

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

17 Feb 2010 To appear in ESOP'10; this version fixes a bug and includes appendix.

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" [30, 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.

- 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 expressions 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 date 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 [31]. 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 simple 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).

This paper is slightly revised from the version to appear in ESOP’10: It removes from the second order frame rule an unsound “admissibility” condition. There is an appendix with proofs, and a few copy edits were made.

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. 1.³ Annotations include method postconditions that refer to a global

³ 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

```

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, *pool*, marked as ghost state. Ghost variables and fields are auxiliary state used 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 $s.rep \overset{\bullet}{\smile} nxt$ denotes the region consisting of *nxt* values of objects in region *s.rep*. It is used in this definition:⁴

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

have an implicit parameter, *self*, which may be elided in field updates; e.g., the assignment $lst := null$ in the body of the *Set* constructor is short for $self.lst := null$. Variable *result* is the returned result; there is no explicit ‘return’ statement.

⁴ The range condition “ $n \in s.rep$ ” is false in case *s* is null, because $n \in s.rep$ is shorthand for $n \in \{s\} \overset{\bullet}{\smile} rep$ and $\{null\} \overset{\bullet}{\smile} rep$ 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 $G \overset{\bullet}{\smile} rep$ is the union of *rep* images whereas $G \overset{\bullet}{\smile} nxt$ is the set of *nxt* images, because *rep* has type *rgn* and *nxt* has class type.

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 $s.rep \text{ ' } next \subseteq s.rep$ says that $s.rep$ is *next*-closed; this is equivalent to $\forall o \mid o \in s.rep \Rightarrow o.next \in s.rep$.⁵ 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 *next*-closed even if $s.rep$ contained extraneous objects, in particular nodes reached from other instances of *Set*. This is prevented by the inclusion $s.rep \text{ ' } own \subseteq \{s\}$; or rather, by requiring the inclusion for every instance of *Set*. So we adopt an invariant to be associated with module *SET*:

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

The need for $\text{null} \notin pool$ is minor and discussed later. A bigger concern is the global nature of I_{set} , which is addressed in Sect. 3.2. Quantification over *pool*, rather than *alloc*, is an important idiom; as explored later, it prevents falsification by allocation.

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

$$\begin{aligned} & \text{var } s: Set := \text{new } Set; \text{ var } n: Node := \text{new } Node; \\ & s.add(1); s.add(2); n.val := 1; s.remove(1); b := s.contains(1); \end{aligned} \quad (1)$$

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

⁵ Quantified variables range over non-null, allocated references.

```

ghost freed: rgn;
var flist: Node; count: int;
alloc(): Node
  ensures result ≠ null ∧ freed = old(freed) - {result} ∧ (result ∈ old(freed) ∨ fresh(result))
  { if count = 0 then result := new Node;
    else result := flist; flist := flist.next; count := count - 1; freed := freed - {result}; endif }
free(n: Node) requires n ≠ null ∧ n ∉ freed  ensures freed = old(freed) ∪ {n}
  { n.next := flist; flist := n; count := count + 1; freed := freed ∪ {n}; }

```

Fig. 2. Module *MM*.

restrict the form of invariants and the use of references, to support modular reasoning 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 $FC(flist, freed, count)$ where $FC(f: Node, r: rgn, c: int)$ is defined, by induction on the size of r , as

$$(f = \text{null} \Rightarrow r = \emptyset \wedge c = 0) \wedge (f \neq \text{null} \Rightarrow f \in r \wedge c > 0 \wedge FC(f.next, r - \{f\}, c - 1))$$

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 $count \neq 0 \Rightarrow flist \neq \text{null}$, as otherwise the dereference *flist.next* 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: Node; x := new Node; y := alloc(); free(x); free(y);
while y ≠ null do y.val := 7; y := y.next; od

```

The loop updates *val* fields of freed objects, but it does not write the *next* fields, on which the invariant depends; the client neither falsifies I_{mm} nor causes a fault. Suppose we replaced the loop by the assignment $y.next := \text{null}$. 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 *next* 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. 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.nxt; od }
  get(): int { result := val; }
  register(b: Observer) { b.nxt := obs; obs := b; O := O ∪ {b}; b.notify(); } }
class Observer { sub: Subject; cache: int; nxt: 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 [32] 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 *nxt*-linked list threaded through the observers themselves, and it relies on the hidden invariant

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

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

```

class ObsSet extends Observer { next: ObsSet; prev: ObsSet;
  ObsSet(s: Subject, os: ObsSet)
    requires  $\forall b: \text{Observer} \in s.O \mid \text{Obs}(b, s, s.val)$ 
    ensures  $\text{self} \in s.O \wedge \forall b: \text{Observer} \in s.O \mid \text{Obs}(b, s, s.val)$ 
  { 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*.

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

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

next and *prev* pointers, whose elements may be observing the same or different subjects. Second, each *ObsSet* is in the *next*-linked list of observers of its subject.

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 $\text{class } K \{ \bar{f}: \bar{T} \}$, where field types may make mutually recursive reference to other classes. We write $\text{Fields}(K)$ for $\bar{f}: \bar{T}$ and for simplicity let names in the list \bar{f} have global scope. Ordinary expressions do not depend on the heap: $y.f$ is not

$Set()$	$wr\ pool$
$add(i: int)$	$wr\ alloc, self.any, self.rep^{\text{any}}$
$remove(i: int)$	$wr\ self.any, self.rep^{\text{any}}$

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

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 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 $m(x: T)$. The form “let $m(x: T)$ be B in C ” is typable in context Π and Γ if both $\Pi, m: (x: T) \vdash^{\Gamma, x: T} B$ and $\Pi, m: (x: T) \vdash^\Gamma C$. 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; in a well formed state 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 , $\text{Type}(o, \sigma)$ is the type of an allocated reference o , $\text{Update}(\sigma, o.f, v)$ overrides σ to map field f of o to v (for $o \in \sigma(\text{alloc})$), $\text{Extend}(\sigma, x, v)$ extends σ to map x to value v (for $x \notin \text{Dom}(\sigma)$). Heaps have no dangling references; we do not model garbage collection or deallocation.

