Local Reasoning for Global Invariants, Part II: Dynamic Boundaries

ANINDYA BANERJEE, IMDEA Software Institute
DAVID A. NAUMANN, Stevens Institute of Technology

Dedicated to the memory of John C. Reynolds (1935–2013).

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. The restrictions amount to a stateful frame condition that must be satisfied by any client; this dynamic encapsulation boundary complements conventional scope-based encapsulation. The technical development is based on a companion article, Part I, that presents Region Logic—a programming logic with stateful frame conditions for commands.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Class invariants; correctness proofs; formal methods; programming by contract; object orientation*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Classes and objects; modules; packages*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Assertions; invariants; logics of programs; specification techniques*

General Terms: Verification, Languages

Additional Key Words and Phrases: Modularity, data abstraction, data invariants, information hiding, heap separation, resource protection

ACM Reference Format:
Banerjee, A. and Naumann, D. A. 2013. Local reasoning for global invariants, Part II: Dynamic boundaries. *J. ACM* 60, 3, Article 19 (June 2013), 73 pages.
DOI: http://dx.doi.org/10.1145/2485981

1. INTRODUCTION

From the simplest collection class to the most complex application framework, software modules provide useful abstractions by hiding the complexity of efficient

---

This is an expanded and revised version of a paper originally appearing in the *European Symposium on Programming*, 2010.

A. Banerjee was partially supported by Madrid Regional Government Project S2009TIC-1465 Prometidos; MINECO Project TIN2009-14599-C03-02 Desafios; EU NoE Project 256980 Nessos; US NSF grants CNS-0627748 and ITR-0326577 and by a sabbatical visit at Microsoft Research, Redmond. D. A. Naumann was supported in part by US NSF grants CNS-0627338, CRI-0708330, CCF-0429894, CCF-0915611; by a sabbatical visit at Microsoft Research, Cambridge, and by a visiting professorship at IMDEA Software Institute.

Authors' addresses: A. Banerjee, IMDEA Software Institute, Edificio IMDEA Software, Campus Montegancedo s/n, 28223 Pozuelo de Alarcón, Madrid, Spain; email: anindya.banerjee@imdea.org; D. A. Naumann, Stevens Institute of Technology, Castle Point on Hudson, Hoboken, NJ 07030-5991; email: naumann@cs.stevens.edu.

DOI: http://dx.doi.org/10.1145/2485981

implementations. Many abstractions and most representations involve state, so the information to be hidden includes invariants on internal data structures. Hoare [1972] described the hiding of data invariants as what amounts to 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 implementations assume the invariant and are obliged to maintain it. 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. [2001] aptly called a "mismatch between the simple intuitions about the way pointer operations work and the complexity of their axiomatic treatments". 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 nonregular data structures, callbacks, and the deliberate use of shared references that cross encapsulation boundaries;
— the need for effective, modular reasoning on both sides of an interface: for client code and for the module implementation;
— the hope of achieving high automation through mature techniques including types and static analyses as well as theorem proving; and
— the need to encompass language features such as parametric polymorphism and code pointers for which semantics is difficult.

This article 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 a key notion: 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, dubbed Region Logic, for object-based sequential programs. Our approach is based on correctness judgments 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—for example, Harel et al. [1977] study a logic with hypothetical judgments and even a "frame axiom".[1] But we build directly on their novel combination in the second order frame rule of separation logic [O'Hearn et al. 2009].

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 [Kroening and Strichman 2008]. For the most part, region logic is standard Hoare logic, with classical first order assertions; it borrows from separation logic atomic commands for heap read/write and the atomic points-to predicate. The

[1]Their name for the Invariance axiom of Hoare logic [Apt et al. 2009]. Another name is Constancy [Reynolds 1981, 1998].

small but crucial novelty is a form of stateful frame condition inspired by the work of Kassios [2011]. This provides local reasoning, that is, a focus on just the locations relevant to a command's behavior—its footprint. Basic region logic, including a first order frame rule, is developed in Part I of this paper [Banerjee et al. 2013].

The notion of dynamic boundary involves a non-standard proof obligation for verifying a command under hypotheses about the procedures it calls: its state updates must respect the dynamic boundaries of the modules containing those procedures. This notion provides 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.

Formally, the payoff from dynamic boundaries is the validation of a second-order frame rule. Various standard rules of Hoare logic can be retained with little change beyond the frame conditions added in Part I, but their soundness needs to be proved with respect to the new proof obligation.

Used directly, our logic requires programs to be instrumented with ghost code to track footprints. This verbosity is the price paid for using standard first order assertions—without recursive predicates—and for not baking in a particular methodology. However, the simplicity of the approach makes it amenable to richer assertion languages and to syntactic sugars, supported by static analyses, for established idioms like ownership types (e.g., Müller and Rudich [2007]). Region logic could serve as translation target for lighter weight but more restrictive notations used for some modules, while other modules are verified using more specialized specifications.

Contributions. The main contribution of Part II is the notion of dynamic boundary: its semantics and its proof rules, in particular the second order frame rule. From this rule we derive a "mismatch rule", inspired by O'Hearn et al. [2009], that embodies Hoare's mismatch. In addition, we derive a novel variant that provides for reentrant callbacks across module boundaries.

The main technical result is soundness of a full set of proof rules. These include basic rules from Part I, which are extended with hypotheses and shown to be sound for the extended notion of correctness including dynamic boundaries. Unlike many presentations of Hoare logics, we do not assume a fixed collection of procedure declarations, but rather include a letrec construct for explicit linking. Soundness of a Hoare logic is usually proved with respect to a denotational semantics, which may be derived from a transition (small-step) semantics. Although the purpose of the logic is to prove pre-post properties, our notion of correctness with respect to dynamic boundaries is defined in terms of transition semantics, so our soundness proof deals directly with small steps.[2] The linking rule is unremarkable, except that proving its soundness in transition semantics is surprisingly difficult, even aside from dynamic boundaries.

For interesting functional specifications, specification-only variables are essential—at least the special case of postconditions with "old" expressions to refer to initial values. For completeness, logics with procedures include substitution or adaptation rules that cater for manipulation of specification-only variables, especially for verification of recursive procedures. We provide a substitution rule and proper treatment of specification-only variables, which is tricky in a logic with hypothetical judgments (or

[2]One might hope to avoid transition semantics, perhaps using some form of denotational model and Kripke semantics, but so far the logic resisted attempts by ourselves and others to find such a model.

even one where procedure specifications are tied to procedure declarations [Apt et al. 2009]).

Dynamic boundaries provide an imperative notion of module, complementary to the functional notions—scoping and parameterization mechanisms—provided by conventional module systems.[3] Following O'Hearn et al. [2009] we refrain from formalizing a full-fledged module system.

Outline. Section 2 surveys the challenges and tensions listed above, using illustrative examples. Section 3 briefly reviews Part I of the paper [Banerjee et al. 2013], on which we rely. Our aim is for Part II to be understandable on its own, at least at a high level, provided one trusts Part I for results and fine points of some formulations. Section 4 adds recursive procedures to the programming language. Correctness judgments are extended with hypotheses (procedure specifications) and proof rules are provided for procedure call and for discharge of hypotheses by procedure linking. A proof of the linking rule is sketched, as the induction hypotheses involved are intricate even without dynamic boundaries. Section 5 develops some technical results, motivated by that proof sketch, that are only used to prove the linking rule (in Section 7.6). Section 6 formalizes dynamic boundaries and their semantics. It formalizes a simple notion of module, comprised of a dynamic boundary associated with a group of procedure specifications. The semantics of correctness judgments is extended to account for dynamic boundaries. The mismatch rule is presented and is shown to be derivable from the ordinary procedure linking rule together with the rule of second order framing. Section 7 extends the proof rules from Section 4 and from Part I of this article to account for dynamic boundaries. Soundness is proved in detail. Section 8 shows some derived rules and applies the logic to the examples in Section 2. Section 9 introduces a mismatch rule for the case of two modules that inter-depend by way of reentrant callbacks; the rule is derived from the second-order frame rule. Section 10 discusses related work and Section 11 concludes.[4]

2. THE CHALLENGE OF HIDING INVARIANTS ON SHARED OBJECTS

2.1. A Collection Implemented by a List

We begin with a textbook example of encapsulation and information hiding, the toy program in Figure 1.[5] Annotations include method postconditions that refer to a global variable, *pool*, marked as ghost state. Ghost variables and fields are auxiliary mutable state used in reasoning, but not mentioned in branch conditions or in expressions

[3]These are essentially functional, even though procedures in languages like ML and Java use imperative code.

[4]There are three major additions compared with the conference version of this article [Naumann and Banerjee 2010]: detailed soundness proofs including the linking rule for recursive procedures and the substitution rule; a new mismatch rule for reentrant callbacks that cross module boundaries; and detailed proofs of examples. The syntax and semantics of dynamic boundaries is revised a little, to avoid an incompleteness in some cases of nested modules. These changes led to numerous revisions.

The published version of Naumann and Banerjee [2010] has an unsound side condition in the second-order frame rule; the fix was announced during the conference presentation.

[5]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; for example, the assignment $lst := \mathsf{null}$ in the body of the *Set* constructor is short for $\mathsf{self}.lst := \mathsf{null}$. Variable result is the returned result; there is no explicit "return" statement.

```
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; }
```

Fig. 1. Module *SET*, together with library class *Node*. (In our formal treatment, the module consists of global variable *pool* and the methods of class *Set*.)

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. (See Owicki and Gries [1976] and Reynolds [1981], but note that Reynolds uses "ghost" for what we call specification-only variables.)

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 in Figure 1 are expressed in terms of an integer set, *elements*, which is defined rather than assigned (a "model field" in JML terminology [Leavens et al. 2003; Leino and Müller 2006]). Abstraction of this sort is commonplace and plays a role in Hoare's paper [1972], 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. A proof of well-foundedness of the recursive definition is possible but left out for simplicity.

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 which will be applied to nonnull references of type *Set*:

$$SetI(s : Set) \;\widehat{=}\; (\forall n, m : Node \in s.rep \cdot n = m \vee n.val \neq m.val) \\ \wedge\; s.lst \in s.rep \;\wedge\; s.rep`nxt \subseteq s.rep \;\wedge\; s.rep`own \subseteq \{s\}.$$

The first conjunct says there are no duplicate values among elements of $s.rep$. The next says that $s.rep$ contains the head, $s.lst$ (which may be null). The inclusion $s.rep`nxt \subseteq s.rep$ says that $s.rep$ is nxt-closed;[6] this is equivalent to the following:[7]

$$\forall o : Node \in \mathsf{alloc} \cdot o \in s.rep \Rightarrow o.nxt \in s.rep.$$

The special variable alloc is always the set of all currently allocated references. One can show by induction that these conditions imply there are no duplicates in the list. So the invariant says what we want, though not itself using any inductive predicates. 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. This expresses that s "owns" every node in $s.rep$. So we adopt an invariant to be associated with module SET:

$$I_{set} \mathrel{\widehat{=}} \forall s : Set \in pool \cdot SetI(s).$$

Assuming that the constructor is called whenever an instance of Set is allocated, $pool$ will invariably hold all instances so that we could as well write $\forall s : Set \in \mathsf{alloc} \cdot SetI(s)$. We do not make that assumption in our formal development. Even with that assumption, care must be taken with quantifiers in invariants, to avoid falsification by allocation (cf. [Naumann and Barnett 2006; Pierik et al. 2005]). Though I_{set} is global in the sense that it effectively quantifies over all instances of Set, it is compatible with local reasoning. We return to this in Section 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 Figure 1 and the client is thus not responsible to establish it before the invocation of $remove$. As articulated by Hoare [1972], 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 initialization in 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 $s.remove$.

For brevity, we call this Hoare's mismatch: the specifications used in reasoning about invocations in client code, that is, 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

[6] 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. Full details are in Part I [Banerjee et al. 2013].

[7] 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. In this particular case, n is not null because quantified variables range over non-null elements of the specified range; and we do not apply $SetI$ to null. According to Remark 3.1 in Part I of the article, we should avoid writing $s.lst \in anything$, and indeed if we expand the shorthand the condition is $\{s\}`lst \subseteq anything$ which is true if s is null—in confusing contrast with the points-to predicate $s.lst = y$, which is false if s is null. We indulge in this abuse of notation only in this section, and only in cases where the relevant reference is nonnull.

$$
\begin{array}{l}
\mathsf{var}\ \mathit{flist} : \mathit{Node};\ \ \mathit{count} : \mathsf{int};\ \ \mathsf{ghost}\ \mathit{freed} : \mathsf{rgn};\\
\mathit{alloc}() : \mathit{Node}\\
\quad \mathsf{ensures\ result} \neq \mathsf{null} \wedge \mathit{freed} = \mathsf{old}(\mathit{freed}) \setminus \{\mathsf{result}\} \wedge (\mathsf{result} \in \mathsf{old}(\mathit{freed}) \vee \mathsf{fresh}(\mathsf{result}))\\
\{\ \mathsf{if}\ \ \mathit{count} = 0\ \mathsf{then\ result} := \mathsf{new}\ \mathit{Node};\\
\quad \mathsf{else\ result} := \mathit{flist};\ \ \mathit{flist} := \mathit{flist}.\mathit{nxt};\ \ \mathit{count} := \mathit{count} - 1;\ \ \mathit{freed} := \mathit{freed} \setminus \{\mathsf{result}\};\ \ \mathsf{endif}\}\\
\mathit{free}(n : \mathit{Node})\\
\quad \mathsf{requires}\ n \neq \mathsf{null}\ \wedge\ n \notin \mathit{freed}\quad \mathsf{ensures}\ \mathit{freed} = \mathsf{old}(\mathit{freed}) \cup \{n\}\\
\{\ n.\mathit{nxt} := \mathit{flist};\ \ \mathit{flist} := n;\ \ \mathit{count} := \mathit{count} + 1;\ \ \mathit{freed} := \mathit{freed} \cup \{n\};\ \}
\end{array}
$$

Fig. 2. Module MM, comprised of global variables freed, flist, count, and procedures alloc and free.

the method's implementation; so the assumption is justified by the semantics of assert and assume. Hoare's mismatch asserts the public precondition at call sites, but assumes for the implementation an added conjunct, the invariant.

The mismatch is unsound if encapsulation is faulty, which can easily happen due to shared references, for example, if in place of $n.\mathit{val} := 1$ the client code had $s.\mathit{lst}.\mathit{val} := 1$. To some extent, encapsulation is provided by lexical scope of state components—for example, field lst—and types. However, scope does not prevent that references can be leaked to clients, for example, 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 (as allowed by "private" scope as in Java). For the sake of focus, we gloss over scope in the examples and formalize only rudimentary scoping constructs.

Besides scope and typing, a popular technique to deal with encapsulation in the presence of pointers is ownership (e.g., Dietl and Müller [2005] and Drossopoulou et al. [2008]). Ownership systems restrict the form of invariants and the use of references, to support modular reasoning at the granularity of a single instance and the internal representation that it "owns" and on which its invariant may depend. Ownership in this sense 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. Figure 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 Figure 1) are treated as a resource. The instances currently "owned" by the module are tracked using variable freed. The hidden invariant in this example is recursively defined as follows.

$$I_{mm} \ \widehat{=}\ \ \mathit{FC}(\mathit{flist}, \mathit{freed}, \mathit{count})$$

where $\mathit{FC}(f : \mathit{Node}, r : \mathsf{rgn}, c : \mathsf{int})$ is defined, by induction on the size of r, as

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

The invariant says freed is the set of nodes reached from flist and count is the size. The implementation of alloc relies on accuracy of count. It relies directly on the condition $\mathit{count} \neq 0 \Rightarrow \mathit{flist} \neq \mathsf{null}$, 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.

$$
\begin{array}{l}
\mathsf{var}\ x, y : \mathit{Node};\ \ x := \mathsf{new}\ \mathit{Node};\ \ y := \mathit{alloc}();\ \ \mathit{free}(x);\ \ \mathit{free}(y);\\
\mathsf{while}\ y \neq \mathsf{null}\ \mathsf{do}\ y.\mathit{val} := 7;\ \ y := y.\mathit{nxt};\ \ \mathsf{od}
\end{array}
\qquad (2)
$$

```
ghost sopool : rgn;
class Subject {
  obs : Observer; val : int; ghost O : rgn;
  Subject() { obs := null; val := 0; O := ∅; sopool := sopool ∪ {self}; }
  update(n : int)
  { 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) { sub := s; s.register(self); sopool := sopool ∪ {self}; }
  notify() { cache := sub.get(); } }
```

Fig. 3. Module SO. Public methods are $Subject$, $update$, get and $Observer$.

The loop updates val fields of freed objects, but it does not write the nxt fields, on which the invariant depends; the client neither falsifies I_{mm} nor causes a fault. Suppose, however, that we replace the loop by the assignment $y.nxt := \mathsf{null}$. This falsifies the invariant I_{mm}, if initially $count$ is sufficiently high, and then subsequent invocations of $alloc$ break.

The strange client (2) is rejected by most ownership systems due to rigid typing or specification rules. In separation logic, "ownership is in the eye of the asserter" [O'Hearn et al. 2009] and there is more flexibility. However, notional ownership is expressed using the separating conjunction, which disallows shared reads. The strange client would not be verifiable in separation logic [O'Hearn et al. 2009] except in the relatively elaborate variants that use permissions to allow shared reads [Bornat et al. 2005]. But we can identify an encapsulation boundary in the example: clients must not write the nxt field of objects in $freed$ (nor write variables $flist$ and $count$). The strange client respects this boundary.

This example may appear contrived. However, sharing of references across encapsulation boundaries is common in system code, at the C level of abstraction. Shared reads also occur in programs at the level of abstraction we consider here, where references are abstract values susceptible only to equality test and field dereference. Platforms such as J2EE provide efficient access to databases by means of connection pools. When a client closes a connection, it may retain a reference to the connection object and make harmless updates, or harmful ones, if the implementation relies on good behavior of clients rather than robust encapsulation.

We aim to facilitate reasoning based on explicit ownership invariants, as in the SET example, but in harmony with reasoning in terms of more- or less-specialized idioms as in the MM example and the following examples.

2.3. Observer Pattern: Cluster Invariants

Aside from separation logic, ownership has been developed mainly for instance-oriented invariants as in the SET example. In many situations, however, the desired granularity of local reasoning pertains to a nonhierarchical cluster of interdependent objects.

Figure 3 is a simple version of the Observer design pattern in which an observer only tracks a single subject. The code is exactly the same as in Part I except for the use of ghost region $sopool$ and its updates in the constructors $Subject$ and $Observer$. Parkinson [2007] used the example to argue against instance-oriented notions of invariant. We

Method	Precondition	Postcondition
$Subject()$	$X = sopool$ $\wedge\ sopool\text{`}(obs, O) \subseteq sopool$	$\mathsf{self}.val = 0 \wedge \mathsf{self}.O = \varnothing$ $\wedge sopool = X \cup \{\mathsf{self}\}$ $\wedge sopool\text{`}(obs, O) \subseteq sopool$
$update(n)$	$\forall o \in \mathsf{self}.O \cdot Obs(o, \mathsf{self}, \_)$	$\forall o \in \mathsf{self}.O \cdot Obs(o, \mathsf{self}, n)$
$get()$	$true$	$\mathsf{result} = val$
$register(b)$	$b \neq \mathsf{null} \wedge b \notin \mathsf{self}.O$ $\wedge\ b.sub = \mathsf{self} \wedge SubH(\mathsf{self})$ $\wedge \forall o \in \mathsf{self}.O \cdot Obs(o, \mathsf{self}, \mathsf{self}.val)$	$b \in \mathsf{self}.O \wedge SubH(\mathsf{self})$ $\wedge \forall o \in \mathsf{self}.O \cdot Obs(o, \mathsf{self}, \mathsf{self}.val)$ $\wedge\ \mathsf{self}.O\text{`}nxto \subseteq sopool$
$Observer(s)$	$\forall o \in s.O \cdot Obs(o, s, s.val)$ $\wedge\ s \neq \mathsf{null}$ $\wedge\ s \in sopool \wedge X = sopool$ $\wedge\ sopool\text{`}(sub, nxto) \subseteq sopool$	$\forall o \in s.O \cdot Obs(o, s, s.val)$ $\wedge\ \mathsf{self} \in s.O$ $\wedge\ sopool = X \cup \{\mathsf{self}\}$ $\wedge\ sopool\text{`}(sub, nxto) \subseteq sopool$
$notify()$	$true$	$\mathsf{self}.cache = \mathsf{self}.sub.val$

Fig. 4. Specifications of methods in SO. For an observer b, subject s, and integer v, the predicate $Obs(b, s, v)$ is defined as $b.sub = s \wedge b.cache = v$.

address that issue using a single invariant predicate that in effect quantifies over clusters of client-visible objects; each cluster is comprised of a subject and its observers. 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

$$I_{so} \ \widehat{=}\ (\forall s : Subject \in sopool \cdot SubH(s)) \wedge (\forall o : Observer \in sopool \cdot o.sub \neq \mathsf{null} \wedge o \in o.sub.O)$$

where $SubH(s)$ is defined as $List(s.obs, s.O) \wedge s.O \subseteq sopool$. Recall from Part I that $List(o, r)$ says the list beginning at o lies in region r (compare FC in Section 2.2). The second conjunct of I_{so} says that any observer tracking a subject lies in that subject's O region. As with I_{set}, the instantiations of I_{so} 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.

The specifications of methods in SO appear in Figure 4 and use predicate Obs defined in the figure's caption. Obs connects a subject s to observer b by tracking in b's cache the current value v of s's internal state. The specifications are the same as in Part I except that they take into account $sopool$.

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 Figure 5 consists of a class, *ObsColl*, that extends *Observer*. Instances of *ObsColl* 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 *ObsColl* is in the *nxto*-linked list of observers of its subject.

```
ghost ocpool : rgn;
class ObsColl extends Observer { next : ObsColl; prev : ObsColl;
  ObsColl(s : Subject, oc : ObsColl)
    requires ∀o ∈ s.O · Obs(o, s, s.val) ∧ s ≠ null ∧ s ∈ sopool
    requires X = sopool ∧ sopool‘(sub, nxto) ⊆ sopool ∧ Y = ocpool ∧ U(oc, ocpool)
    ensures ∀o ∈ s.O · Obs(o, s, s.val) ∧ self ∈ s.O ∧ sopool = X ∪ {self}
    ensures sopool‘(sub, nxto) ⊆ sopool ∧ ocpool = Y ∪ {self}
  { super(s);
    if oc = null then prev := self; next := self;
    else next := oc; prev := oc.prev; oc.prev.next := self; oc.prev := self;
    ocpool := ocpool ∪ {self}; } }
```

Fig. 5. Module OC. The predicate $U(x : ObsColl, g : \mathsf{rgn})$ is defined as $(x = \mathsf{null} \wedge g = \varnothing) \vee (x \neq \mathsf{null} \wedge \mathsf{null} \notin g \wedge x \in g \wedge x.next \in g \wedge x.prev \in g \wedge x \neq x.next \wedge x \neq x.prev)$.

The constructor of $ObsColl$ 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 OC is defined in the context of module SO, because $ObsColl$ is a subclass of $Observer$. The verification of the implementation of $ObsColl$ will require its module invariant I_{oc} but not I_{so}. The invariant I_{oc} expresses a simple property of cyclic doubly linked lists:

$$I_{oc} \;\widehat{=}\; \forall oc : ObsColl \in ocpool \cdot oc.prev.next = oc \wedge oc.next.prev = oc.$$

Despite the overlapping structure, there is no interference between the code and invariants of modules SO and OC because different locations are involved.

Figure 5 gives the specifications of the $ObsColl$ constructor. The precondition is the conjunction of the two requires clauses and the postcondition is the conjunction of the two ensures clauses. The precondition uses the predicate U defined in the caption of the figure. Observe that $U(x, \varnothing) \Leftrightarrow x = \mathsf{null}$ and $U(\mathsf{null}, g) \Leftrightarrow g = \varnothing$.

Interesting variations on the example include observers that track multiple subjects, and observers that are also in the role of subject (cf. Krishnaswami et al. [2010]). Of particular interest are callbacks between modules, as opposed to the $notify/get$ callback within module SO. That is the topic of Section 9.

3. REGION LOGIC REVIEW: EFFECTS AND FIRST-ORDER FRAMING

This section reviews programs and specifications from Part I [Banerjee et al. 2013], to which the reader may refer for details.

3.1. Programming Language, States, Assertions

Our formal results are given for an idealized object-based language with syntax in Figure 6. 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 $\overline{T}$ may make mutually recursive reference to other classes. Classes are nothing more than named record types. We treat type equivalence by name, though it matters little.

The syntax in Figure 6 is the same as in Part I of this article except for the addition of procedure calls, $m(x)$, and procedure blocks (letrec). In this section we review the basic semantics, leaving procedures to Section 4. In Section 6, we introduce a rudimentary form of module, consisting of little more than grouping of related procedures and explicit declaration of the import relation (e.g., OC imports SO). Class declarations

$m \in \mathit{ProcName}$ $x, y, r \in \mathit{VarName}$ $f, g \in \mathit{FieldName}$ $K \in \mathit{DeclaredClassNames}$

(Types)	T	$::= \mathsf{int} \mid \mathsf{rgn} \mid K$
(Program Expressions)	E	$::= x \mid c \mid \mathsf{null} \mid E \oplus E$ where c is in $\mathbb{Z}$ and $\oplus$ is in $\{=, +, -, *, >, \ldots\}$
(Region Expressions)	G	$::= \varnothing \mid x \mid \{E\} \mid G\text{`}f \mid G/K \mid G \otimes G$ where $\otimes$ is in $\{\cup, \cap, \setminus\}$
(Expressions)	F	$::= E \mid G$
(Commands)	C	$::= \mathsf{skip} \mid x := F \mid x := \mathsf{new}\ K \mid x := x.f \mid x.f := F \mid m(x)$
		$\mid \mathsf{if}\ E\ \mathsf{then}\ C\ \mathsf{else}\ C \mid \mathsf{while}\ E\ \mathsf{do}\ C \mid C\,;C \mid \mathsf{var}\ x : T\ \mathsf{in}\ C$
		$\mid \mathsf{letrec}\ m(x : T) = C\ \mathsf{in}\ C$

Fig. 6. Programming language. Procedure call, $m(x)$, and linking, letrec $m(x : T) = C$ in C, are additions to the language of Part I and are not considered until Section 4. The *atomic commands* are those in the first line of the production for C, including procedure call.

and global variable declarations are considered visible in all modules. The "little more" is the crux of Part II: each module has a dynamic boundary for encapsulation.

Typing rules enforce that type int is separated from reference types: there is no pointer arithmetic, but references can be tested for equality. We let Γ range over typing contexts that are *well formed*, meaning that no variable is given more than one type, variable alloc is in $dom(\Gamma)$, and $\Gamma(\mathsf{alloc}) = \mathsf{rgn}$. We write $\Gamma, x : T$ for extension of Γ with x that is not in $dom(\Gamma)$; it is not defined if x is in $dom(\Gamma)$.

Figure 5 is illustrative of the kinds of programs in which we are interested. For economy in the formal development, we do not consider subclassing, and thus we neither consider inheritance nor dynamic dispatch. Particular examples, such as the one in the figure, can come under the formal development by a simple translation. We inline the code of $\mathit{Observer}$ and replace the call super(s) by a call to $\mathit{Observer}.\mathit{Observer}(s)$. Each class must declare its own version of a method. So $\mathit{ObsColl}$ will declare its own notify method, whose body will be the same as that of $\mathit{Observer}.\mathit{notify}$. Adapting the formal development to handle inheritance and dynamic dispatch is left for future work.

The semantics is based on conventional program states. We assume given an infinite 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. The variable alloc 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, alloc holds the set of allocated references and does not contain *null*. The heap maps each allocated reference to its type (which is immutable) and its field values. The values of a class type K are *null* and allocated references of type K. We abstract from the concrete representation and 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, $[\sigma \mid x : v]$ overrides σ to map variable x to v, $\mathit{Type}(o, \sigma)$ is the type of allocated reference o, $[\sigma \mid o.f : v]$ overrides σ to map field f of object o to v (for $o \in \sigma(\mathsf{alloc})$), $\mathit{Extend}(\sigma, x, v)$ extends σ to map x to value v (for $x \notin \mathit{Dom}(\sigma)$), and $\sigma \restriction x$ removes x from the domain of the store. (The symbol $\restriction$ is intended to be mnemonic for "toss".) 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, as mentioned in Footnote 6. 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 Figure 1, then $G\text{`}f$ is the union of the values of f. We assume that field names are unique, so that o has field f just if $\mathit{Type}(o, \sigma)$ is the class that declares f.

Assertions are interpreted with respect to a single state, for example, the semantics of the atomic "points-to" predicate $x.f = E$ is defined by:

$$\sigma \models x.f = E \quad \text{iff} \quad \sigma(x) \neq \mathit{null} \text{ and } \sigma(x.f) = \sigma(E).$$

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

Fig. 7. Effect specifications for methods in Figure 1. For *contains*, the effect is empty.

$$\begin{array}{lll} \text{(Effects)} & \varepsilon & ::= \varepsilon, \varepsilon \mid (\mathit{empty}) \mid \mathsf{rd}\ x \mid \mathsf{rd}\ G\text{`}f \mid \mathsf{wr}\ x \mid \mathsf{wr}\ G\text{`}f \mid \mathsf{fr}\ G \\ \text{(Formulas)} & P & ::= E = E \mid x.f = E \mid G \subseteq G \\ & & \quad \mid (\forall x : \mathsf{int} \cdot P) \mid (\forall x : K \in G \cdot P) \mid P \wedge P \mid \neg P \end{array}$$

Fig. 8. Grammar of effects and state predicates.

The operator "old" used in specifications like those in Figure 1 can be desugared using specification-only variables. (For example, this is the role of X in Figure 4.) We do not use quantified variables of type rgn. Quantified variables of class type range over non-null, currently allocated references: $\sigma \models^{\Gamma} (\forall x : K \in G \cdot P)$ iff $\mathit{Extend}(\sigma, x, o) \models^{\Gamma, x:K} P$ for all $o \in \sigma(\mathsf{alloc}) \cap \sigma(G)$ such that $\mathit{Type}(o, \sigma) = K$. To assert that elements of G have type K one writes $G \subseteq G/K$ using the type restriction $/K$.

3.2. Effect Specifications and the Framing of Commands and Formulas

We augment the specifications in Figure 1 with the effect specifications in Figure 7. Effects are given by the grammar in Figure 8. We omit tags wr and rd in lists of effects of the same kind. In this article, 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 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 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 wr self.rep`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 wr self.$rep$`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 wr G`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 wr $x.f$ abbreviates wr $\{x\}$`f. In case x is null, this is well defined and designates the empty set of locations. One may also allow f to be a data group [Leino et al. 2002], for example, the built-in data group "any" that stands for all fields of an object.

We say σ' is *compatible with* σ, and write $\sigma \asymp \sigma'$, provided $\mathit{Type}(o, \sigma) = \mathit{Type}(o, \sigma')$ for all $o \in \sigma(\mathsf{alloc}) \cap \sigma'(\mathsf{alloc})$. We say σ' *succeeds* σ, and write $\sigma \hookrightarrow \sigma'$, provided $\sigma \asymp \sigma'$ and $\sigma(\mathsf{alloc}) \subseteq \sigma'(\mathsf{alloc})$. These relations make sense for states of any type, not necessarily the same type.

