Modular Verification of the Subject-Observer Pattern via Higher-Order Separation Logic

Neelakantan R. Krishnaswami, Jonathan Aldrich[1], and Lars Birkedal[2]

[1] Carnegie Mellon University, {neelk, aldrich}@cs.cmu.edu
[2] IT University of Copenhagen, birkedal@itu.dk

Abstract. The *subject-observer* design pattern is a very common idiom in object-oriented systems; for example, it is an essential part of the model-view-controller pattern for programming graphical user interfaces. We give a modular proof technique using separation logic to verify this pattern. This proof method is modular in the sense that subjects and observers can be verified independently, and both can be verified independently of client code that calls both.

1 Introduction

The subject-observer design pattern[6] is a ubiquitous design pattern in object oriented programs. The subject is a data structure which changes over time, and the observers are objects whose own invariants depend on the state of the subject. To remain in sync with the subject, the observers pass individual notification methods to the subject, which the subject will call every time it changes.

The subject-observer pattern presents a lovely challenge to verification, because it is a design pattern which is fundamentally reliant on implicit communication through state. Observe that every observer can potentially have a *different* invariant, and that the set of notification methods in the subject changes as new observers register themselves.

Our contribution is:

- We give a simple, modular specification of the subject-observer pattern in separation logic, and then show how this permits the modular verification of an implementation and clients. The specification is written in a higher-order variant of separation logic, and supports a strong form of information hiding between the subject and the client.

2 The Programming Language

The core programming language we have formalized is an extension of the simply-typed lambda calculus with a monadic type constructor[11] to isolate access to the heap. In addition to function types $\tau \to \tau$, we also have pair types

$$
\begin{array}{lll}
\text{Types} & \tau & ::= 1 \mid \tau \times \tau \mid \tau \to \tau \mid \mathbb{N} \mid \mathsf{list}\ \tau \mid \bigcirc \tau \mid \mathsf{ref}\ \tau \\
\text{Expressions} & e & ::= () \mid \langle e, e \rangle \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid n \mid \mathsf{nil} \mid \mathsf{cons}(e, e) \mid l_\tau \\
 & & \quad \mid x \mid e\ e' \mid \lambda x : \tau.\ e \mid [c] \\
\text{Computations} & c & ::= e \mid \mathsf{new}_\tau\ e \mid\ !e \mid e := e \mid \mathsf{let}\ x = e;\ c \mid \mathsf{fix}_\tau x.\ c \\
 & & \quad \mid \mathsf{case}(e, \mathsf{nil} \to c \mid \mathsf{cons}(h, t) \to c) \\
\text{Contexts} & \Gamma & ::= \cdot \mid \Gamma, x : \tau
\end{array}
$$

Fig. 1. Programming Language Syntax

$\tau \times \tau'$, integers $\mathbb{N}$, references $\mathsf{ref}\ \tau$, mutable[3] lists $\mathsf{list}\ \tau$, and the type of monadic computations $\bigcirc\tau$.

The terms inhabiting these types are all standard, and listed in Figure 1, and are typed with a judgement $\Gamma \vdash e : \tau$, which is read as "In variable context Γ, the pure expression e has type τ". Some of the more interesting typing rules are listed in figure 2.

The values of the monadic type $\bigcirc\tau$ are lazy, suspended computations $[c]$. These are not evaluated during the evaluation of pure terms, which allows us to separate the pure and effectful parts of reduction. They are typed using a judgement $\Gamma \vdash c \div \tau$, which is read "In variable context Γ, the effectful computation c produces a value of type τ".

The computations include treating a pure term e as a computation, assignment $e_1 := e_2$, dereference $!e$, allocation $\mathsf{new}_\tau\ e$, general recursion $\mathsf{fix}_\tau x.\ c$, and sequential composition $\mathsf{let}\ x = e;\ c$. Notice that each of the primitive computations (assignment, allocations, and dereference) take pure expressions as arguments, so the order of all side-effects is explicit. The sequential composition is the monadic bind, and it has two additional purposes – it eliminates the suspended monadic values, and binds the result of executing the first command to a variable. So the computation $\mathsf{let}\ x = e;\ c$ can be read as "evaluate the expression e to a suspended computation $[c']$, evaluate c' and bind its result to x, and then execute c".