In a given state the region expression G^{f} (read “ G ’s image under f ”) denotes one of two things. If f has class type then G^{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^{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 $\sigma(x) \neq \text{null}$ 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: $\sigma \models^\Gamma (\forall x: K \mid P)$ iff $\text{Extend}(\sigma, x, o) \models^{\Gamma, x: K} P$ for all $o \in \sigma(\text{alloc})$ such that $\text{Type}(o, \sigma) = K$. 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 $\varepsilon ::= wr\ x \mid rd\ x \mid wr\ G^{\text{f}} \mid rd\ G^{\text{f}} \mid fr\ G$. 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 $\text{fr } G$ is used for commands; it says that the value of G in 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, $\text{Set}()$, says variable pool may be updated. For add , the effect wr alloc means that new objects may be allocated. The effect wr self.any says that any fields of self may be written. The effect $\text{wr self.rep}^{\text{any}}$ says that any field of any object in self.rep may be written; in fact none are written in our implementation, but this caters for other implementations. The effect $\text{wr self.rep}^{\text{any}}$ 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 $\text{wr } G^{\text{any}} f$ 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 $\text{wr } x.f$ abbreviates $\text{wr } \{x\}^{\text{any}} f$. 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 σ' *succeeds* σ , written $\sigma \hookrightarrow \sigma'$, iff $\sigma(\text{alloc}) \subseteq \sigma'(\text{alloc})$ and $\text{Type}(o, \sigma) = \text{Type}(o, \sigma')$ for all $o \in \sigma(\text{alloc})$. (And of course both states are well formed.) We say effect list $\bar{\epsilon}$ *allows transition from* σ *to* σ' , written $\sigma \rightsquigarrow \sigma' \models \bar{\epsilon}$, if and only if $\sigma \hookrightarrow \sigma'$ and

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

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 $\text{SetI}(s)$, which will be used for “local reasoning” [30] at the granularity of a single instance of Set . We choose to frame⁶ it by

$\bar{\delta}_0$: rd $s, s.(rep, lst), s.rep^{\text{any}}(next, val, own)$ (abbreviating $s.rep, s.lst$, etc.)

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

We use a judgement for framing of formulas, e.g., $\text{true} \vdash \bar{\delta}_0$ frames $\text{SetI}(s)$ says that if two states agree on the locations designated by $\bar{\delta}_0$ then they agree on the value of $\text{SetI}(s)$. 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 $s \in \text{pool} \vdash \text{rd } \text{pool}^{\text{any}}(rep, lst)$ frames $s.lst \in s.rep$.

The semantics of judgement $P \vdash \bar{\delta}$ frames P' is specified by the following: If $\sigma \models P$ and $\text{Agree}(\sigma, \sigma', \bar{\delta})$ then $\sigma \models P'$ implies $\sigma' \models P'$. Here $\text{Agree}(\sigma, \sigma', \bar{\delta})$ is defined to

⁶ 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.

mean: $\sigma \rightsquigarrow \sigma'$, $\sigma(x) = \sigma'(x)$ for all rd x in $\bar{\delta}$, and $\sigma(o.f) = \sigma'(o.f)$ for all rd $G'f$ in $\bar{\delta}$ and all $o \in \sigma(G)$ with $f \in \text{Fields}(o, \sigma)$.

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 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 $P \vdash \bar{\eta}$ frames $(\forall x: K \mid x \in G \Rightarrow P')$ has antecedent of the form $P \wedge x \in G \vdash \bar{\eta}'$ frames P' and requires $\bar{\eta}$ 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 $\text{Agree}(\sigma, \sigma', \bar{\eta})$.

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

$\bar{\theta}_{set}: \text{rd } pool, pool^{\text{any}}, pool^{\text{any}}(rep^{\text{any}}, val, own)$

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 $wr \text{ alloc}, \text{self.any}$ and thus those for *add* in Fig. 6. Moreover we can verify the following Hoare triple:

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

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 $n \neq \text{null}$, 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 $G \cap G' \subseteq \{\text{null}\}$. Now I_{set} is logically equivalent to $SetI(\text{self}) \wedge I_{except}$, with $\bar{\delta}_x$ framing I_{except} , defined as

$I_{except}: \text{null} \notin pool \wedge \forall s \in pool - \{\text{self}\} \mid SetI(s)$

$\bar{\delta}_x: \text{rd self}, pool, (pool - \{\text{self}\})^{\text{any}}(rep, lst), (pool - \{\text{self}\})^{\text{any}}(rep^{\text{any}}, val, own)$

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

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

$$\text{FRAME} \frac{\Delta \vdash \{P\} C \{P'\} [\bar{\epsilon}] \quad P \vdash \bar{\delta} \text{ frames } Q \quad P \Rightarrow \bar{\delta} \star \bar{\epsilon}}{\Delta \vdash \{P \wedge Q\} C \{P' \wedge Q\} [\bar{\epsilon}]}$$

Similar reasoning verifies the implementation of *remove*. Note that its effects include $wr \text{ self.rep}^{\text{any}}$. Moreover $\bar{\delta}_x \star wr \text{ self.rep}^{\text{any}}$ yields nontrivial disjointnesses:

$\text{self.rep} \# (\text{pool} - \{\text{self}\}) \wedge \text{self.rep} \# (\text{pool} - \{\text{self}\})^{\text{rep}}$. The first conjunct holds because elements of self.rep have type Node and those of $\text{pool} - \{\text{self}\}$ 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 $\text{true} \wedge I_{\text{set}}$ because true is the precondition of remove in Fig. 1.

3.3 Correctness judgements and program semantics

A *procedure context*, Δ , is a comma-separated list of specifications, each of the form $\{Q\}m(x:T)\{Q'\}[\bar{\epsilon}]$. For the specification to be well formed in a variable context Γ , all of $Q, Q', \bar{\epsilon}$ should be well formed in $\Gamma, x:T$. Moreover the frame condition $\bar{\epsilon}$ must not contain $wr x$, so the use of x in Q' and $\bar{\epsilon}$ refers to its initial value. A correctness judgement takes roughly the form $\Delta \vdash^{\Gamma} \{P\} C \{P'\} [\bar{\epsilon}]$ and is well formed if $\Delta, P, P', \bar{\epsilon}$ are well formed in Γ and $\text{signatures}(\Delta) \vdash^{\Gamma} C$. 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 $\bar{\epsilon}$.

