Dynamic Boundaries: Information Hiding by Second Order Framing with First Order Assertions

David A. Naumann[1,⋆] and Anindya Banerjee[2,⋆⋆]

[1] Stevens Institute of Technology, Hoboken NJ, USA
[2] IMDEA Software, Madrid, Spain

Abstract. The hiding of internal invariants creates a mismatch between procedure specifications in an interface and proof obligations on the implementations of those procedures. The mismatch is sound if the invariants depend only on encapsulated state, but encapsulation is problematic in contemporary software due to the many uses of shared mutable objects. The mismatch is formalized here in a proof rule that achieves flexibility via explicit restrictions on client effects, expressed using ghost state and ordinary first order assertions.

1 Introduction

From the simplest collection class to the most complex application framework, software modules provide useful abstractions by hiding the complexity of efficient implementations. Many abstractions and most representations involve state, so the information to be hidden includes invariants on internal data structures. Hoare described the hiding of invariants as a mismatch between the procedure specifications in a module interface, used for reasoning about client code, and the specifications with respect to which implementations of those procedures are verified. The latter assume the invariant and are obliged to maintain it [17]. The justification is simple: A hidden invariant should depend only on encapsulated state, in which case it is necessarily maintained by client code. Hoare's formalization was set in a high level object-oriented language (Simula 67), which is remarkable because for such languages the encapsulation problem has far too many recent published solutions to be considered definitively solved.

For reasoning about shared, dynamically allocated objects, the last decade has seen major advances, especially the emergence of Separation Logic, which helped reduce what O'Hearn et al. aptly called a "mismatch between the simple intuitions about the way pointer operations work and the complexity of their axiomatic treatments" [29, Sect. 1]. For encapsulation, there remains a gap between the simple idea of hiding an invariant and the profusion of complex encapsulation techniques and methodologies. The profusion is a result of tensions between

- The need to prevent violations of encapsulation due to misuse of shared references.
- The need to encompass useful designs including overlapping and non-regular data structures, callbacks, and the deliberate use of shared references that cross encapsulation boundaries. Illustrative examples are the topic of Sect. 2.

⋆ Partially supported by US NSF awards CNS-0627338, CRI-0708330, CCF-0915611.
⋆⋆ Partially supported by US NSF awards CNS-0627748.

A.D. Gordon (Ed.): ESOP 2010, LNCS 6012, pp. 2–22, 2010.

- The need for effective, modular reasoning on both sides of an interface: for clients and for the module implementation.
- The hope to achieve high automation through mature techniques including types and static analyses as well as theorem proving.
- The need to encompass language features such as parametric polymorphism and code pointers for which semantics is difficult.

This paper seeks to reconcile all but the last of these and to bridge the gap using a simple but flexible idea that complements scope-based encapsulation. The idea is to include in an interface specification an explicit description of the key intuition, the internal state or "heap footprint" on which an invariant rests. This set of locations, called the *dynamic boundary*, is designated by an expression that may depend on ordinary and ghost state.

We formalize the idea using first order assertions in a Hoare logic for object based programs called Region Logic (Sect. 3); it is adapted from a previous paper in which we briefly sketched the idea and approach [2]. Our approach is based on correctness judgements with hypotheses, to account for linking of client code to the modules used, and a frame rule to capture hiding. These two ingredients data back to the 1970's (e.g., [15]) but we build directly on their novel combination in the second order frame rule of separation logic [30]. Our version of the rule is the topic of Sect. 4.

Owing to the explicit expression of footprints, region logic for first order programs and specifications has an elementary semantics and is amenable to automation with SMT solvers [21]. One price to pay is verbosity, but the foundation explored in this paper supports syntactic sugars for common cases while avoiding the need to hard-code those cases. Another price is an additional proof obligation on clients, to respect the dynamic boundaries of modules used. In many cases this can be discharged by type checking. But our main goal is to account for hiding in a way that is sufficiently flexible to encompass ad hoc disciplines for encapsulation; even more, to let the formalization of such a discipline be a matter of program annotation, with its adequacy checked by a verification tool, rather than being fodder for research papers.

The main result is soundness of our second order frame and boundary introduction rules, whose range of applicability is indicated by application, in Sect. 5, to the examples in Sect. 2. For lack of space, technical details are only skimmed, as is related work (Sect. 6). An appendix with the soundness proof can be found online.

2 The Challenge of Hiding Invariants on Shared Mutable Objects

2.1 A Collection Implemented by a List

We begin with a textbook example of encapsulation and information hiding, the toy program in Fig. 1.[1] Annotations include method postconditions that refer to a global variable, *pool*, marked as ghost state. Ghost variables and fields are auxiliary state used

[1] The programming notation is similar to sequential Java. A value of a class type like $Node$ is either null or a reference to an allocated object with the fields declared in its class. Methods have an implicit parameter, self, which may be elided in field updates; e.g., the assignment $lst := \mathsf{null}$ in the body of the Set constructor is short for $\mathsf{self}.lst := \mathsf{null}$.

```
ghost pool : rgn;
class Set { lst : Node; ghost rep : rgn;
  model elements = elts(lst)
        where elts(n : Node) = (if n = null then ∅ else {n.val} ∪ elts(n.nxt))
  Set() ensures elements = ∅ ∧ pool = old(pool) ∪ {self}
  { lst := null; rep := ∅; pool := pool ∪ {self}; }
  add(i : int) ensures elements = old(elements) ∪ {i}
  { if ¬contains(i) then var n : Node := new Node; n.val := i; n.nxt := lst; lst := n;
                         n.own := self; rep := rep ∪ {n}; endif }
  contains(i : int) : boolean ensures result = (i ∈ elements) { "linear search for i" }
  remove(i : int) ensures elements = old(elements) − {i} { "remove first i, if any" } }
class Node { val : int; nxt : Node; ghost own : Object; }   //library code, not part of SET
```

Fig. 1. Module SET, together with class $Node$. Variable result is the returned result.

in reasoning, but not mentioned in branch conditions or expressions assigned to ordinary state. Assignments to ghost state can be removed from a program without altering its observable behavior, so ghosts support reasoning about that behavior. A *region* is a set of object references (which may include the improper reference, null). Type rgn, which denotes regions, is used only for ghost state.

The specifications are expressed in terms of an integer set, $elements$. Abstraction of this sort is commonplace and plays a role in Hoare's paper [17], but it is included here only to flesh out the example. Our concern is with other aspects so we content ourselves with a recursive definition (of $elts$) that may seem naïve in not addressing the possibility of cyclic references.

Suppose the implementation of $remove$ only removes the first occurrence of i, if any. That is, it relies on the invariant that no integer value is duplicated in the singly linked list rooted at lst. To cater for effective automated verification, especially using SMT solvers, we want to avoid using reachability or other recursively defined notions in the invariant. The ghost field rep is intended to refer to the set of nodes reachable from field lst via nxt. The invariant is expressed using elementary set theoretic notions including the image of a region under a field. The expression $s.rep`nxt$ denotes the region consisting of nxt values of objects in region $s.rep$. It is used in this definition:[2]

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

The first conjunct says there are no duplicates among elements of $s.rep$. The next says that $s.rep$ contains the first node, if any (or else null). The inclusion $s.rep`nxt \subseteq s.rep$

[2] The range condition "$n \in s.rep$" is false in case s is null, because $n \in s.rep$ is shorthand for $n \in \{s\}`rep$ and $\{\mathsf{null}\}`rep$ is empty. Our assertion logic is 2-valued and avoids undefined expressions. We do not use sets of regions. The image operator flattens, for region fields: For any region expression G, the image region $G`rep$ is the union of $rep$ images whereas $G`nxt$ is the set of nxt images, because rep has type rgn and nxt has class type.

says that $s.rep$ is nxt-closed; this is equivalent to $\forall o \mid o \in s.rep \Rightarrow o.nxt \in s.rep$.[3] One can show by induction that these conditions imply there are no duplicates; so the invariant says what we want, though not itself using induction. However, $s.rep$ could be nxt-closed even if $s.rep$ contained extraneous objects, in particular nodes reached from other instances of Set. This is prevented by the inclusion $s.rep`own \subseteq \{s\}$; or rather, by requiring the inclusion for every instance of Set. So we adopt an invariant to be associated with module SET:

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

As used here, variable $pool$ is superfluous, but we are hinting at examples where an invariant is not maintained for all instances of a class but, e.g., only those created by a factory method. The need for $\mathsf{null} \notin pool$ is minor and discussed later. A bigger concern is the global nature of I_{set}, which is addressed in Sect. 3.2.

