \sigma\rangle$, is given in Figure 6 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 $\langle e;\ \sigma\rangle \leadsto \textbf{abort}$.

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

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 next two sections. 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 $\text{new}_\tau v$ 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 the 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 $e \leadsto e'$.*

Proposition 2 (Expression Subject Reduction). *If $\cdot \vdash e : \tau$ and $e \leadsto e'$, 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 $e \leadsto^* v$ 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 $e \leadsto^* v$.*

We prove this using a straightforward logical relations argument [17], though it also follows immediately from the fact that the simply typed lambda calculus is strongly normalizing.

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 σ ok, then either c is a value v, or there exists a c' and σ' such that $\langle c;\ \sigma\rangle \leadsto \langle c';\ \sigma'\rangle$, or $\langle c;\ \sigma\rangle \leadsto$* **abort**.

Proposition 5 (Computation Subject Reduction). *If $\cdot \vdash c \div \tau$ and σ ok and $\langle c;\ \sigma\rangle \leadsto \langle c';\ \sigma'\rangle$, then $\cdot \vdash c' \div \tau$ and σ' 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 σ are members of the domain of σ.

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

$$\begin{array}{llcl}
\text{Types} & \tau & ::= & \mathbf{1} \mid \tau \times \tau' \mid \tau + \tau' \mid \tau \to \tau' \mid \text{ref } \tau \mid \bigcirc \tau \\
\\
\text{Expressions} & e & ::= & () \mid (e_1, e_2) \mid \pi_1 e \mid \pi_2 e \\
 & & \mid & \text{inl } e \mid \text{inr } e \mid \text{case}(e,\ x'.\ e',\ x''.\ e'') \\
 & & \mid & \lambda x : \tau.\ e \mid x \mid e_1\ e_2 \mid l_\tau \mid [c] \\
\text{Computations} & c & ::= & e \mid \text{letv } x = e \text{ in } c \mid !e \mid e := e' \\
 & & \mid & \text{new}_\tau e \\
\text{Values} & v & ::= & () \mid (v_1, v_2) \mid \text{inl } e \mid \text{inr } e \\
 & & \mid & \lambda x : \tau.\ e \mid l_\tau \mid [c] \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : \tau \\
\text{Heaps} & \sigma & ::= & \cdot \mid \sigma, l_\tau : v
\end{array}$$

Figure 1: The Basic Syntactic Categories

Although we have no explicit operator for term-level recursion, nontermination is possible in the computation language, because we have higher-order state – that is, pointers to functions. We can code an imperative fixpoint 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.) As a concrete example, consider the following computation:

```
letv r = [ref (fun x:unit. [x])] in
letv f = [fun y:unit. [letv recur = [!r] in
                        recur ()]] in
letv dummy = [r := f] in
letv looper = f() in
  (unreached code)
```

First, we create a function pointer r initially containing a pointer to some dummy value of type $\mathbf{1} \to \bigcirc\mathbf{1}$. Next, we create a function f, whose body is a frozen computation that dereferences r and invokes that function. Third, we update r to point to f. Finally, when we call f, r is dereferenced and binds its contents – f itself – to the variable *recur*, and finally calls f. This gives us an infinite loop, and shows that we have introduced nontermination into our monadic language.

Finally, we can devise a big step semantics for both the expression and computation language, and prove their agreement with the small-step semantics. We will write $e \Downarrow v$ for expression evaluation, and $\langle e;\ \sigma\rangle \Downarrow \langle v';\ \sigma'\rangle$ or $\langle e;\ \sigma\rangle \Downarrow$ **abort** for the big-step computation evaluation. These judgements are primarily a technical convenience, and for space reasons we do not give them; the rules are the obvious ones.

3. THE ASSERTION LANGUAGE AND ITS SEMANTICS

The assertion language in this paper, described in Figure 8, is a multi-sorted first order logic with equality, extended with the additional spatial connectives $*$ and $-\!\!*$. The sorts of the language are drawn form the types of the programming language, and we define a judgement $\Gamma \vdash p$: assert which ensures that any programming language terms that appear in an assertion are well-typed. The equality $e =_\tau e'$

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

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash () : \mathbf{1}}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \text{inl } e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \text{inr } e : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i e : \tau_i}$$

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

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

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \begin{array}{c} \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \\ \Gamma, x_2 : \tau_2 \vdash e_2 : \tau \end{array}}{\Gamma \vdash \text{case}(e,\ x_1.\ e_1,\ x_2.\ e_2) : \tau}$$