So, for example, an imperative factorial function might be written as:

$$
\begin{array}{l}
fact : \mathbb{N} \to \bigcirc N \equiv \lambda x : \mathbb{N}.\ [\mathsf{let}\ count = [\mathsf{new}_\mathbb{N}\ x]; \\
\qquad\qquad\qquad\qquad \mathsf{let}\ acc = [\mathsf{new}_\mathbb{N}\ 1]; \\
\qquad\qquad\qquad\qquad \mathsf{fix}\ loop. \\
\qquad\qquad\qquad\qquad\quad \mathsf{let}\ n = [!count]; \\
\qquad\qquad\qquad\qquad\quad \mathsf{if}\ n = 0\ \mathsf{then} \\
\qquad\qquad\qquad\qquad\qquad !acc; \\
\qquad\qquad\qquad\qquad\quad \mathsf{else} \\
\qquad\qquad\qquad\qquad\qquad \mathsf{let}\ old = [!acc]; \\
\qquad\qquad\qquad\qquad\qquad \mathsf{let}\ () = [acc := old * n];
\end{array}
$$

[3] Notice that the tail of a list is an updatable reference, unlike ML or Haskell linked lists.

$$\begin{array}{l}\mathsf{let}\ () = [count := n - 1];\\ \mathsf{let}\ ans = loop;\\ ans]\end{array}$$

Notice that all of the side effecting operations are explicitly sequenced – we do not permit compound expressions such as $acc :=!acc * n$, whose result could vary depending on the order of evaluation. This is because we intend this particular language as a core calculus to experiment with verification of higher-order programs with general references, and so are willing to accept infelicities in the language in exchange for a simple core.

$$\frac{}{\Gamma \vdash \mathsf{nil} : \mathsf{list}\ \tau}\ \textsc{TNil} \qquad \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash t : \mathsf{ref\ list}\ \tau}{\Gamma \vdash \mathsf{cons}(e, t) : \mathsf{list}\ \tau}\ \textsc{TCons}$$

$$\frac{\Gamma \vdash c \div \tau}{\Gamma \vdash [c] : \bigcirc\tau}\ \textsc{TMonad}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e \div \tau}\ \textsc{TPure} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{new}_\tau\ e \div \mathsf{ref}\ \tau}\ \textsc{TAlloc} \qquad \frac{\Gamma \vdash e : \mathsf{ref}\ \tau}{\Gamma \vdash !e \div \tau}\ \textsc{TDeref}$$

$$\frac{\Gamma \vdash e : \mathsf{ref}\ \tau \qquad \Gamma \vdash e' : \tau}{\Gamma \vdash e := e' \div 1}\ \textsc{TAssign} \qquad \frac{\Gamma, x : \bigcirc\tau \vdash c \div \tau}{\Gamma \vdash \mathsf{fix}_\tau x.\ c \div \tau}\ \textsc{TFix}$$

$$\frac{\Gamma \vdash e : \bigcirc\tau' \qquad \Gamma, x : \tau' \vdash c \div \tau}{\Gamma \vdash \mathsf{let}\ x = e;\ c \div \tau}\ \textsc{TLet}$$

$$\frac{\Gamma \vdash e : \mathsf{list}\ \tau' \qquad \Gamma \vdash c_n \div \tau \qquad \Gamma, h : \tau', t : \mathsf{ref\ list}\ \tau' \vdash c_c \div \tau}{\Gamma \vdash \mathsf{case}(e, \mathsf{nil} \to c_n \mid \mathsf{cons}(h, t) \to c_c) \div \tau}\ \textsc{TCase}$$

Fig. 2. Selected Typing Rules

3 Specifications with Separation Logic

The atomic form of a specification is a Hoare triple $\{P\}c\{a : \tau.\ Q\}$. This triple can be read as "Suppose we begin with a heap described by state P. If the execution of c terminates, the heap will be in a state described by Q, with a bound to the return value of the computation."