Consider this client code, acting on boolean variable b, under precondition $true$:

$$\begin{array}{l} \mathsf{var}\ s : Set := \mathsf{new}\ Set;\ \mathsf{var}\ n : Node := \mathsf{new}\ Node; \\ s.add(1);\ s.add(2);\ n.val := 1;\ s.remove(1);\ b := s.contains(1); \end{array} \tag{1}$$

The implementation of $remove$ relies on the invariant $SetI(s)$, but this is not included as a precondition in Fig. 1 and the client is thus not responsible to establish it before the invocation of $remove$. As articulated by Hoare [17], the justification is that the invariant appears as both pre- and post-condition for verification of the methods add, $remove$, $contains$, and should be established by the Set constructor. And the invariant should depend only on state that is encapsulated. So it is not falsified by the initialization of n and still holds following $s.add(2)$; again by encapsulation it is not falsified by the update $n.val := 1$ so it holds as assumed by $s.remove$.

We call this *Hoare's mismatch*: the specifications used in reasoning about invocations in client code, i.e. code outside the encapsulation boundary, differ from those used to verify the implementations of the invoked methods. By contrast, ordinary procedure call rules in program logic use the same specification at the call site and to verify the procedure implementation. Automated, modular verifiers are often based on an intermediate language using assert and assume statements: At a call site the method precondition is asserted and this same precondition is assumed for the method's implementation; so the assumption is justified by the semantics of assert and assume. Hoare's mismatch asserts the public precondition but assumes an added conjunct, the invariant.

The mismatch is unsound if encapsulation is faulty, which can easily happen due to shared references, e.g., if in place of $n.val := 1$ the client code had $s.lst.val := 1$. Lexical scope and typing can provide encapsulation, e.g., field lst should have module scope. (We gloss over scope in the examples.) However, scope does not prevent that references can be leaked to clients, e.g., via a global variable of type Object. Moreover, code within the module, acting on one instance of Set, could violate the invariant of another instance. Besides scope and typing, a popular technique to deal with encapsulation in the presence of pointers is "ownership" (e.g., [9,11]). Ownership systems restrict the form of invariants and the use of references, to support modular reasoning

[3] Quantified variables range over non-null, allocated references.

ghost freed : rgn;
var $\mathit{flist} : \mathit{Node}$; count : int;

$\mathit{alloc}() : \mathit{Node}$
  ensures result $\neq$ null $\wedge\ \mathit{freed} = \mathsf{old}(\mathit{freed}) - \{\mathsf{result}\} \wedge (\mathsf{result} \in \mathsf{old}(\mathit{freed}) \vee \mathsf{fresh}(\mathsf{result}))$
{ if $\mathit{count} = 0$ then result := new Node;
  else result := flist; $\mathit{flist} := \mathit{flist}.\mathit{nxt}$; $\mathit{count} := \mathit{count} - 1$; $\mathit{freed} := \mathit{freed} - \{\mathsf{result}\}$; endif}

$\mathit{free}(n : \mathit{Node})$ requires $n \neq$ null $\wedge\ n \notin \mathit{freed}$ ensures $\mathit{freed} = \mathsf{old}(\mathit{freed}) \cup \{n\}$
{ $n.\mathit{nxt} := \mathit{flist}$; $\mathit{flist} := n$; $\mathit{count} := \mathit{count} - 1$; $\mathit{freed} := \mathit{freed} \cup \{n\}$; }

Fig. 2. Module MM

at the granularity of a single instance and its representation. Ownership works well for $SetI$ and indeed for invariants in many programs.

2.2 A Toy Memory Manager

It is difficult to find a single notion of ownership that is sufficiently flexible yet sound for invariant hiding. Fig. 2 presents a module that is static in the sense that there is a single memory manager, not a class of them. Instances of class Node (from Fig. 1) are treated as a resource. The instances currently "owned" by the module are tracked using variable freed. The hidden invariant, I_{mm}, is defined to be $FC(\mathit{flist}, \mathit{freed}, \mathit{count})$ where $FC(f : \mathit{Node}, r : \mathsf{rgn}, c : \mathsf{int})$ is defined, by induction on the size of r, as

$$(f = \mathsf{null} \Rightarrow r = \varnothing \wedge c = 0) \wedge (f \neq \mathsf{null} \Rightarrow f \in r \wedge c > 0 \wedge FC(f.\mathit{nxt}, r - \{f\}, c - 1))$$

The invariant says freed is the nodes reached from flist and count is the size. The implementation of alloc relies on accuracy of count. It relies directly on $\mathit{count} \neq 0 \Rightarrow \mathit{flist} \neq \mathsf{null}$, as otherwise the dereference $\mathit{flist}.\mathit{nxt}$ could fault, but for this to hold on subsequent calls the stronger condition I_{mm} 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.

var $x, y : \mathit{Node}$; $x :=$ new Node; $y := \mathit{alloc}()$; $\mathit{free}(x)$; $\mathit{free}(y)$;
while $y \neq$ null do $y.\mathit{val} := 7$; $y := y.\mathit{nxt}$; od

The loop updates val fields of freed objects, but it does not write the nxt fields, on which the invariant depends; the client never causes a fault. Suppose we replaced the loop by the assignment $y.\mathit{nxt} :=$ null. This falsifies the invariant I_{mm}, if initially count is sufficiently high, and then subsequent invocations of alloc break.

The strange client is rejected by most ownership systems. But there is an encapsulation boundary here: clients must not write the nxt field of objects in freed (nor write variables flist and count). The strange client respects this boundary.

Sharing of references across encapsulation boundaries is common in system code, at the C level of abstraction. But it also occurs with notional resources such as database connections in programs at the level of abstraction we consider here, where references are abstract values susceptible only to equality test and field dereference.

```
class Subject { obs: Observer; val: int; ghost O: rgn;
  Subject() { obs := null; val := 0; O := ∅; }
  update(n: int) ensures ∀b: Observer ∈ O | Obs(b, self, n)
  { val := n; var b: Observer := obs; while b ≠ null do b.notify(); b := b.nxto; od }
  get(): int { result := val; }
  register(b: Observer) { b.nxto := obs; obs := b; O := O ∪ {b}; b.notify(); } }
class Observer { sub: Subject; cache: int; nxto: Observer;
  Observer(s: Subject) requires ∀b: Observer ∈ s.O | Obs(b, s, s.val)
                       ensures self ∈ s.O ∧ ∀b: Observer ∈ s.O | Obs(b, s, s.val)
  { sub := s; s.register(self); }
  notify() { cache := sub.get(); } }
```

Fig. 3. Module OB. We define $Obs(b, s, v)$ as $b.sub = s \wedge b.cache = v$.

2.3 Observer Pattern: Cluster Invariants

Fig. 3 is a simple version of the Observer design pattern in which an observer only tracks a single subject. Parkinson [31] used the example to argue against instance-oriented notions of invariant. We address that issue using a single invariant predicate that in effect quantifies over clusters of client-visible objects. Classes $Subject$ and $Observer$ are together in a module, in which methods $register$ and $notify$ should have module scope. The implementation maintains the elements of O in the $nxto$-linked list threaded through the observers themselves, and it relies on the hidden invariant

$$I_{ob}: \ (\forall s: Subject \mid List(s.obs, s.O)) \wedge (\forall o: Observer \mid o.sub \neq \mathsf{null} \Rightarrow o \in o.sub.O)$$

where $List(o, r)$ says the list beginning at o lies in region r (compare FC in Sect. 2.2). The second conjunct of I_{ob} says that any observer tracking a subject lies in that subject's O region. As with I_{set}, the instantiations of I_{ob} are local in that they depend on nearby objects, but here a subject and its observers form a cooperating cluster of objects not in an ownership relation. Clients may rely on separation between clusters. As an example, consider a state in which there are two subjects s, t with $s.val = 0$ and $t.val = 5$. Consider this client: $o := \mathsf{new}\ Observer(s); p := \mathsf{new}\ Observer(t); s.update(2)$. Owing to separation, $t.val = 5$ holds in the final state.