Figure 2: Static Semantics of Expressions

$$\boxed{\Gamma \vdash c \div \tau}$$

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

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

$$\frac{\Gamma \vdash e_1 : \text{ref } \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 \div \mathbf{1}}$$

Figure 3: Static Semantics of Computations

$$\boxed{\sigma \text{ ok}}$$

$$\frac{}{\cdot \text{ ok}} \qquad \frac{\sigma \text{ ok} \quad \cdot \vdash v : \tau \quad l_\tau \notin \text{dom}(\sigma)}{\sigma, l_\tau : v \text{ ok}}$$

Figure 4: Heap Well-Typing

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

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

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

$$\frac{e \rightsquigarrow e'}{\mathrm{inl}\ e \rightsquigarrow \mathrm{inl}\ e'} \qquad \frac{e \rightsquigarrow e'}{\mathrm{inr}\ e \rightsquigarrow \mathrm{inr}\ e'}$$

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

Figure 5: Small-step Dynamic Semantics of Expressions

and the points-to assertion $e \hookrightarrow_\tau e'$ are both indexed by types, but we will normally elide this annotation. Finally, we lift substitution of values for variables to assertions in the completely predictable way.

The semantics of a well-formed assertion p (that is, $\Gamma \vdash p : \mathrm{assert}$) is given with a forcing relation of the form

$$\gamma, \sigma \text{ ok} \models p$$

Here γ is an *environment*, a function mapping the free variables in Γ to closed values of the appropriate type. We will write $\gamma(x)$ to apply the function to a variable, and we will abuse notation and write $\gamma(e)$ to represent the application of an appropriately lifted function that substitutes values for the free variables of an expression e, consistently with γ. Likewise, we will write $\gamma(c)$, $\gamma(p)$ and so on to for liftings to computations, assertions, and so on.

We write σ ok to denote a *derivation* that a heap σ is well-typed. Using a derivation differs from the traditional presentation of separation logic, where the heap itself models a proposition. We make this choice in order to ensure that we only consider well-typed heaps. It also explains why our heap well-typing judgement in the previous section does not contain a closure condition that forbids dangling pointers: we want to be able to split and merge derivations in the same way that we can split and merge heaps.[2]

We use the following notation to represent this idea (taken from [8]):

- $\sigma \# \sigma'$ indicates that the domains of σ and σ' are disjoint.
- $\sigma \cdot \sigma'$ indicates the union of disjoint heaps, that is, when $\sigma \# \sigma'$ have disjoint domains, we take the concatenation of the two heaps.
- $\sigma \sqsupseteq \sigma'$ indicates that σ is an extension of σ'; that is, σ contains all of the pointer/value pairs of σ' and possibly some additional ones.

[2]The appropriate intuition is that the two rules for typing a heap give rise to a list datatype with a nil and a cons, and that lists form a monoid, which lets us use the monoidal resource semantics of BI [15].

$$\boxed{\begin{array}{c}\langle c;\ \sigma\rangle \rightsquigarrow \langle c';\ \sigma'\rangle \\[1ex] \langle c;\ \sigma\rangle \rightsquigarrow \textbf{abort}\end{array}}$$

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

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

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

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

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

$$\frac{e \rightsquigarrow e'}{\langle \mathrm{new}_\tau e;\ \sigma\rangle \rightsquigarrow \langle \mathrm{new}_\tau e';\ \sigma\rangle}$$

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

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

$$\frac{l_\tau : v \in \sigma}{\langle !l_\tau;\ \sigma\rangle \rightsquigarrow \langle v;\ \sigma\rangle}$$

$$\frac{l_\tau \notin \mathrm{dom}(\sigma)}{\langle !l_\tau;\ \sigma\rangle \rightsquigarrow \textbf{abort}}$$

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

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

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

$$\frac{l_\tau \notin \mathrm{dom}(\sigma)}{\langle l_\tau := v;\ \sigma\rangle \rightsquigarrow \textbf{abort}}$$