The pre- and post-states are described using separation logic[17], which is an extension of Hoare logic[7] originally proposed by Reynolds and O'Hearn to simplify reasoning about aliasing. The basic idea is to extend logic with two additional connectives, the separating conjunction $p * q$ and the separating implication $p -\!\!* q$ (usually called the "magic wand"). The proposition $p * q$ is read

as "one part of the heap is described by the state p, and a separate disjoint part is described by the state q". This contrasts with the reading of the ordinary conjunction $p \wedge q$ as "the heap is described by the predicate p, and also described the predicate q".

$$
\begin{array}{lll}
\text{Assertion Sorts } \omega & ::= & \text{prop} \mid \tau \mid \omega \Rightarrow \omega \mid \omega \times \omega \mid \mathsf{seq}\ \omega \\
\text{Assertions} & p, q ::= & \top \mid p \wedge q \mid p \supset q \mid \bot \mid p \vee q \mid \mathsf{emp} \mid p * q \mid p -\!\!* q \\
 & & \mid\ e \mapsto e \mid p = q \mid \forall x : \omega.\ p \mid \exists x : \omega.\ p \\
 & & \mid\ x \mid p\ q \mid \lambda x : \omega.\ p \mid (p, q) \mid \pi_1\ p \mid \pi_2\ p \mid \epsilon \mid p \cdot q \\
\text{Specifications} & S & ::= \{p\}c\{a : \tau.\ q\} \mid S\ \mathsf{implies}\ S' \mid S\ \mathsf{and}\ S' \mid S\ \mathsf{or}\ S' \\
 & & \mid\ \forall x : \omega.\ S \mid \exists x : \omega.\ S
\end{array}
$$

Fig. 3. Syntax of Assertions and Specifications

We extend separation logic to higher-order, following Biering *et. al.* [3]. That is, the quantifiers are not just permitted to range over program values, but also over arbitrary predicates. Doing so allows us to model information hiding – we can abstract over the heap, and thereby allow programs to manipulate heap through methods without revealing the precise representation of a data structure to a client program. The terms of our higher order logic are given in Figure 3; the assertion sorts ω give the sorts quantifiers can range over, and include terms of program types τ, propositions prop, and pairs, sequences and function spaces (used to represent predicates) over these base sorts.

While pure Hoare triples are sufficient to specify purely imperative programs, we must extend the specification language to handle features such as procedures. We do this by adapting the logic in the style of Reynolds's specification logic [15], which turns Hoare triples into the atomic propositions of a multi-sorted first-order logic. The sorts are the sorts of the assertion language, and the logical operators of this logic are the usual connectives of intuitionistic first-order logic. The special features of our specification language are merely axioms, given in Figure 4. We have rules corresponding to the "small footprint" rules of separation logic, plus induction axioms and a frame rule.

As a simple example, consider this signature for a counter module:

$$
\begin{array}{l}
\exists \mathit{counter} : \tau_c \times \mathbb{N} \Rightarrow \mathit{prop} \\
\exists \mathsf{create} : () \rightarrow \bigcirc \tau_c \\
\exists \mathsf{next} : \tau_c \rightarrow \bigcirc \mathbb{N} \\
\quad \{\mathsf{emp}\}\mathsf{create}()\{a : \tau_c.\ \mathit{counter}(a, 0)\} \\
\quad \mathsf{and} \\
\quad \{\mathit{counter}(c, n)\}\mathsf{next}(c)\{a : \mathbb{N}..\ a = n \wedge \mathit{counter}(c, n + 1)\}
\end{array}
$$

We assert the existence of a *counter* heap predicate, and give two functions create and next, whose behavior is defined in terms of this predicate. The exis-

tential quantifier over the predicate encapsulates the counter's state, rendering it opaque to clients – they do not know what the internal heap representation might be. Likewise, the implementations of the create and next functions are hidden, since they are the witnesses to the existential quantifiers. One way of thinking about this is that in our specification methodology, a program is part of the evidence that a specification can be met. The only information about them that a client can use are the triples describing their behavior, which are grouped together with a specification-level conjunction.

