A Logical Account of Hoare's Mismatch

Information Hiding via Second Order Framing in Region Logic

Anindya Banerjee
IMDEA Software, Madrid, Spain
anindya.banerjee@imdea.org

David A. Naumann
Stevens Institute of Technology, Hoboken NJ, USA
naumann@cs.stevens.edu

Abstract

We investigate information hiding in object-based programs and the associated mismatch. While client reasoning is in terms of interface specifications, the implementation of an interface is verified against different specifications that involve invariants about internal data structures. Soundness of this mismatched reasoning depends on encapsulation of internal data structures. The problem is that encapsulation is notoriously difficult to achieve in contemporary software in which shared mutable objects are ubiquitous. We account for the mismatch via proof rules that phrase the mismatch using explicit conditions that are imposed on client effects. Effects are tracked using ghost state and separation assertions in a style that has been used in a number of verification tools. Our approach permits the formulation of encapsulation disciplines (such as ownership, or package confinement) as part of the interface specification, in the form of a dynamic boundary, rather than as a discipline directly baked into a verifier. One implication is that we can provide a foundation for the axiomatic semantics of these verifiers. Our approach is flexible in that disciplines can be used on a per-module basis and then combined to achieve end to end soundness.

1. Introduction

Many programs manipulate shared, mutable objects, be they memory segments in C code, heap records in ML code, or objects in the sense of Java-like languages. Most software designs are based on abstraction at many scales. For example: A $Point$ offers operations such as repositioning without revealing whether the internal representation uses polar or cartesian coordinates. A $Collection$ offers addition and removal of elements without revealing whether the internal representation is a balanced binary tree or something else. An application framework (e.g., the Google Web Toolkit) offers complex concepts and facilities without revealing the still more complex underlying infrastructure. Abstraction is achieved by hiding irrelevant details, in particular internal data and invariants on which the implementation relies. Hiding of internal invariants means that they do not appear in specifications used by "clients" of the abstraction. This mismatch between what is verified about implementations and what is assumed by clients was articulated and justified by Hoare: invariants should depend only on data that is encapsulated to prevent clients from falsifying the invariants [17].

It is notoriously difficult to achieve encapsulation in the presence of shared, dynamically allocated mutable objects [20, 27]. Current tools for software verification either do not support hiding of invariants (e.g., Jahob [35], jStar [12]), do not treat object invariants soundly (e.g., ESC/Java [14], Eiffel) or at best offer soundness for restricted situations where hierarchical structure can be imposed on the heap (e.g. Spec# [4]) which is the best that is provided by current theory.

Scoping does provide some encapsulation. Class $Point$ declares its fields as private, i.e., visible only to code within the class. Classes pertinent to collections —Set, $List$, iterators, various data structures— may be grouped together in a module. Scoping mechanisms are essential but insufficient for reasoning about dynamically allocated mutable objects.

Building on scope and type mechanisms it is possible to enforce module level alias control (as in Confined Types [15]) to justify hiding of invariants that depend on some or all instances of some classes declared in the module. For encapsulation at the granularity of instance-oriented "object invariants", various ownership disciplines have been introduced [9, 7, 11, 25, 23]. Some have been deployed in verification systems, e.g., in JML tools [8] and in Spec# [4]. For example, suppose for simplicity that an instance of class Set is viewed as owning the nodes of a linked list and a type system enforces that neither clients nor other instances of Set can access these nodes. Code in Set relies on an invariant, such as the absence of duplicate elements, that can be hidden from clients. The inflexibility of various formalizations of ownership has led to quite a few competing and incompatible variations, each described in a way that requires its global imposition on all program components (e.g., [9, 7, 11]). Another problem is that, although it is easy to embody Hoare's mismatch in the axiomatic semantics used by a verifier, the details are not often justified rigorously.

Separation Logic [28] (SL) offers elegant means to reason about the *footprint* of an invariant, i.e., the heap locations on which it depends, and the footprint of a command, i.e., the locations possibly written. On this basis, O'Hearn et al [27] give what we call a *second order frame* rule (SOF) that directly embodies Hoare's mismatch. By contrast with ownership disciplines, the SOF rule pertains directly to hiding on the granularity of a module (i.e., a group of procedures sharing some encapsulated state). We show that this flexibly encompasses invariants such as those that involve cooperating clusters of objects such as a Subject and its Observers. Soundness of the SOF rule has been shown directly in terms of a standard, non-axiomatic program semantics [29]. The rule relies on two critical features of SL. The *separating conjunction* $P * Q$ expressess that formulas P and Q are both true, of disjoint parts of the heap. The *tight interpretation* of a correctness judgement $\{P\}C\{Q\}[\overline{x}]$ says that C is not only correct but neither reads nor writes outside the footprint of P. The Hoare triple is augmented by a *modifies clause* that lists the variables $\overline{x}$ that may be written.

Our contribution is a SOF rule for a classical first-order assertion language that uses an ordinary interpretation of correctness judgements (for fault-avoiding partial correctness). This is achieved by making footprints explicit using *regions* (sets of references), including expressions for the image of a region under a field name and for disjointness of regions. State-dependent region expressions are used in the modifies clause and in a subsidiary judgement about footprints of formulas. Read and write footprints are expressed using (mutable) ghost state which avoids the need to express footprints using inductively defined predicates that traverse data structures.

Contributions

- We augment module interface specifications by including a *dynamic encapsulation boundary* which must be respected by clients. The dynamic boundary is described via read effects that approximate, in a way suitable to appear in the interface, the footprint of the hidden invariant.
- We extend *region logic* (RL) [3] to include procedures and correctness judgements with hypotheses (Sec. 3 and 4). We improve the treatment of hiding sketched briefly in that paper. On this basis we give our SOF rule (Sec. 5). We prove its soundness in a straightforward operational semantics, which validates standard proof rules, such as Hoare's rule of Conjunction, and can use a deterministic or nondeterministic memory allocator.
- By contrast with SL, (a) our SOF rule allows the hidden invariant to depend on state that is also read by client programs (e.g., global entities in an application framework); and (b) it allows invariants that are not *precise* [27], i.e., they do not have a unique minimal footprint. Precise predicates typically traverse a data structure (e.g., "every reachable node is balanced") whereas imprecise ones involve existential quantification (e.g., "some queue is nonempty"). In Sec. 7 we examine why our setting is not susceptible to Reynolds' conundrum [27] which shows that in SL the SOF rule needs to be restricted to precise invariants or precise specifications to be consistent with the rule of Conjunction.
- We show by examples (Sec. 2 and 6) that our SOF rule can hide invariants that pertain to several objects with a single owner, as well as design patterns in which several peers cooperate (which are incompatible with ownership and remain as challenge problems in the current literature [20, 6, 19]). A program may link together multiple modules, each with its own hidden invariant and dynamic boundary. Our approach encompasses alias confinement disciplines that are enforceable by static analysis [11] as well as less restrictive disciplines that impose proof obligations on clients as in ownership transfers that are "in the eye of the asserter" [27].

Our explicit use of ghost state for frame specifications and separation reasoning was directly inspired by the *dynamic frames* of Kassios [18], whence our term "dynamic boundary". Similar uses of ghost state, including pure first order encodings of reachability properties, have been found effective in verification tools based on automated theorem provers [16, 10, 35]. One of our aims is to justify the axiomatic semantics used in such tools, which often embody Hoare's mismatch. We pay a price in verbosity compared with SL, in that ghost fields and variables need to be declared, assigned, and used in specifications. For example, the code of class Set in Sec. 2 assigns newly allocated list nodes to a "*rep*" field, and the code in our memory manager example (Sec. 6) updates a region variable holding the objects currently "owned" by the manager.

For readability, the formalization (Sec. 3 and 4) is for bare-bones programs with rudimentary procedures. Related work is discussed further in Sec. 7.

Our aim is not to advocate a particular proof system (and certainly not a module system) but rather to move beyond competing attempts to provide the "right" specification notation and programming discipline for hiding invariants on mutable state. We provide a logical bridge between internal and external views of a module interface, in which the encapsulation boundary is fully described through dynamic framing (to augment the conventional syntactic signature of the interface). On this basis we hope to shift attention to creating a corpus of specification patterns, with a range of granularity, generality, and balance of proof obligations between modules

```
class Node { val : int; nxt : Node; own : Object; }
globalvar pool : rgn;
class Set {
  lst : Node; rep : rgn;
  Set()
  { self.lst := null; self.rep := ∅; pool := pool ∪ {self}; }
  add(i : int)
  { if not(contains(i)) then
      var n : Node := new Node();
      n.val := i; n.own := self; n.nxt := self.lst;
      self.lst := n; self.rep := self.rep ∪ {n}; }
  contains(i : int) : boolean   { "linear search for i" }
  remove(i : int)   { "remove first node containing i, if any" }}
```

Figure 1. Library code for Set example. Class Set and variable $pool$ comprise a module.

Method	Post-condition
$Set()$	$elements = empty$ $\wedge\ pool = \mathsf{old}(pool) \cup \{\mathsf{self}\}$
$add(i : int)$	$elements = \mathsf{old}(elements) \cup \{i\}$
$contains(i : int)$	$result = (i \in elements)$
$remove(i : int)$	$elements = \mathsf{old}(elements) - \{i\}$

Method	Effects
$Set()$	wr $pool$
$add(i : int)$	wr {self}‘any, wr alloc
$contains(i : int)$	(none)
$remove(i : int)$	wr {self}‘any, wr $pool$‘rep‘any

Figure 2. Public specifications for class Set. Preconditions: $true$.

```
var s : Set := new Set(); var n : Node := new Node();
s.add(1); s.add(2); n.val := 1; s.remove(1);
b := s.contains(1);
```

Figure 3. Example client, in context of variable b : boolean.

and their clients. Modular verification tools will eventually be able to support integrated use of complementary disciplines for a wide range of program design patterns and application frameworks.

2. Synopsis

For a first example of second order framing, we consider the simple program in Fig. 1.[1] This section begins by introducing region logic as used by the interface specifications in Fig 2. Then we consider reasoning about the implementation (Fig. 1) using an invariant, which serves to illustrate local reasoning via a first order "frame rule" akin to Hoare's rule of Invariance. Finally, we consider reasoning about a client program, using the interface specifications. To justify hiding the invariant, the interface includes a dynamic boundary for the invariant and the client code is obliged not to write within that boundary. We conclude by sketching how soundness of this reasoning is captured by the SOF rule.

Interface specifications —the client's view. The interface specifications for methods of collection class Set do not expose the internal representation but rather refer to the abstraction of interest.

[1] The programming notation is similar to Java, in particular a value of a class type like $Node$ is either null or a reference to an allocated object with the fields declared in the class. Methods have an implicit parameter, self.

The specifications (Fig. 2) are expressed in terms of an integer set, $elements$, that could be formalized as a "model field".[2] Abstraction of this sort is commonplace and important, but not the focus of this paper so we don't formalize $elements$.

What is important is that we shall add to the interface a dynamic boundary which abstracts from the state to be encapsulated for a hidden invariant. To this end, the implementation in Fig. 1 is instrumented using ghost state of type rgn; a *region* is a set of allocated references of any type, possibly also containing null. Inclusion of null is one of several small deviations from our paper [3], which remains useful for a more detailed introduction to the basic logic.

The effects clause in Fig. 2 for the constructor, $Set()$, is a conventional modifies specification that says variable $pool$ is allowed to be assigned. Our chosen notation and terminology reflects a feature that deals with framing issues beyond the scope of the paper, namely, that both read and write effects may be ascribed to commands and procedures.

For add, the write effect wr $\{\mathsf{self}\}$'any is also a conventional one that says fields of self may be written (cf. self.$state$ in a JML or Spec# modifies clause). The expression $\{\mathsf{self}\}$ denotes a *singleton region* containing the value of self (an object reference). For any field name f and region expression G, the effect wr G'f allows update of the f fields of objects in G (and is read "write G image f"). The special name "any" abstracts from specific field names to allow any field to be written; this can be refined to "data groups" [24, 8] that abstract from specific field names that should not be exposed to clients.

The primitive region expression "alloc" denotes the current domain of the heap, i.e., the set of allocated references. The effect of add includes wr alloc which means new objects may be allocated.

For $remove$, the most interesting effect is wr $pool$'rep'any. Here $pool$'rep is a region expression; it denotes the union of the rep fields of all objects in $pool$. The effect says fields of objects in $pool$'rep (in the initial state) may be written. For any s of type Set, field rep is intended to hold the (references to) the nodes reached from $s.lst$, and $pool$'rep is the union of those regions, i.e., all the nodes used by Sets in $pool$. (To allow for alternate implementations it might be wise for add to also have the effect wr $pool$'rep'any.)

The effect wr $pool$'rep'any is a "dynamic frame" because rep is a mutable field and $pool$ a mutable variable: the meaning of the effect is state dependent.[3]

A module invariant. For efficiency, our implementation of $remove$ relies on the invariant that no integer value is duplicated in the list rooted at lst. This invariant must be established by the constructor and preserved by all methods of class Set. However, as per what we call *Hoare's mismatch* [17], the invariant does not appear in the interface specifications as viewed by clients.