Figure 6: Dynamic Semantics of Computations

$$\begin{array}{rcl} p & ::= & e =_\tau e' \mid e \hookrightarrow_\tau e' \mid p \wedge p' \mid p \supset p' \mid p * p' \mid p -\!\!* p' \\ & \mid & \top \mid \bot \mid p \vee p' \mid \forall x : \tau.\ p \mid \exists x : \tau.\ p \end{array}$$

$$\frac{}{\Gamma \vdash : \top : \text{assert}} \qquad \frac{}{\Gamma \vdash : \bot : \text{assert}}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e = e' : \text{assert}} \qquad \frac{\Gamma \vdash e : \text{ref}\ \tau \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e \hookrightarrow e' : \text{assert}}$$

$$\frac{\Gamma \vdash p : \text{assert} \quad \Gamma \vdash q : \text{assert} \quad \text{Op} \in \{\wedge, \supset, \vee, *, -\!\!*\}}{\Gamma \vdash p\ \text{Op}\ q : \text{assert}}$$

$$\frac{\Gamma, x : \tau \vdash p : \text{assert} \quad \text{Q} \in \forall, \exists}{\Gamma \vdash \text{Q}x : \tau.\ p : \text{assert}}$$

Figure 7: Syntax of Separation Logic Assertions

Since every heap well-typing derivation σ ok identifies a unique heap (up to permutation), we can overload our notation and write $(\sigma \text{ ok}) \# (\sigma' \text{ ok})$ or $(\sigma \text{ ok}) \cdot (\sigma' \text{ ok})$ without confusion. We will also sometimes say that a proposition "holds in a heap", with the understanding that we really mean the derivation that proves the heap well-typed.

We give the cases of the forcing relation in figure 7. This semantics satisfies the *Kripke monotonicity condition*, which is that if $\gamma, \sigma \text{ ok} \models p$ and $\sigma' \text{ ok} \sqsupseteq \sigma \text{ ok}$, then $\gamma, \sigma' \text{ ok} \models p$.

This means that we have chosen to use the *intuitionistic* semantics of separation logic [8], in which if a proposition holds in a heap, it continues to hold in all extensions of that heap. There is no technical obstacle to using the classical semantics; however, we choose this version of separation logic because our language does not contain a command for de-allocating pointers.[3]

The equality connective $e =_\tau e'$ holds when e and e' are equal. The notion of equality that we choose is beta-eta equality for the functional subset, and the monad laws for computations. For space reasons, we only give a subset of these rules in Figure 9.

Our choice raises a couple of interesting issues. First, in the previous section, we gave an operational semantics, rather than a denotational semantics, for our language. This means that we have a relatively free choice for what we define equality to mean, as long as we prove that evaluation of expressions preserve equality – that is, if $\Gamma \vdash e_1 = e_2 : \tau$, and $e_1 \rightsquigarrow e_1'$, then $\Gamma \vdash e_1' = e_2 : \tau$. Verifying this is a straightforward induction on the reduction relation.

Second, we have to consider what it means for two computations to be equal; that is, if we have the assertion $[c] = [c']$, then we have to judge whether $\Gamma \vdash [c] = [c'] : \bigcirc\tau$, and hence whether $\Gamma \vdash c = c' \div \tau$. To our knowledge, this question has not appeared in other presentations of separation logic, since for the most part they do not have heaps that contain code. Obviously, we cannot give a full characterization of equality of computations – it is because of the weakness of imperative programs' equational theory that we are trying to devise a separation logic at all. Instead, we take a conservative approach, and take only the monad laws as our equational theory of computations. These are the rules given for the judgement $\Gamma \vdash c = c \div \tau$ in Figure 9. (The monad laws are the natural transformations of a Kleisli triple in category theory [9], and are the commuting conversions in a proof-theoretic presentation of lax logic [1].) Verifying that the computation reduction rules for states agree with these monad laws is again straightforward. It may be possible to give a richer notion of equality of computations employing more of the particular algebraic structure of heaps, e.g. [14], but we have not yet investigated this possibility.

[3]In a real implementation, memory will be collected using garbage collection. Giving a garbage-collection aware semantics for separation logic is quite subtle [6], and so we ignore garbage collection in our operational semantics and defer further consideration of this point for future work.

The connective $e \hookrightarrow_\tau e'$ looks one deep into the heap. It is true exactly when the heap contains a pairing $l_\tau : v$, such that e equals l_τ and $e' = v$. As noted above, our points-to connective is intuitionistic; it only asks whether there is at least a memory location e in the heap pointing to e' – there can be more cells.