A client program that uses the counter spec will be verified against a specification $S_{counter}$ implies S, where $S_{counter}$ is the specification for the counter module. This lets us verify the client and the implementation separately, and we can combine the specifications for the client and the implementation with the rule of modus ponens.

Pure	$\{\mathsf{emp}\}e\{a : \tau.\ a = e \wedge \mathsf{emp}\}$
New	$\{\mathsf{emp}\}\mathsf{new}_\tau\ e\{a : \mathsf{ref}\ \tau.\ a \mapsto e\}$
Assign	$\{\exists x : \tau.\ e \mapsto x\}e := e'\{a : 1.\ e \mapsto e'\}$
Deref	$\{e \mapsto e'\}!e\{a : \tau.\ a = e' \wedge e \mapsto e'\}$
Bind	$[\langle P\rangle e\langle x : \tau.\ Q\rangle\ \mathsf{and}\ \{Q\}c\{a : \tau.\ R\}]\ \mathsf{implies}\ \{P\}\mathsf{let}\ x = e;\ c\{a : \tau.\ R\}$
Fix	$(\forall x : \bigcirc\tau.\ \{P\}x\{a : \tau.\ Q\}\ \mathsf{implies}\ \{P\}c\{a : \tau.\ Q\})\ \mathsf{implies}\ \{P\}\mathsf{fix}_\tau x.\ c\{a : \tau.\ Q\}$
Induction	$[S(0)\ \mathsf{and}\ \forall x : \mathbb{N}.\ (S(x)\ \mathsf{implies}\ S(x+1))]\ \mathsf{implies}\ \forall x : \mathbb{N}.\ S(x)$
Sequence Ind.	$[S(\epsilon)\ \mathsf{and}\ \forall x, xs.\ (S(xs)\ \mathsf{implies}\ S(x \cdot xs))]\ \mathsf{implies}\ \forall xs : \mathsf{seq}\ \omega.\ S(xs)$
Frame	$\{P\}c\{a : \tau.\ Q\}\ \mathsf{implies}\ \{P * R\}c\{a : \tau.\ Q * R\}$
Consequence	$\{P'\}c\{a : \tau.\ Q'\}\ \mathsf{implies}\ \{P\}c\{a : \tau.\ Q\},$ when $P \models P'$ and $Q' \models Q$

Fig. 4. Axioms for Specification Logic

4 The Subject Observer Pattern

4.1 Specification

In Figure 5, we give the specification of a subject observer pattern in which the observers synchronize on an integer-valued property of the subject. The intuitive, informal reading of this specification is "There exists an implementation of the subject (lines 1-3), such that if we have a notify method which updates

$\tau_s \equiv \mathsf{ref}\ \mathbb{N} \times \mathsf{ref}\ \mathsf{list}\ (\mathbb{N} \to \bigcirc 1)$

```
1   ∃ sub : τ_s × ℕ × seq ((ℕ → ◯1) × (ℕ ⇒ prop)) ⇒ prop.
2   ∃ register : τ_s × (ℕ → ◯1) → ◯1.
3   ∃ broadcast : τ_s × ℕ → ◯1.
4   ∀ obs : ℕ ⇒ prop.
5   ∀ notify : ℕ → ◯1.
6     ∀m, n. {obs(m)}
7            notify(n)
8            {a : 1. obs(n)}
9     implies
10    ∀s, n, os. {sub(s, n, os)}
11               register(s, notify)
12               {a : 1. sub(s, n, (notify, obs) · os)}
13    and
14    ∀s, m, n. {sub(s, m, ϵ)}
15               broadcast(s, n)
16               {sub(s, n, ϵ)}
17    and
18    [∀s, m, n, os. {sub(s, n, os) * observers(os)}
19                   broadcast(s, n)
20                   {sub(s, n, os) * observers_at(n, os)}
21     implies
22                   {sub(s, n, (notify, obs) · os) * observers((notify, obs) · os)}
23                   broadcast(s, n)
24                   {a : 1. sub(s, n, (notify, obs) · os) * observers_at(n, (notify, obs) · os)}]
```