The invariant "no duplicates" pertains to a single instance s of Set, together with its list. The ghost field $s.rep$ is intended to refer to the set of nodes reachable from field $s.lst$. To avoid the need for reachability, which is costly for automated verification and not available in all assertion languages, we approximate it using the invariant that $s.rep$ contains only nodes "owned" by s. Consider the following condition. The first conjunct says there are no duplicates. The next two say that $s.rep$ is nxt-closed and contains $s.lst$. The last says all nodes in $s.rep$ are owned by s.

$$\begin{aligned} SetI(s:Set):\ & (\forall n,m:Node \in s.rep \mid n = m \vee n.val \neq m.val) \\ & \wedge\ s.lst \in s.rep\ \wedge\ s.rep\text{'}nxt \subseteq s.rep \\ & \wedge\ s.rep\text{'}own \subseteq \{s\} \end{aligned}$$

Note that "$n \in s.rep$" is considered false in case $s = \mathsf{null}$.

Were we to extend the example by adding iterators, we might find that the natural granularity for an invariant would be a Set together with its iterators as well as its list. To avoid commitment to invariants of a specific granularity, we consider *module invariants*, associated with the program syntax and unit of scope: a "module", consisting of one or more class declarations. For example, the module invariant could say $SetI$ holds for every instance of Set. To illustrate flexibility in choosing encapsulation boundaries, we choose instead to say $SetI$ holds for only those instances that are in $pool$ (which, e.g., could be those returned by a factory method):

$$I:\ \mathsf{null} \notin pool \wedge \forall s:Set \in pool \mid SetI(s)$$

We shall find a "frame" or footprint for I, that can serve as a dynamic boundary expressing the state-dependent aspect of the encapsulation that will allow I to be hidden from clients. But first we seek a frame for the object invariant $SetI(s)$, which will be used for "local reasoning" [28] at the granularity of a single instance of Set. The frame is given by read effects that describe the part of the state on which the value of $SetI(s)$ may depend:

$$\overline{\delta}_0:\ \mathsf{rd}\, s,\ \mathsf{rd}\, \{s\}\text{'}(rep, lst),\ \mathsf{rd}\, s.rep\text{'}(nxt, val, own)$$

Region logic provides rules for a judgement that says certain read effects bound the footprint of a formula, in this case:

$$true \vdash \overline{\delta}_0\ \mathsf{frm}\ SetI(s) \tag{1}$$

The effect rd s says $SetI(s)$ may depend on variable s, and rd $\{s\}$'(rep, lst) says it may depend on $s.rep$ and $s.lst$. The dynamic part is rd $s.rep$'(nxt, val, own) which says it depends on the nxt, val, and own fields of some objects in $s.rep$. The judgement (1) involves a formula, here $true$, because effects can involve region expressions like $s.rep$ that depend on mutable state, so framing relationships may hold only under some conditions on that state. For example, we can derive

$$s \in pool \vdash \mathsf{rd}\, pool\text{'}(rep, lst)\ \mathsf{frm}\ s.lst \in s.rep \tag{2}$$

Using judgements including (2) and (1) we can derive a frame judgement $true \vdash \overline{\delta}_I$ frm I for the module invariant, where

$$\overline{\delta}_I:\ \mathsf{rd}\, pool,\ pool\text{'}(rep, lst),\ pool\text{'}rep\text{'}(nxt, val, own)$$

An abstraction of $\overline{\delta}_I$ will be used later, when we turn to verification of the client.

Framing for local reasoning using the invariant. For the implementations in Fig. 1, we would like to reason "locally" in terms of a single Set. The ownership conditions in $SetI(s)$ yield an "island confinement" property (where # denotes disjointness of sets):

$$I \Rightarrow (\forall s,t:Set \in pool \mid s = t \vee \{s\}\text{'}rep \,\#\, \{t\}\text{'}rep) \tag{3}$$

because if $s \neq t$, $n \neq$ null, and n is in $s.rep \cap t.rep$ then $n.own = s$ and $n.own = t$, a contradiction. To verify that method add preserves I, we exploit island confinement in order to focus on the object invariant. Now, I is equivalent to $SetI(\mathsf{self}) \wedge Iexcept$, where

$$Iexcept:\ \mathsf{null} \notin pool \wedge \forall s \in pool - \{\mathsf{self}\} \mid SetI(s)$$

We can frame $Iexcept$ by the effects

[2] The term is from JML [8]. Various specification languages offer different ways to express abstractions represented by concrete data. For example, $elements$ might be defined as a zeroary function that traverses the list and accumulates the set of its elements.

[3] Because regions are untyped, $pool$ could potentially contain references of many types; nonetheless, the region $pool$'rep involves only to those objects in $pool$ that have field rep, which by uniqueness of field names means objects of type Set. Also, we have extended the first version of the logic [3]: if G is a region and f is a field of some reference type, G'f is a region of the f-images, but if f is of type rgn then G'f is a region, formed as the union of the f-images.

$$\overline{\delta}_x: \quad \mathsf{rd\,self},\ pool,\ (pool - \{\mathsf{self}\})\text{‘}(rep, lst),$$
$$(pool - \{\mathsf{self}\})\text{‘}rep\text{‘}(nxt, val, own)$$

The footprint of $Iexcept$ is disjoint from the footprint of $SetI(\mathsf{self})$. More to the point, let B_{add} be the body of method add. By ordinary means we can verify the following Hoare triple:

$$\{SetI(\mathsf{self})\}B_{add}\{SetI(\mathsf{self}) \wedge elements = \mathsf{old}(elements) \cup \{i\}\}$$

Next, we exploit separation to conjoin $Iexcept$ to the pre and post conditions because the write effects of add (Fig. 2) are separate from the read effect of $Iexcept$. To make this precise, we define an operator: If $\overline{\delta}$ is a set of read effects and $\overline{\varepsilon}$ is a set of write effects then $\overline{\delta} \star \overline{\varepsilon}$ is a conjunction of disjointness formulas, validity of which ensures that writes allowed by $\overline{\varepsilon}$ cannot affect the value of a formula with footprint $\overline{\delta}$. (The formula $\overline{\delta} \star \overline{\varepsilon}$ is defined by induction on the syntax of the effects.) Here is our first order frame rule (from [3]):

$$\textsc{Frame}\ \frac{\vdash \{\ P\ \}\ C\ \{\ P'\ \}\ [\overline{\varepsilon}] \qquad P \vdash \overline{\delta}\ \mathsf{frm}\ Q \qquad P \Rightarrow \overline{\delta}\ \star\ \overline{\varepsilon}}{\vdash \{\ P \wedge Q\ \}\ C\ \{\ P' \wedge Q\ \}\ [\overline{\varepsilon}]}$$

It happens that $\overline{\delta}_x \ \star\ (\mathsf{wr}\,\{\mathsf{self}\}\text{‘}\mathsf{any}, \mathsf{wr\,alloc})$ is $true$, so we can take Q to be $Iexcept$ in rule FRAME to complete the proof of $\{I\}B_{add}\{I \wedge elements = \mathsf{old}(elements) \cup \{i\}\}$.

Reasoning about a client while hiding the invariant. Besides verification of the module's method bodies with respect to specifications in which the invariant I is explicit, there is another obligation on the module. It must declare a dynamic boundary that frames I. It would not be appropriate to use $\overline{\delta}_I$, which mentions field lst that would be declared private. We choose to use

$$\overline{\theta}_I: \ \mathsf{rd}\,pool,\ pool\text{‘}\mathsf{any},\ pool\text{‘}rep\text{‘}\mathsf{any}$$

The obligation is $I \vdash \overline{\theta}_I\ \mathsf{frm}\ I$, which is derivable from $true \vdash \overline{\theta}_1\ \mathsf{frm}\ I$ by a subsumption rule.

Something is needed to ensure that I is initially true. Typical formalizations include an initializer command, so the client program takes the form $\mathsf{let}\ m\ \mathsf{be}\ B\ \mathsf{in}\ (init; C)$. With dynamic allocation, it is constructors that do much of the work to establish invariants. In the present example, let us define $Init$ to be the condition $pool = \varnothing$ which is suitable to be declared in the module interface. Note that $Init \Rightarrow I$ is valid.

Finally, we can turn to verifying the simple client command in Fig. 3. We will verify that under precondition $Init$ the client establishes postcondition $b = false$. Here is a proof outline.

$\{Init\}$
$s := \mathsf{new}\ Set();$
$\{s.elements = \varnothing \wedge pool = \{s\}\}$ // by spec of Set
$n := \mathsf{new}\ Node();$
$s.add(1);\ s.add(2);$
$\{s.elements = \{1,2\} \wedge pool = \{s\}\}$ // by spec of add
$n.val := 1;$
$s.remove(1);$
$\{s.elements = \{2\} \wedge pool = \{s\}\}$ // by spec of $remove$
$b := s.contains(1);$
$\{b = false\}$ // by spec of $contains$

As it should, this reasoning uses the interface specifications (Fig. 2). However, there is an additional proof obligation. For it to be sound to hide I from the client, we need that the client's write effects are separate from the dynamic bound $\overline{\theta}_I$ —not just pre/post effects but also effects at intermediate steps, at which module methods like add are called. We include the assignment $n.val := 1$ as a simple example of how encapsulation might be violated (but is not). If this assignment was replaced by $s.lst.val := 1$ then indeed it would break the invariant and render the program incorrect.

The effect of $n.val := 1$ is $\mathsf{wr}\ \{n\}\text{‘}val$ and it must be shown to be outside the boundary $\overline{\theta}_I$. After all, I reads field val as seen in the more precise footprint, $\overline{\delta}_I$, subsumed by $\overline{\theta}_I$. By definition of $\star$, we have that $\overline{\theta}_I\ \star\ \mathsf{wr}\ \{n\}\text{‘}val$ is $\{n\} \# pool \wedge \{n\} \# pool\text{‘}rep$. We must show it holds just before the assignment $n.val := 1$.

The condition $\{n\}\ \#\ pool$ is equivalent to $n \notin pool$ which clearly holds —none of the client code writes $pool$ and the value of n is fresh.[4] There are several ways to show $\{n\}\ \#\ pool\text{‘}rep$. One way is to notice that a form of "package confinement" [15] applies here: references to the instances of $Node$ used by the Set implementation are never made available to client code. (That is stronger than the property required here, which is merely that such references are not misused by clients.)

To be very explicit about the reasoning that could be embodied by a static analysis for confinement, we shall exploit field own of class $Node$. The analysis might ensure the all-states invariant that any $Node$ pointer p available to the client code has $p.own = \mathsf{null}$ (the default from $Node$'s constructor). Moreover, nodes are not "leaked" by the module code, which can be shown using

$$R: \ pool\text{‘}rep\text{‘}own \subseteq pool\ \wedge\ \mathsf{null} \notin pool$$

Unlike I, formula R is suitable to appear in the interface. A general fact about region images is that $G\text{‘}f \subseteq H$ and $x.f \notin H$ imply $x \notin G$ (where G, H are any region expressions and f any field). So using R we get $n \notin pool\text{‘}rep$ as required by the obligation to respect the dynamic encapsulation boundary.

A dynamic boundary is expressed in terms of state potentially mutated by the module implementation, e.g., the effect of add in Fig. 1 allows writing state on which θ_I depends.[5] So interface specifications need to provide clients with sufficient information to reason about the boundary. In the example, R could be explicitly conjoined with the interface's method specifications, or declared as a public invariant [21]. In our memory manager example (Sec 6), it is not a fixed invariant but rather the individual method specifications that allow clients to reason about the boundary.[6]

In summary, the verification of our client and its module are justified by the following rule which embodies Hoare's mismatch:

$$\frac{\begin{array}{c}\Delta\langle\overline{\delta}\rangle\ ;\ \Theta\langle\overline{\theta}\rangle \vdash \{\ P\ \}\ C\ \{\ P'\ \}\ [\overline{\varepsilon}] \\ \Delta\langle\overline{\delta}\rangle\ ;\ (\Theta \circledast I)\langle\rangle \vdash \{\ Q \wedge I\ \}\ B\ \{\ Q' \wedge I\ \}\ [\overline{\varepsilon}] \\ I \vdash \overline{\theta}\ \mathsf{frm}\ I \qquad Init \Rightarrow I\end{array}}{\Delta\langle\overline{\delta}\rangle \vdash \{\ P \wedge Init\ \}\ \mathsf{let}\ m\ \mathsf{be}\ B\ \mathsf{in}\ C\ \{\ P'\ \}\ [\overline{\varepsilon}]}$$

Here for simplicity Θ consists of a single method specification, $\{Q\}m\{Q'\}[\overline{\eta}]$. The first antecedent expresses the proof obligation on the client program C: in addition to the usual pre/post/modifies specification, it must respect the dynamic boundary $\overline{\theta}$ which (please note!) is written together with the specification of m. In general, a boundary is associated with each group of methods taken to comprise the interface of a module, e.g., here we include a second module interface $\Delta\langle\overline{\delta}\rangle$. To respect the boundary means that every primitive action within C must stay outside $\overline{\theta}$ and $\overline{\delta}$. It is not enough that the end-to-end effect $\overline{\varepsilon}$ of C is outside $\overline{\theta}, \overline{\delta}$, as $\overline{\varepsilon}$ only describes effects on objects that exist in the initial state. Moreover, $\overline{\varepsilon}$ includes

[4] Had we chosen to use typed regions, or to maintain an invariant that every element of $pool$ is of type Set, then $n \notin pool$ would hold because the type $Node$ of n is not a subtype of Set.

[5] Dynamic framing raises the issue of interference with effects, which is handled in the rule for sequential composition in [3].

[6] Nonetheless, we believe the present situation, where $I \Rightarrow R$, is typical. It suggests that the module interface specification may include, in addition to the dynamic bound, a public invariant that is framed by the bound and can be seen as part of the encapsulation discipline. In this case clients are not responsible for maintaining R because (from $I \Rightarrow R$) it is protected by the bound they are already required to respect. Nor is there an additional proof obligation on the module implementation.

