Design Patterns in Separation Logic

Neelakantan R. Krishnaswami
Jonathan Aldrich
Carnegie Mellon University
{neelk,jonathan.aldrich}@cs.cmu.edu

Lars Birkedal Kasper Svendsen
Alexandre Buisse
IT University of Copenhagen
{birkedal,kasv,abui}@itu.dk

Abstract

Object-oriented programs are notable for making use of both higher-order abstractions and mutable, aliased state. Either feature alone is challenging for formal verification, and the combination yields very flexible program designs and correspondingly difficult verification problems. In this paper, we show how to formally specify and verify programs that use several common design patterns in concert.

Categories and Subject Descriptors F.3 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.2.8 [*Software Engineering*]: Metrics—complexity measures, performance measures

General Terms Languages, Verification

Keywords Separation Logic, Design Patterns, Formal Verification

1. Introduction

The widespread use of object-oriented languages creates an opportunity for designers of formal verification systems, above and beyond a potential "target market". Object-oriented languages have been used for almost forty years, and in that time practitioners have developed a large body of informal techniques for structuring object-oriented programs called *design patterns*(4). Design patterns were developed to both take best advantage of the flexibility object-oriented languages permit, and to control the potential complexities arising from the unstructured use of these features.

This pair of characteristics make design patterns an excellent set of benchmarks for a program logic. First, design patterns use higher order programs to manipulate aliased, mutable state. This is a difficult combination for program verification systems to handle, and attempting to verify these programs will readily reveal weaknesses or lacunae in the program logic. Second, the fact that patterns are intended to structure and modularize programs means that we can use them to evaluate whether the proofs in a program logic respect the conceptual structure of the program – we can check to see if we need to propagate conceptually irrelevant information out of program modules in order to meet our proof obligations. Third, we have the confidence that these programs, though small, actually reflect realistic patterns of usage.

TLDI'09, January 24, 2009, Savannah, Georgia, USA.

In this paper, we describe Idealized ML, a core higher-order imperative language, and a specification language based on separation logic for it. Then, we give good specifications for and verify the following programs:

- We prove a collection and iterator implementation, which builds the Java aliasing rules for iterators into its specification, and which allows the construction of new iterators from old ones via the composite and decorator patterns.
- We prove a general version of the flyweight pattern (also known as hash-consing in the functional programming community), which is a strategy for aggressively creating aliased objects to save memory and permit fast equality tests. This also illustrates the use of the factory pattern.
- We prove a general version of the subject-observer pattern in a way that supports a strong form of information hiding between the subject and the observers.

Finally, we give machine-verified proofs of the correctness of the iterator and flyweight patterns in the Ynot extension of Coq, and compare them with the paper proofs. We also see that proper treatment of the subject-observer pattern seems to call for the use of an impredicative type theory.

2. Formal System

The formal system we present has three layers. First, we have a core programming language we call Idealized ML. This is a simply-typed functional language which isolates all side effects inside a monadic type (20). The side effects include nontermination and the allocation, access, and modification of general references (including pointers to closures). Then, we give an assertion language based on higher-order separation logic (3) to describe the state of a heap. Separation logic allows us to give a clean treatment of issues related to specifying and controlling aliasing, and higher-order predicates allow us to abstract over the heap. This enables us to enforce encapsulation by hiding the exact layout of a module's heap data structures. Finally, we have a specification logic to describe the effects of programs, which is a first-order logic whose atomic propositions are Hoare triples $\{P\}c\{a : A.\ Q\}$, which assert that if the heap is in a state described by the assertion P, then executing the command c will result in a postcondition state Q (with the return value of the command bound to a). specifications.

Programming Language. The core programming language we have formalized is an extension of the simply-typed lambda calculus with a monadic type constructor to represent side-effecting computations. The types of our language are the unit type 1, the function space $A \to B$, the natural number type $\mathbb{N}$, the reference type $\mathsf{ref}\ A$, as well as the option type $\mathsf{option}\ A$ and the mutable linked list type $\mathsf{list}\ A$. In addition, we have the monadic type $\bigcirc A$, which is the type of suspended side-effecting computations produc-

$$
\begin{array}{lrcl}
\text{Types} & A & ::= & 1 \mid A \to B \mid \mathsf{list}\ A \mid \mathsf{option}\ A \mid \mathbb{N} \mid \mathsf{ref}\ A \mid \bigcirc A \\
\text{Pure Terms} & e & ::= & () \mid x \mid e\ e' \mid \lambda x : A.\ e \mid \mathsf{Nil} \mid \mathsf{Cons}(e, e') \mid \mathsf{None} \mid \mathsf{Some}(e) \\
 & & \mid & \mathsf{case}(e,\ \mathsf{Nil} \to e',\ \mathsf{Cons}(x, xs) \to e'') \mid \mathsf{case}(e,\ \mathsf{None} \to e',\ \mathsf{Some}\ x \to e'') \\
 & & \mid & \mathsf{z} \mid \mathsf{s}(e) \mid \mathsf{prec}(e, e_z, x.\ e_s) \mid \mathsf{fix}\ e \mid [c] \\
\text{Computations} & c & ::= & e \mid \mathsf{letv}\ x = e\ \mathsf{in}\ c \mid \mathsf{run}\ e \mid \mathsf{new}_A(e) \mid\ !e \mid e := e'
\end{array}
$$

Figure 1. Types and Syntax of the Programming Language

ing values of type A. Side effects include both heap effects (such as reading, writing, or allocating a reference) and nontermination.

We maintain such a strong distinction between pure and impure code for two reasons. First, it allows us to validate very powerful equational reasoning principles for our language: we can validate the full β and η rules of the lambda calculus for each of the pure types. This simplifies reasoning even about imperative programs, because we can relatively freely restructure the program source to follow the logical structure of a proof. Second, when program expressions appear in assertions – that is, the pre- and post-conditions of Hoare triples – they must be pure. However, allowing a rich set of program expressions like function calls or arithmetic in assertions makes it much easier to write specifications. So we restrict which types can contain side-effects, and thereby satisfy both requirements. The pure terms of the language are typed with the typing judgment $\Gamma \vdash e : A$, given in Figure 2, and which can be read as "In variable context Γ, the pure expression e has type A." Computations are typed with the judgment $\Gamma \vdash c \div A$ (also in Figure 2), which can be read as "In the context Γ, the computation c is well-typed at type A ."

We have $()$ as the inhabitant of 1, natural numbers z and $\mathsf{s}(e)$, functions $\lambda x : A.\ e$, optional expressions None and $\mathsf{Some}\ e$, and list expressions Nil and $\mathsf{Cons}(e, e')$. Nil is the constructor for an empty list, and $\mathsf{Cons}(e, e')$ is the constructor for a cons-cell. In Figure 2, ELISTCONS states the tail of a list is of *reference* type $\mathsf{ref\ list}\ A$, which means that our list type is a type of *mutable* lists. We also have the corresponding eliminations for each type, including case statements for option types and list types. For the natural numbers, we add a primitive recursion construct $\mathsf{prec}(e, e_z, x.\ e_s)$. If $e = \mathsf{z}$, this computes e_z , and if $e = \mathsf{s}(e')$, it computes $e_s[(\mathsf{prec}(e', e_z, x.\ e_s))/x]$. This bounded iteration allows us to implement (for example) arithmetic operations as pure expressions.

Suspended computations $[c]$ inhabit the monadic type $\bigcirc A$. These computations are not immediately evaluated, which allows us to embed them into the pure part of the programming language. Furthermore, we can take fixed points of monad-valued functions (EFIX), which gives us a general recursion facility. (We restrict fix to monad-valued domains because of the possibility of nontermination. Also, we will write recursive functions as syntactic sugar for fix.) As CPURE shows, we can treat any pure expression of type A as a computation that coincidentally has no side-effects. In CRUN, a suspended computation of type $\bigcirc A$ can be forced to execute with the $\mathsf{run}\ e$ command.

In CLET, we have a sequential composition $\mathsf{letv}\ x = e\ \mathsf{in}\ c$. Intuitively, the behavior of this command is as follows. We evaluate e until we get some $[c']$, and then evaluate c', modifying the heap and binding its return value to x. Then, in this augmented environment, we run c. The fact that monadic commands have return values explains why our sequential composition is also a binding construct. Finally, we have computations $\mathsf{new}_A(e)$, $!e$, and $e := e'$, which let us allocate, read and write references, respectively.

This language has been given a typed denotational semantics, which for space reasons we do not give here. The details of the semantics (including the assertion and specification levels) can be found in the companion tech report (8).

Assertion Language. The sorts and syntax of the assertion language are given in Figure 3. The assertion language is a version of separation logic, extended to higher order.

In ordinary Hoare logic, a predicate describes a set of program states (in our case, heaps), and a conjunction like $p \wedge q$ means that a heap in $p \wedge q$ is in the set described by p and the described by q. While this is a natural approach, aliasing can become quite difficult to treat – if x and y are pointer variables, we need to explicitly state whether they alias or not. This means that as the number of variables in a program grows, the number of aliasing conditions grows quadratically. With separation logic, we add the *spatial* connectives to address this difficulty. A separating conjunction $p * q$ means that the state can be broken into two *disjoint* parts, one of which is in the state described by p, and the other of which is in the state described by q. The disjointness property makes the noninterference of p and q implicit. This avoids the unwanted quadratic growth in the size of our assertions. In addition to the separating conjunction, we have its unit emp, which is true of the empty heap, and the points-to relation $e \mapsto e'$, which holds of the one-element heap in which the reference e has contents e'. A heap is described by the "magic wand" $p \mathrel{-\!\!*} q$, when we can merge it with any disjoint heap described by p, and the combination is described by q.

