Boogie Meets Regions: a Verification Experience Report (Extended Version)[*]

Anindya Banerjee[†]
Kansas State University
Manhattan KS 66506 USA
ab@cis.ksu.edu

Mike Barnett
Microsoft Research
Redmond WA 98052 USA
mbarnett@microsoft.com

David A. Naumann[‡]
Stevens Institute of Technology
Hoboken NJ 07030 USA
naumann@cs.stevens.edu

May 22, 2008

Abstract

We use region logic specifications to verify several programs exhbiting the classic hard problem for object-oriented systems: the framing of heap updates. We use BoogiePL and its associated SMT solver, Z3, to prove both implementations and client code.

1 Introduction

Many programs use dynamically allocated mutable storage, which poses challenges for encapsulation and exacerbates the frame problem: how to specify that "everything else is unchanged" for mutable state not directly named by program variables? Several lines of work approach this problem in terms of *footprints*, i.e., the sets of locations that are written or read by a phrase of a program or specification.

- In the Boogie methodology [4], "owned state" is represented by a mutable ghost field that points to an object's owner if it has one. An *effect specification* ("modifies" clause) that licenses update of an object o implicitly licenses update of the objects transitively owned by o.

- In Separation Logic [16], if P is the precondition of a procedure then the procedure may be viewed as owning the state on which P depends. It has license to modify that part of the heap but no other.

- Kassios [13] shows how to manipulate footprints as such, rather than implicitly (in formulas) or indirectly (via transitive ownership).

A key point is that if the read footprint of a formula or pure expression is disjoint from the write footprint of a command, then the command preserves the value of the formula or expression.

Kassios works in a higher order logic, which can directly express that a formula depends on certain locations. He showed how reasoning idioms developed in Boogie or separation logic can be elegantly and

---

effectively articulated using explicit footprints. His approach does not canonize such idioms as a methodology to be imposed on all parts of all programs. He develops a relational refinement calculus, following Hehner [11].

Smans et al [19] explore Kassios' approach in terms of conventional sequential specifications with distinct first order precondition, postcondition, and effect. They focus on framing of pure methods used in specifications for data abstraction. They report on encouraging case studies using a prototype verifier based on an SMT solver.

Banerjee et al [3] also explore the approach using first-order specifications, focusing on foundational justification in terms of a Hoare logic that uses footprint expressions in the effects clause. Their Region Logic features a frame rule inspired by that of separation logic (which in turn adapts Hoare's rule of Invariance) to encompass interference via mutable heap objects. In region logic, the frame rule involves a static analysis for read effects of formulas. The effect **rd** $G{\bullet}f$ expresses that field f of some objects in G may be read. A region expression G denotes a set of non-null object references and footprints are of the form $G{\bullet}f$. This treatment caters for notations like $G{\bullet}f \subseteq G'$ which says that the f-image of G is a subset of G'.[1] Region logic features the use of region fields and variables to encode dynamic frames whereas Smans *et al.* use pure methods. Banerjee *et al.* claim that: "A benefit of treating regions as ghost state is that it can be done using first-order specification languages based on classical logic with modest use of set theory. Thus it fits with mostly-automated tools based on verification condition generation...."

This paper reports on some case studies to investigate Banerjee *et al.*'s claim. We demonstrate the utility of using region specifications for

- local reasoning about data structures (Sect. 4),

- encapsulation, even in the presence of callbacks (Sect. 5),

- layered abstractions, using the examples from Smans et al. [19] but without the need for conditional effects (Sect. 6).

In addition, our experience supports their suggestion that the region logic provides a neutral framework in which disciplines such as ownership can be, but need not be imposed system-wide.

In ongoing work we are investigating decision procedures for region logic primitives such as $G{\bullet}f \subseteq G'$. Here, we use the Z3 solver [9] which does not have a decision procedure for set theory. This poses another research question: Is a first order axiomatization of set theory effective for automating verification of programs and specifications that encode footprints in ghost state? Our answer is yes (Sect. 9) and no (Sect. 7).

2   Relevant features of BoogiePL

Rather than implement a verification condition generator from scratch for our assertion language, we use the BoogiePL intermediate language advocated at VSTTE'05 by Barnett et al [5]. The BoogiePL tool [10] generates verification conditions suited for SMT solvers such as Simplify, the CVC family, and Z3 [9]. These integrate multiple decision procedures with heuristic instantiation of quantifiers driven by pragmas called *triggers*. They have been used with some success in verifiers like ESC/Java and Spec#, where user interaction is limited to the insertion of assert and assume statements and loop invariants. Our experiments consist entirely of manually written encodings in BoogiePL of the example programs and specifications,

---

[1]These features are useful for reasoning about simulations, as in the context of information flow [1, 2].

with Z3 as solver. We are therefore in a position to provide triggers on an ad hoc basis, but we do not tinker with the verifier.

A BoogiePL program is a collection of procedure specifications and implementations, together with global variables, uninterpreted functions, and axioms. The Boogie tool generates verification conditions for the procedure implementations with respect to their specifications. These conditions, together with the axioms, are translated into an input format for SMT solvers, in our case Z3. We do not explain that translation: the verification conditions represent the weakest precondition of the program [6]. Instead we focus on the features provided in BoogiePL and their use for encoding region specifications and programs.

Types. The type system is coarser than that of a high-level object-oriented language: objects are represented by the type **ref** (of which **null** is a distinguished element). We did not make any finer distinction. The other primitive types are **bool**, **int**, and **name**, all subtypes of **any**. We use the type **name** for fields. Another subtype of **any** is a type constructor for maps, using array-style notation. E.g., we encode regions as [**ref**]**bool**, denoting a boolean-valued function on references.

Heaps. The heap is modeled as an infinite two-dimensional array, mapping pairs of object references and field names to values, roughly type [**ref**, **name**]**any**. Logically, all objects "have" all fields, but of course, each object uses only those fields which are defined in the corresponding high-level program. There is no built-in notion of an object being allocated or not. We model it using an explicit boolean field, $alloc$, and assume that the field is set appropriately in the necessary places (see below).

We declare a single global variable, "$Heap$". All operations involving the heap are explicit: what would be written $o.f$ in source code is written as $Heap[o, f]$. We write "Hp" for the type of heaps. (It is not actually [**ref**, **name**]**any** but rather a refinement that uses a form of type dependency to distinguish integer fields from reference fields but we elide the details [10].) We write **field** for the type of fields, again hiding the dependency notation for field names.

Source programs only compute heaps h that are self-contained in the sense that for any allocated object o, and for any field f of o, if f's content is a reference, p, then p is also allocated in h. We use an uninterpreted predicate, GH, read "is good heap", and in our initial experiments an axiom was used saying that $GH(h)$ implies self-containment. It was not a defining axiom, as we expected for some purposes one would want additional conditions such as typing. It turned out that our results do not even need self-containment. There is also a predicate GO ("is good object") with a defining axiom that says $GO(o, h)$ iff $GH(h)$ and o is non-null and o is allocated (i.e., $h[o, alloc]$ is true).

Procedures and functions. Procedures in BoogiePL model the specification of imperative code (with optional implementations). Functions introduce arbitrary axiomatizations and model user-defined methods within specifications. All parameters are explicit, including the receiver of an instance method and the return value.

Procedures have *preconditions* and *postconditions* for encoding the assertions they demand of a caller and the guarantees that a caller can assume, respectively. *Free* preconditions and postconditions are not checked (asserted) but instead assumed. Such assumptions model facts that are guaranteed by the high-level programming language. Free preconditions represent facts that a callee can depend on, while free postconditions facts that a caller can depend on. For example, we use a free precondition that the receiver of an instance method is non-null and allocated and a free postcondition that all allocated objects remain so. In this paper we elide most of them.

Implementations are written using the usual high-level control structures of assignment statements, conditional statements, and while loops. Loops are allowed to have loop invariants. There are also assert statements, which introduce a proof obligation, and assume statments which introduce unchecked facts. For example, every assignment to the heap is followed by an assumption $GH(Heap)$ which is justified because the source language would make GH an all-states invariant. Finally, the statement **havoc** o loses all information about the value of the variable o.

When calls of (source code) methods are allowed in specifications, those methods must be *pure*, i.e., not modify any pre-existing state. All calls to a pure method, m, in specifications are encoded by an uninterpreted function, #m. The axioms defining #m are either derived from the specification for m [8] or else directly from the implementation of m [19].

Allocation. We encode the high-level source statement $o := $ **new** $T(\ldots)$ as

$$\textbf{havoc } o; \ \textbf{assume } \neg Heap[o, alloc]; \ \textbf{call } T..ctor(\ldots);$$

where $T..ctor$ is the procedure that encodes the appropriate constructor for the type T. In the interests of brevity, we use the high-level statement throughout. All constructors have a free postcondition that the object is a good object, hence allocated. That is, there is no explicit state change that updates the *alloc* field to allocate an object, but the effect of calling the constructor provides the same functionality. Constructors also have a free precondition that the object is allocated. The free precondition that o is allocated does not lead to a contradiction with the assumption $\neg Heap[o, alloc]$ because only *checked* preconditions become proof obligations (assertions) at call sites to the constructor. The result is that code within a constructor can use the object, e.g., as a parameter to a call.

The default write effect of any constructor is that only fields of the object under construction can be updated.

Quantifiers. Quantifiers are written in BoogiePL as $(Q\ v \bullet body)$ where Q is the kind of the quantifier, v is a list of bound variables and their types and $body$ is a predicate dependent on the variables in v. For axioms involving quantifiers, immediately following the $\bullet$ and before the quantifier body a *trigger* might appear. Triggers are syntactic patterns delimited by {...} and are used to provide hints to the SMT prover on how to instantiate the quantified variable. Triggers are discussed in more detail in Sect. 7.

Modifies clauses. BoogiePL specifications include a *modifies* clause, but this merely lists variables. Thus every mutator method lists $Heap$, and for brevity in this paper we elide it throughout. More important are the source level effect specifications. The high level specification of a method might include effect **wr** $\langle o \rangle \bullet f$, for a parameter o and field f (this region logic notation corresponds to just $o.f$ in the modifies clause of Spec# or JML). As usual, this translates to a postcondition of the form

$$(\forall p : \textbf{ref}, g : \textbf{field} \bullet GO(p, \textbf{old}(Heap)) \Longrightarrow$$
$$(\textbf{old}(Heap)[p, g] = Heap[p, g]) \vee (p = o \wedge g = f) \vee \ldots) \tag{1}$$

where the elided part comes from other effect specifications. There is no restriction on modifying any field of an object that is allocated during the method call.

3 Regions

Recall that regions, i.e., reference sets, are encoded by their characteristic functions using the type [**ref**]**bool**. Region logic features effects of the form **wr** $this.O\bullet f$ where O is a region type field and f a fieldname. (In our experiments, the only form of region expression used are field dereferences and variables.) The effect **wr** $this.O\bullet f$ is translated exactly as Equation 1, but replacing $p = o$ with the test whether p is member of **old**$(Heap)[this, O]$. Note that write effects are interpreted in the *pre* state of a method.

Although sets are represented by the interpreted type [**ref**]**bool**, this BoogiePL type comes equipped only with equality test and the select/update operations. Other set theoretic operations and predicates can be given straightforward defining axioms; we followed an axiomatization provided by Rustan Leino. Here are some of the declarations and defining axioms.

> **function** $SingletonSet(\textbf{ref})$ **returns** $([\textbf{ref}] \ \textbf{bool})$;
> **axiom** $(\forall r:\textbf{ref}, o:\textbf{ref} \bullet \ \{SingletonSet(r)[o]\} \ SingletonSet(r)[o] \iff r = o)$;
>
> **function** $DisjointSet([\textbf{ref}] \ \textbf{bool}, [\textbf{ref}] \ \textbf{bool})$ **returns** $(\textbf{bool})$;
> **axiom** $(\forall a:[\textbf{ref}] \ \textbf{bool}, b:[\textbf{ref}] \ \textbf{bool} \bullet \ \{DisjointSet(a,b)\}$
> $DisjointSet(a,b) \iff (\forall o:\textbf{ref} \bullet \ \{a[o]\}\{b[o]\} \ \neg a[o] \vee \neg b[o]) \ )$;

In the $DisjointSet$ axiom, either $a[o]$ or $b[o]$ can be triggers for the inner quantifier. In the sequel, we write $\varnothing$ for $EmptySet$, $\{o\}$ for $SingletonSet(o)$, $A \cup B$ for $UnionSet(A, B)$, $A = B$ for $EqualSet(A, B)$ to save space. To avoid clutter we omit triggers from most axioms.

Predicate $GR(G, h)$ is defined to say that every object in region G is a good object in h. As mentioned in the introduction, Region Logic features predicates involving the f-image of a region. For example, region G is closed with respect to field f (of type **ref**) in heap h provided that for every object o in G, the value of $o.f$ is either **null** or in G:

> **function** $RegionClosed(G:[\textbf{ref}]\textbf{bool}, h:Hp, f:\textbf{field})$ **returns** $(\textbf{bool})$;
> **axiom** $\forall G:[\textbf{ref}]\textbf{bool}, h:Hp, f:\textbf{field} \bullet$
> $RegionClosed(G,h,f) \iff (\forall o:\textbf{ref} \bullet \ G[o] \Longrightarrow h[o,f] = \textbf{null} \vee G[h[o,f]])$

Region G is *fresh* with respect to a pre heap provided its elements are unallocated in the pre heap:

> **function** $fresh(h:Hp, k:Hp, G:[\textbf{ref}]\textbf{bool})$ **returns** $(\textbf{bool})$;
> **axiom** $\forall h:Hp, k:Hp, G:[\textbf{ref}]\textbf{bool} \bullet$
> $fresh(h,k,G) \iff GH(h) \wedge GR(G,k) \wedge AllNewRegion(h,G)$

$AllNewRegion(h, G)$ asserts that all objects in region G are unallocated in heap h.
The predicate $DifferenceIsNew(h, G, G')$ holds provided G' is a newly allocated region with respect to both G and h: in other words if o is any object allocated in h, either o is **null** or o is an element of G or o is not an element of G'.

4 Local reasoning example: List Copy

Fig. 1 shows the body of a $ListCopy$ method that takes a linked list of nodes as input and produces as output a new list which is a copy of the original list. Our goal is to verify that if the original list is non-empty and resides in a region —say $origRg$— then the copied list is also non-empty and resides in a region, say,

```
procedure ListCopy(root : ref) returns (result : ref)
var newRoot, prev, tmp, oldListPtr : ref; newRg, origRg : [ref]bool
{ assume root = null ∨ (origRg[root] ∧ RegionClosed(origRg, Heap, nxt));
   newRoot := null; tmp := null; oldListPtr := root; newRg := ∅;
   if (oldListPtr ≠ null){
      newRoot := new Node(); newRg := newRg∪{newRoot};
      prev := newRoot; oldListPtr := Heap[oldListPtr, nxt];
      while (oldListPtr ≠ null){
         tmp := new Node(); newRg := newRg∪{tmp};
         Heap[prev, nxt] := tmp; assume GH(Heap);
         prev := tmp; oldListPtr := Heap[oldListPtr, nxt]; }}
   result := newRoot;
   assert (root = null ∧ result = null) ∨ (root ≠ null ∧ RegionClosed(newRg, Heap, nxt)
      ∧newRg[result] ∧ fresh(old(Heap), Heap, newRg) ∧ DisjointSet(newRg, origRg)); }
```

Figure 1: List Copy.

$newRg$, that is fresh and disjoint from $origRg$. Notice that we are not verifying full functional correctness, namely, that the new list is indeed a copy of the original. For this purpose, $origRg$ and $newRg$ would be auxiliary variables, scoped over both the requires and ensures clause. Such variables are supported by JML and Region Logic but not the current version of BoogiePL, so for illustration we make them local variables instead of parameters and use assert and assume statements to encode the specification.

At the beginning of every loop iteration, the variable $oldListPtr$ contains a pointer to the remainder of the original list that is yet to be copied. The variable $prev$ contains a pointer to the current end node of the new list: the new list grows when a new node pointed to by tmp is added at its end. All nodes in the copied list exist in $newRg$. This region is fresh with respect to the heap at the entry to the method. Freshness of each node in $newRg$ is ensured by the allocation of a new object (Sect. 2). The constructor call for $Node$ ensures that tmp is a good object, that is, clients calling the constructor are guaranteed that tmp is non-null and allocated. Since tmp is assumed to be unallocated before the constructor call, $\{tmp\}$ is a fresh region.

The loop invariant below is obtained from the postconditions except for last conjunct which should be mechanically generated for every loop translated from a high-level source program; ghost variable $loopPreHeap$ is the snapshot of the heap before the loop body is entered.

$$newRg[newRoot] \wedge RegionClosed(newRg, Heap, nxt)\wedge$$
$$fresh(old(Heap), Heap, newRg) \wedge NoDeallocs(loopPreHeap, Heap)$$

This suffices to prove $DisjointSet(newRg, origRg)$.

5   Encapsulation example: Subject/Observer

Fig. 2 shows a more involved example, *Subject/Observer*. A subject, s, has an internal state in field val and a pointer to a list of observers with typical element, o. The head of the list is reached via $s.obs$ and other observers in the list are reached following the observers' nxt fields. The entire list of observers resides in a region contained in (ghost) field O of s. The observer o maintains a pointer to its subject in its sub field and $o.cache$ contains o's current view of s's internal state.

// *Observer*: fields $sub, nxt : \textbf{ref}$, $cache : \textbf{int}$
// methods: *ctor* (constructor), *notify*
procedure $notify(this : \textbf{ref})\{$ **procedure** $ctor(this : \textbf{ref}, s : \textbf{ref})\{$
     var $tmp : \textbf{int}$; $Heap[this, sub] := s$; **assume** $GH(Heap)$;
     $tmp := \textbf{call } get(Heap[this, sub])$; $Heap[this, nxt] := \textbf{null}$; **assume** $GH(Heap)$;
     $Heap[this, cache] := tmp$; **assume** $GH(Heap); \}$ **call** $register(s, this); \}$

// *Subject*: fields $obs : \textbf{ref}$, $val : \textbf{int}$, $O : [\textbf{ref}]\textbf{bool}$
// methods: *ctor* (constructor), *register*, *add*, *update*, *get*
procedure $update(this : \textbf{ref}, n : \textbf{int})\{$
     var $o : \textbf{ref}$; **var** $r1 : [\textbf{ref}]\textbf{bool}$;
     $Heap[this, val] := n$; **assume** $GH(Heap)$; $o := Heap[this, obs]$; $r1 := \varnothing$;
     while $(o \neq \textbf{null})\{\textbf{call } notify(o)$; $r1 := r1 \cup \{o\}$; $o := Heap[o, nxt]; \}$

Figure 2: Subject/Observer implementation excerpts.

function $List(o : \textbf{ref}, h : Hp)$ **returns** $([\textbf{ref}]\textbf{bool})$;
axiom $\forall o : \textbf{ref}, h : Hp \bullet o = \textbf{null} \iff List(o, h) = \varnothing$
axiom $\forall o : \textbf{ref}, h : Hp, r : [\textbf{ref}]\textbf{bool} \bullet o \neq \textbf{null} \implies List(o, h) = \{o\} \cup List(h[o, nxt], h)$
axiom $\forall h : Hp, k : Hp, o : \textbf{ref} \bullet List(o, h) = List(o, k) \Longleftarrow$
         $(\forall p : \textbf{ref} \bullet p \neq \textbf{null} \wedge List(o, h)[p] \wedge List(o, k)[p] \implies h[p, nxt] = k[p, nxt])$

Figure 3: List Axioms (two defining axioms and a lemma).