For effect ε that is well formed in Γ, and Γ'-states σ, σ' for some $\Gamma' \supseteq \Gamma$, we say ε *allows change from* σ *to* σ', written $\sigma \rightarrow \sigma' \models \varepsilon$, if and only if $\sigma \hookrightarrow \sigma'$ and

(a) for every y in $dom(\Gamma')$, either $\sigma(y) = \sigma'(y)$ or wr y is in ε
(b) for every o in $\sigma(\mathsf{alloc})$ and every f in $\mathit{Fields}(\mathit{Type}(o, \sigma))$, either $\sigma(o.f) = \sigma'(o.f)$ or there is G such that wr G`f is in ε and o is in $\sigma(G)$
(c) for each fr G in ε, we have $\sigma'(G) \subseteq \sigma'(\mathsf{alloc}) \setminus \sigma(\mathsf{alloc})$.

The need to consider Γ' different from Γ should become clear in Section 4.2.

We extend Hoare triples with frame conditions in brackets, so that $\{\,P\,\}\ C\ \{\,P'\,\}\ [\varepsilon]$ says that $\sigma{\rightarrow}\sigma' \models \varepsilon$ as well as $\sigma' \models P'$, for any P-state σ from which C can yield σ'.

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 encapsulation that will allow I_{set} to be hidden from clients. First, we frame the object invariant $SetI(s)$ (Section 2.1), which will be used for "local reasoning" at the granularity of a single instance of Set. We choose to frame[8] it by

$$\delta_0 \mathrel{\widehat{=}} \mathsf{rd}\ s,\ s.(rep, lst),\ s.rep\text{`}(nxt, val, own)$$

(abbreviating $s.rep$, $s.lst$, $s.rep\text{`}nxt$, $s.rep\text{`}val$, and $s.rep\text{`}own$). A read effect designates l-values. Here, δ_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 judgment for framing of formulas, for example, $true \vdash \delta_0\ \mathsf{frm}\ SetI(s)$ says that $SetI(s)$ depends at most on the locations designated by δ_0. The judgment involves a formula, here $true$, because framing by state-dependent effects may hold only under some conditions on that state. For example, we have

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

The semantics of judgment $P \vdash \delta\ \mathsf{frm}\ P'$ is specified as follows. We write $Agree(\sigma, \tau, \delta)$ to say σ agrees with τ on locations designated by δ. More precisely (see Definition 5.2 in Part I): Let δ be an effect that is well-formed in Γ. Let $\Gamma' \supseteq \Gamma$ and $\Gamma'' \supseteq \Gamma$. Let σ be a Γ'-state and τ a Γ''-state. We say that σ and τ ***agree on*** δ, written $Agree(\sigma, \tau, \delta)$, provided that $\sigma \asymp \tau$ and moreover the following hold.

(a) For all $\mathsf{rd}\, x$ in δ, we have $\sigma(x) = \tau(x)$.
(b) For all $\mathsf{rd}\, G\text{`}f$ in δ and all $o \in \sigma(G) \cap \tau(\mathsf{alloc})$ with $Type(o, \sigma) = DeclClass(f)$, we have $\sigma(o.f) = \tau(o.f)$.

In Section 6, we instantiate this definition with intermediate states that may extend Γ with additional local variables and procedure parameters. That is why we need Γ', Γ''.

A framing judgment $P \vdash^{\Gamma} \delta\ \mathsf{frm}\ P'$ is called *valid* written $P \models^{\Gamma} \delta\ \mathsf{frm}\ P'$, iff for all Γ-states σ, σ',

$$\text{if } Agree(\sigma, \sigma', \delta) \text{ and } \sigma \models^{\Gamma} P \wedge P' \text{ then } \sigma' \models^{\Gamma} P'. \tag{3}$$

(See Definition 6.1 in Part I.) There are two ways to establish a framing judgment. One is to directly check its validity. That can be automated using an SMT prover provided the heap model admits quantification over field names, to express agreement. The validity condition is in the $\forall$ fragment and automation worked well in our experiments [Rosenberg et al. 2010]. The other way is to use inference rules for the framing judgment. These include syntax-directed rules together with first-order provability and effect subsumption. As an example, the rule for framing a quantified formula, say a judgment $P \vdash \eta\ \mathsf{frm}\ (\forall x : K \cdot x \in G \Rightarrow P')$, has a premise of the form $P \wedge x \in G \vdash \eta'\ \mathsf{frm}\ P'$ and requires η to subsume the footprint of G. (See Part I for these rules.) For I_{set}, we can use the specific judgments above to derive $true \vdash \delta_{set}\ \mathsf{frm}\ I_{set}$, where δ_{set} is $\mathsf{rd}\ pool,\ pool\text{`}(rep, lst),\ pool\text{`}rep\text{`}(nxt, val, own)$. This is subsumed by θ_{set}, where

$$\theta_{set} \mathrel{\widehat{=}} \mathsf{rd}\ pool,\ pool\text{`}\mathsf{any},\ pool\text{`}rep\text{`}\mathsf{any}.$$

[8]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" by itself as a noun.

A Frame Rule. To verify the implementations in Figure 1 we would like to reason in terms of a single instance of Set. Let B_{add} be the body of method add. We can verify that B_{add} satisfies the frame conditions $\mathsf{wr}\,\mathsf{alloc}, \mathsf{self.any}$ and thus those for add in Figure 7. Moreover, we can verify the following.

$$\{\, SetI(\mathsf{self}) \,\}\; B_{add}\; \{\, SetI(\mathsf{self}) \wedge elements = \mathsf{old}(elements) \cup \{i\} \,\}\; [\mathsf{alloc}, \mathsf{self.any}] \qquad (4)$$

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 separation property:

$$I_{set} \Rightarrow (\forall s, t : Set \in pool \cdot s = t \vee s.rep \,\#\, t.rep) \qquad (5)$$

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. To be precise, $G \,\#\, G'$ abbreviates $G \cap G' \subseteq \{\mathsf{null}\}$. Now I_{set} is logically equivalent to $SetI(\mathsf{self}) \wedge Iexcept$, with δ_x framing $Iexcept$, defined as

$$Iexcept \;\widehat{=}\; \forall s : Set \in pool \setminus \{\mathsf{self}\} \cdot SetI(s)$$
$$\delta_x \;\widehat{=}\; \mathsf{rd}\ \mathsf{self},\ pool,\ (pool \setminus \{\mathsf{self}\})^{\boldsymbol{\backprime}}(rep, lst),\ (pool \setminus \{\mathsf{self}\})^{\boldsymbol{\backprime}} rep^{\boldsymbol{\backprime}}(nxt, val, own)$$

We aim to conjoin $Iexcept$ to the pre- and postconditions of (4). To make this precise, we use a syntax-directed operator $\cdot/.$ on effects. Given read effects δ and write effects ε, the *separator formula* given by $\delta \cdot/. \varepsilon$ is a conjunction of disjointness formulas, describing states in which writes allowed by ε cannot affect the value of a formula with footprint δ. The formula $\delta \cdot/. \varepsilon$ is defined by induction on the syntax of effects. For example, $\mathsf{rd}\, x \cdot/. \mathsf{wr}\, y$ is true or false according to whether x and y are the same variable, and $\mathsf{rd}\, G^{\boldsymbol{\backprime}}f \cdot/. \mathsf{wr}\, H^{\boldsymbol{\backprime}}f$ is the disjointness formula $G \,\#\, H$. The key property of a separator formula is this. For effects δ, ε that are well formed in Γ, $\Gamma' \supseteq \Gamma$, and Γ'-states σ, τ, if $\sigma{\to}\tau \models \varepsilon$ and $\sigma \models \delta \cdot/. \varepsilon$, then $Agree(\sigma, \tau, \delta)$ (Lemma 6.8, "separator agreement" in Part I). Γ' is used to account for agreement for intermediate states that may have local variables and procedure parameters as in Section 6.

For our example, it happens that $\delta_x \cdot/. (\mathsf{wr}\,\mathsf{self.any}, \mathsf{wr}\,\mathsf{alloc})$ is $true$. So, to complete the proof of $\{I_{set}\}B_{add}\{I_{set} \wedge elements = \mathsf{old}(elements) \cup \{i\}\,\}$, the key step is to take Q to be $Iexcept$ and δ to be δ_x in the frame rule:

$$\textsc{Frame}\,\frac{\vdash \{\,P\,\}\; C\; \{\,P'\,\}\; [\varepsilon] \qquad P \vdash \delta\ \mathsf{frm}\ Q \qquad P \wedge Q \Rightarrow \delta \cdot/. \varepsilon}{\vdash \{\,P \wedge Q\,\}\; C\; \{\,P' \wedge Q\,\}\; [\varepsilon]}.$$

The proof is completed using the rule of consequence and the logical equivalence of I_{set} to $SetI(\mathsf{self}) \wedge Iexcept$.

Soundness of rule Frame is a direct consequence of frame validity. Note that by definition $P \vdash \delta\ \mathsf{frm}\ Q$ is valid iff $P \wedge Q \vdash \delta\ \mathsf{frm}\ Q$ is valid.

Similar reasoning verifies the implementation of $remove$. For verifying $remove$, the precondition P in Frame will be $true \wedge I_{set}$ because $true$ is the precondition of $remove$ in Figure 1. The effects of $remove$ include $\mathsf{wr}\,\mathsf{self}.rep^{\boldsymbol{\backprime}}\mathsf{any}$ and the relevant separator formula $\delta_x \cdot/. \mathsf{wr}\,\mathsf{self}.rep^{\boldsymbol{\backprime}}\mathsf{any}$ involves nontrivial disjointnesses:

$$(pool \setminus \{\mathsf{self}\}) \,\#\, \mathsf{self}.rep \;\wedge\; (pool \setminus \{\mathsf{self}\})^{\boldsymbol{\backprime}} rep \,\#\, \mathsf{self}.rep.$$

The second conjunct is a consequence of the ownership property (5) which follows from I_{set}. The first conjunct follows from the fact that elements of $\mathsf{self}.rep$ have type $Node$ and those of $pool \setminus \{\mathsf{self}\}$ have type Set. This fact could be used if we strengthen $SetI(s)$ to say $\mathsf{type}(Node, s.rep)$ and strengthen I_{set} to say $\mathsf{type}(Set, pool)$. Alternatively, one can verify the implementation of $remove$ for effects $\mathsf{wr}\,\mathsf{self}.lst, \mathsf{self}.rep^{\boldsymbol{\backprime}}(nxt, own)$, which

gives rise to a different separator formula. These effects are not suitable to appear in the interface specification, but are subsumed by the ones in Figure 1.

4. PROCEDURES AND HYPOTHETICAL JUDGMENTS

This section extends the language of Part I with procedure blocks and procedure calls. The construct "letrec $m(x:T) = B$ in C" links client command C with implementation B of procedure m. Often program logic is formalized using a fixed association between procedure specifications and implementations (e.g. Apt et al. [2012]), but here we model linking using program configurations that contain a procedure environment. The command letrec $m(x:T) = B$ in C extends the environment, binding m to B, and proceeds to execute C. Calls to m in C retrieve and execute B. The linking rule, that is, the proof rule for letrec, requires C to be verified under a hypothesis—the specification of m—and discharges that hypothesis by requiring B to satisfy the specification. This is the gist of the procedure call rule of Hoare [1971].

For the semantics of a hypothetical judgment like the correctness of C under an assumption about m, there is an intuitively appealing semantics in terms of fully linked programs: C is correct when linked with any B' that satisfies the specification of m. In a denotational model, the semantics of C can be parameterized on the semantics of m and quantification taken over all meanings for m that satisfy the specification. That quantification can be avoided by interpreting the client command C with respect to a single meaning that uses nondeterminacy to represents all correct implementations of m. This is called the "worst program" by O'Hearn et al. [2009], where this technique helps streamline the difficult proof of soundness of the second order frame rule. The technique has also been used in a big-step operational semantics (see Pierik and de Boer [2005a] or Pierik [2006, Sect. 8.3]).

Refinement calculi carry the idea one step further, augmenting the programming language with specification statements which behave as the least refined program that satisfies a particular specification [Back and von Wright 1998; Morgan 1994; Shaner et al. 2007]. Verification-condition generators often use the idea: The axiomatic semantics of a procedure call becomes roughly an assertion (of the procedure precondition) followed by an assumption (of its postcondition). Refinement calculi have been interpreted using predicate transformers, a sufficiently rich domain to include models of least-refined programs with respect to total correctness. For partial correctness, pre-post relations suffice for denotational semantics, in particular for the "worst programs" in O'Hearn et al. [2009].

In this article, we do not use specification statements but rather a conventional logic of correctness with hypotheses. The operational semantics of a program under hypothesized procedure specifications is a conventional transition (small-step) semantics, including calls of procedures in the environment (*environment calls*). But for *context calls*, that is, calls of procedures in the hypothesis context, the semantics takes a big step to the final state of the call, in accord with the specification, as in the relational semantics of "worst programs". The ultimate justification of our semantics is the linking rule: Its soundness says precisely that correctness of C under a hypothesis for m implies correctness of letrec $m(x:T) = B'$ in C for every B' that satisfies that hypothesis.

Many works prove soundness of a program logic with respect to a transition semantics, but usually indirectly, via a denotational or big-step operational semantics (e.g., Apt et al. [2009]). Here, small steps are integral to our semantics of dynamic boundaries, presented in Section 6. In this section we give the proof rules for procedure calls and the linking of procedure blocks in simplified form, without dynamic boundaries. The soundness argument for the linking rule is sketched. The sketch brings to light the

need for somewhat intricate technical results about the semantics which are developed in Section 5.

4.1. Syntax Extended with Procedures

As in Part I, we restrict the syntax and semantics as follows.

Assumption 4.1. A set $SpecOnlyVar \subseteq VarName \setminus \{\mathsf{alloc}\}$ is designated as *specification-only*. These do not occur in any command, not even in ghost code. They do not influence allocation: $\mathit{Fresh}(\sigma) = \mathit{Fresh}(\tau)$ if σ differs from τ only on some specification-only variables. Finally, we disallow $\mathsf{wr}\, x$ for specification-only x.

Owing to this treatment of specification-only variables, our proof rules and specifications follow the conventional pattern; although we are explicit about a typing context, our syntax does not explicitly quantify specification-only variables over specifications.

Recall that typing is formalized with respect to a collection of named record-pointer types (class declarations) that are not explicit in the typing judgments. We extend typing contexts to include not only variable typings $x : T$ but also procedure typings, written $m : (x : T)$. This designates a procedure named m with a single parameter x. (The generalization to multiple parameters and out-parameters should be evident, and it is not difficult to add an implicit self parameter.) We often say *variable* for variable name, and assume that variables are distinct from procedure names. We still use the letter Γ for typing contexts. Note that variables x have data types T and are in general mutable, whereas procedure names are bound by letrec and are not reassignable.

As usual in Hoare logics with procedures, we want to prevent calls $m(z)$ where z is a global variable accessible to the implementation of m, as this would complicate the substitutions and restrictions needed for sound proof rules. This loses no generality: such a call can be expressed as $\mathsf{var}\ x : T\ \mathsf{in}\ x := z; m(x)$. The need for this restriction arises only in a few places; to formalize it, we posit an infinite set $Locals$ such that

$$Locals \subseteq VarName \setminus SpecOnlyVar.$$

Elements of $Locals$ can be used as local variables and procedure parameters. Elements of $VarName \setminus Locals$ can be used as global variables of the program, which model both the program input/outputs and also those variables (like $pool$ in SET) that are used within particular modules. We restrict the typing rule for local variable blocks, adding condition $x \in Locals$:

$$\frac{\Gamma, x : T \vdash C \qquad x \in Locals}{\Gamma \vdash \mathsf{var}\ x : T\ \mathsf{in}\ C}.$$

We add the corresponding condition to the semantics:

$$\frac{x' \notin dom(\sigma) \qquad x' \in Locals \qquad C' = C^x_{x'}}{\langle \mathsf{var}\ x : T\ \mathsf{in}\ C,\ \sigma\rangle \longmapsto \langle C'; \mathsf{evar}(x'),\ \mathit{Extend}(\sigma, x', \mathit{default}(T))\rangle}. \tag{6}$$

The proof rule for local blocks carries over from Part I unchanged. Here and in the sequel, we use the compact notation like $C^x_{x'}$ for substitution of x' in place of x, rather than $C/x{\to}x'$ which is used in Part I.

The typing rule for procedure call is as follows:

$$\frac{\Gamma(z) = T \qquad \Gamma(m) = (x : T) \qquad z \in Locals}{\Gamma \vdash m(z)}.$$

It imposes the restriction on calls discussed above. For procedure linking, the rule is

$$\frac{\Gamma \restriction Locals, m : (x : T), x : T \vdash B \qquad \Gamma, m : (x : T) \vdash C \qquad x \in Locals \qquad B\ \mathsf{is\ letrec\text{-}free}}{\Gamma \vdash \mathsf{letrec}\ m(x : T) = B\ \mathsf{in}\ C}.$$

The parameter in a procedure binding must be in $Locals$. Furthermore, removing $Locals$ from the domain of Γ restricts the procedure body B so it cannot refer to variables bound by enclosing local variable blocks. (The "toss" symbol $\restriction$ is used for dropping an element or set of elements from the domain of a mapping.) The restrictions model a language like C (and like Java without inner classes) with only top-level procedure declarations, and it helps keep the semantics simple. A complete program will have no free procedure names, but will have some global variables, for input and output and module variables like $pool$; these will be in Γ. There may also be specification-only variables in Γ, for use in proofs and procedure specifications. According to our notational conventions, m in the typing rule must not occur in Γ (but x may be in $dom(\Gamma)$). This precludes shadowing in C of one procedure binding by an inner binding to the same name. That is convenient in the semantics because it lets us treat method environments simply as maps. Note that the implementation B of m can make recursive reference to m.

For the most part, we focus on linking a single procedure as in the grammar (Figure 6). However, in discussing soundness of the proof rule for letrec we will encounter a fine point that pertains to the general case: simultaneous linking of n procedures that may be mutually recursive. Here is the typing rule.

$$\frac{\begin{array}{c}\Gamma' = m_1 : (x_1 : T_1), \ldots, m_n(x_n : T_n) \\ \Gamma, \Gamma' \vdash C \qquad \Gamma \restriction Locals, \Gamma', x_i : T_i \vdash B_i \qquad x_i \in Locals \text{ and } B_i \text{ is letrec-free}, \quad \text{for all } i\end{array}}{\Gamma \vdash \mathsf{letrec}\ m_1(x_1 : T_1) = B_1; \ldots; m_n(x_n : T_n) = B_n\ \mathsf{in}\ C}. \quad (7)$$

We do not include a rule for extending the context, but as in Part I for the other binding constructs (var for commands and quantifiers for formulas), it is admissible. To be precise: if $\Gamma \vdash C$ and m is not in $dom(\Gamma)$ and is not bound in C then $\Gamma, m : (x : T) \vdash C$ is derivable. So we retain the hygiene property from Part I that in derivable typings no variable or procedure occurs both bound and free, nor is a binding shadowed.

4.2. Correctness Judgments and Program Semantics with Hypotheses

The operational semantics of procedure blocks uses environments. Substitution is only used for renaming of variables, in procedure call (and variable blocks, cf. (6)), so the store can be modeled simply as a mapping of variables to values. End-marker commands are used so the control state can be modeled using a single command. In this section, we first address the use of end-markers. Then we define specifications, on which basis we define configurations and the transition relation. The transition semantics of context calls is explained at length.

Extended Commands. Command $\mathsf{elet}(m)$ ends the scope of a procedure block, in the same way as $\mathsf{evar}(x)$ is used for a local variable in Eq. (6). For procedure parameters we use "end-call", $\mathsf{ecall}(x)$, which has the same semantics as evar but is distinguished for technical reasons explained in Section 5.

We use the term *ordinary commands* for the ones defined in Figure 6. The *end-marker* commands are $\mathsf{evar}(x)$, $\mathsf{ecall}(x)$, and $\mathsf{elet}(m)$. An *extended command* is a command that may include end-markers, but only at the top level to encode a stack of open scopes. That is, an extended command is one that matches the regular pattern

$$(C(; (\mathsf{evar}(x) \mid \mathsf{ecall}(x) \mid \mathsf{elet}(m)))^*)^*,$$

where C ranges over ordinary commands. For this to make sense, we consider semicolon to be associative. To be very precise, the set of extended commands is quotiented

by the equivalence relation defined by[9] $B;(C;D) \equiv (B;C);D$ and $\mathsf{skip};C \equiv C \equiv C;\mathsf{skip}$. Treating skip as left unit obviates the need for an explict transition rule for the terminating step of C in a sequence $C;D$.

The 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. Define $\mathit{Active}(C_1;C_2) = \mathit{Active}(C_1)$ and $\mathit{Active}(C) = C$ if there are no C_1, C_2 such that C is $C_1;C_2$.

In the following, and in contrast with Part I of the article, we have several reasons to interpret predicates and effects at intermediate configurations. So we consider in detail the typability of intermediate configurations, including the control state. There are no typing rules for end-markers, so $\Gamma \vdash C$ implies that C has no end-markers. In order to formulate an invariant of the semantics concerning typing of the control state, we inductively define a judgment $\Gamma \Vdash C$ on extended commands C, which deals with the end markers as follows.

$$\frac{\Gamma \vdash C}{\Gamma \Vdash C} \qquad \frac{\Gamma \vdash C \qquad \Gamma\restriction x \Vdash D}{\Gamma \Vdash C;\mathsf{evar}(x);D} \qquad \frac{\Gamma \vdash C \qquad \Gamma\restriction x \Vdash D}{\Gamma \Vdash C;\mathsf{ecall}(x);D} \qquad \frac{\Gamma \vdash C \qquad \Gamma\restriction m \Vdash D}{\Gamma \Vdash C;\mathsf{elet}(m);D}.$$

One property of this judgment is that if $\Gamma \Vdash C$ and $\mathit{Active}(C)$ is not an end marker, then $\Gamma \vdash \mathit{Active}(C)$. Typability in this sense is preserved by transitions: see item (2) in Definition 4.3 and Lemma 4.6. As an example, a configuration with control state $\mathsf{var}\ x:T\ \mathsf{in}\ \mathsf{skip}$ transitions to one with control state $\mathsf{evar}(x')$. Because $\mathsf{evar}(x')$ can be written $\mathsf{skip};\mathsf{evar}(x');\mathsf{skip}$, we get $\Gamma \Vdash \mathsf{skip};\mathsf{evar}(x');\mathsf{skip}$ using the previous rules.[10]

Specifications and procedure contexts

Definition 4.2 (*Procedure Specification, Context*). A *procedure context*, Δ, is a set of *specifications*, each of the form

$$\{\ Q\ \}\ m(x:T)\ \{\ Q'\ \}\ [\varepsilon] \tag{8}$$

such that no procedure name m has more than one specification.[11] For the specification (8) to be *well formed in* Γ, all of Q, Q', ε should be well-formed in $\Gamma\restriction \mathit{Locals}, x:T$ and no m specified in Δ is in $\mathit{dom}(\Gamma)$. Moreover the frame condition ε should not contain $\mathsf{wr}\,x$. (Nor can it contain $\mathsf{wr}\,s$ for any s in $\mathit{SpecOnlyVar}$, according to Assumption 4.1.)

The use of $\Gamma\restriction \mathit{Locals}, x:T$, with locals removed from Γ, allows specifications to refer to global variables in Γ. The exclusion of $\mathsf{wr}\,x$ from ε enforces the usual constraint in Hoare logics that the procedure body does not update the parameter, so that any use of x in Q' and ε refers to its initial value. Note that ε may still refer to x in forms such as $\mathsf{wr}\,(\{x\}\cup r)\text{`}f$. A specification is not useful unless the parameter x is in Locals, in accord with the typing rule for letrec. We write "m in Δ" to mean there is a specification of m in Δ.

Specification languages often allow postconditions to refer to the initial state using old-expressions. One can desugar, for example, $\mathsf{old}(E)$ by adding specification-only

[9] We do not need to extend this to a congruence with respect to other program constructs, as it is only needed to simplify notions that pertain to the control state.

[10] The definition of $\Vdash$ is slightly more general than needed, in that $\Gamma \Vdash C;\mathsf{evar}(x);D$ can hold even if $x \notin \mathit{dom}(\Gamma)$. That could be eliminated, but there is no need to complicate the definition of $\Vdash$ because it is only used in conjunction with the other conditions of Definition 4.3.

[11] We gain some minor streamlining of notation by restricting to a single specification per procedure, but it is straightforward to allow multiple specifications.

variable s and precondition $s = E$. Our formalization uses specification-only variables, and not old-expressions, because they support simpler proof rules (e.g., for sequential composition). A minor benefit is increased expressiveness: occasionally it is convenient for the precondition not to determine s uniquely.

In this section, we consider hypothetical correctness judgments of the form

$$\Delta \vdash^{\Gamma} \{ P \}\ C\ \{ P' \}\ [\varepsilon]\,. \tag{9}$$

The judgment is *well formed* if $\Delta, P, P', \varepsilon$ are well formed in Γ; moreover C is an ordinary command and $\Gamma, \mathit{sigs}(\Delta) \vdash C$. Here sigs extracts the procedure signatures; for example, sigs applied to the single specification (8) yields $m : (x : T)$.

Note that Γ may include locals used in C and in the specification of C, but the specifications in Δ only refer to $\Gamma \restriction \mathit{Locals}$. We also allow Γ to declare procedures, but this is only a technical convenience for the soundness proofs. Procedures called in C should be in Δ, not Γ, in order to use the proof rule for procedure call.

Specification-only variables that occur in Δ might seem to be scoped over the whole judgment. But this is merely a technical artifact to streamline notation and typing. As usual in Hoare logic, such variables are interpreted by universal quantification over the pre- and post-condition. This is formalized in Figure 9, explained later.

As in Part I of the article, the correctness judgment is intended to say that from any initial state that satisfies P, C does not fault and if it terminates then the final state satisfies P' and the effects are allowed by ε. But we need to consider computations of C in the context of procedures that satisfy the specifications Δ. We use a 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.

The correctness judgment expresses not only that C satisfies its pre/post/effect specification, but also that it does so for any implementations of the procedures in Δ. Such implementations include those that fail unless the specified precondition holds; so we need a semantics in which precondition failure can be expressed. One possibility is for there to be no transition in case the precondition does not hold. But it turns out to be slightly more succinct to formulate various definitions using an explicit transition to a failure state. In Naumann and Banerjee [2010], we used *fault* for this purpose, but ordinary faults due to null dereference are observable, whereas here we are trying to capture reasoning under hypotheses, not an operational notion per se. So we use a distinct quasi-state, *p-fault*, that signifies an attempt to invoke a context procedure outside its specified precondition.

The Transition Relation. Transitions relate configurations; there are also faulting transitions. A *configuration* is a triple $\langle C, \sigma, \mu \rangle$ where C is an extended command, σ is a state, and the *procedure environment* μ is a partial function from procedure names to parameterized commands of the form $(x : T.C)$. A Γ-*state* is comprised of a heap and a store that assigns values to the variables of Γ. Procedure typings in Γ are not relevant. So this definition is the same as a state, as defined in Part I of this article, for $\Gamma \restriction \mathit{ProcName}$. A Γ-*environment* μ is a procedure environment such that $\mathit{dom}(\mu)$ is the set of procedure names in $\mathit{dom}(\Gamma)$ and for every $m : (x : T)$ in Γ we have $\mu(m) = (x : T.B)$ for some ordinary command B such that $\Gamma \restriction \mathit{Locals}, x : T \vdash B$, no specification-only variable occurs in B, and B is letrec-free.

Definition 4.3. Let Δ be well formed for Γ. A configuration $\langle C, \sigma, \mu \rangle$ is *compatible with* Γ and Δ provided there is some Γ' such that the following hold:

(1) $\Gamma' \supseteq \Gamma$ and $\mathit{VarName} \cap (\mathit{dom}(\Gamma') \setminus \mathit{dom}(\Gamma)) \subseteq \mathit{Locals}$
(2) $\Gamma', \mathit{sigs}(\Delta) \Vdash C$ and σ is a Γ'-state and μ is a Γ'-environment

$$\frac{\langle C,\,\sigma\rangle \longmapsto \langle C',\,\sigma'\rangle}{\langle C,\,\sigma,\,\mu\rangle \overset{\Delta}{\longmapsto} \langle C',\,\sigma',\,\mu\rangle} \qquad \frac{\langle C,\,\sigma\rangle \longmapsto \mathit{fault}}{\langle C,\,\sigma,\,\mu\rangle \overset{\Delta}{\longmapsto} \mathit{fault}}$$

$$\frac{m \notin \mathit{dom}(\mu)}{\langle \mathsf{letrec}\ m(x:T) = B\ \mathsf{in}\ C,\,\sigma,\,\mu\rangle \overset{\Delta}{\longmapsto} \langle C; \mathsf{elet}(m)\,,\,\sigma,\,\mathit{Extend}(\mu, m, (x:T.B))\rangle}$$

$$\frac{\mu(m) = (x:T.C) \qquad x' \notin \mathit{Dom}(\sigma) \qquad x' \in \mathit{Locals} \qquad C' = C^{x}_{x'}}{\langle m(z),\,\sigma,\,\mu\rangle \overset{\Delta}{\longmapsto} \langle C'; \mathsf{ecall}(x')\,,\,\mathit{Extend}(\sigma, x', \sigma(z)),\,\mu\rangle}$$

$$\langle \mathsf{elet}(m),\,\sigma,\,\mu\rangle \overset{\Delta}{\longmapsto} \langle \mathsf{skip},\,\sigma,\,\mu\restriction m\rangle \qquad \langle \mathsf{ecall}(x),\,\sigma,\,\mu\rangle \overset{\Delta}{\longmapsto} \langle \mathsf{skip},\,\sigma\restriction x,\,\mu\rangle$$

$$\frac{\langle C,\,\sigma,\,\mu\rangle \overset{\Delta}{\longmapsto} \langle C',\,\sigma',\,\mu'\rangle}{\langle C\,;D,\,\sigma,\,\mu\rangle \overset{\Delta}{\longmapsto} \langle C'\,;D,\,\sigma',\,\mu'\rangle} \qquad \frac{\langle C,\,\sigma,\,\mu\rangle \overset{\Delta}{\longmapsto} \text{(p-)}\mathit{fault}}{\langle C\,;D,\,\sigma,\,\mu\rangle \overset{\Delta}{\longmapsto} \text{(p-)}\mathit{fault}}$$

$$\frac{\begin{array}{c}\Delta \text{ contains } \{P\}m(x:T)\{P'\}[\varepsilon] \qquad \overline{s} \text{ lists all specification-only variables in } P, P', \varepsilon \\ \Gamma' \supseteq \Gamma \qquad \exists \overline{n} \cdot (\sigma \models^{\Gamma'} P^{x,\overline{s}}_{z,\overline{n}}) \qquad \sigma{\to}\sigma' \models \varepsilon^{x}_{z} \qquad \forall \overline{n} \cdot (\sigma \models^{\Gamma'} P^{x,\overline{s}}_{z,\overline{n}}) \Rightarrow (\sigma' \models^{\Gamma'} P'^{x,\overline{s}}_{z,\overline{n}})\end{array}}{\langle m(z),\,\sigma,\,\mu\rangle \overset{\Delta}{\underset{\Gamma}{\longmapsto}} \langle \mathsf{skip},\,\sigma',\,\mu\rangle}$$

$$\frac{\begin{array}{c}\Delta \text{ contains } \{P\}m(x:T)\{P'\}[\varepsilon] \qquad \overline{s} \text{ lists all specification-only variables in } P, P', \varepsilon \\ \Gamma' \supseteq \Gamma \qquad \sigma \text{ is a } \Gamma'\text{-state} \qquad \neg(\exists \overline{n} \cdot (\sigma \models^{\Gamma'} P^{x,\overline{s}}_{z,\overline{n}}))\end{array}}{\langle m(z),\,\sigma,\,\mu\rangle \overset{\Delta}{\underset{\Gamma}{\longmapsto}} \mathit{p\text{-}fault}}$$

Fig. 9. The transition relation $\overset{\Delta}{\underset{\Gamma}{\longmapsto}}$, eliding Γ in most cases.