The universal and existential quantifiers $\forall x : \omega.\ p$ and $\exists x : \omega.\ p$ are higher-order quantifiers ranging over all sorts ω. The sorts include the language types A, the sort of propositions prop, function spaces over sorts $\omega \Rightarrow \omega'$, and mathematical sequences $\mathsf{seq}\ \omega$. Constructors for terms of all these sorts in the syntax given in Figure 3. For the function space, we include lambda-abstraction and application. For sequence sorts, we have sequence-formers ϵ for the empty sequence and $p \cdot ps$ for adding one element to a sequence, as well as a primitive iteration construct over sequences, $\mathsf{prec}_{\mathsf{seq}}(p, p_\epsilon, (x, acc).\ p.)$. With iteration, we can write constructor-level map and filter functions (as in functional programming) and use them in our specifications. Our examples will also make use of finite sets and functions, without formalizing them in our syntax.

Finally, we include the atomic formulas $S\ \mathsf{valid}$, which are *assertions* that a *specification* S holds. This facility is useful when we write assertions about pointers to code – for example, the assertion $r \mapsto cmd \wedge (\{p\}\mathsf{run}\ cmd\{a : A.\ q\})\ \mathsf{valid}$ says that the reference r points to a monadic term cmd, whose behavior is described by the Hoare triple $\{p\}\mathsf{run}\ cmd\{a : A.\ q\}$.

Specification Language. Given programs and assertions about the heap, we need specifications to relate the two. We begin with the Hoare triple $\{p\}c\{a : A.\ q\}$. This specification represents the claim that if we run the computation c in any heap the predicate p describes, then if c terminates, it will end in a heap described by the predicate q. Since monadic computations can return a value in addition to having side-effects, we add the binder $a : A$ to the third clause of the triple to let us name and use the return value in the postcondition.

We then treat Hoare triples as one of the atomic proposition forms of a first-order intuitionistic logic (see Figure 3). The other form of atomic proposition are the specifications $\{p\}$, which are $specifications$ saying that an *assertion* p is true. These formulas are useful for expressing aliasing relations between defined predi-

$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash e' : \mathsf{ref\ list}\ A}{\Gamma \vdash \mathsf{Cons}(e, e') : \mathsf{list}\ A}\ \textsc{EListCons} \qquad \frac{\Gamma \vdash c \div A}{\Gamma \vdash [c] : \bigcirc A}\ \textsc{EMonad} \qquad \frac{\Gamma \vdash e : (A \to \bigcirc B) \to (A \to \bigcirc B)}{\Gamma \vdash \mathsf{fix}\ e : A \to \bigcirc B}\ \textsc{EFix}$$

$$\frac{\Gamma \vdash e : \bigcirc A \qquad \Gamma, x : A \vdash c \div B}{\Gamma \vdash \mathsf{letv}\ x = e\ \mathsf{in}\ c \div B}\ \textsc{CLet} \qquad \frac{\Gamma \vdash e : A}{\Gamma \vdash e \div A}\ \textsc{CPure} \qquad \frac{\Gamma \vdash e : \bigcirc A}{\Gamma \vdash \mathsf{run}\ e \div A}\ \textsc{CRun} \qquad \frac{\Gamma \vdash e : A}{\Gamma \vdash \mathsf{new}_A(e) \div \mathsf{ref}\ A}\ \textsc{CRefNew}$$

$$\frac{\Gamma \vdash e : \mathsf{ref}\ A}{\Gamma \vdash !e \div A}\ \textsc{CRefRead} \qquad \frac{\Gamma \vdash e : \mathsf{ref}\ A \qquad \Gamma \vdash e' : A}{\Gamma \vdash e := e' : 1}\ \textsc{CRefWrite}$$

Figure 2. Selected Typing Rules

Assertion Sorts	ω	::=	$A \mid \omega \Rightarrow \omega \mid \mathsf{seq}\ \omega \mid \mathsf{prop}$
Assertion Constructors	p	::=	$e \mid x \mid \lambda x : \omega.\ p \mid p\ q$
		|	$\mathsf{prec}(p, p_\epsilon, (x, acc).\ p.) \mid \epsilon \mid p \cdot ps$
		|	$\top \mid p \wedge q \mid p \supset q \mid \bot \mid p \vee q$
		|	$\mathsf{emp} \mid p * q \mid p -\!\!* q \mid e \mapsto e'$
		|	$\forall x : \omega.\ p \mid \exists x : \omega.\ p \mid S\ \mathsf{valid}$
Specifications	S	::=	$\{p\}c\{a : A.\ q\} \mid \{p\}$
		|	$S\ \mathsf{and}\ S' \mid S\ \mathsf{implies}\ S' \mid S\ \mathsf{or}\ S'$
		|	$\forall x : \omega.\ S \mid \exists x : \omega.\ S$

Figure 3. Syntax of Assertions and Specifications

cates, without necessarily revealing the implementations. In addition, we can form specifications with conjunction, disjunction, implication, and universal and existential quantification over the sorts of the assertion language.

Having a full logic of triples also lets us express program modules as formulas of the specification logic. We can expose a module to a client as a collection of existentially quantified functions variables, and provide the client with Hoare triples describing the behavior of those functions. Furthermore, modules can existentially quantify over predicates to grant client programs access to module state without revealing the actual implementation. A client program that uses an existentially quantified specification cannot depend on the concrete implementation of this module, since the existential quantifier hides that from it – for example, we can expose a $table(t, map)$ predicate that does not reveal whether a hash table is implemented with single or double hashing.

3. Iterators, Composites and Decorators

The iterator pattern is a design pattern for uniformly enumerating the elements of a collection. The idea is that in addition to a collection, we have an auxiliary data structure called the iterator, which has an operation next. Each time next is called, it produces one more element of the collection, with some signal when all of the elements have been produced. The iterators are mutable data structures whose invariants depend on the collection, itself another mutable data structure. Therefore, most object oriented libraries state that while an iterator is active, a client is only permitted to call methods on a collection that do not change the collection state (for example, querying the size of a collection). If destructive methods are invoked (for example, adding or removing an element), it is no longer valid to query the iterator again.

We also support operations to create new iterators from old ones, and to aggregate them into composite iterators. For example, given an iterator and a predicate, we can construct a new iterator that only returns those elements for which the predicate returns true. This sort of decorator takes an iterator object, and *decorates* it to yield an iterator with different behavior. Likewise, we can take two iterators and a function, and combine them into a new, *composite* iterator that returns the result of a parallel iteration over them. These sorts of synthesized iterators are found in the `itertools` library in the Python programming language, the Google Java collections library, or the C5 library (6) for C#.

Aliasing enters into the picture, above and beyond the restrictions on the underlying collections, because iterators are stateful objects. For example, if we create a filtering iterator, and advance the underlying iterator, then what the filtering iterator will return may change. Even more strikingly, we cannot pass the same iterator twice to a parallel iteration constructor – the iterators must be disjoint in order to correctly generate the two sequences of elements to combine.

Below, we give a specification of an iterator pattern. We'll begin by describing the interface informally, in English, and then move on to giving formal specifications and explaining them.

The interface consists of two types, one for collections, and one for iterators. The operations the collection type supports are 1) creating new mutable collections, 2) adding new elements to an existing collection, and 3) querying a collection for its size. Adding new elements to a collection is a destructive operation which modifies the existing collection, whereas getting a collections size does not modify the collection.

The interface that the iterator type supports are 1) creating a new iterator on a collection, 2) destructively getting the next element from an iterator (returning an error value if the iterator is exhausted), and 3) operations to produce new iterators from old. The iterator transformations we support are 1) a filter operation, which takes an iterator along with a boolean predicate, and returns an iterator which enumerates the elements satisfying the predicate, and 2) a parallel map operation, which takes two iterators and a two-argument function, and returns an iterator which returns the result of enumerating the elements of the two iterators in parallel, and applying the function to each pair of elements.

The aliasing protocol that our iterator protocol will satisfy is essentially the same as the one the Java standard libraries specify in their documentation.

- Any number of iterators can be created from a given collection. Each of these iterators has the collection as its underlying state.
- An iterator constructed from other iterators has underlying state consisting of its arguments, as well as their backing states.
- An iterator is valid as long as none of its underlying state has been destructively modified from the time of the iterator's creation. An iterators underlying state consists of any collections it depends on, as well as any iterators it was constructed from.
- It is legal to call functions on an iterator only when it is in a valid state. Performing a destructive operation on any part of an iterator's underlying state invalidates it. For example,

adding an element to a collection in an iterator's underlying state will invalidate it, as will trying to get elements from any other iterators in its underlying state.