Method *register* adds an object o to the region O of a subject, s, and notifies o of s's current state. When the subject's state is *update*d, it notifies all of its observers. The purpose of method *notify* is to update an observer's view of its subject's internal state. Note that *notify* is called on an observer from within the *update* method but this results in a callback to the *Subject*'s *get* method via $this.sub.get()$.

For the specifications of the methods of Fig. 2, the predicates *SubObs*, *Sub*, *Obs* are used (inspired by Parkinson [17]). The predicate $Sub(s, v)$ says that the current internal state of subject s is v and all observers of s are in a list which lies in region $s.O$. The predicate $Obs(o, s, v)$ says that o is an observer of subject s and that o's view of s's internal state is v. Finally, *SubObs* is a predicate for the entire aggregate structure comprising an instance of *Subject* together with its *Observer*s. $SubObs(s, v)$ holds for a subject s with internal state v when $Sub(s, v)$ holds and for each observer o in s's list of observers, $Obs(o, s, v)$ holds.

Unlike Parkinson's formulation in separation logic, our version includes the heap as explicit parameter. In Fig. 3 we introduce a function *List* so that $List(o, h)$ is a set containing all objects reachable from o following nxt. Using *List*, the defining axioms for *Sub*, *Obs* and *SubObs* look as follows:

$$Sub(s, v, h) \iff GO(s, h) \wedge h[s, val] = v \wedge List(h[s, obs], h) = h[s, O]$$
$$Obs(o, s, v, h) \iff GO(s, h) \wedge GO(o, h) \wedge h[o, cache] = v \wedge h[o, sub] = s$$
$$SubObs(s, v, h) \iff Sub(s, v, h) \wedge (\forall o : \textbf{ref} \bullet GO(o, h) \wedge h[s, O][o] \implies Obs(o, s, v, h))$$

The specification of *update*, omitting effects **wr** $\langle this \rangle.val$ and **wr** $this.O.cache$, is

     requires $SubObs(this, val, Heap) \wedge GO(this, Heap)$ **ensures** $SubObs(this, n, Heap)$

The implementation of *update* uses local variables $o, r1$. As observers in the subject's (i.e., *this*'s) list of observers get notified they are put in region $r1$. This leads to the loop invariant that notified observers are up to date:

$$\forall p : \mathbf{ref} \bullet GO(p, Heap) \wedge r1[p] \implies Obs(p, this, n, Heap)$$

Another loop invariant says that region $this.O$ comprises $r1$ together with $List(o, Heap)$ which is the region containing objects yet to be notified.

$$List(o, Heap) \cup r1 = Heap[this, O]$$

The verification goes through, provided we include the equality axiom listed third in Fig. 3. This follows from the first two axioms using induction but here we work in pure first order logic. Note also the apparently superflous variable r in the second axiom. It is needed for triggering (Sect. 7).

We consider a client that creates two Subjects and updates one.

$$sub0 := \mathbf{new} \; Subject(); \; obs0 := \mathbf{new} \; Observer(sub0); \; obs1 := \mathbf{new} \; Observer(sub0);$$
$$sub := \mathbf{new} \; Subject(); \; obs := \mathbf{new} \; Observer(sub); \; \mathbf{call} \; sub0.update(5);$$

We were able to verify the following postconditions for the client:

$$DisjointSet(Heap[sub0, O], Heap[sub, O]) \wedge SubObs(sub0, 5, Heap) \wedge SubObs(sub, 0, Heap)$$

These assertions say that region $sub0.O$ is disjoint from $sub.O$ and updating the internal state of $sub0$ has no effect on the internal state of sub. The key links: condition $Obs(o, s, v, h)$ implies that $o.sub = s$ and thus $SubObs(s, v, h)$ implies that every o in $s.O$ has $o.sub = s$. This is much like an encoding of ownership (as pointed out in [3] and used in VCC) and it implies that if $o.sub \neq o'.sub$ then $o.sub.O$ is disjoint from $o'.sub.O$.

BoogiePL is modular in the sense that in verifying this client code, the verifier uses only the specifications of the constructors and other methods. The next section considers modularity in more depth.

6 Abstraction and hiding examples

A flaw of the preceding Subject/Observer specification is that it mixes conditions on the internal data structures (the list) with those of interest to clients (including O which could be a model field). In Sect. 6.1 we factor apart the specification so that one part can be hidden from clients, in the manner of Hoare's treatment of invariants [12]. Hiding in this fashion introduces a potentially unsound mis-match between the specs used to verify invocations of a method and those with respect to which its implementation is verified. Such a mis-match can be justified by encapsulation, one technique for which is ownership: roughly, the invariant depend only on owned objects and clients are prevented from writing them. In Sect. 6.2 we explore the use of a field to hold the owned objects. Rather than fully develop the use of ownership for hiding, we consider abstraction, whereby an invariant is mentioned in specifications but treated opaquely as a pure method —of which the owned objects are the footprint.

6.1 Hiding

As suggested in [3], suppose we put classes *Subject* and *Observer* together in a module, giving method *register* module scope while the other methods are public. Sect. 5 considered an invariant, *SubObs*, that

pertains to a single Subject and its Observers. Let us factor it into the externally visible part, $SubObsX$, and a hidden part, $SubObsH$, used only in verification of the implementations of the $Subject$ and $Observer$ methods.

$$SubX(s, v, h) \iff GO(s, h) \wedge h[s, val] = v$$
$$SubObsX(s, v, h) \iff SubX(s, v, h) \wedge \forall o : \mathbf{ref} \in h[s, O] \bullet Obs(o, s, v, h)$$

We verified client code, including the previous example, using $SubX$ and $SubObsX$ in place of Sub and $SubObs$ in the method specifications (excepting method $register$ which would be module scoped and not used by clients). The client verifications were successful —confirming that they rely on the disjointness reasoning focused on the O field and not on properties of the list data structure.

The implementations of the methods of $Subject$ and $Observer$ are verified using specifications that use not only $SubObsX$ but also $SubObsH$ where $SubObsH(s, v, h) \iff SubH(s, h) \wedge SubObsX(s, v, h)$ and $SubH(s, h) \iff List(h[s, obs], h) = h[s, O]$. Of course these verifications go through; they merely repackage the earlier specifications.

To cater for a second order frame rule to justify hiding, Banerjee et al [3] propose to hide a single "module invariant" that for this example could be the conjunction of $\forall s : \mathbf{ref} \bullet SubObsH(s, val)$ and

$$\forall p : \mathbf{ref}, s : \mathbf{ref} \bullet p \in Heap[s, O] \wedge Heap[p, sub] \neq \mathbf{null} \Longrightarrow Heap[p, sub] = s$$

We added these as pre- and post-condition for each method, which entails disjointness reasoning with respect to arbitrary other Subjects. The verifications succeed.

6.2   Ownership and abstraction

Although the clients considered above are well behaved, the proposed hiding is actually unsound since client code like $s.obs := s.obs.nxt$ could falsify $SubH(s)$. Some form of encapsulation is needed to preclude such clients. We now consider ownership, a popular device for achieving heap encapsulation. But full justification of invariant hiding is beyond our scope. Instead, we consider ownership for the alternative to hiding: abstraction. Abstraction has been used for invariants by Müller (model fields), Parkinson [7] (existentially quantified predicates), Leino's thesis and others. Smans *et al.* [19] use a pure boolean method $invariant()$ to abstract the invariant in specifications.

Like model fields, pure methods are also useful for properties other than invariants. We explore pure methods and ownership in an example of Smans et al [19]. A stack is implemented in terms of an $ArrayList$, a dynamically resizable array which implements its functionality with a primitive fixed-size array. The interface for $ArrayList$ allows elements to be added at the end of the array, to retrieve or delete the element stored at a specific index, and to query the current size of the array. The interface for $Stack$ is $push$, pop, $empty$. In addition, $Stack$ provides a pure method, $size$, that returns the current number of elements in the stack. This is used in the postconditions of the stack's constructor, which says that its value is zero, and the method $push$, which says $\#size() = \mathbf{old}(\#size()) + 1$.

Now $size$ is implemented using the internal representation of the stack. This internal representation consists of an owned object, the $ArrayList$, and its representation. Field fp will hold references to these, and method $size$ is specified to have read effect $\mathbf{rd}\, this.fp\bullet \mathsf{any}$. The write effect of methods $push$ and pop is $\mathbf{wr}\, this.fp\bullet f$.

An interesting part of the example is the verification of the method $switch$ that exchanges the representations of two stacks. The specifications for most mutators would allow footprints to grow but only by fresh objects, but that is not the case for $switch$. Their collective footprint, however, need not change at all. We choose a specification (Fig. 4) that allows the collective footprint to grow with fresh objects, to cater

procedure $switch(this : \textbf{ref}, other : \textbf{ref})$
  free requires $GO(this, Heap) \wedge (other = null \vee GO(other, Heap))$;
  requires $other \neq null \wedge Inv(Heap, this) \wedge Inv(Heap, other)$;
  requires $DisjointSet(Heap[this,\!fp], Heap[other,\!fp])$;
  ensures $Inv(Heap, this) \wedge Inv(Heap, other) \wedge DisjointSet(Heap[this,\!fp], Heap[other,\!fp])$;
  ensures $\#size(Heap, this) = \#size(\textbf{old}(Heap), other) \wedge \#size(Heap, other) = \#size(\textbf{old}(Heap), this)$;
  ensures $DifferenceIsNew(\textbf{old}(Heap), \textbf{old}(Heap)[this,\!fp] \cup \textbf{old}(Heap)[other,\!fp],$
      $Heap[this,\!fp] \cup Heap[other,\!fp])$;
  // write effect: $\textbf{wr}\, fp\text{\textbf{.}any}, other.fp\text{\textbf{.}any}$
  ensures $(\forall p : \textbf{ref}, f : \textbf{field} \bullet GO(p, \textbf{old}(Heap)) \implies$
      $(\textbf{old}(Heap)[p, f] = Heap[p, f] \vee \textbf{old}(Heap)[this,\!fp][p] \vee \textbf{old}(Heap)[other,\!fp][p]))$;

Figure 4: Specification for Stack switch

procedure $switch(this : \textbf{ref}, other : \textbf{ref})$
$\{$ **var** $tmp : \textbf{ref}$;
  $tmp := Heap[this, contents]$;
  $Heap[this, contents] := Heap[other, contents]$;
  $Heap[other, contents] := tmp$;

  $Heap[this,\!fp] := \{this\} \cup \{Heap[this, contents]\}$
      $\cup Heap[Heap[this, contents], ArrayList.footprint]$;
  $Heap[other,\!fp] := \{other\} \cup \{Heap[other, contents]\}$
      $\cup Heap[Heap[other, contents], ArrayList.footprint]$;
  assume $GH(Heap)$;
$\}$

Figure 5: Implementation for Stack switch

for benevolent side effects that the *ArrayList* might have. Note the free precondition guaranteed by the language semantics: the receiver of an instance method is non-null and allocated, while the parameter that would have been explicit in the source language is guaranteed to be allocated only if it is not null. However, we chose to add the explicit precondition that it be non-null.

Unsurprisingly, it is necessary to have a precondition that the footprints of both stacks are disjoint. This is not guaranteed by the two stacks being different objects because it is possible for one stack to be part of the other stack's footprint. In reasoning about client code, the verifier is able to track disjointness of newly allocated stacks.

The postcondition that encodes the write effect just says that all fields in the heap retain their old values unless the field is in an object in the footprint of either of the parameters.

We consider client reasoning using the following code:

> **var** $s1$: **ref**, $s2$: **ref**; $s1 :=$ **new** $Stack()$; $s2 :=$ **new** $Stack()$;
> **assert** $\#size(Heap, s1) = 0 \wedge \#size(Heap, s2) = 0$;
> **call** $push(s2,5)$;
> **assert** $\#size(Heap, s1) = 0 \wedge \#size(Heap, s2) = 1$;

Consider the first assertion: the second conjunct, that $\#size$ is zero for $s2$ is directly from the constructor's postcondition (not shown). But for the first conjunct to hold it must be the case that the execution of the constructor for $s2$ did not affect the state that $s1$ depends on. This is done without revealing any of the details of the stack's implementation with the following axiom, which expresses that the effect of $size$ is **rd** $this.fp$•any.

$$(\forall h, k : Hp, o : \textbf{ref} \bullet$$
$$h[o,fp] = k[o,fp] \wedge (\forall p; \textbf{ref}, f : \textbf{field} \bullet h[o,fp][p] \Longrightarrow h[p,f] = k[p,f])$$
$$\Longrightarrow \#size(h, o) = \#size(k, o));$$

(Some GO conditions are elided.) Together with the default frame condition for constructors (Section 2), this is sufficient for the prover to be able to retain the knowledge about the size of the stack $s1$. The same holds for the second assert statement: the second conjunct is directly from the postcondition for $push$ while the first conjunct relies on knowing that calling $push$ on $s2$ cannot change the state that $s1$ depends on.

The above axiom is sound as long as $size$ does not depend on the field of any object which is not contained in the stack's footprint. Utility of the axiom depends on how easy it is to determine that two states satisfy the antecedent, which in turn depends on the encapsulation of the representation objects.

7   Experiences with prover

Verification failure might be due to (a) weak program specifications (pre- and postconditions), (b) weak loop invariant, and (c) incompleteness of triggers. In the first two cases one receives limited feedback from counterexamples.

A trigger of a universal quantifier is a set of expressions that determines how the SMT solver instantiates the quantifier. In theory the SMT solver can instantiate a universal quantifier with any ground term whatsoever. In practice it is better to limit the number of instantiations in some way so that the solver considers only a finite set of ground instantiations from its e-graph data structure. For example, consider the second axiom in Fig. 3 with the trigger restored:

$$\forall o : \textbf{ref}, h : Hp, r : [\textbf{ref}]\textbf{bool} \bullet \ \{List(o,h) = r\}$$
$$o \neq \textbf{null} \Longrightarrow List(o,h) = \{o\} \cup List(h[o, nxt], h)$$

This trigger instructs Z3 to instantiate the quantifier only with those o, h, r for which there is a term $List(o, h) = r$ in its e-graph.

Choosing a proper trigger is a craft. Had we removed the apparently redundant bound variable r and used $List(o, h)$ as a trigger, a matching loop would have arisen because the quantifier could be instantiated by any of $List(o, h)$, $List(o.nxt, h)$,... We tried experimenting with the trigger $List(o.nxt, h)$ instead but this still did not suffice to verify the invariant $(List(o, h) \cup r1 = Heap[this, O])$ (see Sect. 5). Finally after much experimentation and with help from more experienced colleagues we arrived at the trigger $List(o, h) = r$: the intuition was that we needed one instantiation of the quantified variable r to be $Heap[this, O]$.

At the end of Sect. 1 we posed the question: Is a first order axiomatization of set theory effective for automating verification of programs and specifications that encode footprints in ghost state? We have to answer no, unless one is an expert at triggering quantifiers or has expert colleagues close by. The need for good triggers prevented us from introducing an abstract data type for regions in our specifications.

Since Boogie uses an automatic theorem prover, there is always the danger of having a verification succeed due to it having been given a set of unsound axioms. Michał Moskal has implemented an option in Boogie, `smoke`, that creates multiple verification conditions, one per program path. It then injects the statement **assert false** at the end of each path, converts all existing assert statements into assume statements and passes the result to the theorem prover. If the prover is able to establish the **false** assertion then the facts along that path reveal inconsistencies. Our experience has been that this is the single most valuable option in Boogie. It is sobering to find out how easy it is to unintentionally introduce inconsistencies into the axiomatization.

8   Related work

The VCC project [18] is focused on the verification of low-level systems code written in C. It uses regions (sets) to structure the flat address space that the C memory model provides access to. Regions are disjoint sets of memory locations: pointers in C are modeled as offsets into a region. Footprints (read/write effects) are specified using (pure) functions, as opposed to our use of a designated field.

Smans et al [19] report on extensions of the Spec# tool to support dynamic frames where pure methods are used to represent footprints. In this work (and in [13]), footprints are sets of locations, where a *location* pairs an object reference with a field name. The language includes a notation, $\&e.f$, for the location (o, f) where o is the value of expression e; this is needed not only in specifications but in the ghost assignments that instrument code. This treatment offers more fine-grained expressiveness; in particular, it provides for abstraction over field names. Region logic needs a separate means to abstract over field names (notation "any" or model fields), but its instrumentation code can be written in Java and C# without need for the & operator (using classes like `Collection<Object>` for reference sets). It would be interesting to see whether the static analysis for framing in region logic [3] can be adapted to location sets. Besides checking write effects (using two-state postconditions as in our work), Smans *et al.* [19] check read effects in a small-step way: for each primitive expression that reads, a verification condition is generated that says the state read is within the specified read footprint. Our experiments did not check read effects. Modular reasoning is enforced in [19] like in our Sect. 6.1: In reasoning about code inside a module, the verifier is given an axiom that connects a pure method with its implementation. In reasoning about code outside a module, the verifier is given axioms about the frame of a pure method as well as its postcondition, but not its implementation.

Another difference between our work and [19] is our use of fields rather than pure methods to represent footprints. Fields are interpreted within the theorem prover so that reasoning is potentially more precise than for pure methods, though sometimes at the cost of more ghost assignments (e.g., Fig. 5). Some of the pure methods in [19] are recursively defined, which requires care [8]. Finally, they use conditional effects, for which we did not find a need.

9   Conclusion

We used BoogiePL to specify and verify several examples using effect specifications in the style of Region Logic, inspired by the dynamic frames of Kassios [13]. The standard theories of the underlying SMT solver, Z3, were augmented with some axioms for set theory. Russell famously said the advantages of postulation

are those of theft, but we found ample opportunity for honest toil in the instrumentation of axioms with triggers. Our toil would have been far greater without help from Rustan Leino, Peter Müller, and Michał Moskal. Nonetheless, we read our results as confirming the efficacy of effect specifications using ghost fields and auxiliaries holding just sets of references, lightweight machinery indeed.