$x, y, r \in \mathit{VarName} \quad f, g \in \mathit{FieldName} \quad K \in \mathit{DeclaredClassNames}$
$T ::= \mathsf{int} \mid K \mid \mathsf{rgn}$
$E ::= x \mid c \mid \mathsf{null} \mid E \oplus E \quad c$ is in $\mathbb{Z}$ and $\oplus$ is in $\{=, +, -, *, >, \ldots\}$
$G ::= x \mid \{E\} \mid \mathsf{emp} \mid \mathsf{alloc} \mid G\text{‘}f \mid G \otimes G \quad \otimes$ is in $\{\cup, \cap, -\}$
$F ::= E \mid G$
$C ::= x := F \mid x := \mathsf{new}\ K \mid x := x.f \mid x.f := F \mid \ldots$

Figure 5. Programming language (excerpts). Category E is for program expressions and G for region expressions.

effects from calls to methods of Θ and those effects are not required to be outside the boundary $\overline{\theta}$.

The second antecedent is the proof obligation on the implementation, B, of m, in which the invariant I is explicit; the dynamic boundary for Θ is empty, so $\overline{\theta}$ is not imposed on B, but the bound $\overline{\delta}$ on the ambient library must be respected by B. Any recursive calls to m in B are verified using the specifications $\Theta \circledwedge I$ which means I is conjoined, here $\{Q \wedge I\}m\{Q' \wedge I\}[\overline{\eta}]$. The side conditions $I \vdash \overline{\theta}$ frm I and $\mathit{Init} \Rightarrow I$ are obligations on the module implementation.

The rule above is a derived rule. Fig. 4 gives a generic derivation, using an ordinary rule that links a procedure to its implementation with no mismatch in specifications, together with the SOF rule and a rule to forget the dynamic boundary once it has served its purpose. The beauty of the SOF, the form of which is due to O'Hearn et al [28], is that it distills the essence of Hoare's mismatch —which, with proper encapsulation, is an intricate match.

3. Background: Region logic without procedures

Programming language. Fig. 3 gives the programming language syntax, except for procedures. The semantics is standard and deferred to Sec. 4. A program consists of a command C in the context of some class declarations. Commands are standard: assignment, object allocation, field access, field update, conditionals, loops, and local variable blocks.

A class declaration $\mathsf{class}\ K\ \{\ \overline{T}\ \overline{f}\ \}$ introduces a type name K. The values of this type are null and all allocated references to mutable objects of type K with fields $\overline{f} : \overline{T}$. Here and throughout, identifiers with an overline range over lists. In addition to int and reference types, there is type rgn with values ranging over finite sets of references of any type.

Program expressions, E, do not depend on the heap: $y.f$ is not an expression but rather part of the primitive field access command, $x := y.f$, for reading a field, as in SL. Region expressions, G, cannot influence control flow or the value of non-region fields/variables (as no integer expressions have subexpressions of type rgn); their purpose is to serve as ghosts for reasoning.

Typing. There is an ambient class table comprising a well formed collection of class declarations. We write $\mathit{fields}(K)$ for the field declarations $\overline{f} : \overline{T}$ of class K. Judgement $\Gamma \vdash F : T$ says that in context Γ that assigns types to variable names, region or program expression F has type T, and $\Gamma \vdash C$ says C is a well formed command in Γ. Type int is separated from reference types: there is no pointer arithmetic. The typing rule for singleton region $\{E\}$ enforces that E is of reference type. The rule for region dereference, $G\text{‘}f$, checks that f is declared in some class K and either f is a field of class type, K', or f is of type rgn. The primitive command for field access, $x := y.f$, is restricted to non-region fields. For fields of type rgn, note that, e.g., $x := \{y\}\text{‘}f$ is permitted and is an instance of ordinary assignment, $x := F$. An implicit side condition on all typing and also proof rules is that both the consequent and the antecedents are well formed.

$$\begin{array}{lll} P & ::= & E = E \mid x.f = E \mid G \subseteq G \mid G \mathbin{\#} G \mid \mathsf{type}(K, G) \\ & \mid & (\forall x : \mathsf{int} \mid P) \mid (\forall x : K \in G \mid P) \mid P \wedge P \mid \neg P \end{array}$$

$$\begin{array}{lll} \sigma \models x.f = E & \text{iff} & \sigma(x) \neq \mathit{null} \text{ and } \sigma(x.f) = [\![E]\!]\sigma \\ \sigma \models G_1 \mathbin{\#} G_2 & \text{iff} & [\![G_1]\!]\sigma \cap [\![G_2]\!]\sigma \subseteq \{\mathit{null}\} \\ \sigma \models^{\Gamma} \forall x : K {\in} G \mid P & \text{iff} & \mathit{extend}(\sigma, x, o) \models^{\Gamma, x:K} P \text{ for all} \\ & & o \in [\![G]\!]\sigma \text{ s.t. } o \neq \mathit{null} \text{ and } \mathit{type}(o, \sigma) = K \end{array}$$

Figure 6. Formulas: grammar and semantics (excerpts).

Semantics. We assume given a set *Ref* of reference values including a distinguished value, *null*. A Γ-*state* assigns values to the variables in Γ and also has a heap. We abstract from the concrete representation of states and merely assume the following operations are available for a state σ: $\sigma(x)$ is the value of x, $\mathit{alloc}(\sigma)$ is the set of all allocated references, $\mathit{type}(o, \sigma)$ is the type of an allocated reference o, $\mathit{update}(\sigma, o.f, v)$ overrides σ to map field f of o to v (for $o \in \mathit{alloc}(\sigma)$), $\mathit{extend}(\sigma, x, v)$ extends σ to map x to value v (for $x \notin \mathit{dom}(\sigma)$), $\mathit{new}(\sigma, o, K, \overline{v})$ extends σ to map fresh o (i.e., assuming $o \notin \mathit{alloc}(\sigma)$) to a K-record with field values $\overline{v}$ and type K. States are assumed to be well typed and to have no dangling references.

The semantics of program expressions E, written $[\![E]\!]\sigma$, is straightforward and omitted. Note that $[\![E]\!]\sigma$ is always a value (of appropriate type), never *fault*; moreover it only depends on the store, not the heap. For region expressions, the meaning of singleton region $\{E\}$ is $\{[\![E]\!]\sigma\}$. The meaning of emp is the empty set and that of alloc is $\mathit{alloc}(\sigma)$, i.e., the set of all allocated references. The meaning of $G\text{‘}f$, if f is a reference type, is the set containing the f-images of all non-*null* references in G's denotation; but if $f : \mathsf{rgn}$ then $G\text{‘}f$ denotes the union of the f-images.

Assertions and effects. The assertion language appears in Fig. 6. Quantification is over int and reference types only (not over rgn). For reference types, a bounding region is required (e.g., region expression G in $\forall x : K \in G \mid P$) and the bound variable must not appear in the bound of the quantification. This facilitates framing. Fig. 6 also gives the semantics of a well formed formula, $\Gamma \vdash P$, as a satisfaction relation $\sigma \models^{\Gamma} P$ that is defined for all Γ-states σ. A formula is *valid* iff it is true in all states.

Effects are given by the grammar

$$\varepsilon ::= \mathsf{rd}\ x \mid \mathsf{rd}\ G\text{‘}f \mid \mathsf{wr}\ x \mid \mathsf{wr}\ G\text{‘}f \mid \mathsf{rd\ alloc} \mid \mathsf{wr\ alloc} \mid \mathsf{fr}\ G$$

Effect $\mathsf{wr}\ G\text{‘}f$ says f fields of any pre-existing object in G may be written. Note that, e.g., in $\mathsf{wr}\ \{x\}\text{‘}f\text{‘}g$ the first apostrophe is forming a region expression $\{x\}\text{‘}f$ and then the second is indicating what field (of objects in the region) is allowed to be written. Effect wr alloc allows allocation and $\mathsf{fr}\ G$ says that all elements of G in the final state are freshly allocated. Read effect $\mathsf{rd}\ G\text{‘}f$ says f fields of any pre-existing object in G may be read and rd alloc allows reading the set alloc. An *effect set* is a comma-separated list of effects, ranged over by $\overline{\varepsilon}$ etc. Effects are well formed in Γ if their constituent expressions are.

Freshness effects are needed to annihilate write effects of freshly allocated objects in sequences. Consider, e.g., $x := \mathsf{new}\ K; x.f := 0$; While the effect of the field update is $\mathsf{wr}\ \{x\}\text{‘}f$, the pre-state of the entire sequence does not contain the freshly allocated object. Indeed, in this case no pre-existing objects are updated. See the sequence rule in [3].

Definition 1 (allows transition) Let effect set $\overline{\varepsilon}$ be well formed in Γ and let σ, σ' be Γ-states. We say $\overline{\varepsilon}$ *allows transition from σ to σ'*, written $\sigma \rightsquigarrow \sigma' \models \overline{\varepsilon}$, iff σ' extends[7] σ and the following all hold:

[7] σ' extends σ provided $\mathit{alloc}(\sigma) \subseteq \mathit{alloc}(\sigma')$ and $\mathit{type}(o, \sigma) = \mathit{type}(o, \sigma')$ for all $o \in \mathit{alloc}(\sigma)$.

$$
\dfrac{\Delta\langle\overline{\delta}\rangle;(\Theta \owedge I)\langle\rangle \vdash \{\, Q \wedge I \,\}\; B \;\{\, Q' \wedge I \,\}\; [\overline{\eta}] \qquad \dfrac{\dfrac{\Delta\langle\overline{\delta}\rangle;\Theta\langle\overline{\theta}\rangle \vdash \{\, P \,\}\; C \;\{\, P' \,\}\; [\overline{\varepsilon}]}{\Delta\langle\overline{\delta}\rangle;(\Theta \owedge I)\langle\overline{\theta}\rangle \vdash \{\, P \wedge I \,\}\; C \;\{\, P' \wedge I \,\}\; [\overline{\varepsilon}]}\ \text{SOF}}{\Delta\langle\overline{\delta}\rangle;(\Theta \owedge I)\langle\rangle \vdash \{\, P \wedge I \,\}\; C \;\{\, P' \wedge I \,\}\; [\overline{\varepsilon}]}\ \text{DYNBNDELIM}}{\dfrac{\Delta\langle\overline{\delta}\rangle \vdash \{\, P \wedge I \,\}\; \mathsf{let}\ m\ \mathsf{be}\ B\ \mathsf{in}\ C \;\{\, P' \wedge I \,\}\; [\overline{\varepsilon}]}{\Delta\langle\overline{\delta}\rangle \vdash \{\, P \wedge \mathit{Init} \,\}\; \mathsf{let}\ m\ \mathsf{be}\ B\ \mathsf{in}\ C \;\{\, P' \,\}\; [\overline{\varepsilon}]}\ \text{CONSEQ}}\ \text{LINK}
$$

Figure 4. Schematic derivation for second order framing, assuming that Θ is a single specification $\{Q\}m(x:T)\{Q'\}[\overline{\eta}]$. For SOF the side condition is $I \vdash (\overline{\theta}, \mathsf{rd\,alloc})\ \mathsf{frm}\ I$ and for CONSEQ it is validity of $\mathit{Init} \Rightarrow I$.

(a) for every y in $\mathit{dom}(\Gamma)$, either $\sigma(y) = \sigma'(y)$ or $\mathsf{wr}\ y$ is in $\overline{\varepsilon}$
(b) for every $o \in \mathit{alloc}(\sigma)$ and every $f \in \mathit{fields}(o,\sigma)$, either $\sigma(o.f) = \sigma'(o.f)$ or there is $\mathsf{wr}\ G\text{‘}f$ in $\overline{\varepsilon}$ such that $o \in [\![G]\!]\sigma$
(c) if $\mathit{alloc}(\sigma') \neq \mathit{alloc}(\sigma)$ then $\mathsf{wr\,alloc}$ is in $\overline{\varepsilon}$.
(d) for each $\mathsf{fr}\ G$ in $\overline{\varepsilon}$, we have $[\![G]\!]\sigma' \subseteq \mathit{alloc}(\sigma') - \mathit{alloc}(\sigma)$.

Read effects in $\overline{\varepsilon}$ are not significant in Def. 1. As per part (b), expressions in write effects are interpreted in the initial state whereas (see (d)) the region in a freshness effect is interpreted in the final state.

We formalize the separation of read effects from write effects (as in rules FRAME and SOF) via conditions under which two states σ, σ' look the same when viewed through some read effects: define $\mathit{Agree}(\sigma, \sigma', \overline{\varepsilon})$ to hold just if σ' extends σ and moreover (a) $\sigma(x) = \sigma'(x)$ for all $\mathsf{rd}\ x$ in $\overline{\varepsilon}$; (b) $\mathit{alloc}(\sigma) = \mathit{alloc}(\sigma')$ if $\mathsf{rd\,alloc}$ in $\overline{\varepsilon}$; (c) $\sigma(o.f) = \sigma'(o.f)$ for all $\mathsf{rd}\ G\text{‘}f$ in $\overline{\varepsilon}$ and all $o \in [\![G]\!]\sigma$ with $f \in \mathit{fields}(o,\sigma)$.

Subeffects. A natural situation for effect subsumption is when the effect is very detailed or refers to local variable(s) and is thus unsuitable to be exposed for use in interfaces. Subeffecting allows abstraction of such an effect. For example, effect $\mathsf{wr}\ \mathit{pool}\text{‘}\mathit{rep}\text{‘}\mathsf{any}$ of the remove method in Sec. 2 is obtained by subeffecting and it abstracts from whatever particular object fields are written.