The separating conjunction $p_1 * p_2$ holds whenever a heap typing σ ok can be split into two disjoint subheaps σ_1 and σ_2, in which p_1 and p_2 hold separately. For example, with this definition, the proposition $l^1_{\texttt{nat}} \hookrightarrow 0 * l^2_{\texttt{nat}} \hookrightarrow 5$ holds of any extension of the heap $\sigma = l^1_{\texttt{nat}} : 0, l^2_{\texttt{nat}} : 5$.

With this semantics, we can prove soundness for the logic. We do this by defining *the semantic consequence relation* $p \rhd_\Gamma q$ to hold between well-formed formulas $\Gamma \vdash p : \text{assert}$ and $\Gamma \vdash q : \text{assert}$ if and only if for all γ that are well-typed environments under Γ, and for all heap typings σ ok, if $\gamma, \sigma \text{ ok} \models p$ holds then $\gamma, \sigma \text{ ok} \models q$ holds.

Proposition 7 (Assertion Soundness). *The usual rules of intuitionistic logic are sound for the semantic consequence relation, along with the following rules:*

- $*$ *is associative and commutative, with unit* $\top$.
- *If* $p \rhd_\Gamma q$, *and* Γ' *extends* Γ, *then* $p \rhd_{\Gamma'} q$.
- $*$ *supports weakening:* $p * q \rhd_\Gamma p$
- *The following inference rules hold:*

$$\frac{p' \rhd_\Gamma p \quad q' \rhd_\Gamma q}{p' * q' \rhd_\Gamma p * q}$$

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

4. SPECIFICATION LOGIC

With a semantics for the assertion language in hand, we now consider how to construct a specification language from it. Our general approach closely mirrors specification logic [16] 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 change the side effects of an expression).

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

$\gamma, \sigma \text{ ok} \models e = e'$	iff	$\cdot \vdash \gamma(e) = \gamma(e') : \tau$
$\gamma, \sigma \text{ ok} \models e \hookrightarrow e'$	iff	$\cdot \vdash \gamma(e) = l_\tau : \text{ref } \tau$ and $\cdot \vdash \gamma(e') = v : \tau$ and $l_\tau : v \in \sigma$
$\gamma, \sigma \text{ ok} \models p_1 \wedge p_2$	iff	$\gamma, \sigma \text{ ok} \models p_1$ and $\gamma, \sigma \text{ ok} \models p_2$
$\gamma, \sigma \text{ ok} \models p_1 \vee p_2$	iff	$\gamma, \sigma \text{ ok} \models p_1$ or $\gamma, \sigma \text{ ok} \models p_2$

$\gamma, \sigma \text{ ok} \models p_1 \supset p_2$	iff	for all $\sigma' \text{ ok} \sqsupseteq \sigma \text{ ok}$.
		if $\gamma, \sigma' \text{ ok} \models p_1$
		then $\gamma, \sigma' \text{ ok} \models p_2$

$\gamma, \sigma \text{ ok} \models p_1 * p_2$	iff	there exist $\sigma_1 \text{ ok}, \sigma_2 \text{ ok}$. $\sigma = \sigma_1 \cdot \sigma_2$ and $\sigma_1 \# \sigma_2$ and $\gamma, \sigma_1 \text{ ok} \models p_1$ and $\gamma, \sigma_2 \text{ ok} \models p_2$

$\gamma, \sigma \text{ ok} \models p_1 -\!\!* p_2$	iff	for all σ_1 ok.
		if $\sigma \# \sigma_1$ and $\gamma, \sigma_1 \text{ ok} \models p_1$
		then $(\gamma, \sigma_1 \cdot \sigma) \text{ ok} \models p_2$

$\gamma, \sigma \text{ ok} \models \forall x : \tau.\ p$	iff	for all $\cdot \vdash v : \tau$. $\gamma, \sigma \text{ ok} \models p[v/x]$
$\gamma, \sigma \text{ ok} \models \exists x : \tau.\ p$	iff	there exists $\cdot \vdash v : \tau$. $\gamma, \sigma \text{ ok} \models p[v/x]$
$\gamma, \sigma \text{ ok} \models \top$	iff	always
$\gamma, \sigma \text{ ok} \models \bot$	iff	never