The BoogiePL tool defines an axiomatic semantics for the BoogiePL programming language, insofar as it generates first order verification conditions, and we augmented that semantics with a few assumptions intended to express additional invariants of source language like Java or C#. We are unaware of any formal connection between BoogiePL and an actual language implementation. In his talk at VSTTE'05, J Moore argued that "we should build a [verification] system in some programming language for which we have a mechanically supported formal semantics and a mechanically supported reasoning engine – and [. . .] be able at least to state within the system what it means for our system to be correct. If you are working on a smaller piece of the problem – if you are building a system whose expressive power and implementation is beyond the scope of your own work – then you should find somebody to deal with the problem you are creating!". One use for a syntax-directed proof system like Region Logic is as bridge between verification conditions and the underlying semantics. The logic has been proved sound, on paper, with respect to a denotational semantics simplified from a Java/JML model that has been encoded by deep embedding in PVS [14]. Region Logic has been extended with a second order frame rule which captures hiding of invariants and has been proved admissible, on paper [15]. Given the promising experiences by ourselves and others with dynamic framing, we see the remaining gaps as well worth bridging.

All example programs have been verified in Boogie version 0.90. Complete listings of these and a number of other verified programs are available online and in the appendix.

References

[1] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *POPL*, 2006. Extended version available as KSU CIS-TR-2005-1.

[2] T. Amtoft, J. Hatcliff, E. Rodriguez, Robby, J. Hoag, and D. Greve. Specification and checking of software contracts for conditional information flow. In *Formal Methods: International Conference of Formal Methods Europe*, volume 5014 of *LNCS*, 2008. To appear.

[3] A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP*, 2008. To appear.

[4] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[5] M. Barnett, R. DeLine, B. Jacobs, M. Fähndrich, K. R. M. Leino, W. Schulte, and H. Venter. The Spec# programming system: Challenges and directions. In *Verified Software: Theories, Tools, and Experiments (VSTTE)*, 2005.

[6] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *PASTE*, 2005.

[7] G. Bierman and M. Parkinson. Separation logic and abstraction. In *POPL*, pages 247–258, 2005.

[8] Á. Darvas and P. Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology*, 5(5):59–85, 2006.

[9] L. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In *CADE*, pages 183–198, 2007.

[10] R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, Mar. 2005.

[11] E. C. R. Hehner. Predicative programming part I. *Commun. ACM*, 27:134–143, 1984.

[12] C. A. R. Hoare. Proofs of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.

[13] I. T. Kassios. Dynamic framing: Support for framing, dependencies and sharing without restriction. In *Formal Methods: International Conference of Formal Methods Europe*, volume 4085 of *LNCS*, pages 268–283, 2006.

[14] G. T. Leavens, D. A. Naumann, and S. Rosenberg. Preliminary definition of core JML. Technical Report CS Report 2006-07, Stevens Institute of Technology, 2006.

[15] D. A. Naumann. An admissible second order frame rule in region logic. Technical Report CS Report 2008-02, Stevens Institute of Technology, 2008.

[16] P. O'Hearn, H. Yang, and J. Reynolds. Separation and information hiding. In *POPL*, pages 268–280, 2004.

[17] M. Parkinson. Class invariants: the end of the road? In *International Workshop on Aliasing, Confinement and Ownership*, 2007.

[18] W. Schulte. Building a verifying compiler for C. Presentation at Dagstuhl Seminar 08061 for Types, Logics and Semantics of State, 2008.

[19] J. Smans, B. Jacobs, F. Piessens, and W. Schulte. An automatic verifier for Java-like programs based on dynamic frames. In *FASE*, 2008.

A   Sets and Regions

A.1   Sets

```
// Rustan's set axioms (from Queue.bpl)

const EmptySet: [ref]bool;
axiom (forall o: ref :: { EmptySet[o] } !EmptySet[o]);

function SingletonSet(ref) returns ([ref]bool);
axiom (forall r: ref, o: ref :: { SingletonSet(r)[o] } SingletonSet(r)[o] <==> r == o);

function UnionSet([ref]bool, [ref]bool) returns ([ref]bool);
axiom (forall a: [ref]bool, b: [ref]bool, o: ref :: { UnionSet(a,b)[o] }
  UnionSet(a,b)[o] <==> a[o] || b[o]);

function SubSet([ref]bool, [ref]bool) returns (bool);
axiom(forall a: [ref]bool, b: [ref]bool :: { SubSet(a,b) }
  SubSet(a,b) <==> (forall o: ref :: {a[o]} {b[o]} a[o] ==> b[o]));

function EqualSet([ref]bool, [ref]bool) returns (bool);
axiom(forall a: [ref]bool, b: [ref]bool :: { EqualSet(a,b) }
  EqualSet(a,b) <==> (forall o: ref :: {a[o]} {b[o]} a[o] <==> b[o]));

function DisjointSet([ref]bool, [ref]bool) returns (bool);
axiom (forall a: [ref]bool, b: [ref]bool :: { DisjointSet(a,b) }
  DisjointSet(a,b) <==> (forall o: ref :: {a[o]} {b[o]} !a[o] || !b[o]));
```

A.2   Regions

```
// Definitions
// every object has an 'alloc' field, which says whether or not
// the object has been allocated
const unique alloc: <bool>name;

function IsGoodHeap(h : [ref,<x>name]x) returns (bool);

// After this point, some helpful functions

function IsGoodObject(r: ref, h: [ref,<x>name]x) returns (bool);
axiom (forall o: ref, h: [ref,<x>name]x ::
        { IsGoodObject(o,h) }
        IsGoodObject(o,h) <==> IsGoodHeap(h) && o != null && h[o,alloc]);

function IsGoodRegion(r: [ref]bool, h: [ref,<x>name]x) returns (bool);

axiom (forall r: [ref]bool, h: [ref,<x>name]x ::
      { IsGoodRegion(r, h) }
      IsGoodRegion(r, h) <==>
         (forall o: ref :: r[o] ==> IsGoodObject(o, h))
      );
```

```
/* The following axiom is helpful */
axiom (forall r: [ref]bool, o: ref, h: [ref,<x>name]x ::
      { IsGoodRegion(UnionSet(SingletonSet(o), r), h) }
      IsGoodRegion(UnionSet(SingletonSet(o), r), h) <==>
        IsGoodObject(o, h) && IsGoodRegion(r, h)
      );


function RegionClosure(r: [ref]bool, h: [ref, <x>name]x, field: <ref>name)
        returns (bool);
// make sure that r is a good region in h
axiom (forall r: [ref]bool, h: [ref, <x>name]x, f: <ref>name ::
      { RegionClosure(r, h, f) }
          RegionClosure(r,h,f)
          <==>
          (forall o: ref ::
              r[o] ==> (h[o,f] == null || r[h[o,f]]))
      );


function NoDeallocs(oldHeap: [ref, <x>name]x, newHeap: [ref, <x>name]x) returns (bool);
axiom (forall oldHeap: [ref, <x>name]x, newHeap: [ref, <x>name]x ::
  { NoDeallocs(oldHeap, newHeap) }
  NoDeallocs(oldHeap, newHeap) <==>
    (forall o: ref :: { newHeap[o,alloc] }
      oldHeap[o,alloc] ==> newHeap[o,alloc]));


function AllNewRegion(oldHeap: [ref, <x>name]x, region: [ref]bool) returns (bool);
axiom (forall oldHeap: [ref, <x>name]x, region: [ref]bool ::
  { AllNewRegion(oldHeap, region) }
  AllNewRegion(oldHeap, region) <==>
    (forall o: ref :: { oldHeap[o,alloc] }{ region[o] }
      o != null && region[o] ==> !oldHeap[o,alloc]));


function DifferenceIsNew(oldHeap: [ref, <x>name]x,
                         oldRegion: [ref]bool, newRegion: [ref]bool)
        returns (bool);
axiom (forall oldHeap: [ref, <x>name]x, oldRegion: [ref]bool, newRegion: [ref]bool ::
  { DifferenceIsNew(oldHeap, oldRegion, newRegion) }
  DifferenceIsNew(oldHeap, oldRegion, newRegion) <==>
    (forall o: ref :: { oldHeap[o,alloc] }
      o != null && !oldRegion[o] && newRegion[o] ==> !oldHeap[o,alloc]));


function fresh(oldHeap: [ref, <x>name]x, newHeap: [ref, <x>name]x, r: [ref]bool)
        returns (bool);
axiom (forall oldHeap: [ref, <x>name]x, newHeap: [ref, <x>name]x, r: [ref]bool ::
          { fresh(oldHeap, newHeap, r) }
        fresh(oldHeap, newHeap, r) <==>
          IsGoodHeap(oldHeap) && IsGoodRegion(r, newHeap) && AllNewRegion(oldHeap, r)
      );


axiom (forall oldHeap: [ref, <x>name]x, newHeap: [ref, <x>name]x,
          r1: [ref]bool, r2: [ref]bool ::
```

```
    { fresh(oldHeap, newHeap, UnionSet(r1, r2)) }
  fresh(oldHeap, newHeap, UnionSet(r1, r2)) <==>
    IsGoodHeap(oldHeap) && IsGoodRegion(r1, newHeap) &&
    IsGoodRegion(r2, newHeap) && AllNewRegion(oldHeap, r1) &&
    AllNewRegion(oldHeap, r2)
);
```

B List Reverse

```
// List Reverse
// No allocation in list reverse
var $Heap: [ref, <x>name]x where IsGoodHeap($Heap);

// This is the "nxt" field of the class Node, used for linked lists.
const unique Node.nxt: <ref>name;

procedure ReverseInPlace(x: ref) returns ($result: ref)
    free requires IsGoodHeap($Heap);
    free requires x == null || IsGoodObject(x, $Heap);
    //following comments are the Region Logic preconditions
    // < is for subset
    //requires x in r && r.nxt < r;
    //ensures $result in r && r.nxt < r;
    //modifies [wr(r.nxt)]
    modifies $Heap;
    free ensures IsGoodHeap($Heap);
    free ensures NoDeallocs(old($Heap),$Heap);
  {
    var current : ref where current == null || IsGoodObject(current,$Heap);
    var y : ref;
    var tmp : ref;
    var r : [ref]bool;

    // simulate preconditions for things not visible outside of the method
    assume x == null || r[x];
    assume RegionClosure(r,$Heap,Node.nxt); // r.nxt < r

    current := x;
    y := null;
    while (current != null)
      //invariant x < r && y < r && r.nxt < r;
      invariant current == null || r[current];
      invariant y == null || r[y];
      invariant RegionClosure(r,$Heap,Node.nxt);
      invariant // wr r.nxt
       (forall p: ref, f: name ::
          IsGoodObject(p,old($Heap)) ==>
          old($Heap[p,f]) == $Heap[p,f] || (r[p] && f == Node.nxt)
       );

    {
      tmp := $Heap[current,Node.nxt];
      $Heap[current,Node.nxt] := y;
      y := current;
      current := tmp;
    }
    $result := y;
    // simulate postconditions for things not visible outside of the method
```

```
    assert $result == null || r[$result];
    assert RegionClosure(r,$Heap,Node.nxt);
    // modifies wr(r.nxt)
    assert
      (forall p: ref, f: name ::
        IsGoodObject(p,old($Heap)) ==>
        old($Heap[p,f]) == $Heap[p,f] || (r[p] && f == Node.nxt)
      );

}
```

C List Copy

```
// List Copy
var $Heap: [ref, <x>name]x where IsGoodHeap($Heap);

// This is the "nxt" field of the class Node, used for linked lists.
const unique Node.nxt: <ref>name;

procedure Node..ctor(this: ref );
  free requires IsGoodHeap($Heap);
  modifies $Heap;
  // user-declared postconditions
  ensures $Heap[this, Node.nxt] == null;
  // target object is allocated upon return
  free ensures IsGoodObject(this, $Heap);
  ensures
      (forall p: ref, f: name ::
        old($Heap)[p, alloc] ==> old($Heap[p,f]) == $Heap[p,f] || p == this
      );
  free ensures NoDeallocs(old($Heap),$Heap);

  // invariant: r.nxt < r;
  //           forall Node n in r : n.nxt < r
  // invariant head == null || head < r

procedure ListCopy(root: ref) returns ($result: ref)
    free requires IsGoodHeap($Heap);
    free requires (root == null) || IsGoodObject(root, $Heap);

    modifies $Heap;

    free ensures IsGoodHeap($Heap);
    free ensures NoDeallocs(old($Heap),$Heap);
  {
    var newRoot: ref, prev: ref, tmp: ref, oldListPtr: ref;
    var newRegion: [ref]bool where IsGoodRegion(newRegion,$Heap);
    var existingRegion: [ref]bool where IsGoodRegion(existingRegion, $Heap);
    var h: [ref,<x>name]x;
    var loopPreHeaderHeap: [ref,<x>name]x;

    // simulate preconditions for things not visible outside of the method
    assume root == null || existingRegion[root];
    assume RegionClosure(existingRegion,$Heap,Node.nxt);
    newRoot := null;
    tmp := null;
    oldListPtr := root;
    newRegion := EmptySet;
    if (oldListPtr != null) {
      // newRoot = new Node();
        havoc newRoot;
        assume !$Heap[newRoot,alloc];
```

```
    call Node..ctor(newRoot); // after this call IsGoodObject(newRoot) holds
  newRegion := UnionSet(newRegion, SingletonSet(newRoot)); //r = r+<newRoot>;
  prev := newRoot;
  loopPreHeaderHeap := $Heap;
  oldListPtr := $Heap[oldListPtr,Node.nxt]; //root = root.nxt;
  while (oldListPtr != null)
    invariant newRegion[newRoot];
    invariant RegionClosure(newRegion, $Heap, Node.nxt);
    invariant fresh(old($Heap), $Heap, newRegion);
    invariant NoDeallocs(loopPreHeaderHeap, $Heap);
  {
    // tmp = new Node();
      havoc tmp;
      assume !$Heap[tmp,alloc];
      call Node..ctor(tmp); // after this call IsGoodObject(tmp) holds
    //r = r+<tmp>;
    newRegion := UnionSet(newRegion,SingletonSet(tmp));
    // prev.nxt = tmp; tmp goes on the end of the growing new list
    $Heap[prev,Node.nxt] := tmp;
    assume IsGoodHeap($Heap); // need after every heap update
    prev := tmp;
    //root = root.nxt;
    oldListPtr := $Heap[oldListPtr,Node.nxt];
  }
}
$result := newRoot;

// Simulated post-conditions because "newRegion" is not visible in the method contract
assert $result == null || newRegion[$result];
assert RegionClosure(newRegion,$Heap,Node.nxt);
// everything in newRegion is fresh
assert fresh(old($Heap), $Heap, newRegion);
// newRegion # exisitingRegion
assert DisjointSet(newRegion, existingRegion);
assert IsGoodRegion(newRegion, $Heap);

}
```