2.4 Overlapping Data Structures and Nested Modules

One feature of the preceding example is that there is an overlapping data structure because a list structure is threaded through observer objects that are client visible. We now consider another example which further illustrates overlapping data structures and also hiding in the presence of nested modules. The module in Fig. 4 consists of a class, $ObsSet$, that extends $Observer$. Instances of $ObsSet$ are in two overlapping data structures. First, these objects are arranged in a cyclic doubly-linked list, traversed using $next$ and $prev$ pointers, whose elements may be observing the same or different subjects. Second, each $ObsSet$ is in the $nxto$-linked list of observers of its subject.

```
class ObsSet extends Observer { next: ObsSet; prev: ObsSet;
  ObsSet(s: Subject, os: ObsSet)
    requires ∀b: Observer ∈ s.O | Obs(b, s, s.val)
    ensures self ∈ s.O ∧ ∀b: Observer ∈ s.O | Obs(b, s, s.val)
  { super(s);
    if os = null then prev := self; next := self;
    else next := os; prev := os.prev; os.prev.next := self; os.prev := self; endif } }
```

Fig. 4. Module OS

The constructor of $ObsSet$ first calls the superclass constructor, $Observer$, with subject s. This call adds the newly allocated object to the front of the list of observers of s. The newly allocated object is then added to the cyclic doubly-linked list by manipulating $next$ and $prev$ pointers.

Module OS is defined in the context of module OB, because $ObsSet$ is a subclass of $Observer$. The verification of the implementation of $ObsSet$ will require its module invariant, but not I_{ob}. The invariant I_{os} expresses a simple property of cyclic doubly-linked lists: $os.prev.next = os \wedge os.next.prev = os$ for all allocated os of type $ObsSet$. Despite the overlapping structure, there is no interference between the code and invariants of modules OB and OS because different locations are involved.

Interesting variations on the example include observers that track multiple subjects, and observers that are also in the role of subject (cf. [19]). Of particular interest are callbacks between modules (as opposed to the $notify/get$ callback within module OB), which are within reach of our approach but not formalized in this paper.

3 Region Logic Background: Effects and First Order Framing

3.1 Preliminaries: Programming Language, States, Assertions

Our formal results are given for an idealized object-based language with syntax sketched in Fig. 5. Programs are considered in the context of a fixed collection of class declarations, of the form class K { $\overline{f}: \overline{T}$ }, where field types may make mutually recursive reference to other classes. We write Fields(K) for $\overline{f}: \overline{T}$ and for simplicity let names in the list $\overline{f}$ have global scope. Ordinary expressions do not depend on the heap: $y.f$ is not an expression but rather part of the command $x := y.f$ for reading a field, as in separation logic. Instead of methods associated with classes, we formalize simple

T	::= int $\mid K \mid$ rgn where K is in $DeclaredClassName$	data types
E	::= $x \mid c \mid$ null $\mid E \oplus E$ where c is in $\mathbb{Z}$, $\oplus$ in $\{=,+,>,\ldots\}$	ordinary expressions
G	::= $x \mid \{E\} \mid \varnothing \mid G$`f` $\mid G \otimes G$ where $\otimes$ is in $\{\cup,\cap,-\}$	region expressions
F	::= $E \mid G$	expressions
C	::= $m(x) \mid x := F \mid x :=$ new $K \mid x := x.f \mid x.f := F$	primitive commands
	$\mid$ let $m(x: T)$ be C in $C \mid$ var $x: T$ in C end $\mid C\,;C \mid \ldots$	binding, control struct.

Fig. 5. Program syntax, where $x \in VarName$, $f \in FieldName$, $m \in ProcName$

procedures without an implicit self parameter. The typing judgement for commands is written as $\Pi \vdash^{\Gamma} C$ where Γ is a variable context and Π is a list of procedure signatures of the form $m(x\colon T)$. The form "let $m(x\colon T)$ be B in C" is typable in context Π and Γ if $\Pi, m\colon(x\colon T) \vdash^{\Gamma, x\colon T} B$ and $\Pi, m\colon(x\colon T) \vdash^{\Gamma} C$. The generalization to multiple parameters and mutually recursive procedures is straightforward and left to the reader. Typing rules enforce that type int is separated from reference types: there is no pointer arithmetic, but pointers can be tested for equality. The variable alloc, being of type rgn, cannot occur in non-ghost code.

The semantics is based on conventional program states. We assume given a set Ref of reference values including a distinguished value, null. A Γ-*state* has a global heap and a store. The store assigns values to the variables in Γ and to the variable alloc: rgn which is special in that its updates are built in to the semantics of the language: newly allocated references are added and there are no other updates, so it holds the set of allocated references. The heap maps each allocated reference to its type (which is immutable) and field values. The values of a class type K are null and allocated references of type K. We assume the usual operations are available for a state σ. For example, $\sigma(x)$ is the value of variable x, $\sigma(F)$ is the value of expression F, $\mathrm{Type}(o, \sigma)$ is the type of an allocated reference o, $\mathrm{Update}(\sigma, o.f, v)$ overrides σ to map field f of o to v (for $o \in \sigma(\mathsf{alloc})$), $\mathrm{Extend}(\sigma, x, v)$ extends σ to map x to value v (for $x \notin \mathrm{Dom}(\sigma)$). Heaps have no dangling references; we do not model garbage collection or deallocation.

In a given state the region expression $G\text{`}f$ (read "$G$'s image under $f$") denotes one of two things. If $f$ has class type then $G\text{`}f$ is the set of values $o.f$ where o ranges over (non-null) elements of G that have field f. If f has region type, like *rep* in our example, then $G\text{`}f$ is the union of the values of f.

Assertions are interpreted with respect to a single state, e.g., the semantics of the primitive $x.f = E$ that reads a field is defined: $\sigma \models x.f = E$ iff $\sigma(x) \neq \mathsf{null}$ and $\sigma(x.f) = \sigma(E)$. The operator "old" used in specifications can be desugared using auxiliaries quantified over specifications (omitted from this version of the paper). We do not use quantified variables of type rgn. Quantified variables of class type range over non-null, currently allocated references: $\sigma \models^{\Gamma} (\forall x\colon K \mid P)$ iff $\mathrm{Extend}(\sigma, x, o) \models^{\Gamma, x\colon K} P$ for all $o \in \sigma(\mathsf{alloc})$ such that $\mathrm{Type}(o, \sigma) = K$. In a richer language with subclassing, this would be $\leq K$.

3.2 Effect Specifications and the Framing of Commands and Formulas

Let us augment the specifications in Fig. 1 with the effect specifications in Fig. 6. Effects are given by the grammar $\varepsilon ::= \mathsf{wr}\, x \mid \mathsf{rd}\, x \mid \mathsf{wr}\, G\text{`}f \mid \mathsf{rd}\, G\text{`}f \mid \mathsf{fr}\, G$. We omit tags wr and rd in lists of effects of the same kind. In this paper, read effects are used for formulas and write effects as frame conditions for commands and methods; commands are allowed to read anything. Freshness effect fr G is used for commands; it says that the value of G in

$Set()$	wr *pool*
$add(i\colon int)$	wr alloc, self.any, self.*rep*`any
$remove(i\colon int)$	wr self.any, self.*rep*`any

Fig. 6. Effect specifications for methods in Fig. 1. For *contains* the specification has no effects.

the final state contains only (but not necessarily all) references that were not allocated in the initial state.

The effect specification for the constructor method, $Set()$, says variable $pool$ may be updated. For add, the effect $\mathsf{wr}\,\mathsf{alloc}$ means that new objects may be allocated. The effect $\mathsf{wr}\,\mathsf{self}.\mathsf{any}$ says that any fields of self may be written. The effect $\mathsf{wr}\,\mathsf{self}.rep\text{`}\mathsf{any}$ says that any field of any object in $\mathsf{self}.rep$ may be written; in fact none are written in our implementation, but this caters for other implementations. The effect $\mathsf{wr}\,\mathsf{self}.rep\text{`}\mathsf{any}$ is state dependent, because rep is a mutable field.

In general, let G be a region expression and f be a field name. The effect $\mathsf{wr}\,G\text{`}f$ refers to l-values: the locations of the f fields of objects in G —where G is interpreted in the initial state. A location is merely a reference paired with a field name.

An effect of the form $\mathsf{wr}\,x.f$ abbreviates $\mathsf{wr}\,\{x\}\text{`}f$. In case x is null, this is well defined and designates the empty set of locations. We also allow f to be a data group [26], e.g., the built-in data group "any" that stands for all fields of an object.