Now, we will describe the specification in detail. The type of collections is just the type of mutable linked lists, consisting of pointers to list cells. (For simplicity, we only consider lists of natural numbers.) An iterator is an element of a recursive tree structure in the style of an ML datatype declaration. (In Java, we would have a class hierarchy for iterators.) If it is an iterator over a single collection, then it will be in the branch Coll r, where r is a pointer to a linked list – a finger into the middle of the collection. For filtering iterators, we use a constructor of the form $\mathsf{Filter}(p, i)$, where p is a boolean predicate function and i is the iterator whose elements we are selectively yielding. We give pairwise mapping iterators via a constructor $\mathsf{Map2}(f, i_1, i_2)$, which enumerates elements of i_1 and i_2 in parallel, and applies the binary function f to those pairs to produce the yielded elements.

$$
\begin{array}{lrcl}
\text{Collection Type} & A_c & = & \mathsf{ref\ list}\ \mathbb{N} \\
\text{Iterator Type} & A_i & = & \mathsf{Coll\ of\ ref}\ A_c \\
 & & | & \mathsf{Filter\ of}\ ((\mathbb{N} \to \mathsf{bool}) \times A_i) \\
 & & | & \mathsf{Map2\ of}\ ((\mathbb{N} \times \mathbb{N} \to \mathbb{N}) \times A_i \times A_i)
\end{array}
$$

1 $\quad \exists coll : A_c \times \mathsf{seq}\ \mathbb{N} \times \mathsf{prop} \Rightarrow \mathsf{prop}.$

2 $\quad \exists iter : A_i \times \mathcal{P}^{fin}(A_c \times \mathsf{seq}\ \mathbb{N} \times \mathsf{prop}) \times \mathsf{seq}\ \mathbb{N} \Rightarrow \mathsf{prop}.$

3 $\quad \exists \mathsf{newcoll} : \mathbf{1} \to \bigcirc A_c,\ \ \mathsf{size} : A_c \to \bigcirc \mathbb{N},\ \ \mathsf{add} : A_c \times \mathbb{N} \to \bigcirc \mathbf{1}.$

4 $\quad \exists \mathsf{newiter} : A_c \to \bigcirc A_i,\ \ \mathsf{filter} : (\mathbb{N} \to \mathsf{bool}) \times A_i \to \bigcirc A_i.$

5 $\quad \exists \mathsf{map2} : (\mathbb{N} \times \mathbb{N} \to \mathbb{N}) \times A_i \times A_i \to \bigcirc A_i.$
$\quad\ \ \exists \mathsf{next} : A_i \to \bigcirc(\mathsf{option}\ \mathbb{N}).$

6 $\quad \{\mathsf{emp}\}\mathsf{run\ newcoll}()\{a : A_c.\ \exists P.\ coll(a, \epsilon, P)\}$
$\quad\ \ \mathsf{and}$

7 $\quad \forall c, P, xs.\ \{coll(c, xs, P)\}$
$\quad\quad\quad \mathsf{run\ size}(c)$
$\quad\quad\quad \{a : \mathbb{N}.\ coll(c, xs, P) \wedge a = |xs|\}$
$\quad\ \ \mathsf{and}$

8 $\quad \forall c, P, x, xs.\ \{coll(c, xs, P)\}$
$\quad\quad\quad\ \ \mathsf{run\ add}(c, x)$
$\quad\quad\quad\ \ \{a : 1.\ \exists Q.\ coll(c, x \cdot xs, Q)\}$
$\quad\ \ \mathsf{and}$

9 $\quad \forall c, P, xs.\ \{coll(c, xs, P)\}$
$\quad\quad\quad \mathsf{run\ newiter}(c)$
$\quad\quad\quad \{a : A_i.\ coll(c, xs, P) * iter(a, \{(c, xs, P)\}, xs)\}$
$\quad\ \ \mathsf{and}$

10 $\quad \forall p, i, S, xs.\ \{iter(i, S, xs)\}$
$\quad\quad\quad\ \ \mathsf{run\ filter}(p, i)$
$\quad\quad\quad\ \ \{a : A_i.\ iter(a, S, filter\ p\ xs)\}$
$\quad\ \ \mathsf{and}$
$\quad \forall f, i, S, xs, i', S', xs'.$
11 $\quad\quad\quad \{iter(i, S, xs) * iter(i', S', xs') \wedge S \cap S' = \emptyset\}$
12 $\quad\quad\quad \mathsf{run\ map2}(f, i, i')$
13 $\quad\quad\quad \{a : A_i.\ iter(a, S \cup S', map\ f\ (zip\ xs\ xs'))\}$
$\quad\ \ \mathsf{and}$

14 $\quad \forall i, S.\ \{colls(S) * iter(i, S, \epsilon)\}$
$\quad\quad\ \mathsf{run\ next}(i)$
$\quad\quad\ \{a : \mathsf{option}\ \mathbb{N}.\ colls(S) * iter(i, S, \epsilon) \wedge a = \mathsf{None}\}$
$\quad\ \ \mathsf{and}$

15 $\quad \forall i, S, x, xs.\ \ \{colls(S) * iter(i, S, x \cdot xs)\}$
$\quad\quad\ \mathsf{run\ next}(i)$
$\quad\quad\ \{a : \mathsf{option}\ \mathbb{N}.\ colls(S) * iter(i, S, xs) \wedge a = (\mathsf{Some}\ x)\}$

$$
\begin{array}{ll}
colls(\emptyset) & \equiv \mathsf{emp} \\
colls(\{(c, xs, P)\} \cup S) & \equiv coll(c, xs, P) * colls(S)
\end{array}
$$

In this specification, the predicate $coll(c, xs, P)$ (on line 1) is a three place predicate describing the state of a collection. The first argument c names the collection object that owns this state in the heap. The second argument, xs, is the abstract sequence that the collection c currently represents. As c is mutated, xs can change. The final argument, P, represents the *abstract state* of the collection. We use this argument to track whether an operation that uses c makes any destructive changes to it. Since the iterator protocol asks that we not call any destructive operations, this field lets us track whether a function has made such a change, or not, without actually revealing the internal state of the collection to a client.

The predicate $iter(i, S, xs)$ (on line 2) is the predicate describing the state of an iterator. The first argument i is the iterator constructor which owns this heap state. The finite set S is a set of triples describing the *support* of the iterator – the triples (c, xs, P) in this set describe all the collections the iterator will examine in its enumeration. That is, the collections in the support are the collections that are in i's underlying state. (We track the iterators in an underlying state by another mechanism, which we will describe subsequently.) The third argument, xs, is a sequence corresponding to the elements that the iterator has yet to produce.

newcoll (specified on line 6) creates a new, empty collection, unaliased with any other collection. The postcondition specifies that the collection is empty, and that it begin in some arbitrary abstract state. $\mathsf{size}(c)$ (line 7) takes a collection c, and returns the number of elements in c. The abstract state P of the $coll(c, xs, P)$ predicate is unchanged in the pre- and post-conditions, indicating that this function does not change the abstract state. $\mathsf{add}(c, x)$ (line 8) takes a collection c, and imperatively adds the element x to the collection. The abstract state is existentially quantified in the postcondition, indicating that it can be modified by the call to add. This ensures that clients cannot assume that the abstract state is the same before and after a call to add.

$\mathsf{newiter}(c)$ (line 9) takes a collection c, and returns an iterator over it. The returned iterator predicate $iter(a,\ \{(c, xs, P)\}, xs)$ states that a is the iterator object, whose support is the collection $\{(c, xs, P)\}$, and which will produce the elements xs (the same as the elements of c).

$\mathsf{filter}(p, i)$ (line 10) takes a boolean function p and an iterator i, and returns a new iterator which will enumerate only those elements which for which p returns true. (We express this with a logical function $filter$ called on the sequence xs.) Note that $\mathsf{filter}(p, i)$ consumes the original iterator state $iter(i, S, xs)$ – the postcondition state only mentions the state associated with the return value of the the call to filter. This reflects the fact that the filtered iterator takes ownership of the underlying iterator, in order to prevent third parties from making calls to $\mathsf{next}(i)$ and possibly changing the state of the filtered iterator.

This is also why the support only needs to track the collections in each iterator's underlying state. When we take ownership of the argument's iterator state, we prevent third parties from being able to call functions on the argument after creating the new iterator. This takes advantage of the resource-conscious nature of separation logic: a specification must have access to its footprint, and so we can hide state inside a predicate to control which operations are allowed.

$\mathsf{map2}(f, i_1, i_2)$ (lines 11-13) takes a binary function f, and two iterators i_1 and i_2. From an initial state $iter(i_1, S, xs) * iter(i_2, S', ys)$, a call to map2 will return a new iterator, whose support is the union of each argument's support, and whose supply of values is the result of pairing the elements of i_1 and i_2 and applying f to them. As with filter, map2 takes ownership of the state of its argument iterators and consumes them in the postcondition.

$\mathsf{next}(i)$ (lines 14-15) takes an iterator i as an argument. In each precondition we ask for both an iterator predicate $iter(i, S, -)$, and the collections $colls(S)$. $colls(S)$ iterates over the set S, joining each (c, xs, P) in S into a large separated conjunction $coll(c_1, xs_1, P_1) * \ldots * coll(c_k, xs_k, P_k)$. This expresses the requirement that we need *all* of the collections i depends on, all in the correct abstract state.

If the iterator is exhausted, $\mathsf{next}(i)$ returns None (line 14). If the iterator still has elements (i.e., is in a state $iter(i, S, x \cdot xs)$), it returns the first element as $\mathsf{Some}\ x$, and sets the state to $iter(i, S, xs)$ in the postcondition (line 15). We give two specifications purely for readability; it lets us avoid giving a cumbersome combined specification.