The judgement for subeffecting takes the form $P \vdash \overline{\varepsilon}_1 \leq \overline{\varepsilon}_2$ and express that the writes/reads in a bigger effect are more permissive in P-states. There are axioms like $G_0 \subseteq G_1 \vdash \mathsf{wr}\ G_0\text{‘}f \leq \mathsf{wr}\ G_1\text{‘}f$ and rules like strengthening the antecedent.

Framing. Recall from Sec. 2 that rule FRAME has a side condition that the write effect of a command be separated from the read footprint of the framed formula. The judgement, $P \vdash \overline{\varepsilon}\ \mathsf{frm}\ Q$ is intended to say the truth or falsity of predicate Q depends only on the state read according to $\overline{\varepsilon}$, i.e., $\overline{\varepsilon}$ covers the footprint of Q in P-states. We first provide a simple syntactic analysis for calculating read effects. For any expression F, define the read footprint of F, written $\mathit{ftpt}(F)$, as follows: If F is a program expression, E, define $\mathit{ftpt}(E) = \{\mathsf{rd}\ x \mid x \in \mathit{Vars}(E)\}$. For a region expression G, define $\mathit{ftpt}(G)$ by: $\mathit{ftpt}(G\text{‘}f) = \{\mathsf{rd}\ G\text{‘}f\} \cup \mathit{ftpt}(G)$; $\mathit{ftpt}(\mathsf{alloc}) = \{\mathsf{rd\,alloc}\}$; $\mathit{ftpt}(\mathsf{emp}) = \varnothing$. Other expressions and primitive formulas are straightforward, e.g., $\mathit{ftpt}(x.f = E) = \{\mathsf{rd}\ x, \mathsf{rd}\ \{x\}\text{‘}f\} \cup \mathit{ftpt}(E)$. This is lifted to judgements by the axiom $\mathit{true} \vdash \mathit{ftpt}(P)\ \mathsf{frm}\ P$ for primitive formula P. Proof rules of non-primitive formulas are inductively defined relative to validity of formulas (see [3]), e.g.,

$$
\frac{\mathit{ftpt}(G) \subseteq \overline{\varepsilon} \qquad P \wedge x \in G \vdash \overline{\varepsilon}, \mathsf{rd}\ x, \mathsf{rd}\ \{x\}\text{‘}\overline{f}\ \mathsf{frm}\ P'}{P \vdash \overline{\varepsilon}, \mathsf{rd}\ G\text{‘}\overline{f}\ \mathsf{frm}\ \forall x : K \in G \mid P'}
$$

Lemma 2 (agreement) For any states, σ, σ', any expression F, any predicates P, P', and any set of effects $\overline{\varepsilon}$: (a) Suppose $\mathit{Agree}(\sigma, \sigma', \mathit{ftpt}(F))$. Then $[\![F]\!]\sigma = [\![F]\!]\sigma'$. (b) Suppose $P \vdash \overline{\varepsilon}\ \mathsf{frm}\ P'$ and $\sigma \models P$ and $\mathit{Agree}(\sigma, \sigma', \overline{\varepsilon})$. Then $\sigma \models P'$ iff $\sigma' \models P'$.

$$
\vdash \{\, x \neq \mathsf{null} \wedge y = F \,\}\; x.f := F \;\{\, x.f = y \,\}\; [\mathsf{wr}\ \{x\}\text{‘}f]
$$

$$
\frac{\mathit{fields}(K) = \overline{f} : \overline{T} \qquad Q' \equiv (\mathit{type}(\{x\}, K) \wedge x.\overline{f} = \mathit{default}(\overline{T}))}{\vdash \{\, \mathit{true} \,\}\; x := \mathsf{new}\ K \;\{\, Q' \,\}\; [\mathsf{wr}\ x, \mathsf{wr\,alloc}, \mathsf{fr}\ \{x\}]}
$$

$$
\text{SUBEFF}\ \frac{\vdash \{\, P \,\}\; C \;\{\, P' \,\}\; [\overline{\varepsilon}] \qquad P \vdash \overline{\varepsilon} \leq \overline{\varepsilon}'}{\vdash \{\, P \,\}\; C \;\{\, P' \,\}\; [\overline{\varepsilon}']}
$$

$$
\text{CONJ}\ \frac{\vdash \{\, P_1 \,\}\; C \;\{\, P_1' \,\}\; [\overline{\varepsilon}] \qquad \vdash \{\, P_2 \,\}\; C \;\{\, P_2' \,\}\; [\overline{\varepsilon}]}{\vdash \{\, P_1 \wedge P_2 \,\}\; C \;\{\, P_1' \wedge P_2' \,\}\; [\overline{\varepsilon}]}
$$

$$
\text{SUBST}\ \frac{\begin{array}{c}\vdash \{\, P \,\}\; C \;\{\, P' \,\}\; [\overline{\varepsilon}] \qquad x \text{ is specification-only} \\ (P/x{\to}F) \Rightarrow \mathit{ftpt}(F) \star (\overline{\varepsilon}/x{\to}F)\end{array}}{\vdash \{\, P/x{\to}F \,\}\; C \;\{\, P'/x{\to}F \,\}\; [\overline{\varepsilon}/x{\to}F]}
$$

Figure 7. Selected correctness rules and axioms for commands.

Separators. Given effect sets $\overline{\delta}$ and $\overline{\varepsilon}$, the separator formula $\overline{\delta} \star \overline{\varepsilon}$ is defined to be the conjunction of certain disjointness formulas. For example, $\mathsf{rd}\ G_1\text{‘}f \ \star\ \mathsf{wr}\ G_2\text{‘}g$ is defined to be $G_1 \# G_2$ provided $f \equiv g$; otherwise, it is just true.[8] In a state where $\overline{\delta} \star \overline{\varepsilon}$ holds, nothing that the read effects in $\overline{\delta}$ allow to be read can be written according to the write effects $\overline{\varepsilon}$. The following result, together with Lemma 2, proves soundness of FRAME.

Lemma 3 (separator agreement) Consider any effect sets $\overline{\delta}$ and $\overline{\varepsilon}$. Suppose $\sigma \rightsquigarrow \sigma' \models \overline{\varepsilon}$ and $\sigma \models \overline{\delta} \star \overline{\varepsilon}$. Then $\mathit{Agree}(\sigma, \sigma', \overline{\delta})$.

Proof rules and correctness. A *correctness judgement* takes the form $\vdash^{\Gamma} \{\, P \,\}\; C \;\{\, P' \,\}\; [\overline{\varepsilon}]$ and is *well formed in* Γ just if $P, P', \overline{\varepsilon}$ are well formed in Γ and also $\Gamma \vdash C$. *Validity* of the judgement consists of a standard part —from any initial state that satisfies P, C does not fault (terminate with error), and if it terminates then the final state satisfies P'— and a non-standard one: any allocation and update effects are allowed by $\overline{\varepsilon}$ (Def. 1).

Selected proof rules appear in Fig. 7. For field update we choose a "small axiom", inspired by [28], which snapshots F with an auxiliary variable y in the precondition. The effect in the rule is a write to the f field of the single object in region $\{x\}$ that exists in the pre-state. The other rules shown are structural rules. The rule of substitution refers to *specification-only* variables that are not allowed to appear in code.[9]

4. Region logic with procedures

In this section, procedures are added to the programming language. Correctness judgements are extended with hypotheses, i.e., proce-

[8] We do not need to recurse further with G_1 and G_2. The ftpt function at the core of the frames judgement generates convex sets; if the set contains $\mathsf{rd}\ G\text{‘}f$ then it also contains $\mathit{ftpt}(G)$.

[9] Note that [3] conjectures mistakenly that read effects of commands can be used for the substitution rule; here we give a sound rule.

dure specifications. The program semantics is given and used to define the semantics of correctness judgements with hypotheses.

Program syntax and semantics. Command syntax has these additional forms

$$C ::= m(z) \mid \mathsf{let}\ m(x : T)\ \mathsf{be}\ C\ \mathsf{in}\ C \mid \mathsf{skip} \mid \mathsf{end}(x) \mid \mathsf{end}(m)$$

where m ranges over procedure names. The form "$\mathsf{let}\ m(x : T)\ \mathsf{be}\ B\ \mathsf{in}\ C$" binds m to B in C. Typing rules enforce that procedure body B is typed in a context with its parameter x and also m in scope —recursion is allowed. To streamline notation we avoid generalizing to multiple parameters and multiple, mutually recursive procedures, but there is no difficulty in that. The command end is a technical device, not allowed in source programs and used only in the semantics to mark the end of the scope of a variable or method. The *primitive commands* are method calls, skip, and assignments including field update and new.

Let Γ range over contexts like $y : T, m : (x : U)$ that, in addition to variable declarations, declare procedure parameters. Then the typing rule for linking to procedures is standard.

$$\frac{\Gamma, x : T, m : (x : T) \vdash B \qquad \Gamma, m : (x : T) \vdash C}{\Gamma \vdash \mathsf{let}\ m(x : T)\ \mathsf{be}\ B\ \mathsf{in}\ C}$$

We refrain from giving other typing rules as they are quite standard.

We use a small-step operational semantics in which configurations carry a *procedure environment* that binds procedure names to their bodies. Configurations take the form $\langle C,\ \sigma,\ \mu\rangle$ where C is a command, σ is a state (as defined in Sec. 3), and procedure environment μ is a partial function from procedure names to parameterized commands of the form $(\lambda x : T.C)$. We consider only well formed programs and initial configurations, so reachable configurations enjoy standard well-formedness properties that will not be discussed in detail. Every reachable configuration has a successor unless the command is skip. A terminating computation ends in a configuration of the form $\langle \mathsf{skip},\ \sigma,\ \mu\rangle$ or else *fault*.

The only unusual feature of the semantics is that it depends on assumed specifications of procedures that are not bound in the procedure environment. The identifier Δ will range over such assumptions. The semantics is given as a transition relation $\overset{\Delta}{\longmapsto}$ for commands that may have free occurrences of procedures specified in Δ. The procedures in Δ are to be distinct from those in the procedure environment. Semantics of a call $m(z)$, for m in Δ, depends on the specification; its definition is deferred (to Fig. 9) and discussed later.

The transition semantics $\overset{\Delta}{\longmapsto}$ for all other cases is independent from Δ. It is defined in Fig. 8. For allocation, we assume *fresh* is an arbitrary function such that *fresh*(σ) is a non-empty set of non-null references that are not allocated in σ. Thus our results encompass deterministic allocators as well as the maximally non-deterministic one used in separation logic. We write $\overset{\Delta}{\longmapsto}{}^*$ for the reflexive, transitive closure of $\overset{\Delta}{\longmapsto}$.

In addition to the well-formedness properties of reachable configurations that were already mentioned, we note that the *context procedures* specified in Δ are always distinct from the *linked procedures* bound in the procedure environment. Furthermore, if C is not an end then $\langle C,\ \sigma,\ \mu\rangle \overset{\Delta}{\longmapsto}{}^* \langle \mathsf{skip},\ \sigma',\ \mu'\rangle$ implies $\mu' = \mu$. There is no deallocation so the domain of the heap only grows, and once allocated the type of a reference never changes.

Procedure specifications and hypothetical judgements. A hypothesis Δ is a comma-separated list of procedure specifications, each of the form

$$\{Q\}m(x : T)\{Q'\}[\overline{\varepsilon}] \tag{4}$$

$$\frac{\sigma(x) = \mathsf{null}}{\langle x.f := F,\ \sigma,\ \mu\rangle \longmapsto \mathit{fault}}$$

$$\frac{\sigma(x) = o \qquad o \neq \mathsf{null}}{\langle x.f := F,\ \sigma,\ \mu\rangle \longmapsto \langle \mathsf{skip},\ \mathit{update}(\sigma, o.f, [\![F]\!]\sigma),\ \mu\rangle}$$

$$\frac{o \in \mathit{fresh}(\sigma) \qquad \mathit{fields}(K) = \overline{f} : \overline{T} \qquad \mathit{default}(\overline{T}) = \overline{v}}{\langle x := \mathsf{new}\ K,\ \sigma,\ \mu\rangle \longmapsto \langle \mathsf{skip},\ \mathit{new}(\sigma, o, K, \overline{v}),\ \mu\rangle}$$

$$\langle (\mathsf{skip}; C),\ \sigma,\ \mu\rangle \longmapsto \langle C,\ \sigma,\ \mu\rangle$$

$$\frac{\langle C_0,\ \sigma,\ \mu\rangle \longmapsto \langle C_0',\ \sigma',\ \mu'\rangle}{\langle (C_0; C_1),\ \sigma,\ \mu\rangle \longmapsto \langle (C_0'; C_1),\ \sigma',\ \mu'\rangle}$$

$$\langle \mathsf{let}\ m(x : T)\ \mathsf{be}\ B\ \mathsf{in}\ C,\ \sigma,\ \mu\rangle \\ \longmapsto \langle (C; \mathsf{end}(m)),\ \sigma,\ \mathit{extend}(\mu, m, (\lambda x : T.B))\rangle$$

$$\frac{\mu(m) = \lambda x : T.B \qquad x' \notin \mathit{dom}(\sigma) \qquad B' = B/x \to x'}{\langle m(z),\ \sigma,\ \mu\rangle \longmapsto \langle (B'; \mathsf{end}(x')),\ \mathit{extend}(\sigma, x', \sigma(z)),\ \mu\rangle}$$

$$\langle \mathsf{end}(x),\ \sigma,\ \mu\rangle \longmapsto \langle \mathsf{skip},\ \mathit{retract}(\sigma, x),\ \mu\rangle$$

$$\langle \mathsf{end}(m),\ \sigma,\ \mu\rangle \longmapsto \langle \mathsf{skip},\ \sigma,\ \mathit{retract}(\mu, m)\rangle$$

Figure 8. Definition of transition relation $\overset{\Delta}{\longmapsto}$. For calls of procedures in Δ, for which see Fig. 9. Here Δ is the same throughout and omitted. Also *retract*(σ, x) removes x from the state. Rules for ordinary assignment, if, and while are standard.

where m is a procedure name and x is the parameter name. For (4) to be well formed, $Q, Q', \overline{\varepsilon}$ should be wf in $\Gamma, x : T$. Less obviously, $\overline{\varepsilon}$ must not contain wr x; this enforces the usual constraint on value-parameters: so that use of x in postconditions or in effects like wr x‘f refers to its initial value).