We say effect list $\overline{\varepsilon}$ *allows transition from* σ *to* σ', written $\sigma \rightsquigarrow \sigma' \models \overline{\varepsilon}$, if and only if σ' succeeds[4] σ and

(a) for every y in $\mathrm{Dom}(\Gamma) \cup \{\mathsf{alloc}\}$, either $\sigma(y) = \sigma'(y)$ or $\mathsf{wr}\,y$ is in $\overline{\varepsilon}$
(b) for every o in $\sigma(\mathsf{alloc})$ and every f in $\mathrm{Fields}(\mathrm{Type}(o,\sigma))$, either $\sigma(o.f) = \sigma'(o.f)$ or there is G such that $\mathsf{wr}\,G\text{`}f$ is in $\overline{\varepsilon}$ and o is in $\sigma(G)$
(c) for each $\mathsf{fr}\,G$ in $\overline{\varepsilon}$, we have $\sigma'(G) \subseteq \sigma'(\mathsf{alloc}) - \sigma(\mathsf{alloc})$.

Formulas are framed by read effects. We aim to make explicit the footprint of I_{set}, which will serve as a dynamic boundary expressing the state-dependent aspect of the encapsulation that will allow I_{set} to be hidden from clients. First we frame the object invariant $SetI(s)$, which will be used for "local reasoning" [29] at the granularity of a single instance of Set. We choose to frame[5] it by

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

A read effect designates l-values. Here, $\overline{\delta}_0$ allows to read variable s, fields rep and lst of the object currently referenced by s if any, and the fields nxt, val, and own of any objects in the current value of $s.rep$.

We use a judgement for framing of formulas, e.g., $true \vdash \overline{\delta}_0\ \mathsf{frames}\ SetI(s)$ says that if two states agree on the locations designated by $\overline{\delta}_0$ then they agree on the value of $SetI(s)$. The judgement involves a formula, here $true$, because framing by state-dependent effects may hold only under some conditions on that state. For example we have $s \in pool \vdash \mathsf{rd}\,pool\text{`}(rep, lst)\ \mathsf{frames}\ s.lst \in s.rep$.

The semantics of judgement $P \vdash \overline{\delta}\ \mathsf{frames}\ P'$ is specified by the following: If $\sigma \models P$ and $\mathrm{Agree}(\sigma, \sigma', \overline{\delta})$ then $\sigma \models P'$ implies $\sigma' \models P'$. Here $\mathrm{Agree}(\sigma, \sigma', \overline{\delta})$ is defined to mean: σ' succeeds σ, $\sigma(x) = \sigma'(x)$ for all $\mathsf{rd}\,x$ in $\overline{\delta}$, and $\sigma(o.f) = \sigma'(o.f)$ for all $\mathsf{rd}\,G\text{`}f$ in $\overline{\delta}$ and all $o \in \sigma(G)$ with $f \in \mathrm{Fields}(o, \sigma)$.

There are two ways to establish a framing judgement. One is to directly check the semantics, which is straightforward but incomplete using an SMT prover, provided the

[4] σ' *succeeds* σ iff $\sigma(\mathsf{alloc}) \subseteq \sigma'(\mathsf{alloc})$ and $\mathrm{Type}(o, \sigma) = \mathrm{Type}(o, \sigma')$ for all $o \in \sigma(\mathsf{alloc})$.

[5] The term "frame" traditionally refers to that which does not change, but frame conditions specify what may change. To avoid confusion we refrain from using "frame" as a noun.

heap model admits quantification over field names (to express agreement). The other way is to use inference rules for the judgement [2]. These include syntax-directed rules together with first-order provability and subsumption. As an example, the rule for $P \vdash \overline{\eta}\ \mathsf{frames}\ (\forall x : K \mid x \in G \Rightarrow P')$ has antecedent of the form $P \wedge x \in G \vdash \overline{\eta}'\ \mathsf{frames}\ P'$ and requires $\overline{\eta}$ to subsume the footprint of G. Our rules are proved to yield a stronger property than the specification: $\sigma \models P'$ iff $\sigma' \models P'$ when $\sigma \models P$ and $\mathrm{Agree}(\sigma, \sigma', \overline{\eta})$.

For I_{set}, we can use the specific judgements above to derive $true \vdash \overline{\delta}_{set}\ \mathsf{frames}\ I_{set}$, where $\overline{\delta}_{set}$ is $\mathsf{rd}\ pool, pool\text{`}(rep, lst), pool\text{`}rep\text{`}(nxt, val, own)$. This is subsumed by

$\overline{\theta}_{set}$: $\mathsf{rd}\ pool, pool\text{`}\mathsf{any}, pool\text{`}rep\text{`}\mathsf{any}$

A frame rule. To verify the implementations in Fig. 1 we would like to reason in terms of a single instance of Set. Let B_{add} be the body of method add. By ordinary means we can verify that B_{add} satisfies the frame conditions $\mathsf{wr}\,\mathsf{alloc}, \mathsf{self.any}$ and thus those for add in Fig. 6. Moreover we can verify the following Hoare triple:

$$\{SetI(\mathsf{self})\}\ B_{add}\ \{SetI(\mathsf{self}) \wedge elements = \mathsf{old}(elements) \cup \{i\}\} \tag{2}$$

From this local property we aim to derive that B_{add} preserves the global invariant I_{set}. It is for this reason that $SetI(s)$ includes ownership conditions. These yield a confinement property: $I_{set} \Rightarrow (\forall s, t : Set \in pool \mid s = t \vee s.rep \,\#\, t.rep)$, because if $n \neq \mathsf{null}$, and n is in $s.rep \cap t.rep$ then $n.own = s$ and $n.own = t$. Here # denotes disjointness of sets; more precisely, $G \,\#\, G'$ means $G \cap G' \subseteq \{\mathsf{null}\}$. Now I_{set} is logically equivalent to $SetI(\mathsf{self}) \wedge Iexcept$, with $\overline{\delta}_x$ framing $Iexcept$, defined as

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

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

We aim to conjoin $Iexcept$ to the pre and post conditions of (2). To make this precise we use an operator $\star$, called the *separator*. 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, describing states in which writes allowed by $\overline{\varepsilon}$ cannot affect the value of a formula with footprint $\overline{\delta}$. The formula $\overline{\delta} \star \overline{\varepsilon}$ can be defined by induction on the syntax of effects [2]. Its meaning is specified by this property: If $\sigma \rightsquigarrow \sigma' \models \overline{\varepsilon}$ and $\sigma \models \overline{\delta} \star \overline{\varepsilon}$ then $\mathrm{Agree}(\sigma, \sigma', \overline{\delta})$.

It happens that $\overline{\delta}_x \star (\mathsf{wr}\,\mathsf{self.any}, \mathsf{wr}\,\mathsf{alloc})$ is $true$. So, to complete the proof of $\{I_{set}\} B_{add} \{elements = \mathsf{old}(elements) \cup \{i\} \wedge I_{set}\}$ we can take Q to be $Iexcept$ and $\overline{\delta}$ to be $\overline{\delta}_x$ in this rule which uses notations explained in Sect. 3.3:

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

Similar reasoning verifies the implementation of $remove$. Note that its effects include $\mathsf{wr}\,\mathsf{self}.rep\text{`}\mathsf{any}$. Moreover $\overline{\delta}_x \star \mathsf{wr}\,\mathsf{self}.rep\text{`}\mathsf{any}$ yields nontrivial disjointnesses: $\mathsf{self}.rep \,\#\, (pool - \{\mathsf{self}\}) \wedge \mathsf{self}.rep \,\#\, (pool - \{\mathsf{self}\})\text{`}rep$. The first conjunct holds because elements of $\mathsf{self}.rep$ have type $Node$ and those of $pool - \{\mathsf{self}\}$ have type Set (details left to reader). The second conjunct is a consequence of the ownership confinement property mentioned earlier, which follows from I_{set}. For verifying $remove$, the precondition P in FRAME will be $true \wedge I_{set}$ because $true$ is the precondition of $remove$ in Fig. 1.