For space reasons, we cannot give the complete proof, so we give the implementation of this module and its predicates below:

```
1   list(c, ϵ)        ≡ c ↦ Nil
2   list(c, x · xs)   ≡ ∃c'. c ↦ Cons(x, c') * list(c', xs)
3   coll(c, xs, P) ≡ list(c, xs) ∧ P ∧ exact(P)
4   iter(Map2(f, i1, i2), S, xs)≡
        ∃xs1, xs2, S1, S2.
          xs = (map f (zip xs1 xs2)) ∧ (S = S1 ⊎ S2) ∧
          iter(i1, S1, xs1) * iter(i2, S2, xs2)
5   iter(Filter(p, i), S, xs) ≡
          ∃xs'. xs = filter p xs'  ∧  iter(i, S, xs')
6   iter(Coll(l), {(c, xs, P)}, zs) ≡
        ∃c', ys. xs = ys · zs ∧ l ↦ c' *
        (coll(c, xs, P) −*
            [(coll(c, xs, P) ∧ (segment(c, c', ys) * list(c', zs)))]
7   segment(c, c', ϵ)        ≡ c = c' ∧ emp
8   segment(c, c', x · xs) ≡ ∃c''. c ↦ Cons(x, c'') * segment(c'', c', xs)
9   newcoll() ≡ [new_{ref list ℕ} Nil]
10  size(lst) ≡ [letv cell = [!lst] in
11               run case(cell,
12                        Nil → [0],
13                        Cons(h, t) → [letv n = size(t) in n + 1])]
14  add(c, x) ≡ [letv cell = [!c] in
15               letv r = [new_{ref list ℕ} cell] in
16               c := Cons(x, r)]
17  newiter(c) ≡ [letv r = [new_{ref ref list ℕ}(c)] in Coll r]
18  filter(p, i) ≡ [Filter(p, i)]
19  map2(f, i1, i2) ≡ [Map2(f, i1, i2)]
20  next(Coll r) ≡ [letv list = [!r] in
21                  letv cell = [!list] in
22                  run case(cell,Nil →[None],
23                           Cons(x, t) →
                                [letv dummy = [r := t] in
                                 Some x])]
24  next(Filter(p, i)) ≡ [letv v = next(i) in
25                        run case(v,None → [None]
26                                 Some x →
                                      if(p x,[Some x],
                                         next(Filter(p, i))))]
27  next(Map2(f, i1, i2)) ≡
        [letv v1 = next i1 in
28       letv v2 = next i2 in
29       case(v1, None → None,
30            Some x1 → case(v2, None → None,
31                           Some x2 → Some(f(x1, x2))))]
```

The inductive predicate $list(c, xs)$ is defined on lines 1-2. It says that c is an empty list if it is a pointer to the value Nil, and that is a non-empty list if c points to $\mathsf{Cons}(x, c')$, where x is the first element of the sequence and c' disjointly represents the tail of the list.

The predicate $coll(c, xs, P)$ (line 3) is defined as the conjunction of a linked list predicate $list(c, xs)$ and an exact predicate P. An exact predicate is one that is true of exactly one heap, and the conjunction of P with $list(c, xs)$ ensures that this linked list cannot be modified at all, without changing P. This is a relatively common idiom when verifying programs whose invariants depend on a notion of destructive update. It is necessary because the predicate $list(c, xs)$ can be maintained even when c is modified. Concretely, suppose that xs is nonempty. Then any modification of c which changes c's tail to some different physical list representing the same tail sequence will still satisfy the predicate $list(c, xs)$. However, if a predicate is known to be exact, then there is exactly one heap it can be, and so no other heaps can satisfy it.

The inductive definition of $iter$ is on lines 4-6. The base case (line 6) is the most complex. In essence, it says that the iterator's pointer is a finger into the middle of the list, c'.

We express this requirement with the clause $coll(c, xs, P) \wedge (segment(c, c', ys) * list(c', zs))$, which says that heap associated with the collection can be viewed in two ways, both as the collection itself, and as a partial list segment representing what has already been seen, together with the remainder of the list beginning at c'. However, the iterator state does not own the collection; this clause will only hold if we have the collection state in addition to the iterator. We use the magic wand to say this: it means that when we supply the iterator with $coll(x, xs, P)$, then we can view the state in this way. So as a whole, the predicate $iter(\mathsf{Coll}(l), \{(c, xs, P)\}, zs)$ can be read as "the iterator is a pointer l to some c', such that if we are given the collection state $coll(c, xs, P)$, then c' is a pointer into the middle of the list, with zs as the remaining elements".

The reason we go to this effort is to simplify the specification and proof of client programs – we could eliminate the use of the magic wand in the base case if iterators owned their collections, but this would complicate verifying programs that use multiple iterators over the same collection, or which want to call pure methods on the underlying collection. In those cases, the alternative would require us to explicitly transfer ownership in the proofs of client programs, which is quite cumbersome, and forces clients to reason using the magic wand. The current approach isolates that reasoning within the proof of the implementation.

On line 5, the $\mathsf{Filter}(p, i)$ case says that i must have some iterator state with the same support S, and a sequence of elements xs' that yield xs once filtered. We use the logical function $filter$, defined at the assertion level, to describe the effect on the iterator's logical sequence. On line 4, we specify $iter$ for the $\mathsf{Map2}(f, i_1, i_2)$ case. We assert that the support S must be divisible into two disjoint parts, one for i_1 and one for i_2, and that there is iterator state for i_1 and i_2, and that the sequences i_1 and i_2 combine to yield the output sequence.

In both of these cases, we define the behavior of the imperative linked list in terms of purely functional sequences. This is a very common strategy in many verification efforts, but here we see that we can use it in a local way – in the base case, we are forced to consider issues of aliasing and ownership, but in the inductive cases we can largely avoid that effort.

The collection operations are mostly straightforward – newcoll (line 9) allocates a new linked list, size (lines 10-13) recursively traverses the list to calculate the length, and add (lines 14-16) adds a cons cell to the front of the list. newiter (lines 17) allocates a pointer to the front of the list, and then wraps that in the Coll constructor. For filter (line 18) and map2 (line 19), we just apply the constructor to the arguments and then return the result.

next (lines 20-31) recursively walks down the structure of the iterator tree, and combines the results from the leaves upwards. The base case is the Coll r case (lines 20-23). The iterator pointer is doubly-dereferenced, and then the contents examined. If the end of the list has been reached and the contents are Nil, then None is returned to indicate there are no more elements. Otherwise, the pointer r is advanced, and the head returned as the observed value. The Filter(p,i) case (lines 24-26) will return None if i is exhausted, and if it is not, it will pull elements from i until it finds one that satisfies p, calling itself recursively until it succeeds or i is exhausted. Finally, in the $\mathsf{Map2}(f, i_1, i_2)$ case (lines 27-31), next

will draw a value from both i_1 and i_2, and will return None if either is exhausted, and otherwise it will return f applied to the pair of values.

Below, we give an example use of this module in annotated program style:

1 $\{\mathsf{emp}\}$
2 letv $c_1 = \mathsf{newcoll}()$ in
3 $\{\exists P_1'.\ coll(c_1, \epsilon, P_1')\}$
4 $\{coll(c_1, \epsilon, P_1)\}$
5 letv $() = \mathsf{add}(c_1, 4)$ in
6 $\{\exists P_2'.\ coll(c_1, 4 \cdot \epsilon, P_2')\}$
7 $\{coll(c_1, 4 \cdot \epsilon, P_2)\}$
8 letv $() = \mathsf{add}(c_1, 3)$ in
9 letv $() = \mathsf{add}(c_1, 2)$ in
10 $\{coll(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4)\}$
11 letv $c_2 = \mathsf{newcoll}()$ in
12 letv $() = \mathsf{add}(c_2, 3)$ in
13 letv $() = \mathsf{add}(c_2, 5)$ in
14 $\{coll(c_2, 5 \cdot 3 \cdot \epsilon, Q_2) * coll(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4)\}$
15 letv $i_1 = \mathsf{newiter}(c_1)$ in
16 $\{iter(i_1, \{(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4)\}, 2 \cdot 3 \cdot 4 \cdot \epsilon)$
$* coll(c_2, 5 \cdot 3 \cdot \epsilon, Q_2) * coll(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4)\}$
17 letv $i_1' = \mathsf{filter}(even?, i_1)$ in
18 $\{iter(i_1', \{(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4)\}, 2 \cdot 4 \cdot \epsilon)$
$* coll(c_2, 5 \cdot 3 \cdot \epsilon, Q_2) * coll(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4)\}$
19 letv $i_2 = \mathsf{newiter}(c_2)$ in
20 $\{iter(i_1', \{(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4)\}, 2 \cdot 4 \cdot \epsilon)$
$* iter(i_2, \{(c_2, 5 \cdot 3 \cdot \epsilon, Q_2)\}, 5 \cdot 3 \cdot \epsilon)$
$* coll(c_2, 5 \cdot 3 \cdot \epsilon, Q_2) * coll(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4)\}$
21 letv $i = \mathsf{map2}(plus, i_1', i_2)$ in
22 $\{iter(i, \{(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4), (c_2, 5 \cdot 3 \cdot \epsilon, Q_2)\}, 7 \cdot 7 \cdot \epsilon)$
$* coll(c_2, 5 \cdot 3 \cdot \epsilon, Q_2) * coll(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4)\}$
23 letv $n = \mathsf{size}(c_2)$ in
24 $\{n = 2 \wedge iter(i, \{(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4), (c_2, 5 \cdot 3 \cdot \epsilon, Q_2)\}, 7 \cdot 7 \cdot \epsilon)$
$* coll(c_2, 5 \cdot 3 \cdot \epsilon, Q_2) * coll(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4)\}$
25 letv $x = \mathsf{next}(i)$ in
26 $\{n = 2 \wedge x = \mathsf{Some}\ 7 \wedge$
$iter(i, \{(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4), (c_2, 5 \cdot 3 \cdot \epsilon, Q_2)\}, 7 \cdot \epsilon)$
$* coll(c_2, 5 \cdot 3 \cdot \epsilon, Q_2) * coll(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4)\}$
27 $\mathsf{add}(c_2, 17)$
28 $\{n = 2 \wedge x = \mathsf{Some}\ 7 \wedge$
$iter(i, \{(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4), (c_2, 5 \cdot 3 \cdot \epsilon, Q_2)\}, 7 \cdot \epsilon)$
$* (\exists Q_3.\ coll(c_2, 17 \cdot 5 \cdot 3 \cdot \epsilon, Q_3)) * coll(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4)\}$

In line 1 of this example, we begin in an empty heap. In line 2, we create a new collection c_1, which yields us the state $\exists P_1'.\ coll(c_1, \epsilon, P_1')$, with an existentially quantified abstract state.