D Subject Observer

```
// Subject Observer: one subject per observer
var $Heap: [ref, <x>name]x where IsGoodHeap($Heap);

// These are the fields of the class Subject
const unique Subject.obs: <ref>name;
const unique Subject.val: <int>name;
const unique Subject.O: <[ref]bool>name;

// These are the fields of the class Observer
const unique Observer.sub: <ref>name;
const unique Observer.cache: <int>name;
const unique Observer.nxt: <ref>name;

// In Spec#, these would be from the translation of [Pure] methods
// written as part of the program
// The following axioms appear in Figure 3 of the paper.
function List(o : ref, h: [ref,<x>name]x) returns ([ref]bool);
axiom (forall o : ref, h: [ref,<x>name]x :: { List(o,h) }
        (o == null <==> EqualSet(List(o,h),EmptySet))
     );

axiom (forall o : ref, h: [ref,<x>name]x, r: [ref]bool :: { EqualSet(List(o,h), r) }
// removing trigger cannot verify update
        (o != null ==> EqualSet(List(o,h),
                              UnionSet(SingletonSet(o),List(h[o,Observer.nxt],h))))
     );

axiom (forall H: [ref,<x>name]x, K: [ref,<x>name]x, o : ref ::
        { List(o,H), List(o,K) }
        (forall o2 : ref  ::
           o2 != null && List(o,H)[o2] && List(o,K)[o2]
           ==> H[o2,Observer.nxt] == K[o2,Observer.nxt])
        ==>
        EqualSet(List(o,H), List(o,K))
     );

// Sub, Obs and SubObs are functions used in the Subject/Observer specs.
function Sub(s : ref, v : int, h: [ref,<x>name]x) returns (bool);
axiom (forall s : ref, v : int, h: [ref,<x>name]x :: { Sub(s,v,h) }
        Sub(s,v,h)  <==> (IsGoodObject(s,h) && h[s,Subject.val] == v &&
                          EqualSet(List(h[s,Subject.obs],h),h[s,Subject.O]))
     );

function Obs(o : ref, s : ref, v : int, h: [ref,<x>name]x) returns (bool);
axiom (forall o : ref, s : ref, v : int, h: [ref,<x>name]x :: { Obs(o,s,v,h) }
        Obs(o,s,v,h)  <==> IsGoodObject(o,h) && IsGoodObject(s,h) &&
                           h[o,Observer.cache] == v && h[o,Observer.sub] == s
     );
```

```
function SubObs(s : ref, v : int, h: [ref,<x>name]x) returns (bool);
axiom (forall s : ref, v : int, h: [ref,<x>name]x :: { SubObs(s,v,h) }
          SubObs(s,v,h) <==> Sub(s,v,h) &&
                              (forall o : ref :: IsGoodObject(o,h) && h[s,Subject.O][o]
                                                  ==> Obs(o,s,v,h))
      );


// If s is a good object then s.O is a good region
axiom (forall h: [ref,<x>name]x, s: ref :: { IsGoodRegion(h[s, Subject.O], h) }
                IsGoodObject(s, h) ==> IsGoodRegion(h[s, Subject.O], h)
      );


axiom (forall h: [ref,<x>name]x, s: ref, o: ref :: { h[s, Subject.O][o] }
                IsGoodObject(s, h) ==> IsGoodRegion(h[s, Subject.O], h)
      );


// SUBJECT
procedure Subject..ctor(this: ref)
  free requires IsGoodObject(this,$Heap);
  modifies $Heap;
  // user-declared postconditions
  ensures SubObs(this,0,$Heap);
  ensures EqualSet($Heap[this, Subject.O], EmptySet);
  // target object is allocated upon return
  free ensures $Heap[this, alloc];
  // wr <this>\Dot val, <this>\Dot obs, <this>\Dot O
  ensures
      (forall p: ref, f: name ::
        IsGoodObject(p,old($Heap)) ==>
        old($Heap[p,f]) == $Heap[p,f] || p == this
      );
  free ensures NoDeallocs(old($Heap), $Heap);

  {
    $Heap[this,Subject.obs] := null;
    assume IsGoodHeap($Heap);
    $Heap[this,Subject.val] := 0;
    assume IsGoodHeap($Heap);
    $Heap[this,Subject.O] := EmptySet;
    assume IsGoodHeap($Heap);
  }

procedure Subject.register(this: ref, o: ref)
    free requires IsGoodObject(this,$Heap);
    free requires IsGoodObject(o,$Heap); // implies o != null
    // user-declared preconditions
    requires !$Heap[this, Subject.O][o]; // to prevent o from being added to the list twice
    requires Sub(this,$Heap[this,Subject.val],$Heap);
    requires $Heap[o,Observer.nxt] == null;
    // above not enough to prevent o from already being registered with another subject
    requires $Heap[o,Observer.sub] == this;
```

```
      modifies $Heap;
      // user-declared postconditions

      ensures Sub(this,$Heap[this,Subject.val],$Heap);
      ensures Obs(o,this,$Heap[this,Subject.val],$Heap);
      ensures EqualSet(UnionSet(SingletonSet(o),old($Heap)[this,Subject.O]),
                       $Heap[this,Subject.O]) ;

      // wr this.O\Dot nxt, <this>\Dot O, <o>\Dot nxt, <o>\Dot cache, <this>\Dot obs
      // recall write effects are with respect to pre-state of method
      ensures
        (forall p: ref, f: name ::
          IsGoodObject(p,old($Heap)) ==>
          old($Heap[p,f]) == $Heap[p,f] ||
            (old($Heap)[this,Subject.O][p] && f == Observer.nxt) ||
            (p == this && f == Subject.O) ||
            (p == o && f == Observer.nxt) ||
            (p == o && f == Observer.cache) ||
            (p == this && f == Subject.obs)
        );
      free ensures NoDeallocs(old($Heap), $Heap);
  {
      call Subject.add(this,o);
      call Observer.notify(o);
  }


procedure Subject.update(this: ref, n: int)
      // user-declared preconditions
      free requires IsGoodObject(this, $Heap);
      requires SubObs(this,$Heap[this,Subject.val],$Heap);
      modifies $Heap;
      // user-declared postconditions
      ensures SubObs(this, n, $Heap);
      // wr <this>\Dot val, <this>.O\Dot cache
      ensures
        (forall p: ref, f: name ::
          IsGoodObject(p,old($Heap)) ==>
          old($Heap[p,f]) == $Heap[p,f] ||
            (p == this && f == Subject.val) ||
            (old($Heap)[this,Subject.O][p] && f == Observer.cache)
        );
      free ensures NoDeallocs(old($Heap), $Heap);
  {
      var o : ref where (o == null) || IsGoodObject(o, $Heap);
      var r1 : [ref]bool where IsGoodRegion(r1, $Heap);

      $Heap[this,Subject.val] := n;
      assume IsGoodHeap($Heap);
      o := $Heap[this,Subject.obs];
      r1 := EmptySet;
```

```
        while (o != null)
          invariant Sub(this,n,$Heap);

          /* the following three invariants are essential */
          invariant SubSet(r1, $Heap[this, Subject.O]);
          invariant (forall p : ref :: IsGoodObject(p, $Heap) && r1[p]
                          ==> Obs(p, this, n, $Heap)); /* notified observers are uptodate */
          invariant EqualSet(UnionSet(List(o, $Heap), r1), $Heap[this, Subject.O]);

          /* translation needs to generate the following inv. automatically */
          invariant (forall p: ref, f: name ::
                      { $Heap[p,f] } // this trigger is unnecessary
                      IsGoodObject(p,old($Heap)) ==>
                          old($Heap[p,f]) == $Heap[p,f] ||
                          (p == this && f == Subject.val) ||
                          ($Heap[this,Subject.O][p] && f == Observer.cache));

      {
        call Observer.notify(o);
        r1 := UnionSet(r1, SingletonSet(o));
        o := $Heap[o,Observer.nxt];
      }
    }

procedure Subject.get(this: ref) returns ($result : int)
    free requires IsGoodObject(this, $Heap);
    // user-declared preconditions
    requires Sub(this,$Heap[this,Subject.val],$Heap);
    // user-declared postconditions
    ensures Sub(this,$Heap[this,Subject.val],$Heap);
    ensures $result == $Heap[this,Subject.val];
  {
    $result := $Heap[this,Subject.val];
  }

procedure Subject.add(this: ref, o: ref)
    // user-declared preconditions
    requires IsGoodObject(o, $Heap); // implies o != null;
    requires !$Heap[this, Subject.O][o];
    requires Sub(this,$Heap[this,Subject.val],$Heap);
    requires $Heap[o,Observer.nxt] == null;

    modifies $Heap;

    // user-declared postconditions
    ensures Sub(this,$Heap[this,Subject.val],$Heap);
    ensures EqualSet(UnionSet(SingletonSet(o),old($Heap)[this,Subject.O]),
                     $Heap[this,Subject.O]) ;

    // wr this.O\Dot nxt, <this>\Dot O, <o>\Dot nxt, <this>\Dot obs
    ensures
```

```
          (forall p: ref, f: name ::
            IsGoodObject(p,old($Heap)) ==>
            old($Heap[p,f]) == $Heap[p,f] ||
              (old($Heap)[this,Subject.O][p] && f == Observer.nxt) ||
              (p == o && f == Observer.nxt) ||
              (p == this && f == Subject.O) ||
              (p == this && f == Subject.obs)
        );
    free ensures NoDeallocs(old($Heap), $Heap);
  {
    $Heap[this,Subject.O] := UnionSet(SingletonSet(o),$Heap[this,Subject.O]);
    assume IsGoodHeap($Heap);
    // Add o to the front of the list of observers
    $Heap[o,Observer.nxt] := $Heap[this,Subject.obs];
    assume IsGoodHeap($Heap);
    $Heap[this,Subject.obs] := o;
    assume IsGoodHeap($Heap);

  }
// END OF SUBJECT
// OBSERVER
procedure Observer..ctor(this: ref, s : ref)
  free requires IsGoodObject(s, $Heap);
  // needed because we are passing self to register
  free requires IsGoodObject(this,$Heap);
  requires !$Heap[s, Subject.O][this];
  requires Sub(s, $Heap[s,Subject.val], $Heap);
  modifies $Heap;
  // user-declared postconditions
  ensures Sub(s, $Heap[s,Subject.val], $Heap);
  ensures Obs(this, s, $Heap[s,Subject.val], $Heap);

  ensures $Heap[s,Subject.O][this];
  //needed for client
  ensures EqualSet(UnionSet(SingletonSet(this), old($Heap)[s,Subject.O]),
                   $Heap[s,Subject.O]) ;
  // target object is allocated upon return
  free ensures $Heap[this, alloc]; // needed for callers of Observer..ctor
  // wr s.O\Dot nxt, <s>\Dot O
  // Since the only nxt field that is changed is "this" one, don't need to own
  // up to it here in the modifies clause.
    ensures
      (forall p: ref, f: name ::
        IsGoodObject(p,old($Heap)) ==>
        old($Heap[p,f]) == $Heap[p,f] || p == this ||
          (old($Heap)[s,Subject.O][p] && f == Observer.nxt) ||
          (p == s && f == Subject.O) ||
          (p == s && f == Subject.obs)          // needed because of add
      );
    free ensures NoDeallocs(old($Heap), $Heap);
  {
```

26

```
    $Heap[this,Observer.sub] := s;
    assume IsGoodHeap($Heap);
    $Heap[this,Observer.nxt] := null;
    assume IsGoodHeap($Heap);
    call Subject.register(s,this);
  }


procedure Observer.notify(this: ref)
    free requires IsGoodObject(this,$Heap);
    // user-declared preconditions
    requires Sub($Heap[this,Observer.sub],
                 $Heap[$Heap[this,Observer.sub],Subject.val],
                 $Heap);
    modifies $Heap;
    // user-declared postconditions
    ensures Sub($Heap[this,Observer.sub],
                $Heap[$Heap[this,Observer.sub],Subject.val],
                $Heap);
    ensures Obs(this,$Heap[this,Observer.sub],
                $Heap[$Heap[this,Observer.sub],Subject.val], $Heap);
    // wr <this>.\Dot cache
    ensures
      (forall p: ref, f: name :: { $Heap[p,f] }
        IsGoodObject(p,old($Heap)) ==>
        old($Heap[p,f]) == $Heap[p,f] ||
          (p == this && f == Observer.cache)
      );
    free ensures NoDeallocs(old($Heap), $Heap);
  {
    var tmp : int;
    call tmp := Subject.get($Heap[this,Observer.sub]);
    $Heap[this,Observer.cache] := tmp;
    assume IsGoodHeap($Heap); // needed after each update to $Heap
  }


procedure Observer.val(this: ref) returns ($result : int)
    free requires IsGoodObject(this,$Heap);
    // user-declared preconditions
    requires Obs(this,$Heap[this,Observer.sub],
                 $Heap[$Heap[this,Observer.sub],Subject.val],$Heap);
    requires $Heap[$Heap[this,Observer.sub],Subject.O][this];
    // user-declared postconditions
    ensures Obs(this,$Heap[this,Observer.sub],
                $Heap[$Heap[this,Observer.sub],Subject.val],$Heap);
    ensures $result == $Heap[$Heap[this,Observer.sub],Subject.val];
  {
    $result := $Heap[this,Observer.cache];
  }


// END OF OBSERVER
```

E   A Client for Subject Observer

```
// A client for subject/observer
procedure client()
    modifies $Heap;
    // user-declared postconditions
    // effects
  {
        var sub0 : ref;
        var obs0 : ref;
        var obs1 : ref;
        var sub  : ref;
        var obs  : ref;

    // sub0 = new Subject();
        havoc sub0;
        assume !$Heap[sub0,alloc];
        call Subject..ctor(sub0);

    // obs0 = new Observer(sub0);
        havoc obs0;
        assume !$Heap[obs0,alloc];
        assume EqualSet($Heap[obs0, Subject.O],  EmptySet);
        call Observer..ctor(obs0, sub0);

    // obs1 = new Observer(sub0);
        havoc obs1;
        assume !$Heap[obs1,alloc];
        assume EqualSet($Heap[obs1, Subject.O],  EmptySet);
        call Observer..ctor(obs1, sub0);

  //   sub  = new Subject();
        havoc sub;
        assume !$Heap[sub,alloc];
        call Subject..ctor(sub);

   //  obs = new Observer(sub);
        havoc obs;
        assume !$Heap[obs,alloc];
        assume EqualSet($Heap[obs, Subject.O],  EmptySet);
        call Observer..ctor(obs, sub);

        call Subject.update(sub0, 5);

        assert DisjointSet($Heap[sub0, Subject.O], $Heap[sub, Subject.O]);

        assert SubObs(sub0, 5, $Heap);

        assert SubObs(sub, 0, $Heap);
  }
```

F Subject Observer with Hiding

```
// Hiding: This conforms to Section 6.1 of the paper
// Using SubX(s,v,h), SubObsX(s,v,h) and the hidden invariant SubObsH(s,v,h)
var $Heap: [ref, <x>name]x where IsGoodHeap($Heap);

// These are the fields of the class Subject
const unique Subject.obs: <ref>name;
const unique Subject.val: <int>name;
const unique Subject.O: <[ref]bool>name;

// These are the fields of the class Observer
const unique Observer.sub: <ref>name;
const unique Observer.cache: <int>name;
const unique Observer.nxt: <ref>name;

// In Spec#, these would be from the translation of [Pure] methods
// written as part of the program
function List(o : ref, h: [ref,<x>name]x) returns ([ref]bool);
axiom (forall o : ref, h: [ref,<x>name]x :: { List(o,h) }
        (o == null <==> EqualSet(List(o,h),EmptySet))
    );

axiom (forall o : ref, h: [ref,<x>name]x, r: [ref]bool :: { EqualSet(List(o,h), r) }
        (o != null ==> EqualSet(List(o,h),
                                UnionSet(SingletonSet(o),List(h[o,Observer.nxt],h))))
    );

axiom (forall H: [ref,<x>name]x, K: [ref,<x>name]x, o : ref ::
         { List(o,H), List(o,K) }
         (forall o2 : ref  ::
             o2 != null && List(o,H)[o2] && List(o,K)[o2]
             ==> H[o2,Observer.nxt] == K[o2,Observer.nxt])
          ==>
          EqualSet(List(o,H), List(o,K))
    );

function SubX(s : ref, v : int, h: [ref,<x>name]x) returns (bool);
axiom (forall s : ref, v : int, h: [ref,<x>name]x :: { SubX(s,v,h) }
        SubX(s,v,h)  <==> (IsGoodObject(s,h) && h[s,Subject.val] == v)
    );

function SubH(s : ref, h: [ref,<x>name]x) returns (bool);
axiom (forall s : ref, h: [ref,<x>name]x :: { SubH(s,h) }
        SubH(s,h)  <==> EqualSet(List(h[s,Subject.obs],h),h[s,Subject.O])
    );

function Obs(o : ref, s : ref, v : int, h: [ref,<x>name]x) returns (bool);
axiom (forall o : ref, s : ref, v : int, h: [ref,<x>name]x :: { Obs(o,s,v,h) }
        Obs(o,s,v,h)  <==> IsGoodObject(o,h) && IsGoodObject(s,h) &&
                           h[o,Observer.cache] == v && h[o,Observer.sub] == s
```

```
      );

function SubObsX(s : ref, v : int, h: [ref,<x>name]x) returns (bool);
axiom (forall s : ref, v : int, h: [ref,<x>name]x :: { SubObsX(s,v,h) }
        SubObsX(s,v,h) <==> SubX(s,v,h) &&
                              (forall o : ref :: IsGoodObject(o,h) && h[s,Subject.O][o]
                                        ==> Obs(o,s,v,h))
      );

function SubObsH(s : ref, v : int, h: [ref,<x>name]x) returns (bool);
axiom (forall s : ref, v : int, h: [ref,<x>name]x :: { SubObsH(s,v,h) }
        SubObsH(s,v,h) <==> SubX(s,v,h) && SubH(s,h)
      );

axiom (forall h: [ref,<x>name]x, s: ref :: { IsGoodRegion(h[s, Subject.O], h) }
              IsGoodObject(s, h) ==> IsGoodRegion(h[s, Subject.O], h)
      );

axiom (forall h: [ref,<x>name]x, s: ref, o: ref :: { h[s, Subject.O][o] }
              IsGoodObject(s, h) ==> IsGoodRegion(h[s, Subject.O], h)
      );

procedure Subject..ctor(this: ref)

    free requires IsGoodObject(this,$Heap);
    modifies $Heap;
    // user-declared postconditions
    ensures SubObsX(this,0,$Heap);
    ensures EqualSet($Heap[this, Subject.O], EmptySet);
    // target object is allocated upon return
    free ensures $Heap[this, alloc];
    // wr <this>\Dot val, <this>\Dot obs, <this>\Dot O
    ensures
      (forall p: ref, f: name ::
         IsGoodObject(p,old($Heap)) ==>
         old($Heap[p,f]) == $Heap[p,f] || p == this
      );
    free ensures NoDeallocs(old($Heap), $Heap);

  {
    $Heap[this,Subject.obs] := null;
    assume IsGoodHeap($Heap);
    $Heap[this,Subject.val] := 0;
    assume IsGoodHeap($Heap);
    $Heap[this,Subject.O] := EmptySet;
    assume IsGoodHeap($Heap);
  }

procedure Subject.register(this: ref, o: ref)
    free requires IsGoodObject(this,$Heap);
    free requires IsGoodObject(o,$Heap); // implies o != null
```

```
    // user-declared preconditions
    requires !$Heap[this, Subject.O][o]; /* to prevent o from being added to the list twice */
    requires SubX(this,$Heap[this,Subject.val],$Heap);
    // not enough to prevent o from already being registered with another subject
    requires $Heap[o,Observer.nxt] == null;
    //about to register o in above, so require its sub field to point to this
    requires $Heap[o,Observer.sub] == this;
    requires SubObsH(this, $Heap[this,Subject.val],$Heap);

    modifies $Heap;
    // user-declared postconditions

    ensures SubX(this,$Heap[this,Subject.val],$Heap);
    ensures Obs(o,this,$Heap[this,Subject.val],$Heap);
    ensures EqualSet(UnionSet(SingletonSet(o),old($Heap)[this,Subject.O]),
                             $Heap[this,Subject.O]) ;

    ensures SubObsH(this, $Heap[this,Subject.val],$Heap);
    // wr this.O\Dot nxt, <this>\Dot O, <o>\Dot nxt, <o>\Dot cache, <this>\Dot obs
    ensures
      (forall p: ref, f: name ::
        IsGoodObject(p,old($Heap)) ==>
        old($Heap[p,f]) == $Heap[p,f] ||
          (old($Heap)[this,Subject.O][p] && f == Observer.nxt) ||
          (p == this && f == Subject.O) ||
          (p == o && f == Observer.nxt) ||
          (p == o && f == Observer.cache) ||
          (p == this && f == Subject.obs)
      );
    free ensures NoDeallocs(old($Heap), $Heap);
  {
    call Subject.add(this,o);
    call Observer.notify(o);

  }

procedure Subject.update(this: ref, n: int)
    // user-declared preconditions
    free requires IsGoodObject(this, $Heap);
    requires SubObsX(this,$Heap[this,Subject.val],$Heap);
    requires SubObsH(this,$Heap[this,Subject.val],$Heap);

    modifies $Heap;
    // user-declared postconditions
    ensures SubObsX(this, n, $Heap);
    ensures SubObsH(this, $Heap[this,Subject.val],$Heap);

    ensures
      (forall p: ref, f: name ::
        IsGoodObject(p,old($Heap)) ==>
        old($Heap[p,f]) == $Heap[p,f] ||
```

```
              (p == this && f == Subject.val) ||
              (old($Heap)[this,Subject.O][p] && f == Observer.cache)
        );
    free ensures NoDeallocs(old($Heap), $Heap);
  {
    var o : ref where (o == null) || IsGoodObject(o, $Heap);
    var r1 : [ref]bool where IsGoodRegion(r1, $Heap);

    $Heap[this,Subject.val] := n;
    assume IsGoodHeap($Heap);
    o := $Heap[this,Subject.obs];
    r1 := EmptySet;

    while (o != null)
      invariant SubX(this,n,$Heap);
      /* the following invariants are essential */
      invariant SubSet(r1, $Heap[this, Subject.O]);
      invariant (forall p : ref :: IsGoodObject(p, $Heap) && r1[p]
                      ==> Obs(p, this, n, $Heap)); /* notified observers are uptodate */
      invariant EqualSet(UnionSet(List(o, $Heap), r1), $Heap[this, Subject.O]);
      invariant SubObsH(this, $Heap[this,Subject.val],$Heap);

      /* translation needs to generate the following inv. automatically */
      invariant (forall p: ref, f: name ::
                    { $Heap[p,f] } // this trigger is unnecessary
                    IsGoodObject(p,old($Heap)) ==>
                          old($Heap[p,f]) == $Heap[p,f] ||
                        (p == this && f == Subject.val) ||
                        (old($Heap)[this,Subject.O][p] && f == Observer.cache));

    {
      call Observer.notify(o);
      r1 := UnionSet(r1, SingletonSet(o));
      o := $Heap[o,Observer.nxt];
    }
  }

procedure Subject.get(this: ref) returns ($result : int)
    free requires IsGoodObject(this, $Heap);
    // user-declared preconditions
    requires SubX(this,$Heap[this,Subject.val],$Heap);
    requires SubObsH(this, $Heap[this,Subject.val],$Heap);
    // user-declared postconditions
    ensures SubX(this,$Heap[this,Subject.val],$Heap);
    ensures $result == $Heap[this,Subject.val];
    ensures SubObsH(this, $Heap[this,Subject.val],$Heap);

  {
    $result := $Heap[this,Subject.val];
  }
```

```
procedure Subject.add(this: ref, o: ref)
    // user-declared preconditions
    requires IsGoodObject(o, $Heap); // implies o != null;
    requires !$Heap[this, Subject.O][o];
    requires SubX(this,$Heap[this,Subject.val],$Heap);
    requires $Heap[o,Observer.nxt] == null;
    requires $Heap[o, Observer.sub] == this; //this is critical
    requires SubObsH(this, $Heap[this,Subject.val],$Heap);
    modifies $Heap;

    // user-declared postconditions
    ensures SubX(this,$Heap[this,Subject.val],$Heap);
    ensures EqualSet(UnionSet(SingletonSet(o),old($Heap)[this,Subject.O]),
                     $Heap[this,Subject.O]) ;
    ensures SubObsH(this, $Heap[this,Subject.val],$Heap);

    // wr this.O\Dot nxt, <this>\Dot O, <o>\Dot nxt, <this>\Dot obs
    ensures
      (forall p: ref, f: name ::
        IsGoodObject(p,old($Heap)) ==>
        old($Heap)[p,f] == $Heap[p,f] ||
          (old($Heap)[this,Subject.O][p] && f == Observer.nxt) ||
          (p == o && f == Observer.nxt) ||
          (p == this && f == Subject.O) ||
          (p == this && f == Subject.obs)
      );
    free ensures NoDeallocs(old($Heap), $Heap);
  {
    $Heap[this,Subject.O] := UnionSet(SingletonSet(o),$Heap[this,Subject.O]);
    assume IsGoodHeap($Heap);
    // Add o to the front of the list of observers
    $Heap[o,Observer.nxt] := $Heap[this,Subject.obs];
    assume IsGoodHeap($Heap);
    $Heap[this,Subject.obs] := o;
    assume IsGoodHeap($Heap);
  }

procedure Observer..ctor(this: ref, s : ref)
    free requires IsGoodObject(s, $Heap);
    free requires IsGoodObject(this,$Heap);
    // above needed because we are passing self to register */
    requires (forall p : ref :: IsGoodObject(p, $Heap) ==> !$Heap[p, Subject.O][this]);
    requires SubX(s, $Heap[s,Subject.val], $Heap);
    requires SubObsH(s, $Heap[s,Subject.val],$Heap);
    modifies $Heap;
    // user-declared postconditions
    ensures SubX(s, $Heap[s,Subject.val], $Heap);
    ensures Obs(this, s, $Heap[s,Subject.val], $Heap);

    ensures $Heap[s,Subject.O][this];
    ensures EqualSet(UnionSet(SingletonSet(this), old($Heap)[s,Subject.O]),
```

```
                               $Heap[s,Subject.O]) ; //needed for client

        ensures SubObsH(s, $Heap[s,Subject.val],$Heap);
        // wr s.O\Dot nxt, <s>\Dot O
        // Since the only nxt field that is changed is "this" one, don't need to own
        // up to it here in the modifies clause.
        ensures
          (forall p: ref, f: name ::
            IsGoodObject(p,old($Heap)) ==>
            old($Heap)[p,f] == $Heap[p,f] || p == this ||
              (old($Heap)[s,Subject.O][p] && f == Observer.nxt) ||
              (p == s && f == Subject.O) ||
              (p == s && f == Subject.obs)          /* needed because of add */
          );
        free ensures NoDeallocs(old($Heap), $Heap);
        {
          // observer takes responsibility of setting its sub field
          $Heap[this,Observer.sub] := s;
          assume IsGoodHeap($Heap);
          $Heap[this,Observer.nxt] := null;
          assume IsGoodHeap($Heap);
          call Subject.register(s,this); // pre for this call is this.sub == s
        }


procedure Observer.notify(this: ref)
        free requires IsGoodObject(this,$Heap);
        // user-declared preconditions
        requires SubX($Heap[this,Observer.sub],
                      $Heap[$Heap[this,Observer.sub],Subject.val],$Heap);
        requires SubObsH($Heap[this,Observer.sub],
                         $Heap[$Heap[this,Observer.sub],Subject.val],$Heap);
        modifies $Heap;
        // user-declared postconditions
        ensures SubX($Heap[this,Observer.sub],
                     $Heap[$Heap[this,Observer.sub],Subject.val],$Heap);
        ensures Obs(this,$Heap[this,Observer.sub],
                    $Heap[$Heap[this,Observer.sub],Subject.val], $Heap);
        // wr <this>.\Dot cache
        ensures
          (forall p: ref, f: name :: { $Heap[p,f] }
            IsGoodObject(p,old($Heap)) ==>
            old($Heap[p,f]) == $Heap[p,f] ||
              (p == this && f == Observer.cache)
          );
        free ensures NoDeallocs(old($Heap), $Heap);
        {
          var tmp : int;
          call tmp := Subject.get($Heap[this,Observer.sub]);
          $Heap[this,Observer.cache] := tmp;
          assume IsGoodHeap($Heap); // needed after each update to $Heap
        }
```

```
procedure Observer.val(this: ref) returns ($result : int)
    free requires IsGoodObject(this,$Heap);
    // user-declared preconditions
    requires SubObsX($Heap[this,Observer.sub],
                     $Heap[$Heap[this,Observer.sub],Subject.val],$Heap);
    requires $Heap[$Heap[this,Observer.sub],Subject.O][this];
    // user-declared postconditions
    ensures SubObsX($Heap[this,Observer.sub],
                    $Heap[$Heap[this,Observer.sub],Subject.val],$Heap);
    ensures $result == $Heap[$Heap[this,Observer.sub],Subject.val];
    {
      $result := $Heap[this,Observer.cache];
    }
```