(3) For every m in $\mathit{dom}(\Gamma') \setminus \mathit{dom}(\Gamma)$, there is exactly one $\mathsf{elet}(m)$ in C, and these are the only elet commands in C.
(4) For every variable x in $\mathit{dom}(\Gamma') \setminus \mathit{dom}(\Gamma)$, there is exactly one end-marker, either $\mathsf{evar}(x)$ or $\mathsf{ecall}(x)$, in C, and these are its only evars/ecalls.

Item (1) says that Γ' extends Γ only with procedures and local variables (for the currently open scopes of letrec blocks, local variable blocks, and procedure invocations). Item (2) states simple well-formedness conditions, using the extended typing judgment $\Vdash$. One consequence is that no m specified in Δ is in $\mathit{dom}(\mu)$, as otherwise $\Gamma', \mathit{sigs}(\Delta)$ would not be a well formed typing context. Furthermore, no m that is letrec-bound in C is in $\mathit{dom}(\mu)$ or in Δ. Recall that the typing rule for letrec already prevents shadowing within procedure bodies and within C. Apropos μ being a Γ'-environment, note that procedure bodies in μ are actually well formed with respect to the initial environment Γ, as they cannot refer to the locals added in Γ'. Items (3) and (4) say there is no shadowing among pending end-markers, and the domain of Γ' is exactly the domain of Γ together with those end-markers.

The transition relation $\overset{\Delta}{\underset{\Gamma}{\longmapsto}}$ is defined inductively by the rules in Figure 9. A configuration may step to another configuration, to *fault* (for runtime error), and to *p-fault*.

The first two rules in Figure 9 are intended to succinctly reuse the semantics of Part I, but now the commands are for the language extended with procedures. (For example, C can be $\mathsf{if}\ E\ \mathsf{then}\ B\ \mathsf{else}\ D$ where B and D include procedure calls and procedure blocks.) Moreover, the semantics of local variable blocks is revised as per (6).

The next rule is the transition for letrec; it opens the scope of a procedure block and adds the procedure to the environment. The condition $m \notin dom(\mu)$, which holds in any compatible configuration, makes clear that the use of *Extend* is defined. The next rule invokes a procedure by retrieving its body from the environment, renaming the parameter apart from any in scope and adding it, with initial value, to the store. Next come the transitions for the scope enders ecall and elet, with semantics analogous to that for evar.

Next are the inductive cases for sequencing.[12] For brevity, we sometimes write "$\langle C, \sigma, \mu\rangle \longmapsto^*$ *(p-)fault*" to abbreviate "$\langle C, \sigma, \mu\rangle \longmapsto^*$ *fault* or $\langle C, \sigma, \mu\rangle \overset{\Delta}{\longmapsto}{}^*$ *p-fault*". Here we abbreviate transition rules for each kind of fault.

Remark 4.4. Apt et al. [2009] use an elegant semantics for local blocks and environment procedure calls that avoids the need for renaming. It relies on the use of states where every possible variable is assigned a value. The transition axiom for var is $\langle \mathsf{var}\ x : T := E\ \mathsf{in}\ C, \sigma\rangle \longmapsto \langle x := E; C; x := \sigma(x), \sigma\rangle$. Note that C is used unchanged, rather than a renamed copy as in our axiom (6). The initial value $\sigma(x)$ is used as a literal in the trailing assignment, which restores the value of x in case it is in use in the outer scope. This embodies the stack discipline while retaining a flat model of state. The initializer expression E, evaluated in the outer scope, caters for use of local blocks in semantics of procedure call: the axiom $\langle m(z), \sigma\rangle \longmapsto \langle \mathsf{var}\ x : T := z\ \mathsf{in}\ B, \sigma\rangle$, where m is $(x : T.B)$, serves even for recursive calls $m(x)$.

In this article, we choose a somewhat different model of states and objects, and more explicit formalization of typing. We find the renaming semantics convenient, for example, for formulating the notion of compatible configuration (Definition 4.3); in particular the typing condition for C there is simple and useful. We also avoid the need for reference literals to occur in extended commands. Moreover, to prove soundness of the linking rule we need to distinguish procedure call/return from entry/exit of local variable blocks.

Semantics of Context Procedure Calls. In case Δ is empty, the transition relation provides a standard environment-based semantics. In particular, *p-fault* never occurs. The last two rules in Figure 9 amount to a relational semantics of procedures specified in Δ. The ultimate justification of these transition rules is that the semantics is sound for reasoning about closed programs, where Δ is empty. The details are delicate.

The basic idea is that a configuration $\langle m(z), \sigma, \mu\rangle$, where m is specified in Δ, takes a single step to $\langle \mathsf{skip}, \sigma', \mu\rangle$, provided that σ satisfies the precondition and σ' is any state that satisfies the postcondition and frame condition. If instead σ does not satisfy the precondition, the step goes to *p-fault*, an artifact without operational meaning that signifies precondition failure. For example, let Δ contain the single specification $\{\, x \geq 0 \,\}\ m(x : \mathsf{int})\ \{\, w > 0 \,\}\ [\mathsf{wr}\ w]$, in $\Gamma = [w : \mathsf{int}, z : \mathsf{int}]$. The successors of configuration $\langle m(z), [w : 0, z : 7], \mu\rangle$ include $\langle \mathsf{skip}, [w : 8, z : 7], \mu\rangle$ and $\langle \mathsf{skip}, [w : 9, z : 7], \mu\rangle$. (Here μ is irrelevant.) On the other hand, from the configuration $\langle m(z), [w : 0, z : -1], \mu\rangle$ where the precondition is not satisfied, the only successor is *p-fault*. By contrast, suppose m is not in Δ but rather the environment μ has a correct implementation, say $\mu(m) = (x : \mathsf{int}.\ w := x; w := w + 1)$. Then, the successor of $\langle m(z), [w : 0, z : 7], \mu\rangle$ is $\langle w := x'; w := w + 1; \mathsf{ecall}(x'), [w : 0, z : 7, x' : 7], \mu\rangle$ for some fresh x'. This proceeds by steps $\langle w := w + 1; \mathsf{ecall}(x'), [w : 7, z : 7, x' : 7], \mu\rangle\langle \mathsf{ecall}(x'), [w : 8, z : 7, x' : 7], \mu\rangle\langle \mathsf{skip}, [w : 8, z : 7], \mu\rangle$.

[12] There is not a separate rule for termination of the first command in a sequence: if $\langle C, \sigma, \mu\rangle \overset{\Delta}{\longmapsto} \langle \mathsf{skip}, \sigma', \mu'\rangle$ then we have $\langle C ; D, \sigma, \mu\rangle \overset{\Delta}{\longmapsto} \langle D, \sigma', \mu'\rangle$ by one of the given rules and the identity $\mathsf{skip}; D \equiv D$.

The basic idea for calls of context procedures might be formalized using the following putative definition:

$$\frac{\begin{array}{ll} & \Delta \text{ contains } \{P\}m(x:T)\{P'\}[\varepsilon] \\ \Gamma' \supseteq \Gamma & \sigma \models^{\Gamma'} P^x_z \qquad \sigma' \models^{\Gamma'} {P'}^x_z \qquad \sigma{\rightarrow}\sigma' \models \varepsilon^x_z \end{array}}{\langle m(z),\ \sigma,\ \mu\rangle \xrightarrow[\Gamma]{\Delta} \langle \mathsf{skip},\ \sigma',\ \mu\rangle}. \tag{10}$$

The transition is instantiated with Γ', a typing context for which σ is a state and μ an environment. This is needed because the configuration may have been reached after entering var and letrec blocks as well as procedure calls, so that σ and μ may include variables and procedures not in Γ or Δ. By constraining Γ' to suit σ, the definition stipulates[13] that the final state σ' has the same variables (and typing) as the initial state σ. This is expressed by $\sigma' \models^{\Gamma'} P'$, because in Part I of this article we define $\tau \models^{\Gamma} Q$ to mean: τ is a Γ-state and it satisfies Q (for any Γ, τ, Q). The condition $\sigma{\rightarrow}\sigma' \models \varepsilon^x_z$ ensures that none of the extra variables are changed: the specification, including ε, will be well formed in Γ and thus not contain $\mathsf{wr}\, w$ for w in $dom(\Gamma') \setminus dom(\Gamma)$. The second rule for context call in Figure 9 needs to explicitly constrain Γ' to be the right one for σ, because the negation of $\sigma \models^{\Gamma'} P^x_z$ says either σ is not a Γ'-state or does not satisfy the formula.

The putative transition rule (10) has a flaw that is fixed by quantifying the specification-only variables in the specification of m. We want the effect that those variables are scoped, and universally quantified, over the specification, but as usual in Hoare logic the quantification is not explicit in the syntax. The desired effect is manifest in proof rules, especially SUBST, and embodied in the transition rules for context call (in Figure 9).

As an example, let Δ contain the single specification $\{x = s\}m(x:\mathsf{int})\{w > s\}[\mathsf{wr}\, w]$ in context $\Gamma = [s:\mathsf{int}, w:\mathsf{int}, z:\mathsf{int}]$ where s is in $SpecOnlyVar$. Consider configuration $\langle m(z), [s:1, w:2, z:3]\,, \mu\rangle$. In keeping with the intention that s is effectively universally quantified over the specification, we expect the current value of s to be irrelevant (!) and instead what matters is that there is some value for s, namely 3, for which the precondition is satisfied. The configuration should step to any configuration of the form $\langle \mathsf{skip},\ [s:1, w:n, z:3]\,,\ \mu\rangle$ where $n > 3$.

To streamline the transition rules for context call (in Figure 9), we indulge in a minor abuse of notation. Suppose $\overline{s}$ is a list of specification-only variables that are in scope, and $\overline{n}$ a list of values—not variables—of corresponding type. We write

$$\sigma \models P^{\overline{s}}_{\overline{n}} \quad \text{to abbreviate} \quad [\sigma \mid \overline{s}: \overline{n}] \models P. \tag{11}$$

The point is that $\overline{n}$ may include reference values and we do not want reference literals in formulas, so strictly speaking the substitution $P^{\overline{s}}_{\overline{n}}$ is not defined. In the transition rules, we mix notation, for example, $\sigma \models P^{x,\overline{s}}_{z,\overline{n}}$ abbreviates $[\sigma \mid \overline{s}: \overline{n}] \models P^x_z$. We use identifier $\overline{n}$ for values, whereas the letters $r, s, t \cdots z$ are used for variable identifiers. Specification-only variables are disjoint from $Locals$, so $\overline{s}$ is disjoint from $dom(\Gamma') \setminus dom(\Gamma)$ in the transition rules.

In practice, most preconditions P have the property that if $\sigma \models P$, then the values $\overline{n} = \sigma(\overline{s})$ are uniquely determined. For example, let P be $s = x$. If $\sigma \models P$, then for any $n \neq \sigma(s)$, $[\sigma \mid s: n] \not\models P$. For such precondition, the first transition rule can be

[13]There may be more than one Γ' for which a σ is well formed—only because the value null has many types—but the choice makes no difference because no formula can make the distinction. An alternate formalization would include typing contexts in configurations, but we prefer not.

simplified considerably: this is the rule we used in this example in which the value of s is uniquely determined to be 3 for $\sigma \models z = s$ to hold.

$$\frac{\begin{array}{c}\Delta \text{ contains } \{P\}m(x:T)\{P'\}[\varepsilon] \qquad \overline{s} \text{ lists all specification-only var's in } P, P', \varepsilon \\ \Gamma' \supseteq \Gamma \qquad \sigma \models^{\Gamma'} P^{x,\overline{s}}_{z,\overline{n}} \qquad \sigma \to \sigma' \models^{\Gamma'} \varepsilon^{x}_{z} \qquad \sigma' \models^{\Gamma'} P'^{x,\overline{s}}_{z,\overline{n}}\end{array}}{\langle m(z),\ \sigma,\ \mu\rangle \overset{\Delta}{\underset{\Gamma}{\longmapsto}} \langle \mathsf{skip},\ \sigma',\ \mu\rangle}. \tag{12}$$

In this rule, $\overline{n}$ is now a schematic variable. In case there are no specification-only variables, this is equivalent to (10).

The details of our treatment of specification-only variables are not essential. It would be cleaner, though more verbose, to explicitly declare them in specifications, in which case they would not be in the typing context. But our treatment achieves the same effect. Further insight about the semantics may be gleaned from the soundness proof for rule SUBST in Section 7.4 and from Lemma 4.10.

Remark 4.5. In a state where the precondition of a context procedure does not hold, there is a faulting transition. We could as well add two additional transition rules for such states. One would yield the initial configuration, so that the configuration would have diverging computations. This would be appropriate if we were working with total correctness. The other rule would yield a terminated configuration in which the state is arbitrary. This would be needed if we were considering refinement between programs.

4.3. Some Definitions and Properties of Semantics

We will almost always streamline the notation, writing just $\overset{\Delta}{\longmapsto}$ when Γ is clear from context. For that reason we usually refer to compatibility just with Δ.

There are three sources of nondeterminacy in the semantics. One is technical: the choice of names for local variables, and parameters in calls of environment procedures. The second is calls of context procedures: a specification need not determine a unique final state. Finally, the semantics of new is defined with respect to a given choice function, *Fresh*, that we allow to be nondeterministic (see Part I). Only the second leads to non-determinacy that can be detected by postconditions, because formulas do not involve pointer arithmetic or reference literals.

A *trace via* $\overset{\Delta}{\longmapsto}$ is a finite sequence of configurations, possibly ending with *fault* or *p-fault*, compatible with Δ and consecutively related by $\overset{\Delta}{\longmapsto}$. For example, for suitable states $\sigma, \ldots$ we have traces

$$\langle x := \mathsf{null}; x.f := 0,\ \sigma,\ \mu\rangle\langle x.f := 0,\ \sigma',\ \mu\rangle \mathit{fault}$$

and $\langle \mathsf{ecall}(z); y := 0, \sigma, \mu\rangle\langle y := 0, \sigma', \mu\rangle\langle \mathsf{skip}, \sigma'', \mu\rangle$. These happen to be maximal. Note that the first configuration of a trace need not be an initial one, that is, it may contain end-markers.

The conditions in Definition 4.3 are designed to ensure that compatibility is invariant.

LEMMA 4.6 (PRESERVATION OF COMPATIBILITY). *Suppose* $\langle C, \sigma, \mu\rangle$ *is compatible with* Δ. *If* $\langle C,\ \sigma,\ \mu\rangle \overset{\Delta}{\longmapsto} \langle C',\ \sigma',\ \mu'\rangle$, *then* $\langle C',\ \sigma',\ \mu'\rangle$ *is compatible with* Δ.

The proof is a matter of checking all the conditions for all the transition rules, and using the typing rules. Preservation of compatibility relies on a restriction in the typing rule for letrec, disallowing letrec-bindings in procedure bodies (a restriction like in ML). The loop-unfolding transition duplicates code, which may contain letrec-bindings, but in sequence rather than nested, so shadowing is not introduced. There may be local

variables with the same name as parameters of procedures in the environment, but it is renamed instances of those procedures that are used for calls.

LEMMA 4.7 (ENVIRONMENT BRACKETING). *If* $\langle C, \sigma, \mu\rangle$ *is compatible with* Δ *and* C *has no* elet*, then the final environment is the initial one:* $\langle C, \sigma, \mu\rangle \overset{\Delta}{\longmapsto}{}^{*} \langle \mathsf{skip}, \sigma', \mu'\rangle$ *implies* $\mu' = \mu$.

LEMMA 4.8 (ENVIRONMENT INVARIANCE). *If* m *is in* $dom(\mu)$ *and* elet(m) *does not occur in* C*, then* $\langle C, \sigma, \mu\rangle \overset{\Delta}{\longmapsto}{}^{*} \langle C', \sigma', \mu'\rangle$ *implies* $\mu'(m) = \mu(m)$.

The following is used to prove soundness of proof rules where the premises involve different procedure specifications from the conclusion.[14]

LEMMA 4.9 (CORRESPONDENCE). *Consider any* $\langle C, \sigma, \mu\rangle$ *compatible with* Δ*. Let* Θ *and* Υ *be procedure contexts that specify procedures neither in* $dom(\mu)$ *nor in* Δ*. Then* $\langle C, \sigma, \mu\rangle \overset{\Delta\Theta}{\longmapsto} \langle C', \sigma', \mu'\rangle$ *iff* $\langle C, \sigma, \mu\rangle \overset{\Delta\Upsilon}{\longmapsto} \langle C', \sigma', \mu'\rangle$*, for any* C', σ', μ'*. Moreover* $\langle C, \sigma, \mu\rangle \overset{\Delta\Theta}{\longmapsto}$ *fault iff* $\langle C, \sigma, \mu\rangle \overset{\Delta\Upsilon}{\longmapsto}$ *fault and* mutatis mutandis *for p-fault.*

PROOF. *Active*(C) cannot be the call of a context procedure except one in Δ; and the relations $\overset{\Delta\Theta}{\longmapsto}$ and $\overset{\Delta\Upsilon}{\longmapsto}$ differ only on calls of context procedures, more precisely those in Θ or Υ. □

The following is used to prove soundness of the substitution and linking rules.

LEMMA 4.10 (SPECIFICATION-ONLY VARIABLES). *Suppose that* $\langle C, \sigma, \mu\rangle$ *is compatible with* Γ, Δ *and specification-only variable* s *is in scope. Then for any value* n *and any* C', σ', μ' *we have the following three properties:*

$$\langle C, \sigma, \mu\rangle \overset{\Delta}{\longmapsto}{}^{*} \langle C', \sigma', \mu'\rangle \;\; \textit{implies} \;\; \sigma(s) = \sigma'(s)$$

$$\langle C, \sigma, \mu\rangle \overset{\Delta}{\longmapsto}{}^{*} \textit{(p-)fault} \quad \textit{iff} \quad \langle C, [\sigma \mid s : n], \mu\rangle \overset{\Delta}{\longmapsto}{}^{*} \textit{(p-)fault}$$

$$\langle C, \sigma, \mu\rangle \overset{\Delta}{\longmapsto}{}^{*} \langle C', \sigma', \mu'\rangle \quad \textit{iff} \quad \langle C, [\sigma \mid s : n], \mu\rangle \overset{\Delta}{\longmapsto}{}^{*} \langle C', [\sigma' \mid s : n], \mu'\rangle.$$

PROOF. By induction on number of transitions. (That is why we have C', though we only need the result in case $C' \equiv \mathsf{skip}$.) These properties rely on the fact that specification-only variables occur neither in code (Assumption 4.1) nor in procedure bodies in the environment (definition of Γ-environment). The second and third properties, which say the set of possible outcomes does not depend on the initial value of s, also rely on the assumption (Assumption 4.1) that the allocator is insensitive to specification-only variables.

The interesting issue is context procedure calls. Consider a call $m(z)$ where Δ contains specification $\{P\}m(x : T)\{P'\}[\varepsilon]$. By well formedness, ε does not contain $\mathsf{wr}\, s$, so the first property holds owing to semantics of context call. The second and third properties say that the initial value of s does not influence the set of outcomes. For the second property, note that the semantics (Figure 9) is defined so that it is not the value $\sigma(s)$ that determines whether the transition goes to *p-fault*, but rather the existence of some value of s for which P^x_z is satisfied. Moreover, the possible final states are those which satisfy P'^x_z for any value of s that satisfies the precondition; the final value of s is in any case the same as its initial value. □

Note that the alternative semantics (10) fails to satisfy the second property. For example, the specification for $fact(x)$ could have $P \equiv y = x$ for specification-only y,

[14] As Δ and Θ are comma-separated lists, one should write $\overset{\Delta,\Theta}{\longmapsto}$ but we omit the comma.

with postcondition $P' \equiv res = y!$ and effect $\mathsf{wr}\, res$. In the context of this specification, execution of the procedure body from an initial state where $y = 3$ and $x = 3$ would reach a call $fact(2)$ and then *p-fault* according to (10). In our semantics, the call $fact(2)$ transitions to a state where $res = 2$.

Our correctness judgment expresses only safety properties, but as sanity check we proved liveness of the transition relation. We say a specification $\{Q\}m(x:T)\{Q'\}[\varepsilon]$ with specification variables $\overline{s}$ is *satisfiable* if for every σ with $\exists \overline{n} \cdot (\sigma \models Q^{\overline{s}}_{\overline{n}})$ there is at least one σ' with $\sigma \rightarrow \sigma' \models \varepsilon$ and $\forall \overline{n} \cdot (\sigma \models Q^{\overline{s}}_{\overline{n}}) \Rightarrow (\sigma' \models Q'^{\overline{s}}_{\overline{n}})$. In case there is no such σ', a configuration $\langle m(x), \sigma, \ldots\rangle$ has no successor via $\overset{\Delta}{\longmapsto}$. We also say "$m(z)$ is an unsatisfiable call from σ", if $\exists \overline{n} \cdot (\sigma \models Q^{x,\overline{s}}_{z,\overline{n}})$ but there is no σ' for which $\sigma \rightarrow \sigma' \models \varepsilon$ and $\forall \overline{n} \cdot (\sigma \models Q^{x,\overline{s}}_{z,\overline{n}}) \Rightarrow (\sigma' \models Q'^{x,\overline{s}}_{z,\overline{n}})$.

Proposition 4.11 (*Liveness*). Suppose $\langle C, \sigma, \mu\rangle$ is compatible with Γ and Δ. Then $\langle C, \sigma, \mu\rangle$ has at least one $\overset{\Delta}{\longmapsto}$-successor unless either C is skip or $\mathit{Active}(C)$ is some $m(z)$ that is an unsatisfiable call from σ.

PROOF. By inspection of the transition rules. We assume that *Locals* is infinite, so there is an unbounded supply of local variables as needed for local blocks and parameters (in recursive procedures). (In this way, we are modeling unbounded stack space, just as we are modeling unbounded heap.) If $\mathit{Active}(C)$ is a call of procedure m, Lemma 4.6 tells us that either m is in the environment μ, satisfying the condition for calls of environment procedures, or it is in the context Δ. If m is in Δ, then either σ does not satisfy the precondition for m, in which case the successor is *p-fault*, or it does satisfy the precondition, in which case there are successors for all σ' that satisfy the postcondition. If there are no such σ', it is an unsatisfiable call. □

4.4. Proof Rules for Procedures

A correctness judgment $\Delta \vdash^{\Gamma} \{\, P \,\}\ C\ \{\, P' \,\}\ [\varepsilon]$ expresses properties of C with respect to traces via $\overset{\Delta}{\longmapsto}$. In particular, from P-states C should not fault and its terminal states should satisfy P' and be allowed by ε—the conditions dubbed "Safety", "Post", and "Effect" in Part I of this article. In addition, C should not *p-fault* due to calling a context procedure outside its specified precondition, which we dub the "Ctx-pre" condition in Section 6 where we define validity of judgments. In Section 6, we also associate the procedures in Δ with modules, the dynamic boundaries of which must be respected by C. In preparation for that, let us consider proof rules for procedures.

The axiom for procedure call should not be surprising:

$$\{P\}m(x:T)\{P'\}[\varepsilon] \vdash^{\Gamma} \{\, P^x_z \,\}\ m(z)\ \{\, P'^x_z \,\}\ [\varepsilon^x_z]\,.$$

Recall from Part I that we allow proof rules to be instantiated only with well formed judgments. Here, that means that z must be declared in Γ and indeed have type T. Furthermore, x and z are in *Locals* (by typing), and ε cannot contain $\mathsf{wr}\, x$ or $\mathsf{wr}\, z$ (by well-formedness of the specification).

What is interesting is the rule for procedure blocks. The rule itself is straightforward, but not so its soundness proof. For clarity we consider the special case of one procedure. In the rule, Θ specifies the procedure to be linked and Δ specifies the ambient library.

$$\textsc{Link0}\ \frac{\Theta \text{ is } \{Q\}m(x:T)\{Q'\}[\eta] \qquad \Delta, \Theta \vdash^{\Gamma} \{\, P \,\}\ C\ \{\, P' \,\}\ [\varepsilon] \qquad \Delta, \Theta \vdash^{\Gamma, x:T} \{\, Q \,\}\ B\ \{\, Q' \,\}\ [\eta]}{\Delta \vdash^{\Gamma} \{\, P \,\}\ \mathsf{letrec}\ m(x:T) = B\ \mathsf{in}\ C\ \{\, P' \,\}\ [\varepsilon]}.$$

The presence of Θ in the premise for B allows recursive calls, as in the typing rule.

The premise for C says not only that it does not fault, but also that it never calls a context procedure (i.e., in Δ, Θ) outside its specified precondition. The latter property, for procedure m of Θ, is forgotten in the conclusion of the rule. This exhibits a sense in which modular reasoning is incomplete. Consider as an example the program $\mathsf{letrec}\ m(x) = (y := x)\ \mathsf{in}\ m(0)$. It establishes postcondition $y = 0$, but this cannot be proved if m is given a specification with precondition $x \neq 0$. This should be no more surprising than the fact that a correct loop may be unprovable with respect to a poorly chosen loop invariant.

What is surprising is how difficult it is to prove the linking rule in transition semantics. We consider that now, in some detail. We sketch the soundness proof for LINK0 in the case where B makes no recursive calls—neither direct calls to m nor calls to environment procedures that call m. In the sketch, the semantics (10) is used for calls of context procedures, that is, we ignore specification-only variables. A sketch of the sketch is in Figure 10.

Suppose Θ is $\{Q\}m(x:T)\{Q'\}[\eta]$. Suppose both premises of LINK0 are valid, that is, $\Delta, \Theta \models^{\Gamma} \{\, P \,\}\ C_0\ \{\, P' \,\}\ [\varepsilon]$ and $\Delta, \Theta \models^{\Gamma, x:T} \{\, Q \,\}\ B\ \{\, Q' \,\}\ [\eta]$. (The identifier C_0 caters for nomenclature used in Section 5.) We must show validity of the conclusion:

$$\Delta \models^{\Gamma} \{\, P \,\}\ \mathsf{letrec}\ m(x:T) = B\ \mathsf{in}\ C_0\ \{\, P' \,\}\ [\varepsilon]\,.$$

Let μ_0 be any Γ-environment, let σ_0 be any Γ-state such that $\sigma_0 \models P$. According to the definition of validity in Part I, we must show Safety, that is, it is not the case that $\langle \mathsf{letrec}\ m(x:T) = B\ \mathsf{in}\ C_0,\ \sigma_0,\ \mu_0\rangle \overset{\Delta}{\longmapsto}{}^{*}\ \mathit{fault}$. And we must show that for any σ', if

$$\langle \mathsf{letrec}\ m(x:T) = B\ \mathsf{in}\ C_0,\ \sigma_0,\ \mu_0\rangle \overset{\Delta}{\longmapsto}{}^{*}\ \langle \mathsf{skip},\ \sigma',\ \mu_0\rangle,$$

then (Post) $\sigma' \models P'$ and (Effect) $\sigma_0 \rightarrow \sigma' \models \varepsilon$. Validity of the premises tells us about B and C_0 under hypotheses Δ, Θ. We must therefore relate $\overset{\Delta\Theta}{\longmapsto}$ to $\overset{\Delta}{\longmapsto}$. More specifically, from a given trace of $\mathsf{letrec}\ m(x:T) = B\ \mathsf{in}\ C_0$ via $\overset{\Delta}{\longmapsto}$ we must obtain traces of C_0 and B via $\overset{\Delta\Theta}{\longmapsto}$ in order to make use of the premises.

By semantics, the first step is $\langle \mathsf{letrec}\ m(x:T) = B\ \mathsf{in}\ C_0,\ \sigma_0,\ \mu_0\rangle \overset{\Delta}{\longmapsto} \langle C_0; \mathsf{elet}(m),\ \sigma_0,\ \dot{\mu}_0\rangle$ where $\dot{\mu}_0$ extends μ_0 by mapping m to $(x:T.B)$. By convention, we will use dotted names like $\dot{\mu}$ to indicate environments for which we are either proving or assuming that m is in the domain and is bound to $(x:T.B)$. (As per Lemma 4.8, m will be in each reached environment.) Continuing on, a terminating trace from $C_0; \mathsf{elet}(m)$ looks like

$$\langle C_0; \mathsf{elet}(m),\ \sigma_0,\ \dot{\mu}_0\rangle \overset{\Delta}{\longmapsto}{}^{*}\ \langle \mathsf{elet}(m),\ \sigma',\ \dot{\mu}_0\rangle \overset{\Delta}{\longmapsto}\ \langle \mathsf{skip},\ \sigma',\ \mu_0\rangle,$$

where $\mathsf{elet}(m)$ is present in all the configurations except the last. (See upper part of Figure 10.) The trace $\langle C_0; \mathsf{elet}(m),\ \sigma_0,\ \dot{\mu}_0\rangle \overset{\Delta}{\longmapsto}{}^{*}\ \langle \mathsf{elet}(m),\ \sigma',\ \dot{\mu}_0\rangle$ corresponds step by step with a trace $\langle C_0,\ \sigma_0,\ \dot{\mu}_0\rangle \overset{\Delta}{\longmapsto}{}^{*}\ \langle \mathsf{skip},\ \sigma',\ \dot{\mu}_0\rangle$ obtained by deleting $\mathsf{elet}(m)$ and noting that $\mathsf{elet}(m)$ is identified with $\mathsf{skip}; \mathsf{elet}(m)$. Conversely, any trace from $\langle C_0,\ \sigma_0,\ \dot{\mu}_0\rangle$ gives rise to one from $\langle C_0; \mathsf{elet}(m),\ \sigma_0,\ \dot{\mu}_0\rangle$, by suffixing $\mathsf{elet}(m)$ to the control state. Owing to these observations, it suffices to prove Safety, Post, and Effect for traces from $\langle C_0,\ \sigma_0,\ \dot{\mu}_0\rangle$ via $\overset{\Delta}{\longmapsto}$.