Because P_1' is existentially quantified, we do not know what value it actually takes on. However, if we prove the rest of the program using a freshly-introduced variable P_1, then we know that the rest of the program will work for *any* value of P_1, because free variables are implicitly universally quantified. So it will work with whatever value P_1' had. So we drop the quantifier on line 4, and try to prove this program with the universally-quantified P_1.[1]

This permits us to add the element 4 to c_1 on line 5. Its specification puts the predicate $coll()$ on line 6 again into an existentially quantified state P_2'. So we again replace P_2' with a fresh variable P_2 on line 7, and will elide these existential introductions and unpackings henceforth.

In lines 8-9, we add two more elements to c_1, and on lines 11-13, we create another collection c_2, and add 3 and 5 to it, as can be seen in the state predicate on line 14. On line 15, we create the iterator i_1 on the collection c_1. The $iter$ predicate on line 16 names i_1 as its value, and lists c_1 in state P_4 as its support, and promises to enumerate the elements 2, 3, and 4.

[1] A useful analogy is the existential elimination rule in the polymorphic lambda calculus: we prove that we can use an existential by showing that our program is well-typed no matter what the contents of the existential are.

On line 17, $\mathsf{filter}(even?, i_1)$ creates the new iterator i_1'. This iterator yields only the even elements of i_1, and so will only yield 2 and 4. On line 18, i_1's iterator state has been consumed to make i_1''s state. We can no longer call $\mathsf{next}(i_1)$, since we do not have the resource invariant needed to prove anything about that call. Thus, we cannot write a program that would break i_1''s representation invariant.

On line 19, we create a third iterator i_2 enumerating the elements of c_2. The state on line 20 now has predicates for i_1', i_2, c_1 and c_2. On line 21, $\mathsf{map2}(plus, i_1', i_2)$ creates a new iterator i, which produces the pairwise sum of the elements of i_1' and i_2, and consumes the iterator states for i_1' and i_2 to yield the state for the new iterator i. Note that the invariant for i does not make any mention of what it was constructed from, naming only the collections it needs as support.

On line 23, the size call on c_2 illustrates that we can call non-destructive methods while iterators are active. The call to $\mathsf{next}(i)$ on line 24 binds Some 7 to x, and the the iterator's sequence argument (line 27) shrinks by one element. On line 28, we call $\mathsf{add}(c_2, 17)$ the state of c_2 changes to $\exists Q_3.\ coll(c, 17 \cdot 5 \cdot 3 \cdot \epsilon, Q_3)$ (line 27). So we can no longer call $\mathsf{next}(i)$, since it needs c_2 to be in the state Q_2.

Discussion. This example shows a pleasant synergy between higher-order quantification and separation logic. We can give a relatively simple specification to the clients of the collection library, even though the internal invariant is quite subtle (as the use of the magic wand suggests). Higher-order logic also lets us freely define new data types, and so our specifications can take advantage of the pure, non-imperative nature of the mathematical world, as can be seen in the specifications of the filter and map2 functions – we can use equational reasoning on purely functional lists in our specifications, even though our algorithms are imperative.

4. The Flyweight and Factory Patterns

The flyweight pattern is a style of cached object creation. Whenever a constructor method is called, it first consults a table to see if an object corresponding to those arguments has been created. If it has, then the preexisting object is returned. Otherwise, it allocates a new object, and updates the table to ensure that future calls with the same arguments will return this object. Because objects are re-used, they become pervasively aliased, and must be used in an immutable style to avoid surprising updates. (Functional programmers call this style of value creation "hash-consing".)

This is an interesting design pattern to verify, for two reasons. First, the constructor has a memo table to cache the result of constructor calls, which needs to be hidden from clients. Second, this pattern makes pervasive use of aliasing, in a programmer-visible way. In particular, programmers can test two references for identity in order to establish whether two values are equal or not. This allows constant-time equality testing, and is a common reason for using this pattern. Therefore, our specification has to be able to justify this reasoning.

Below, we specify a program that uses the flyweight pattern to create and access glyphs (i.e., refs of pairs of characters and fonts) of a particular font f. We have a function newglyph to create new glyphs, which does the caching described above, using a predicate variable I to refer to the table invariant; and a function getdata to get the character and font information from a glyph.

Furthermore, these functions will be created by a call to another function, make_flyweight, which receives a font as an argument and will return appropriate newglyph and getdata functions.

Flyweight(I : prop,
newglyph : char $\to \bigcirc$glyph,
getdata : glyph $\to \bigcirc$(char $\times$ font),

```
        f : font) ≡
1   ∃glyph : glyph × char × font ⇒ prop.
2    ∀c, S. {I ∧ chars(S)}
             run newglyph(c)
             {a : glyph. I ∧ chars({(a, (c, f))} ∪ S)}
     and
3   ∀l, c, f, P. {glyph(l, c, f) ∧ P}
                  run getdata(l)
                  {a : char × font. glyph(l, c, f) ∧ P ∧ a = (c, f)}
     and
4   {∀l, l′, c, c′. I ∧ glyph(l, c, f) ∧ glyph(l′, c′, f′) ⊃
                      (l = l′ ⟺ (c = c′ ∧ f = f′))}

chars(∅)                  ≡ ⊤
chars({(l, (c, f))} ∪ S)  ≡ glyph(l, c, f) ∧ chars(S)
```

In the opening , we informally parametrize our specification over the predicate variable I, the function variable newglyph, the function variable getdata, and the variable f of font type. The reason we do this instead of existentially quantifying over them will become clear shortly, once we see the factory function that creates flyweight constructors.

On line 1, we assert the existence of a three-place predicate $glyph(l, c, f)$, which is read as saying the glyph value l is a glyph of character c and font f.

On line 2, we specify the newglyph procedure. Its precondition says the pre-state must be the private flyweight state I, and that this state overlaps with the character state for the glyph/data pairs in S. The definition of $chars$ takes a set of glyph/data pairs and produces the conjunction of $glyph(l, c, f)$ for all the $(l, (c, f)) \in S$. Running newglyph(c) will yield a postcondition state in which $(a, (c, f))$ is added to the set S – that is, the postcondition state is $I \wedge chars(S) \wedge glyph(a, c, f)$.

The intuition for this specification is that I represents the private state of the memo table. We use an ordinary conjunction instead of the separating conjunction in the definition of $chars$ to say that the different glyphs may alias with each other, and with the private state I. In other words, even though the newglyph function returns a new glyph value, the ownership of the state associated with that value is not transferred – it remains with the constructor. All the client can know is that some glyph state exists for the value it created, and that the glyph state is owned by the constructor.

On line 3, we specify the getdata function. If the predicate $glyph(r, c, f)$ is in the precondition, then getdata(r) will return (c, f). To enforce the flyweight invariant that the glyph objects are read-only, we conjoin the pre- and post-conditions with an arbitrary predicate variable P. Since P must be preserved for any instantiation, we know that getdata cannot make any changes to the underlying data. (It might be possible to internalize this style of argument into the logic via some sort of relational parametricity proof. However, we have not yet done so.)

Finally, on lines 4, we give an axiom about the interaction of I and $glyph(l, c, f)$, which says that if we know that $I \wedge glyph(l, c, f) \wedge glyph(l', c', f')$ holds, then $l = l'$ holds if and only $c = c'$ and $f = f'$. This axiom gives clients the ability to take advantage of the fact that we are caching object creation and conclude that two calls to newglyph with the same arguments will yield the same result.

Requiring an axiom of separation hold of the abstract predicates is how we state reasoning principles about aliasing as part of the interface of a module. When we verify the implementation, we will need to give concrete definitions of I and $glyph$, and show that the formula is actually a tautology of separation logic.

The specification of the flyweight factory looks like this:

```
1   ∃make_flyweight : font → ◯((char → ◯glyph)×
                               (glyph → ◯(char × font))).
2    ∀f. {emp}
         run make_flyweight(f)
         {a. ∃I : prop. I ∧ Flyweight(I, fst a, snd a, f) valid}
```

Here, we assert the existence of a function make_flyweight, which takes a font f as an input argument, and returns two functions to serve as the getchar and getdata functions of the flyweight. In the postcondition, we assert the existence of some private state I, which contains the table used to cache glyph creations.

This pattern commonly arises when encoding aggressively object-oriented designs in a higher-order style — we call a function, which creates a hidden state, and returns other procedures which are the only way to access that state. This style of specification resembles the existential encodings of objects into type theory. The difference is that instead of making the fields of an object an existentially quantified *value* (21), we make use of existentially-quantified *state*.