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 .⁷ 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 $(\lambda x:T.C)$. By assuming that in a well formed program no procedure names are shadowed, we can use this simple representation, together with a special command $\text{end}(m)$ 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 $\text{end}(x)$.

The transition relation $\xrightarrow{\Delta}$ 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 $\langle \text{skip}, \sigma, \mu \rangle$, or else “fault” which results from null dereference. The cases omitted from Fig. 7 are quite standard. We note only that the semantics of 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.

4 Dynamic boundaries and second order framing

Rule FRAME is useful for reasoning about preservation of a predicate by a command that is explicitly responsible for it, like I_{set} and B_{add} in Sect. 3.2. For the client (1), we want I_{set} to be preserved, and 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

⁷ Such semantics is popular in work on program refinement; see also O’Hearn et al. [31].

$$\begin{array}{c}
\langle \text{let } m(x:T) \text{ be } B \text{ in } C, \sigma, \mu \rangle \mapsto \langle (C; \text{end}(m)), \sigma, \text{Extend}(\mu, m, (\lambda x:T.B)) \rangle \\
\frac{\mu(m) = \lambda x:T.B \quad x' \notin \text{Dom}(\sigma) \quad x' \notin \text{params}(\Delta) \quad B' = B_{x'}}{\langle m(z), \sigma, \mu \rangle \mapsto \langle (B'; \text{end}(x')), \text{Extend}(\sigma, x', \sigma(z)), \mu \rangle} \\
\frac{\Delta \text{ contains } \{P\}m(x:T)\{P'\}[\bar{\epsilon}] \quad \sigma \models P_z^x \quad \sigma' \models P_z^{x'} \quad \sigma \rightsquigarrow \sigma' \models \bar{\epsilon}_z^x}{\langle m(z), \sigma, \mu \rangle \mapsto \langle \text{skip}, \sigma', \mu \rangle} \\
\frac{\Delta \text{ contains } \{P\}m(x:T)\{P'\}[\bar{\epsilon}] \quad \sigma \not\models P_z^x \quad \sigma \hookrightarrow \sigma'}{\langle m(z), \sigma, \mu \rangle \mapsto \langle \text{skip}, \sigma', \mu \rangle \quad \text{and also} \quad \langle m(z), \sigma, \mu \rangle \mapsto \text{fault}}
\end{array}$$

Fig. 7. The transition relation \mapsto^{Δ} . Here Δ is the same throughout and omitted.

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 $\bar{\delta}$ is associated with a list Δ of procedure specifications using notation $\Delta \langle \bar{\delta} \rangle$. The general form for correctness judgement would have a sequence $\Delta_1 \langle \bar{\delta}_1 \rangle; \dots; \Delta_n \langle \bar{\delta}_n \rangle$ of hypotheses, for n modules, $n \geq 0$. In an attempt to improve readability, we will state the rules for the special case of just two modules, typically using name Θ for Δ_n . So a correctness judgement has the form

$$\Delta \langle \bar{\delta} \rangle; \Theta \langle \bar{\theta} \rangle \vdash^{\Gamma} \{P\} C \{P'\} [\bar{\epsilon}] \quad (3)$$

where $\bar{\delta}$ and $\bar{\theta}$ 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 $\bar{\delta}$. For a dynamic boundary to be useful it should frame the invariant to be hidden, e.g., $\bar{\theta}_{set}$ frames I_{set} . That proof obligation is on the module.

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

$$\text{MISMATCH} \frac{\frac{\Delta \langle \bar{\delta} \rangle; \Theta \langle \bar{\theta} \rangle \vdash \{P\} C \{P'\} [\bar{\epsilon}] \quad I \vdash \bar{\theta} \text{ frames } I}{\Delta \langle \bar{\delta} \rangle; (\Theta \circledast I) \langle \rangle \vdash \{Q \wedge I\} B \{Q' \wedge I\} [\bar{\eta}] \quad \text{Init} \Rightarrow I}}{\Delta \langle \bar{\delta} \rangle \vdash \{P \wedge \text{Init}\} \text{let } m \text{ be } B \text{ in } C \{P'\} [\bar{\epsilon}]}$$

The client C is obliged to respect $\bar{\theta}$ (and also $\bar{\delta}$) 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 $\bar{\delta}$, but note that it is not required (or likely) to respect $\bar{\theta}$. The obligation on B refers to context $\Theta \circledast I$, not Θ ; this is only relevant if B recursively invokes m (or, in general, other methods of the same module). The operation $\circledast I$ conjoins a for-

mula I to pre- and post-conditions of specifications: $(\{Q\}m(x:T)\{Q'\}[\bar{\eta}]) \otimes I = \{Q \wedge I\}m(x:T)\{Q' \wedge I\}[\bar{\eta}]$.

Typical formalizations of data abstraction include a command for initialization, so a closed client program takes the form `let m be B in $(init; C)$` . 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 $pool = \emptyset$ 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 $\bar{\delta}$ and $\bar{\theta}$: No step of C 's execution may write locations designated by $\bar{\delta}$ —interpreted in the pre-state of that step— unless it is a step of a procedure of Δ ; *mutatis mutandis* for $\bar{\theta}$ and $\bar{\Theta}$. 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 $\bar{\theta}_{set}$. 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 $wr\ n.val$ and it must be shown to be outside the boundary $\bar{\theta}_{set}$. By definition of \star , we have that $\bar{\theta}_{set} \star wr\ n.val$ is $\{n\} \# pool \wedge \{n\} \# pool^{\star} rep$, which simplifies to $n \notin pool \wedge n \notin pool^{\star} rep$. We have $n \notin pool$ because n is fresh and variable $pool$ is not updated by the client (or one can argue using the evident type invariant about $pool$). The condition $n \notin pool^{\star} rep$ is more interesting. Note that I_{set} implies

$$R: pool^{\star} rep^{\star} own \subseteq pool \wedge null \notin pool$$

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 $n.own = null$ imply $n \notin pool^{\star} rep$.

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 $\bar{\theta}_{set}$ depends.⁸ 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).