The helper predicates *observers* and *observers_at* are defined as follows:

$$
\begin{array}{ll}
observers(\epsilon) & = \mathsf{emp} \\
observers((f, o) \cdot os) & = (\exists n : \mathbb{N}.\ o(n)) * observers(os) \\
 & \\
observers\_at(n, \epsilon) & = \mathsf{emp} \\
observers\_at(n, (f, o) \cdot os) & = o(n) * observers\_at(n, os)
\end{array}
$$

Fig. 5. Simple Subject-Observer Specification

its observer's state (lines 4-8), then we may attach notify to the subject (lines 10-12) and the broadcast will update all of the observers (lines 14-24)".

Line 1 of the specification requires the existence of a *sub* predicate describing the mutable state of the subject, and lines 2 and 3 requires the existence of appropriately-typed register and broadcast methods.

The *sub* predicate is a three-place higher-order predicate. The first argument, of type τ_s, links a term in the programming language with its associated storage.[4] The second argument, of type $\mathbb{N}$, is the abstract state of the subject. This corresponds to model fields in notations such as JML[5]; the concrete representation of the subject might or might not contain an integer. (In our case, for simplicity's sake it does.) The third argument is the higher-order term, and tracks the relationship between the observers and the subject. It consists of a sequence of the notification methods that have been passed to the subject, paired with the predicates that each method updates.

The rest of the specification is written as an implication. This is because we wish to constrain the notify methods passed to register. Since we want to allow any suitable notification method, we universally quantify over the *obs* predicate – representing the observer's state – and the notify method that modifies it in lines 4 and 5, and then in lines 6-8 we assert that the notify method updates the observer state *obs* appropriately.

Given this hypothetical, we can specify the behavior of the register method in lines 10-12, where we say that attaching a notify method satisfying the assumption will result in the state of the subject changing to reflect the fact that the subject will update this observer. (As an aside, note that this function is written in a functional style; the appropriate notification function is passed to the subject as a higher-order function, which allows us to avoid having to grant the subject a reference to the observer.)

Then, in lines 14-24, we can assert that the broadcast action will update all of the observers. We use two auxilliary assertion-level functions, $observers$ and $observers\_at$. The $observers(os)$ function asserts that for each observer in os, we have its associated state separately conjoined with the rest. We existentially quantify each observer's state, to reflect the fact that their states do not have to be consistent with each other or the subjects prior to the method call. The $observers\_at(n, os)$ says that every observer is in the state n. [5]

The specification of the broadcast function is given in two parts. The first part, in lines 14-16, asserts that broadcast works correctly when it is used with an empty sequence of observers. The second part, in lines 18-24, is more complex. It asserts that if broadcast works correctly on sequence of observers *os* (lines 18-20), then it will continue to work correctly when the observers are extended with notify. This enables us to incrementally build up the specification of broadcast, as we add observers in the program source.

[4] τ_s is an abbreviation for the concrete type ref $\tau \times$ ref list $(\mathbb{N} \to \bigcirc 1)$ from our type syntax, but for simplicity we write it as a parameter here. True System F-style quantification over types in programs and specifications remains future work.

[5] Since our assertion language is a higher-order logic, they are actually definitions within the logic, and not new extensions to it.

Satisfyingly, the use of the separating conjunction automatically makes it impossible to call broadcast if an observer has been added to the subject multiple times – the definition of the *observers* and *observers_at* functions means that the state associated with each entry in the subject's observer list must be disjoint from all of the other observers. If an observer has been added multiple times, this condition is impossible to satisfy and verification will rightly be blocked. It would also be possible to change the precondition of register to require $sub(s, n, os) * observers(os) * \exists x.\ obs(x)$, which would ensure that we could not register an observer twice in the first place.