Below, we define make_flyweight and its predicates:

```
1    make_flyweight ≡
       λf :font.
2      [letv t = newtable() in
3       letv newglyph =
4           [λc.[letv x = lookup(t, c) in
5                run case(x,None → [letv r = [new_glyph(c, f)] in
6                                   letv _ = update(t, c, r) in r],
7                         Some r → [r])] in
8       letv getdata = [λr. [!r]] in
9       (newglyph, getdata)]

glyph(r, c, f) ≡ r ↦ (c, f) ∗ ⊤

I ≡ table(t, mapping) ∗ refs(mapping, dom(mapping))

refs(mapping, ∅)          ≡ emp
refs(mapping, {c} ∪ D)    ≡ mapping(c) ↦ (c, f) ∗ refs(f, D)
```

In this implementation we have assumed the existence of a hash table implementation with operations newtable, lookup, and update, whose specifications we omit for space reasons. The make_flyweight function definition takes a font argument f, and then in its body it creates a new table t. It then constructs two functions as closures which capture this state (and the argument f) and operate on it. In lines 4-7, we define $newglyph$, which takes a character and checks to see (line 5) if it is already in the table. If it is not (lines 5-6), it allocates a new glyph reference, stores it in the table, and returns the reference. Otherwise (line 7), it returns the existing reference from the table. On lines 8, we define $getdata$, which dereferences its pointer argument and returns the result. This implementation does no writes, fulfilling the promise made in the specification. The definition of the invariant state I describes the state of the table t (and $mapping$), which are hidden from clients.

Observe how the post-condition to make_flyweight nests the existential state I with the validity assertion to specialize the flyweight spec to the *dynamically* created table. Each created flyweight factory receives its own private state, and we can reuse specifications and proofs with no possibility that the wrong getdata will be called on the wrong reference, even though they have compatible types.

5. Subject-Observer

The subject-observer pattern is one of the most characteristic patterns of object-oriented programming, and is extensively used in GUI toolkits. This pattern features a mutable data structure called the *subject*, and a collection of data structures called *observers* whose invariants depend on the state of the subject. Each observer registers a callback function with the subject to ensure it remains in

sync with the subject. Then, whenever the subject changes state, it iterates over its list of callback functions, notifying each observer of its changed state. While conceptually simple, this is a lovely problem for verification, since every observer can have a different invariant from all of the others, and the implementation relies on maintaining lists of callback functions in the heap.

In our example, we will model this pattern with one type of subjects, and three functions. A subject is simply a pair, consisting of a pointer to a number, the subject state; and a list of observer actions, which are imperative procedures to be called with the new value of the subject whenever it changes. There is a function newsub to create new subjects; a function register, which attaches observer actions to the subject; and finally a function broadcast, which updates a subject and notifies all of its observers of the change.

We give a specification for the subject-observer pattern below:

```
1   ∃sub : A_s × ℕ × seq ((ℕ ⇒ prop) × (ℕ → ◯1)).
2   ∃newsub : ℕ → ◯A_s,
3   ∃register : A_s × (ℕ → ◯1) → ◯1,
4   ∃broadcast : A_s × ℕ → ◯1.

5   ∀n. {emp}run newsub(n){a : A_s. sub(a, n, ϵ)}
     and
6   ∀f, O, s, n, os.(∀i, k.{O(i)}run f(k){a : 1. O(k)})
7                       implies {sub(s, n, os)}
8                               run register(s, f)
9                               {a : 1. sub(s, n, (O, f) · os)}
     and
10  ∀s, i, os, k. {sub(s, i, os) * obs(os)}
                   run broadcast(s, k)
                   {a : 1. sub(s, k, os) * obs_at(os, k)}

obs(ϵ)          ≡ emp
obs((O, f) · os) ≡ (∃i. O(i)) * obs(os)

obs_at(ϵ, k)          ≡ emp
obs_at((O, f) · os, k) ≡ O(k) * obs_at(os, k)
```

On line 1 we assert the existence of a three-place predicate $sub(s, n, os)$. The first argument is the subject s's whose state this predicate represents. The second argument n is the data the observers depend on, and the field os is a sequence of callbacks paired their invariants. That is, os is a sequence of pairs, consisting of the observer functions which act on a state, along with the predicate describing what that state should be.

On lines 2-4, we assert the existence of newsub, register and broadcast, which create a new subject, register a callback, and broadcast a change, respectively.

register is a higher order function, which takes a subject and an observer action its two arguments. The observer action is a function of type $\mathbb{N} \to \bigcirc 1$, which can be read as saying it takes the new value of the subject and performs a side-effect. Because register depends on code, its specification must say how this observer action should behave. register's specification on lines 6-9 accomplishes this via an implication over Hoare triples. It says that *if* the function f is a good observer callback, *then* it can be safely registered with the subject. Here, a "good callback" f is one that takes an argument k and sends an observer state to $O(k)$. If this condition is satisfied, then register(s, f) will add the pair (O, f) to the sequence of observers in the sub predicate.

broadcast updates a subject and all its interested observers. The precondition state of broadcast(s, k) requires the subject state $sub(s, n, os)$, and all of the observer states $obs(os)$. The definition $obs(os)$ takes the list of observers and yields the separated conjunction of the observer states. So when broadcast is invoked, it can modify the subject and any of its observers. Then, after the call, the postcondition puts the sub predicate and all of the observers in the same state k. The $obs\_at(os, k)$ function generates the separated conjunction of all the O predicates, all in the same state k.

The implementation follows:

```
1   A_s ≡ ref ℕ × ref list (ℕ → ◯1)
2   sub(s, n, os) ≡fst s ↦ n * list(snd s, map snd os) ∧ Good(os)
3   Good(ϵ)           ≡ ⊤
4   Good((O, f) · os) ≡ (∀i, k. {O(i)}run f(k){a : 1. O(k)}) valid
                        ∧Good(os)
5   register(s, f) ≡ [letv cell = [!(snd s)] in
6                     letv r = [new_{ref list (ℕ→◯1)} cell] in
7                     snd s := Cons(f, r)]
8   broadcast(s, k) ≡
9         [letv dummy = [fst s := k] in loop(k, snd s)]
10  loop(k, list) ≡
        [letv cell = [!list] in
11       run case(cell,Nil → [()],
12                     Cons(f, tl) → [letv dummy = f(k) in
13                                    run loop(k, tl)])
14  newsub(n) ≡ [letv data = new_ℕ n in
15               letv callbacks = new_{list (ℕ→◯1)} Nil in
16               (data, callbacks)]
```

In line 1, we state concrete type of the subject A_s is a pair of a pointer to a reference, and a pointer to a list of callback functions. (This is *not* an existential quantifier. Since our language is simply typed, we have no form of type abstraction and simply use A_s as an abbreviation.) On line 2, we define the three-place subject predicate, $sub(s, n, os)$. The first two subclauses of the predicate's body describe the physical layout of the subject, and assert that the first component of s should point to n, and that the second component of s should be a linked list containing the function pointers in os. (The $list$ predicate is described in Section 3, when we give the definition of the iterator predicates.)

Then we require that os be "Good". $Good$-ness is defined on lines 3 and 4, and says a sequence of predicates and functions is good when every (O, f) pair in the sequence satisfies the same validity requirement the specification of register demanded – that is, that each observer function f update O properly. Note that we interleave assertions and specifications to constrain the behavior of code stored in the heap.

Next, we give the implementations of register and broadcast. register, on lines 5-7, adds its argument to the list of callbacks. Though the code is trivial, its correctness depends on the fact the $Good$ predicate holds for the extended sequence — we use the fact that the argument f updates O properly to establish that the extended list remains $Good$.

broadcast, on lines 8-9, updates the subject's data field (the first component), and then calls loop (on lines 10-13) to invoke all the callbacks. loop$(k,$ snd $s)$ just recurs over the list and calls each callback with argument k. The correctness of this function also relies on the $Good$ predicate – each time we call one of the functions in the observer list, we use the hypothesis of its behavior given in $Good(os)$ to be able to make progress in the proof.

Below, we give a simple piece of client code using this interface.

```
1   {emp}
2   letv s = newsub(0) in
3   {sub(s, 0, ϵ)}
4   letv d = new_ℕ(0) in
5   letv b = new_bool(true) in
6   {sub(s, 0, ϵ) * d ↦ 0 * b ↦ true}
7   letv () = register(s, f) in
8   {sub(s, 0, (double, f) · ϵ) * double(0) * b ↦ true}
9   letv () = register(s, g) in
10  {sub(s, 0, (even, g) · (double, f) · ϵ) * double(0) * even(0)}
11  broadcast(s, 5)
```

12 $\{sub(s, 5, (even, g) \cdot (double, f) \cdot \epsilon) * double(5) * even(5)\}$
13 $\{sub(s, 5, (even, g) \cdot (double, f) \cdot \epsilon) * d \mapsto 10 * b \mapsto \mathsf{false}\}$

14 $f \equiv \lambda n : \mathbb{N}.\ [d := 2 \times n]$
15 $double(n) \equiv d \mapsto (2 \times n)$
16 $g \equiv \lambda x : \mathbb{N}.\ [b := even?(x)]$
17 $even(n) \equiv b \mapsto even?(n)$

We start in the empty heap, and create a new subject s on line 2. On line 4, we create a new reference to 0, and on line 5, we create a reference to true. So on line 6, the state consists of a subject state, and two references. On line 7, we call register on the function f (defined on line 14), which sets d to twice its argument. To the observer list in sub, we add f and the predicate $double$ (defined on line 15), which asserts that indeed, d points to two times the predicate argument. On line 8, we call register once more, this time with the function g (defined on line 16) as its argument, which stores a boolean indicating whether its argument was even into the pointer b. Again, the state of sub changes, and we equip g with the $even$ predicate (defined on line 17) indicating that b points to a boolean indicating whether the predicate argument was even or not. Since $d \mapsto 0$ and $b \mapsto \mathsf{true}$ are the same as $double(0)$ and $even(0)$, so we can write them in this form on line 10. We can now invoke $\mathsf{broadcast}(s, 5)$ on line 11, and correspondingly the states of all three components of the state shift in line 12. In line 13, we expand $double$ and $even$ to see d points to 10 (twice 5), and b points to false (since 5 is odd).