From $\langle C_0,\ \sigma_0,\ \dot{\mu}_0\rangle$, subsequent steps via $\overset{\Delta}{\longmapsto}$ are matched by steps via $\overset{\Delta\Theta}{\longmapsto}$ with same code and states, but without m in the environment, until we reach a configuration with

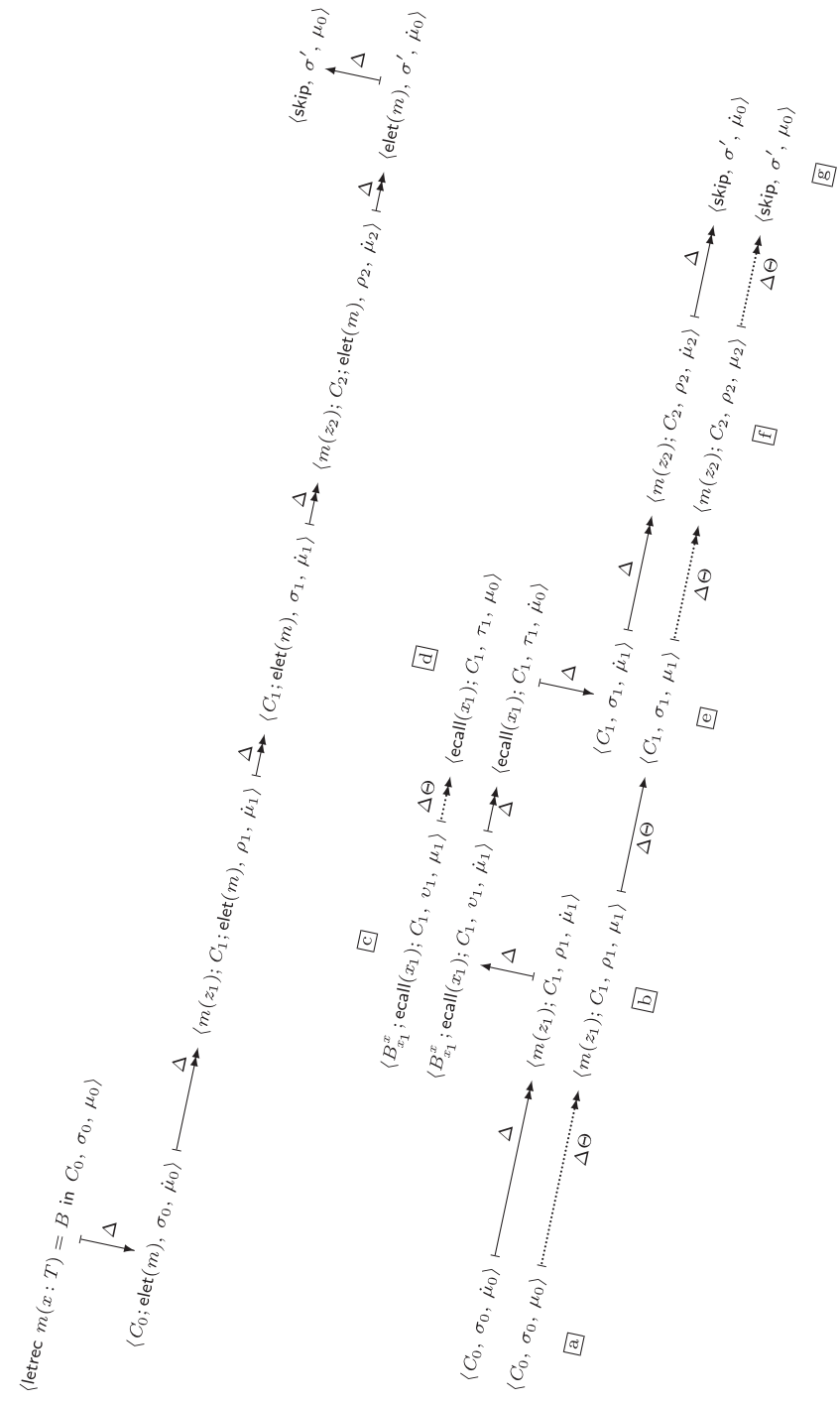

Fig. 10. Obtaining a trace of C_0 via $\overset{\Delta\Theta}{\longmapsto}$ from a trace via $\overset{\Delta}{\longmapsto}$ of letrec $m(x:T) = B$ in C_0. For reasons of typsetting we use double-head arrows to indicate transitive closure.

an active call of m. (See trace from a to b in Figure 10, where dotted arrows indicate step-by-step match.) That is, there are C_1, ρ_1, z_1, μ_1 such that

$$\langle C_0,\ \sigma_0,\ \dot{\mu}_0\rangle \overset{\Delta}{\longmapsto}{}^{*} \langle m(z_1); C_1,\ \rho_1,\ \dot{\mu}_1\rangle \qquad (*)$$
$$\text{and}\quad \langle C_0,\ \sigma_0,\ \mu_0\rangle \overset{\Delta\Theta}{\longmapsto}{}^{*} \langle m(z_1); C_1,\ \rho_1,\ \mu_1\rangle$$

and the last configurations are *matched*, meaning that $\dot{\mu}_1 = \mathit{Extend}(\mu_1, m, (x : T.B))$ and they have the same code and same state.[15] We call this a *topmost call* of m, noting that it may be reached by a chain of calls of other procedures, the first of which is a top level call in the ordinary sense. (In terms of stack implementation, what we call topmost would have bottom-most stack frame.)

From $\langle m(z_1); C_1,\ \rho_1,\ \mu_1\rangle$, computation via $\overset{\Delta\Theta}{\longmapsto}$ does not fault, nor does it p-fault, because that would contradict validity of the premise for C_0 (i.e., $\Delta, \Theta \models^{\Gamma} \{\, P \,\}\ C_0\ \{\, P' \,\}\ [\varepsilon]$). Thus, by definition of $\overset{\Delta\Theta}{\longmapsto}$ it must be that $\rho_1 \models Q^x_{z_1}$, as otherwise $m(z_1)$ would p-fault (by the last transition rule in Figure 9). Hence,

$$\langle m(z_1); C_1,\ \rho_1,\ \mu_1\rangle \overset{\Delta\Theta}{\longmapsto} \langle C_1,\ \sigma,\ \mu_1\rangle \tag{13}$$

for all σ such that $\sigma \models {Q'}^x_{z_1}$ and $\rho_1{\rightarrow}\sigma \models \eta^x_{z_1}$. To be very precise, ρ_1 and σ are states for some Γ' that extends Γ with procedures and local variables, z_1 is in $dom(\Gamma')$, $\rho_1 \models^{\Gamma'} Q^x_{z_1}$, and $\rho_1{\rightarrow}\sigma \models^{\Gamma'} \eta^x_{z_1}$. (Point e in Figure 10 shows one choice of σ, which is determined in the following discussion.)

As for the computation via $\overset{\Delta}{\longmapsto}$—about which we must prove Safety, Post, and Effect—the first step after $(*)$ is

$$\langle m(z_1); C_1,\ \rho_1,\ \dot{\mu}_1\rangle \overset{\Delta}{\longmapsto} \langle B^x_{x_1}; \mathsf{ecall}(x_1); C_1,\ \upsilon_1,\ \dot{\mu}_1\rangle, \tag{14}$$

where we look up $\dot{\mu}_1(m) = (x : T.B)$, choose a fresh identifier x_1 for the parameter, and extend the state as

$$\upsilon_1 = \mathit{Extend}(\rho_1, x_1, \rho_1(z_1)). \tag{15}$$

(Now we are at the point marked c in Figure 10.) From the validity of the premise for B, we get, by renaming of x_1 for x (i.e., Lemma 7.3):

$$\Delta, \Theta \models^{\Gamma, x_1 : T} \{\, Q^x_{x_1} \,\}\ B^x_{x_1}\ \{\, {Q'}^x_{x_1} \,\}\ [\eta^x_{x_1}]\,.$$

Then we can extend to Γ' by soundness of rule ExtendCtx (cf. Figure 15 in the sequel):

$$\Delta, \Theta \models^{\Gamma', x_1 : T} \{\, Q^x_{x_1} \,\}\ B^x_{x_1}\ \{\, {Q'}^x_{x_1} \,\}\ [\eta^x_{x_1}]\,. \tag{16}$$

(By well-formedness of the judgment for C_0, x is not free in either Δ or Θ.) Note that υ_1 is a $(\Gamma', x_1 : T)$-state; by $\rho_1 \models^{\Gamma'} Q^x_{z_1}$ from earlier, and definition (15) of υ_1, we get $\upsilon_1 \models^{\Gamma', x_1 : T} Q^x_{x_1}$ where we have made the contexts explicit for clarity. Hence, owing to (16), computation from $\langle B^x_{x_1},\ \upsilon_1,\ \mu_1\rangle$ via $\overset{\Delta\Theta}{\longmapsto}$ does not fault. If it terminates, it terminates in a configuration $\langle \mathsf{skip},\ \tau_1,\ \mu_1\rangle$ for some τ_1 such that

$$\tau_1 \models {Q'}^x_{x_1} \quad \text{and} \quad \upsilon_1{\rightarrow}\tau_1 \models \eta^x_{x_1}. \tag{17}$$

[15] We can assume such C_1 exists, without loss of generality, by introducing gratuitous skip commands so that every call occurs as the first part of some sequence.

Moreover—in the absence of any recursive invocations of m—the trace

$$\langle B^x_{x_1},\ \upsilon_1,\ \mu_1\rangle \overset{\Delta\Theta}{\longmapsto}{}^* \langle \mathsf{skip},\ \tau_1,\ \mu_1\rangle$$

is matched step-for-step by a trace

$$\langle B^x_{x_1},\ \upsilon_1,\ \dot\mu_1\rangle \overset{\Delta}{\longmapsto}{}^* \langle \mathsf{skip},\ \tau_1,\ \dot\mu_1\rangle$$

since $\overset{\Delta\Theta}{\longmapsto}$ and $\overset{\Delta}{\longmapsto}$ agree except for calls to m. (See the traces from c to d in Figure 10.) Thus, continuing from (14), we have

$$\langle B^x_{x_1}; \mathsf{ecall}(x_1); C_1,\ \upsilon_1,\ \dot\mu_1\rangle \overset{\Delta}{\longmapsto}{}^* \langle \mathsf{ecall}(x_1); C_1,\ \tau_1,\ \dot\mu_1\rangle.$$

After another step under $\overset{\Delta}{\longmapsto}$, we reach $\langle C_1,\ \sigma_1,\ \dot\mu_1\rangle$ where $\sigma_1 = \tau_1 \restriction x_1$.

We aim to show that σ_1 is one of the possible states σ on the right-hand side of (13), so that $\langle C_1,\ \sigma_1,\ \dot\mu_1\rangle$ matches one of the possible configurations $\langle C_1,\ \sigma,\ \mu_1\rangle$ reached by $\overset{\Delta\Theta}{\longmapsto}$ as described earlier (see the point marked e in Figure 10). From (17), we get $\sigma_1 \models^{\Gamma'} Q'^x_{z_1}$ and $\rho_1{\to}\sigma_1 \models^{\Gamma'} \eta^x_{z_1}$, by careful calculation with substitutions and using that $\mathsf{wr}\, x$ does not occur in η (because the specification is well formed). This completes the argument that σ_1 is among the outcomes σ from the trace via $\overset{\Delta,\Theta}{\longmapsto}$.

So far, we have shown that the computation of C_0, up through the first completed invocation of m, under $\overset{\Delta}{\longmapsto}$ (and from the environment $\dot\mu_1$ using the procedure body B for m) leads to a configuration that is also a possible configuration from C_0 under $\overset{\Delta\Theta}{\longmapsto}$ with m as a context procedure (and thus not in the environment). (See point e in Figure 10.) We proceed to extend the matching through the rest of the trace for C_0 by induction on the number of (topmost) calls to m. For example, the next phase, up to the next call of m, looks like

$$\langle C_1,\ \sigma_1,\ \dot\mu_1\rangle \overset{\Delta}{\longmapsto}{}^* \langle m(z_2); C_2,\ \sigma_2,\ \dot\mu_2\rangle$$

and is matched by a trace via $\overset{\Delta\Theta}{\longmapsto}{}^*$ up to matching $\langle m(z_2); C_2,\ \sigma_2,\ \mu_2\rangle$. (Point f.) If the computation under $\overset{\Delta}{\longmapsto}$ terminates then its last configuration is of the form $\langle \mathsf{skip},\ \sigma',\ \mu_0\rangle$ and the trace under $\overset{\Delta\Theta}{\longmapsto}$ also reaches $\langle \mathsf{skip},\ \sigma',\ \mu_0\rangle$. (Point g, which is reached from f after some number of completed topmost calls of m, interspersed with other steps.) That being so, we can appeal to validity of the premise for C_0 to get the Post condition $\sigma' \models P$ and Effect $\sigma{\to}\sigma' \models \varepsilon$. This concludes the soundness proof of rule LINK0, if there are no recursive calls to m from B.

Remark 4.12. One might hope to formulate the proof as a simulation between the semantics of C_0 via $\overset{\Delta\Theta}{\longmapsto}$ and its semantics via $\overset{\Delta}{\longmapsto}$, but this approach does not seem fruitful. The Safety property involves intermediate steps (as does the Encap property to come in Section 6); these steps are for both code of C_0 and code of B so we need to treat the computation via $\overset{\Delta}{\longmapsto}$ as spliced together traces of both C_0 and B. Also, the simulation approach does not help with recursion. So we take the somewhat ad hoc approach sketched here.

In case there are recursive calls, we cannot simply appeal to the premise for $B^x_{x_1}$, that is, to (16), to reason about execution of $B^x_{x_1}$ under $\overset{\Delta}{\longmapsto}$ with m in the environment, because the execution of $B^x_{x_1}$ under the two transition relations $\overset{\Delta}{\longmapsto}$ and $\overset{\Delta\Theta}{\longmapsto}$ no longer

matches. The proof rule embodies the usual induction pattern for partial correctness, that is, induction on recursion depth [Hoare 1971]; we need a matching induction in the soundness proof. In particular, we want to be able to assume that the inner call of m, executing a substitution instance of B under $\stackrel{\Delta}{\longmapsto}$ (with m in the environment), is correct.

To make this precise, we consider a variant of the semantics in which configurations carry a bound on calling depth. It gets decreased in the transition for invocation of a procedure in the environment, and such invocations get stuck if the bound is zero. For any trace in the non-bounded semantics, there is a bound such that the bounded semantics runs through the same series of states. We prove that B satisfies its specification by showing that in the bounded semantics, for all values of the bound. The proof is by induction on the bound. In the induction case, the argument is similar to the one above concerning C_0, up to the point where B makes a recursive call to m. At that point, the bound gets decremented and we can appeal to the induction hypothesis which says that the inner call is correct. So the premise of the rule is used to reason about execution of B in each topmost invocation of m, but that premise has hypothesis Θ that recursive calls are correct, and the hypothesis is discharged by induction on depth of recursion. Recall that what we are calling topmost calls of a procedure m are not necessarily top level procedure calls; m may be invoked from the body of some other procedure. In case several procedures are linked simultaneously, this means a topmost call of any one of them, cf. Remark 5.4.

On casual reading, the term "bounded" may give the mistaken impression that we are considering termination. We are not. As an extreme case, if the specification of m is unsatisfiable we can still prove that B implements m, where B is simply the call $m(x)$. This diverges everywhere but neither faults nor writes so it satisfies premise $\Delta, \Theta \vdash \{\, Q \,\}\ B\ \{\, Q' \,\}\ [\eta]$.

5. TECHNICAL RESULTS FOR SOUNDNESS OF THE LINK RULE

Section 4.4 explains the soundness argument for a procedure linking rule. The intricacy of the sketch makes clear that a detailed proof is needed, all the more so once dynamic boundaries are added (in Section 6) to the semantics of correctness judgments. This section develops some tools needed for the detailed proof of the link rule (Section 7.6), and for nothing else. The section is here because it depends on nothing more than the semantics defined in Section 4, but on first reading we suggest skipping to Section 6.

LEMMA 5.1. *Suppose Q is well formed in Γ and ρ is a Γ-state. Suppose $\Gamma(x) = T$, $\Gamma(z) = T$, and $u \notin dom(\Gamma)$. Let $\upsilon = \mathit{Extend}(\rho, u, \rho(z))$. If $\rho \models^{\Gamma} Q^x_z$, then $\upsilon \models^{\Gamma, u:T} Q^x_u$.*

The proof is an exercise in substitution, using that Q does not depend on u.

The following is similar to the correspondence Lemma 4.9, but this one lets us compare the semantics when a particular procedure is in the environment versus when it is in the context.

LEMMA 5.2 (SPECIAL CORRESPONDENCE). *Consider any $m, C, \sigma, \dot\mu, \Delta$ such that $\langle C,\ \sigma,\ \dot\mu\rangle$ is compatible with Δ and $m \in dom(\dot\mu)$. (Here C can be an extended command.) Let $\mu = \dot\mu \restriction m$ and suppose Θ specifies m. Suppose C has no* elet(m)*, so that $\langle C,\ \sigma,\ \mu\rangle$ is compatible with Δ, Θ (noting that by compatibility, C has no* letrec *binding of m). Suppose $\mathit{Active}(C)$ is not a call to m. Then, for any $C', \sigma', \dot\mu'$, we have $\langle C,\ \sigma,\ \dot\mu\rangle \stackrel{\Delta}{\longmapsto} \langle C',\ \sigma',\ \dot\mu'\rangle$ if and only if $\langle C,\ \sigma,\ \mu\rangle \stackrel{\Delta\Theta}{\longmapsto} \langle C',\ \sigma',\ \mu'\rangle$, where $\mu' = \dot\mu' \restriction m$. Moreover, $\langle C,\ \sigma,\ \dot\mu\rangle \stackrel{\Delta}{\longmapsto}$ fault iff $\langle C,\ \sigma,\ \mu\rangle \stackrel{\Delta\Theta}{\longmapsto}$ fault and* mutatis mutandis *for p-fault.*

PROOF. By cases on $\mathit{Active}(C)$. When $\mathit{Active}(C)$ is not a call to m, the semantics does not depend on Θ or $\dot{\mu}(m)$. □

Call-Depth Bounded Semantics. A *bounded configuration* has the form $\langle C,\ \sigma,\ \mu\rangle^k$ where k is a natural number. One can think of k as the size of the available stack space. A computation will get stuck (and does not fault) if it attempts to invoke a procedure when k is 0.

The transition relation on depth-bounded configurations is written $\overset{\Delta}{\longmapsto}$ just like for standard configurations. It is defined so that the bound is decreased in the invocation step and increased when the end-marker of the procedure body is reached:

$$\frac{k > 0 \qquad \mu(m) = (x : T.C) \qquad x' \notin \mathit{Dom}(\sigma) \qquad C' = C^x_{x'}}{\langle m(z),\ \sigma,\ \mu\rangle^k \overset{\Delta}{\longmapsto} \langle C'; \mathsf{ecall}(x'),\ \mathit{Extend}(\sigma, x', \sigma(z)),\ \mu\rangle^{k-1}}$$

$$\langle \mathsf{ecall}(x),\ \sigma,\ \mu\rangle^k \overset{\Delta}{\longmapsto} \langle \mathsf{skip},\ \sigma \restriction x,\ \mu\rangle^{k+1}.$$

The bound needs to be propagated in one of the transitions for sequence:

$$\frac{\langle C,\ \sigma,\ \mu\rangle^k \overset{\Delta}{\longmapsto} \langle C',\ \sigma',\ \mu'\rangle^{k'}}{\langle C\,; D,\ \sigma,\ \mu\rangle^k \overset{\Delta}{\longmapsto} \langle C'\,; D,\ \sigma',\ \mu'\rangle^{k'}}.$$

In all other cases, the transition rule is the same as for non–depth-bounded configurations except that a single bound k is added uniformly to every configuration in the rule. For example, the transition for ending the scope of a local variable is

$$\langle \mathsf{evar}(x),\ \sigma,\ \mu\rangle^k \overset{\Delta}{\longmapsto} \langle \mathsf{skip},\ \sigma \restriction x,\ \mu\rangle^k.$$

Note that, if $\langle C,\ \sigma,\ \mu\rangle^k \overset{\Delta}{\longmapsto} \langle C',\ \sigma',\ \mu'\rangle^j$, then the sum of k and the number of ecall-commands in C is the same as the sum of j and the number of ecall-commands in C'. Owing to condition $k > 0$ in the rule for invocation, the invocation of an environment procedure gets "stuck" if the bound is 0. This is the only change to liveness, cf. Lemma 4.11.

LEMMA 5.3 (DEPTH-BOUNDED AND NORMAL SEMANTICS). *Consider any* Δ *and any compatible configuration* $\langle C,\ \sigma,\ \mu\rangle$.

(*1*) $\langle C,\ \sigma,\ \mu\rangle \overset{\Delta}{\longmapsto}{}^* \langle C',\ \sigma',\ \mu'\rangle$ *iff there are* k, j *such that* $\langle C,\ \sigma,\ \mu\rangle^k \overset{\Delta}{\longmapsto}{}^* \langle C',\ \sigma',\ \mu'\rangle^j$
(*2*) $\langle C,\ \sigma,\ \mu\rangle \overset{\Delta}{\longmapsto}{}^*$ *fault iff there is some* $k \geq 0$ *such that* $\langle C,\ \sigma,\ \mu\rangle^k \overset{\Delta}{\longmapsto}{}^*$ *fault.*
(*3*) mutatis mutandis *for p-fault.*

PROOF. For (1), left implies right: Let k be the maximum number of ecall commands in the configurations of the given trace. Show by induction on steps that from initial configuration $\langle C, \sigma, \mu\rangle^k$ that there is a suitable bound to add for each successive configuration. For the converse, show by induction on steps and cases on transition rules that erasing bounds yields a standard computation. Now (2) and (3) follow, as the bound has no influence on faulting. □

There is also a depth-bounded version of the special correspondence Lemma 5.2, the gist of which is: $\langle C,\ \sigma,\ \dot{\mu}\rangle^k \overset{\Delta}{\longmapsto} \langle C',\ \sigma',\ \dot{\mu}'\rangle^l$ iff $\langle C,\ \sigma,\ \mu\rangle^k \overset{\Delta\Theta}{\longmapsto} \langle C',\ \sigma',\ \mu'\rangle^l$ for all k, l, under conditions suitably adapted from that Lemma.

Decomposing Traces into Phases. For proof of the LINK rule, we need to break computation sequences into "phases" comprised of topmost invocations of some procedures of interest, alternating with other steps. The intuitive idea is not surprising but we need to spell out some details in order to establish notation for later use. Keep in mind that we are often considering nonmaximal traces, that is, they may end in a configuration for which successors exist. Also, a topmost invocation of a specific procedure may well be nested inside several levels of invocations of other procedures. We take some care with nomenclature in the formal details, as this will be helpful for the proof of the LINK rule.

First, we describe the basic idea as it applies to calls of both context procedures and environment procedures. Suppose $\langle C_0,\ \sigma_0,\ \mu_0\rangle \overset{\Delta}{\longmapsto}{}^* \langle D,\ \tau,\ \nu\rangle$ and suppose m is a procedure name (which might be declared in Δ or in μ_0). Then there is some number $n \geq 0$ of completed topmost invocations of m, possibly followed by an incomplete topmost invocation if D is not skip. In more detail: for $0 < i \leq n$, there are configurations $\langle C_i,\ \sigma_i,\ \mu_i\rangle$, variables z_i (the arguments of those calls), and states ρ_i (the initial states of those calls) such that

— For all i $(0 < i \leq n)$, we have $\langle C_{i-1},\ \sigma_{i-1},\ \mu_{i-1}\rangle \overset{\Delta}{\longmapsto}{}^* \langle m(z_i); C_i,\ \rho_i,\ \mu_i\rangle$ without any intermediate configurations in which m is the active command. (For example, from point a to point b in Figure 10, and also from e to f there.)

— For all i $(0 < i \leq n)$, we have $\langle m(z_i); C_i,\ \rho_i,\ \mu_i\rangle \overset{\Delta}{\longmapsto}{}^* \langle C_i,\ \sigma_i,\ \mu_i\rangle$. (For example, from point b to point e in Figure 10.) Note that the final environment, μ_i, is the same as at the beginning of the invocation.

— $\langle C_n,\ \sigma_n,\ \mu_n\rangle \overset{\Delta}{\longmapsto}{}^* \langle D,\ \tau,\ \nu\rangle$ without any completed topmost invocations of m.

The first item describes (possibly empty) phases of the trace in which control is not inside any invocation of m, but leading up to a call of m. In case $i = 1$, this is the phase of the trace before the first invocation of m (e.g., from a to b in Figure 10).

The second item describes phases consisting of a *completed topmost invocation* of m. In case m is in the context, it is a single step. In case m is in the environment, there are at least two steps (exactly two, in case the body of m is skip, namely from $m(z_i); C_i$ to ecall$(\cdots)$; C_i to C_i). The body of m may make recursive invocations of m, which are complete but not topmost.

The third item allows that a trace ends by invoking m but not yet finishing that. Nested within, there may be complete (recursive) invocations of m possibly trailing incomplete invocations; what matters is that the topmost one is not complete. What this looks like can be made more precise by distinguishing between context procedures and environment procedures, as we do in this section,

Here is the pattern, with $n = 2$:

$\langle C_0,\ \sigma_0,\ \mu_0\rangle$

$\overset{\Delta}{\longmapsto}{}^* \langle m(z_1); C_1,\ \rho_1,\ \mu_1\rangle$	without any m calls
$\overset{\Delta}{\longmapsto}{}^* \langle C_1,\ \sigma_1,\ \mu_1\rangle$	first topmost m invocation (body of m, to completion)
$\overset{\Delta}{\longmapsto}{}^* \langle m(z_2); C_2,\ \rho_2,\ \mu_2\rangle$	without any m calls
$\overset{\Delta}{\longmapsto}{}^* \langle C_2,\ \sigma_2,\ \mu_2\rangle$	second topmost m invocation
$\overset{\Delta}{\longmapsto}{}^* \langle D,\ \tau,\ \nu\rangle$	without any (completed) m calls

Note that there may be many intermediate states between the states σ_{i-1} and σ_i in the preceding conditions. To avoid confusion, we refrain from using subscripted indices for step-by-step numbering of configurations.

There is no loss of generality in assuming that each call $m(z_i)$ occurs in a sequence followed by some C_i, because semicolon has unit element skip (see Section 4.2).

Generalizing from a single method m, we can consider a set X of procedures, say those declared in a particular module, which might be mutually recursive. A trace can be decomposed to a series of topmost invocations of any m in X. This is needed to prove the LINK rule in the general case, but for readability we will prove LINK for the special case of a single procedure. So we state the following results just for that case.

Remark 5.4. In variable context with non-Local variable $res : nat$, consider the following.

$$\begin{array}{l}\mathsf{letrec}\ p(x:nat) = even(x); odd(x+1);\\ \quad even(x:nat) = \mathsf{if}\ x > 1\ \mathsf{then}\ odd(x-1)\ \mathsf{else\ if}\ x = 0\ \mathsf{then\ skip\ else}\ diverge;\\ \quad odd(y:nat) = \mathsf{if}\ x > 1\ \mathsf{then}\ even(x-1)\ \mathsf{else\ if}\ x = 1\ \mathsf{then\ skip\ else}\ diverge\\ \mathsf{in}\ p(2); p(4)\end{array}$$

We are interested in the "topmost invocations of procedures in the set $\{even, odd\}$". The only "top level invocations" are the two calls of p. But there are four topmost invocations of the designated procedures: $even(2)$ and then $odd(3)$ then $even(4)$ then $odd(5)$. There are several other invocations of $even$ and odd, but those are recursive calls, not topmost invocations.

For a trace to have an *incomplete invocation* of a context procedure m means that a configuration is reached where a call $m(z)$ is the active command, that is, the control state is of the form $m(z); C$, but there is no successor in the trace. In this case the possible successors would be either *fault* or a configuration with control state C. If instead m is an environment procedure, an incomplete invocation takes the initial step to $B^x_u; \mathsf{ecall}(u); C$ where m is bound to $(x : T.B)$,[16] and may take successive steps possible even to the point where the control state is $\mathsf{ecall}(u); C$—but no further. That is, it has not done item (5.5) in the following.

LEMMA 5.5 (DECOMPOSITION FOR ENVIRONMENT PROCEDURES). *Suppose* $\mu_0(m) = (x : T.B)$ *and* $\langle C_0, \sigma_0, \mu_0\rangle^{k_0}$ *is compatible with* Δ. *Suppose* $\langle C_0, \sigma_0, \mu_0\rangle^{k_0} \overset{\Delta}{\longmapsto}{}^* \langle D, \tau, \nu\rangle^j$. *Then, there is* $n \geq 0$ *and, for all* i $(0 < i \leq n)$, *there are configurations* $\langle C_i, \sigma_i, \mu_i\rangle^{k_i}$, *variables* z_i *and* x_i, *states* ρ_i, τ_i, *and* υ_i *such that for all* i $(0 < i \leq n)$

(1) $\langle C_{i-1}, \sigma_{i-1}, \mu_{i-1}\rangle^{k_{i-1}} \overset{\Delta}{\longmapsto}{}^* \langle m(z_i); C_i, \rho_i, \mu_i\rangle^{k_i}$ *without any intermediate configurations in which* m *is the active command*

(2) $\langle m(z_i); C_i, \rho_i, \mu_i\rangle^{k_i} \overset{\Delta}{\longmapsto} \langle B^x_{x_i}; \mathsf{ecall}(x_i); C_i, \upsilon_i, \mu_i\rangle^{k_i - 1}$ *and* $\upsilon_i = \mathit{Extend}(\rho_i, x_i, \rho_i(z_i))$ *(note that* x_i *is a fresh parameter name)*

(3) $\langle B^x_{x_i}, \upsilon_i, \mu_i\rangle^{k_i-1} \overset{\Delta}{\longmapsto}{}^* \langle \mathsf{skip}, \tau_i, \mu_i\rangle^{k_i-1}$ *and hence by semantics*[17]
$\langle B^x_{x_i}; \mathsf{ecall}(x_i); C_i, \upsilon_i, \mu_i\rangle^{k_i-1} \overset{\Delta}{\longmapsto}{}^* \langle \mathsf{ecall}(x_i); C_i, \tau_i, \mu_i\rangle^{k_i-1}$

(4) $\langle \mathsf{ecall}(x_i); C_i, \tau_i, \mu_i\rangle^{k_i-1} \overset{\Delta}{\longmapsto} \langle C_i, \sigma_i, \mu_i\rangle^{k_i}$ *and* $\sigma_i = \tau_i \restriction x_i$

[16] In case B is skip, the initial step goes directly from $m(z); C$ to $\mathsf{ecall}(u); C$.

[17] This is formulated to ensure that τ_i is the state just after execution of $B^x_{x_i}$. By semantics, it is clear that $\langle B, \sigma, \mu\rangle \longmapsto^* \langle \mathsf{skip}, \sigma', \mu'\rangle$ implies $\langle B; C, \sigma, \mu\rangle \longmapsto^* \langle C, \sigma', \mu'\rangle$. Beware, however, that $\langle B; C, \sigma, \mu\rangle \longmapsto^* \langle C, \sigma', \mu'\rangle$ does not imply $\langle B, \sigma, \mu\rangle \longmapsto^* \langle \mathsf{skip}, \sigma', \mu'\rangle$. (For example, C could be a loop with body B and σ' reached only after several iterations.) But it does imply there are τ, μ'' with $\langle B, \sigma, \mu\rangle \longmapsto^* \langle \mathsf{skip}, \tau, \mu''\rangle$ and $\langle C, \tau, \mu''\rangle \longmapsto^* \langle C, \sigma', \mu'\rangle$.

(5) $\langle C_n, \sigma_n, \mu_n\rangle^{k_n} \overset{\Delta}{\longmapsto}{}^* \langle D, \tau, \nu\rangle^j$ *without any completed invocations of* m*—but allowing a topmost call that is incomplete (and which may contain nested invocations of* m*, both complete and not).*

This exhibits the role of depth bounds: the procedure body B is executed with a bound, $k_i - 1$, smaller than that of its calling context.

Regarding item (5.5), note that the bounds k_{i-1} and k_i may differ: although there are no calls to m in these configurations, there may be calls to other procedures (decreasing the bound) or ecalls for the parameters of other procedures (increasing the bound).

In light of Lemma 5.3, we have the same decomposition for normal semantics: that is, Lemma 5.5 holds if all depth bounds are erased.