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

$$\frac{\mu(m) = \lambda x\colon T.B \qquad x' \notin \mathrm{Dom}(\sigma) \qquad x' \notin \mathit{params}(\Delta) \qquad B' = B^x_{x'}}{\langle m(z),\ \sigma,\ \mu\rangle \longmapsto \langle (B';\mathsf{end}(x')),\ \mathrm{Extend}(\sigma, x', \sigma(z)),\ \mu\rangle}$$

$$\frac{\begin{array}{c}\Delta \text{ contains } \{P\}m(x\colon T)\{P'\}[\overline{\varepsilon}] \\ \sigma \rightsquigarrow \sigma' \models \overline{\varepsilon} \qquad \mathrm{Extend}(\sigma, x, \sigma(z)) \models P \qquad \mathrm{Extend}(\sigma', x, \sigma(z)) \models P'\end{array}}{\langle m(z),\ \sigma,\ \mu\rangle \longmapsto \langle \mathsf{skip},\ \sigma',\ \mu\rangle}$$

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

Fig. 7. The transition relation $\overset{\Delta}{\longmapsto}$. Here Δ is the same throughout and omitted.

3.3 Correctness Judgements and Program Semantics

A *procedure context*, Δ, is a comma-separated list of specifications, each of the form $\{Q\}m(x\colon T)\{Q'\}[\overline{\varepsilon}]$. For the specification to be well formed in a variable context Γ, all of $Q, Q', \overline{\varepsilon}$ should be well formed in $\Gamma, x\colon T$. Moreover the frame condition $\overline{\varepsilon}$ must not contain $\mathsf{wr}\, x$, so the use of x in Q' and $\overline{\varepsilon}$ refers to its initial value. A correctness judgement takes roughly the form $\Delta \vdash^{\Gamma} \{P\}\ C\ \{P'\}\ [\overline{\varepsilon}]$ and is well formed if $\Delta, P, P', \overline{\varepsilon}$ are well formed in Γ and $\mathit{signatures}(\Delta) \vdash^{\Gamma} C$. In Sect. 4 we partition Δ into modules (see Def. 1). A correctness judgement is intended to mean that from any initial state that satisfies P, C does not fault (due to null dereference) and if it terminates then the final state satisfies P'. Moreover, any transition from initial state to final is allowed by $\overline{\varepsilon}$.

The hypothesis Δ is taken into account as well. One semantics would quantify over all implementations of Δ. Instead, we use a mixed-step semantics in which a call $m(z)$ for m in Δ takes a single step to an arbitrary outcome allowed by the specification of m.[6] A configuration has the form $\langle C, \sigma, \mu\rangle$ where C is a command, σ is a state, and the *procedure environment* μ is a partial function from procedure names to parameterized commands of the form $(\lambda x\colon T.C)$. By assuming that in a well formed program no procedure names are shadowed, we can use this simple representation, together with a special command $\mathsf{end}(m)$ to mark the end of the scope of a let-bound procedure m. Renaming is used for a parameter or local variable x, together with end marker $\mathsf{end}(x)$. The transition relation $\overset{\Delta}{\longmapsto}$ is defined in Fig. 7. The procedures in Δ are to be distinct from those in the procedure environment. A terminating computation ends in a configuration of the form $\langle \mathsf{skip}, \sigma, \mu\rangle$, or else "fault" which results from null dereference. The cases omitted from Fig. 7 are quite standard. We note only that the semantics of $\mathsf{new}\ K$, which updates alloc, is parameterized on a function which, given a state, returns a non-empty set of fresh references. Thus our results encompass deterministic allocators as well as the maximally nondeterministic one on which some separation logics rely.

[6] Such semantics is popular in work on program refinement; see also O'Hearn et al [30].

4 Dynamic Boundaries and Second Order Framing

Rule FRAME is useful for reasoning about a predicate that a command is explicitly responsible for preserving, like I_{except} and B_{add} in Sect. 3.2. For the client (1), we want I_{set} to be preserved; semantically, the rationale amounts to framing, but rule FRAME is not helpful because our goal is to hide I_{set} from clients. A client command in a context Δ is second order in that the behavior of the command is a function of the procedures provided by Δ, as is evident in the transition semantics (Fig. 7). Second order framing is about a rely-guarantee relationship: the module relies on good behavior by the client, such that the client unwittingly preserves the hidden invariant, and in return the module guarantees the behavior specified in Δ.

Our rely condition is list of read effects, called the *dynamic boundary*, that must be respected by the client in the sense that it does not write the locations designated by those effects. A dynamic boundary $\overline{\delta}$ is associated with a list Δ of procedure specifications using notation $\Delta\langle\overline{\delta}\rangle$. The general form for correctness judgement would have a sequence $\Delta_1\langle\overline{\delta}_1\rangle; \ldots; \Delta_n\langle\overline{\delta}_n\rangle$ of hypotheses, for n modules, $n \geq 0$. In an attempt to improve readability, we will state the rules for the case of just two modules, typically using name Θ for Δ_n. So a correctness judgement has the form

$$\Delta\langle\overline{\delta}\rangle; \Theta\langle\overline{\theta}\rangle \vdash^{\Gamma} \{\, P \,\}\ C\ \{\, P' \,\}\,[\overline{\varepsilon}] \tag{3}$$

where $\overline{\delta}$ and $\overline{\theta}$ are lists of read effects that are well formed in Γ. The order of modules is significant: the implementation of Θ may use procedures from Δ and is obliged to respect dynamic boundary $\overline{\delta}$. For a dynamic boundary to be useful it should frame the invariant to be hidden, e.g., $\overline{\theta}_{set}$ frames I_{set}. That proof obligation is on the module.

The following derived rule embodies Hoare's mismatch in the case where module Θ is a single procedure specification $\{Q\}m(x\colon T)\{Q'\}[\overline{\eta}]$.

$$\textsc{Mismatch}\ \frac{\begin{array}{c}\Delta\langle\overline{\delta}\rangle; \Theta\langle\overline{\theta}\rangle \vdash \{\, P \,\}\ C\ \{\, P' \,\}\,[\overline{\varepsilon}] \qquad I \vdash \overline{\theta}\ \mathsf{frames}\ I \\ \Delta\langle\overline{\delta}\rangle; (\Theta \mathbin{\textcircled{\triangle}} I)\langle\rangle \vdash \{\, Q \wedge I \,\}\ B\ \{\, Q' \wedge I \,\}\,[\overline{\eta}] \qquad \mathit{Init} \Rightarrow I\end{array}}{\Delta\langle\overline{\delta}\rangle \vdash \{\, P \wedge \mathit{Init} \,\}\ \mathsf{let}\ m\ \mathsf{be}\ B\ \mathsf{in}\ C\ \{\, P' \,\}\,[\overline{\varepsilon}]}$$

The client C is obliged to respect $\overline{\theta}$ (and also $\overline{\delta}$) but does not see the hidden invariant. The implementation B is verified under additional precondition I and has additional obligation to reestablish I. (In the general case there is a list of bodies B_i, each verified in the same context against the specification for m_i.) The context Δ is another module that may be used both by C and by the implementation B of m. So B must respect $\overline{\delta}$, but note that it is not required (or likely) to respect $\overline{\theta}$. The obligation on B refers to context $\Theta \mathbin{\textcircled{\triangle}} I$, not Θ; this is only relevant if B recursively invokes m (or, in general, other methods of the same module). The operation $\textcircled{\triangle} I$ conjoins a formula I to pre- and post-conditions of specifications: $(\{Q\}m(x\colon T)\{Q'\}[\overline{\eta}]) \mathbin{\textcircled{\triangle}} I = \{Q \wedge I\}m(x\colon T)\{Q' \wedge I\}[\overline{\eta}]$.

Typical formalizations of data abstraction include a command for initialization, so a closed client program takes the form $\mathsf{let}\ m\ \mathsf{be}\ B\ \mathsf{in}\ (\mathit{init}; C)$. With dynamic allocation, it is constructors that do much of the work to establish invariants. In order to avoid the need to formalize constructors, we use an initial condition. For the Set example, take

$Init_{set}$ to be the condition $pool = \varnothing$ which is suitable to be declared in the module interface. Note that $Init_{set} \Rightarrow I_{set}$ is valid.

Remarkably, there is a simple interpretation of judgement (3) that captures the idea that C respects the boundaries $\overline{\delta}$ and $\overline{\theta}$: No step of C's execution may write locations designated by $\overline{\delta}$ —interpreted in the pre-state of that step— unless it is a step of a procedure of Δ; *mutatis mutandis* for $\overline{\theta}$ and Θ. Before turning to the formal details, we discuss this proof obligation.