⁸ State-dependent effects may interfere, which is handled by the sequence rule [2].

$$\frac{\frac{\Delta\langle\bar{\delta}\rangle;(\Theta\otimes I)\langle\rangle\vdash\{Q\cdot I\}B\{Q'\cdot I\}[\bar{\eta}]}{\Delta\langle\bar{\delta}\rangle\vdash\{P\cdot I\}\text{let }m\text{ be }B\text{ in }C\{P'\cdot I\}[\bar{\epsilon}]}\quad\frac{\Delta\langle\bar{\delta}\rangle;\Theta\langle\bar{\theta}\rangle\vdash\{P\}C\{P'\}[\bar{\epsilon}]}{\Delta\langle\bar{\delta}\rangle;(\Theta\otimes I)\langle\bar{\theta}\rangle\vdash\{P\cdot I\}C\{P'\cdot I\}[\bar{\epsilon}]}\text{SOF}}{\Delta\langle\bar{\delta}\rangle\vdash\{P\cdot\text{Init}\}\text{let }m\text{ be }B\text{ in }C\{P'\}[\bar{\epsilon}]}$$

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

Formalization. The beauty of the second order frame rule, the form of which is due to O’Hearn et al. [30], 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 $\text{Active}(C_1; C_2) = \text{Active}(C_1)$; and $\text{Active}(C) = C$ if there are no C_1, C_2 such that C is $C_1; C_2$.

Definition 1. A correctness judgement $\Delta\langle\bar{\delta}\rangle; \Theta\langle\bar{\theta}\rangle \vdash^\Gamma \{P\} C \{P'\} [\bar{\epsilon}]$ is valid iff the following holds. Let Δ' be the catenated list (Δ, Θ) . Let C_0 be C . Let μ_0 be an arbitrary environment that binds procedures disjoint from those 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 $\langle C_0, \sigma_0, \mu_0 \rangle \xrightarrow{\Delta'}^* \text{fault}$.
- (ii) Every terminating computation $\langle C_0, \sigma_0, \mu_0 \rangle \xrightarrow{\Delta'}^* \langle \text{skip}, \sigma_n, \mu_n \rangle$ satisfies $\sigma_n \models P'$ and $\sigma_0 \rightsquigarrow \sigma_n \models \bar{\epsilon}$.
- (iii) For any reachable computation step, i.e. $\langle C_0, \sigma_0, \mu_0 \rangle \xrightarrow{\Delta'}^* \langle C_{i-1}, \sigma_{i-1}, \mu_{i-1} \rangle \xrightarrow{\Delta'} \langle C_i, \sigma_i, \mu_i \rangle$, either $\text{Active}(C_{i-1})$ is a call to some m in Δ (respectively, in Θ) or else $\text{Agree}(\sigma_{i-1}, \sigma_i, \bar{\delta})$ (respectively, $\text{Agree}(\sigma_{i-1}, \sigma_i, \bar{\theta})$).

Let us paraphrase (iii) in a way that makes clear the generalization to contexts with more modules: Every dynamic encapsulation boundary must be respected by every step of a 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 [30]. 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 $P \Rightarrow \bar{\theta} \star \bar{\epsilon}$ only enforces the dynamic encapsulation boundary $\bar{\theta}$ for the initial and final states —there are no intermediate steps in the semantics of these commands. Note that CTXINTRO introduces a dynamic boundary $\bar{\theta}$ that will not be imposed on the implementations of the procedures of the outer module Δ . This works because, due

$$\begin{array}{c}
\text{SOF} \frac{\Delta\langle\bar{\delta}\rangle; \Theta\langle\bar{\theta}\rangle \vdash \{P\} C \{P'\} [\bar{\epsilon}] \quad I \vdash \bar{\theta} \text{ frames } I}{\Delta\langle\bar{\delta}\rangle; (\Theta \otimes I)\langle\bar{\theta}\rangle \vdash \{P \wedge I\} C \{P' \wedge I\} [\bar{\epsilon}]} \\
\text{CTXINTRO} \frac{\Delta\langle\bar{\delta}\rangle \vdash \{P\} C \{P'\} [\bar{\epsilon}] \quad C \text{ is primitive} \quad P \Rightarrow \bar{\theta} \star \bar{\epsilon}}{\Delta\langle\bar{\delta}\rangle; \Theta\langle\bar{\theta}\rangle \vdash \{P\} C \{P'\} [\bar{\epsilon}]} \\
\text{CALL} \frac{\{P\} m(x:T)\{P'\} [\bar{\epsilon}] \text{ is in } \Theta \quad P_z^x \Rightarrow \bar{\delta} \star \bar{\epsilon}_z^x}{\Delta\langle\bar{\delta}\rangle; \Theta\langle\bar{\theta}\rangle \vdash \{P_z^x\} m(z) \{P_z^x\} [\bar{\epsilon}_z^x]} \\
\text{LINK} \frac{\Theta \text{ is } \{Q\} m(x:T)\{Q'\} [\bar{\eta}] \quad \Delta\langle\bar{\delta}\rangle; \Theta\langle\bar{\theta}\rangle \vdash^{\Gamma} \{P\} C \{P'\} [\bar{\epsilon}] \quad \Delta\langle\bar{\delta}\rangle; \Theta\langle\bar{\theta}\rangle \vdash^{\Gamma, x:T} \{Q\} B \{Q'\} [\bar{\eta}]}{\Delta\langle\bar{\delta}\rangle \vdash^{\Gamma} \{P\} \text{ let } m(x:T) \text{ be } B \text{ in } C \{P'\} [\bar{\epsilon}]}
\end{array}$$

Fig. 9. Selected proof rules.

to nesting, those implementations cannot invoke procedures of Θ at all. On the other hand, 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 $\bar{\theta}$, 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 $\bar{\delta}$ with only the obligation that the end-to-end effect $\bar{\epsilon}_z^x$ is separate from $\bar{\delta}$. 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 $\Delta\langle\bar{\delta}\rangle; \Theta\langle\bar{\theta}\rangle; \Upsilon\langle\bar{\nu}\rangle$ using rule CTXINTRO. In case there is only a single module, rule CALL can be used with Δ and $\bar{\delta}$ empty.