Correctness judgements now take the form

$$\Delta \vdash^{\Gamma} \{\ P\ \}\ C\ \{\ P'\ \}\ [\overline{\varepsilon}] \tag{5}$$

Judgement (5) is well formed just if P, P', $\overline{\varepsilon}$, and all specifications in Δ are well formed in Γ. Moreover C is well formed with respect to the procedure signatures in Δ, i.e., $\Gamma; sigs(\Delta) \vdash C$ where $sigs$ extracts the procedure signatures from a set of procedure specifications.

There are two plausible interpretations of a hypothetical correctness judgement (5), both expressing that C is "modularly correct" with respect to the specifications of procedures of its context Δ. The first interpretation can be described as follows: Consider the semantics of C started from a procedure environment that links each procedure in Δ to an arbitrary command that satisfies the procedure's specification. That is, C is correct with respect to all correct implementations of Δ. The second interpretation avoids this universal quantification by interpreting C with respect to a single transition relation (not necessarily denotable by a command) that models the least refined or "worst" meaning that satisfies the specification. This embodies all possible behaviors of correct implementations. The second interpretation is popular in work on program refinement. It is used as well by O'Hearn et al [27, 29]. Although typically the second interpretation is associated with a big step semantics, we use it in mixed step form (as in, e.g., [31]).

Fig. 9 completes the definition of transition relation $\overset{\Delta}{\longmapsto}$, by giving the step from configuration $\langle m(z),\ \sigma,\ \mu\rangle$ where m is a context procedure rather than being in the environment μ. To explain the definition, let us first consider a specification $\{P\}m(x : T)\{P'\}[\overline{\varepsilon}]$. There are two cases. If σ satisfies the pre-

$$\frac{\Delta \text{ contains } \{P\}m(x:T)\{P'\}[\overline{\varepsilon}] \qquad \sigma \rightsquigarrow \sigma' \models \overline{\varepsilon} \qquad extend(\sigma,x,\sigma(z)) \models P \qquad extend(\sigma',x,\sigma(z)) \models P'}{\langle m(z),\ \sigma,\ \mu\rangle \overset{\Delta}{\longmapsto} \langle \mathsf{skip},\ \sigma',\ \mu\rangle}$$

$$\frac{\Delta \text{ contains } \{P\}m(x:T)\{P'\}[\overline{\varepsilon}] \qquad extend(\sigma,x,\sigma(z)) \not\models P}{\langle m(z),\ \sigma,\ \mu\rangle \overset{\Delta}{\longmapsto} \langle \mathsf{skip},\ \sigma',\ \mu\rangle \quad \text{and also} \quad \langle m(z),\ \sigma,\ \mu\rangle \overset{\Delta}{\longmapsto} fault}$$

Figure 9. Definition of $\overset{\Delta}{\longmapsto}$ for calls of *context procedures*, i.e., those in Δ.

$$\textsc{Call}\ \frac{\Delta \text{ contains } \{P\}m(x:T)\{P'\}[\overline{\varepsilon}] \qquad Q \equiv P/x{\to}z \qquad Q' \equiv P'/x{\to}z \qquad \overline{\varepsilon}' \equiv \overline{\varepsilon}/x{\to}z}{\Delta \vdash \{\ Q\ \}\ m(z)\ \{\ Q'\ \}\ [\overline{\varepsilon}']}$$

$$\textsc{Link}\ \frac{\Theta \text{ is } \{Q\}m(x:T)\{Q'\}[\overline{\delta}] \qquad \Delta,\Theta \vdash^{\Gamma} \{\ P\ \}\ C\ \{\ P'\ \}\ [\overline{\varepsilon}] \qquad \Delta,\Theta \vdash^{\Gamma,x:T} \{\ Q\ \}\ B\ \{\ Q'\ \}\ [\overline{\delta}]}{\Delta \vdash^{\Gamma} \{\ P\ \}\ \mathsf{let}\ m(x:T)\ \mathsf{be}\ B\ \mathsf{in}\ C\ \{\ P'\ \}\ [\overline{\varepsilon}]}$$

Figure 10. Proof Rules for hypothetical judgements.

condition, then the configuration can step to $\langle \mathsf{skip},\ \sigma',\ \mu\rangle$ for any σ' such that σ' satisfies P' and moreover $\overline{\varepsilon}$ allows the transition from σ to σ'. If σ does not satisfy the precondition, then the configuration can step to any σ' whatsoever, and may also step to *fault*.

A correctness judgement is *valid*, written $\Delta \models^{\Gamma} \{\ P\ \}\ C\ \{\ P'\ \}[\overline{\varepsilon}]$, iff the following holds. Let μ_0 be an arbitrary procedure environment disjoint from the procedures bound in C or present in Δ. Let $C_0 = C$. Then for all Γ-states σ_0 such that $\sigma_0 \models P$ we require (i) It is not the case that $\langle C_0,\ \sigma_0,\ \mu_0\rangle \overset{\Delta}{\longmapsto}{}^{*} fault$; and (ii) For all computations $\langle C_0,\ \sigma_0,\ \mu_0\rangle \overset{\Delta}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \sigma_n,\ \mu_n\rangle$ of n steps (noting that $n \geq 0$ and $\mu_n = \mu_0$) we have both $\sigma_n \models P'$ and $\sigma_0 \rightsquigarrow \sigma_n \models \overline{\varepsilon}$. The new proof rules are given in Fig. 10. The proof rules of Sect. 3 are all adapted by adding hypotheses Δ to all correctness judgements.

Extension to a richer language. In examples we use an extended programming language, where methods have an implicit parameter named self and may have a return value, and there are constructor methods. Formalization of the additional features complicates the rules ALLOC, LINK, and CALL. Calls without return value take the form $y.m(z)$. The specification may mention both self and parameter x, so y is substituted for self (and z for x as before). With a return value, there is an additional substitution.

The LINK rule no longer has an exact match between the specification used by the client and the specification with respect to which a method implementation is verified. That is because a method call involves first testing nullity of the receiver and only then dispatching the body of the method; so precondition $\mathsf{self} \neq \mathsf{null}$ is added. For the constructor of some class K, the proof obligation also reflects in its precondition that the fields are initialized to defaults before the constructor runs; the effect specification reflects that self is freshly allocated and thus writes to fields of self must not be reported.

5. Module invariants and second order framing

This section augments hypotheses with dynamic boundaries. A dynamic boundary $\overline{\delta}$ is associated with a group (list) Δ of procedure specifications using notation $\Delta\langle\overline{\delta}\rangle$. The general form for correctness judgement would have a sequence $\Delta_0\langle\overline{\delta}_0\rangle\ ;\ \ldots\ ;\ \Delta_n\langle\overline{\delta}_n\rangle$ of hypotheses, and rules for permuting the groups. But for readability it suffices to spell out the case of two groups. So a correctness judgement has the form

$$\Delta\langle\overline{\delta}\rangle\ ;\ \Theta\langle\overline{\theta}\rangle \vdash^{\Gamma} \{\ P\ \}\ C\ \{\ P'\ \}\ [\overline{\varepsilon}] \tag{6}$$

The judgement is wf if the conditions for (5) hold and the read effects $\overline{\delta}$ and $\overline{\theta}$ are wf in Γ.[10]

The current command in a configuration can always be written as a sequence of one or more commands that are not themselves sequences; the first is the *active command*, the one that is rewritten in the next step. Formally we define $active(C_1; C_2) = active(C_1)$ and $active(C) = C$ if there are no C_1, C_2 such that C is $C_1; C_2$.

Definition 4 A correctness judgement is *valid*, written

$$\Delta\langle\overline{\delta}\rangle\ ;\ \Theta\langle\overline{\theta}\rangle \models^{\Gamma} \{\ P\ \}\ C\ \{\ P'\ \}\ [\overline{\varepsilon}]$$

iff the following holds. Let Δ' be the union (Δ, Θ), let C_0 be C, and let μ_0 be an arbitrary procedure environment disjoint from the procedures linked within C or present in Δ, Θ. Then for all Γ-states σ_0 such that $\sigma_0 \models P$

- It is not the case that $\langle C_0,\ \sigma_0,\ \mu_0\rangle \overset{\Delta'}{\longmapsto}{}^{*} fault$.
- For all computations $\langle C_0,\ \sigma_0,\ \mu_0\rangle \overset{\Delta'}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \sigma_n,\ \mu_n\rangle$
  (a) $\sigma_n \models P'$ and $\sigma_0 \rightsquigarrow \sigma_n \models \overline{\varepsilon}$
  (b) for all $0 < i \leq n$, either $active(C_{i-1})$ is a call to some procedure m in Δ or else $Agree(\sigma_{i-1}, \sigma_i, \overline{\delta})$
  (c) like (b) but *mutatis mutandis* for Θ and $\overline{\theta}$.

Note that in the case $\overline{\delta}$ and $\overline{\theta}$ are empty, this coincides with the definition of validity in Sec. 4.

The axioms for assignment forms are now given hypotheses $\Delta\langle\rangle; \Theta\langle\rangle$ with empty dynamic bounds. Proper rules are revised to have $\Delta\langle\overline{\delta}\rangle; \Theta\langle\overline{\theta}\rangle$ in both antecedents and conclusions, except that CALL and LINK are replaced by the ones in Fig. 11. Note that LINK' requires the procedure body to respect the other dynamic bounds and linked procedures must have empty dynamic bound. In addition, the Figure adds rules used to introduce and eliminate the dynamic boundary.

Rule SOF uses an operation $\circledast I$ that conjoins a formula I to pre- and post-conditions of specifications. It is defined by

$$(\{Q\}m(x:T)\{Q'\}[\overline{\varepsilon}]) \circledast I = \{Q \wedge I\}m(x:T)\{Q' \wedge I\}[\overline{\varepsilon}]$$

Define $\Delta \circledast I$ by distributing $\circledast I$ over the specifications in Δ. Rule SOF also imposes a mild admissibility condition on I. The problem is that some useful invariants include alloc in their effect, e.g., if in the example of Sec. 2 we drop variable *pool* and instead let I quantify over all allocated Sets. Typical clients do allocation, and thus write alloc, which would conflict with a dynamic boundary containing $\mathsf{rd\ alloc}$. The solutions are based on the limited way that alloc gets written: only by the new command and only by adding the newly allocated object.

Our first solution to the above problem is similar to one used in some specific invariant disciplines [26, 32].

Definition 5 Formula Q is *admissible* with respect to the procedure specifications Δ, written $admiss(Q, \Delta)$, iff it is not falsifiable by allocation. That is, for any σ with $\sigma \models Q$, we also have $\sigma' \models Q$ where σ' is $new(\sigma, o, K, default(\overline{T}))$ and $\overline{T}$ is the field types for K and o is any fresh reference (not in $alloc(\sigma)$).