4.2 Subject Implementation

As mentioned earlier, a module correponds to an existential type in the specification, and the implementations are the witnesses to that existential. In Figure 6, we give an example of some predicates and programs that will satisfy the invariants of the spec in Figure 5, consisting of the concrete predicate for *sub*, and implementations of register and broadcast.

The verification of these procedures is straightforward; we instantiate the *sub* predicate with our definition, and then verify each of the bodies of register and broadcast. Because notify is universally quantified, we can make no assumptions about the implementation of the observer – in fact, the subject doesn't even know the *type* of the observer, because we pass in the notification as a higher-order function that encapsulates all of the appropriate update behavior. Then the verification merely needs to ensure that register adds an element to the linked list of observers, and that notify calls all of the notification functions.

4.3 A Client Program

In Figure 7, we give an example of how to verify a client program that uses this specification. In lines 1-4, we give some function definitions and predicates which we will use later on in the verification of the client, corresponding to (in lines 1-2) an observer method that will track the value of the subject and its invariant, and (in lines 3-4) an observer method that will track twice the value of the subject and its associated invariant. It should be clear that for any given r, we can apply $f_1\ r$ to get a notification action that closes over r, and likewise for the invariant and f_2.

Now, subsequently we will assume that the existential package for the subject has been unpacked, and we have bindings for *sub*, register and broadcast in our context.

Now, in line 5, we begin by assuming that we have a subject s state, with no observers attached. Then, in lines 6-9, we allocate 2 new references r_1 and r_2. Now, recall that the assertion $r_1 \mapsto n$ is equivalent to $o_1\ r_1$, so we can register $f_1\ r_1$ in line 10. This updates the state of the subject predicate, and we can use the specification of the broadcast along with the fact it works with an empty list of observers to deduce that it will work when the *sub* predicate is extended with $(f_1\ r_1, o_1\ r_1)$. In lines 12 and 13, we repeat the process with f_2, o_2 and r_2.

```
1   τ_s ≡ ref ℕ × ref list (ℕ → ◯1)

2   listof : (ref list τ) × seq τ ⇒ prop.
3   listof(c, ϵ)       = c ↦ nil
4   listof(c, x · xs)= ∃c'. (c ↦ cons(x, c')) ∗ listof(c', xs)

5   sub : τ_s × ℕ × seq ((ℕ → ◯1) × (ℕ ⇒ prop)) ⇒ prop.
6   sub(s, n, os)     = (fst s ↦ n) ∗ listof(snd s, map π_1 os)

7   register(s, notify) ≡
8             let cell = [!(snd s)];
9             let tail = [new_{list τ} cell];
10            snd s := cons(notify, tail)


11  broadcast(s, n) ≡
12            let r = [new_{ref list (ℕ→◯1)} (snd s)];
13            fix_1 loop.
14                 let list = [!r];
15                 let cell = [!list];
16                 case (cell,
17                       | nil → fst s := n,
18                       | cons(notify, tail) →let () = [r := tail];
19                                             let () = notify(n);
20                                             let () = loop;
21                                              ())
```

Fig. 6. Subject-Observer Implementation

In line 14, we use the beta-equality to change all the explicit points-to a call to o_1 and o_2, and in line 15 we existentially quantify over their contents. This lets us invoke the specification of broadcast(5) in line 16, which in line 17 shows that both observer predicates have now been given the value 5. In line 18, we expand out the definitions again, and see that r_1 points to 5, and r_2 points to 10, just as we would expect.