Discussion. One nice feature of the proof of the subject-observer implementation is that the proofs are totally oblivious to the concrete implementations of the notification callbacks, or to any details of the observer invariants. Just as existential quantification hides the details of a module implementation from the clients, the universal quantification in the specification of register and broadcast hides all details of the client callbacks from the proof of the implementation – since they are free variables, we are unable to make any assumptions about the code or predicates beyond the ones explicitly laid out in the spec. Another benefit of the passage to higher-order logic is the smooth treatment of observers with differing invariants; higher-order quantification lets us store and pass formulas around, making it easy to allow each callback to have a totally different invariant.

6. Ynot Experiments

In this section we give a brief description of our experiments with translating the design pattern specifications and implementations from the earlier sections into Hoare Type Theory (HTT) and verifying them in the Ynot implementation of HTT. More details can be found in the technical report (26).

These experiments serve to (1) increase our confidence in the earlier given specifications and implementations and their associated paper proofs; (2) provide a starting point for a comparison of the specification logic of the present paper and the Ynot type theory; (3) exercise the Ynot implementation.

Ynot is an axiomatic extension to the Coq proof assistant, that supports writing, reasoning about, and extracting higher-order, dependently-typed programs with side-effects (14). Coq already includes a powerful functional language that supports dependent types, but that language is limited to pure, total functions. Ynot extends Coq with support for computations that may have effects such as non-termination, accessing a mutable store, and throwing/catching exceptions. The axioms of Ynot form a small trusted computing base which has been formally justified in previous work on Hoare Type Theory (HTT) (13; 12; 19).

As in the specification logic described in the earlier sections, Ynot also makes use a monads, to ensure a monadic separation of effects and pure Coq. In Ynot specifications are types and one of the types is the monadic type of computations $\{P\}x : \tau\{Q\}$; if a computation has this type and it is run in a heap i satisfying P and it terminates, then it will produce a value x of type τ and result in a new heap j such that the predicate $Q(i, j)$ holds. Loosely speaking, we may thus think of Ynot as a type theory corresponding to the specification logic for Idealized ML presented earlier under a Curry-Howard style correspondence. Following this intuition, we have experimented with translating the earlier described design pattern specifications and implementations into Ynot and formally verified them in Ynot. We now describe the translations and the lessons learned.

Note first, however, that in Ynot post-conditions are expressed in terms of both the initial and the final heap. This is an alternative to the use of logical variables as expressed by universally quantifying over variables whose scope extends to both the pre- and post-condition. We can thus translate an Idealized ML specification,

$$\forall x : \tau.\ \{P(x)\}\ comp\ \{a : 1.\ Q(x)\}$$

into the following Ynot type

$$\{\lambda i : heap.\ \exists x : \tau.\ P\ x\ i\}$$
$$a : 1$$
$$\{\lambda i : heap.\ \lambda j : heap.\ \forall x : \tau.\ P\ x\ i \rightarrow Q\ x\ j\}$$

where i is the initial heap and j is the terminal heap. We will usually abbreviate this type as follows:

$$\{i.\ \exists x.\ P\ x\ i\}\ a : 1\ \{i\ j.\ \forall x.\ P\ x\ i \rightarrow Q\ x\ j\}$$

6.1 Flyweight in Ynot

Besides Hoare triples, Idealized ML's specification language contains specifications of the form $\{P\}$, for asserting that P is true. In the flyweight specification this is used to express that calling $getdata$ with the same character multiple times, produces the same reference. In HTT we can express that a proposition P is true by returning an element of the subset type, $\{x : 1 \mid P\}$, where x is not free in P.

The assertion language of Idealized ML also contains an expression for asserting that a given specification holds. In the Flyweight specification this is used in the post-condition of $make\_flyweight$, to assert that the code returned implements a Flyweight. In HTT, we can express the same by simply giving a more precise type for the return value of the $make\_flyweight$ computation.

In the Ynot implementation, we have generalized the specification, such that the computation can generate a flyweight for values of an arbitrary monotype. The flyweight factory computation therefore also has to take as an argument, a function, α_{eq}, for deciding equality between α values.

The rest of the specification can be translated almost directly into HTT, however, we have made a few changes, to simplify the formal verification of the implementation in Ynot.

- In the specification of $newchar$, instead of using a set to associate arguments with objects, we have used a partial function (i.e., a total function from α to option loc).
- In the above specification the predicate I has to specify the representation of both the object table and the objects. We have split I into two predicates, $table$ and $refs$, and changed the precondition of $newchar$ to the HTT equivalent of $table(...) * (refs(...) \wedge chars(S))$, to make it explicit that the object table and the objects are in separate subheaps, to simplify verification.

The final HTT type of the Flyweight factory thus looks as follows:

$$\Pi\alpha : \mathbf{mono}.\ \Pi\alpha_{eq} : (\Pi x : \alpha.\ \Pi y : \alpha.\ \{z : 1 \mid x = y\} + \{z : 1 \mid x \neq y\}).$$
$$\{i.\ emp\ i\}$$
$$r : \Sigma table : (\alpha \to \mathsf{option}\ \ loc) \to heap \to Prop.$$
$$\Sigma refs : (\alpha \to \mathsf{option}\ \ loc) \to heap \to Prop.$$
$$\Sigma objat : loc \to \alpha \to heap \to Prop.$$
$$\Sigma prf_1 : \{x : 1 \mid \forall h, l, l', a, a', f.\ objat\ l\ a\ h \wedge objat\ l'\ a'\ h \wedge$$
$$refs\ f\ h \to (l = l' \leftrightarrow a = a')\}.$$
$$\Pi a : \alpha.$$
$$\{i.\ \exists f.\ (table\ f * (\lambda h.\ allobjat(\alpha, objat, f, h) \wedge refs\ f\ h))\ i\}$$
$$l : loc$$
$$\{i\ j.\ \forall f.\ (table\ f * (\lambda h.\ allobjat(\alpha, objat, f, h) \wedge refs\ f\ h))\ i \to$$
$$((\forall l'.\ f\ a = Some\ l' \to l = l') \wedge$$
$$(table\ f[a \mapsto l] * (\lambda h.\ allobjat(\alpha, objat, f[a \mapsto l], h) \wedge$$
$$refs\ f[a \mapsto l]\ h))\ j)\} \times$$
$$\Pi l : loc.$$
$$\{i.\ \exists a : \alpha,\ objat\ l\ a\ i\}$$
$$r : \alpha$$
$$\{i\ j.\ \forall a : \alpha,\ objat\ l\ a\ i \to (i = j \wedge r = a)\}$$
$$\{i\ j.\ ((fst\ r)\ []\ * (\lambda h.\ allobjat(\alpha, fst\ (snd\ (snd\ r)), [], h) \wedge$$
$$(fst\ (snd\ r))\ []\ h))\ j\}$$

where

$$allobjat(\alpha, objat, f, h) \equiv \forall l : loc, o : \alpha.\ f\ o = Some\ l\ \to$$
$$(objat\ l\ o * (\lambda h.\ True))\ h$$

and $[] \equiv (\lambda x.\ None)$.

We were able to formally verify that the earlier given implementation of the Flyweight pattern has the above type in Ynot.[2]

6.2 Iterators in Ynot

The translation of the Iterator specification and implemenation and the formal verification in Ynot was straightforward, except for the verification of `next`, when the iterator is a `Map2`.[3] In that case, the implementation makes two recursive calls to `next` that each work on two subheaps of the initial heap and the current Ynot implementation based on the $nextvc$ tactic (see (14)) for simplifying proof obligations forces one to prove some preciseness properties because of the use of binary post-conditions. It is unclear whether the preciseness problem encountered is a limitation of binary post-conditions in general or the current Ynot implementation; we think it is the latter. We did not finish the proof for `next` in this case, either with or without $nextvc$ (without $nextvc$ the proof became too long for us to finish by hand). New versions of Ynot should provide better tactic support for such examples. We did succeed in completing the formal verification of the iterator pattern without the `Map2` iterator.