Verifying a client of SET. Using the public specifications of the four methods of Set, it is straightforward to prove that the client (1) establishes $b = false$. But there is an additional obligation, that every step respects the dynamic boundary $\overline{\theta}_{set}$. Consider the assignment $n.val := 1$ in (1), which is critical because I_{set} depends on field val. The effect of $n.val := 1$ is $\mathsf{wr}\, n.val$ and it must be shown to be outside the boundary $\overline{\theta}_{set}$. By definition of $\star$, we have that $\overline{\theta}_{set} \star \mathsf{wr}\, n.val$ is $\{n\} \# pool \wedge \{n\} \# pool\text{`}rep$, which simplifies to $n \notin pool \wedge n \notin pool\text{`}rep$. We have $n \notin pool$ because n is fresh and variable $pool$ is not updated by the client. The condition $n \notin pool\text{`}rep$ is more interesting. Note that I_{set} implies

$R\colon\ \ pool\text{`}rep\text{`}own \subseteq pool \wedge \mathsf{null} \notin pool$

Unlike I_{set}, this is suitable to appear in the module interface, as a public invariant [23] or explicitly conjoined to the procedure specifications of SET. The client does not update the default value, null, of $n.own$. Together, R and $n.own = \mathsf{null}$ imply $n \notin pool\text{`}rep$.

One point of this example is that "package confinement" [14] applies here: references to the instances of $Node$ used by the Set implementation are never made available to client code. Thus a lightweight, type-based confinement analysis of the module could be used together with simple syntactic checks on the client to verify that the boundary is respected. The results of an analysis could be expressed in first order assertions like R and thus be checked rather than trusted by a verifier.

As in rule FRAME, the separator can be used to express that a primitive command respects a dynamic boundary, allowing precise reasoning in cases like module MM (Sect. 5) that are not amenable to general purpose static analyses. 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 $\overline{\theta}_{set}$ depends.[7] So interface specifications need to provide clients with sufficient information to reason about the boundary. For MM, it is not an invariant like R but rather the individual method specifications that facilitate such reasoning (see Sect. 5).

Formalization. The beauty of the second order frame rule, the form of which is due to O'Hearn et al [29], is that it distills the essence of Hoare's mismatch. Rule MISMATCH is derived in Fig.8 from our rule SOF together with two unsurprising rules which are among those given in Fig. 9. Before turning to the rules we define the semantics.

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. We define $\mathrm{Active}(C_1; C_2) = \mathrm{Active}(C_1)$ and $\mathrm{Active}(C) = C$ if there are no C_1, C_2 such that C is $C_1; C_2$.

[7] State-dependent effects may interfere, which is handled by the sequence rule [2].

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

Fig. 8. Derivation of rule MISMATCH, where Θ is a single specification $\{Q\}m(x\colon T)\{Q'\}[\overline{\eta}]$ and we write $\cdot$ for $\wedge$ to save space. The side condition for SOF is $I \vdash (\overline{\theta}, \mathsf{rd\,alloc})\ \mathsf{frames}\ I$. The next step is by rule LINK, followed by CONSEQ with side condition $\mathit{Init} \Rightarrow I$.

$$\text{SOF}\ \frac{\Delta\langle\overline{\delta}\rangle;\Theta\langle\overline{\theta}\rangle \vdash \{P\}\ C\ \{P'\}\,[\overline{\varepsilon}] \qquad I \vdash (\overline{\theta}, \mathsf{rd\,alloc})\ \mathsf{frames}\ I \qquad \mathrm{Admiss}(I,\Theta)}{\Delta\langle\overline{\delta}\rangle;(\Theta \circledcirc I)\langle\overline{\theta}\rangle \vdash \{P\wedge I\}\ C\ \{P'\wedge I\}\,[\overline{\varepsilon}]}$$

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

$$\text{CALL}\ \frac{\{P\}m(x\colon T)\{P'\}[\overline{\varepsilon}]\ \text{ is in } \Theta \qquad P^x_z \Rightarrow \overline{\delta}\star\overline{\varepsilon}^x_z}{\Delta\langle\overline{\delta}\rangle;\Theta\langle\overline{\theta}\rangle \vdash \{P^x_z\}\ m(z)\ \{P'^x_z\}\,[\overline{\varepsilon}^x_z]}$$

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

Fig. 9. Selected proof rules

Definition 1. *A correctness judgement* $\Delta\langle\overline{\delta}\rangle;\Theta\langle\overline{\theta}\rangle \vdash^{\Gamma} \{P\}\ C\ \{P'\}\,[\overline{\varepsilon}]$ *is* valid *iff the following holds. Let* Δ' *be the catenated list* (Δ,Θ)*, let* C_0 *be* C*, and let* μ_0 *be an arbitrary procedure environment disjoint from the procedures bound within* C *or present in* Δ,Θ*. Then for all* Γ*-states* σ_0 *such that* $\sigma_0 \models P$

(i) It is not the case that $\langle C_0, \sigma_0, \mu_0\rangle \overset{\Delta'}{\longmapsto}{}^*$ fault.

(ii) Every terminating computation $\langle C_0, \sigma_0, \mu_0\rangle \overset{\Delta'}{\longmapsto}{}^*\langle \mathsf{skip}, \sigma_n, \mu_n\rangle$ *satisfies* $\sigma_n \models P'$ *and* $\sigma_0 \rightsquigarrow \sigma_n \models \overline{\varepsilon}$.

(iii) For any reachable computation step, i.e. $\langle C_0, \sigma_0, \mu_0\rangle \overset{\Delta'}{\longmapsto}{}^*\langle C_{i-1}, \sigma_{i-1}, \mu_{i-1}\rangle \overset{\Delta'}{\longmapsto} \langle C_i, \sigma_i, \mu_i\rangle$*, either* $\mathrm{Active}(C_{i-1})$ *is a call to some* m *in* Δ *(respectively, in* Θ*) or else* $\mathrm{Agree}(\sigma_{i-1},\sigma_i,\overline{\delta})$ *(respectively,* $\mathrm{Agree}(\sigma_{i-1},\sigma_i,\overline{\theta})$*).*

Let us paraphrase (iii) in a way that makes clear the generalization to contexts with more modules: Every dynamic encapsulation bound must be respected by every step of computation (terminating or not), with the exception that a call of a context procedure is exempt from the bound of its module.

Selected proof rules are given in Fig. 9. An implicit side condition on all proof rules is that both the consequent and the antecedents are well formed. We omit standard rules for control structures, and structural rules like consequence, which do not manipulate the procedure context. Rule FRAME also leaves its context unchanged. For the

assignment commands we can use "small axioms" inspired by [29]. The axioms have empty context; rule CTXINTRO is used to add hypotheses.

Rule CTXINTRO is restricted to primitive commands (Fig. 5), because the side condition $P \Rightarrow \overline{\theta} \star \overline{\varepsilon}$ only enforces the dynamic encapsulation boundary $\overline{\theta}$ for the initial and final states —there are no intermediate steps in the semantics of these commands. Note that CTXINTRO introduces a dynamic boundary $\overline{\theta}$ that will not be imposed on the implementations of the procedures of the outer module Δ. This works because, due to nesting, those implementations cannot invoke procedures of Θ at all. The implementation of a procedure m in Θ may invoke a procedure p of enclosing module Δ. The effect of that invocation might even violate the dynamic boundary $\overline{\theta}$, but there is no harm —indeed, the implementation of m is likely to temporarily falsify the invariant for Θ but is explicitly obliged to restore it.

The implementation of an inner module is required (by rule LINK) to respect the encapsulation boundaries of enclosing modules. That is why it is sound for procedure m in rule CALL to be in the scope of the dynamic effect bound $\overline{\delta}$ with only the obligation that the end-to-end effect $\overline{\varepsilon}^x_z$ is separate from $\overline{\delta}$. The general form of CALL has n contexts and the called procedure is in the innermost. Additional context can subsequently be introduced on the inside, e.g., CALL can be used for a procedure of Θ and then the context extended to $\Delta\langle\overline{\delta}\rangle; \Theta\langle\overline{\theta}\rangle; \Upsilon\langle\overline{\upsilon}\rangle$ using rule CTXINTRO. In case there is only a single module, rule CALL can be used with Δ and $\overline{\delta}$ empty.