G   Subject Observer with Module Invariant

```
// Hiding part2 : hiding a single Module Invariant (Section 6.1 of paper)
var $Heap: [ref, <x>name]x where IsGoodHeap($Heap);

// These are the fields of the class Subject
const unique Subject.obs: <ref>name;
const unique Subject.val: <int>name;
const unique Subject.O: <[ref]bool>name;

// These are the fields of the class Observer
const unique Observer.sub: <ref>name;
const unique Observer.cache: <int>name;
const unique Observer.nxt: <ref>name;

// In Spec#, these would be from the translation of [Pure] methods
// written as part of the program
function List(o : ref, h: [ref,<x>name]x) returns ([ref]bool);
axiom (forall o : ref, h: [ref,<x>name]x :: { List(o,h) }
        (o == null <==> EqualSet(List(o,h),EmptySet))
     );

axiom (forall o : ref, h: [ref,<x>name]x, r: [ref]bool :: { EqualSet(List(o,h), r) }
        (o != null ==> EqualSet(List(o,h),
                                UnionSet(SingletonSet(o),List(h[o,Observer.nxt],h))))
     );

axiom (forall H: [ref,<x>name]x, K: [ref,<x>name]x, o : ref ::
         { List(o,H), List(o,K) }
         (forall o2 : ref  ::
             o2 != null && List(o,H)[o2] && List(o,K)[o2]
             ==> H[o2,Observer.nxt] == K[o2,Observer.nxt])
          ==>
          EqualSet(List(o,H), List(o,K))
     );

function SubX(s : ref, v : int, h: [ref,<x>name]x) returns (bool);
axiom (forall s : ref, v : int, h: [ref,<x>name]x :: { SubX(s,v,h) }
        SubX(s,v,h)  <==> (IsGoodObject(s,h) && h[s,Subject.val] == v)
     );

function SubH(s : ref, h: [ref,<x>name]x) returns (bool);
axiom (forall s : ref, h: [ref,<x>name]x :: { SubH(s,h) }
        SubH(s,h)  <==> EqualSet(List(h[s,Subject.obs],h),h[s,Subject.O])
     );

function Obs(o : ref, s : ref, v : int, h: [ref,<x>name]x) returns (bool);
axiom (forall o : ref, s : ref, v : int, h: [ref,<x>name]x :: { Obs(o,s,v,h) }
        Obs(o,s,v,h)  <==> IsGoodObject(o,h) && IsGoodObject(s,h) &&
                           h[o,Observer.cache] == v && h[o,Observer.sub] == s
     );
```

```
function SubObsX(s : ref, v : int, h: [ref,<x>name]x) returns (bool);
axiom (forall s : ref, v : int, h: [ref,<x>name]x :: { SubObsX(s,v,h) }
        SubObsX(s,v,h) <==> SubX(s,v,h) &&
                            (forall o : ref :: IsGoodObject(o,h) && h[s,Subject.O][o]
                                              ==> Obs(o,s,v,h))
       );


function SubObsH(s : ref, v : int, h: [ref,<x>name]x) returns (bool);
axiom (forall s : ref, v : int, h: [ref,<x>name]x :: { SubObsH(s,v,h) }
        SubObsH(s,v,h) <==> SubX(s,v,h) && SubH(s,h)
       );


axiom (forall h: [ref,<x>name]x, s: ref :: { IsGoodRegion(h[s, Subject.O], h) }
              IsGoodObject(s, h) ==> IsGoodRegion(h[s, Subject.O], h)
       );


axiom (forall h: [ref,<x>name]x, s: ref, o: ref :: { h[s, Subject.O][o] }
              IsGoodObject(s, h) ==> IsGoodRegion(h[s, Subject.O], h)
       );


procedure Subject..ctor(this: ref)

    free requires IsGoodObject(this,$Heap);
    /* pre of module invariant */
    requires (forall s: ref :: SubObsH(s,$Heap[s,Subject.val],$Heap));
    requires (forall p:ref, s:ref ::
                IsGoodObject(p, $Heap) && IsGoodObject(s, $Heap) &&
                $Heap[s, Subject.O][p] && $Heap[p, Observer.sub] != null
                ==> $Heap[p, Observer.sub] == s
             );

    modifies $Heap;
    // user-declared postconditions
    ensures SubObsX(this,0,$Heap);
    ensures EqualSet($Heap[this, Subject.O], EmptySet);
    // target object is allocated upon return
    free ensures $Heap[this, alloc];
    // wr <this>\Dot val, <this>\Dot obs, <this>\Dot O
    ensures
      (forall p: ref, f: name ::
         IsGoodObject(p,old($Heap)) ==>
         old($Heap[p,f]) == $Heap[p,f] || p == this
      );
    /* post of module invariant */
    ensures (forall s: ref :: SubObsH(s,$Heap[s,Subject.val],$Heap));
    ensures (forall p:ref, s:ref ::
                IsGoodObject(p, $Heap) && IsGoodObject(s, $Heap) &&
                $Heap[s, Subject.O][p] && $Heap[p, Observer.sub] != null
                ==> $Heap[p, Observer.sub] == s
             );
```

```
      free ensures NoDeallocs(old($Heap), $Heap);


  {
    $Heap[this,Subject.obs] := null;
    assume IsGoodHeap($Heap);
    $Heap[this,Subject.val] := 0;
    assume IsGoodHeap($Heap);
    $Heap[this,Subject.O] := EmptySet;
    assume IsGoodHeap($Heap);
  }


procedure Subject.register(this: ref, o: ref)
    free requires IsGoodObject(this,$Heap);
    free requires IsGoodObject(o,$Heap); // implies o != null
    // user-declared preconditions
    requires !$Heap[this, Subject.O][o]; /* to prevent o from being added to the list twice */
    requires SubX(this,$Heap[this,Subject.val],$Heap);
    // not enough to prevent o from already being registered with anothr subject
    requires $Heap[o,Observer.nxt] == null;
    //about to register o, so require its sub field to point to this
    requires $Heap[o,Observer.sub] == this;
    /* pre of module invariant */
    requires (forall s: ref :: SubObsH(s,$Heap[s,Subject.val],$Heap));
    requires (forall p:ref, s:ref ::
                IsGoodObject(p, $Heap) && IsGoodObject(s, $Heap) &&
                $Heap[s, Subject.O][p] && $Heap[p, Observer.sub] != null
                ==> $Heap[p, Observer.sub] == s
             );

    modifies $Heap;
    // user-declared postconditions

    ensures SubX(this,$Heap[this,Subject.val],$Heap);
    ensures Obs(o,this,$Heap[this,Subject.val],$Heap);
    ensures EqualSet(UnionSet(SingletonSet(o),old($Heap)[this,Subject.O]),
                     $Heap[this,Subject.O]) ;

    /* post of module invariant */
    ensures (forall s: ref :: SubObsH(s,$Heap[s,Subject.val],$Heap));
    ensures (forall p:ref, s:ref ::
                IsGoodObject(p, $Heap) && IsGoodObject(s, $Heap) &&
                $Heap[s, Subject.O][p] && $Heap[p, Observer.sub] != null
                ==> $Heap[p, Observer.sub] == s
             );
    // wr this.O\Dot nxt, <this>\Dot O, <o>\Dot nxt, <o>\Dot cache, <this>\Dot obs
    ensures
      (forall p: ref, f: name ::
        IsGoodObject(p,old($Heap)) ==>
        old($Heap[p,f]) == $Heap[p,f] ||
          (old($Heap)[this,Subject.O][p] && f == Observer.nxt) ||
          (p == this && f == Subject.O) ||
```

```
                (p == o && f == Observer.nxt) ||
                (p == o && f == Observer.cache) ||
                (p == this && f == Subject.obs)
        );
      free ensures NoDeallocs(old($Heap), $Heap);
  {
      call Subject.add(this,o);
      call Observer.notify(o);


  }


procedure Subject.update(this: ref, n: int)
      // user-declared preconditions
      free requires IsGoodObject(this, $Heap);
      requires SubObsX(this,$Heap[this,Subject.val],$Heap);
      /* pre of module invariant */
      requires (forall s: ref :: SubObsH(s,$Heap[s,Subject.val],$Heap));
      requires (forall o: ref, s: ref ::
                  IsGoodObject(o, $Heap) && IsGoodObject(s, $Heap) &&
                  $Heap[s, Subject.O][o] && $Heap[o, Observer.sub] != null
                  ==>
                  $Heap[o, Observer.sub] == s
              );
      modifies $Heap;
      // user-declared postconditions
      ensures SubObsX(this, n, $Heap);
      /* post of module invariant */
      ensures (forall s: ref :: SubObsH(s, $Heap[s,Subject.val],$Heap));
      ensures (forall o: ref, s: ref ::
                  IsGoodObject(o, $Heap) && IsGoodObject(s, $Heap) &&
                  $Heap[s, Subject.O][o] && $Heap[o, Observer.sub] != null
                  ==>
                  $Heap[o, Observer.sub] == s
              );
      // wr <this>\Dot val, <this>.O\Dot cache
      ensures
        (forall p: ref, f: name ::
          IsGoodObject(p,old($Heap)) ==>
          old($Heap[p,f]) == $Heap[p,f] ||
            (p == this && f == Subject.val) ||
            (old($Heap)[this,Subject.O][p] && f == Observer.cache)
        );
      free ensures NoDeallocs(old($Heap), $Heap);
  {
      var o : ref where (o == null) || IsGoodObject(o, $Heap);
      var r1 : [ref]bool where IsGoodRegion(r1, $Heap);

      $Heap[this,Subject.val] := n;
      assume IsGoodHeap($Heap);
      o := $Heap[this,Subject.obs];
      r1 := EmptySet;
```

```
    while (o != null)
      invariant SubX(this,n,$Heap);
      /* the following invariants are essential */
      invariant SubSet(r1, $Heap[this, Subject.O]);
      invariant (forall p : ref :: IsGoodObject(p, $Heap) && r1[p]
                      ==> Obs(p, this, n, $Heap)); /* notified observers are uptodate */
      invariant EqualSet(UnionSet(List(o, $Heap), r1), $Heap[this, Subject.O]);
      invariant (forall s: ref ::
                    SubObsH(s, $Heap[s,Subject.val],$Heap)
                );

      invariant (forall p: ref, s: ref ::
                    IsGoodObject(p, $Heap) && IsGoodObject(s, $Heap) &&
                    $Heap[s, Subject.O][p] && $Heap[p, Observer.sub] != null
                    ==>
                    $Heap[p, Observer.sub] == s
                );

      /* translation needs to generate the following inv. automatically */
      invariant (forall p: ref, f: name ::
                    { $Heap[p,f] } // this trigger is unnecessary
                    IsGoodObject(p,old($Heap)) ==>
                       old($Heap[p,f]) == $Heap[p,f] ||
                       (p == this && f == Subject.val) ||
                       (old($Heap)[this,Subject.O][p] && f == Observer.cache));

    {
      call Observer.notify(o);
      r1 := UnionSet(r1, SingletonSet(o));
      o := $Heap[o,Observer.nxt];
    }
  }

procedure Subject.get(this: ref) returns ($result : int)
    free requires IsGoodObject(this, $Heap);
    // user-declared preconditions
    requires SubX(this,$Heap[this,Subject.val],$Heap);

    /* pre of module invariant */
    requires (forall s: ref :: SubObsH(s,$Heap[s,Subject.val],$Heap));
    requires (forall p:ref, s:ref ::
                IsGoodObject(p, $Heap) && IsGoodObject(s, $Heap) &&
                $Heap[s, Subject.O][p] && $Heap[p, Observer.sub] != null
                ==> $Heap[p, Observer.sub] == s
            );

    // user-declared postconditions
    ensures SubX(this,$Heap[this,Subject.val],$Heap);
    ensures $result == $Heap[this,Subject.val];
    /* post of module invariant */
```

```
    ensures (forall s: ref :: SubObsH(s,$Heap[s,Subject.val],$Heap));
    ensures (forall p:ref, s:ref ::
                IsGoodObject(p, $Heap) && IsGoodObject(s, $Heap) &&
                $Heap[s, Subject.O][p] && $Heap[p, Observer.sub] != null
                ==> $Heap[p, Observer.sub] == s
            );

  {
    $result := $Heap[this,Subject.val];
  }


procedure Subject.add(this: ref, o: ref)
    // user-declared preconditions
    requires IsGoodObject(o, $Heap); // implies o != null;
    requires !$Heap[this, Subject.O][o];
    requires SubX(this,$Heap[this,Subject.val],$Heap);
    requires $Heap[o,Observer.nxt] == null;
    requires $Heap[o, Observer.sub] == this; //this is critical
    /* pre of module invariant */

    requires (forall s: ref :: SubObsH(s,$Heap[s,Subject.val],$Heap));
    requires (forall p:ref, s:ref ::
                IsGoodObject(p, $Heap) && IsGoodObject(s, $Heap) &&
                $Heap[s, Subject.O][p] && $Heap[p, Observer.sub] != null
                ==> $Heap[p, Observer.sub] == s
            );
    modifies $Heap;

    // user-declared postconditions
    ensures SubX(this,$Heap[this,Subject.val],$Heap);
    ensures EqualSet(UnionSet(SingletonSet(o),old($Heap)[this,Subject.O]),
                    $Heap[this,Subject.O]) ;
    /* post of module invariant */
    ensures (forall s: ref :: SubObsH(s,$Heap[s,Subject.val],$Heap));
    ensures (forall p:ref, s:ref ::
                IsGoodObject(p, $Heap) && IsGoodObject(s, $Heap) &&
                $Heap[s, Subject.O][p] && $Heap[p, Observer.sub] != null
                ==> $Heap[p, Observer.sub] == s
            );
    // wr this.O\Dot nxt, <this>\Dot O, <o>\Dot nxt, <this>\Dot obs
    ensures
      (forall p: ref, f: name ::
        IsGoodObject(p,old($Heap)) ==>
        old($Heap)[p,f] == $Heap[p,f] ||
          (old($Heap)[this,Subject.O][p] && f == Observer.nxt) ||
          (p == o && f == Observer.nxt) ||
          (p == this && f == Subject.O) ||
          (p == this && f == Subject.obs)
    );
    free ensures NoDeallocs(old($Heap), $Heap);
```

```
  {
    $Heap[this,Subject.O] := UnionSet(SingletonSet(o),$Heap[this,Subject.O]);
    assume IsGoodHeap($Heap);
    // Add o to the front of the list of observers
    $Heap[o,Observer.nxt] := $Heap[this,Subject.obs];
    assume IsGoodHeap($Heap);
    $Heap[this,Subject.obs] := o;
    assume IsGoodHeap($Heap);
  }

procedure Observer..ctor(this: ref, s : ref)
    free requires IsGoodObject(s, $Heap);
    // needed because we are passing self to register */
    free requires IsGoodObject(this,$Heap);
    requires (forall p : ref :: IsGoodObject(p, $Heap) ==> !$Heap[p, Subject.O][this]);

    requires SubX(s, $Heap[s,Subject.val], $Heap);
    /* pre of module invariant */

    requires (forall s0: ref :: SubObsH(s0,$Heap[s0,Subject.val],$Heap));
    requires (forall p:ref, s0:ref ::
                IsGoodObject(p, $Heap) && IsGoodObject(s0, $Heap) &&
                $Heap[s0, Subject.O][p] && $Heap[p, Observer.sub] != null
                ==> $Heap[p, Observer.sub] == s0
            );

    modifies $Heap;
    // user-declared postconditions
    ensures SubX(s, $Heap[s,Subject.val], $Heap);
    ensures Obs(this, s, $Heap[s,Subject.val], $Heap);

    ensures $Heap[s,Subject.O][this];
    //needed for client
    ensures EqualSet(UnionSet(SingletonSet(this), old($Heap)[s,Subject.O]),
                    $Heap[s,Subject.O]) ;

    /* post of module invariant */

    ensures (forall s0: ref :: SubObsH(s0,$Heap[s0,Subject.val],$Heap));
    ensures (forall p:ref, s0:ref ::
            IsGoodObject(p, $Heap) && IsGoodObject(s0, $Heap) &&
            $Heap[s0, Subject.O][p] && $Heap[p, Observer.sub] != null
            ==> $Heap[p, Observer.sub] == s0
        );

    // wr s.O\Dot nxt, <s>\Dot O
    // Since the only nxt field that is changed is "this" one, don't need to own
    // up to it here in the modifies clause.
    ensures
      (forall p: ref, f: name ::
        IsGoodObject(p,old($Heap)) ==>
```