6.3 Subject-Observers in Ynot

The Idealized ML implementation of the subject-observer pattern uses an assertion S valid in the predicate $good$ to express that callback functions are "good". HTT does not support this form of assertion. However, since specifications are types, we can express the type of pairs of (O, f) such that f is a good call function with respect to the predicate O with the following type:

$$T \equiv \Sigma O : \mathbb{N} \to heap \to Prop.$$
$$(\Pi m : \mathbb{N}.\ \{i.\ \exists k : \mathbb{N}.\ O\ k\ i\}\ a : 1\ \{i\ j.\ \forall k : \mathbb{N}.\ O\ k\ i \to O\ m\ j\})$$

Hence, we can restrict the quantification of os in the specification of $broadcast$ and $register$ to lists of good callback functions, list T. We can thus express the subject-observer pattern with the following HTT type:

$$\Sigma\alpha : Type.\ \Sigma sub : \alpha \times \mathbb{N} \times \mathsf{list}\ T \to heap \to Prop.$$
$$\Pi n : \mathbb{N}.\ \{i.\ emp\ i\}a : \alpha\{i\ j.\ sub\ (a, n, [])\ j\} \times$$
$$\Pi a : \alpha.\ \Pi t : T.$$
$$\{i.\ \exists n : \mathbb{N}, os : list\ T.sub\ (a, n, os)\ i\}$$
$$r : 1$$
$$\{i\ j.\ \forall n : \mathbb{N}, os : list\ T.\ sub\ (a, n, os)\ i \to sub\ (a, n, t :: os)\ j\} \times$$
$$\Pi a : \alpha.\ \Pi m : \mathbb{N}.$$
$$\{i.\ \exists n : \mathbb{N}, os : list\ T.\ (sub\ (a, n, os) * obs\ os)\ i\}$$
$$r : 1$$
$$\{i\ j.\ \forall n : \mathbb{N}, os : list\ T.\ (sub\ (a, n, os) * obs\ os)\ i$$
$$\to (sub\ (a, m, os) * obs\_at\ (os, m))\ j\}$$

In the Idealized ML implementation of the subject-observer, the registered callback functions are stored in the heap. Since types and specifications are separate in Idealized ML, the type of these computations can be very weak, i.e., $\mathbb{N} \to \bigcirc 1$, because the specification language allows us to express that if these are "good" callback functions then the broadcast computation will do a broadcast when performed. In HTT there is no separate specification language, so these callback functions have to be stored with a much stronger type, so that it is possible to infer from their type that they are "good" callback functions when they are retrieved from the heap.[4]

Ynot is based on the predicative version of HTT(13) in which dependent sums are predicative, i.e., for $\Sigma x : A.B$ to be a monotype, both A and B have to be monotypes. Since the type of heaps in Ynot is defined as a subset of the type $\mathbb{N} \times \Sigma\alpha : \mathbf{mono}.\alpha$ and $\mathbf{mono}$ is not a monotype, it follows that the T type above is not a monotype either and that values of type T cannot be stored in the heap. It is thus unclear whether it is possible to give an implementation of the above type in Ynot; the obvious attempt leads to a universe inconsistency error in Coq, reflecting the predicativity issues just discussed.

The impredicative version of HTT (19) has an impredicative sum type, $\Sigma^T x : A.B$, which is a monotype if B is. Hence, in the impredicative version of HTT, we can store values of type T, by using impredicative sums. We conjecture that the implementation derived from translating the Idealized ML implementation has the above type in impredicative HTT.

7. Related Work

The proof system is a synthesis of O'Hearn and Reynolds's (24) work on separation logic, with Reynolds's system of specification logic (23) for Algol, which introduced the idea of turning Hoare triples into the atomic formulas of a program logic. Birkedal, Biering, and Torp-Smith (3) first extended separation logic to higher-order.

Parkinson developed a version of separation logic for Java in his doctoral dissertation (15). His logic does not have a notion of implications over specifications, instead using behavioral subtyping to determine what specification dynamically dispatched method calls could have. Parkinson and Bierman have also introduced a notion of abstract predicate family (17) related to the higher-order quantification of Birkedal *et al*.

[2] The Coq script was 780 lines long.

[3] The Coq script was 2122 lines long.

[4] In the technical report (26) we discuss an alternative translation into HTT based on the idea that implications between specifications in Idealized ML should be translated into function types in HTT, but that leads to an implementation that does not capture subject-observer pattern because it essentially results in a functional implementation.

In addition to systems based on separation, there is also a line of research based on the concept of object invariants and ownership. The Java modeling language (JML) (9) and the Boogie methodology (1) are two of the most prominent systems based on this research stream. In Boogie, each object tracks its owner object with a ghost field, and the ownership discipline enforces that the heap have a tree structure. This allows the calculation of frame properties without explosions due to aliasing, even though the specification language remains ordinary first-order logic.

In his dissertation, Parkinson gave as an example a simple iterator protocol, lacking the integration with composites we have exhibited. Subsequently, we formalized a similar account of iterators (7), again lacking the integration with composites. Jacobs, Meijer, Piessens and Schulte (5) extend Boogie with new rules for the coroutine constructs C# uses to define iterators. Their solution typifies the difficulties ownership-based approaches face with iterators, which arise from the fact that iterators must have access to the private state of a collection but may have differing lifetimes. This work builds on Barnett and Naumann's generalization of ownership to friendship (2), which allows object invariants to have some dependency on non-owned objects.

The subject-observer pattern has been the focus of a great deal of effort, given its prominence in important applications. Simultaneously with our own initial formulation, Parkinson gave an example of verifying the subject-observer protocol (16). Recently, Parkinson and Distefano (18) have implemented a tool to verify these programs, and have demonstrated several examples including a verification of a subject-observer pattern specified along these lines. The tool includes automatic generation of loop invariants.

The work of Barnett and Naumann is also capable of reasoning about the subject-observer pattern, but only if all of the possible observers are known at verification. Leino and Schulte (10) made use of Liskov and Wing's concept of history invariants or monotonic predicates (11) to give a more modular solution. More recently, Shaner, Naumann and Leavens (25) gave a "gray-box" treatment of the subject-observer pattern. Instead of tracking the specifications of the observers in the predicate, they give a model program that should approximates the behavior of any actual notification method.

Pierik, Clarke and de Boer (22) formalize another extension to the Boogie framework which they name *creation guards*, specifically to handle flyweights. They consider flyweights an instance of a case where object invariants can be invalidated by the allocation of new objects, and add guards to their specifications to control allocation to the permitted cases.

Acknowledgments

This work was supported in part by NSF grant CCF-0541021, NSF grant CCF-0546550, DARPA contract HR00110710019, and the United States Department of Defense.

References

[1] M. Barnett, K. Leino, and W. Schulte. The Spec# Programming System: An Overview. *Proceedings CASSIS 2004*, 2005.

[2] M. Barnett and D. Naumann. Friends Need a Bit More: Maintaining Invariants Over Shared State. *Mathematics of Program Construction (MPC)*, 2004.

[3] B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines, higher-order separation logic, and abstraction. *ACM TOPLAS*, 29(5):24, 2007.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

[5] B. Jacobs, E. Meijer, F. Piessens, and W. Schulte. Iterators revisited: Proof rules and implementation. *Proceedings FTfJP*, 2005.

[6] N. Kokholm. An extended library of collection classes for .NET. Master's thesis, IT University of Copenhagen, Copenhagen, Denmark, 2004.

[7] N. R. Krishnaswami. Reasoning about iterators with separation logic. In *SAVCBS '06*, pages 83–86, New York, NY, USA, 2006. ACM.

[8] N. R. Krishnaswami. The semantics of higher-order separation logic for a higher-order language. Technical report, Carnegie Mellon University, 2008.

[9] G. Leavens, A. Baker, and C. Ruby. JML: a Java modeling language. *Formal Underpinnings of Java Workshop (at OOPSLA'98)*, 1998.

[10] K. R. M. Leino and W. Schulte. Using history invariants to verify observers. In *European Symposium on Programming (ESOP)*, pages 80–94. Springer, 2007.

[11] B. H. Liskov and J. M. Wing. Behavioural subtyping using invariants and constraints. In *Formal Methods for Distributed Processing: a Survey of Object-Oriented Approaches*, pages 254–280. Cambridge University Press, New York, NY, USA, 2001.

[12] A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract Predicates and Mutable ADTs in Hoare Type Theory. In *In Proceedings of European Symposium on Programming'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 189–204. Springer, 2007.

[13] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare Type Theory. In *In Proceedings of International Conference on Functional Programming'06*, pages 62–73, Portland, Oregon, 2006.

[14] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Reasoning with the awkward squad. In *In Proceedings of International Conference on Functional Programming'08*, 2008.

[15] M. Parkinson. *Local Reasoning for Java*. PhD in Computer Science, University of Cambridge, August 2005.

[16] M. Parkinson. Class invariants: The end of the road. *Proceedings IWACO*, 2007.

[17] M. Parkinson and G. Bierman. Separation logic and abstraction. *SIGPLAN Not.*, 40(1):247–258, 2005.

[18] M. Parkinson and D. Distefano. jstar: Towards practical verification for java. In *OOPSLA*, 2008, to appear.

[19] R. L. Petersen, L. Birkedal, A. Nanevski, and G. Morrisett. A realizability model for impredicative Hoare Type Theory. In *In Proceedings of European Symposium on Programming'08*, 2008.

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

[21] B. C. Pierce and D. N. Turner. Statically typed friendly functions via partially abstract types. Technical Report ECS-LFCS-93-256, University of Edinburgh, LFCS, Apr. 1993. Get by anonymous ftp from `ftp.dcs.ed.ac.uk` in `pub/bcp/friendly.ps.Z`. Also available as INRIA-Rocquencourt Rapport de Recherche No. 1899.

[22] C. Pierik, D. Clarke, and F. de Boer. Creational Invariants. *Proceedings FTfJP*, 2004.

[23] J. C. Reynolds. An introduction to specification logic. In *Proceedings of the Carnegie Mellon Workshop on Logic of Programs*, page 442, London, UK, 1984. Springer-Verlag.

[24] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS 2002)*, pages 55–74. IEEE Computer Society, 2002.

[25] S. M. Shaner, G. T. Leavens, and D. A. Naumann. Modular verification of higher-order methods with mandatory calls specified by model programs. In R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., editors, *OOPSLA*, pages 351–368. ACM, 2007.

[26] K. Svendsen, A. Buisse, and L. Birkedal. Verifying design patterns in Hoare Type Theory. Technical Report ITU-TR-2008-112, IT University of Copenhagen, sep 2008.