[10] In a proof, the dynamic bounds will appear in many judgements, including those for procedure implementations and in the scope of local variables. Thus to be useful they must be well formed in a number of contexts — typically, they involve only some global variables and public field names.

$$\text{SOF}\ \frac{\Delta\langle\overline{\delta}\rangle\,;\,\Theta\langle\overline{\theta}\rangle \vdash \{\,P\,\}\ C\ \{\,P'\,\}\ [\overline{\varepsilon}] \qquad I \vdash (\overline{\theta}, \mathsf{rd\,alloc})\ \mathsf{frm}\ I \qquad \mathit{admiss}(I,\Theta)}{\Delta\langle\overline{\delta}\rangle\,;\,(\Theta \circledwedge I)\langle\overline{\theta}\rangle \vdash \{\,P \wedge I\,\}\ C\ \{\,P' \wedge I\,\}\ [\overline{\varepsilon}]}$$

$$\text{DynBndElim}\ \frac{\Delta\langle\overline{\delta}\rangle\,;\,\Theta\langle\overline{\theta}\rangle \vdash \{\,P\,\}\ C\ \{\,P'\,\}\ [\overline{\varepsilon}]}{\Delta\langle\overline{\delta}\rangle\,;\,\Theta\langle\rangle \vdash \{\,P\,\}\ C\ \{\,P'\,\}\ [\overline{\varepsilon}]}$$

$$\text{DynBndIntro}\ \frac{\Delta\langle\overline{\delta}\rangle\,;\,\Theta\langle\rangle \vdash \{\,P\,\}\ C\ \{\,P'\,\}\ [\overline{\varepsilon}] \qquad C \text{ is primitive} \qquad P \Rightarrow \overline{\theta} \star \overline{\varepsilon}}{\Delta\langle\overline{\delta}\rangle\,;\,\Theta\langle\overline{\theta}\rangle \vdash \{\,P\,\}\ C\ \{\,P'\,\}\ [\overline{\varepsilon}]}$$

$$\text{DynBndIntroM}\ \frac{\Delta\langle\overline{\delta}\rangle\,;\,\Theta\langle\rangle \vdash \{\,P\,\}\ m(z)\ \{\,P'\,\}\ [\overline{\varepsilon}] \qquad m \text{ is in } \Theta}{\Delta\langle\overline{\delta}\rangle\,;\,\Theta\langle\overline{\theta}\rangle \vdash \{\,P\,\}\ m(z)\ \{\,P'\,\}\ [\overline{\varepsilon}]}$$

$$\text{Call'}\ \frac{\Delta \text{ or } \Theta \text{ contains } \{P\}m(x:T)\{P'\}[\overline{\varepsilon}] \qquad Q \equiv P/x{\to}z \qquad Q' \equiv P'/x{\to}z \qquad \overline{\varepsilon}' \equiv \overline{\varepsilon}/x{\to}z}{\Delta\langle\overline{\delta}\rangle\,;\,\Theta\langle\rangle \vdash \{\,Q\,\}\ m(z)\ \{\,Q'\,\}\ [\overline{\varepsilon}']}$$

$$\text{Link'}\ \frac{\Theta \text{ is } \{Q\}m(x:T)\{Q'\}[\overline{\eta}] \qquad \Delta\langle\overline{\delta}\rangle\,;\,\Theta\langle\rangle \vdash^{\Gamma} \{\,P\,\}\ C\ \{\,P'\,\}\ [\overline{\varepsilon}] \qquad \Delta\langle\overline{\delta}\rangle\,;\,\Theta\langle\rangle \vdash^{\Gamma,x:T} \{\,Q\,\}\ B\ \{\,Q'\,\}\ [\overline{\eta}]}{\Delta\langle\overline{\delta}\rangle \vdash^{\Gamma} \{\,P\,\}\ \mathsf{let}\ m(x:T)\ \mathsf{be}\ B\ \mathsf{in}\ C\ \{\,P'\,\}\ [\overline{\varepsilon}]}$$

Figure 11. Rules involving dynamic boundaries. Omitted is a rule that swaps $\Delta\langle\overline{\delta}\rangle\,;\,\Theta\langle\overline{\theta}\rangle$ with $\Theta\langle\overline{\theta}\rangle\,;\,\Delta\langle\overline{\delta}\rangle$, unconditionally.

This is formulated as a semantic property of Q, unlike all the other side conditions in the proof rules. We include Δ in the notation, even though it is not used in the definition, in order to highlight the way admissibility should be treated in a richer language that includes constructor methods. Before discussing that further, we note the following which follows immediately using Lemma 2.

Lemma 6 Suppose $P \vdash (\overline{\delta}, \mathsf{rd\,alloc})\ \mathsf{frm}\ Q$ and $\mathit{admiss}(Q,\Delta)$. Suppose $\sigma \models P \wedge Q$ and let $\sigma' = \mathit{new}(\sigma, o, K, \overline{v})$. If $\mathit{Agree}(\sigma, \sigma', \overline{\delta})$ then $\sigma' \models Q$.

The point is that by condition $P \vdash (\overline{\delta}, \mathsf{rd\,alloc})\ \mathsf{frm}\ Q$, it appears that Q depends on alloc, but by semantic condition $\mathit{admiss}(Q,\Delta)$ it does not.

Suppose Q is of the form $\forall x : K \in \mathsf{alloc} \mid x.init \Rightarrow P(x)$ with $init$ a boolean field. Although the footprint of Q includes alloc, it is not falsifiable by allocation (because the default value for $x.init$ is false). Such a formula would be suitable as an invariant in a program where $x.init$ only gets truthified by procedures that also establish $P(x)$. In a richer language with constructor methods, this pattern is in some sense part of the semantics and we can define $\mathit{admiss}(Q,\Delta)$ by the following syntactic condition: every occurrence of alloc in Q occurs in a subformula of the form $\forall x : K \in \mathsf{alloc} \mid \ldots$ where the constructor for class K is among the procedures of Δ.

A verification condition generator could impose the conditions of rule DynBndIntro on each primitive in a program. However, for general purpose encapsulation disciplines based on type systems or other static analyses, a dynamic bound $\overline{\eta}$ could be introduced by a rule of the form

$$\frac{\Delta\langle\overline{\delta}\rangle\,;\,\Theta\langle\overline{\theta}\rangle \vdash^{\Gamma} \{\,P\,\}\ C\ \{\,P'\,\}\ [\overline{\varepsilon}] \qquad \mathit{Ok}(\overline{\eta}, C, \Gamma, \Delta\langle\overline{\delta}\rangle, \Theta\langle\overline{\theta}\rangle, P)}{\Delta\langle\overline{\delta}\rangle\,;\,\Theta\langle\overline{\theta}, \overline{\eta}\rangle \vdash^{\Gamma} \{\,P\,\}\ C\ \{\,P'\,\}\ [\overline{\varepsilon}]}$$

It posits a condition $\mathit{Ok}(\overline{\eta}, C, \Gamma, \Delta\langle\overline{\delta}\rangle, \Theta\langle\overline{\theta}\rangle, P)$ which performs some static analysis of C, taking into account that methods of Θ are exempt from the effect bound $\overline{\theta}$. What is required of Ok is that it makes this rule sound, i.e., it ensures that every step of C, other than calls to methods in Θ, respects the bound $\overline{\eta}$.

We leave this topic and turn to the main result of the paper.

Theorem 7 Any judgement $\Delta\langle\overline{\delta}\rangle\,;\,\Theta\langle\overline{\theta}\rangle \vdash^{\Gamma} \{\,P\,\}\ C\ \{\,P'\,\}\ [\overline{\varepsilon}]$ that is derivable is valid, i.e., $\Delta\langle\overline{\delta}\rangle\,;\,\Theta\langle\overline{\theta}\rangle \models^{\Gamma} \{\,P\,\}\ C\ \{\,P'\,\}\ [\overline{\varepsilon}]$.

Proof: By induction on the derivation and by cases on the last rule used. Each case simply appeals to Lemma 9. □

Proposition 8 For all $C, C', \sigma, \sigma', \mu, \mu', \Delta, \Delta'$, such that Δ and Δ' declare the same methods and $\mathit{active}(C)$ is not a call to a method in Δ, we have: $\langle C,\ \sigma,\ \mu\rangle \xmapsto{\Delta} \langle C',\ \sigma',\ \mu'\rangle$ if and only if $\langle C,\ \sigma,\ \mu\rangle \xmapsto{\Delta'} \langle C',\ \sigma',\ \mu'\rangle$.

Proof of the proposition is by induction on the structure of C; under the given conditions, only the transition rules in Fig. 8 are at play.

Lemma 9 (rule soundness) Every axiom is valid. For every proper rule, derives valid conclusions from valid antecedents.

Proof: By cases on the axioms and rules. We give the main case, SOF. Assume the antecedent is valid, $\Delta\langle\overline{\delta}\rangle\,;\,\Theta\langle\overline{\theta}\rangle \models^{\Gamma} \{\,P\,\}\ C\ \{\,P'\,\}\ [\overline{\varepsilon}]$. To prove validity of the conclusion, i.e.,

$$\Delta\langle\overline{\delta}\rangle\,;\,(\Theta \circledwedge I)\langle\overline{\theta}\rangle \models \{\,P \wedge I\,\}\ C\ \{\,P' \wedge I\,\}\ [\overline{\varepsilon}] \qquad (7)$$

let Δ_c be the hypotheses $(\Delta, \Theta \circledwedge I)$, for the conclusion of the rule, and let Δ_a be the hypotheses (Δ, Θ) for the antecedent. Consider any σ_0 with $\sigma_0 \models P \wedge I$, let μ_0 be any method environment disjoint from Δ, Θ, and let C_0 be C.

Claim A: Consider any computation from $\langle C_0,\ \sigma_0,\ \mu_0\rangle$ under transition relation $\xmapsto{\Delta_c}$. Then (a) that sequence of configurations is also a computation of $\xmapsto{\Delta_a}$ and (b) if $\sigma_0 \models I$ then I holds in every configuration. Claim A implies (7) as follows. It is not the case that $\langle C_0,\ \sigma_0,\ \mu_0\rangle \xmapsto{\Delta_c}{}^{*} \mathit{fault}$, because that would imply $\langle C_0,\ \sigma_0,\ \mu_0\rangle \xmapsto{\Delta_a}{}^{*} \mathit{fault}$ which contradicts validity of the antecedent. Furthermore, by validity of the antecedent we get $\sigma_n \models P'$ and $\sigma_0 \rightsquigarrow \sigma_n \models \overline{\varepsilon}$. We get $\sigma_n \models I$ from the claim. What remains is to show that for all $0 < i \leq n$, either $\mathit{active}(C_{i-1})$ is a call to some method m in $(\Theta \circledwedge I)$ or else $\mathit{Agree}(\sigma_{i-1}, \sigma_i, \overline{\theta})$, and *mutatis mutandis* for Δ and $\overline{\delta}$. This follows immediately from the corresponding condition in validity of the antecedent.

It remains to prove Claim A, which is by induction on the length of the computation sequence for $\langle C_0,\ \sigma_0,\ \mu_0\rangle \xmapsto{\Delta_c}{}^{*} \langle C_n,\ \sigma_n,\ \mu_n\rangle$. The induction step uses **Claim B:** For any $D, D', \sigma, \sigma', \mu, \mu'$, suppose that $\langle D,\ \sigma,\ \mu\rangle$ is reachable under $\xmapsto{\Delta_a}$ from an initial configuration $\langle C_0,\ \mu_0,\ \sigma_0\rangle$. Suppose $\sigma \models I$ and $\langle D,\ \sigma,\ \mu\rangle \xmapsto{\Delta_c} \langle D',\ \sigma',\ \mu'\rangle$. Then $\langle D,\ \sigma,\ \mu\rangle \xmapsto{\Delta_a} \langle D',\ \sigma',\ \mu'\rangle$ and $\sigma' \models I$. To prove Claim B there are 3 cases: **(a)** If $\mathit{active}(D)$ is not a call to a context method (noting that Δ_a and Δ_c declare the same methods) then by Proposition 8, $\langle D,\ \sigma,\ \mu\rangle \xmapsto{\Delta_a} \langle D',\ \sigma',\ \mu'\rangle$. If $\sigma' \neq \mathit{new}(\sigma, o, K, \overline{v})$ then $\mathit{alloc}(\sigma) = \mathit{alloc}(\sigma')$ and from $\mathit{Agree}(\sigma, \sigma', \overline{\theta})$ (by valid antecedent) and $\sigma \models I$, we get $\sigma' \models I$ by Lemma 2. In case $\sigma' = \mathit{new}(\sigma, o, K, \overline{v})$, from $\sigma \models I$ and side conditions $I \vdash \mathsf{rd\,alloc}, \overline{\theta}\ \mathsf{frm}\ I$ and $\mathit{admiss}(I, \Theta)$ of SOF, we get $\sigma' \models I$ by Lemma 6.

For the second and third cases, suppose that $\mathit{active}(D)$ is a call to a method m with specification $\{V\}m(x:T)\{V'\}[\overline{\eta}]$.

Case **(b)** $m \in \Delta$: It must be that $\sigma \in [\![V]\!]$. For, if not, then by context call semantics (Fig. 9) we would have $\langle D,\ \sigma,\ \mu\rangle \xmapsto{\Delta_a} \mathit{fault}$

which, because $\langle D,\ \sigma,\ \mu\rangle$ is reachable, contradicts the antecedent $\Delta\langle\overline{\delta}\rangle\ ;\ \Theta\langle\overline{\theta}\rangle \models^{\Gamma} \{\ P\ \}\ C\ \{\ P'\ \}\ [\overline{\varepsilon}]$. So by the first rule in Fig. 9 for both Δ_a and Δ_c we get $\langle D,\ \sigma,\ \mu\rangle \xmapsto{\Delta_a} \langle D',\ \sigma',\ \mu'\rangle$. We show that I is maintained, by cases on whether $\sigma' = \mathit{new}(\sigma, o, K, \overline{v})$. If so, from $\sigma \models I$ and side conditions $I \vdash \mathsf{rd\,alloc}, \overline{\theta}\ \mathsf{frm}\ I$ and $\mathit{admiss}(I, \Theta)$ we get $\sigma' \models I$ by Lemma 6. Otherwise, from $\mathit{Agree}(\sigma, \sigma', \overline{\theta})$ and $\sigma \models I$, we have $\sigma' \models I$ by Lemma 2.