If there is an incomplete topmost call, then item (5) can be further decomposed to the form

$$\langle C_n, \sigma_n, \mu_n\rangle^{k_n} \overset{\Delta}{\longmapsto}{}^* \langle m(z); C', \sigma', \mu'\rangle^l \overset{\Delta}{\longmapsto}{}^* \langle D, \tau, \nu\rangle^j$$

for some z, C', σ', μ', l. We will need to refer to the case where there are zero steps following the middle configuration and so D is $m(z); C'$.

Definition 5.6 (m*-truncated*). For a trace $\langle C, \sigma, \mu\rangle \overset{\Delta}{\longmapsto}{}^* \langle D, \tau, \nu\rangle$ to be m*-truncated* means: If there is an incomplete invocation of m, then its call is the active command in the last configuration. That is, D is $m(z); C'$ for some z and C'.

A trace that is not m-truncated has m in the environment, not in Δ. It takes the form

$$\langle C, \sigma, \mu\rangle \overset{\Delta}{\longmapsto}{}^* \langle m(z); C', \rho, \mu'\rangle \overset{\Delta}{\longmapsto} \langle B_u^x; \mathsf{ecall}(u); C', \upsilon, \mu'\rangle \overset{\Delta}{\longmapsto}{}^* \langle D, \tau, \nu\rangle$$

for some z, C', u, μ', x, T, B with $\mu'(m) = (x : T.B)$ and $\upsilon = \mathit{Extend}(\rho, u, \rho(z))$. And either

— $D = A; \mathsf{ecall}(u); C'$ for some A such that $\langle B_u^x, \upsilon, \mu'\rangle \overset{\Delta}{\longmapsto}{}^* \langle A, \tau, \nu\rangle$, or

— $D = \mathsf{ecall}(u); C'$ and $\langle B_u^x, \upsilon, \mu'\rangle \overset{\Delta}{\longmapsto}{}^* \langle \mathsf{skip}, \tau, \nu\rangle$

To deal with the general case of linking a set X of procedures simultaneously, we would use the notion of X-truncated, meaning: m-truncated for all m in X.

6. 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 Section 3.2. For the client in Eq. (1) of Section 2.1, 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 (Figure 9). Second-order framing is about a rely-guarantee relationship between a program component—a "module"—and a command that uses procedures of the module. The module relies on good behavior by the client: that it respects an encapsulation boundary and thus unwittingly preserves the hidden invariant. In return, the module guarantees the behavior specified in Δ. Our rely condition is a read effect, 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.

For a dynamic boundary to be useful it should frame the invariant to be hidden. For example, in Section 3.2 we defined for module SET the effect θ_{set} which frames I_{set}. Checking the framing judgment is a proof obligation on the module, but the dynamic boundary must also be well formed for the client, which is obliged to respect it.

```
module Lib//boundary is empty
  q(c : Ctr) requires true ensures true modifies c.f

module OD
  global x : Ctr;
  boundary x, x.val
  pri-inv x.val > 0
  odd(z : int)
    requires z ≥ 0 ∧ x ≠ y
    modifies x.val, y.val, x.f, y.f
  { x.val := x.val + 1; if z = 1 then skip else q(x); even(z − 1) fi}

module EV
  global y : Ctr;
  boundary y, y.val
  pri-inv y.val > 0
  even(z : int)
    requires z ≥ 0 ∧ x ≠ y
    modifies x.val, y.val, x.f, y.f
  { y.val := y.val + 1; if z = 0 then skip else q(y); odd(z − 1) fi}
```

Fig. 11. Modules Lib, EV and OD.

6.1. A Lightweight Formalization of Modules

For a lightweight abstract syntax of modules, we postulate

— a set $ModName$,
— a function $mdl : ProcName \to ModName$ to associate each procedure with its module,
— a preorder $\preceq$ on $ModName$, and
— a function bnd from $ModName$ to read effects.

We call $bnd(M)$ the *dynamic boundary* of M. The intended interpretation of $M \preceq N$, read "M imports N", is that $N = M$ or N is directly or transitively imported by M in the sense that some procedure in M calls, directly or indirectly, some procedure in N. In order to allow callbacks that cross module boundaries, $\preceq$ is not required to be antisymmetric. We write $M \in \Delta$ to abbreviate that there is some procedure m in Δ with $mdl(m) = M$.

We have not formalized an association of global variables with modules; rather, Γ should include them all. This streamlines the formal development, though it creates clutter in some examples: We have to resort to dynamic boundaries to restrict access in cases where module-scoped global variables would have sufficed.

Hypothetical correctness judgments in the logic have the form given in (9) but with the addition of a module name:

$$\Delta \vdash^{\Gamma}_{M} \{ P \}\ C\ \{ P' \}\ [\varepsilon]\,. \tag{18}$$

The idea is that C may appear in the body of a procedure of module M. For the judgment (18) to be well formed, it must satisfy the conditions following (9); in addition, for each M in Δ the boundary $bnd(M)$ must be well formed in $\Gamma \restriction (Locals \cup SpecOnlyVar)$.

Figure 11 shows a simple example with intermodule callbacks, using illustrative concrete syntax. For this example, we choose $\preceq$ to be the reflexive, transitive closure of $OD \preceq Lib$, $OD \preceq EV$, $EV \preceq Lib$, $EV \preceq OD$.

As a guide for understanding certain side conditions in the proof rules, we briefly consider a slightly heavier formalization of modules. The heavier formalization uses

typing judgment $\Gamma \vdash_M C$ for code to be used in procedures of M. The typing rule for calls requires $M \preceq mdl(m)$ for $\Gamma \vdash_M m(z)$, so M will "import" any module whose procedures it calls. The typing rule for letrec ensures that all of the procedures of a given module are linked at once, and linked to a client of a different module. Of course, to allow for mutual recursion between procedures of different modules, the procedures of more than one module may be linked simultaneously. Here is the rule in the special case where modules N_0 and N_1 each provide a single procedure.

$$\frac{\begin{array}{c}\Gamma' = m_0 : (x_0 : T_0), m_1 : (x_1 : T_1) \qquad \Gamma \restriction \mathit{Locals}, \Gamma', x_i : T_i \vdash B_i \text{ for all } i \\ \Gamma, \Gamma' \vdash C \qquad x_i \in \mathit{Locals} \text{ and } B_i \text{ is letrec-free, for all } i \\ mdl(m_i) = N_i \text{ and } N_i \neq M \text{ for all } i \qquad mdl(p) \not\preceq N_i \text{ for all } p \text{ in } \Gamma \text{ and all } i\end{array}}{\Gamma \vdash_M \mathsf{letrec}\ m_0(x_0 : T_0) = B_0;\ m_1(x_1 : T_1) = B_1\ \mathsf{in}\ C}. \tag{19}$$

(Here "all i" means $i \in \{0, 1\}$, and N_0 need not be distinct from N_1.) This includes the conditions in the actual typing rule (see (7) in Section 4), for example, each B_i may call both m_0 and m_1 as well as procedures of the ambient context Γ. In addition, it requires that the client's module M is distinct from the modules N_0 and N_1. Furthermore, Γ has no other procedures of N_0 and N_1. A closed program has a main command for some module name Main that is not in the range of mdl. The actual rule (7) for letrec already embodies scoping of procedure names. Both (7) and (19) allow the extreme case where all modules are linked simultaneously, as well as the separate linking in the absence of mutual recursion.

There are two reasons why we do not adopt the heavier formalization of modules. First, if the typing judgment were to depend on modules it would complicate the semantics, for example, Definition 4.3 of well-formed configurations, which in turn would complicate the soundness proofs. The second reason is that, as we discuss in connection with the CALL rule in Section 7, there may be more than one choice of $\preceq$ that yields sound proofs. The price we pay for the lighter formalization is that some proof rules have side conditions conditions like $N_i \neq M$ and $mdl(p) \not\preceq N_i$ in (19).

Remark 6.1. The judgment form is a bit odd in that the procedure specifications are explicit, but the grouping of procedures into modules, and the associated dynamic boundaries are not explicit. One could go further and assume a fixed association of specifications with procedures, but we are interested in reasoning principles that involve changed specifications, for example, rules LINK and SOF. For the same reason, we do not formalize the association of invariants with modules. Another alternative is to include bnd etc. explicitly in the judgment form, but in this article we do not have reasoning principles that change dynamic boundaries or the import structure.

In Naumann and Banerjee [2010], the grouping of some procedures into a module with boundary δ is expressed as $\Delta\langle\delta\rangle$ and hypotheses are a list $\Delta_1\langle\delta_1\rangle, \ldots, \Delta_n\langle\delta_n\rangle$. This is not quite rich enough to handle interesting situations with nested modules.

Remark 6.2. At several points, we have remarked on the practical value of data groups. Using modules, they could be formalized along the following lines. Inside some module, with module-scoped fields f and g, the declaration "group d includes f, g" would declare d to be a group that abstracts f and g. For example, in the Observer example from Section 2.3 the fields obs, sub, $nxto$ could be included in a data group dg.

An effect wr H`d would license writes of $o.f$ for $o \in H$. Abstraction from module-scoped global variables could be done as well. If the concrete fields of distinct data groups are distinct, then data groups would not be needed in formulas (i.e., we only use data groups for their l-value). For example, the separator of rd G`$d$ and wr $H$`d would be $G \# H$, so it is G and H that a client needs to reason about. Inside the module, the

separator would need to take the group declaration into account, for example, $\mathsf{rd}\, G \text{`} d \not\perp \mathsf{wr}\, H \text{`} f = G \# H$ if f in $group(d)$.

6.2. The Mismatch Rule

The following derived rule embodies Hoare's mismatch in the special case of a single procedure specification Θ, where $\Theta = \{Q\}m(x:T)\{Q'\}[\eta]$ and $N = mdl(m)$.

$$\textsc{Mismatch}\ \frac{\begin{array}{c}\Delta, \Theta \vdash_M \{\, P \,\}\ C\ \{\, P' \,\}\ [\varepsilon] \\ P \Rightarrow I \qquad \Delta, (\Theta \between I) \vdash_N \{\, Q \wedge I \,\}\ B\ \{\, Q' \wedge I \,\}\ [\eta] \\ \vdash bnd(N)\ \mathsf{frm}\ I \qquad N \neq M \qquad mdl(p) \not\preceq N\ \text{for all } p \text{ in } \Delta\end{array}}{\Delta \vdash_M \{\, P \,\}\ \mathsf{letrec}\ m = B\ \mathsf{in}\ C\ \{\, P' \,\}\ [\varepsilon]}.$$

The side conditions $N \neq M$ and $mdl(p) \not\preceq N$ may be viewed as mere syntax, ensuring that C is the full scope of use of the procedure m that comprises module N (as discussed above in connection with (19)). The important feature of rule MISMATCH is that the client C is obliged to respect $bnd(N)$—and also $bnd(L)$ for all other modules L in Δ—but does not see the hidden invariant I. In practical terms, reasoning about C is done in the scope of module M, in which I is not visible. Reasoning about the side conditions, and about B, is done in the scope of N. 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 $bnd(L)$ for L in Δ, but it is not required (or likely) to respect $bnd(N)$. The obligation on B refers to context $\Theta \between I$, not Θ; this is relevant if B recursively invokes m (or, in the general case, other procedures of N). The operation $\between I$ conjoins a formula I to pre- and postconditions of specifications:

$$(\{\, Q \,\}\ m(x:T)\ \{\, Q' \,\}\ [\eta]\ ) \between I \quad \widehat{=} \quad \{\, Q \wedge I \,\}\ m(x:T)\ \{\, Q' \wedge I \,\}\ [\eta]\,.$$

Typical formalizations of data abstraction include a command for initialization [He et al. 1986], so a closed client program takes the form $\mathsf{letrec}\ m = B\ \mathsf{in}\ (init; C)$. 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 (Section 2.1), P in rule MISMATCH can be instantiated to have a conjunct $pool = \varnothing$ which is suitable to be declared in the module interface. Note that $pool = \varnothing \Rightarrow I_{set}$ is valid.

Remarkably, there is a simple interpretation of judgment (18) that captures the idea that C respects a boundary $bnd(M)$.

> No step of C's execution may write locations designated by $bnd(M)$ unless it is a step of a context procedure m with $mdl(m) \preceq M$.

Those locations are determined by interpreting $bnd(M)$ in the pre-state of that step.

Before turning to the formal details, we discuss this proof obligation.

6.3. Verifying a Client of *SET*

Using the public specifications of the four methods of SET, it is straightforward to prove that the client in Eq. (1) establishes $b = false$. But there is an additional obligation, that every step respects the dynamic boundary θ_{set}. Consider the assignment $n.val := 1$ in (1), which is critical because I_{set} depends on field val. Since it is not a call to a procedure of the module, we need to show the dynamic boundary is not crossed. Formally this is similar to the condition in rule FRAME and it is expressed using $\not\perp$. The effect of $n.val := 1$ is $\mathsf{wr}\, n.val$ and it must be shown to be outside the boundary θ_{set}. By definition of $\not\perp$, we have that $\theta_{set} \not\perp \mathsf{wr}\, n.val$ is $\{n\} \# pool \wedge \{n\} \# pool \text{`} rep$, which

$$\dfrac{\dfrac{\Delta, (\Theta \circledcirc I) \vdash_N \{Q \wedge I\}\ B\ \{Q' \wedge I\}\ [\eta] \qquad \dfrac{\Delta, \Theta \vdash_M \{P\}\ C\ \{P'\}\ [\varepsilon] \qquad \vdash bnd(N)\ \mathsf{frm}\ I}{\Delta, (\Theta \circledcirc I) \vdash_M \{P \wedge I\}\ C\ \{P' \wedge I\}\ [\varepsilon]}\ \text{SOF}}{\Delta \vdash_M \{P \wedge I\}\ \mathsf{letrec}\ m = B\ \mathsf{in}\ C\ \{P' \wedge I\}\ [\varepsilon]}}{\Delta \vdash_M \{P\}\ \mathsf{letrec}\ m = B\ \mathsf{in}\ C\ \{P'\}\ [\varepsilon]}$$

Fig. 12. Derivation of rule MISMATCH, where Θ is a single specification $\{Q\}\ m\ \{Q'\}\ [\eta]$, with $mdl(m) = N$, and we elide the parameter of m. Rule SOF adds I to the specifications for C. Then rule LINK combines the judgments for B and C, followed by CONSEQ using the side condition $P \Rightarrow I$ of MISMATCH. Two side conditions for SOF are in rule MISMATCH: $N \neq M$, $mdl(p) \not\leq N$ for all p in Δ. A third, $N \in \Theta$, holds by definition of Θ and $mdl(m) = N$.

simplifies to $n \notin pool \wedge n \notin pool`rep$. 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 $n \notin pool`rep$ is more interesting. Note that I_{set} implies

$$R \mathrel{\widehat{=}} pool`rep`own \subseteq pool \setminus \{\mathsf{null}\}$$

The client does not update the default value, null, of $n.own$. Together, R and $n.own = \mathsf{null}$ imply $n \notin pool`rep$. Rule CXTINTRO in the sequel formalizes this kind of reasoning.

It happens that the specifications of SET provide enough information to prove, in reasoning about a client under precondition $pool = \varnothing$, that $\mathsf{null} \notin pool$, whence $n \notin pool`rep$ above. However, formula R manifests the way field own is being used to describe a module-specific ownership discipline. Unlike I_{set}, formula R is suitable to appear in the module interface, as a public invariant [Leavens and Müller 2007] or explicitly conjoined to the procedure specifications of SET. As it is an easy consequence of I_{set}, it comes at little cost to the module specifier. Another invariant suitable to appear in the interface is $\mathsf{type}(Set, pool)$, which abbreviates $pool \subseteq pool\,/\,Set$. Yet another is the separation condition (5) based on ownership; or we could expose field own and the invariant that $s.rep`own \subseteq pool$ for all s in $pool$. In this article, we do not formalize public invariants and their use by clients.

One point of this example is that "package confinement" [Grothoff et al. 2007] 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.

The notion of dynamic boundary encompasses situations that are not amenable to general purpose static analyses. A dynamic boundary is expressed in terms of state potentially mutated by the module implementation, for example, the effect of add in Figure 1 allows writing state on which θ_{set} depends.[18] So interface specifications need to provide clients with sufficient information to reason about the boundary. This may be provided by a public invariant like R above, or by method contracts; we consider MM as an example of the latter, in Section 8.

The beauty of the second-order frame rule, the form of which is due to O'Hearn et al. [2009], is that it distills the essence of Hoare's mismatch. Rule MISMATCH is derived in Figure 12 from our rule SOF together with rule LINK (see Figure 13 or Section 7.6). Omitting a couple of side conditions, rule SOF looks as follows:

$$\dfrac{\Delta, \Theta \vdash_M \{\,P\,\}\ C\ \{\,P'\,\}\ [\varepsilon] \qquad \vdash bnd(N)\ \mathsf{frm}\ I \qquad N \in \Theta}{\Delta, (\Theta \circledcirc I) \vdash_M \{\,P \wedge I\,\}\ C\ \{\,P' \wedge I\,\}\ [\varepsilon]}.$$

[18] State-dependent effects may interfere, which is handled by the sequence rule in Part I.

Notice that the conclusion conjoins I to the specification of C, just like the ordinary FRAME rule. But the meaning of C depends on its context—that is the sense in which the rule is second order—and I is also conjoined around the context. Another difference from FRAME is that separation between C and the footprint, $bnd(N)$, of I is expressed not in terms of the end-to-end effect ε of C but rather by the step-by-step condition that C respects the dynamic boundary (which it must because $N \in \Theta$).

We conclude this section by formalizing the semantics of correctness judgments with hypotheses. Section 7 presents the proof rules.

6.4. Semantics of Hypothetical Judgments

The meaning of a judgment (18)—its validity—depends not only on what is explicit in the judgment but also on the grouping of procedures in modules (mdl), the import relation ($\preceq$), and the given dynamic boundaries (bnd).

Definition 6.3 (*Respects*). A step $\langle C,\ \sigma,\ \mu\rangle \overset{\Delta}{\longmapsto} \langle C',\ \sigma',\ \mu'\rangle$ *respects* N iff either *Agree*$(\sigma, \sigma', bnd(N))$ or *Active*(C) is a procedure call for some m in Δ with $mdl(m) \preceq N$.

Definition 6.4. A correctness judgment $\Delta \vdash^{\Gamma}_{M} \{\, P \,\}\ C\ \{\, P' \,\}\ [\varepsilon]$ is *valid*, written

$$\Delta \models^{\Gamma}_{M} \{\, P \,\}\ C\ \{\, P' \,\}\ [\varepsilon]$$

iff the following holds for all Γ-environments μ and all Γ-states σ such that $\sigma \models P$:

(Safety) It is not the case that $\langle C,\ \sigma,\ \mu\rangle \overset{\Delta}{\longmapsto}{}^{*}$ *fault*.

(Ctx-pre) It is not the case that $\langle C,\ \sigma,\ \mu\rangle \overset{\Delta}{\longmapsto}{}^{*}$ *p-fault*.

(Post) $\sigma' \models P'$ for every σ' such that $\langle C,\ \sigma,\ \mu\rangle \overset{\Delta}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \sigma',\ \mu\rangle$

(Effect) $\sigma \rightarrow \sigma' \models \varepsilon$ for every σ' such that $\langle C,\ \sigma,\ \mu\rangle \overset{\Delta}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \sigma',\ \mu\rangle$

(Encap) Every step reachable from $\langle C,\ \sigma,\ \mu\rangle$ via $\overset{\Delta}{\longmapsto}$ respects N for every N in Δ with $N \neq M$.

The Safety and Ctx-pre conditions say it is not the case that $\langle C,\ \sigma,\ \mu\rangle \overset{\Delta}{\longmapsto}{}^{*}$ *(p-)fault*, using an abbreviation. The Encap condition says that for every trace

$$\langle C,\ \sigma,\ \mu\rangle \overset{\Delta}{\longmapsto}{}^{*} \langle C'',\ \sigma'',\ \mu''\rangle \overset{\Delta}{\longmapsto} \langle C',\ \sigma',\ \mu'\rangle$$

and every N, m for which Δ specifies some p with $N = mdl(p)$ and $N \neq M$, either Agree$(\sigma'', \sigma', bnd(N))$ or *Active*(C'') is a call to some m in Δ such that $mdl(m) \preceq N$. That is, assignments in C are exempt from respecting $bnd(M)$ and moreover the definition of "respects" exempts procedure calls from the boundaries of the modules they import.

Note that the stated assumptions imply that $\langle C,\ \sigma,\ \mu\rangle$ is compatible with Γ and Δ because by well-formedness of the judgment, the procedures in Δ are not in $dom(\Gamma)$. Moreover, the Post and Effect conditions lose no generality by using the same final environment as the initial one, by bracketing (Lemma 4.7).

In case Δ is empty, validity is the same as in Part I of this article, as the Encap and Ctx-pre conditions become vacuously true.

Remark 6.5. Consider two distinct modules M and N that both refer to a global variable x in their respective dynamic boundaries. Suppose $N \in \Delta$. Then, an atomic assignment such as $x := 0$ in code of M cannot satisfy Encap because it cannot respect the boundary of N. Therefore, Encap forces dynamic boundaries of any two distinct modules either to be nonoverlapping or, if they do overlap, to not have any updates on the overlapped part. We return to this point in Section 11.

$$\textsc{Call} \quad \{P\}m(x:T)\{P'\}[\varepsilon] \vdash^{\Gamma}_{M} \{\, P^{x}_{z} \,\}\ m(z)\ \{\, P'^{x}_{z} \,\}\,[\varepsilon^{x}_{z}]$$

$$\textsc{Link}\ \frac{\Delta, \Theta \vdash^{\Gamma}_{M} \{\, P \,\}\ C\ \{\, P' \,\}\,[\varepsilon] \qquad \Delta, \Theta \vdash^{\Gamma, x_i : T_i}_{mdl(m_i)} \{\, Q_i \,\}\ B_i\ \{\, Q'_i \,\}\,[\eta_i] \text{ for all } i \qquad mdl(m_i) \notin \Delta \text{ for all } i}{\Delta \vdash^{\Gamma}_{M} \{\, P \,\}\ \mathsf{letrec}\ m_1(x_1 : T_1) = B_1; \ldots; m_n(x_n : T_n) = B_n\ \mathsf{in}\ C\ \{\, P' \,\}\,[\varepsilon]}$$

$$\textsc{FieldUpd} \quad \vdash \{\, x \neq \mathsf{null} \wedge y = F \,\}\ x.f := F\ \{\, x.f = y \,\}\,[x.f]$$

$$\textsc{Var}\ \frac{\Delta \vdash^{\Gamma, x:T} \{\, P \wedge x = \mathit{default}(T) \,\}\ C\ \{\, P' \,\}\,[x, \varepsilon]}{\Delta \vdash^{\Gamma} \{\, P \,\}\ \mathsf{var}\ x : T\ \mathsf{in}\ C\ \{\, P' \,\}\,[\varepsilon]}$$

$$\textsc{If}\ \frac{\Delta \vdash \{\, P \wedge E \neq 0 \,\}\ C_1\ \{\, P' \,\}\,[\varepsilon] \qquad \Delta \vdash \{\, P \wedge E = 0 \,\}\ C_2\ \{\, P' \,\}\,[\varepsilon]}{\Delta \vdash \{\, P \,\}\ \mathsf{if}\ E\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \{\, P' \,\}\,[\varepsilon]}$$

Fig. 13. Selected syntax directed rules. We elide M from $\vdash_M$ when it is the same for every judgment in the rule. The notation in rule LINK assumes Θ the list $\{Q_1\}m_1(x_1 : T_1)\{Q'_1\}[\eta_1], \ldots, \{Q_n\}m_n(x_n : T_n)\{Q'_n\}[\eta_n]$.

7. PROOF RULES AND SOUNDNESS

This section is devoted to presenting the proof rules and proving their soundness.

7.1. The Rules

Figure 13 gives selected rules for program constructs. Every one of the syntax-directed rules in Part I of this article is adapted the same way: axioms have an empty procedure context (like FIELDUPD) and the proper rules have the same context throughout (like IF and VAR). As in Part I, an implicit side condition on all proof rules is that both the conclusion and the premises are well formed; for example, x in rules VAR and EXTENDCTX must not be in $dom(\Gamma)$ as otherwise the typing context $\Gamma, x : T$ would not be well formed.

The authors were surprised that the axiom CALL does not need side condition $M \preceq mdl(m)$, as the expected interpretation of $\preceq$ as an abstraction of the "can call" relation between procedures (or reflexive transitive closure thereof). Indeed, in the proof rules the only constraints on $\preceq$ are the negative one in SOF and the positive one in rule CTXINTROCALL. To verify a program with multiple modules and procedures, the "context intro" rules are needed in order to lift judgments about program fragments to have contexts for which the whole program can be linked. Rule CTXINTRO can be used for intramodule procedure calls where there are no effects inside module boundaries, for example, procedures with no heap effects. But this is the exception, since frame conditions expose effects on encapsulated state. In usual cases, rule CTXINTROCALL is needed. Its use gives rise to constraints on the $\preceq$ relation. The gist of it is that the syntactic "imports" relation can be used for $\preceq$, but proofs can also be constructed using a smaller relation: instead of "can call", it only tracks "can call with effect on encapsulated state". There is also a sense in which $\preceq$ needs to be compatible with lexical nesting of letrec blocks; this is manifest in rule MISMATCH.

Rule LINK is given in the general form where one or more procedures are linked simultaneously. See Section 7.6 for a more readable version special case, LINK1. The difference from LINK0 discussed in Section 4.4 is the side condition that says that the ambient modules, that is, for procedures in Δ, are different from the module(s) of the procedures to be linked. The linked procedures need not be from the same module; this caters for simultaneous linking of mutually recursive procedures whose calls cross module boundaries, as discussed in Section 9. One might prefer to add side condition

$$\textsc{Sof}\ \frac{\Delta, \Theta \vdash_M \{\, P \,\}\ C\ \{\, P' \,\}\ [\varepsilon] \qquad \vdash bnd(N)\ \mathsf{frm}\ I \qquad N \neq M \qquad N \in \Theta \qquad mdl(m) \not\preceq N \text{ for all } m \text{ in } \Delta}{\Delta, (\Theta \circledcirc I) \vdash_M \{\, P \wedge I \,\}\ C\ \{\, P' \wedge I \,\}\ [\varepsilon]}$$

$$\textsc{CtxIntro}\ \frac{\Delta \vdash_M \{\, P \,\}\ C\ \{\, P' \,\}\ [\varepsilon] \qquad C \text{ is atomic} \qquad P \Rightarrow bnd(mdl(m)) \not\cdot \varepsilon}{\Delta, \{Q\} m(x:T) \{Q'\} [\eta] \vdash_M \{\, P \,\}\ C\ \{\, P' \,\}\ [\varepsilon]}$$

$$\textsc{CtxIntroIn}\ \frac{\Delta \vdash_M \{\, P \,\}\ C\ \{\, P' \,\}\ [\varepsilon] \qquad mdl(m) \in \Delta \text{ or } mdl(m) = M}{\Delta, \{Q\} m(x:T) \{Q'\} [\eta] \vdash_M \{\, P \,\}\ C\ \{\, P' \,\}\ [\varepsilon]}$$

$$\textsc{CtxIntroCall}\ \frac{\Delta \vdash_M \{\, P \,\}\ m(z)\ \{\, P' \,\}\ [\varepsilon] \qquad mdl(m) \preceq mdl(q)}{\Delta, \{Q\} q(x:T) \{Q'\} [\eta] \vdash_M \{\, P \,\}\ m(z)\ \{\, P' \,\}\ [\varepsilon]}$$

Fig. 14. Structural rules that manipulate procedure contexts.

$M \notin \Theta$, because the rule is not really useful if M is in Θ (as letrec is not allowed in procedure bodies); but this is not needed for soundness.

Rule FldUpdate is the same as in Part I of this article. As noted previously, all of the atomic commands except method call are treated this way: the procedure context is simply empty. Rule If illustrates how the rules for control constructs are adapted simply by adding a procedure context. This works because a command respects dynamic boundaries if its constituents do.

Figure 14 gives the structural rules that manipulate procedure contexts.

Rule SOF allows invariant I for module N to be added to the specifications for procedures of several modules (Θ). Other specifications (Δ) are unchanged; these must be procedures outside N in accord with the side condition: $mdl(m) \not\preceq N$ for all m in Δ. Side conditions $N \in \Theta$ and $N \neq M$, together with the premise judgment for C, ensure that C respects the boundary of N. The elegant SOF rule of O'Hearn et al. [2009] has no side conditions of this sort. Whereas the conditions in our rule disallow the inference in case C can interfere with I, their rule does allow to infer $\{P * I\} C \{P' * I\}$. If C can write on the footprint of I, then the footprint, P, of C will overlap I and the precondition $P * I$ will be false, so the conclusion is sound but useless.

The "context intro" rules CtxIntro, CtxIntroIn, and CtxIntroCall play a crucial role: axioms like Call and FieldUpd have the minimum necessary context, so the context needs to be extended in order to compose commands in control structure and by linking. Extending the context is, in general, a strengthening of the correctness property, owing to the Encap condition. Rule CtxIntro is restricted to atomic commands (i.e., procedure call and assignments, cf. Figure 6), because the side condition only enforces the dynamic encapsulation boundary θ for the initial and final states—there are no intermediate steps in the semantics of these commands. The frame condition of a procedure often exposes that it writes within the boundary of its own module, in which case CtxIntroCall is needed. It enforces that the relation $\preceq$ reflects at least the import relations (i.e., parts of the call graph) that involve such effects. Rule CtxIntroIn embodies two cases. The case $mdl(m) = M$ reflects that the Encap condition in Definition 6.4 exempts code in module M from respecting the boundary of M. The case $mdl(m) \in \Delta$ reflects that the Encap condition depends on what modules are in context; adding more procedures for an existing module makes no difference.

A verification condition generator based on these rules would generate, for each assignment command, the obligation that its effect is separate from the dynamic

$$\textsc{Frame}\ \frac{\Delta \vdash \{\, P \,\}\ C\ \{\, P' \,\}\ [\varepsilon] \qquad P \vdash \delta\ \mathsf{frm}\ Q \qquad P \wedge Q \Rightarrow \delta\ \not\cdot\ \varepsilon}{\Delta \vdash \{\, P \wedge Q \,\}\ C\ \{\, P' \wedge Q \,\}\ [\varepsilon]}$$

$$\textsc{ExtendCtx}\ \frac{\Delta \vdash^{\Gamma} \{\, P \,\}\ C\ \{\, P' \,\}\ [\varepsilon] \qquad \Gamma' \text{ is } (\Gamma, x : T) \text{ or } \Gamma' \text{ is } (\Gamma, m : (y : U))}{\Delta \vdash^{\Gamma'} \{\, P \,\}\ C\ \{\, P' \,\}\ [\varepsilon]}$$

$$\textsc{Subst}\ \frac{\Delta \vdash \{\, P \,\}\ C\ \{\, P' \,\}\ [\varepsilon] \qquad (P^x_F) \Rightarrow \mathit{ftpt}(F)\ \not\cdot\ (\varepsilon^x_F) \qquad x \in \mathit{SpecOnlyVar}}{\Delta \vdash \{\, P^x_F \,\}\ C\ \{\, {P'}^x_F \,\}\ [\varepsilon^x_F]}$$

$$\textsc{Exist}\ \frac{\Delta \vdash^{\Gamma, x : K} \{\, x \in G \wedge P \,\}\ C\ \{\, P' \,\}\ [\varepsilon]}{\Delta \vdash^{\Gamma} \{\, \exists x : K \in G \cdot P \,\}\ C\ \{\, P' \,\}\ [\varepsilon]}$$

$$\textsc{Conj}\ \frac{\Delta \vdash \{\, P_1 \,\}\ C\ \{\, P'_1 \,\}\ [\varepsilon] \qquad \Delta \vdash \{\, P_2 \,\}\ C\ \{\, P'_2 \,\}\ [\varepsilon]}{\Delta \vdash \{\, P_1 \wedge P_2 \,\}\ C\ \{\, P'_1 \wedge P'_2 \,\}\ [\varepsilon]}$$

Fig. 15. Selected structural rules adapted from Part I of this article. We elide M from $\vdash_M$ because it is the same for the conclusion and premise judgments.

boundary of every relevant module except the current one. The relevant modules would be determined from the syntactic import relations among modules.