1 $f_1 : \mathsf{ref}\ \mathbb{N} \to \mathbb{N} \to \bigcirc 1 \quad \equiv \lambda r : \mathsf{ref}\ \mathbb{N}.\ \lambda n : \mathbb{N}.\ [r := n]$
2 $o_1 : \mathsf{ref}\ \mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathrm{prop} \quad \equiv \lambda r : \mathsf{ref}\ \mathbb{N}.\ \lambda n : \mathbb{N}.\ r \mapsto n$
3 $f_2 : \mathsf{ref}\ \mathbb{N} \to \mathbb{N} \to \bigcirc 1 \quad \equiv \lambda r : \mathsf{ref}\ \mathbb{N}.\ \lambda n : \mathbb{N}.\ [r := n \times 2]$
4 $o_2 : \mathsf{ref}\ \mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathrm{prop} \quad \equiv \lambda r : \mathsf{ref}\ \mathbb{N}.\ \lambda n : \mathbb{N}.\ r \mapsto (n \times 2)$

5 $\{sub(s, n, \epsilon)\}$
6 $\mathsf{let}\ r_1 = \mathsf{new}_{\mathbb{N}}\ 0;$
7 $\{sub(s, n, \epsilon) * (r_1 \mapsto 0)\}$
8 $\mathsf{let}\ r_2 = \mathsf{new}_{\mathbb{N}}\ 0;$
9 $\{sub(s, n, \epsilon) * (r_1 \mapsto 0) * (r_2 \mapsto 0)\}$
10 $\mathsf{let}\ () = \mathsf{register}(s, f_1\ r_1);$
11 $\{sub(s, n, (f_1\ r_1, o_1\ r_1) \cdot \epsilon) * (r_1 \mapsto 0)\}$
Deduce extended spec for broadcast.
12 $\mathsf{let}\ () = \mathsf{register}(f_2\ r_2);$
13 $\{sub(s, n, (f_2\ r_2, o_2\ r_2) \cdot (f_1\ r_1, o_1\ r_1) \cdot \epsilon) * (r_1 \mapsto 0) * (r_2 \mapsto 0)\}$
Deduce extended spec for broadcast.
14 $\{sub(s, n, (f_2\ r_2, o_2\ r_2) \cdot (f_1\ r_1, o_1\ r_1) \cdot \epsilon) * (o_1\ r_1\ 0) * (o_2\ r_2\ 0)\}$
15 $\{sub(s, n, (f_2\ r_2, o_2\ r_2) \cdot (f_1\ r_1, o_1\ r_1) \cdot \epsilon) * (\exists x : \mathbb{N}.\ o_1\ r_1\ x) * (\exists y : \mathbb{N}.\ o_2\ r_2\ y)\}$
16 $\mathsf{let}\ () = \mathsf{broadcast}(5);$
17 $\{sub(s, 5, (f_2\ r_2, o_2\ r_2) \cdot (f_1\ r_1, o_1\ r_1) \cdot \epsilon) * (o_1\ r_1\ 5) * (o_2\ r_2\ 5)\}$
18 $\{sub(s, 5, (f_2\ r_2, o_2\ r_2) \cdot (f_1\ r_1, o_1\ r_1) \cdot \epsilon) * (r_1 \mapsto 5) * (r_2 \mapsto 10)\}$

Fig. 7. Sample Client Program

5 Discussion

An interesting feature of our specification is that it enforces a form of information hiding – the implementation of the subject is hidden behind an existential quantifier, so clients and observers cannot depend on the particulars of the subject's implementation; they can only rely on the specified interface. Conversely, the use of a universal quantifier over the observers means that the verification of a subject implementation *cannot* rely on any particular observer's implementation – it must work for all possible observers, whose only common behavior is that mandated by the spec.

This is not the same as classical data abstraction[16], since the representation type τ_s is known to clients, but it has a similar flavor – we use abstract predicates to keep portions of the heap abstract to different parts of our program. We tend to describe this as the difference between abstraction and encapsulation.

Our specification currently forbids reentrant notification methods; for example, we cannot write a notification action which registers another observer within the notification. This is because our specification requires the notify method to be oblivious to the state of the *sub* predicate. Relaxing this restriction should be straightforward, at the price of complicating the invariant that must hold.