Case **(c)** $m \in \Theta \circledwedge I$. Then V is $Q \circledwedge I$ and V' is $Q' \circledwedge I$ for some Q, Q' such that $\{Q\}m(x:T)\{Q'\}[\overline{\eta}]$ is in Θ. It must be that $\sigma \models Q$: for, if not, then by method call semantics (Fig. 9) we would have $\langle D,\ \sigma,\ \mu\rangle \xmapsto{\Delta_a} \mathit{fault}$, contradicting $\Delta\langle\overline{\delta}\rangle\ ;\ \Theta\langle\overline{\theta}\rangle \models^{\Gamma} \{\ P\ \}\ C\ \{\ P'\ \}\ [\overline{\varepsilon}]$. Because $\sigma \models Q$, by specification of m in $\Theta \circledwedge I$ we get $\sigma' \models Q' \wedge I$.[11] □

Case DynBndIntro. For DYNBNDINTRO, suppose C is a primitive command and the side condition $P \Rightarrow \overline{\theta} \star \overline{\varepsilon}$ holds. Suppose the antecedent is valid: $\Delta\langle\overline{\delta}\rangle\ ;\ \Theta\langle\rangle \models \{\ P\ \}\ C\ \{\ P'\ \}\ [\overline{\varepsilon}]$. To show validity of the conclusion, $\Delta\langle\overline{\delta}\rangle\ ;\ \Theta\langle\overline{\theta}\rangle \models \{\ P\ \}\ C\ \{\ P'\ \}\ [\overline{\varepsilon}]$, consider any μ and any σ with $\sigma \models P$. If C is skip then there is no transition and the only thing to prove is $\sigma \models P'$ — which we have by validity of the antecedent. If C is an assignment, field update, or call of a method, the possible transitions have the form $\langle C,\ \sigma,\ \mu\rangle \longmapsto \langle\mathsf{skip},\ \sigma',\ \mu'\rangle$ (by the antecedent we do not have to consider faults) and by the antecedent we have $\sigma' \models P'$. To show $\mathit{Agree}(\sigma, \sigma', \overline{\theta})$, we can use Lemma 3 owing to side condition $P \Rightarrow \overline{\theta}\star\overline{\varepsilon}$. The fact that $\overline{\delta}$ is respected follows from the antecedent.

Case Link. Suppose Θ is $\{Q\}m(x:T)\{Q'\}[\overline{\eta}]$. Suppose both antecedents of the rule are valid, i.e., $\Delta\langle\overline{\delta}\rangle\ ;\ \Theta\langle\rangle \models^{\Gamma} \{\ P\ \}\ C\ \{\ P'\ \}\ [\overline{\varepsilon}]$ and $\Delta\langle\overline{\delta}\rangle\ ;\ \Theta\langle\rangle \models^{\Gamma, x:T} \{\ Q\ \}\ B\ \{\ Q'\ \}\ [\overline{\eta}]$. To show

$$\Delta\langle\overline{\delta}\rangle \models^{\Gamma} \{\ P\ \}\ \mathsf{let}\ m(x:T)\ \mathsf{be}\ B\ \mathsf{in}\ C\ \{\ P'\ \}\ [\overline{\varepsilon}]$$

suppose $\sigma_0 \models P$ and μ_0 be any environment. Let C_0 be $\mathsf{let}\ m(x:T)\ \mathsf{be}\ B\ \mathsf{in}\ C$. We must show that (a) it is not the case that $\langle C_0,\ \sigma_0,\ \mu_0\rangle \xmapsto{\Delta}{}^{*} \mathit{fault}$; and (b) If $\langle C_0,\ \sigma_0,\ \mu_0\rangle \xmapsto{\Delta}{}^{*} \langle\mathsf{skip},\ \sigma_n,\ \mu_n\rangle$ then $\sigma_n \models P'$ and $\sigma_0 \rightsquigarrow \sigma_n \models \overline{\varepsilon}$ and $\mathit{Agree}(\sigma_{i-1}, \sigma_i, \overline{\delta})$ for all steps $\sigma_i \longmapsto \sigma_i$. Validity of the antecedents tells us about $\xmapsto{\Delta'}$, where Δ' is Δ, Θ. We must therefore relate $\xmapsto{\Delta'}$ to $\xmapsto{\Delta}$. By semantics, the first step is $\langle C_0,\ \sigma_0,\ \mu_0\rangle \xmapsto{\Delta} \langle C_1,\ \sigma_1,\ \mu_1\rangle$ where C_1 is C, $\sigma_1 = \sigma_0$, and μ_1 extends μ_0 to map m to $\lambda x : T.B$. It respects $\overline{\delta}$ because the state is unchanged. Subsequent steps are matched exactly by steps via $\xmapsto{\Delta'}$ —which respect $\overline{\delta}$ by validity of the antecedent for C— until we reach a configuration $\langle C_i,\ \sigma_i,\ \mu_i\rangle$ where the active command in C_i is a call of m, i.e., C_i is a sequence $m(z); D$, for some D. In this case, $\xmapsto{\Delta'}$ does not fault (as otherwise the antecedent for C would not be valid). Thus by definition of $\xmapsto{\Delta'}$ it must be that $\sigma \models Q$ (as otherwise it could fault). Instead, it goes in one step to $\langle\mathsf{skip}; D,\ \sigma',\ \mu_i\rangle$ for all σ' such that $\sigma' \models Q'$ and $\sigma_i \rightsquigarrow \sigma' \models \overline{\eta}$. The next step goes to $\langle D,\ \sigma',\ \mu_i\rangle$, which we will show is matched.

As for $\xmapsto{\Delta}$, it steps first to $\langle(B'; \mathsf{end}(x'); D),\ \sigma'',\ \mu_i\rangle$ where $\mu_i(m) = \lambda x : T.B$ and B' has x renamed to fresh identifier x' that is initialized in σ'' to $\sigma_i(z)$. From there, by validity of the antecedent for B and using $\sigma \models P$, there is no fault and if the overall computation terminates then it first reaches the end of the B'-computation, i.e., a configuration $\langle(\mathsf{skip}; \mathsf{end}(x'); D),\ \sigma',\ \mu'\rangle$, and moreover here $\sigma' \models P'$ and $\sigma_i \rightsquigarrow \sigma' \models \overline{\eta}$. Also, by hypothesis for B, $\overline{\delta}$ is respected at each step. After a couple more steps we reach $\langle D,\ \sigma'',\ \mu'\rangle$ where $\sigma'' = \mathit{retract}(x', \sigma')$. This is one of the possible configurations $\langle D,\ \sigma',\ \mu_i\rangle$ reached by $\xmapsto{\Delta'}$ so we are at a matching point in the computations and can proceed by induction on the rest of the computation of $\xmapsto{\Delta}$.

6. Examples

Toy memory manager. This example is adapted from the one of O'Hearn et al [27] who use it to exemplify ownership transfer. It also exemplifies situations where clients retain access to locations on which which they are not currently allowed to act. Those references must be known when reasoning about the client. This might be done by tracking a "typestate" that distinguishes between, e.g., updatable and non-updatable phases of an object's life [5]. We will do it by tracking the set of objects that are currently in the "freed" state. This is a "static" module, i.e., unlike our other examples it is not individual instances of a class that represent an abstraction, but rather some global variables.

$\mathsf{global\text{-}vars}\ \mathit{result} : \mathit{Node};\ \ \mathit{freed} : \mathsf{rgn};$
$\mathsf{module\text{-}vars}\ \mathit{flist} : \mathit{Node};\ \ \mathit{count} : \mathsf{int};$
$\mathit{alloc}()$
$\{\ \mathsf{if}\ \mathit{count} = 0\ \mathsf{then}\ \mathit{result} := \mathsf{new}\ \mathit{Node}();$
$\quad \mathsf{else}\ \mathit{freed} := \mathit{freed} - \{\mathit{flist}\};$
$\qquad \mathit{result} := \mathit{flist};\ \mathit{flist} := \mathit{flist}.\mathit{nxt};\ \mathit{count}\text{-=}1;\ \}$
$\mathit{free}(n : \mathit{Node})$
$\{\ n.\mathit{nxt} := \mathit{flist};\ \mathit{flist} := n;\ \mathit{count}\text{+=}1;\ \mathit{freed} := \mathit{freed} \cup \{n\};\ \}$

The interface specifications are in Fig. 12. The module invariant is I_M defined as $FC(\mathit{flist}, \mathit{freed}, \mathit{count})$. Here FC is defined by induction on the size of its region parameter:

$$\begin{aligned} &FC(f : \mathit{Node}, r : \mathsf{rgn}, c : \mathit{int}) : \\ &\quad (f = \mathsf{null} \Rightarrow r = \varnothing \wedge c = 0) \\ &\quad \wedge (f \neq \mathsf{null} \Rightarrow f \in r \wedge c > 0 \wedge FC(f.\mathit{nxt}, r - \{f\}, c - 1)) \end{aligned}$$

So I_M says freed is the nodes reached from flist and count is the size. As dynamic boundary we choose

$\overline{\delta}_M :\ \ \mathsf{rd}\ \mathit{freed}, \mathit{freed}\text{`}\mathit{nxt}$

To be precise, the read footprint of I_M includes also flist and count, but we shall assume they are hidden from the client by ordinary scoping.

The implementation of alloc relies on accuracy of count. In particular, it relies on $\mathit{count} \neq 0 \Rightarrow \mathit{flist} \neq \mathsf{null}$ (as otherwise the dereference $f.\mathit{nxt}$ could fault), but for this to hold on subsequent calls the stronger condition I_M needs to be maintained as invariant.

Consider this strange client that both reads and writes data in the free list —but not in a way that interferes with the module.

$\mathsf{var}\ x, y : \mathit{Node};\quad x := \mathsf{new}\ \mathit{Node}();$
$\{x \notin \mathit{freed}\}$
$\mathit{alloc}();\ y := \mathit{result};$
$\{x \notin \mathit{freed} \wedge y \notin \mathit{freed}\}\quad$ // by spec of alloc
$\mathit{free}(x);\ \mathit{free}(y);$
$\mathsf{while}\ y \neq \mathsf{null}\ \{\ y.\mathit{val} := 7;\ y := y.\mathit{nxt};\ \}$
$\{x \in \mathit{freed} \wedge y \in \mathit{freed}\}$

This is provable using the specifications (Fig. 12) and rules DYNBNDINTRO and DYNBNDINTROM. The point is that although the loop updates val, that effect is separate from the dynamic boundary, $\overline{\delta}_M$.

Combining modules. Consider a client program that uses both the memory manager and the Set module of Sec. 2:

$\mathsf{let}\ \mathit{alloc}, \mathit{free}\ \mathsf{be}\ \ldots\ \mathsf{in}\ \mathsf{let}\ \mathit{add}, \mathit{contains}, \mathit{remove}\ \mathsf{be}\ \ldots\ \mathsf{in}\ \mathit{Cli}$

[11] In part (c) of the proof of the Claim B, the argument would also apply to a σ' arising as a result of a call to a constructor method. The postcondition of such a method must be of the form $Q' \circledwedge I$ and thus I must be established by σ' (replacing the appeal to Lemma 6).

Method	Pre-condition	Post-condition	Effects
$alloc()$	true	$result \neq \mathsf{null} \wedge freed = \mathsf{old}(freed) - result$	$\mathsf{wr}\ result, freed, \mathsf{alloc}, freed\text{`}nxt$
$free(n : Node)$	$n \neq \mathsf{null} \wedge n \notin freed$	$freed = \mathsf{old}(freed) \cup \{n\}$	$\mathsf{wr}\ freed, freed\text{`}nxt$

Figure 12. Interface specifications for memory manager. Dynamic boundary: $\mathsf{rd}\ freed, freed\text{`}nxt$

(eliding the parameter lists, implementations, and constructor to save space). Let us write Δ for the two specifications in Fig. 12 and Θ for the four in Fig. 2. The main program Cli is verified in context $\Delta\langle\overline{\delta}_M\rangle; \Theta\langle\overline{\theta}_I\rangle$, i.e., it must respect the dynamic boundary $\overline{\delta}_M$ for the memory manager and $\overline{\theta}_I$ for the Set module. The implementations of add, $contains$, etc. are verified in context $\Delta\langle\overline{\delta}_M\rangle; \Theta\langle\rangle$, i.e., they may use the memory manager but must respect its dynamic boundary. Finally, $alloc$ and $free$ are verified in context $\Delta\langle\rangle$.

Observer pattern. This example illustrates the verification of a client of the Observer pattern (Figs. 13, 14). A subject has a list of observers cognizant of its internal state (here represented as an integer value held in field val of a subject). A new observer registers itself with a subject (the observer's sub field tracks its subject) and is notified of its subject's state which it caches in field $cacheVal$. When a subject's state is updated it notifies each of its observers. Each observer then calls back into the subject interface and gets the new value.

A subject and its observers together form a cooperating cluster of objects that are not in an ownership relation. Accordingly we can consider a module containing both classes $Subject$ and $Observer$ whose interface exports the methods $update$, get and the constructor for $Observer$[12]. In Fig. 13, $s.O$ is a region containing the observers of subject s. The subject employs a list data structure to manage its observers — this is hidden from clients as is the list header, $s.obs$. Let $SubH(s)$ be the predicate $List(s.obs, s.O)$ that says "for subject s, list beginning at $s.obs$ lies in region $s.O$". The exact definition of $List$ is immaterial here (but see [3]); it suffices to know that the read effects of $SubH(s)$ are $\mathsf{rd}\ s, \{s\}\text{`}(obs, O), \{s\}\text{`}O\text{`}nxt$.

Hidden invariant, I, is defined as $I(\varnothing, \varnothing)$ where $I(m, n)$ is

$$(\forall s : Subject \in \mathsf{alloc} - m \mid SubH(s)) \wedge$$
$$(\forall o : Observer \in \mathsf{alloc} - n \mid o.sub \neq \mathsf{null} \Rightarrow o \in o.sub\text{`}O)$$

The second conjunct says that any observer, o, with non-null subject $o.sub$, is in the subject's O region. The specifications use the predicate $Obs(b, s, v):\ b.sub = s \wedge b.cacheVal = v$. Its read effect is $\mathsf{rd}\ b, s, v, \{b\}\text{`}(sub, cacheVal)$.