The general form of LINK binds a list \bar{B} of bodies to a list \bar{m} of procedure names, the specifications of which comprise Θ .

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 $wr \text{ result}, freed, flist, count, alloc, freed^{\bullet} \text{ next}$. For *free*($n: Node$) the effects are $wr \text{ freed}, flist, count, freed^{\bullet} \text{ next}$. 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. not writeable outside module *MM*. To frame I_{mm} we choose as dynamic boundary $rd \text{ freed}, flist, count, freed^{\bullet} \text{ next}$. The interesting part is $freed^{\bullet} \text{ next}$, 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 $\text{freed}^{\text{val}}$ but it does not write $\text{freed}^{\text{next}}$, nor variable freed itself, and thus it respects the dynamic boundary. So it can be linked with alloc and free using rule MISMATCH (i.e. the general form that links multiple procedures). By contrast with the use of an invariant, R , to verify that client (1) respects the dynamic boundary $\bar{\theta}_{\text{set}}$, here it is the procedure specifications themselves that support reasoning about the dynamic boundary. Suppose we add the assignment $y.\text{next} := \text{null}$ just after $y := \text{alloc}()$; although this writes a next 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 $\text{wr self.val, self.O}^{\text{cache}}$. The effects for $\text{Observer}(u)$ are $\text{wr } u.\text{O}^{\text{next}}, u.(O, dg)$. 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. For the invariant I_{ob} , as written the quantified variables s and o range over allocated objects of their type, so the footprint includes variable alloc .⁹ For dynamic boundary we might choose $\text{rd alloc}, \text{alloc}^{\text{O}(O, dg)}, \text{alloc}^{\text{O}^{\text{next}}}$. Region alloc is very coarse, but fields O, dg, next could be protected from clients by scoping (and next should be abstracted by a data group). There is a problem with this choice, however. Client code like $p := \text{new Observer}(t)$ writes not only the effects specified for Observer but also p and alloc —which would not respect a dynamic boundary that includes alloc . Invariants that quantify over allocated objects are at risk of violation by mere allocation [34, 29]. For our purposes, the desired effect can be achieved by introducing a ghost variable to serve as the range of quantification, like pool in the SET example in Sect. 4. For module OB , one variation uses a single pool containing both the subjects and the observers, in the dynamic boundary $\text{rd pool}, \text{pool}^{\text{O}(O, dg)}, \text{pool}^{\text{O}^{\text{next}}}$. The constructors need to update pool just like the Set constructor (and this is an idiom worthy of syntax sugar). For verification of the implementations in Sect. 2.3, we use rule FRAME to exploit per-subject separation, similar to the Set example. Then rule MISMATCH links the client.

Finally, recall the example of nested modules and overlapping data structures in Sect. 2.4. Again we maintain variable pool so the dynamic boundary for I_{os} can be $\text{rd pool}, \text{pool}^{\text{O}(next, prev)}$. 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 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, 31]. Current tools for automated software verification either do not support hiding of invariants (e.g., Jahob [40], 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.

⁹ A variation on our logic uses for each class K a distinct variable alloc_K to hold the allocated objects of type K , but that does not eliminate the issue discussed here.

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, 40]). 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 [38]. Another verifier uses novel assertions for an implicit encoding (inspired by separation logic) of frame conditions by preconditions [37]. 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 [35] 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 that 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# [38] generates verification conditions at intermediate steps to approximate read footprints, in addition to the usual end-to-end check that a method body satisfies its modifies specification. 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 [39]. The difficulty of generalizing ownership to fit important design patterns led Parkinson and Bierman [5, 32] 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 [31], 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 conjunction¹⁰ that shows the SOF rule of SL is not sound without restriction to predicates that are “precise” in the sense of determining a unique footprint [31].¹¹ The semantic analysis in [31] 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 [31] 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 something like visible state semantics for invariants even in programs using code pointers (e.g., [36]). The metatheory underlying the Ynot tool for interactive verification [27] uses a model that does not validate the conjunction rule [33]. 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 her 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 rule 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

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

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

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” [31].

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 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 suggestions 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. A. Banerjee, M. Barnett, and D. A. Naumann. Boogie meets regions: A verification experience report. In *VSTTE*, pages 177–191, 2008.
2. A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP*, pages 387–411, 2008. Draft journal version available online.
3. 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.
4. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O’Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, pages 178–192, 2007.
5. G. Bierman and M. Parkinson. Separation logic and abstraction. In *POPL*, pages 247–258, 2005.
6. L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. *Logical Methods in Comp. Sci.*, 2(5), 2006.
7. R. Bornat, C. Calcagno, P. O’Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270, 2005.
8. E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: a practical system for verifying concurrent C. In *TPHOLs*, pages 23–42, 2009.
9. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4:5–32, 2005.

10. D. Distefano and M. J. Parkinson. jStar: Towards practical verification for Java. In *OOPSLA*, pages 213–226, 2008.
11. 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.
12. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification (tool paper). In *CAV*, pages 173–177, 2007.
13. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245, 2002.
14. C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. *ACM TOPLAS*, 29(6), 2007.
15. D. Harel, A. Pnueli, and J. Stavi. A complete axiomatic system for proving deductions about recursive programs. In *STOC*, pages 249–260, 1977.
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*, pages 268–283, 2006.
19. N. R. Krishnaswami, J. Aldrich, and L. Birkedal. Verifying event-driven programs using ramified frame properties. In *TLDI*, 2010.
20. N. R. Krishnaswami, J. Aldrich, L. Birkedal, K. Svendsen, and A. Buisse. Design patterns in separation logic. In *TLDI*, 2009.
21. D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 2008.
22. 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.
23. G. T. Leavens and P. Müller. Information hiding and visibility in interface specifications. In *ICSE*, pages 385–395, 2007.
24. K. R. M. Leino. Specification and verification in object-oriented software. Marktoberdorf lecture notes, 2008.
25. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, pages 491–516, 2004.
26. K. R. M. Leino, A. Poetsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *PLDI*, pages 246–257, 2002.
27. G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Toward a verified relational database management system. In *POPL*, 2010.