42

```
            old($Heap)[p,f] == $Heap[p,f] || p == this ||
              (old($Heap)[s,Subject.O][p] && f == Observer.nxt) ||
              (p == s && f == Subject.O) ||
              (p == s && f == Subject.obs)        /* needed because of add */
        );
      free ensures NoDeallocs(old($Heap), $Heap);
      {
        $Heap[this,Observer.sub] := s; // observer takes responsibility of setting its sub field
        assume IsGoodHeap($Heap);
        $Heap[this,Observer.nxt] := null;
        assume IsGoodHeap($Heap);
        call Subject.register(s,this); // pre for this call is this.sub == s
      }


procedure Observer.notify(this: ref)
      free requires IsGoodObject(this,$Heap);
      // user-declared preconditions
      requires SubX($Heap[this,Observer.sub],$Heap[$Heap[this,Observer.sub],Subject.val],$Heap);
      /* pre of module invariant */

      requires (forall s: ref :: SubObsH(s,$Heap[s,Subject.val],$Heap));
      requires
        (forall p:ref, s:ref ::
          IsGoodObject(p, $Heap) && IsGoodObject(s, $Heap) &&
          $Heap[s, Subject.O][p] && $Heap[p, Observer.sub] != null
          ==> $Heap[p, Observer.sub] == s
        );

      modifies $Heap;
      // user-declared postconditions
      ensures SubX($Heap[this,Observer.sub],
                   $Heap[$Heap[this,Observer.sub],Subject.val],
                   $Heap);
      ensures Obs(this,$Heap[this,Observer.sub],
                  $Heap[$Heap[this,Observer.sub],Subject.val], $Heap);
      // wr <this>.\Dot cache
      ensures
        (forall p: ref, f: name :: { $Heap[p,f] }
          IsGoodObject(p,old($Heap)) ==>
          old($Heap[p,f]) == $Heap[p,f] ||
            (p == this && f == Observer.cache)
        );
      /* post of module invariant */

      ensures (forall s: ref :: SubObsH(s,$Heap[s,Subject.val],$Heap));
      ensures
        (forall p:ref, s:ref ::
          IsGoodObject(p, $Heap) && IsGoodObject(s, $Heap) &&
          $Heap[s, Subject.O][p] && $Heap[p, Observer.sub] != null
          ==> $Heap[p, Observer.sub] == s
        );
```

43

```
        free ensures NoDeallocs(old($Heap), $Heap);
        {
          var tmp : int;
          call tmp := Subject.get($Heap[this,Observer.sub]);
          $Heap[this,Observer.cache] := tmp;
          assume IsGoodHeap($Heap); // needed after each update to $Heap
        }


procedure Observer.val(this: ref) returns ($result : int)
        free requires IsGoodObject(this,$Heap);
        // user-declared preconditions
        requires SubObsX($Heap[this,Observer.sub],
                         $Heap[$Heap[this,Observer.sub],Subject.val],
                         $Heap);
        requires $Heap[$Heap[this,Observer.sub],Subject.O][this];
        /* pre of module invariant */

        requires (forall s: ref :: SubObsH(s,$Heap[s,Subject.val],$Heap));
        requires
          (forall p:ref, s:ref ::
            IsGoodObject(p, $Heap) && IsGoodObject(s, $Heap) &&
            $Heap[s, Subject.O][p] && $Heap[p, Observer.sub] != null
            ==> $Heap[p, Observer.sub] == s
          );


        // user-declared postconditions
        ensures SubObsX($Heap[this,Observer.sub],
                       $Heap[$Heap[this,Observer.sub],Subject.val],
                       $Heap);
        ensures $result == $Heap[$Heap[this,Observer.sub],Subject.val];
        /* post of module invariant */

        ensures (forall s: ref :: SubObsH(s,$Heap[s,Subject.val],$Heap));
        ensures
          (forall p:ref, s:ref ::
            IsGoodObject(p, $Heap) && IsGoodObject(s, $Heap) &&
            $Heap[s, Subject.O][p] && $Heap[p, Observer.sub] != null
            ==> $Heap[p, Observer.sub] == s
          );

        {
          $result := $Heap[this,Observer.cache];
        }
```

H Subject Observer and Client with External Specifications Only

```
/* Verification of Client using SubX and SubObsX (Section 6.1 of paper) */
/* Hidden module invariant is not used */

var $Heap: [ref, <x>name]x where IsGoodHeap($Heap);

// These are the fields of the class Subject
const unique Subject.obs: <ref>name;
const unique Subject.val: <int>name;
const unique Subject.O: <[ref]bool>name;

// These are the fields of the class Observer
const unique Observer.sub: <ref>name;
const unique Observer.cache: <int>name;
const unique Observer.nxt: <ref>name;

function SubX(s : ref, v : int, h: [ref,<x>name]x) returns (bool);
axiom (forall s : ref, v : int, h: [ref,<x>name]x :: { SubX(s,v,h) }
        SubX(s,v,h)  <==> (IsGoodObject(s,h) && h[s,Subject.val] == v)
      );

function Obs(o : ref, s : ref, v : int, h: [ref,<x>name]x) returns (bool);
axiom (forall o : ref, s : ref, v : int, h: [ref,<x>name]x :: { Obs(o,s,v,h) }
        Obs(o,s,v,h)  <==> IsGoodObject(o,h) && IsGoodObject(s,h) &&
                           h[o,Observer.cache] == v && h[o,Observer.sub] == s
      );

function SubObs(s : ref, v : int, h: [ref,<x>name]x) returns (bool);
axiom (forall s : ref, v : int, h: [ref,<x>name]x :: { SubObs(s,v,h) }
        SubObs(s,v,h) <==> Sub(s,v,h) &&
                           (forall o : ref :: IsGoodObject(o,h) && h[s,Subject.O][o]
                                              ==> Obs(o,s,v,h))
      );

axiom (forall h: [ref,<x>name]x, s: ref :: { IsGoodRegion(h[s, Subject.O], h) }
              IsGoodObject(s, h) ==> IsGoodRegion(h[s, Subject.O], h)
      );

axiom (forall h: [ref,<x>name]x, s: ref, o: ref :: { h[s, Subject.O][o] }
              IsGoodObject(s, h) ==> IsGoodRegion(h[s, Subject.O], h)
      );

procedure Subject..ctor(this: ref);
    free requires IsGoodObject(this,$Heap);
    modifies $Heap;
    // user-declared postconditions
    ensures SubObsX(this,0,$Heap);
    ensures EqualSet($Heap[this, Subject.O], EmptySet);
    // target object is allocated upon return
    free ensures $Heap[this, alloc];
```

```
      // wr <this>\Dot val, <this>\Dot obs, <this>\Dot O

      ensures
        (forall p: ref, f: name ::
          IsGoodObject(p,old($Heap)) ==>
          old($Heap[p,f]) == $Heap[p,f] || p == this
        );
      free ensures NoDeallocs(old($Heap), $Heap);


/* Subject.register and Subject.add are MODULE-SCOPED hence omitted*/


procedure Subject.update(this: ref, n: int);
      // user-declared preconditions
      free requires IsGoodObject(this, $Heap);
      requires SubObsX(this,$Heap[this,Subject.val],$Heap);
      modifies $Heap;
      // user-declared postconditions
      ensures SubObsX(this, n, $Heap);
      // wr <this>\Dot val, <this>.O\Dot cache

      ensures
        (forall p: ref, f: name ::
          IsGoodObject(p,old($Heap)) ==>
          old($Heap[p,f]) == $Heap[p,f] ||
            (p == this && f == Subject.val) ||
            ($Heap[this,Subject.O][p] && f == Observer.cache)
        );

      free ensures NoDeallocs(old($Heap), $Heap);



procedure Subject.get(this: ref) returns ($result : int);
      free requires IsGoodObject(this, $Heap);
      // user-declared preconditions
      requires SubX(this,$Heap[this,Subject.val],$Heap);
      // user-declared postconditions
      ensures SubX(this,$Heap[this,Subject.val],$Heap);
      ensures $result == $Heap[this,Subject.val];

procedure Observer..ctor(this: ref, s : ref);
      free requires IsGoodObject(s, $Heap);
      // needed because we are passing self to register
      free requires IsGoodObject(this,$Heap);
      requires (forall p : ref :: IsGoodObject(p, $Heap) ==> !$Heap[p, Subject.O][this]);

      requires SubX(s, $Heap[s,Subject.val], $Heap);
      modifies $Heap;
      // user-declared postconditions
      ensures SubX(s, $Heap[s,Subject.val], $Heap);
      ensures Obs(this, s, $Heap[s,Subject.val], $Heap);
```

```
        ensures $Heap[s,Subject.O][this];
        //needed for client
        ensures EqualSet(UnionSet(SingletonSet(this), old($Heap)[s,Subject.O]),
                         $Heap[s,Subject.O]) ;
        // wr s.O\Dot nxt, <s>\Dot O
        // Since the only nxt field that is changed is "this" one, don't need to own
        // up to it here in the modifies clause.
        ensures
          (forall p: ref, f: name ::
            IsGoodObject(p,old($Heap)) ==>
            old($Heap[p,f]) == $Heap[p,f] || p == this ||
              (old($Heap)[s,Subject.O][p] && f == Observer.nxt) ||
              (p == s && f == Subject.O) ||
              (p == s && f == Subject.obs)        /* for abstraction, Subject.any */
          );
        free ensures NoDeallocs(old($Heap), $Heap);


procedure Observer.notify(this: ref);
        free requires IsGoodObject(this,$Heap);
        // user-declared preconditions
        requires SubX($Heap[this,Observer.sub],
                      $Heap[$Heap[this,Observer.sub],Subject.val],
                      $Heap);
        modifies $Heap;
        // user-declared postconditions
        ensures SubX($Heap[this,Observer.sub],
                     $Heap[$Heap[this,Observer.sub],Subject.val],
                     $Heap);
        ensures Obs(this,$Heap[this,Observer.sub],
                    $Heap[$Heap[this,Observer.sub],Subject.val], $Heap);
        // wr <this>.\Dot cache

        ensures
          (forall p: ref, f: name :: { $Heap[p,f] }
            IsGoodObject(p,old($Heap)) ==>
            old($Heap[p,f]) == $Heap[p,f] ||
              (p == this && f == Observer.cache)
          );

        free ensures NoDeallocs(old($Heap), $Heap);


procedure Observer.val(this: ref) returns ($result : int);
        free requires IsGoodObject(this,$Heap);
        // user-declared preconditions
        requires SubObsX($Heap[this,Observer.sub],
                         $Heap[$Heap[this,Observer.sub],Subject.val],$Heap);
        requires $Heap[$Heap[this,Observer.sub],Subject.O][this];
        // user-declared postconditions
        ensures SubObsX($Heap[this,Observer.sub],
```

```
                          $Heap[$Heap[this,Observer.sub],Subject.val],
                          $Heap);
       ensures $result == $Heap[$Heap[this,Observer.sub],Subject.val];


   ////////////////////////////////////////////////////////////////////
   ///////////////        /////CLIENT/////////////////////////////////////
   ////////////////////////////////////////////////////////////////////
   procedure client()
       modifies $Heap;
       // user-declared postconditions
       // effects
     {
           var sub0 : ref;
           var obs0 : ref;
           var obs1 : ref;
           var sub  : ref;
           var obs  : ref;

       // sub0 = new Subject();
           havoc sub0;
           assume !$Heap[sub0,alloc];
           call Subject..ctor(sub0);

       // obs0 = new Observer(sub0);
           havoc obs0;
           assume !$Heap[obs0,alloc];
           assume EqualSet($Heap[obs0, Subject.O],  EmptySet);
           call Observer..ctor(obs0, sub0);

       // obs1 = new Observer(sub0);
           havoc obs1;
           assume !$Heap[obs1,alloc];
           assume EqualSet($Heap[obs1, Subject.O],  EmptySet);
           call Observer..ctor(obs1, sub0);

      //   sub  = new Subject();
           havoc sub;
           assume !$Heap[sub,alloc];
           call Subject..ctor(sub);

      //   obs = new Observer(sub);
           havoc obs;
           assume !$Heap[obs,alloc];
           assume EqualSet($Heap[obs, Subject.O],  EmptySet);
           call Observer..ctor(obs, sub);

           call Subject.update(sub0, 5);
           assert DisjointSet($Heap[sub0, Subject.O], $Heap[sub, Subject.O]);
           assert SubObsX(sub0, 5, $Heap);
           assert SubObsX(sub, 0, $Heap);
     }
```