Rule SOF imposes an admissibility condition on I. In this paper, Admiss is defined to say I must not be falsifiable by allocation (i.e. $\sigma \models I$ implies $\sigma' \models I$, if σ' is just σ extended with a new object). The issue is that some useful invariants include alloc in their footprint, especially if the footprint is derived using our rules for framing [2].[8] Typical clients do allocation, and thus write alloc, which would conflict with a dynamic boundary containing $\mathsf{rd}\,\mathsf{alloc}$ (cf. [33]). The rule explicitly allows this conflict: by condition $P \vdash (\overline{\delta}, \mathsf{rd}\,\mathsf{alloc})\ \mathsf{frames}\ Q$, it appears that Q depends on alloc, but by condition $\mathrm{Admiss}(Q,\Theta)$ it does not. We include Θ in the notation, even though it is not used in the definition, because in a richer language with constructor methods there is a more practical definition of Admiss that allows the conflict. We can allow a module invariant I to have subformulas $\forall x : K \in \mathsf{alloc} \mid P(x)$ that do depend on alloc, and yet not include alloc in the dynamic bound, because the constructor will be obliged to maintain I.

Theorem 1. *Each of the rules is sound. Hence any derivable correctness judgement is valid.*

5 Specification and Verification of the Examples

For the toy memory manager of Sect. 2.2, we specify the effects for procedure *alloc* to be $\mathsf{wr}\,\mathsf{result}, \mathit{freed}, \mathit{flist}, \mathit{count}, \mathsf{alloc}, \mathit{freed}`\mathit{nxt}$. For $\mathit{free}(n : \mathit{Node})$ the effects are $\mathsf{wr}\,\mathit{freed}, \mathit{flist}, \mathit{count}, \mathit{freed}`\mathit{nxt}$. Ordinary scoping could be used to hide effects on the module variables *flist* and *count*, and the ghost *freed* could be "spec-public", i.e.

[8] An example such I is $\forall x : K \in \mathsf{alloc} \mid x.\mathit{init} \Rightarrow P(x)$ with *init* a boolean field, initially false. Such a formula would be suitable as an invariant in a program where $x.\mathit{init}$ only gets truthified by procedures that also establish $P(x)$.

not writeable outside module MM. To frame I_{mm} we choose as dynamic boundary $\mathsf{rd}\, freed, flist, count, freed`nxt$. The interesting part is $freed`nxt$, as $flist$ and $count$ should be scoped within the module and $freed$ should be spec-public. Using the specifications in Sect. 2.2 together with these effect specifications, it is straightforward to verify the client given there. The client writes $freed`val$ but it does not write $freed`nxt$, nor variable $freed$ itself, and thus it respects the dynamic boundary. So it can be linked with $alloc$ and $free$ using rule MISMATCH. By contrast with the use of an invariant, R, to verify that client (1) respects the dynamic boundary $\overline{\theta}_{set}$, here it is the procedure specifications themselves that support reasoning about the dynamic boundary. Suppose we add the assignment $y.nxt := null$ just after $y := alloc()$; although this writes a nxt field, the object is outside $freed$ according to the specification of $alloc$.

Recall the example of Sect. 2.3. For method $update$ we choose effects $\mathsf{wr}\, \mathsf{self}.val, \mathsf{self}.O`cache$. The effects for $Observer(u)$ are $\mathsf{wr}\, u.O`nxto, u.(O, dg)$. Here dg is a data group that abstracts the private field obs. These suffice to verify the client in Sect. 2.3 which relies on separation between subjects. The dynamic boundary, $\overline{\delta}_{ob}$, is $\mathsf{rd}\, \mathsf{alloc}`(O, dg), \mathsf{alloc}`O`nxto$. Region $\mathsf{alloc}$ is very coarse, but fields $O, dg, nxto$ could be protected from clients by scoping; indeed, we might simply use $\mathsf{alloc}`nxto$.[9] Verification of the implementations uses rule FRAME to exploit per-subject separation, similar to the Set example in Sect. 4. Then rule MISMATCH links the client.

Finally, recall the example of nested modules and overlapping data structures in Sect. 2.4. Let the dynamic boundary be $\mathsf{rd}\, \mathsf{alloc}, \mathsf{alloc}`(next, prev)$, which frames I_{os}. Consider a client that constructs a new $ObsSet$. The implementation of the $ObsSet$ constructor can be verified, assuming and maintaining I_{os}, including the obligation to respect the dynamic boundary $\overline{\delta}_{ob}$ of module OB. The client can be linked to OS using rule MISMATCH and then that rule is used again to link with module OB.

6 Related Work

It is notoriously difficult to achieve encapsulation in the presence of shared, dynamically allocated mutable objects [22,30]. Current tools for automated software verification either do not support hiding of invariants (e.g., Jahob [39], jStar [10], Krakatoa [12]), do not treat object invariants soundly (e.g., ESC/Java [13]) or at best offer soundness for restricted situations where a hierarchical structure can be imposed on the heap (e.g. Spec# [3]). Some of these tools do achieve significant automation, especially by using SMT solvers [21].

The use of ghost state to encode inductive properties without induction has been fruitful in verifications using SMT solvers (e.g., [8,16,39]). Our use of ghost state for frame conditions and separation reasoning was directly inspired by the state-dependent effects of Kassios [18] (who calls them dynamic frames, whence our term "dynamic boundary"). Variations on state-dependent effects have been explored in SMT-based verifiers, e.g., Smans et al implemented a verifier that abstracts footprints using location sets and pure method calls in assertions and in frame conditions [37]. Another verifier uses novel assertions for an implicit encoding (inspired by separation logic) of frame

[9] In fact $nxto$ should be abstracted by a data group, but we report here on the version for which we did a detailed proof.

conditions by preconditions [36]. Leino's Dafny tool [24] features effects in the form we write as G`any. The Boogie tool [3] has been used for experiments with region logic specifications of the Observer [1] and Composite [34] patterns.

Hiding is easy to encode in an axiomatic semantics —it is just Hoare's mismatch, phrased in terms of assert and assume statements. The verifiers above which provide hiding enforce specific encapsulation disciplines through some combination of type checking and extra verification conditions. For example, the Boogie methodology [25] used by Spec# stipulates intermediate assertions (in all code) that guarantees an all-states ownership invariant. Another version of Spec# [37] generates verification conditions at intermediate steps to approximate read footprints, in addition to the usual end-to-end check for modifies specifications of methed bodies. One way to enforce our requirement for respecting dynamic boundaries would be to generate verification conditions for writes at intermediate steps, which could be optimized away in cases where their validity is ensured by a static analysis.

A number of methodologies have been proposed for ownership-based hiding of invariants (e.g., [28]). Drossopoulou et al. [11] introduce a general framework to describe verification techniques for invariants. A number of ownership disciplines from the literature are studied as instances of the framework. The framework encompasses variations on the idea that invariants hold exactly when control crosses module boundaries, e.g., *visible state semantics* requires all invariants to hold on all public method call/return boundaries; other proposals require invariants to hold more often [25] or less [38]. The difficulty of generalizing ownership to fit important design patterns led Parkinson and Bierman [5,31] to pursue abstraction instead of hiding, via second order assertions in separation logic; this has been implemented [10].

Separation logic (SL) is a major influence on our work. Our SOF rule is adapted from [30], as is the example in Sect. 2.2. The SOF rule of SL relies on two critical features: the separating conjunction and the *tight interpretation* of a correctness judgement $\{P\}C\{Q\}$ which requires that C neither reads nor writes outside the footprint of P. These features yield great economy of expression, but conflating read and write has consequences. To get shared reads, the semantics of separating conjunction can embody some notion of permissions [7] which adds complication but is useful for concurrent programs (and to our knowledge has not been combined with SOF). The SOF rule of SL also hides effects on encapsulated state whereas our SOF rule hides only the invariant. By disentangling the footprint from the state condition we enable shared reads (retaining a simple semantics), but that means we cannot hide effects within the dynamic encapsulation boundary —the effects can be visible to clients.

Both our FRAME rule and our SOF rule use ordinary conjunction to introduce an invariant, together with side conditions that designate a footprint of the invariant which is separated from the write effect of a command. In SL these rules use the separating conjunction which expresses the existence of such footprints for the command's precondition and for the invariant. Reynolds gave a derivation using the rule of conjunction[10] that shows the SOF rule of SL is not sound without restriction to predicates that are

[10] From $\{P\}C\{P'\}$ and $\{Q\}C\{Q'\}$ infer $\{P \wedge Q\}C\{P' \wedge Q'\}$.