28. P. Müller, A. Poetsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Programming*, 62(3):253–286, 2006.
29. D. A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. *Theoretical Comput. Sci.*, 365:143–168, 2006.
30. P. W. O’Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, pages 1–19, 2001.
31. P. W. O’Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. *ACM TOPLAS*, 31(3):1–50, 2009. Extended version of POPL 2004.
32. M. Parkinson. Class invariants: The end of the road? In *IWACO*, 2007.
33. R. L. Petersen, L. Birkedal, A. Nanevski, and G. Morrisett. A realizability model for impredicative Hoare type theory. In *ESOP*, pages 337–352, 2008.
34. C. Pierik, D. Clarke, and F. S. de Boer. Controlling object allocation using creation guards. In *Formal Methods*, pages 59–74, 2005.
35. S. Rosenberg, A. Banerjee, and D. A. Naumann. Local reasoning and dynamic framing for the composite pattern and its clients. Submitted, 2009.

36. J. Schwinghammer, H. Yang, L. Birkedal, F. Pottier, and B. Reus. A semantic foundation for hidden state. In *FoSSaCS*, 2010.
37. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP*, pages 148–172, 2009.
38. 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.
39. A. J. Summers and S. Drossopoulou. Considerate reasoning and the composite design pattern. In *VMCAI*, 2010.
40. K. Zee, V. Kuncak, and M. C. Rinard. Full functional verification of linked data structures. In *PLDI*, pages 349–361, 2008.

Appendix

$$\begin{array}{c}
\langle x := F, \sigma, \mu \rangle \mapsto \langle \text{skip}, \text{Update}(\sigma, x, \sigma(F)), \mu \rangle \qquad \frac{\sigma(x) = \text{null}}{\langle x.f := F, \sigma, \mu \rangle \mapsto \text{fault}} \\
\\
\frac{\sigma(x) = o \quad o \neq \text{null}}{\langle x.f := F, \sigma, \mu \rangle \mapsto \langle \text{skip}, \text{Update}(\sigma, o.f, \sigma(F)), \mu \rangle} \\
\\
\frac{o \in \text{Fresh}(\sigma) \quad \text{Fields}(K) = \bar{f} : \bar{T} \quad \text{Default}(\bar{T}) = \bar{v} \quad \sigma' = \text{New}(\sigma, o, K, \bar{v})}{\langle x := \text{new } K, \sigma, \mu \rangle \mapsto \langle \text{skip}, \text{Update}(\sigma', x, o), \mu \rangle} \\
\\
\langle (\text{skip}; C), \sigma, \mu \rangle \mapsto \langle C, \sigma, \mu \rangle \qquad \frac{\langle C_0, \sigma, \mu \rangle \mapsto \langle C'_0, \sigma', \mu' \rangle}{\langle (C_0; C_1), \sigma, \mu \rangle \mapsto \langle (C'_0; C_1), \sigma', \mu' \rangle} \\
\\
\frac{\sigma(E) \neq 0}{\langle \text{if } E \text{ then } C_0 \text{ else } C_1, \sigma, \mu \rangle \mapsto \langle C_0, \sigma, \mu \rangle} \\
\\
\frac{\sigma(E) = 0}{\langle \text{if } E \text{ then } C_0 \text{ else } C_1, \sigma, \mu \rangle \mapsto \langle C_1, \sigma, \mu \rangle} \qquad \frac{\sigma(E) = 0}{\langle \text{while } E \text{ do } C, \sigma, \mu \rangle \mapsto \langle \text{skip}, \sigma, \mu \rangle} \\
\\
\frac{\sigma(E) \neq 0}{\langle \text{while } E \text{ do } C, \sigma, \mu \rangle \mapsto \langle C; \text{while } E \text{ do } C, \sigma, \mu \rangle} \\
\\
\langle \text{end}(x), \sigma, \mu \rangle \mapsto \langle \text{skip}, \text{Retract}(\sigma, x), \mu \rangle \qquad \langle \text{end}(m), \sigma, \mu \rangle \mapsto \langle \text{skip}, \sigma, \text{Retract}(\mu, m) \rangle
\end{array}$$

Fig. 10. The transition relation \mapsto^{Δ} , cases omitted from Fig. 7.

Fig. 10 gives the transition semantics in full. It uses the following: Assuming that o is fresh in σ (i.e. $o \notin \sigma(\text{alloc})$), we write $\text{New}(\sigma, o, K, \bar{v})$ for the state that extends σ by mapping o to a K -record with field values \bar{v} and type K .

For ease of reference we re-state two properties mentioned in the text. The following is proved in [2] where the definition of \star is given.

Lemma 1. Consider any effect sets $\bar{\delta}$ and $\bar{\varepsilon}$. Suppose $\sigma \rightsquigarrow \sigma' \models \bar{\varepsilon}$ and $\sigma \models \bar{\delta} \star \bar{\varepsilon}$. Then $\text{Agree}(\sigma, \sigma', \bar{\delta})$.

The following is a consequence of a slightly stronger result in [2].

Lemma 2. For any states σ, σ' , any predicates P, P' , and any set of effects $\bar{\delta}$: Suppose $P \vdash \bar{\delta}$ frames P' and $\sigma \models P$ and $\text{Agree}(\sigma, \sigma', \bar{\delta})$. Then $\sigma \models P'$ implies $\sigma' \models P'$.

Lemma 3. For all $C, C', \sigma, \sigma', \mu, \mu', \Delta, \Delta'$ such that $\text{Active}(C)$ is not a call to a procedure in Δ or Δ' , we have: $\langle C, \sigma, \mu \rangle \xrightarrow{\Delta} \langle C', \sigma', \mu' \rangle$ if and only if $\langle C, \sigma, \mu \rangle \xrightarrow{\Delta'} \langle C', \sigma', \mu' \rangle$.

Proof sketch: When $\text{Active}(C)$ is not a method call and also when $\text{Active}(C)$ is a call to a method $m \in \text{Dom}(\mu)$, the semantics does not depend on Δ or Δ' .

We write $\Delta\langle\bar{\delta}\rangle; \Theta\langle\bar{\theta}\rangle \models^{\Gamma} \{P\} C \{P'\} [\bar{\varepsilon}]$ to express that the judgement is valid as per Def. 1.

Lemma 4 (rule soundness). Every rule derives valid conclusions from valid antecedents (if it has antecedents), when its side conditions hold.