Figure 15 gives selected structural rules that are adapted from those in Part I of this article. All of the structural rules from Part I are adapted, just like the syntax directed rules; note that ExtendCtx is slightly generalized in that both variables and procedures can be added to the typing context.

Theorem 7.1. *Each of the rules is sound. Hence, any derivable correctness judgment is valid.*

The rest of this section is devoted to proving that each rule is sound. Rule SOF is done first, as it is the centerpiece of the work. Rule Link is done last, as its proof is the most lengthy. Rules for the frames judgment are kept unchanged from Part I of this article where they are proved sound.

7.2. Proof of the SOF Rule

Assume the premise is valid:

$$\Delta, \Theta \models^{\Gamma}_{M} \{\, P \,\}\ C\ \{\, P' \,\}\ [\varepsilon]$$

and assume the side conditions hold: $\vdash \mathit{bnd}(N)\ \mathsf{frm}\ I$, $N \neq M$, $N \in \Theta$, and $\mathit{mdl}(m) \not\leq N$ for all m in Δ. We must prove validity of the conclusion, that is

$$\Delta, (\Theta \between I) \models^{\Gamma}_{M} \{\, P \wedge I \,\}\ C\ \{\, P' \wedge I \,\}\ [\varepsilon]\,. \tag{20}$$

For brevity, let Δ^- be the specifications (Δ, Θ) for the premise, and
Δ^+ be the specifications $(\Delta, \Theta \between I)$ for the conclusion.
Consider any Γ-state σ with $\sigma \models P \wedge I$ and let μ be any Γ-environment.

Claim. Consider any trace from $\langle C,\ \sigma,\ \mu \rangle$ via $\overset{\Delta^+}{\longmapsto}$. Then

(a) that sequence is also a trace via $\overset{\Delta^-}{\longmapsto}$, and
(b) I holds in every configuration of the trace.

The Claim implies (20), for σ and μ, as follows. It is not the case that $\langle C,\ \sigma,\ \mu\rangle \overset{\Delta^+}{\longmapsto}{}^*$ *fault*, because that would imply $\langle C,\ \sigma,\ \mu\rangle \overset{\Delta^-}{\longmapsto}{}^*$ *fault* by Claim (a), and this contradicts validity of the premise. Likewise, there is no *p-fault*. Furthermore, if the trace via Δ^+ ends at $\langle$skip, $\sigma',\ \mu\rangle$, then by validity of the premise we get $\sigma' \models P'$ and $\sigma \rightarrow \sigma' \models \varepsilon$. We get $\sigma' \models I$ from Claim (b). What remains is to show that any reachable step via $\overset{\Delta^+}{\longmapsto}$ respects L for each L in $\Delta, (\Theta \between I)$ with $L \neq M$. This follows from the corresponding condition in validity of the premise, because L is in $\Delta, \Theta \between I$ iff it is in Δ, Θ.

It remains to prove the claim, which we do by induction on the length of the trace from $\langle C,\ \sigma,\ \mu\rangle$ via Δ^+. The base case is when the length is 0. Then, (a) is immediate, and using assumption $\sigma \models P \wedge I$, we get $\sigma \models I$ for (b).

For the induction step, suppose that $\langle C,\ \sigma,\ \mu\rangle \overset{\Delta^+}{\longmapsto}{}^* \langle C'',\ \sigma'',\ \mu''\rangle$ and either $\langle C'',\ \sigma'',\ \mu''\rangle \overset{\Delta^+}{\longmapsto} \langle C',\ \sigma',\ \mu'\rangle$ or $\langle C'',\ \sigma'',\ \mu''\rangle \overset{\Delta^+}{\longmapsto}$ *(p-)fault*. By induction, we have $\langle C,\ \sigma,\ \mu\rangle \overset{\Delta^-}{\longmapsto}{}^* \langle C'',\ \sigma'',\ \mu''\rangle$ and $\sigma'' \models I$. Accordingly, we must show $\langle C'',\ \sigma'',\ \mu''\rangle \overset{\Delta^-}{\longmapsto} \langle C',\ \sigma',\ \mu'\rangle$ and $\sigma' \models I$ or else $\langle C'',\ \sigma'',\ \mu''\rangle \overset{\Delta^-}{\longmapsto}$ *(p-)fault*.

We show the nonfaulting alternative first. So suppose $\langle C'',\ \sigma'',\ \mu''\rangle \overset{\Delta^+}{\longmapsto} \langle C',\ \sigma',\ \mu'\rangle$ and go by cases on *Active*(C'').

Case. *Active*(C'') is not a call to a context procedure, and is not evar, ecall, var, or call of an environment procedure. Then, by the correspondence Lemma 4.9, we have $\langle C'',\ \sigma'',\ \mu''\rangle \overset{\Delta^-}{\longmapsto} \langle C',\ \sigma',\ \mu'\rangle$. By premises $N \neq M$ and $N \in \Theta$, the step respects N, so *Agree*$(\sigma'', \sigma', bnd(N))$. Hence by validity of the premise $\vdash bnd(N)$ frm I, using also $\sigma'' \models I$, we get $\sigma' \models I$ (recall Eq. (3) in Section 3). To be very precise, the premise is $\vdash^\Gamma bnd(N)$ frm I for some typing context Γ (which is the same for every judgment, and hence elided, in rule SOF). States σ'', σ' are for some typing context $\Gamma' \supseteq \Gamma$, by semantics (preservation Lemma 4.6), noting that the only constructs that change the typing context are evar, ecall, var, and call of an environment procedure. By definition of frame validity and semantics of formulas, we have that $\models^\Gamma bnd(N)$ frm I implies $\models^{\Gamma'} bnd(N)$ frm I, whence $\sigma' \models I$ as above.

Case. *Active*(C'') is of the form evar(x), ecall(x), var $x : T$ in $\ldots$, or $m(x)$ where m is in the environment. Again, by Lemma 4.9, we have $\langle C'',\ \sigma'',\ \mu''\rangle \overset{\Delta^-}{\longmapsto} \langle C',\ \sigma',\ \mu'\rangle$. Conceptually, part (b) of the Claim holds for the same reason as in the preceding case: The step respects N, that is, *Agree*$(\sigma'', \sigma', bnd(N))$, and so I is preserved owing to its framing judgment. However, our formulation of "frame validity", see Eq. (3), does not encompass transitions that add or remove variables. That could be done, but would be hard to motivate as it is not needed except in the present case. Instead, we argue directly in terms of semantics. Transitions for evar(x) and ecall(x) change the state only by removing x. By well-formedness conditions, x is not in Γ, and so by semantics of formulas, we have $\sigma'' \models I$ iff $\sigma' \models I$. Transitions for call of an environment procedure, and for var, change the state only by adding a fresh variable not in Γ, so again $\sigma'' \models I$ iff $\sigma' \models I$.

Case. *Active*(C'') is a call $m(z)$ to a procedure m with specification $\{V\}m(x : T)\{V'\}[\eta]$ in $(\Theta \between I)$. Observe that $C'' = m(z); D$, $\mu' = \mu''$, and $C' = D$. And V is $Q \wedge I$ and V' is $Q' \wedge I$ for some Q, Q' such that $\{Q\}m(x : T)\{Q'\}[\eta]$ is in Θ. Let $\overline{s}$ be the specification-only variables that occur in Q, Q', η. Because this context call makes a nonfaulting

transition via Δ^+, there must be values $\overline{n}$ such that $\sigma'' \models (Q \wedge I)^{x,\overline{s}}_{z,\overline{n}}$ (using abbreviation (11)) and also

$$\sigma''{\rightarrow}\sigma' \models \eta^x_z \quad \text{and} \quad \forall \overline{n} \cdot (\sigma'' \models (Q \wedge I)^{x,\overline{s}}_{z,\overline{n}}) \Rightarrow (\sigma' \models (Q' \wedge I)^{x,\overline{s}}_{z,\overline{n}}). \tag{21}$$

Observe that I^x_z is I, because x is local to the specification of m. By well-formedness of $bnd(N)$, no specification-only variables occur in $bnd(N)$, so I does not depend[19] on $\overline{s}$. Thus, we have

$$\exists \overline{n} \cdot \sigma'' \models (Q \wedge I)^{x,\overline{s}}_{z,\overline{n}} \quad \text{iff} \quad \sigma'' \models I \quad \text{and} \quad \exists \overline{n} \cdot \sigma'' \models Q^{x,\overline{s}}_{z,\overline{n}}, \tag{22}$$

whence we have $\sigma'' \models I$ for Claim(b). Furthermore, one of the conditions in the transition rule for $m(z)$ via Δ^- is $\exists \overline{n} \cdot \sigma'' \models Q^{x,\overline{s}}_{z,\overline{n}}$. Another is $\sigma''{\rightarrow}\sigma' \models \eta^x_z$ which we have in (21). The remaining condition for transition via Δ^- is $\forall \overline{n} \cdot (\sigma'' \models Q^{x,\overline{s}}_{z,\overline{n}}) \Rightarrow (\sigma' \models Q'^{x,\overline{s}}_{z,\overline{n}})$. This follows from the second conjunct of (21) using that $\sigma'' \models I$ and I does not depend on $\overline{s}$.

Case. *Active*(C'') is a call to some procedure p in Δ. By Lemma 4.9, we have $\langle C'', \sigma'', \mu'' \rangle \overset{\Delta^-}{\longmapsto} \langle C', \sigma', \mu' \rangle$. By premise $N \neq M$, the step respects N. So, by definition of respects, either *Agree*$(\sigma'', \sigma', bnd(N))$, and we get $\sigma' \models I$ as in the first case above, or else $mdl(p) \preceq N$. But $mdl(p) \preceq N$ contradicts the premise "$mdl(m) \not\preceq N$ for all m in Δ".

Finally, we show the faulting alternative. Suppose $\langle C'', \sigma'', \mu'' \rangle \overset{\Delta^+}{\longmapsto}$ *(p-)fault*, noting that "*(p-)fault*" abbreviates two cases. The case $\langle C'', \sigma'', \mu'' \rangle \overset{\Delta^+}{\longmapsto}$ *fault* happens only if the active command is field read or field update; then, by correspondence Lemma 4.9, we get $\langle C'', \sigma'', \mu'' \rangle \overset{\Delta^-}{\longmapsto}$ *fault*, as required for the Claim (a). And there is nothing to prove for Claim (b). The case $\langle C'', \sigma'', \mu'' \rangle \overset{\Delta^+}{\longmapsto}$ *p-fault* happens for a call of a context procedure, say m. If m is in Δ, we get $\langle C'', \sigma'', \mu'' \rangle \overset{\Delta^-}{\longmapsto}$ *p-fault* by Lemma 4.9, as Δ^- and Δ^+ agree on Δ. Otherwise, m is in Θ and in Δ^+, it has specification of the form $\{V\}m(x:T)\{V'\}[\eta]$, where V is $Q \wedge I$ for some Q. Let the specification-only variables be $\overline{s}$. By semantics, we have $\neg(\exists \overline{n} \cdot \sigma'' \models (Q \wedge I)^{x,\overline{s}}_{z,\overline{n}})$. Since $\sigma'' \models I$ and in light of (22), we have $\neg(\exists \overline{n} \cdot \sigma'' \models Q^{x,\overline{s}}_{z,\overline{n}})$ and thus $\langle C'', \sigma'', \mu'' \rangle \overset{\Delta^-}{\longmapsto}$ *p-fault*.

7.3. Proofs of the Procedure Call and Context Introduction Rules

For CALL, let Θ be $\{P\}m(x:T)\{P'\}[\varepsilon]$. Let the specification-only variables in P, P', ε be $\overline{s}$. To prove $\Theta \models^{\Gamma}_{M} \{ P^x_z \}\, m(z)\, \{ P'^x_z \}[\varepsilon^x_z]$, suppose $\sigma \models P^x_z$ and let μ be a Γ-environment. By semantics, there is no faulting transition and we have $\langle m(z), \sigma, \mu \rangle \overset{\Theta}{\longmapsto} \langle \mathsf{skip}, \sigma', \mu \rangle$ for all σ' such that $\sigma{\rightarrow}\sigma' \models \varepsilon^x_z$ and $\forall \overline{n} \cdot (\sigma \models P^{x,\overline{s}}_{z,\overline{n}}) \Rightarrow (\sigma' \models P'^{x,\overline{s}}_{z,\overline{n}})$. These outcomes satisfy the Safety and Effect conditions. For any such σ', we have $\sigma' \models P'^x_z$ by instantiating $\overline{n}$ as $\sigma(\overline{s})$ and using that $\sigma'(\overline{s}) = \sigma(\overline{s})$ because by well-formedness ε does not allow writes of specification-only variables. Thus, the outcome satisfies Post. For Encap, if $M \neq mdl(m)$, then we need the step to respect $mdl(m)$, which it does by Definition 6.3 of respects since $mdl(m) \preceq mdl(m)$.

[19]But we do not say that $I^{\overline{s}}_{\overline{n}}$ is I, because the substitution for $\overline{s}$ is only an abbreviation used in connection with $\models$; see (11).

For CTXINTRO, suppose C is an atomic command and let Θ be $\{Q\}m(x:T)\{Q'\}[\eta]$. Suppose the premise is valid: $\Delta \models^{\Gamma}_{M} \{P\}\ C\ \{P'\}\ [\varepsilon]$. To show $\Delta, \Theta \models_{M} \{P\}\ C\ \{P'\}\ [\varepsilon]$, consider any Γ-environment μ and any σ with $\sigma \models P$. If C is skip, then there is no transition via $\overset{\Delta\Theta}{\longmapsto}$ and the only thing to prove is $\sigma \models P'$—which we have by validity of the premise. If C is an assignment, field update, or call of a procedure (which must be in Δ, by well formedness of the premise), the possible steps via $\overset{\Delta\Theta}{\longmapsto}$ have the form $\langle C, \sigma, \mu\rangle \overset{\Delta\Theta}{\longmapsto} \langle \mathsf{skip}, \sigma', \mu'\rangle$ and $\langle C, \sigma, \mu\rangle \overset{\Delta\Theta}{\longmapsto} \mathit{fault}$. By correspondence Lemma 4.9 and well formedness of C in Δ, these are also steps via $\overset{\Delta}{\longmapsto}$. So by the premise, the fault case does not happen, and by the premise, we have $\sigma' \models P'$ and $\sigma \rightarrow \sigma' \models \varepsilon$. For the Encap condition, we must show the step respects N for every N in Δ, Θ with $N \neq M$. For N in Δ, this follows by validity of the premise. For $N = mdl(m)$, from premise $P \Rightarrow bnd(mdl(m))$ *rl.* ε, we get $\mathit{Agree}(\sigma, \sigma', bnd(mdl(m)))$ using the separator agreement lemma mentioned in Section 3 (Lemma 6.8 in Part I).

For CTXINTROCALL, the argument is again like that for CTXINTRO, except to show that the step respects $mdl(q)$, where q is the procedure being added to the context. It does respect $mdl(q)$ by Definition 6.3 using the rule's premise $mdl(m) \preceq mdl(q)$.

For CTXINTROIN, let Θ be $\{Q\}m(x:T)\{Q'\}[\eta] \vdash_{M} \{P\}\ C\ \{P'\}\ [\varepsilon]$. Because the premise of the rule is well formed, there are no calls of m in C. So for any environment μ and state σ that satisfies P, any trace from $\langle C, \sigma, \mu\rangle$ via $\overset{\Delta\Theta}{\longmapsto}$ is a trace via $\overset{\Delta}{\longmapsto}$. (In detail, this is proved by induction on the trace, using the correspondence Lemma 4.9.) Thus, the Safety, Post, Effect, and Ctx-pre conditions for the conclusion follow from validity of the premise.

The Encap condition for the conclusion of CTXINTROIN requires every step to respect every module N in Δ, Θ with $N \neq M$. In case, the side condition $mdl(m) \in \Delta$ holds, every N in Δ, Θ is in Δ so the Encap condition for the conclusion is the same as the Encap condition for the premise. In the other case, that is, $mdl(m) = M$ holds, the Encap conditions are again the same.

7.4. Proofs of the Structural Rules

Rule CONJ illustrates the situation with the remaining structural rules: the premises and conclusion have the same procedure context, Δ. Validity of the conclusion involves the transition relation $\overset{\Delta}{\longmapsto}$ and this is the same as the transition relation for the premises. Moreover the same command appears in the premises as in the conclusion. So the soundness proof is quite direct from Definition 6.4. In particular, the Pre-ctx and Encap conditions in the conclusion are the same as those of the premises.

Rules like FRAME, SUBEFF, VARMASK, and CONSEQ have more or less complicated side conditions and manipulation of precondition, postcondition, and/or frame condition. These conditions carry over unchanged from Part I of this article. As in the case of CONJ, the soundness argument from Part I can be extended because again the same procedure context and command is present in the premises as in the conclusion.

The only structural rules that manipulate the typing context are EXTENDCTX, EXIST, and EXISTREGION. These, like CONJ, have the same procedure context and same command in premise and conclusion. The proofs for EXIST and EXISTRGN are straightforward adaptations of their proofs in Part I. Owing to well-formedness of the premises and conclusion in each case, the variable of interest does not occur free in the procedure context or in the dynamic boundaries.

Rule EXTENDCTX has two cases, extending Γ with either a variable or a procedure. In case Γ' is $\Gamma, x:T$, by well-formedness of the premise, x does not occur in C or in the specification. For any $\Gamma, x:T$-state σ we have $\sigma \models^{\Gamma, x:T} P$ iff $\sigma \upharpoonright x \models^{\Gamma} P$ and *mutatis*

mutandis for P'. Any trace from $\langle C,\ \sigma,\ \mu\rangle$ yields a trace from $\langle C,\ \sigma\restriction x,\ \mu\rangle$ with exactly the same configurations except x tossed from the state. Thus, validity of the conclusion follows easily from validity of the premise. In the other case, Γ' is $\Gamma, m:(y:U)$. A Γ'-state τ is also a Γ-state, by definition. For any $\Gamma, m:(y:U)$-environment μ, any computation from $\langle C,\ \tau,\ \mu\rangle$ yields a computation from $\langle C,\ \tau,\ \mu\restriction m\rangle$ with exactly the same configurations except m tossed. Again, validity of the conclusion follows easily from validity of the premise. The main reason to allow procedures in the typing context of correctness judgments is that the soundness of ExtendCtx is used in the proof of rule Link.

Rule Subst looks the same as the one in Part I; note that no substitution is performed on the specifications in the context, nor on the dynamic boundaries (which, to be well-formed, do not depend on specification-only variables). The soundness proof in Part I hinges on the following observation. To prove the conclusion of the rule, consider any σ such that $\sigma \models P^x_F$. By properties of substitution this implies $\tau \models P$ where $\tau = [\sigma \mid x : \sigma(F)]$. Computations from τ are thus among those to which the premise judgment applies. Furthermore, because x is specification-only, computations of C fault from σ iff they fault from τ; and if trace from σ terminates in a final state σ' then there is a trace from τ that terminates in $[\sigma' \mid x : \sigma(F)]$. Lemma 4.10 confirms these properties in the presence of procedures. As argued in detail in Part I of this article, these observations allow the Safety, Post, and Effect conditions for traces from P^x_F-states to be derived from those from P-states, using the premise of the rule. The Ctx-pre-condition can be shown in the same way. The Encap condition also follows directly from the premise, as traces from τ are among those to which the premise applies; the respects condition is interpreted exactly the same in the conclusion as in the premise, as no substitution is involved.

7.5. Proofs of the Syntax-Directed Rules besides Call and Link

Soundness of axiom FieldUpd is proved just as in Part I: the procedure context is empty, so the Encap condition is vacuously true. The other syntax-directed axioms from Part I carry over the same way, with empty procedure context: Alloc, Assign, and FieldAcc.

The other syntax-directed rules are Var and those for control structure. In each case, the only difference from Part I is the addition of a fixed procedure context Δ for all the judgments and the corresponding Encap condition. Let us consider rule If in a bit more detail. To show validity of the conclusion, that is, $\Delta \models \{\, P \,\}\ \mathsf{if}\ E\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \{\, P' \,\}\ [\varepsilon]$, we consider traces via $\overset{\Delta}{\longmapsto}$ from initial configurations $\langle \mathsf{if}\ E\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2,\ \sigma,\ \mu\rangle$ where $\sigma \models P$. The first step respects dynamic boundaries in Δ because it does not change the state. It goes to $\langle C_1,\ \sigma,\ \mu\rangle$ if $\sigma(E) \neq 0$, in which case σ satisfies the precondition of the assumed premise $\Delta \models \{\, P \wedge E \neq 0 \,\}\ C_1\ \{\, P' \,\}\ [\varepsilon]$. That assumption takes care of the remaining steps of computation. Similarly, if $\sigma(E) = 0$.

7.6. Proof of the Link Rule

For clarity, we prove the special case where there is a single procedure. The rule is thus

$$\textsc{Link1}\ \frac{\begin{array}{c}\Theta \text{ is } \{Q\}m(x:T)\{Q'\}[\eta] \\ \Delta, \Theta \vdash^{\Gamma}_{M} \{\, P \,\}\ C\ \{\, P' \,\}\ [\varepsilon] \qquad \Delta, \Theta \vdash^{\Gamma, x:T}_{N} \{\, Q \,\}\ B\ \{\, Q' \,\}\ [\eta] \qquad N \notin \Delta\end{array}}{\Delta \vdash^{\Gamma}_{M} \{\, P \,\}\ \mathsf{letrec}\ m(x:T) = B\ \mathsf{in}\ C\ \{\, P' \,\}\ [\varepsilon]}.$$

In the rest of this subsection, we assume Θ is $\{Q\}m(x:T)\{Q'\}[\eta]$. Proving the general rule (Figure 13) is notationally messy but not significantly different so we merely remark on it in passing. The reader is expected to be familiar with the argument

sketched in Section 4.4. Here, we make the inductive hypotheses more precise and we deal with dynamic boundaries. The technical material in Section 5 is used.

Owing to the form of our letrec construct, soundness of the LINK rule involves inductive reasoning about a possibly recursive procedure implementation. To prove soundness of LINK, we first address the hypothetical correctness of the procedure body, in terms of configurations where m is bound to B in the environment. Lemma 7.2 parallels the Safety, Post, Effect, and Encap conditions that comprise the semantics of hypothetical judgments—but note the use of $\overset{\Delta}{\longmapsto}$ rather than $\overset{\Delta\Theta}{\longmapsto}$.

LEMMA 7.2 (RECURSION). *Suppose* $N \notin \Delta$ *and suppose we have the valid judgment*

$$\Delta, \Theta \models_N^{\Gamma, x:T} \{\, Q \,\}\ B\ \{\, Q' \,\}\ [\eta] \tag{23}$$

Let u *be in Locals but not in* $dom(\Gamma) \cup \{x\}$. *Let* Γ' *be any extension of* $\Gamma, u:T, m:(x:T)$. *Let* $\dot{\mu}$ *be any* Γ'*-environment such that* $\dot{\mu}(m) = (x:T.B)$. *Let* σ *be any* Γ'*-state such that* $\sigma \models Q_u^x$. *Then, we have the following:*[20]

$$\textit{It is not the case that } \langle B_u^x,\ \sigma,\ \dot{\mu}\rangle \overset{\Delta}{\longmapsto}{}^* \textit{ fault or } \langle B_u^x,\ \sigma,\ \dot{\mu}\rangle \overset{\Delta}{\longmapsto}{}^* \textit{ p-fault} \tag{24}$$

$$\textit{For all } \sigma', \textit{ if } \langle B_u^x,\ \sigma,\ \dot{\mu}\rangle \overset{\Delta}{\longmapsto}{}^* \langle \mathsf{skip},\ \sigma',\ \dot{\mu}\rangle \textit{ then } \sigma' \models {Q'}_u^x \textit{ and } \sigma \to \sigma' \models \eta_u^x \tag{25}$$

$$\textit{Every reachable step from } \langle B_u^x,\ \sigma,\ \dot{\mu}\rangle \textit{ via } \overset{\Delta}{\longmapsto} \textit{ respects } L \textit{ for all } L \in \Delta \tag{26}$$

Note that the given conditions ensure that $\langle B_u^x,\ \sigma,\ \dot{\mu}\rangle$ *is compatible with* Γ' *and* Δ, *for example,* $\Gamma', sigs(\Delta) \vdash B_u^x$.

For expository reasons, we defer the proof to after proving Lemma 7.4. In fact, the proof of Lemma 7.4 relies on Lemma 7.2 and not the other way around.

LEMMA 7.3 (JUDGMENT RENAMING). *Let* x *and* y *be in Locals. If* $\Delta \models_M^{\Gamma, x:T} \{\, P \,\}\ C\ \{\, P' \,\}\ [\varepsilon]$ *and* $\Gamma, y:T$ *is well formed, then* $\Delta \models_M^{\Gamma, y:T} \{\, P_y^x \,\}\ C_y^x\ \{\, {P'}_y^x \,\}\ [\varepsilon_y^x]$.

PROOF. As x is in *Locals*, the specifications Δ are well formed in Γ, and so since y not in Γ they are also well formed in $\Gamma, y:T$. To prove the conclusion, consider any $\Gamma, y:T$-state σ and let τ be σ with y renamed to x, which can be written $\mathit{Extend}(\sigma \restriction y, x, \sigma(y))$. For any μ, any trace from $\langle C_y^x,\ \sigma,\ \mu\rangle$ yields a trace from $\langle C,\ \tau,\ \mu\rangle$ with the same configurations except y renamed to x. (To be precise, it may be that the original trace chooses x as a fresh variable for a local block or procedure call, in which case we first obtain a trace by renaming x, and from this a trace with y renamed to x.) If $\sigma \models P_y^x$, then $\tau \models P$, so we may use validity of the premise to prove the required conditions for the conclusion. For example, if the trace from σ reaches state σ', then the trace from τ reaches $\tau' = \mathit{Extend}(\sigma' \restriction y, x, \sigma'(y))$. Then, by the premise, we have $\tau' \models P'$, whence $\sigma' \models {P'}_y^x$. As x and y are in *Locals*, the renaming has no bearing on whether modules in Δ are respected. □

LEMMA 7.4 (SOUNDNESS OF LINK). *Consider an instantiation of the* LINK1 *rule and suppose the side condition* $N \notin \Delta$ *and both premises are valid:*

$$\Delta, \Theta \models_M^{\Gamma} \{\, P \,\}\ C\ \{\, P' \,\}\ [\varepsilon] \tag{27}$$

$$\Delta, \Theta \models_N^{\Gamma, x:T} \{\, Q \,\}\ B\ \{\, Q' \,\}\ [\eta] \tag{28}$$

[20]For (25), note that substitution instances of B do not contain endmarkers, so by bracketing (Lemma 4.7) the final environment is the same as the initial one.

Then, the conclusion is also valid:

$$\Delta \models_M^{\Gamma} \{\, P \,\}\ \mathsf{letrec}\ m(x:T) = B\ \mathsf{in}\ C\ \{\, P' \,\}\ [\varepsilon]\,. \tag{29}$$

PROOF. To prove the conclusion, we must reason about executions via $\overset{\Delta}{\longmapsto}$ whereas the premises pertain to executions via $\overset{\Delta\Theta}{\longmapsto}$. In particular, we must show the five correctness conditions Safety, Ctx-Pre, Post, Effect, and Encap for $\mathsf{letrec}\ m(x:T) = B\ \mathsf{in}\ C$ via $\overset{\Delta}{\longmapsto}$. To that end, consider any Γ-state σ such that $\sigma \models P$ and any Γ-environment μ. These will be fixed throughout the rest of the proof. Let list $\overline{s}$ be the specification-only variables that occur in the specification of m, that is, in Q, Q', or η. □

By semantics, there is a single transition from the initial configuration:

$$\langle \mathsf{letrec}\ m(x:T) = B\ \mathsf{in}\ C,\ \sigma,\ \mu\rangle \overset{\Delta}{\longmapsto} \langle C; \mathsf{elet}(m),\ \sigma,\ \dot{\mu}\rangle,$$

where $\dot{\mu} = \mathit{Extend}(\mu, m, (x:T.B))$. Any trace of $C; \mathsf{elet}(m)$ corresponds step by step with a trace of C containing a trailing $\mathsf{elet}(m)$ in every configuration (and exactly the same states), possibly followed by a final step that executes $\mathsf{elet}(m)$, removing m from the environment. The step for $\mathsf{elet}(m)$ neither faults nor changes the state. Thus, to prove (29), it suffices to show the following:

(i) it is not the case that $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\Delta}{\longmapsto}^*$ *(p-)fault*

(ii) for any σ', if $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\Delta}{\longmapsto}^* \langle \mathsf{skip},\ \sigma',\ \dot{\mu}\rangle$ then $\sigma' \models P'$ and $\sigma \to \sigma' \models \varepsilon$

(iii) every reachable step $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\Delta}{\longmapsto}^* \langle C'',\ \sigma'',\ \mu''\rangle \overset{\Delta}{\longmapsto} \langle C',\ \sigma',\ \mu'\rangle$ respects every $L \in \Delta$ with $L \neq M$.

Recall Definition 5.6 of m-truncated traces. We will prove (i)–(iii) using the following.

Claim A. For all $C', \sigma', \dot{\mu}'$, and m-truncated traces $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\Delta}{\longmapsto}^* \langle C',\ \sigma',\ \dot{\mu}'\rangle$, we have

(a) $\langle C,\ \sigma,\ \mu\rangle \overset{\Delta\Theta}{\longmapsto}^* \langle C',\ \sigma',\ \mu'\rangle$, where $\mu' = \dot{\mu}' \restriction m$

(b) if $C' = m(z); D$ for some z, D, then there are values $\overline{n}$ such that $\sigma' \models Q^{x,\overline{s}}_{z,\overline{n}}$ (i.e., $[\sigma' \mid \overline{s} : \overline{n}] \models Q^x_z$)

To prove (i), suppose $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\Delta}{\longmapsto}^* \langle C',\ \sigma',\ \dot{\mu}'\rangle \overset{\Delta}{\longmapsto}$ *(p-)fault*. If the part of this trace before faulting is m-truncated then we have $\langle C,\ \sigma,\ \mu\rangle \overset{\Delta\Theta}{\longmapsto}^* \langle C',\ \sigma',\ \mu'\rangle$ by Claim A(a). In this case, from $\langle C',\ \sigma',\ \dot{\mu}'\rangle \overset{\Delta}{\longmapsto}$ *(p-)fault*, we have by semantics $\mathit{Active}(C')$ is a field access/update (for *fault*) or a call of a context procedure p in Δ (for *p-fault*). Thus, by the special correspondence Lemma 5.2, we get $\langle C',\ \sigma',\ \mu'\rangle \overset{\Delta\Theta}{\longmapsto}$ *(p-)fault*. But this contradicts validity of the correctness judgment for C, that is, premise (27).