"precise" in the sense of determining a unique footprint [30].[11] The semantic analysis in [30] shows that the need for a unique footprint applies to region logic as well. However, region logic separates the footprint from the formula, allowing the invariant formula to denote an imprecise predicate while framing the formula by effects that in a given state determines a unique set of locations.

The restriction to precise predicates for SOF in SL can be dropped using a semantics that does not validate the rule of conjunction [6]. This was eschewed by the authors of [30] because the rule is patently sound in ordinary readings of Hoare triples. Dropping the rule facilitates the modeling of higher order framing rules that capture visible state semantics for invariants even in programs using code pointers (e.g., [35]). The metatheory underlying the Ynot tool for interactive verification [27] uses a model that does not validate the conjunction rule [32]. Higher order separation logics offer elegant means to achieve data abstraction and strong functional specifications of interesting design patterns [20,19,27]. The ability to explicitly quantify over invariants would seem to lessen the importance of hiding, but it requires considerable sophistication on the part of the user and their reasoning tools.

7 Conclusion

In this paper we explore a novel interface specification feature: the *dynamic boundary* which must be respected by clients. The dynamic boundary is designated by read effects that approximate, in a way suitable to appear in the interface, the footprint of an invariant which is hidden, i.e. does not appear in the interface specifications. Explicit description of footprints is complementary to syntactic mechanisms that encapsulate state named by identifiers. The expressions whose l-values constitute the dynamic boundary are state-dependent and thus denote different sets of locations over time.

Hiding is formalized in a second order frame rule that is proved sound for a simple operational semantics of sequential programs. We show by examples that our SOF handles not only invariants that pertain to several objects with a single owner but also design patterns in which several client-reachable peers cooperate and in which data structures may be overlapping or irregular. These are incompatible with ownership and remain as challenge problems in the current literature [4,22,27]. 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 [9] as well as less restrictive disciplines that impose proof obligations on clients, e.g., ownership transfers that are "in the eye of the asserter" [30].

One of our aims is to provide a logical foundation that can justify the axiomatic semantics used in automated verifiers. Even more, we want a framework in which encapsulation disciplines, both specialized and general-purpose, can be specified in program annotations and perhaps "specification schemas" or aspects —so that soundness for hiding becomes a verification condition rather than a meta-theorem. This could improve usability and applicability of verifiers, e.g., by deploying disciplines on a per-module

[11] A predicate I is *precise* iff $(I * \_)$ distributes over $\wedge$. In this paper our invariants are all precise, but not all useful ones are, e.g., "there exists a non-full queue".

basis. It could also facilitate foundational program proofs, by factoring methodological considerations apart from the underlying program model embodied in axiomatic semantics. Our approach does not rely on inductive predicates, much less higher order ones, but on the other hand it does not preclude the use of more expressive assertions (such as the inductive FC in the example in Sect. 2.2).

It remains to be seen how the approach explored here extends to more advanced programming features such as code pointers and concurrency. There are a number of more immediate issues such as integration with a proper module system, inference of ghost annotations based on static analysis, and full encapsulation for representation independence and for hiding of effects.

Acknowledgements. Many people helped with advice and encouragement, including Lennart Beringer, Lars Birkedal, Sophia Drossopoulou, Bart Jacobs, Gary Leavens, Peter Müller, Peter O'Hearn, Matthew Parkinson, Jan Smans, Stan Rosenberg, Jacob Thamsborg, Hongseok Yang, organizers and participants of Dagstuhl seminars 08061 and 09301.

References

1. Banerjee, A., Barnett, M., Naumann, D.A.: Boogie meets regions: A verification experience report. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 177–191. Springer, Heidelberg (2008)
2. Banerjee, A., Naumann, D.A., Rosenberg, S.: Regional logic for local reasoning about global invariants. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 387–411. Springer, Heidelberg (2008); Draft journal version available at authors' web sites
3. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
4. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape analysis for composite data structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
5. Bierman, G., Parkinson, M.: Separation logic and abstraction. In: POPL, pp. 247–258 (2005)
6. Birkedal, L., Torp-Smith, N., Yang, H.: Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. Logical Methods in Computer Science 2(5) (2006)
7. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: POPL, pp. 259–270 (2005)
8. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: a practical system for verifying concurrent C. In: TPHOLs, pp. 23–42 (2009)
9. Dietl, W., Müller, P.: Universes: Lightweight ownership for JML. Journal of Object Technology 4, 5–32 (2005)
10. Distefano, D., Parkinson, M.J.: jStar: Towards practical verification for Java. In: OOPSLA, pp. 213–226 (2008)
11. Drossopoulou, S., Francalanza, A., Müller, P., Summers, A.J.: A unified framework for verification techniques for object invariants. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 412–437. Springer, Heidelberg (2008)

12. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification (tool paper). In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
13. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI, pp. 234–245 (2002)
14. Grothoff, C., Palsberg, J., Vitek, J.: Encapsulating objects with confined types. ACM TOPLAS 29(6) (2007)
15. Harel, D., Pnueli, A., Stavi, J.: A complete axiomatic system for proving deductions about recursive programs. In: STOC, pp. 249–260 (1977)
16. Hawblitzel, C., Petrank, E.: Automated verification of practical garbage collectors. In: POPL, pp. 441–453 (2009)
17. Hoare, C.A.R.: Proofs of correctness of data representations. Acta Inf. 1, 271–281 (1972)
18. Kassios, I.T.: Dynamic framing: Support for framing, dependencies and sharing without restriction. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 268–283. Springer, Heidelberg (2006)
19. Krishnaswami, N.R., Aldrich, J., Birkedal, L.: Verifying event-driven programs using ramified frame properties. In: TLDI (2010)
20. Krishnaswami, N.R., Aldrich, J., Birkedal, L., Svendsen, K., Buisse, A.: Design patterns in separation logic. In: TLDI (2009)
21. Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. Springer, Heidelberg (2008)
22. Leavens, G.T., Leino, K.R.M., Müller, P.: Specification and verification challenges for sequential object-oriented programs. Formal Aspects of Computing 19(2), 159–189 (2007)
23. Leavens, G.T., Müller, P.: Information hiding and visibility in interface specifications. In: ICSE, pp. 385–395 (2007)
24. Leino, K.R.M.: Specification and verification in object-oriented software. Marktoberdorf lecture notes (2008)
25. Rustan, K., Leino, M., Müller, P.: Object invariants in dynamic contexts. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 491–515. Springer, Heidelberg (2004)
26. Leino, K.R.M., Poetzsch-Heffter, A., Zhou, Y.: Using data groups to specify and check side effects. In: PLDI, pp. 246–257 (2002)
27. Malecha, G., Morrisett, G., Shinnar, A., Wisnesky, R.: Toward a verified relational database management system. In: POPL (2010)
28. Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular invariants for layered object structures. Sci. Comput. Programming 62(3), 253–286 (2006)
29. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001)
30. O'Hearn, P.W., Yang, H., Reynolds, J.C.: Separation and information hiding. ACM TOPLAS 31(3), 1–50 (2009); Extended version of POPL 2004
31. Parkinson, M.: Class invariants: The end of the road. In: IWACO (2007)
32. Petersen, R.L., Birkedal, L., Nanevski, A., Morrisett, G.: A realizability model for impredicative Hoare type theory. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 337–352. Springer, Heidelberg (2008)
33. Pierik, C., Clarke, D., de Boer, F.S.: Controlling object allocation using creation guards. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 59–74. Springer, Heidelberg (2005)
34. Rosenberg, S., Banerjee, A., Naumann, D.A.: Local reasoning and dynamic framing for the composite pattern and its clients (submitted, 2009)

35. Schwinghammer, J., Yang, H., Birkedal, L., Pottier, F., Reus, B.: A semantic foundation for hidden state. In: Ong, L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 2–17. Springer, Heidelberg (2010)
36. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames: Combining dynamic frames and separation logic. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 148–172. Springer, Heidelberg (2009)
37. Smans, J., Jacobs, B., Piessens, F., Schulte, W.: An automatic verifier for Java-like programs based on dynamic frames. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 261–275. Springer, Heidelberg (2008)
38. Summers, A.J., Drossopoulou, S.: Considerate reasoning and the composite design pattern. In: Barthe, G., Hermenegildo (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 328–344. Springer, Heidelberg (2010)
39. Zee, K., Kuncak, V., Rinard, M.C.: Full functional verification of linked data structures. In: PLDI, pp. 349–361 (2008)