The proof goes by cases on the axioms and rules.

Case FRAME. Suppose $\Delta\langle\bar{\delta}\rangle; \Theta\langle\bar{\theta}\rangle \models \{P\} C \{P'\} [\bar{\varepsilon}]$ and moreover the side conditions $P \vdash \bar{\delta}$ frames Q and $P \Rightarrow \bar{\delta} \star \bar{\varepsilon}$ hold. To prove $\Delta\langle\bar{\delta}\rangle; \Theta\langle\bar{\theta}\rangle \models \{P \wedge Q\} C \{P' \wedge Q\} [\bar{\varepsilon}]$ we just need to show that Q is a valid postcondition. (The other requirements—non faulting, write effect allowed, and respect dynamic boundary—are given by the premiss.) So consider any state σ such that $\sigma \models P \wedge Q$ and suppose σ' is a final state of C from σ . By the antecedent we have $\sigma \rightsquigarrow \sigma' \models \bar{\varepsilon}$. So by side condition $P \Rightarrow \bar{\delta} \star \bar{\varepsilon}$ and Lemma 1 we have $\text{Agree}(\sigma, \sigma', \bar{\delta})$. Thus by side condition $P \vdash \bar{\delta}$ frames Q and Lemma 2 we get $\sigma' \models Q$.

Case SOF. Assume the antecedent is valid, $\Delta\langle\bar{\delta}\rangle; \Theta\langle\bar{\theta}\rangle \models^{\Gamma} \{P\} C \{P'\} [\bar{\varepsilon}]$. To prove validity of the conclusion, i.e.

$$\Delta\langle\bar{\delta}\rangle; (\Theta \circledast I)\langle\bar{\theta}\rangle \models \{P \wedge I\} C \{P' \wedge I\} [\bar{\varepsilon}] \quad (4)$$

let Δ_c be the hypotheses $(\Delta, \Theta \circledast I)$, 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 procedure 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 $\xrightarrow{\Delta_c}$. Then (a) that sequence of configurations is also a computation of $\xrightarrow{\Delta_a}$ and moreover (b) I holds in every configuration.

Claim A implies (4) as follows. It is not the case that $\langle C_0, \sigma_0, \mu_0 \rangle \xrightarrow{\Delta_c}^*$ fault, because that would imply $\langle C_0, \sigma_0, \mu_0 \rangle \xrightarrow{\Delta_a}^*$ fault by (a), and this contradicts validity of the antecedent. Furthermore, if the computation terminates in n steps then by validity of the antecedent we get $\sigma_n \models P'$ and $\sigma_0 \rightsquigarrow \sigma_n \models \bar{\varepsilon}$. We get $\sigma_n \models I$ from (b). What

remains is to show that for all $0 < i$, either $\text{Active}(C_{i-1})$ is a call to some procedure m in $(\Theta \otimes I)$ or else $\text{Agree}(\sigma_{i-1}, \sigma_i, \bar{\theta})$, and *mutatis mutandis* for Δ and $\bar{\delta}$. 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 $\langle C_0, \sigma_0, \mu_0 \rangle \xrightarrow{\Delta_c}^* \langle C_n, \sigma_n, \mu_n \rangle$. The induction step is a direct consequence of **Claim B**: For any $D, D', \sigma, \sigma', \mu, \mu'$, suppose that $\langle D, \sigma, \mu \rangle$ is reachable under $\xrightarrow{\Delta_a}$ from an initial configuration $\langle C_0, \mu_0, \sigma_0 \rangle$. Suppose $\sigma \models I$ and $\langle D, \sigma, \mu \rangle \xrightarrow{\Delta_c} \langle D', \sigma', \mu' \rangle$. Then $\langle D, \sigma, \mu \rangle \xrightarrow{\Delta_a} \langle D', \sigma', \mu' \rangle$ and $\sigma' \models I$.

To prove Claim B there are 3 cases: **(a)** If $\text{Active}(D)$ is not a call to a context procedure (noting that Δ_a and Δ_c declare the same procedures) then by Lemma 3, $\langle D, \sigma, \mu \rangle \xrightarrow{\Delta_a} \langle D', \sigma', \mu' \rangle$. By validity of the correctness judgement in the antecedent of SOF we have $\text{Agree}(\sigma, \sigma', \bar{\theta})$, and we also have the rule's side condition $I \vdash \bar{\theta}$ frames I . So from $\sigma \models I$ we get $\sigma' \models I$ by Lemma 2.

For the second and third cases, i.e. **(b)** and **(c)** to follow, suppose that $\text{Active}(D)$ is a call $m(z)$ to a procedure m with specification $\{V\}m(x:T)\{V'\}[\bar{\eta}]$.

Case **(b)** $m \in \Delta$: It must be that $\sigma \models V_z^x$. For, if not, then by context call semantics (Fig. 7) we would have $\langle D, \sigma, \mu \rangle \xrightarrow{\Delta_a} \text{fault}$ —and because $\langle D, \sigma, \mu \rangle$ is reachable that would contradict the antecedent $\Delta(\bar{\delta}); \Theta(\bar{\theta}) \models^{\Gamma} \{P\} C \{P'\} [\bar{\epsilon}]$. So by the third rule in Fig. 7 for both Δ_a and Δ_c we get $\langle D, \sigma, \mu \rangle \xrightarrow{\Delta_a} \langle D', \sigma', \mu' \rangle$. By validity of the antecedent correctness judgement we have $\text{Agree}(\sigma, \sigma', \bar{\theta})$; so from $\sigma \models I$ and side condition $I \vdash \bar{\theta}$ frames I we get $\sigma' \models I$ by Lemma 2.

Case **(c)** $m \in \Theta \otimes I$. Then V is $Q \otimes I$ and V' is $Q' \otimes I$ for some Q, Q' such that $\{Q\}m(x:T)\{Q'\}[\bar{\eta}]$ is in Θ . It must be that $\sigma \models Q_z^x$. For, if not, then by procedure call semantics (Fig. 7) we would have $\langle D, \sigma, \mu \rangle \xrightarrow{\Delta_a} \text{fault}$, contradicting $\Delta(\bar{\delta}); \Theta(\bar{\theta}) \models^{\Gamma} \{P\} C \{P'\} [\bar{\epsilon}]$. Observe that I_z^x is I because x is local to the specification of m . So $\sigma \models (Q \wedge I)_z^x$ iff $\sigma \models Q_z^x$ and $\sigma \models I$. Having established both conjuncts of the precondition for m , we appeal to the specification of m in $\Theta \otimes I$ to get $\sigma' \models (Q' \wedge I)_z^x$. Which implies $\sigma' \models I$.