In the other case for (i), the trace $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\Delta}{\longmapsto}^* \langle C',\ \sigma',\ \dot{\mu}'\rangle$ is not m-truncated. In light of the discussion following Definition 5.6, it has a prefix of the form

$$\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\Delta}{\longmapsto}^* \langle m(z); D,\ \rho,\ \dot{\nu}\rangle \overset{\Delta}{\longmapsto} \langle B^x_u; \mathsf{ecall}(u); D,\ \upsilon,\ \dot{\nu}\rangle \overset{\Delta}{\longmapsto}^* \langle A; \mathsf{ecall}(u); D,\ \sigma',\ \dot{\mu}'\rangle,$$

where u is a fresh variable and υ is $\mathit{Extend}(\rho, u, \rho(z))$. Moreover $\langle A,\ \sigma',\ \dot{\mu}'\rangle \overset{\Delta}{\longmapsto}$ *(p-)fault*, because this was not a completed call. So we have $\langle B^x_u,\ \upsilon,\ \dot{\nu}\rangle \overset{\Delta}{\longmapsto}^* \langle A,\ \sigma',\ \dot{\mu}'\rangle$ and thus $\langle B^x_u,\ \upsilon,\ \dot{\nu}\rangle \overset{\Delta}{\longmapsto}^*$ *(p-)fault*. But by Claim A(b) we have $\exists \overline{n} \cdot \rho \models Q^{x,\overline{s}}_{z,\overline{n}}$ and thus $\exists \overline{n} \cdot \upsilon \models Q^{x,\overline{s}}_{u,\overline{n}}$ by the little substitution Lemma 5.1. Note that locals x, z, u are

distinct from the specification-only variables $\overline{s}$, and by abbreviation (11) we have some $\overline{n}$ with $[\upsilon \mid \overline{s} : \overline{n}] \models Q_u^x$. By Lemma 4.10 we get $\langle B_u^x,\ [\upsilon \mid \overline{s} : \overline{n}]\ ,\ \dot{\nu}\rangle \overset{\Delta}{\longmapsto}{}^* \langle \textit{p-fault}\rangle$. But this contradicts correctness of B_u^x, according to Lemma 7.2(24), and using premise (28). So (i) is proved.

To prove (ii), suppose $\langle C, \sigma, \dot{\mu}\rangle \overset{\Delta}{\longmapsto}{}^* \langle \mathsf{skip}, \sigma', \dot{\mu}\rangle$. This is m-truncated, so by Claim A(a) we get $\langle C, \sigma, \mu\rangle \overset{\Delta\Theta}{\longmapsto}{}^* \langle \mathsf{skip}, \sigma', \mu\rangle$. So by the hypothesis $\sigma \models P$ with which we set forth, and premise (27), we have $\sigma' \models P'$ and $\sigma \to \sigma' \models \varepsilon$.

To prove (iii), suppose $\langle C, \sigma, \dot{\mu}\rangle \overset{\Delta}{\longmapsto}{}^* \langle C'', \sigma'', \dot{\mu}''\rangle \overset{\Delta}{\longmapsto} \langle C', \sigma', \dot{\mu}'\rangle$. Consider any $L \in \Delta$ such that $L \neq M$. We show the step from σ'' to σ' respects L. In case the first part, $\langle C, \sigma, \dot{\mu}\rangle \overset{\Delta}{\longmapsto}{}^* \langle C'', \sigma'', \dot{\mu}''\rangle$, is m-truncated, we get $\langle C, \sigma, \mu\rangle \overset{\Delta\Theta}{\longmapsto}{}^* \langle C'', \sigma'', \mu''\rangle$ by Claim A(a). Then consider subcases on $\mathit{Active}(C'')$:

— If $\mathit{Active}(C'')$ is not a call, or is a call to some $p \in \Delta$, then by the special correspondence Lemma 5.2, using $\langle C, \sigma, \mu\rangle \overset{\Delta\Theta}{\longmapsto}{}^* \langle C'', \sigma'', \mu''\rangle$, we get $\langle C'', \sigma'', \mu''\rangle \overset{\Delta\Theta}{\longmapsto} \langle C', \sigma', \mu'\rangle$, so the step respects L by premise (27) for C.
— The remaining subcase is that $\mathit{Active}(C'')$ is a call of an environment procedure. This includes a call of m, which is allowed by the definition of m-truncated. By semantics, the step only writes the fresh local variable used for the parameter, and this does not violate agreement so it respects L.

Finally, consider the case that $\langle C, \sigma, \dot{\mu}\rangle \overset{\Delta}{\longmapsto}{}^* \langle C'', \sigma'', \dot{\mu}''\rangle$ is not m-truncated. It has a prefix of the form

$$\langle C, \sigma, \dot{\mu}\rangle \overset{\Delta}{\longmapsto}{}^* \langle m(z); D, \rho, \dot{\nu}\rangle \overset{\Delta}{\longmapsto} \langle B_u^x; \mathsf{ecall}(u); D, \upsilon, \dot{\nu}\rangle$$

with u a fresh variable and $\upsilon = \mathit{Extend}(\rho, u, \rho(z))$. Moreover, we have $\exists \overline{n} \cdot \rho \models Q_{z,\overline{n}}^{x,\overline{s}}$ by Claim A(b). Now we consider subcases on whether $\mathit{Active}(C'')$ is code from B:

Case. $\langle B_u^x; \mathsf{ecall}(u); D, \upsilon, \dot{\nu}\rangle \overset{\Delta}{\longmapsto}{}^* \langle \mathsf{ecall}(u); D, \sigma'', \dot{\mu}''\rangle$ so $C'' = \mathsf{ecall}(u); D$. Then, the next configuration is $\langle D, \sigma', \dot{\mu}'\rangle$ where $\sigma' = \sigma'' \restriction u$, $C' = D$, and $\dot{\mu}'' = \dot{\mu}'$. So we have $\mathit{Agree}(\sigma'', \sigma', \mathit{bnd}(L))$ because $\sigma' = \sigma'' \restriction u$. Thus the step respects L.

Case. $\langle B_u^x; \mathsf{ecall}(u); D, \upsilon, \dot{\nu}\rangle \overset{\Delta}{\longmapsto}{}^* \langle A; \mathsf{ecall}(u); D, \sigma'', \dot{\mu}''\rangle$ so $C'' = A; \mathsf{ecall}(u); D$. We have[21] $A \neq \mathsf{skip}$ so there is some A' with $\langle A; \mathsf{ecall}(u); D, \sigma'', \dot{\mu}''\rangle \overset{\Delta}{\longmapsto} \langle A'; \mathsf{ecall}(u); D, \sigma', \dot{\mu}'\rangle$ and $C' = A'; \mathsf{ecall}(u); D$. Thus, $\langle B_u^x, \upsilon, \dot{\nu}\rangle \overset{\Delta}{\longmapsto}{}^* \langle A, \sigma'', \dot{\mu}''\rangle \overset{\Delta}{\longmapsto} \langle A', \sigma', \dot{\mu}'\rangle$. From $\exists \overline{n} \cdot \rho \models Q_{z,\overline{n}}^{x,\overline{s}}$, we get $\exists \overline{n} \cdot \upsilon \models Q_{u,\overline{n}}^{x,\overline{s}}$ by the little substitution Lemma 5.1. Now, by Lemma 4.10, we have $\langle B_u^x, [\upsilon \mid \overline{s} : \overline{n}]\ , \dot{\nu}\rangle \overset{\Delta}{\longmapsto}{}^* \langle A, [\sigma'' \mid \overline{s} : \overline{n}]\ , \dot{\mu}''\rangle \overset{\Delta}{\longmapsto} \langle A', [\sigma' \mid \overline{s} : \overline{n}]\ , \dot{\mu}'\rangle$. This trace is from a state that satisfies the precondition for m, so using Lemma 7.2 and premises $N \notin \Delta$ and (28) of Lemma 7.4 we get that this last step (from A to A') respects L. The step in question, $\langle A, \sigma'', \dot{\mu}''\rangle \overset{\Delta}{\longmapsto} \langle A', \sigma', \dot{\mu}'\rangle$, does not change $\overline{s}$ (by Lemma 4.10) so it too respects L.

It remains to prove Claim A. For this, we use the following.

Claim B. For any $n \geq 0$, we have the following. For all $C_0, \sigma_0, \dot{\mu}_0, C', \sigma', \dot{\mu}'$, and for any m-truncated trace

$$\langle C, \sigma, \dot{\mu}\rangle \overset{\Delta}{\longmapsto}{}^* \langle C_0, \sigma_0, \dot{\mu}_0\rangle \overset{\Delta}{\longmapsto}{}^* \langle C', \sigma', \dot{\mu}'\rangle$$

[21] As otherwise we would be in the preceding case, since in a configuration we identify $\mathsf{skip}; X$ with X.

if the trace $\langle C_0,\ \sigma_0,\ \dot{\mu}_0\rangle \overset{\Delta}{\longmapsto}{}^* \langle C',\ \sigma',\ \dot{\mu}'\rangle$ has exactly n completed topmost calls of m, and we have $\langle C,\ \sigma,\ \mu\rangle \overset{\Delta\Theta}{\longmapsto}{}^* \langle C_0,\ \sigma_0,\ \mu_0\rangle$ where $\mu_0 = \dot{\mu}_0 \restriction m$ and $\mu' = \dot{\mu}' \restriction m$, then we have $\langle C_0,\ \sigma_0,\ \mu_0\rangle \overset{\Delta\Theta}{\longmapsto}{}^* \langle C',\ \sigma',\ \mu'\rangle$

(Note that since $\dot{\mu}_0$ and $\dot{\mu}'$ are reached from $\langle C,\ \sigma,\ \dot{\mu}\rangle$, by bracketing they both contain m, hence our choice of dotted identifiers.)

To prove Claim A(a), instantiate Claim B with $C_0, \sigma_0, \dot{\mu}_0 := C, \sigma, \dot{\mu}$.

To prove Claim A(b), suppose $\langle C, \sigma, \dot{\mu}\rangle \overset{\Delta}{\longmapsto}{}^* \langle C', \sigma', \dot{\mu}'\rangle$ and $C' = m(z); D$ for some z, D. By Claim A(a), we have $\langle C, \sigma, \mu\rangle \overset{\Delta\Theta}{\longmapsto}{}^* \langle C', \sigma', \mu'\rangle$. Suppose, for the sake of contradiction, that there is no $\overline{n}$ with $\sigma' \models Q^{x,\overline{s}}_{z,\overline{n}}$. Then, by semantics, we have $\langle C',\ \sigma',\ \mu'\rangle \overset{\Delta\Theta}{\longmapsto}$ *p-fault* and so $\langle C,\ \sigma,\ \mu\rangle \overset{\Delta\Theta}{\longmapsto}{}^*$ *p-fault*. But this contradicts premise (27), since $\sigma \models P$.

PROOF OF CLAIM B. By induction on n. Using Lemma 5.5, we obtain intermediate states ρ_i, τ_i, σ_i and environments $\dot{\mu}_i$ (using names $\dot{\mu}_i$ to indicate that each binds m to $(x : T.B)$ as there is no shadowing according to Lemma 4.6) such that

$$
\begin{array}{ll}
\langle C_0,\ \sigma_0,\ \dot{\mu}_0\rangle & \\
\overset{\Delta}{\longmapsto}{}^* \langle m(z_1); C_1,\ \rho_1,\ \dot{\mu}_1\rangle & \text{with no invocations of } m \\
\overset{\Delta}{\longmapsto} \langle B^x_{x_1}; \mathsf{ecall}(x_1); C_1,\ \upsilon_1,\ \dot{\mu}_1\rangle & \text{where } \upsilon_1 = \mathit{Extend}(\rho_1, x_1, \rho_1(z_1)) \text{ and } x_1 \text{ fresh} \\
\overset{\Delta}{\longmapsto}{}^* \langle \mathsf{ecall}(x_1); C_1,\ \tau_1,\ \dot{\mu}_1\rangle & \text{where } \langle B^x_{x_1},\ \upsilon_1,\ \dot{\mu}_1\rangle \overset{\Delta}{\longmapsto}{}^* \langle \mathsf{skip},\ \tau_1,\ \dot{\mu}_1\rangle \qquad (+) \\
\overset{\Delta}{\longmapsto} \langle C_1,\ \sigma_1,\ \dot{\mu}_1\rangle & \text{where } \sigma_1 = \tau_1 \restriction x_1 \\
\vdots & \text{containing } n-1 \text{ topmost invocations of } m \\
\overset{\Delta}{\longmapsto} \langle C_n,\ \sigma_n,\ \dot{\mu}_n\rangle & \\
\overset{\Delta}{\longmapsto}{}^* \langle C',\ \sigma',\ \dot{\mu}'\rangle & \text{with no completed topmost invocations of } m.
\end{array}
$$

By hypothesis of Claim B that the trace is m-truncated, it may be that in the final configuration, C' is of the form $m(z); D$ for some z, D.

Recall from the proof sketch (Section 4.4) that, for any configurations $\langle A,\ \tau,\ \dot{\mu}\rangle$ and $\langle A',\ \tau',\ \mu\rangle$, we call them *matching configurations* iff $A = A'$, $\tau = \tau'$, and $\dot{\mu} = \mathit{Extend}(\mu, m, (x : T.B))$ and hence $\mu = \dot{\mu} \restriction m$.

Here, we will construct a trace via $\overset{\Delta\Theta}{\longmapsto}$ that looks as follows:

$$
\begin{array}{ll}
\langle C_0,\ \sigma_0,\ \mu_0\rangle & \\
\overset{\Delta\Theta}{\longmapsto}{}^* \langle m(z_1); C_1,\ \rho_1,\ \mu_1\rangle & \text{matching the configurations above, so } \mu_1 = \dot{\mu}_1 \restriction m \\
\overset{\Delta\Theta}{\longmapsto} \langle C_1,\ \sigma_1,\ \mu_1\rangle & \text{a single step (*)} \\
\vdots & \text{containing } n-1 \text{ additional invocations of } m \\
\overset{\Delta\Theta}{\longmapsto} \langle C_n,\ \sigma_n,\ \mu_n\rangle & \\
\overset{\Delta\Theta}{\longmapsto}{}^* \langle C',\ \sigma',\ \mu'\rangle & \text{again matching configurations.}
\end{array}
$$

In the base case of the induction, $n = 0$, all but one line of the given decomposed trace is empty. That is, we have $\langle C_0, \sigma_0, \dot{\mu}_0\rangle \overset{\Delta}{\longmapsto}{}^* \langle C', \sigma', \dot{\mu}'\rangle$ without any intermediate calls of m (but possibly a call in the last configuration). Using special correspondence Lemma 5.2, we can simply drop m from each environment to get a step by step matching trace $\langle C_0,\ \sigma_0,\ \mu_0\rangle \overset{\Delta\Theta}{\longmapsto}{}^* \langle C',\ \sigma',\ \mu'\rangle$.

For the inductive case, $n > 0$, the initial steps $\langle C_0, \sigma_0, \dot{\mu}_0\rangle \overset{\Delta}{\longmapsto}{}^* \langle m(z_1); C_1, \rho_1, \dot{\mu}_1\rangle$ are matched as in the base case, up to the first invocation of m, in state ρ_1, environment $\dot{\mu}_1$, and with continuation C_1. At that point, we have $\exists \overline{n} \cdot \rho_1 \models Q^{x,\overline{s}}_{z_1,\overline{s}}$, because if there is no such $\overline{n}$ we can we can derive a contradiction: We just established $\langle C_0, \sigma_0, \mu_0\rangle \overset{\Delta\Theta}{\longmapsto}{}^* \langle m(z_1); C_1, \rho_1, \mu_1\rangle$, and if $\neg\exists \overline{n} \cdot \rho_1 \models Q^{x,\overline{s}}_{z_1,\overline{s}}$, then we get $\langle m(z_1); C_1, \rho_1, \mu_1\rangle \overset{\Delta\Theta}{\longmapsto}$ ***p-fault***. Furthermore, by hypothesis of the claim we have $\langle C, \sigma, \mu\rangle \overset{\Delta\Theta}{\longmapsto}{}^* \langle C_0, \sigma_0, \mu_0\rangle$. Putting these together we would obtain a faulting trace from $\langle C, \sigma, \dot{\mu}\rangle$ via $\overset{\Delta\Theta}{\longmapsto}$. This contradicts assumption (27) since $\sigma \models P$. From $\exists \overline{n} \cdot \rho_1 \models Q^{x,\overline{s}}_{z_1,\overline{n}}$, we get $\exists \overline{n} \cdot \upsilon_1 \models Q^{x,\overline{s}}_{x_1,\overline{n}}$ by definition of υ_1 (recall $\upsilon_1 = \mathit{Extend}(\rho_1, x_1, \rho_1(z_1))$, see trace displayed earlier) and the little substitution Lemma 5.1.

Next, we will apply Lemma 7.2 to the trace

$$\langle B^x_{x_1}, \upsilon_1, \dot{\mu}_1\rangle \overset{\Delta}{\longmapsto}{}^* \langle \mathsf{skip}, \tau_1, \dot{\mu}_1\rangle$$

obtained in this decomposition, see (+). Following that trace is one more step that leads to state σ_1. We aim to show that the step to σ_1, marked $(*)$, may be taken via $\overset{\Delta\Theta}{\longmapsto}$. By Lemma 4.10, we have, for any $\overline{n}$,

$$\langle B^x_{x_1}, [\upsilon_1 \mid \overline{s} : \overline{n}], \dot{\mu}_1\rangle \overset{\Delta}{\longmapsto}{}^* \langle \mathsf{skip}, [\tau_1 \mid \overline{s} : \overline{n}], \dot{\mu}_1\rangle.$$

For any $\overline{n}$ such that $[\upsilon_1 \mid \overline{s} : \overline{n}] \models Q^x_{z_1}$ can apply (25) in Lemma 7.2, which lets us conclude

$$[\tau_1 \mid \overline{s} : \overline{n}] \models {Q'}^x_{x_1} \quad \text{and} \quad \upsilon_1 \to \tau_1 \models \eta^x_{x_1}. \tag{30}$$

(By instantiating the Lemma by $\sigma := [\upsilon_1 \mid \overline{s} : \overline{n}]$ and $\sigma' := [\tau_1 \mid \overline{s} : \overline{n}]$.) Furthermore, by Lemma 4.10, we have $\tau_1(\overline{s}) = \upsilon_1(\overline{s})$. We now show that we have a corresponding single step $\langle m(z_1); C_1, \rho_1, \mu_1\rangle \overset{\Delta\Theta}{\longmapsto} \langle C_1, \sigma_1, \mu_1\rangle$. Observe that

$$\tau_1(x_1) = \tau_1(z_1) \tag{31}$$

because

$$\begin{aligned}
&\tau_1(x_1)\\
&= \upsilon_1(x_1) \ \text{ by } \upsilon_1 \to \tau_1 \models \eta^x_{x_1} \text{ and } \mathsf{wr}\, x \text{ not in } \eta \text{ (as the spec is wf)}\\
&= \upsilon_1(z_1) \ \text{ definition of } \upsilon_1\\
&= \tau_1(z_1) \ \text{ by } \upsilon_1 \to \tau_1 \models \eta^x_{x_1} \text{ and } z_1 \in \mathit{Locals} \text{ (as the call is wf) so } \mathsf{wr}\, z_1 \notin \eta \text{ (as the spec is wf).}
\end{aligned}$$

Noting that the locals x, x_1, z_1 are distinct from the specification-only variables $\overline{s}$, we have by (31) that the left conjunct of (30) is equivalent to $[\tau_1 \mid \overline{s} : \overline{n}] \models {Q'}^x_{z_1}$. This is in context $\Gamma', x_1 : T$ for some Γ', but ${Q'}^x_{z_1}$ does not depend on x_1 so we get $[\sigma_1 \mid \overline{s} : \overline{n}] \models^{\Gamma'} {Q'}^x_{z_1}$ as $\sigma_1 = \tau_1 \lceil x_1$. So σ_1 is a candidate outcome of the call $m(z_1)$ in terms of the post condition in the semantics of context procedure call; it remains to check the effect condition. Note that x may occur in η, but $\mathsf{wr}\, x$ cannot, so the use of x in η is only for its r-value. So the second conjunct of (30) can be rewritten using (31) to yield

$$\upsilon_1 \to \tau_1 \models \eta^x_{z_1}$$

and here x_1 does not occur in the effect so we can drop x_1 from υ_1 and τ_1 to get

$$\rho_1 \to \sigma_1 \models \eta^x_{z_1}.$$

This completes the argument that σ_1 is among the possible outcomes of the call $m(z_1)$ via $\overset{\Delta\Theta}{\longmapsto}$, that is, the step $\langle m(z_1);\, C_1,\, \rho_1,\, \mu_1\rangle \overset{\Delta\Theta}{\longmapsto} \langle C_1,\, \sigma_1,\, \mu_1\rangle$, marked $(*)$.

The next step yields $\langle C_1,\, \sigma_1,\, \mu_1\rangle$. What remains from this configuration onward is a trace with $n-1$ completed invocations of m, from a configuration reachable from $\langle C,\, \sigma,\, \dot{\mu}\rangle$. Moreover, we have $\langle C,\, \sigma,\, \dot{\mu}\rangle \overset{\Delta\Theta}{\longmapsto}{}^{*} \langle C_1,\, \sigma_1,\, \mu_1\rangle$. So we can apply the inductive hypothesis to the trace $\langle C_1,\, \sigma_1,\, \dot{\mu}_1\rangle \overset{\Delta}{\longmapsto}{}^{*} \langle C',\, \sigma',\, \dot{\mu}'\rangle$.

That concludes the proof of Claim B and the proof of Lemma 7.4. □

The preceding proof of used Lemma 7.2. The following proof of Lemma 7.2 has a similar structure to the proof of Lemma 7.4, but with additional intricacies—for the bound on calling depth.

PROOF OF LEMMA 7.2. By Lemma 5.3 items (1) and (2), it suffices to prove (24–26) in depth-bounded semantics. That is, we will prove the following by induction on k: For all $k \geq 0$, we have (I)–(III) for all $u, \Gamma', \sigma, \dot{\mu}, \sigma', \dot{\mu}'$ and k' (satisfying the conditions in the Lemma: u is in *Locals* but not in $dom(\Gamma) \cup \{x\}$; Γ' is any extension of $\Gamma, u : T, m : (x : T)$; σ is any Γ'-state; $\dot{\mu}$ is any Γ'-environment such that $\dot{\mu}(m) = (x : T.B)$; $\overline{s}$ lists the specification-only variables in Q, Q', η; and $\sigma \models Q^x_u$.

(I) It is not the case that $\langle B^x_u,\, \sigma,\, \dot{\mu}\rangle^k \overset{\Delta}{\longmapsto}{}^{*}$ *(p-)fault*

(II) $\langle B^x_u,\, \sigma,\, \dot{\mu}\rangle^k \overset{\Delta}{\longmapsto}{}^{*} \langle \mathsf{skip},\, \sigma',\, \dot{\mu}\rangle^k$ implies $\sigma' \models {Q'}^x_u$ and $\sigma \rightarrow \sigma' \models \eta^x_u$ for all σ'

(III) Every reachable step from $\langle B^x_u,\, \sigma,\, \dot{\mu}\rangle^k$ via $\overset{\Delta}{\longmapsto}$ respects every L in Δ.

(The judgment is for module N, but for (III) we do not need to say $L \neq N$ because by hypothesis of Lemma 7.2 we have $N \notin \Delta$.)

Note that (24–26) follow from (I)–(III) by Lemma 5.3. Conditions (I)–(III) correspond to conditions (i)–(iii) in the proof of rule LINK above, but with B^x_u in place of C. And now (I)–(III), with smaller k, will be our main induction hypothesis, used at points where the LINK proof relies on Lemma 7.2. For fixed $k, \sigma, \sigma', \dot{\mu}$ we will prove the following:

Claim A′. For all $C', \sigma', \dot{\mu}', k'$, and m-truncated traces $\langle B^x_u,\, \sigma,\, \dot{\mu}\rangle^k \overset{\Delta}{\longmapsto}{}^{*} \langle C',\, \sigma',\, \dot{\mu}'\rangle^{k'}$ we have

(a) $\langle B^x_u,\, \sigma,\, \mu\rangle^k \overset{\Delta\Theta}{\longmapsto}{}^{*} \langle C',\, \sigma',\, \mu'\rangle^{k'}$, where $\mu' = \dot{\mu}' \restriction m$

(b) if $C' = m(z); D$ for some z, D then there are values $\overline{n}$ with $\sigma' \models Q^{x,\overline{s}}_{z,\overline{n}}$

Claim A′ implies (I) by an argument very similar to the soundness argument for LINK, where Claim A implies (i). The key difference is that in one case we have to consider behavior of an instance of B where C invokes m. In particular, we show that instance does not fault, by appeal to Lemma 7.2. Here, we have to consider a recursive invocation of m in B^x_u, and thus an instance B^x_w—in a configuration with bound less than k. The bound is less than k because the configuration is reached from $\langle B^x_u,\, \sigma,\, \dot{\mu}\rangle^k$ and B has no end-markers. So we can appeal to the induction hypothesis, specifically (I).

It was by carefully checking the details that we arrived at the current formulations in this proof and the soundness proof for LINK. But since everything else is the same, we do not include details here.

Claim A′ implies (II) by an argument very similar to the soundness argument for LINK, where Claim A implies (ii) very directly. Here, we have the same direct argument from Claim A′ using the primary hypothesis (23), the correctness of B.

Claim A′ implies (III) by an argument similar to the soundness argument for LINK where Claim A implies (iii). There, in one subcase, we appeal to Lemma 7.2 to get that a reachable step, executing inside an instance of B respects L. Here, the reachable step is from a configuration with bound less than k, as it is reached from $\langle B^x_u,\ \sigma,\ \dot\mu\rangle^k$, so we can use the induction hypothesis (III).

It remains to prove Claim A′. For this, we use the following.

Claim B′. For any $n \geq 0$, we have the following. For all $C_0, \sigma_0, \dot\mu_0, k_0, C', \sigma', \dot\mu', k'_0$, and for any m-truncated trace

$$\langle B^x_u,\ \sigma,\ \dot\mu\rangle^k \overset{\Delta}{\longmapsto}{}^* \langle C_0,\ \sigma_0,\ \dot\mu_0\rangle^{k_0} \overset{\Delta}{\longmapsto}{}^* \langle C',\ \sigma',\ \dot\mu'\rangle^{k'},$$

if the trace $\langle C_0,\ \sigma_0,\ \dot\mu_0\rangle^{k_0} \overset{\Delta}{\longmapsto}{}^* \langle C',\ \sigma',\ \dot\mu'\rangle^{k'}$ has exactly n completed topmost calls of m, and we have $\langle B^x_u,\ \sigma,\ \mu\rangle^k \overset{\Delta\Theta}{\longmapsto}{}^* \langle C_0,\ \sigma_0,\ \mu_0\rangle^{k_0}$ where $\mu_0 = \dot\mu_0 \restriction m$ and $\mu' = \dot\mu' \restriction m$, then we have $\langle C_0,\ \sigma_0,\ \mu_0\rangle^{k_0} \overset{\Delta\Theta}{\longmapsto}{}^* \langle C',\ \sigma',\ \mu'\rangle^{k'}$.

To prove Claim A′(a), instantiate the Claim with $C_0, \sigma_0, \dot\mu_0 := B^x_u, \sigma, \dot\mu$.

To prove Claim A′(b), suppose $\langle B^x_u,\ \sigma,\ \dot\mu\rangle^k \overset{\Delta}{\longmapsto}{}^* \langle C',\ \sigma',\ \dot\mu'\rangle^{k'}$ and $C' = m(z); D$ for some z, D. By Claim A′(a), we have $\langle B^x_u,\ \sigma,\ \mu\rangle^k \overset{\Delta\Theta}{\longmapsto}{}^* \langle C',\ \sigma',\ \mu'\rangle^{k'}$. Suppose, for the sake of contradiction, that there is no $\overline{n}$ such that $\sigma' \models Q^{x,\overline{s}}_{z,\overline{n}}$. Then, by semantics, we have $\langle C',\ \sigma',\ \mu'\rangle^{k'} \overset{\Delta\Theta}{\longmapsto}$ *p-fault* and thus $\langle B^x_u,\ \sigma,\ \mu\rangle^k \overset{\Delta\Theta}{\longmapsto}{}^*$ *p-fault*. Hence, by Lemma 5.3, we have $\langle B^x_u,\ \sigma,\ \mu\rangle \overset{\Delta\Theta}{\longmapsto}{}^*$ *p-fault*. But this contradicts the hypothesis (23) of the lemma we are proving. Specifically, from (23) we obtain $\Delta, \Theta \models^{\Gamma, u:T}_N \{\ Q^x_u\ \}\ B^x_u\ \{\ Q'^x_u\ \}\ [\eta^x_u]$ by renaming Lemma 7.3, and then by soundness of rule EXTENDCTX we get

$$\Delta, \Theta \models^{\Gamma'}_N \{\ Q^x_u\ \}\ B^x_u\ \{\ Q'^x_u\ \}\ [\eta^x_u]\,, \tag{32}$$

where Γ' is the type of the configuration with σ according to the hypothesis of Lemma 7.2. Nonexistence of $\overline{n}$ would contradict (32) since we assumed at the outset that $\sigma \models Q^x_u$. So Claim A′ is proved.

PROOF OF CLAIM B′. The argument is similar to the proof of Claim B used in proving Lemma A for soundness of LINK, except that where the latter appeals to Lemma 7.2, here we appeal to the main induction hypothesis (II). The details involve careful manipulation of the bound in accord with depth-bounded semantics. Using Lemma 5.5, we obtain intermediate states ρ_i, τ_i, σ_i and environments $\dot\mu_i$ such that

$$\begin{array}{ll}
\langle C_0,\ \sigma_0,\ \dot\mu_0\rangle^{k_0} & \\
\overset{\Delta}{\longmapsto}{}^* \langle m(z_1); C_1,\ \rho_1,\ \dot\mu_1\rangle^{k_1} & \text{with no invocations of } m \\
\overset{\Delta}{\longmapsto} \langle B^x_{x_1}; \mathsf{ecall}(x_1); C_1,\ \upsilon_1,\ \dot\mu_1\rangle^{k_1-1} & \text{where } \upsilon_1 = \mathit{Extend}(\rho_1, x_1, \rho_1(z_1)) \text{ and } x_1 \text{ fresh} \\
\overset{\Delta}{\longmapsto}{}^* \langle \mathsf{ecall}(x_1); C_1,\ \tau_1,\ \dot\mu_1\rangle^{k_1-1} & \text{where } \langle B^x_{x_1},\ \upsilon_1,\ \dot\mu_1\rangle^{k_1-1} \overset{\Delta}{\longmapsto}{}^* \langle \mathsf{skip},\ \tau_1,\ \dot\mu_1\rangle^{k_1-1} \quad (+) \\
\overset{\Delta}{\longmapsto} \langle C_1,\ \sigma_1,\ \dot\mu_1\rangle^{k_1} & \text{where } \sigma_1 = \tau_1 \restriction x_1 \\
\vdots & \text{containing } n-1 \text{ topmost invocations of } m \\
\overset{\Delta}{\longmapsto} \langle C_n,\ \sigma_n,\ \dot\mu_n\rangle^{k_n} & \\
\overset{\Delta}{\longmapsto}{}^* \langle C',\ \sigma',\ \dot\mu'\rangle^{k'} & \text{with no completed invocations of } m
\end{array}$$

By hypothesis of the claim that the trace is m-truncated, it may be that in the final configuration, C' is of the form $m(z); D$ for some z, D.