Figure 8: Semantics of Assertions

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

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e = e : \tau}\ \textit{Refl}$$

$$\frac{\Gamma \vdash e = e' : \tau \quad \Gamma \vdash e' = e'' : \tau}{\Gamma \vdash e = e'' : \tau}\ \textit{Trans}$$

$$\frac{\Gamma \vdash e' = e : \tau}{\Gamma \vdash e = e' : \tau}\ \textit{Sym}$$

$$\frac{\Gamma, x : \tau' \vdash e_1 = e_1' : \tau \quad \Gamma \vdash e_2 = e_2' : \tau'}{\Gamma \vdash (\lambda x : \tau'.\ e_1) e_2 = [e_2'/x]e_1' : \tau}\ \textit{FunBeta}$$

$$\frac{\Gamma, x : \tau' \vdash ex = e'x : \tau}{\Gamma \vdash e = e' : \tau' \to \tau}\ \textit{FunExt}$$

$$\boxed{\Gamma \vdash c = c' \div \tau}$$

$$\frac{\Gamma \vdash c \div \tau}{\Gamma \vdash c = c \div \tau}\ \textit{Refl}$$

$$\frac{\Gamma \vdash c = c' \div \tau \quad \Gamma \vdash c' = c'' \div \tau}{\Gamma \vdash c = c'' \div \tau}\ \textit{Trans}$$

$$\frac{\Gamma \vdash c' = c \div \tau}{\Gamma \vdash c = c' \div \tau}\ \textit{Sym}$$

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

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

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

$$\frac{\Gamma \vdash \text{letv } x = [v] \text{ in } c \div \tau}{\Gamma \vdash \text{letv } x = [v] \text{ in } c = [v/x]c \div \tau}$$

Figure 9: Selected Rules of Typed Equality for Expressions and Computations

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:

```
{true}
letv n = fact(k) in
  (k+1) * n
{x:nat. x = (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 $\Gamma \vdash \{p\}c\{x : \tau.\ q\}$: spec, then we give meaning to the triple as a function of type-correct environments γ:

$$[\![\{p\}c\{x:\tau.\ q\}]\!]\gamma = \begin{array}{l} \forall\sigma \text{ ok}.\ \forall\sigma' \text{ ok}.\ \forall v. \\ \quad \text{if } \gamma, \sigma \text{ ok} \models p \text{ then} \\ \quad\quad \langle\gamma(c);\ \sigma\rangle \not\Downarrow \textbf{abort} \\ \quad\quad \text{and} \\ \quad\quad (\text{if } \langle\gamma(c);\ \sigma\rangle \Downarrow \langle v;\ \sigma'\rangle \\ \quad\quad\ \text{then } \gamma, \sigma' \text{ ok} \models [v/x]q) \end{array}$$

The type of this semantic function is that it takes environments to booleans; it is *not* a relation between heaps. Heap typings appear in the definition, but we universally quantify over all of them. Also, note that is a partial correctness criterion. Likewise, we give a semantics for triples over suspended monadic computations as follows:

$$[\![\langle p\rangle e\langle x:\tau.\ q\rangle]\!]\gamma = \begin{array}{l} \forall\sigma \text{ ok}.\ \forall\sigma' \text{ ok}.\ \forall v.\forall c. \\ \quad \text{if } \gamma, \sigma \text{ ok} \models p \\ \quad\quad \text{and } \boldsymbol{.} \vdash \gamma(e) = [c] : \bigcirc\tau \\ \quad \text{then} \\ \quad\quad \langle c;\ \sigma\rangle \not\Downarrow \textbf{abort} \\ \quad\quad \text{and} \\ \quad\quad (\text{if } \langle c;\ \sigma\rangle \Downarrow \langle v;\ \sigma'\rangle \\ \quad\quad\ \text{then } \gamma, \sigma' \text{ ok} \models [v/x]q) \end{array}$$

This is essentially a duplicate of the previous semantic equation, but is needed for two reasons. First, a let-binding letv $x = [c]$ 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.

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:

$$\begin{array}{lcl} [\![S_1 \text{ and } S_2]\!]\gamma & = & [\![S_1]\!]\gamma \text{ and } [\![S_2]\!]\gamma \\ [\![S_1 \text{ or } S_2]\!]\gamma & = & [\![S_1]\!]\gamma \text{ or } [\![S_2]\!]\gamma \\ [\![S_1 \text{ implies } S_2]\!]\gamma & = & \text{if } [\![S_1]\!]\gamma \text{ then } [\![S_2]\!]\gamma \\ [\![\forall x:\tau.\ S]\!]\gamma & = & \forall(\boldsymbol{.} \vdash v:\tau).\ [\![[v/x]S]\!]\gamma \\ [\![\forall x:\tau.\ S]\!]\gamma & = & \exists(\boldsymbol{.} \vdash v:\tau).\ [\![[v/x]S]\!]\gamma \end{array}$$

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:

```
  (forall m:nat. <true>fact(m)<\x:nat. x = m!>)
implies
  ({true}
   letv n = fact(k) in (k+1) * n
   {x:nat. x = (k+1)!})
```

This specification is substantially more satisfying than before, because we have a clear idea of what the `fact` function is supposed to do. 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 environments. Following Reynolds [16], we will call such specifications *universal*. For a well-formed specification $\Gamma \vdash S$: spec, we say:

$$S \text{ is universal} \quad \text{iff} \quad \forall\gamma.[\![S]\!]\gamma$$

In Figures 12 and 13, we give a collection of deduction rules for inferring universal 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 and ... and S_n) implies S with free variables in Γ.

Proposition 8 (Soundness of Specification Logic). *Every derivation $\Gamma; \Delta \vdash S$ using the rules in Figures 12 and 13 derives a universal 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 [10], they enable local reasoning because the preconditions and postconditions refer to no other pointers than the ones that are accessed.

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

Figure 10: Syntax of Specifications

To prove the Frame rule, we first show that the operational semantics validates the *safety monotonicity* and *frame* properties.[11] 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, with a small extra bit of complexity arising from the use of environments in the semantics.

It is the combination of the frame rule and the substitution property that enables genuinely modular reasoning about programs.

We can prove a client programs that use an imperative module if we specify its interfaces in 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 easily compose the pieces using the existential elimination rule.

The existential elimination rule is 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.

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. The first benefit is that it 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. The second benefit is that it permits the use of an extremely general substitution principle, which allows a simple method of combining specifications to work correctly.

5. RELATED AND FUTURE WORK

In spirit, this work is a direct descendent of Reynolds' work on specification logic for Idealized Algol [16]. 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 [19], that spawned the hypoth-

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

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

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

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

Figure 11: Well-Formedness of Specifications

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

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

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

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

$$\frac{\Gamma; \Delta \vdash \langle p\rangle e\langle x:\tau.\ q\rangle \qquad \Gamma, x:\tau; \Delta \vdash \{q\}c\{y:\tau'.\ r\} \quad x \notin \mathrm{FV}(r)}{\Gamma; \Delta \vdash \{p\}\text{letv } x = e \text{ in } c\{y:\tau'.\ r\}}\ \textit{Seq}$$

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

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

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

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

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

$$\frac{\Gamma; \Delta \vdash \langle p\rangle e\langle x:\tau.\ q\rangle \quad \Gamma \vdash r : \text{assert}}{\Gamma; \Delta \vdash \langle p * r\rangle e\langle x:\tau.\ q * r\rangle}\ \textit{Frame2}$$

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

Figure 12: Universal Specification Inference Rules

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

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

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

$$\frac{\Gamma, x : \tau; \Delta \vdash S \quad \Gamma \vdash e : \tau}{\Gamma; [e/x]\Delta \vdash [e/x]S}\ \textit{Substitute}$$

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

$$\frac{\Gamma; \Delta \vdash S_1 \text{ and } S_2}{\Gamma; \Delta \vdash S_1}\ \textit{AndElim}1$$

$$\frac{\Gamma; \Delta \vdash S_1 \text{ and } S_2}{\Gamma; \Delta \vdash S_2}\ \textit{AndElim}2$$

$$\frac{\Gamma; \Delta \vdash S_1 \quad \Gamma \vdash S_2 : \text{spec}}{\Gamma; \Delta \vdash S_1 \text{ or } S_2}\ \textit{OrIntro}1$$

$$\frac{\Gamma; \Delta \vdash S_2 \quad \Gamma \vdash S_1 : \text{spec}}{\Gamma; \Delta \vdash S_1 \text{ or } S_2}\ \textit{OrIntro}2$$

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

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

$$\frac{\Gamma; \Delta \vdash S' \text{ implies } S \quad \Gamma; \Delta \vdash S'}{\Gamma; \Delta \vdash S}\ \textit{ImpElim}$$

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

$$\frac{\Gamma; \Delta \vdash \forall x : \tau.\ S}{\Gamma, x : \tau; \Delta \vdash S}\ \textit{UnivElim}$$

$$\frac{\Gamma; \Delta \vdash [e/x]S \quad \Gamma \vdash e : \tau}{\Gamma; \Delta \vdash \exists x : \tau.\ S}\ \textit{ExistIntro}$$