Case CtxIntro. For CTXINTRO, suppose C is a primitive command and the side condition $P \Rightarrow \bar{\theta} \star \bar{\epsilon}$ holds. Suppose the antecedent is valid: $\Delta(\bar{\delta}) \models \{P\} C \{P'\} [\bar{\epsilon}]$. To show validity of the conclusion, $\Delta(\bar{\delta}); \Theta(\bar{\theta}) \models \{P\} C \{P'\} [\bar{\epsilon}]$, consider any μ and any σ with $\sigma \models P$. Let Δ' be Δ, Θ . 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 procedure (which must be in Δ), the possible transitions have the form $\langle C, \sigma, \mu \rangle \xrightarrow{\Delta'} \langle \text{skip}, \sigma', \mu' \rangle$ and $\langle C, \sigma, \mu \rangle \xrightarrow{\Delta'} \text{fault}$. By Lemma 3 and well-formedness of C in Δ , these are also transitions for Δ . So by the antecedent the fault case does not happen. And by the antecedent we have $\sigma' \models P'$. To show $\text{Agree}(\sigma, \sigma', \bar{\theta})$, we can use Lemma 1 owing to side condition $P \Rightarrow \bar{\theta} \star \bar{\epsilon}$. The fact that $\bar{\delta}$ is respected follows from the antecedent.

Case LINK. The point of SOF is to disentangle Hoare's mismatch; rule LINK involves matching specifications. However, the interpretation of a specification in a hypothesized

procedure spec is different from the interpretation of the procedure body, so this rule isn't trivial.

We sketch the proof for the special case where the linked procedure is not recursive. To handle the general case, the argument is similar but relies on a secondary induction on calling depth in order to establish the correctness claim for the linked procedure.

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

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

suppose $\sigma_0 \models P$ and μ_0 be any environment. Let C_0 be let $m(x:T)$ be B in C . We must show that (i) it is not the case that $\langle C_0, \sigma_0, \mu_0 \rangle \xrightarrow{\Delta}^* \text{fault}$; and (ii) If $\langle C_0, \sigma_0, \mu_0 \rangle \xrightarrow{\Delta}^* \langle \text{skip}, \sigma_n, \mu_n \rangle$ then $\sigma_n \models P'$ and $\sigma_0 \rightsquigarrow \sigma_n \models \bar{\varepsilon}$, and (iii) Agree($\sigma_{i-1}, \sigma_i, \bar{\delta}$) for all steps $\sigma_{i-1} \xrightarrow{\Delta'} \sigma_i$ ($i > 0$). Validity of the antecedents tells us about $\xrightarrow{\Delta'}$, where Δ' is Δ, Θ . We must therefore relate $\xrightarrow{\Delta'}$ to $\xrightarrow{\Delta}$. By semantics, the first step is $\langle C_0, \sigma_0, \mu_0 \rangle \xrightarrow{\Delta'} \langle C_1, \sigma_1, \mu_1 \rangle$ 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 $\bar{\delta}$ because the state is unchanged. Subsequent steps are matched exactly by steps via $\xrightarrow{\Delta'}$ —which respect $\bar{\delta}$ 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 and z . (Without loss of generality, since we can always add skips.) In this case, $\xrightarrow{\Delta'}$ does not fault (as otherwise the antecedent for C would not be valid). Thus by definition of $\xrightarrow{\Delta'}$ it must be that $\sigma_i \models Q$ (as otherwise $m(z)$ could fault). Instead, it goes in one step to $\langle \text{skip}; D, \sigma', \mu_i \rangle$ for all σ' such that $\sigma' \models Q'$ and $\sigma_i \rightsquigarrow \sigma' \models \bar{\eta}$. The next step goes to $\langle D, \sigma', \mu_i \rangle$, which we will show is matched.

As for $\xrightarrow{\Delta}$, from $\langle C_0, \sigma_0, \mu_0 \rangle$ it steps first to $\langle (B'; \text{end}(x'); D), \sigma'', \mu_i \rangle$ 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 $\langle (\text{skip}; \text{end}(x'); D), \sigma', \mu' \rangle$, and moreover here $\sigma' \models P'$ and $\sigma_i \rightsquigarrow \sigma' \models \bar{\eta}$. Also, by hypothesis for B , $\bar{\delta}$ is respected at each step. After a couple more steps we reach $\langle D, \sigma'', \mu' \rangle$ where $\sigma'' = \text{Retract}(x', \sigma')$. This is one of the possible configurations $\langle D, \sigma', \mu_i \rangle$ reached by $\xrightarrow{\Delta'}$ so we are at a matching point in the computations and can proceed by induction on the rest of the computation of $\xrightarrow{\Delta}$.

Note that no use is made of $\bar{\theta}$.

Case CALL. For clarity we consider the special case where the argument and parameter are the same, i.e. $x \equiv z$ in the rule. So we are given the antecedents: Θ contains $\{P\}m(x:T)\{P'\}[\bar{\varepsilon}]$ and $P \Rightarrow \bar{\delta} * \bar{\varepsilon}$ is valid. We must prove $\Delta(\bar{\delta}); \Theta(\bar{\theta}) \models \{Q\} m(z) \{Q'\} [\bar{\varepsilon}]$, to which end suppose $\sigma \models P$. Let $\Delta' = \Delta, \Theta$. By semantics, we have $\langle m(x), \sigma, \mu \rangle \xrightarrow{\Delta'}^* \langle \text{skip}, \sigma', \mu \rangle$ for all σ' such that $\sigma \rightsquigarrow \sigma' \models \bar{\varepsilon}$ and $\sigma' \models P'$. As m is exempt from the obligations on θ , it remains to show Agree($\sigma, \sigma', \bar{\delta}$) and this follows by Lemma 1 using $P \Rightarrow \bar{\delta} * \bar{\varepsilon}$.