It should be clear that the specification methodology we have used here to specify subject-observers is generally applicable. Indeed, in other work [8], we have used it to specify abstractly the behavior of collections and iterators. In *loc. cit.* we used, in particular, an abstract formal specification (given as a tautology in separation logic) that iterators are valid only as long as the abstract state of the underlying collection is left unchanged (forbidding, e.g., adding or removing new elements, but permitting caching).

6 Related and Future Work

Separation logic was originally introduced by Reynolds and O'Hearn [17] to reason about low-level pointer programs. Biering, Birkedal and Torp-Smith [3] extended separation logic to higher order. Parkinson [13] has formalized a version of separation logic for a subset of Java. He and Bierman also developed a notion of *abstract predicate*[14], which was generalized which was generalized via higher-order separation logic in [4]. In [4] it was suggested that higher-order separation logic could be useful for data abstraction, and here we demonstrate that by our verification of the subject-observer pattern. Nanevski, Morisett, and Birkedal have proposed Hoare Type Theory[12], which is a dependently-typed functional programming language which uses a spatial logic equivalent to higher-order separation logic to describe monadic effects. Our work is very similar in scope, but strongly separates the type system and the specification language.

Barnett and Nauman[2] proposed a method for reasoning about the subject observer pattern, which works when all of the observer classes are known at verification time. Recently, Leino and Schulte[9] proposed adapting Liskov and Wing's monotonic constraints[10] to model the subject-observer pattern. This does not require all of the observers to be known to verify the subject, but each observer must maintain a pointer to the subject and respect its field updates. This contrasts with our verification approach, where the observers can be completely oblivious to the subject, and the connection between subject and observer is maintained in the subject's invariant. Both of these papers are in the context of the Boogie methodology[1], which was designed for OO languages and handles features such as subclassing, which we do not.

In the future, we plan on extending our core language to better model features found in object-oriented languages and languages like ML; in particular, we plan on adding impredicative polymorphism in the style of System F.

7 Acknowledgements

This work was supported in part by NSF grant CCF-0541021 and the Department of Defense.

References

1. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.
2. M. Barnett and D. Naumann. Friends need a bit more: Maintaining invariants over shared shate. In *Proceedings of Mathematics of Program Construction 2004*, 2004.
3. B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines and higher order separation logic. In *Proc. of ESOP 2005: The European Symposium on Programming*, pages 233–247, Edinburgh, Scotland, April 2005.
4. B. Biering, L. Birkedal, and N. Torp-Smith. Bi-hyperdoctrines, higher-order separation logic, and abstraction. *To appear in ACM Transactions on Programming Languages and Systems*, page 45, 2007.
5. G. Burdy, Y. Cheon, D. R. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, , and E. Poll. An overview of jml tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of reusable object-oriented software*. Addison-Wesley, 1995.
7. C. A. R. Hoare. An axiomatic approach to computer programming. *Communications of the ACM*, 12(583):576–580, 1969.
8. N. R. Krishnaswami. Reasoning about iterators with separation logic. In *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, pages 83–86, New York, NY, USA, 2006. ACM Press.
9. K. R. M. Leino and W. Schulte. Using history invariants to verify observers. In *ESOP 2007, Proceedings of the Sixteenth European Symposium on Programming*, 2007.
10. B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
11. E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
12. A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in hoare type theory. In *Proceedings of International Conference on Functional Programming 2006*, 2006. To Appear.
13. M. Parkinson. *Local Reasoning for Java*. PhD thesis, Cambridge University, 2005.
14. M. Parkinson and G. Bierman. Separation logic and abstraction. In *Proceedings of the 32nd Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL'05)*, pages 247–258, Long Beach, CA, USA, January 2005.
15. J. Reynolds. Idealized algol and its specification logic. In P. O'Hearn and R. Tennent, editors, *ALGOL like languages*, volume 1. Birkhäuser, 1997.
16. J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.
17. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 55–74, Copenhagen, Denmark, July 2002. IEEE Press.