Suppose there are two subjects s, t with $s.val = 0$ and $t.val = 5$. Here is a client, C:

$$o := \mathsf{new}\ Observer(s); p := \mathsf{new}\ Observer(t); s.update(2);$$

Then we can show, using only the specifications in Fig. 14 that after the call to update, $p.cacheVal = 5$ while $o.cacheVal = 2$.
Let $P :\ \ \forall b : Observer \in \{s\}\text{`}O \mid Obs(b, s, 0)$
$P' :\ \ \forall b : Observer \in \{s\}\text{`}O \mid Obs(b, s, 2)$
$Q :\ \ \forall b : Observer \in \{t\}\text{`}O \mid Obs(b, t, 5)$
$W :\ \ P \wedge Q \wedge s.val = 0 \wedge t.val = 5$
in the proof outline

[12] It also exports the constructor for $Subject$ but we elide it. The other procedures are module scoped.

```
class Subject{obs : Observer; val : int; O : rgn;
  Subject(){obs := null; val := 0; O := emp; }
  register(b : Observer){add(b); b.notify(); }
  update(n : int){val := n;  b : Observer := obs;
                  while b ≠ null{b.notify();  b := b.nxt; }}
  get() : int {return self.val; }
  add(b : Observer){O := O∪{b}; b.nxt := obs;  obs := b; }}

class Observer{sub : Subject; cacheVal : int; nxt : Observer;
  Observer(u : Subject){sub := u; u.register(self); }
  notify(){cacheVal := sub.get(); }}
```

Figure 13. Subject/Observer implementation.

$\{W\}$
$\{P \wedge Q \wedge s \neq t \ldots\}$ // by CONSEQ
$o := \mathsf{new}\ Observer(s);$
$\{P \wedge Q \wedge o \in \{s\}\text{`}O\}$ // by $s \neq t$, spec of $Observer$, FRAME
$p := \mathsf{new}\ Observer(t);$
$\{P \wedge Q \wedge o \in \{s\}\text{`}O \wedge p \in \{t\}\text{`}O \ldots\}$ // ditto
$s.update(2);$
$\{P' \wedge Q \wedge o \in \{s\}\text{`}O \wedge p \in \{t\}\text{`}O \ldots\}$ // $s \neq t$, $update$, FRAME
$\{o.cacheVal = 2 \wedge p.cacheVal = 5 \ldots\}$ // by CONSEQ

Verification of Cli: let $get()$ be ...
$update(n : \mathsf{int})$ be ...
$Observer(u : Subject)$ be ... in C

requires showing (by LINK') $\vdash \{\ W \wedge I\ \}\ Cli\ \{\ W' \wedge I\ \}\ [\overline{\varepsilon}]$ where W' is $s.val = 2 \wedge t.val = 5$ and $\overline{\varepsilon}$ is $\mathsf{wr}\ o, p, \mathsf{alloc}$, $\mathsf{wr}\ \{s\}\text{`}(O, val, dg), s.O\text{`}(nxt, cacheVal), \{t\}\text{`}(O, dg), t.O\text{`}nxt$. Dynamic boundary $\overline{\theta}$ is $\mathsf{rd}\ \mathsf{alloc}\text{`}(O, dg), \mathsf{alloc}\text{`}O\text{`}nxt$. It is easy to see $I \vdash (\overline{\theta}, \mathsf{rd}\ \mathsf{alloc})\ \ \mathsf{frm}\ \ I$. Read effect $\mathsf{alloc}\text{`}sub\text{`}O$ of $I$ is subsumed by $\mathsf{alloc}\text{`}O$ because $\mathsf{alloc}\text{`}sub \subseteq \mathsf{alloc}$. We first need

$$\Theta\langle\overline{\theta}\rangle \vdash \{\ W\ \}\ C\ \{\ W'\ \}\ [\overline{\varepsilon}] \tag{8}$$

where Θ is a list containing the specifications in Fig. 14. But using the proof outline above established $\Theta\langle\rangle \vdash \{\ W\ \}\ C\ \{\ W'\ \}\ [\overline{\varepsilon}]$ from which (8) follows by a few applications of DYNBNDINTROM and sequencing. Therefore, first by SOF, we can conjoin I and then by DYNBNDELIM drop $\overline{\theta}$ to obtain

$$(\Theta \circledast I)\langle\rangle \vdash \{\ W \wedge I\ \}\ C\ \{\ W' \wedge I\ \}\ [\overline{\varepsilon}]$$

Now for each $\{Q\}m\{Q'\}[\overline{\eta}]$ in Θ we must show

$$(\Theta \circledast I)\langle\rangle \vdash \{\ Q \wedge I\ \}\ B_m\ \{\ Q' \wedge I\ \}\ [\overline{\eta}]$$

where Q and Q' are the pre- and post-conditions and $\overline{\eta}$ are the effects according to Fig. 14 and B_m is the implementation of m. For example, to verify $Observer$'s body, B_o, we factor I into $J \wedge I(\{u\}, \{\mathsf{self}\})$ and establish $\vdash \{\ Q \wedge J\ \}\ B_o\ \{\ Q' \wedge J\ \}\ [\overline{\varepsilon}_o]$. where J is $SubH(u) \wedge sub \neq \mathsf{null} \Rightarrow \mathsf{self} \in sub\text{`}O$. Now FRAME yields $\vdash \{\ Q \wedge I\ \}\ B_o\ \{\ Q' \wedge I\ \}\ [\overline{\varepsilon}_o]$.

7. Related work

The influence of SL on our work is clear. We follow the direction of several recent works [10, 16] that have exploited ghost state

Method	Pre-condition	Post-condition	Effects
update(n)	$\forall b : Observer \in \{\mathsf{self}\}`O \mid Obs(b, \mathsf{self}, \mathsf{self}.val)$	$\forall b : Observer \in \{\mathsf{self}\}`O \mid Obs(b, \mathsf{self}, n)$	$\mathsf{wr}\ \{\mathsf{self}\}`val, \mathsf{self}.O`cacheVal$
get	$true$	$res = val$	
Observer(u)	$\forall b : Observer \in \{u\}`O \mid Obs(b, u, u.val)$	$\forall b : Observer \in \{u\}`O \mid Obs(b, u, u.val)$ $\wedge \mathsf{self} \in \{u\}`O$	$\mathsf{wr}\ \{u\}`O`nxt, \{u\}`(O, dg)$

Figure 14. Specifications for Subject/Observer example. $Observer$'s effects publishes data group dg that abstracts from obs.

to abstract away low-level reasoning (e.g. maintain reachability information as needed while not saying so explicitly in specs.) The price is that programs need to be instrumented with ghost updates and our specifications are less compact in contrast to [27].

Both our Frame rule and our SOF rule use ordinary conjunction to introduce an invariant, together with side conditions that designate a footprint with respect to which the invariant is separated from the write effect of a command. By contrast, in SL these rules use the separating conjunction $*$ which expresses the existence of such a footprint. Reynolds gave a derivation using the rule of conjunction that shows the SOF is not sound in SL (the "conundrum") without restriction to predicates that are "precise" in the sense of determining a unique footprint [27]. A predicate I is precise iff $(I * \_)$ distributes over $\wedge$, i.e., $I * (Q \wedge R) \Leftrightarrow (I * Q) \wedge (I * R)$. In an unpublished proof of admissibility of SOF in region logic, Naumann proved the case of the conjunction rule using that $(I \wedge \_)$ distributes over $\wedge$. The admissibility proof essentially says that a use of SOF can be replaced by explicitly conjoining the invariant throughout the proof, justified by many uses of ordinary FRAME to introduce not only the invariant but also a chosen footprint.

In earlier work [1] we automatically verified the Observer pattern in Boogie, with the invariant hidden — by postulate. Our current work validates the postulate by way of the SOF. Through Peter Müller (personal communication) we have learnt that our Boogie encodings of regions have been adapted to successfully verify the priority inheritance protocol in real time OS code.

Drossopoulou et al. [13] introduce a general framework to describe verification techniques for invariants based on visible state semantics which requires all invariants to hold on all method call/return boundaries. The framework handles subtyping as well. A general result shows that certain constraints on the framework parameters are sufficient for soundness. A number of ownership type disciplines from the literature are studied as instances of the framework, which promises great improvement in comparing and assessing disciplines. In its present form, the framework does not encompass disciplines like Boogie [4] that rely on state-based encapsulation rather than types or disciplines that deal with design patterns. However, the framework does handle callbacks.

One reaction to the difficulty of hiding invariants is to abandon hiding entirely, in favor of abstraction as advocated by Bierman and Parkinson [6]. They use second order separation logic: method specifications refer to "abstract predicates" that are existentially quantified over the specification. This has been implemented in the jStar prototype [12] based on symbolic execution of SL assertions. Parkinson has reacted to the challenge of invariants over object clusters by advocating explicit but abstract invariants in method specifications [30]. We think more experience with realistic clients is needed to evaluate the practicality of explicitly carrying around a conjunct for the invariant of each abstraction in use.

Smans et al. [34] build a prototype verifier to explore ways to do dynamic framing in the setting of ordinary assertion languages (and also SL), gaining precision through use of location sets and abstraction via pure method calls in assertions. Invariants are addressed via abstraction (model fields and pure methods) also in their recent work [33]. Leino's Dafny language features dynamic frames of the form $G`\mathsf{any}$ as well as pure functions that can compute recursive predicates like our FC and $List$ examples [22].

8. Conclusion

We presented an imperative notion of module interface, which complements static scoping constructs with state dependent expression of the dynamic part of the encapsulation boundary. Each primitive action of the client must respect this boundary —it is not enough that the end to end effect of the client respects it. This entails a small step interpretation of specifications. (It is achieved in SL using a big step interpretation but a stronger separation property.) What is achieved is sound and flexible modular reasoning that encompasses various design patterns. We conclude that Hoare's "mis"match is rather an intricate match between modules and their clients and we show how to get it right.

Apart from the specifications for interface methods, a specifier needs to make explicit the dynamic encapsulation boundary. Then the verifier can fulfill its usual obligations, namely, generate verification conditions (a) for the client by using the interface method specifications and (b) for the bodies of interface methods taking the module invariant into account. But the verifier also needs to verify that each primitive action of the client respects the boundary.

In future work we plan to extend our results to subclassing and inheritance as well as state based representation independence [2]. A particularly exciting challenge is callbacks. Our rendition of the Observer pattern (following Parkinson [30]) has callbacks within the module. It appears that the small-step correctness property enables a SOF rule that supports callbacks back and forth across dynamic encapsulation boundaries.

References

[1] A. Banerjee, M. Barnett, and D. A. Naumann. Boogie meets regions: A verification experience report. In *VSTTE*, pages 177–191, 2008.

[2] A. Banerjee and D. A. Naumann. State based ownership, reentrance, and encapsulation. In *ECOOP*, pages 387–411, 2005.

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

[4] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS*, volume 3362 of *LNCS*, pages 49–69, 2005.

[5] N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of atomic blocks and typestate. In *OOPSLA*, pages 227–244, 2008.

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

[7] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *POPL*, pages 213–223, 2003.

[8] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. Rustan M. Leino, and E. Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, 2005.

[9] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, pages 292–310, 2002.

[10] M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. VCC: Contract-based modular verification of concurrent C. In *ICSE-Companion*, pages 429–430, 2009.

[11] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4:5–32, 2005.

[12] D. Distefano and M. J. Parkinson. jstar: Towards practical verification for java. In *OOPSLA*, pages 213–226, 2008.

[13] S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. A unified framework for verification techniques for object invariants. In *ECOOP*, pages 412–437, 2008.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI*, 2002.

[15] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. *ACM TOPLAS*, 29(6), 2006.

[16] C. Hawblitzel and E. Petrank. Automated verification of practical garbage collectors. In *POPL*, pages 441–453, 2009.

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

[18] I. T. Kassios. Dynamic framing: Support for framing, dependencies and sharing without restriction. In *Formal Methods*, 2006.

[19] N. R. Krishnaswami, J. Aldrich, L. Birkedal, K. Svendsen, and A. Buisse. Design patterns in separation logic. In *TLDI*, 2009.

[20] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007.

[21] G. T. Leavens and P. Müller. Information hiding and visibility in interface specifications. In *ICSE*, pages 385–395, 2007.

[22] K. R. M. Leino. Specification and verification in object-oriented software. Marktoberdorf lecture notes, 2008.

[23] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, pages 491–516, 2004.

[24] K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *PLDI*, pages 246–257, 2002.

[25] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Programming*, 62(3):253–286, 2006.

[26] D. A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state (extended abstract). In *LICS*, pages 313–323, 2004.

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

[28] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, pages 1–19, 2001.

[29] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. *ACM TOPLAS*, 31(3):1–50, 2009.

[30] M. Parkinson. Class invariants: The end of the road? In *IWACO*, 2007.

[31] C. Pierik. Validation techniques for object-oriented proof outlines. Dissertation, Universiteit Utrecht, 2006.

[32] C. Pierik, D. Clarke, and F. S. de Boer. Controlling object allocation using creation guards. In *Formal Methods*, pages 59–74, 2005.

[33] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP*, 2009.

[34] J. Smans, B. Jacobs, F. Piessens, and W. Schulte. An automatic verifier for Java-like programs based on dynamic frames. In *FASE*, pages 261–275, 2008.

[35] K. Zee, V. Kuncak, and M. C. Rinard. Full functional verification of linked data structures. In *PLDI*, pages 349–361, 2008.