I   Multiple Subject Observer

```
// Multiple Subject Observer: an observer can observe multiple subjects
var $Heap: [ref, <x>name]x where IsGoodHeap($Heap);

// These are the fields of the class Node
const unique Node.info: <ref>name; // reference to an Observer
const unique Node.nxt: <ref>name;

// These are the fields of the class Subject
const unique Subject.obs: <ref>name; // pointer to the first Node
       //holding the list of observers
const unique Subject.val: <int>name;
const unique Subject.O: <[ref]bool>name;

// These are the fields of the class Observer
// Observer.caches is now an array
// maps a subject to the cached value for that subject
const unique Observer.caches: <[ref]int>name;

// In Spec#, these would be from the translation of [Pure] methods
// written as part of the program
function List2observers(o : ref, h: [ref,<x>name]x) returns ([ref]bool);
axiom (forall o : ref, h: [ref,<x>name]x :: { List2observers(o,h) }
        (o == null <==> EqualSet(List2observers(o,h),EmptySet))
      );


axiom (forall o : ref, h: [ref,<x>name]x, r: [ref]bool ::
        { EqualSet(List2observers(o,h), r) }
        (o != null ==> EqualSet(List2observers(o,h),
                               UnionSet(SingletonSet(h[o,Node.info]),
                                        List2observers(h[o,Node.nxt],h))))
      );

axiom (forall H: [ref,<x>name]x, K: [ref,<x>name]x, o : ref ::
         { List2observers(o,H), List2observers(o,K) }
         (forall o2 : ref  ::
            o2 != null && List2observers(o,H)[o2] && List2observers(o,K)[o2]
           ==> H[o2,Observer.nxt] == K[o2,Observer.nxt])
         ==>
         EqualSet(List2observers(o,H), List2observers(o,K))
      );

function Sub(s : ref, v : int, h: [ref,<x>name]x) returns (bool);
axiom (forall s : ref, v : int, h: [ref,<x>name]x :: { Sub(s,v,h) }
        Sub(s,v,h)  <==> (IsGoodObject(s,h) && h[s,Subject.val] == v &&
                          EqualSet(List2observers(h[s,Subject.obs],h),
                                   h[s,Subject.O]))
      );

function Obs(o : ref, s : ref, v : int, h: [ref,<x>name]x) returns (bool);
```

```
axiom (forall o : ref, s : ref, v : int, h: [ref,<x>name]x :: { Obs(o,s,v,h) }
        Obs(o,s,v,h)  <==> IsGoodObject(o,h) && IsGoodObject(s,h) &&
                            h[o,Observer.caches][s] == v
      );


function SubObs(s : ref, v : int, h: [ref,<x>name]x) returns (bool);
axiom (forall s : ref, v : int, h: [ref,<x>name]x :: { SubObs(s,v,h) }
        SubObs(s,v,h) <==> Sub(s,v,h) &&
                            (forall o : ref :: IsGoodObject(o,h) && h[s,Subject.O][o]
                             ==> Obs(o,s,v,h))
      );


axiom (forall h: [ref,<x>name]x, s: ref :: { IsGoodRegion(h[s, Subject.O], h) }
              IsGoodObject(s, h) ==> IsGoodRegion(h[s, Subject.O], h)
      );


axiom (forall h: [ref,<x>name]x, s: ref, o: ref :: { h[s, Subject.O][o] }
              IsGoodObject(s, h) ==> IsGoodRegion(h[s, Subject.O], h)
      );


// NODE
procedure Node..ctor(this: ref, element: ref where IsGoodObject(element,$Heap));
  free requires IsGoodHeap($Heap);
  modifies $Heap;
  // user-declared postconditions
  ensures $Heap[this, Node.nxt] == null;
  ensures $Heap[this, Node.info] == element;
  // target object is allocated upon return
  free ensures IsGoodObject(this, $Heap);
  ensures
      (forall p: ref, f: name ::
        old($Heap)[p, alloc] ==> old($Heap[p,f]) == $Heap[p,f] || p == this
      );
  free ensures NoDeallocs(old($Heap),$Heap);
// END OF NODE


// SUBJECT
procedure Subject..ctor(this: ref)
    free requires IsGoodObject(this,$Heap);
    modifies $Heap;
    // user-declared postconditions
    ensures SubObs(this,0,$Heap);
    ensures EqualSet($Heap[this, Subject.O], EmptySet);
    // target object is allocated upon return
    free ensures $Heap[this, alloc];
    // wr <this>\Dot val, <this>\Dot obs, <this>\Dot O
    ensures
      (forall p: ref, f: name ::
        IsGoodObject(p,old($Heap)) ==>
        old($Heap[p,f]) == $Heap[p,f] || p == this
      );
```

```
      free ensures NoDeallocs(old($Heap), $Heap);


   {
      $Heap[this,Subject.obs] := null;
      assume IsGoodHeap($Heap);
      $Heap[this,Subject.val] := 0;
      assume IsGoodHeap($Heap);
      $Heap[this,Subject.O] := EmptySet;
      assume IsGoodHeap($Heap);
   }


procedure Subject.register(this: ref where IsGoodObject(this, $Heap),
                           o: ref where IsGoodObject(o, $Heap))
   // user-declared preconditions
   // to prevent o from being added to the list twice */
   requires !$Heap[this, Subject.O][o];
   requires Sub(this, $Heap[this,Subject.val], $Heap);
   modifies $Heap;

   // user-declared postconditions

   ensures Sub(this, $Heap[this,Subject.val], $Heap);
   ensures Obs(o,this,$Heap[this,Subject.val],$Heap);
   ensures $Heap[this,Subject.O][o];
   ensures EqualSet(UnionSet(SingletonSet(o),old($Heap)[this,Subject.O]),
                    $Heap[this,Subject.O]) ;

   // wr <this>\Dot O, (<o>\Dot caches)[this], <this>\Dot obs, wr all
   ensures
     (forall p: ref, f: name ::
       IsGoodObject(p,old($Heap)) ==>
       old($Heap[p,f]) == $Heap[p,f] ||
         (p == this && f == Subject.O) ||
         (p == o && f == Observer.caches) ||
         (p == this && f == Subject.obs)
     );
   // Part of "wr (<o>\Dot caches)[this]":
   // no other cache line has changed except for "this" one.
   ensures (forall s : ref ::
             IsGoodObject(s,$Heap) && s != this
               ==>
             $Heap[o,Observer.caches][s] == old($Heap)[o,Observer.caches][s]
           );
   free ensures NoDeallocs(old($Heap), $Heap);
 {
   call Subject.add(this,o);
   call Observer.notify(o, this);
 }

procedure Subject.update(this: ref, n: int)
   // user-declared preconditions
```

```
        free requires IsGoodObject(this, $Heap);
        requires SubObs(this,$Heap[this,Subject.val],$Heap);
        modifies $Heap;

        // user-declared postconditions
        ensures SubObs(this, n, $Heap);
        // wr <this>\Dot val, (<this>.O\Dot caches)[this]
        ensures
          (forall p: ref, f: name ::
            IsGoodObject(p,old($Heap)) ==>
            old($Heap[p,f]) == $Heap[p,f] ||
              (p == this && f == Subject.val) ||
              (old($Heap)[this,Subject.O][p] && f == Observer.caches)
          );
        free ensures NoDeallocs(old($Heap), $Heap);
    {
        var node : ref where (node == null) || IsGoodObject(node, $Heap);
        var r1 : [ref]bool where IsGoodRegion(r1, $Heap);

        $Heap[this,Subject.val] := n;
        assume IsGoodHeap($Heap);
        node := $Heap[this,Subject.obs];
        r1 := EmptySet;
        while (node != null)
          invariant Sub(this,n,$Heap);

          /* the following three invariants are essential */
          invariant SubSet(r1, $Heap[this, Subject.O]);
          invariant (forall p : ref :: IsGoodObject(p, $Heap) && r1[p]
                          ==> Obs(p, this, n, $Heap)); /* notified observers are uptodate */
          invariant EqualSet(UnionSet(List2Observers(node, $Heap), r1), $Heap[this, Subject.O]);

          /* translation needs to generate the following inv. automatically */

          invariant (forall p: ref, f: name ::
                          { $Heap[p,f] }
                          IsGoodObject(p,old($Heap)) ==>
                              old($Heap)[p,f] == $Heap[p,f] ||
                              (p == this && f == Subject.val) ||
                              (old($Heap)[this,Subject.O][p] && f == Observer.caches)
                      );
        {
          call Observer.notify($Heap[node,Node.info], this);
          r1 := UnionSet(r1, SingletonSet($Heap[node,Node.info]));
          node := $Heap[node,Node.nxt];
        }
    }

procedure Subject.get(this: ref) returns ($result : int)
        free requires IsGoodObject(this, $Heap);
        // user-declared preconditions
```

```
      requires Sub(this,$Heap[this,Subject.val],$Heap);
      // user-declared postconditions
      ensures Sub(this,$Heap[this,Subject.val],$Heap);
      ensures $result == $Heap[this,Subject.val];
      free ensures NoDeallocs(old($Heap), $Heap);
  {
      $result := $Heap[this,Subject.val];
  }


procedure Subject.add(this: ref where IsGoodObject(this, $Heap),
                      o: ref where IsGoodObject(o, $Heap))
    // user-declared preconditions
    requires !$Heap[this, Subject.O][o];
    requires Sub(this,$Heap[this,Subject.val],$Heap);

    modifies $Heap;

    // user-declared postconditions
    ensures Sub(this, $Heap[this,Subject.val], $Heap);
    ensures $Heap[this, Subject.O][o];

    ensures EqualSet(UnionSet(SingletonSet(o),old($Heap)[this,Subject.O]),
                     $Heap[this,Subject.O]) ;

    // wr <this>\Dot O, <this>\Dot obs, wr all
    ensures
      (forall p: ref, f: name ::
        IsGoodObject(p,old($Heap)) ==>
        old($Heap[p,f]) == $Heap[p,f] ||
          (p == this && f == Subject.O) ||
          (p == this && f == Subject.obs)
      );
    free ensures NoDeallocs(old($Heap), $Heap);
  {

    var n : ref;
    $Heap[this,Subject.O] := UnionSet(SingletonSet(o),$Heap[this,Subject.O]);
    assume IsGoodHeap($Heap);

    // Create a new Node
    // n := new Node(o)
      havoc n;
      assume !$Heap[n,alloc];
      call Node..ctor(n,o); // after this call IsGoodObject(n) holds

    // Stick it on the front of the list of observers
    $Heap[n,Node.nxt] := $Heap[this,Subject.obs];
    assume IsGoodHeap($Heap);
    $Heap[this,Subject.obs] := n;
    assume IsGoodHeap($Heap);
```

```
  }
// END OF SUBJECT

// OBSERVER
procedure Observer..ctor(this: ref where IsGoodObject(s, $Heap),
                         s : ref where IsGoodObject(this, $Heap))
  // can we relax the following?
  requires (forall p : ref :: IsGoodObject(p, $Heap) ==> !$Heap[p, Subject.O][this]);
  requires Sub(s, $Heap[s,Subject.val], $Heap);
  modifies $Heap;
  // user-declared postconditions
  ensures Sub(s, $Heap[s,Subject.val], $Heap);
  ensures Obs(this, s, $Heap[s,Subject.val], $Heap);

  ensures $Heap[s, Subject.O][this];
  ensures EqualSet(UnionSet(SingletonSet(this), old($Heap)[s, Subject.O]),
                   $Heap[s, Subject.O]) ;
  // target object is allocated upon return
  free ensures $Heap[this, alloc]; /* needed for callers of Observer..ctor */
  // <s>\Dot O, wr all // "wr all" is because Subject.register is called
  ensures
    (forall p: ref, f: name ::
       IsGoodObject(p,old($Heap)) ==>
       old($Heap[p,f]) == $Heap[p,f] || p == this ||
        (p == s && f == Subject.O) ||
        (p == s && f == Subject.obs)          /* needed because of add */
    );
  free ensures NoDeallocs(old($Heap), $Heap);
  {
    call Subject.register(s, this);
  }

procedure Observer.notify(this: ref where IsGoodObject(this, $Heap),
                          s: ref where IsGoodObject(s, $Heap))
    // user-declared preconditions
    requires $Heap[s, Subject.O][this];
    requires Sub(s, $Heap[s, Subject.val], $Heap);
    modifies $Heap;

    // user-declared postconditions
    ensures Sub(s, $Heap[s, Subject.val], $Heap);
    ensures Obs(this, s, $Heap[s, Subject.val], $Heap);
    // wr <this>.\Dot caches[s]
    ensures
      (forall p: ref, f: name :: { $Heap[p,f] }
        IsGoodObject(p,old($Heap)) ==>
        old($Heap[p,f]) == $Heap[p,f] ||
           (p == this && f == Observer.caches)
      );

    ensures
```

54

```
      (forall otherSubject: ref ::
         otherSubject != s ==> $Heap[this, Observer.caches][otherSubject] ==
                                 old($Heap)[this, Observer.caches][otherSubject]
      );

    free ensures NoDeallocs(old($Heap), $Heap);

  {
    var tmp : int;
    var localCaches : [ref]int;
    call tmp := Subject.get(s);
    localCaches := $Heap[this,Observer.caches];
    localCaches[s] := tmp;
    $Heap[this, Observer.caches] := localCaches;
    assume IsGoodHeap($Heap); // needed after each update to $Heap
  }

procedure Observer.val(this: ref where IsGoodObject(this, $Heap),
                       s: ref where IsGoodObject(s, $Heap)) returns ($result : int)
    // user-declared preconditions
    requires SubObs(s, $Heap[s, Subject.val], $Heap);
    requires $Heap[s, Subject.O][this];
    // user-declared postconditions
    ensures SubObs(s, $Heap[s, Subject.val], $Heap);
    ensures $result == $Heap[s, Subject.val];
    free ensures NoDeallocs(old($Heap), $Heap);
  {
    $result := $Heap[this,Observer.caches][s];
  }
// END OF OBSERVER
```

J A Client for Multiple Subject Observer

```
//////////////////////////////////////////////////////////////
// In the following client, sub0's observer is obs0 and sub1's observers
// are obs1 and obs2; sub0 is first updated;*/
// Then obs2 is registered as sub1's observer.
//////////////////////////////////////////////////////////////

procedure client()
    modifies $Heap;
    // user-declared postconditions
    // effects
  {
        var sub0 : ref;
        var sub1 : ref;
        var obs0 : ref;
        var obs1 : ref;
        var obs2 : ref;
        var m    : int;
        var n    : int;
        var tmp  : int;

    //sub0 = new Subject();
        havoc sub0;
        assume !$Heap[sub0,alloc];
        call Subject..ctor(sub0);

    //obs0 = new Observer();
        havoc obs0;
        assume !$Heap[obs0,alloc];
        // this is here because we are not using a type system
        assume EqualSet($Heap[obs0, Subject.O],  EmptySet);
        // obs0 is registered with sub0
        call Observer..ctor(obs0, sub0);

    //sub1  = new Subject();
        havoc sub1;
        assume !$Heap[sub1,alloc];
        call Subject..ctor(sub1);

    //obs1 = new Observer();
        havoc obs1;
        assume !$Heap[obs1,alloc];
        assume EqualSet($Heap[obs1, Subject.O],  EmptySet);
        // obs1 is registered with sub1
        call Observer..ctor(obs1, sub1);

    //obs2 = new Observer();
        havoc obs2;
        assume !$Heap[obs2,alloc];
        assume EqualSet($Heap[obs2, Subject.O],  EmptySet);
```

```
        // obs2 is registered with sub1
        call Observer..ctor(obs2, sub1);

    tmp := $Heap[obs1, Observer.caches][sub0];
    call Subject.update(sub0, 5);
    assert $Heap[obs1, Observer.caches][sub0] == tmp;

    assert SubObs(sub0, 5, $Heap);

    assert $Heap[obs0, Observer.caches][sub0] == 5;

    assert SubObs(sub1, 0, $Heap);
    // now obs0 also starts observing sub1
    call Subject.register(sub1, obs0);

    call m := Observer.val(obs0, sub1);

    assert m == 0;

    call n := Observer.val(obs0, sub0);

    assert n == 5;

  }
////////////////////////////////////////////////////////////////////
// In the following client sub0's observers are obs0 and obs1;
// sub1's observer is obs2; sub0 is updated
////////////////////////////////////////////////////////////////////
procedure client0()
    modifies $Heap;
    // user-declared postconditions
    // effects
  {
        var sub0 : ref;
        var obs0 : ref;
        var obs1 : ref;
        var sub1 : ref;
        var obs2 : ref;
        var   n1 : int;

    // sub0 = new Subject();
        havoc sub0;
        assume !$Heap[sub0,alloc];
        call Subject..ctor(sub0);

    // obs0 = new Observer();
        havoc obs0;
        assume !$Heap[obs0,alloc];
        assume EqualSet($Heap[obs0, Subject.O],  EmptySet);
        // obs0 is registered with sub0
        call Observer..ctor(obs0, sub0);
```

```
  // obs1 = new Observer();
     havoc obs1;
     assume !$Heap[obs1,alloc];
     assume EqualSet($Heap[obs1, Subject.O],  EmptySet);
     // obs1 is registered with sub0
     call Observer..ctor(obs1, sub0);

//   sub1  = new Subject();
     havoc sub1;
     assume !$Heap[sub1,alloc];
     call Subject..ctor(sub1);

 // obs2 = new Observer();
     havoc obs2;
     assume !$Heap[obs2,alloc];
     assume EqualSet($Heap[obs2, Subject.O],  EmptySet);
     // obs2 is registered with sub1
     call Observer..ctor(obs2, sub1);

     call Subject.update(sub0, 5);

     assert SubObs(sub0, 5, $Heap);
     assert SubObs(sub1, 0, $Heap);

     call n1 := Observer.val(obs2, sub1);
     assert n1 == 0;
}
```

K Ownership: ArrayList and Stack

K.1 ArrayList Declarations

```
const unique ArrayList.footprint: <[ref]bool>name;

procedure ArrayList..ctor(this: ref);
  free requires IsGoodObject(this,Heap);
  modifies Heap;
  // user-declared postconditions
  ensures #ArrayList.size(Heap,this) == 0;
  // fresh(this.footprint - { this });
  ensures (forall p: ref:: Heap[this,ArrayList.footprint][p]
            ==> (p == this || !old(Heap)[p,alloc]));

  // target object is allocated upon return
  free ensures IsGoodObject(this, Heap);
  // frame condition
  ensures
      (forall p: ref, f: name ::
        old(Heap)[p, alloc] ==> old(Heap[p,f]) == Heap[p,f] || p == this
      );
  ensures ArrayList.INV(Heap,this);
  free ensures NoDeallocs(old(Heap),Heap);

function ArrayList.INV(h: [ref, <x>name]x, o: ref) returns (bool);

function #ArrayList.size(h: [ref, <x>name]x, o: ref) returns (int);

procedure ArrayList.Add(this: ref, o: any);
  free requires IsGoodObject(this,Heap);
  requires ArrayList.INV(Heap,this);
  modifies Heap;
  ensures ArrayList.INV(Heap,this);
  ensures #ArrayList.size(Heap,this) == #ArrayList.size(old(Heap),this) + 1;
  ensures #ArrayList.get(Heap,this,#ArrayList.size(Heap,this) - 1) == o;
  ensures (forall i:int ::
            0 <= i && i < #ArrayList.size(Heap,this) - 1
              ==>
            #ArrayList.get(Heap,this,i) == old(#ArrayList.get(Heap,this,i)));

  // frame condition: "modifies footprint"
  ensures
      (forall p: ref, f: name ::
        old(Heap)[p, alloc] ==>
         old(Heap[p,f]) == Heap[p,f] // everything remains unchanged
         || old(Heap)[this,ArrayList.footprint][p] // unless it is
        //an object contained in self's footprint
      );
  ensures DifferenceIsNew(old(Heap),
                          old(Heap)[this,ArrayList.footprint],
                          Heap[this, ArrayList.footprint]);
```

59

```
  free ensures NoDeallocs(old(Heap),Heap);
  free ensures IsGoodHeap(Heap);

function #ArrayList.get(h: [ref, <x>name]x, o: ref, i : int) returns (any);

procedure ArrayList.Get(this: ref, index: int) returns (result: any);
  requires ArrayList.INV(Heap,this);
  requires 0 <= index && index < #ArrayList.size(Heap,this);
  ensures result == #ArrayList.get(Heap,this,index);
```