$$\frac{\Gamma; \Delta \vdash \exists x : \tau.\ S' \quad \Gamma, x : \tau; \Delta, S' \vdash S}{\Gamma; \Delta \vdash S}\ \textit{ExistElim}$$

Figure 13: Universal Specifications, Continued

esis 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 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 [5] have done work on a version of separation logic for a restricted variant of Algol: like us, they remove variable assignment and put all aliasing into the heap. The heap, as in Algol but in the current work, is merely a first-order map of integers. 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. However, they support a very powerful kind of hypothetical frame rule in their language.

Parkinson [12] 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 the lack of pure higher-order functions sharply limits the program expressions that can appear in assertions. He also 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.

Our version of separation logic supports reasoning with procedures, but it does not support information hiding in procedures, either in the sense of having a hypothetical frame rule in the sense of O'Hearn, Yang and Reynolds [11], or as with Parkinson's abstract predicates. Instead of directly adding features to address this deficiency, we are instead considering moving to a higher-order variant of separation logic, such as Biering, Birkedal and Torp-Smith propose in [3]. Then, modules and their clients could communicate through existentially quantified assertions, as proposed in [4]. The latter paper also develops a specification logic using universal specifications (which they call *valid*) for a simple first-order programming language.

Additionally, it would be interesting to add polymorphism and existential types to the underlying programming language – the combination of existential types concealing the exact term representations of an ADT and existential separation predicates concealing the exact heap layout could be very illuminating about the nature of encapsulation. We also plan on adding recursive types and term-level recursion operators to the programming language, and to add recursively defined assertions to the assertion language.

Finally, Berger, Honda and Yoshida recently described a new logic for analysing aliasing in imperative functional programs [2]. 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.

6. ACKNOWLEDGMENTS

I would like to thank John Reynolds and Jonathan Aldrich for their advice, encouragement, and patience. This work was supported in part by NASA cooperative agreements NCC-2-1298 and NNA05CS30A, and National Science Foundation Grants CCR-0204242 and CCR-0204047.

7. REFERENCES

[1] P. Benton, G. Bierman, and V. D. Paiva. Computational types from a logical perspective. *Journal of Functional Programming*, 8(2):177–193, March 1998.

[2] M. Berger, K. Honda, and N. Yoshida. A logical analysis of aliasing in imperative higher-order functions. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 280–293. ACM Press, September 2005.

[3] B. Biering, L. Birkedal, and N. Torp-Smith. BI Hyperdoctrines and Higher-Order Separation Logic. In *Programming Languages and Systems: 14th European Symposium on Programming, ESOP 2005*, volume 3444/2005, pages 233–247, Edinburgh, UK, April 2005. Springer-Verlag GmbH.

[4] 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, July 2005.

[5] L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules. In P. Panangaden, editor, *Proceedings of the Twentieth Annual IEEE Symp. on Logic in Computer Science, LICS 2005*, pages 260–269. IEEE Computer Society Press, June 2005.

[6] C. Calcagno, P. W. O'Hearn, and R. Bornat. Program logic and equivalence in the presence of garbage collection. *Theoretical Computer Science*, 298(3):557–581, April 2003.

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

[8] S. S. Ishtiaq and P. W. O'Hearn. BI as an Assertion Language for Mutable Data Structures. In *POPL*, pages 14–26, 2001.

[9] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[10] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19, Berlin, 2001. Springer-Verlag.

[11] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Conference Record of POPL 2004: The 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 268–280, New York, 2004. ACM Press.

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

[13] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge MA, 2002.

[14] G. D. Plotkin and A. J. Power. Computational effects and operations: An overview. *Electr. Notes Theor. Comput. Sci*, 73:149–163, 2004.

[15] D. J. Pym, P. W. O'Hearn, and H. Yang. Possible worlds and resources: the semantics of bi. *Theoretical Computer Science*, 315(1):257–305, 2004.

[16] J. C. Reynolds. *The Craft of Programming*. Prentice-Hall International, London, 1981.

[17] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B. V. (North-Holland).

[18] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Los Alamitos, California, 2002. IEEE Computer Society.

[19] S. Weeks and M. Felleisen. On the orthogonality of assignments and procedures in Algol. In *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 57–70, 1993.