K.2   ArrayList Implementation

```
//
// Private fields, axioms for functions, and implementations of procedures
//

const unique ArrayList.count: <int>name;
const unique ArrayList.items: <[int]any>name;

// function ArrayList.INV(h: [ref, <x>name]x, o: ref) returns (bool);
// INV depends only on the things in its footprint
axiom
  (forall H: [ref, <x>name]x, o: ref
    :: { ArrayList.INV(H,o) }
      (
        IsGoodObject(o, H)
        && 0 <= H[o,ArrayList.count]
        && H[o,ArrayList.count] <= #ArrayList.size(H,o)
        && (forall p: ref :: H[o,ArrayList.footprint][p] <==> p == o)
      )
      <==>
      ArrayList.INV(H,o)
  );

// #ArrayList.size(h: [ref, <x>name]x, o: ref) returns (int);
axiom
  (forall h: [ref, <x>name]x, o: ref
    :: { #ArrayList.size(h,o) }
   IsGoodObject(o, h)
      ==>
   #ArrayList.size(h,o) == h[o,ArrayList.count]
  );

implementation ArrayList.Add(this: ref, o: any)
{
  var newValue : [int]any;
  assume (forall i: int ::
          (i != Heap[this,ArrayList.count]
           ==> newValue[i] == Heap[this, ArrayList.items][i])
          &&
```

```
          (i == Heap[this,ArrayList.count]
           ==> newValue[i] == o)
        );
  Heap[this, ArrayList.items] := newValue;
  assume IsGoodHeap(Heap);
  Heap[this,ArrayList.count] := Heap[this,ArrayList.count] + 1;
  assume IsGoodHeap(Heap);
}


// function #ArrayList.get(h: [ref, <x>name]x, o: ref, i : int) returns (any);
axiom
  (forall h: [ref, <x>name]x, o: ref, i: int
    :: { #ArrayList.get(h,o,i) }
   IsGoodObject(o, h) && 0 <= i && i < #ArrayList.size(h,o)
     ==>
   #ArrayList.get(h,o,i) == h[o,ArrayList.items][i]
  );


implementation ArrayList.Get(this: ref, index: int) returns (result: any)
{
  result := Heap[this, ArrayList.items][index];
}
```

K.3   Array List Client Axioms

```
//
// Axioms for client view of functions
//


// function ArrayList.INV(h: [ref, <x>name]x, o: ref) returns (bool);
// INV depends only on the things in its footprint
axiom
  (forall H: [ref, <x>name]x, K: [ref, <x>name]x, o: ref
    :: { ArrayList.INV(H,o), ArrayList.INV(K,o) }

     (
       IsGoodObject(o, H) && IsGoodObject(o,K)
       && H[o,ArrayList.footprint] == K[o,ArrayList.footprint]
       && (forall p : ref, f: name ::
            IsGoodObject(p,H) && IsGoodObject(p,K) && H[o,ArrayList.footprint][p]
            ==>  H[p,f] == K[p,f])
     )
     ==>
     ArrayList.INV(H,o) == ArrayList.INV(K,o)
  );
axiom
  (forall H: [ref, <x>name]x, o: ref
    :: { ArrayList.INV(H,o) }

     ArrayList.INV(H,o)
```

```
      ==>
      (forall p: ref :: H[o,ArrayList.footprint][p] ==> IsGoodObject(p,H))
  );


// #ArrayList.size(h: [ref, <x>name]x, o: ref) returns (int);
axiom
  (forall H: [ref, <x>name]x, K: [ref, <x>name]x, o: ref
    :: { #ArrayList.size(H,o), #ArrayList.size(K,o) }

      (
        IsGoodObject(o, H) && IsGoodObject(o,K)
        && H[o,ArrayList.footprint] == K[o,ArrayList.footprint]
        && (forall p : ref, f: name ::
             IsGoodObject(p,H) && IsGoodObject(p,K) && H[o,ArrayList.footprint][p]
             ==>  H[p,f] == K[p,f])
      )
      ==>
      #ArrayList.size(H,o) == #ArrayList.size(K,o)
  );


// function #ArrayList.get(h: [ref, <x>name]x, o: ref, i: int) returns (any);
axiom
  (forall H: [ref, <x>name]x, K: [ref, <x>name]x, o: ref, i: int
    :: { #ArrayList.get(H,o,i), #ArrayList.get(K,o,i) }

      (
        IsGoodObject(o, H) && IsGoodObject(o,K)
        && H[o,ArrayList.footprint] == K[o,ArrayList.footprint]
        && (forall p : ref, f: name ::
             IsGoodObject(p,H) && IsGoodObject(p,K) && H[o,ArrayList.footprint][p]
             ==>  H[p,f] == K[p,f])
      )
      ==>
      #ArrayList.get(H,o,i) == #ArrayList.get(K,o,i)
  );
```

K.4   Array List Client

```
procedure ArrayList.Client() returns ()
  modifies Heap;
{
  var a1: ref;
  var a2: ref;
  var i : int;
  var tmp: any;

  // a1 = new ArrayList();
    havoc a1;
    assume !Heap[a1,alloc];
    call ArrayList..ctor(a1);
```

```
  // a2 = new ArrayList();
    havoc a2;
    assume !Heap[a2,alloc];
    call ArrayList..ctor(a2);

  call ArrayList.Add(a1, 3);

  call tmp := ArrayList.Get(a1,0);
  i := cast(tmp,int);
  assert i == 3;

  call ArrayList.Add(a2,5);

  call tmp := ArrayList.Get(a1,0);
  i := cast(tmp,int);
  assert i == 3;

  call ArrayList.Add(a1, 5);

  call tmp := ArrayList.Get(a1,0);
  i := cast(tmp,int);
  assert i == 3;

  call tmp := ArrayList.Get(a1,1);
  i := cast(tmp,int);
  assert i == 5;

  call tmp := ArrayList.Get(a2,0);
  i := cast(tmp,int);
  assert i == 5;
}
```

K.5   Stack Declarations

```
// Stack<T>

const unique Stack.footprint: <[ref]bool>name;

procedure Stack..ctor(this: ref);
  free requires IsGoodObject(this,Heap);
  modifies Heap;
  // user-declared postconditions
  ensures #Stack.size(Heap,this) == 0;
  // fresh(this.footprint - { this });
  ensures (forall p: ref:: Heap[this,Stack.footprint][p]
             ==> (p == this || !old(Heap)[p,alloc]));

  // target object is allocated upon return
  free ensures IsGoodObject(this, Heap);
  ensures
      (forall p: ref, f: name ::
        old(Heap)[p, alloc] ==> old(Heap[p,f]) == Heap[p,f] || p == this
      );
  ensures Stack.INV(Heap,this);
  free ensures NoDeallocs(old(Heap),Heap);


function Stack.INV(h: [ref, <x>name]x, o: ref) returns (bool);

function #Stack.size(h: [ref, <x>name]x, o: ref) returns (int);

procedure Stack.Push(this: ref, o : int);
  free requires IsGoodObject(this,Heap);
  requires Stack.INV(Heap,this);
  modifies Heap;
  ensures Stack.INV(Heap,this);
  ensures #Stack.size(Heap,this) == #Stack.size(old(Heap),this) + 1;
  // frame condition from "modifies footprint"
  ensures
      (forall p: ref, f: name ::
        old(Heap)[p, alloc] ==>
         old(Heap[p,f]) == Heap[p,f] // everything remains unchanged
         || old(Heap)[this,Stack.footprint][p] // or it is an object
         // contained in self's footprint
      );
  ensures DifferenceIsNew(old(Heap),
                          old(Heap)[this, Stack.footprint],
                          Heap[this, Stack.footprint]);
  free ensures NoDeallocs(old(Heap),Heap);

procedure Stack.Pop(this: ref) returns (result: int);
  requires Stack.INV(Heap,this);
  modifies Heap;
  ensures Stack.INV(Heap,this);
```

```
    ensures #Stack.size(Heap,this) == #Stack.size(old(Heap),this) - 1;
//  ensures result == old(contents.get(contents.size()-1));


procedure Stack.Empty(this: ref) returns (result: bool);
  requires Stack.INV(Heap,this);
  ensures result == (#Stack.size(Heap,this) == 0);


procedure Stack.Switch(this: ref, other: ref);
  free requires IsGoodObject(this, Heap);
  free requires IsGoodObject(other, Heap);

  // user-declared preconditions
  requires Stack.INV(Heap, other);
  requires Stack.INV(Heap,this);

  // Note that this *cannot* be replaced by "this != other". Just knowing that
  // the two stacks are different is not equivalent to knowing that somehow
  // one of them isn't stuck inside the other's footprint.
  requires DisjointSet(Heap[this,Stack.footprint],Heap[other,Stack.footprint]);

  modifies Heap;

  ensures Stack.INV(Heap,this);
  ensures Stack.INV(Heap,other);
  ensures #Stack.size(Heap,this) == #Stack.size(old(Heap), other);
  ensures #Stack.size(Heap,other) == #Stack.size(old(Heap), this);
  ensures DisjointSet(Heap[this,Stack.footprint],Heap[other,Stack.footprint]);
  ensures DifferenceIsNew(old(Heap),
                          UnionSet(old(Heap)[this,Stack.footprint],
                                   old(Heap)[other,Stack.footprint]),
                          UnionSet(Heap[this,Stack.footprint],
                                   Heap[other,Stack.footprint]));
  // frame condition from "modifies footprint union other.footprint"
  ensures
      (forall p: ref, f: name ::
        old(Heap)[p, alloc] ==>
         old(Heap[p,f]) == Heap[p,f] // everything remains unchanged
         || old(Heap)[this,Stack.footprint][p] // or it is an object contained
            //in self's footprint
         || old(Heap)[other,Stack.footprint][p] // or it is
            //in other's footprint
      );
  free ensures NoDeallocs(old(Heap),Heap);
```

K.6 Stack Implementation

```
//
// Private fields, axioms for functions, and implementations of procedures
//

const unique Stack.contents: <ref>name; // ArrayList<T>
```

```
implementation Stack..ctor(this: ref)
{
  var a: ref;
  // a = new ArrayList();
    havoc a;
    assume !Heap[a,alloc];
    call ArrayList..ctor(a);

  Heap[this, Stack.contents] := a;

  // Need to manually update the footprint
  Heap[this, Stack.footprint] := Heap[a, ArrayList.footprint];
  Heap[this, Stack.footprint] := UnionSet(Heap[this, Stack.footprint],
                                          SingletonSet(a));
  Heap[this, Stack.footprint] := UnionSet(Heap[this, Stack.footprint],
                                          SingletonSet(this));
  assume IsGoodHeap(Heap); // necessary
}

// function Stack.INV(h: [ref, <x>name]x, o: ref) returns (bool);
// INV depends only on the invariants of things in its footprint
axiom
  (forall H: [ref, <x>name]x, o: ref
    :: { Stack.INV(H,o) }

   IsGoodObject(o, H) && ArrayList.INV(H,H[o,Stack.contents])
     && H[o, Stack.footprint][o]
     && H[o, Stack.footprint][H[o, Stack.contents]]
     && SubSet(H[H[o,Stack.contents],ArrayList.footprint], H[o,Stack.footprint])
     // This next clause allows the stack to be verified
     // relative to the ArrayList client axioms,
     // not internal axioms
     && !H[H[o, Stack.contents], ArrayList.footprint][o]
     && H[o, Stack.contents] != o // needed because we don't have
        //the types encoded (see Switch)
     && IsGoodObject(H[o, Stack.contents],H)
     <==>
   Stack.INV(H,o)
  );

// function #Stack.size(h: [ref, <x>name]x, o: ref) returns (int);
axiom
  (forall h: [ref, <x>name]x, o: ref
    :: { #Stack.size(h,o) }
   IsGoodObject(o, h)
     ==>
   #Stack.size(h,o) == #ArrayList.size(h,h[o,Stack.contents])
  );

implementation Stack.Push(this: ref, o : int)
```

```
{
  call ArrayList.Add(Heap[this,Stack.contents],o);
  // Need to maintain the relationship as part of stack's invariant and since
  // footprint is a field, need to manually update it to keep it a superset of
  // the arraylist's footprint since ArrayList.Add is allowed to grow its footprint.
  Heap[this, Stack.footprint] := UnionSet(Heap[this,Stack.footprint],
                                          Heap[Heap[this,Stack.contents],
                                               ArrayList.footprint]);
  assume IsGoodHeap(Heap); // necessary
}


// implementation Stack.Pop(this: ref) returns (result: int)

// implementation Stack.Empty(this: ref) returns (result: bool)

implementation Stack.Switch(this: ref, other: ref)
{
  var tmp: ref;
  var tmpFootprint: [ref]bool;

  tmp := Heap[this,Stack.contents];
  Heap[this,Stack.contents] := Heap[other,Stack.contents];
  Heap[other,Stack.contents] := tmp;

  // Since the footprint is a field, need to manually switch the footprints
  Heap[this,Stack.footprint] := UnionSet(Heap[Heap[this,Stack.contents],
                                              ArrayList.footprint],
                                         SingletonSet(this));
  Heap[this,Stack.footprint] := UnionSet(Heap[this,Stack.footprint],
                                         SingletonSet(Heap[this,Stack.contents]));
  Heap[other,Stack.footprint] := UnionSet(Heap[Heap[other,Stack.contents],
                                              ArrayList.footprint],
                                          SingletonSet(other));
  Heap[other,Stack.footprint] := UnionSet(Heap[other,Stack.footprint],
                                          SingletonSet(Heap[other,Stack.contents]));

  assume IsGoodHeap(Heap); // necessary

}
```

K.7   Stack Client Axioms

```
//
// Axioms for client view of functions
//

// function Stack.INV(h: [ref, <x>name]x, o: ref) returns (bool);
// INV depends only on the things in its footprint
axiom
  (forall H: [ref, <x>name]x, K: [ref, <x>name]x, o: ref
    :: { Stack.INV(H,o), Stack.INV(K,o) }
```

```
      (
        IsGoodObject(o, H) && IsGoodObject(o,K)
        && H[o,Stack.footprint] == K[o,Stack.footprint]
        && (forall p : ref, f: name ::
              IsGoodObject(p,H) && IsGoodObject(p,K) && H[o,Stack.footprint][p]
              ==>  H[p,f] == K[p,f])
      )
      ==>
      Stack.INV(H,o) == Stack.INV(K,o)
  );


axiom
  (forall H: [ref, <x>name]x, o: ref
    :: { Stack.INV(H,o) }

      Stack.INV(H,o)
      ==>
      (forall p: ref :: H[o,Stack.footprint][p] ==> IsGoodObject(p,H))
  );

// #Stack.size(h: [ref, <x>name]x, o: ref) returns (int);
axiom
  (forall H: [ref, <x>name]x, K: [ref, <x>name]x, o: ref
    :: { #Stack.size(H,o), #Stack.size(K,o) }

      (
        IsGoodObject(o, H) && IsGoodObject(o,K)
        && H[o,Stack.footprint] == K[o,Stack.footprint]
        && (forall p : ref, f: name ::
              IsGoodObject(p,H) && IsGoodObject(p,K) && H[o,Stack.footprint][p]
              ==>  H[p,f] == K[p,f])
      )
      ==>
      #Stack.size(H,o) == #Stack.size(K,o)
  );
```

K.8   Stack Client

```
procedure Stack.Client() returns ()
  modifies Heap;
{
    var s1: ref;
    var s2: ref;
    var i : int;

  // s1 = new Stack();
    havoc s1;
    assume !Heap[s1,alloc];
    call Stack..ctor(s1);
```

```
assert #Stack.size(Heap,s1) == 0;

  // s2 = new Stack();
    havoc s2;
    assume !Heap[s2,alloc];
    call Stack..ctor(s2);
assert #Stack.size(Heap,s2) == 0;
assert #Stack.size(Heap,s1) == 0;

  call Stack.Push(s1,5);
assert #Stack.size(Heap,s2) == 0;
assert #Stack.size(Heap,s1) == 1;

  call i := Stack.Pop(s1);
assert #Stack.size(Heap,s1) == 0;
//  assert i == 5; // can't assert this because Stack's specs aren't functionally complete
}

procedure Stack.Client2() returns ()
  modifies Heap;
{
    var s1: ref;
    var s2: ref;
    var i : int;

  // s1 = new Stack();
    havoc s1;
    assume !Heap[s1,alloc];
    call Stack..ctor(s1);

  // s2 = new Stack();
    havoc s2;
    assume !Heap[s2,alloc];
    call Stack..ctor(s2);

  call Stack.Push(s1,5);

  // meets precondition: DisjointSet(Heap[s,Stack.footprint], Heap[s2,Stack.footprint]);
  call Stack.Switch(s1,s2);
}

procedure Stack.Client3(s: ref) returns ()
  free requires IsGoodObject(s, Heap);
  requires Stack.INV(Heap,s);
  modifies Heap;
  ensures Stack.INV(Heap,s);
{
  var s2: ref;
  var i : int;

  // s2 = new Stack();
```

```
    havoc s2;
    assume !Heap[s2,alloc];
    call Stack..ctor(s2);

  call Stack.Push(s,5);

  // meets precondition: DisjointSet(Heap[s,Stack.footprint], Heap[s2,Stack.footprint]);
  call Stack.Switch(s,s2);
}


// Expected errors because taking two arbitrary heaps means that one might be within
// the other's footprint. This is because we do not differentiate between the data
// structures used to hold the contents of the stack and the contents themselves
// (i.e., the footprint includes both the array or linked list and the elements pointed
// to from the array or linked list).
// StackClient.bpl(87,3): Error BP5001: This assertion might not hold.
// Execution trace:
//     StackClient.bpl(85,3): anon0
// StackClient.bpl(88,3): Error BP5001: This assertion might not hold.
// Execution trace:
//     StackClient.bpl(85,3): anon0
// StackClient.bpl(90,1): Error BP5003: A postcondition might not hold at this return statement.
// StackClient.bpl(81,3): Related location: This is the postcondition that might not hold.
//
procedure Stack.Client4(s1: ref, s2: ref) returns ()
  free requires IsGoodObject(s1, Heap);
  free requires IsGoodObject(s2, Heap);
  // user defined preconditions
  requires Stack.INV(Heap,s1);
  requires Stack.INV(Heap,s2);
  modifies Heap;
  ensures Stack.INV(Heap,s1);
  ensures Stack.INV(Heap,s2);
{
  var i : int;

  call Stack.Push(s1,5);

  assert !Heap[s2,Stack.footprint][s1];
  assert DisjointSet(Heap[s1,Stack.footprint], Heap[s2,Stack.footprint]);

}
```