Concerning the bounds, these configurations are reached from $\langle B_u^x,\, \sigma,\, \dot{\mu}\rangle^k$ according to the statement of Claim B′, so we have $k_i \le k$ for all i.

Here, we will construct a trace via $\overset{\Delta\Theta}{\longmapsto}$ that looks as follows.

$$
\begin{array}{ll}
\langle C_0,\, \sigma_0,\, \mu_0\rangle^{k_0} & \\
\overset{\Delta\Theta}{\longmapsto}{}^*\langle m(z_1); C_1,\, \rho_1,\, \mu_1\rangle^{k_1} & \text{matching steps as per Lemma 5.2} \\
\overset{\Delta\Theta}{\longmapsto} \langle C_1,\, \sigma_1,\, \mu_1\rangle^{k_1} & \text{a single step justified in the following text} \\
\vdots & \text{containing } n-1 \text{ additional invocations of } m \\
\overset{\Delta\Theta}{\longmapsto} \langle C_n,\, \sigma_n,\, \mu_n\rangle^{k_n} & \\
\overset{\Delta\Theta}{\longmapsto}{}^*\langle C',\, \sigma',\, \mu'\rangle^{k'} & \text{again matching steps}
\end{array}
$$

In the base case of the induction, $n = 0$, all but one line of the given decomposed trace is empty. That is, we have $\langle C_0,\, \sigma_0,\, \dot{\mu}_0\rangle^{k_0} \overset{\Delta}{\longmapsto}{}^* \langle C',\, \sigma',\, \dot{\mu}'\rangle^{k'}$ without any intermediate calls of m. Using Lemma 5.2, we can simply drop m from each environment to get a step-by-step matching trace $\langle C_0,\, \sigma_0,\, \mu_0\rangle^{k_0} \overset{\Delta\Theta}{\longmapsto}{}^* \langle C',\, \sigma',\, \mu'\rangle^{k'}$.

For the inductive case, $n > 0$, the initial steps $\langle C_0,\, \sigma_0,\, \dot{\mu}_0\rangle^{k_0} \overset{\Delta}{\longmapsto}{}^* \langle m(z_1); C_1,\, \rho_1,\, \dot{\mu}_1\rangle^{k_1}$ are matched as in the base case, up to the first invocation of m, in state ρ_1, environment $\dot{\mu}_1$, and with continuation C_1. At that point, we have that there exists value(s) $\overline{n}$ such that $\rho_1 \models Q^{x,\overline{s}}_{z_1,\overline{n}}$, because otherwise we get a contradiction as follows. We just established $\langle C_0,\, \sigma_0,\, \mu_0\rangle^{k_0} \overset{\Delta\Theta}{\longmapsto}{}^* \langle m(z_1); C_1,\, \rho_1,\, \mu_1\rangle^{k_1}$ and if $\neg(\exists \overline{n} \cdot \rho_1 \models Q^{x,\overline{s}}_{z_1,\overline{n}})$, then we would have $\langle m(z_1); C_1,\, \rho_1,\, \mu_1\rangle^{k_1} \overset{\Delta\Theta}{\longmapsto}$ *p-fault*. Furthermore, by hypothesis of the Claim, we have $\langle B_u^x,\, \sigma,\, \mu\rangle^k \overset{\Delta\Theta}{\longmapsto}{}^* \langle C_0,\, \sigma_0,\, \mu_0\rangle^{k_0}$. Putting these together, we would obtain a faulting trace from $\langle B_u^x,\, \sigma,\, \dot{\mu}\rangle^k$ via $\overset{\Delta\Theta}{\longmapsto}$, and thus (by Lemma 5.3) from $\langle B_u^x,\, \sigma,\, \dot{\mu}\rangle$ in nonbounded semantics via $\overset{\Delta\Theta}{\longmapsto}$, which contradicts assumption $\sigma \models^{\Gamma'} Q_u^x$ and assumption (23) or rather its consequence (32).

Consider any $\overline{n}$ such that $[\rho_1 \,|\, \overline{s} : \overline{n}] \models Q^x_{z_1}$ (existence of such having been established previously). Now $[\rho_1 \,|\, \overline{s} : \overline{n}] \models Q^x_{z_1}$ implies $[\upsilon_1 \,|\, \overline{s} : \overline{n}] \models Q^x_{x_1}$ by definition of $\upsilon_1 = \mathit{Extend}(\rho_1, x_1, \rho_1(z_1))$. (Noting that $\overline{s}$ is distinct from x, x_1, z_1.) So we have $[\upsilon_1 \,|\, \overline{s} : \overline{n}] \models Q^x_{x_1}$. Because $k_1 - 1$ is less than k, we can apply the main induction hypothesis (II) (instantiated by $u := x_1$ etc.) to the trace

$$\langle B^x_{x_1},\, \upsilon_1,\, \dot{\mu}_1\rangle^{k_1} \overset{\Delta}{\longmapsto}{}^* \langle \mathsf{skip},\, \tau_1,\, \dot{\mu}_1\rangle^{k_1}$$

whose existence is established previously, see (+). Now we proceed as in the proof of Claim B, but using (23), or rather its consequence (32), we obtain $[\sigma_1 \,|\, \overline{s} : \overline{n}] \models Q'^x_{z_1}$ and $\rho_1 \to \sigma_1 \models \eta^x_{z_1}$. This completes the argument that σ_1 is among the possible outcomes of the call $m(z_1)$ via $\overset{\Delta\Theta}{\longmapsto}$, justifying the single step in the trace we are constructing.[22]

The next step yields $\langle C_1,\, \sigma_1,\, \mu_1\rangle$. What remains from this configuration onward is a trace with $n-1$ completed invocations of m, from a configuration reachable from $\langle B_u^x,\, \sigma,\, \dot{\mu}\rangle^k$. Moreover, we have $\langle B_u^x,\, \sigma,\, \dot{\mu}\rangle^k \overset{\Delta\Theta}{\longmapsto}{}^* \langle C_1,\, \sigma_1,\, \mu_1\rangle^{k_1}$. So we can apply the

[22]One may wonder here whether the argument is sound in case of an unsatisfiable specification. The point is that the induction hypothesis (II) is conditioned on termination. In case of a call in a state from which the specification is unsatisfiable, there will be no terminated configuration that needs to be matched via $\overset{\Delta}{\longmapsto}$.

inductive hypothesis—of the inner induction on n—to the trace $\langle C_1,\ \sigma_1,\ \dot{\mu}_1\rangle^{k_1} \overset{\Delta}{\longmapsto}{}^{*}$ $\langle C',\ \sigma',\ \dot{\mu}'\rangle^{k'}$. That concludes the proof of Claim B′ and of Lemma 7.2. □

8. VERIFICATION OF THE EXAMPLES

The primary goal of this section is to round off the examples in Section 2 other than module SET, whose treatment was finished in Section 6.3. We will provide the effect specifications and dynamic boundaries for the examples and illustrate features of the novel proof rules, particularly rules MISMATCH and the context introduction rules. We will show the logic at play by working out the verification of a client in some detail. The verification will also be justified by way of derived proof rules for constructors and private procedures, to which we turn now.

8.1. Some Derived Rules

Constructors. In Java-like languages, constructors are distinguished procedures named after a corresponding class and invoked only on newly allocated instances. In contrast, this article does not consider constructors to be distinguished in this manner.

Our purpose is to derive a proof rule for constructors where a constructor has the form $\mathsf{new}\ K(y)$, where K is a class name. We will consider the form $x := \mathsf{new}\ K(y)$ to be sugar for $x := \mathsf{new}\ K; K(x, y)$. The parameter x gets substituted for K's implicit parameter self during constructor call. Note that we are stretching slightly the procedure syntax in Figure 6 in that K now has two parameters instead of one. For simplicity, we consider that K has a single field $f : T$—the generalization to multiple fields is obvious.

We proceed to derive the following CONSTRUCTOR rule for the constructor command $x := \mathsf{new}\ K(y)$ from the sequential composition $x := \mathsf{new}\ K; K(x, y)$ thus showing that the sugaring here is sensible.

$$\textsc{Constructor}\ \frac{\begin{array}{c}\Delta \text{ is } \{P \wedge \mathsf{self} \notin r \wedge \mathsf{self}.f = \mathit{default}(T)\}K(\mathsf{self}: K, u: T)\{P'\}[\mathsf{self}.f, \varepsilon] \\ x \in \mathit{Locals} \qquad y \not\equiv x \qquad y \not\equiv \mathsf{alloc} \qquad \mathsf{alloc} \text{ does not occur in } P \\ P^u_y \Rightarrow \mathit{bnd}(\mathit{mdl}(K))\ \text{`/.}\ \mathsf{wr}\, x, \mathsf{alloc} \qquad \varepsilon^u_y \text{ is } P^u_y/\mathsf{wr}\, x, \mathsf{alloc}\text{-immune}\end{array}}{\Delta \vdash^{\Gamma, x:K}_{M} \{\, P^u_y \,\}\ x := \mathsf{new}\ K(y)\ \{\, P'^{\mathsf{self},u}_{x,y} \,\}\ [\varepsilon^u_y,\ x,\ \mathsf{alloc},\ \mathsf{fr}\{x\}]}$$

Note that, because of the premise $x \in \mathit{Locals}$, we have by well-formedness (8) of K's specification that x does not occur in P. Moreover, ε cannot write any locals. In particular, ε cannot write x and it cannot write self because, being a procedure parameter, self is in Locals.

The rule depicts several facets of a constructor call. It shows how constructor specifications can be instantiated by substituting for the formal parameter u and the implicit parameter self. The rule reflects allocation by way of alloc in the frame condition. Note that constructor bodies are not precluded from performing allocations, so ε can contain wr alloc. For the immunity of ε^u_y, sufficient conditions are $\mathsf{wr}\, x$ is not in ε, $y \not\equiv x$, and alloc does not appear in ε in the form $\mathsf{wr}\,\mathsf{alloc}\text{`}k$ for some field k. The rule masks writes to fields of self because self is a freshly allocated object. In light of this, one can sensibly omit writes to self-fields in constructor specifications as we have done in the effect specifications for the Set constructor in Figure 7 and in the ones to come later in this section. The preconditions $\mathsf{self} \notin r$ and $\mathsf{self}.f = \mathit{default}(T)$ are used in verification of the body of the constructor.

Derivation. We are permitted to assume the premises of CONSTRUCTOR. From ALLOC by CONSEQ, we have

$$\vdash \{\, \mathit{true} \,\}\ x := \mathsf{new}\ K\ \{\, x.f = \mathit{default}(T) \,\}\ [x, \mathsf{alloc}, \mathsf{fr}\{x\}]\,.$$

By FRTOPOST, we get

$$\vdash \{\, true \,\}\ x := \mathsf{new}\ K\ \{\, x.f = \mathit{default}(T) \wedge \{x\} \cap r = \varnothing \,\}\ [x, \mathsf{alloc}, \mathsf{fr}\,\{x\}]$$

whence by CONSEQ

$$\vdash \{\, true \,\}\ x := \mathsf{new}\ K\ \{\, x.f = \mathit{default}(T) \wedge x \notin r \,\}\ [x, \mathsf{alloc}, \mathsf{fr}\,\{x\}]$$

Let δ be a list of read effects such that $\vdash \delta\ \mathsf{frm}\ P^u_y$. We can assume that δ does not contain $\mathsf{rd}\,x$ because x is not in P (as argued earlier) and $y \not\equiv x$ (premise). Likewise, δ does not need to contain $\mathsf{rd}\,\mathsf{alloc}$ because alloc does not occur in P and $y \not\equiv \mathsf{alloc}$ (premises). Hence $\delta \not\downarrow \mathsf{wr}\,x, \mathsf{alloc} \Leftrightarrow true$. So by FRAME, conjoining P^u_y, we have

$$\vdash \{\, P^u_y \,\}\ x := \mathsf{new}\ K\ \{\, P^u_y \wedge x.f = \mathit{default}(T) \wedge x \notin r \,\}\ [x, \mathsf{alloc}, \mathsf{fr}\,\{x\}]\,.$$

By CTXINTRO, owing to premise $P^u_y \Rightarrow \mathit{bnd}(\mathit{mdl}(K)) \not\downarrow \mathsf{wr}\,x, \mathsf{alloc}$,

$$\Delta \vdash_M \{\, P^u_y \,\}\ x := \mathsf{new}\ K\ \{\, P^u_y \wedge x.f = \mathit{default}(T) \wedge x \notin r \,\}\ [x, \mathsf{alloc}, \mathsf{fr}\,\{x\}]\,. \tag{33}$$

Using CALL, we get

$$\Delta \vdash_M \{\, P^u_y \wedge x.f = \mathit{default}(T) \wedge x \notin r \,\}\ K(x, y)\ \{\, P'^{\mathsf{self},u}_{x,y} \,\}\ [x.f,\ \varepsilon^u_y]\,. \tag{34}$$

We now use SEQ on (33) and (34) to obtain the conclusion of CONSTRUCTOR. This instance of SEQ has one immunity condition, ε^u_y is $P^u_y/(\mathsf{wr}\,x, \mathsf{alloc})$-immune that is an explicit premise of CONSTRUCTOR. The other immunity condition is

$$\{x\} \text{ is } P^u_y \wedge x.f = \mathit{default}(T) \wedge x \notin r/(\varepsilon^u_y, \mathsf{wr}\,x.f)\text{-immune},$$

which holds because $y \not\equiv x$ (premise) and because ε cannot write x as argued earlier.

Finally, rule SEQ takes care of removing the effect $\mathsf{wr}\,x.f$ because of $\mathsf{fr}\,\{x\}$.

In a language that treats constructors specially, every allocation is followed by a constructor call. One advantage of this treatment is the maintenance of invariants over allocated objects of a certain type—the invariants can be established by the constructor body. For example, were we to formalize constructors as obligatory calls in connection with allocation, it would also be the case that all Set instances are in pool, that is, $\mathsf{alloc}\,/\,\mathit{Set} \subseteq \mathit{pool}$. Instead, we permit allocation to occur without a constructor being called. Consequently, care must be taken because invariants that quantify over all allocated objects are at risk of falsification owing to mere allocation (leading to the pool idiom used earlier). Another consequence of treating constructors in the manner in this article is that the preconditions $\mathsf{self} \notin r$ and $\mathsf{self}.f = \mathit{default}(T)$ can be provided by a reasoning system directly but not appear in public specifications. For that reason, they are called *free preconditions* [Leino 2008]. Because we do not treat constructors specially, the free preconditions are made explicit in the premise of the rule.

Private Procedures. Here we derive proof rule PRIVATEPROC that caters for a situation when a module implementation uses some procedures that are not present in the module's API. For example, in Figure 3, the $\mathit{Observer}$ constructor appears in module SO's API. Its implementation, called $\mathit{register}$, is absent from SO's API. Such procedures do not necessarily rely on (or preserve) the module invariant; and if they do there is no harm in making it explicit in their specifications, which are only used for reasoning within the module implementation.

Derivation. Consider linking procedure m to client C where m may call a procedure p not in the API of $\mathit{mdl}(m)$, that is, p is module-scoped or private to $\mathit{mdl}(m)$. We have $\mathit{mdl}(m) = \mathit{mdl}(p)$. The generalization to multiple such module-scoped procedures is obvious so we will consider a single m and a single p.

As in the LINK rule let $\Theta \mathrel{\widehat{=}} \{Q\}m(x:T)\{Q'\}[\eta]$. For the private procedure, let $\Pi \mathrel{\widehat{=}} \{R\}p(y:U)\{R'\}[\omega]$. Let $mdl(m) = N = mdl(p)$. Then, $N \in \Theta$.

$$\textsc{PrivateProc}\ \frac{\Delta, \Theta \vdash_M^{\Gamma} \{\,P\,\}\ C\ \{\,P'\,\}\ [\varepsilon] \qquad \Delta, \Pi, \Theta \vdash_N^{\Gamma, x:T} \{\,Q\,\}\ B_m\ \{\,Q'\,\}\ [\eta] \qquad \Delta, \Pi, \Theta \vdash_N^{\Gamma, y:U} \{\,R\,\}\ B_p\ \{\,R'\,\}\ [\omega] \qquad N \notin \Delta}{\Delta \vdash_M^{\Gamma} \{\,P\,\}\ \mathsf{letrec}\ m(x:T) = B_m;\ p(y:U) = B_p\ \mathsf{in}\ C\ \{\,P'\,\}\ [\varepsilon]}.$$

PRIVATEPROC is derivable using LINK. The main observation is that from $\Delta, \Theta \vdash_M^{\Gamma} \{\,P\,\}\ C\ \{\,P'\,\}\ [\varepsilon]$ one can obtain $\Delta, \Pi, \Theta \vdash_M^{\Gamma} \{\,P\,\}\ C\ \{\,P'\,\}\ [\varepsilon]$ by CTXINTROIN: $N \in \Theta$ implies $N \in (\Delta, \Theta)$ that permits application of CTXINTROIN. Now LINK can be used on the judgments for C, B_m and B_p—because all have procedure contexts Δ, Π, Θ.

Rule MISMATCH can be generalized in a similar manner by extension with a single procedure p that is private. Let $\Theta \mathrel{\widehat{=}} \{Q\}m(x:T)\{Q'\}[\eta]$ and $\Pi \mathrel{\widehat{=}} \{R\}p(y:U)\{R'\}[\omega]$. Let $mdl(m) = N = mdl(p)$.

$$\textsc{GenMismatch}\ \frac{\begin{array}{c}\Delta, \Theta \vdash_M \{\,P\,\}\ C\ \{\,P'\,\}\ [\varepsilon] \\ P \Rightarrow I \qquad \Delta, \Pi, (\Theta \between I) \vdash_N \{\,Q \wedge I\,\}\ B_m\ \{\,Q' \wedge I\,\}\ [\eta] \\ \Delta, \Pi, (\Theta \between I) \vdash_N \{\,R\,\}\ B_p\ \{\,R'\,\}\ [\omega] \\ \vdash bnd(N)\ \mathsf{frm}\ I \qquad N \neq M \qquad mdl(q) \not\leq N\ \text{for all}\ q\ \text{in}\ \Delta\end{array}}{\Delta \vdash_M^{\Gamma} \{\,P\,\}\ \mathsf{letrec}\ m(x:T) = B_m;\ p(y:U) = B_p\ \mathsf{in}\ C\ \{\,P'\,\}\ [\varepsilon]}.$$

Rule GENMISMATCH allows private procedures that neither rely on nor preserve the hidden invariant I. That is, the rule conjoins I to Θ but not to Π.

To derive this rule, we follow the derivation of MISMATCH in Figure 12. First, from $\Delta, \Theta \vdash_M \{\,P\,\}\ C\ \{\,P'\,\}\ [\varepsilon]$, we derive $\Delta, (\Theta \between I) \vdash_M \{\,P \wedge I\,\}\ C\ \{\,P' \wedge I\,\}\ [\varepsilon]$ by SOF. Next side condition $N \in \Theta$ implies $N \in \Delta, (\Theta \between I)$. This allows us to derive $\Delta, \Pi, (\Theta \between I) \vdash_M \{\,P \wedge I\,\}\ C\ \{\,P' \wedge I\,\}\ [\varepsilon]$ by CTXINTROIN. Finally LINK, together with side condition $P \Rightarrow I$ and CONSEQ can be used on C, B_m and B_p to obtain the conclusion.

8.2. Module *MM*

This section gives the effect specifications and the dynamic boundary for the toy memory manager of Section 2.2 and makes high-level remarks on how MM works with clients.

We specify the effects for procedure *alloc* to be wr result, *freed*, *flist*, *count*, alloc, *freed*`*nxt*. For *free*($n$ : *Node*) the effects are wr *freed*, *flist*, *count*, *freed*`*nxt*. The read effects rd *freed*, *flist*, *count*, *freed*`*nxt* can be used to frame the hidden invariant I_{mm}. In a practical implementation, ordinary scoping could be used to hide effects on the module variables *flist* and *count* from clients and the ghost *freed* could be "spec-public" (as in JML [Leavens et al. 2003]) that is, not accessible in code outside module MM but accessible in specifications. So the dynamic boundary of MM could be simply *freed*`*nxt*.

Using the specifications in Section 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*, and thus it respects the dynamic boundary. So the client can be linked with *alloc* and *free* using rule MISMATCH. By contrast with the *SET* example in Section 6.3 where we use a public invariant, R, to verify that client (1) respects the dynamic boundary θ_{set}, here it is the procedure specifications themselves that support reasoning about the dynamic boundary. Suppose we consider a client that contains the assignment $y.nxt := null$ just after $y := alloc()$ in (2); although this writes a *nxt* field, the object is outside *freed* according to the specification of *alloc*. Hence this client is verifiable.

Method	Effects
Subject()	wr $sopool$
update(n)	wr self.val, self.O‘$cache$
get()	
register(b)	wr self.(O, obs), $b.(nxto, cache)$
Observer(s)	wr $s.(O, obs)$, $sopool$
notify()	wr self.$cache$

Fig. 16. Effects of methods in SO.

8.3. Module *SO*

This section adresses module SO from Section 2.3, giving its effect specifications, its dynamic boundary, and a client. Figure 16 gives the effects of the methods in SO as they would appear in an API. The effects are the same as in Part I except that the effects for constructors $Subject$ and $Observer$ take into account the write to $sopool$. Furthermore, the effects for $Subject$ and $Observer$ do not mention updates to self-fields in accord with rule CONSTRUCTOR. Also in accord with this rule the verification of the body of $Subject$ must be done with respect to the additional effects wr self.(O, obs, val) and the free preconditions (cf. Section 8.1)

$$\mathsf{self}.obs = \mathsf{null} \wedge \mathsf{self}.val = 0 \wedge \mathsf{self}.O = \varnothing \wedge \mathsf{self} \notin sopool$$

conjoined to $X = sopool \wedge sopool\text{‘}(obs, O) \subseteq sopool$ (Figure 4).

We will write $P(s)$ and $P'(s)$ respectively for the pre- and postcondition of $Observer$ in Figure 4. We will write $\varepsilon_o(s)$ for the public effects of $Observer(s)$ in Figure 16. Similar to $Subject$ above, the body of the $Observer$ constructor must be verified with respect to the effects $\varepsilon_o(s)$ together with the effects wr self.$(cache, nxto, sub)$. We will denote this expanded set of effects by $\varepsilon_o^+(s)$. The body will also be verified with respect to $P(s)$ conjoined with the free preconditions PS, where

$$PS \;\widehat{=}\; \mathsf{self}.sub = \mathsf{null} \wedge \mathsf{self}.cache = 0 \wedge \mathsf{self}.nxto = \mathsf{null} \wedge \mathsf{self} \notin s.O \wedge \mathsf{self} \notin sopool$$

The effects of $update$ and $Observer(s)$ in Figure 16 suffice to verify this client (from Section 2.3):

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

with the precondition $s.val = 0 \wedge t.val = 5$ and the postcondition $t.val = 5$ (or postcondition $p.cache = 5$). The verification relies on separation between subjects (i.e., $s \neq t$), the specifications of $Observer$ and $update$, and rule FRAME.

For verification of the implementations in Section 2.3, we use FRAME to exploit per-subject separation, similar to the Set example. Then, rule MISMATCH can be used to link the client.

For framing purposes in the verification of the body of $Observer$, we generalize the hidden invariant I_{so} in the following manner.

$$I_{so}(m : \mathsf{rgn}, n : \mathsf{rgn}) \;\widehat{=}\; \begin{array}{l}(\forall s : Subject \in sopool \setminus m \cdot SubH(s)) \wedge \\ (\forall o : Observer \in sopool \setminus n \cdot o.sub \neq \mathsf{null} \wedge o \in o.sub.O).\end{array}$$

Now we observe that $I_{so} \;\widehat{=}\; I_{so}(\varnothing, \varnothing)$. The verification of the body of $Observer$ will split off a cluster containing the subject u and the current observer self and establish $SubH(\mathsf{self})$ and $\mathsf{self}.sub \neq \mathsf{null} \wedge \mathsf{self} \in \mathsf{self}.sub.O$. These local properties of the cluster will then be used to establish the global invariant I_{so} (cf. the establishment of I_{set} in Section 3.2).

$$\begin{array}{l}\mathsf{ghost}\ \ sopool, ocpool : \mathsf{rgn};\\ \mathsf{letrec}\ sigs(\Delta_{so}) = \ldots; sigs(\Pi_{so}) = \ldots\ \mathsf{in}\\ \mathsf{letrec}\ sigs(\Delta_{oc}) = \ldots\ \mathsf{in}\\ s := \mathsf{new}\ Subject(); x := \mathsf{new}\ ObsColl(s, null);\end{array}$$

Fig. 17. Linked *ObsCall* client. The ellipses elide the implementations of the methods in Δ_{so}, Π_{so} and Δ_{oc}.

Observe that in the invariant I_{so} the quantified variables s and o range over allocated objects in *sopool* rather than in alloc (cf. I_{set}). The use of alloc as the range of quantification would mean that the footprint of I_{so} would include variable alloc. Thus the dynamic boundary, $bnd(SO)$, could be rd alloc, alloc$'(sub, obs, nxto, O)$, but this choice is problematic. Client code like $p := \mathsf{new}\ Observer(t)$ writes not only the effects specified for *Observer* but also p and alloc. Thus, a dynamic boundary that includes alloc would be violated by such client code. Therefore, we consider

$$bnd(SO) \ \widehat{=}\ \mathsf{rd}\ sopool, sopool'(sub, obs, nxto, O)$$

This illustrates that the procedures of a module may write within the module's boundary, and these effects will be explicit in the public specifications, for example, wr *sopool* of *Subject* and $Observer(s)$. However, the obligation of a client is that its own direct writes should not violate this boundary. The separator formula in the side condition of proof rule CTXINTRO captures this obligation.

8.4. Module *OC*: Illustrating New Proof Rules

This section will show the use of rule MISMATCH and the different flavors of the CTXINTRO rule on an example that involves nested modules. Recall the example of overlapping data structures from Section 2.4 which we now develop in some detail. We will consider the verification of nested clients. The detailed proofs are intricate and we present the highlights of the proofs in this section.

The public effects of *ObsColl* are $\varepsilon_c(s, oc)$ where

$$\varepsilon_c(s, oc) \ \widehat{=}\ \mathsf{wr}\ s.(O, obs), sopool, oc.prev, oc.prev'next, ocpool$$

In particular, $\varepsilon_c(s, \mathsf{null})$ is wr $s.(O, obs), sopool$. We will write Q and Q', respectively, for the pre- and postcondition of $ObsColl(s, oc)$ (Figure 5). Using P, P' for the *Observer* specification as in Section 8.3, we have $Q \Leftrightarrow P(s) \wedge Y = ocpool \wedge U(oc, ocpool)$ and $Q' \Leftrightarrow P'(s) \wedge ocpool = Y \cup \{\mathsf{self}\}$. The definition of $U(x, r)$ is reproduced below from Section 2.4.

$$\begin{array}{rl} U(x, r) \ \widehat{=} & (x = \mathsf{null} \wedge r = \varnothing) \vee \\ & (x \neq \mathsf{null} \wedge \mathsf{null} \notin r \wedge x \in r \wedge x.next \in r \wedge x.prev \in r \wedge x \neq x.next \wedge x \neq x.prev).\end{array}$$

Owing to CONSTRUCTOR, verification of $body(ObsColl)$ must take into account updates to self-fields. Therefore the body must be verified with respect to the expanded set of effects $\varepsilon_c^+(s, oc)$ where

$$\varepsilon_c^+(s, oc) \ \widehat{=}\ \varepsilon_c(s, oc), \mathsf{wr}\ \mathsf{self}.(cache, nxto, sub), \mathsf{self}.(next, prev)$$

Furthermore, $body(ObsColl)$ is verified with QS conjoined to Q where, with PS defined as in Section 8.3,

$$QS \ \widehat{=}\ PS \wedge \mathsf{self}.next = \mathsf{null} \wedge \mathsf{self}.prev = \mathsf{null} \wedge \mathsf{self} \notin ocpool$$

Consider a client that constructs a new *ObsColl* as in Figure 17. Let Δ_{so} be the specifications of the public methods *Observer*, *Subject*, *update* and *get*, let Π_{so} be the specifications of the module-scoped methods *register* and *notify* (see Figures 4 and 16).

Let Δ_{oc} be the specification of the public method $ObsColl$ (in Figure 5 and near the beginning of Section 8.4). For brevity in Figure 17, we write, for example, $sigs(\Delta_{oc})$ rather than spelling out the procedure signatures.

The implementation of the $ObsColl$ constructor can be verified, assuming and maintaining I_{oc}, including the obligation to respect the dynamic boundary of module OC. The client can be linked to the inner module OC using rule MISMATCH; that assembly is then linked to the outer module SO using rule GENMISMATCH. We now sketch this verification.

Similar to the generalization of I_{so} in Section 8.3, we generalize the hidden invariant I_{oc} in the following manner for the purpose of framing in the verification of $body(ObsColl)$.

$$I_{oc}(r : \mathsf{rgn}) \;\widehat{=}\; \forall oc : ObsColl \in ocpool \setminus r \cdot oc.next.prev = oc \wedge oc.prev.next = oc$$

Now we observe that $I_{oc} \;\widehat{=}\; I_{oc}(\varnothing)$.

The dynamic boundary for OC is defined as

$$bnd(OC) \;\widehat{=}\; \mathsf{rd}\, ocpool, ocpool`(next, prev)$$

Here, we will need the initial precondition $Init \;\widehat{=}\; sopool = \varnothing \wedge ocpool = \varnothing$. Finally, let

$$Cli \;\widehat{=}\; s := \mathsf{new}\ Subject(); \; x := \mathsf{new}\ ObsColl(s, \mathsf{null})$$

End-to-end Verification of Linked Clients: Illustrating MISMATCH. One may say there are two clients in Figure 17. The outer client is $\mathsf{letrec}\ sigs(\Delta_{oc}) = \cdots\ \mathsf{in}\ Cli$. The inner, nested client is Cli. There are multiple public methods and multiple private methods in SO, each of whose bodies needs to be verified according to rule GENMISMATCH (Section 8.1). In this section, we only show the verification of the body of the $Observer$ constructor in detail, eliding the rest.

To link the outer client with module SO, we use GENMISMATCH as shown in (35)–(37) to follow. In the rule, C is instantiated with the outer client, in context Δ_{so}, as follows:

$$\begin{array}{l} \Delta_{so} \vdash_{Main} \\ \{Init\} \\ \mathsf{letrec}\ sigs(\Delta_{oc}) = \cdots\ \mathsf{in}\ s := \mathsf{new}\ Subject(); x := \mathsf{new}\ ObsColl(s, \mathsf{null}) \\ \{true\} \\ [x, s, \mathsf{alloc}, sopool]\,. \end{array} \tag{35}$$

Whereas rule GENMISMATCH shows a single public method m with body B_m, we actually have four public methods, $Observer$, $Subject$, $update$, and get, and so four proof obligations for their bodies. One of these, for the body of $Observer$, is

$$\Pi_{so}, (\Delta_{so} \between I_{so}) \vdash_{SO} \{\, P(u) \wedge PS \wedge I_{so} \,\}\ body(Observer)\ \{\, P'(u) \wedge I_{so} \,\}\ [\varepsilon_o^+(s)]\,. \tag{36}$$

Note that this, and the premises for the other three bodies, is in context $\Pi_{so}, (\Delta_{so} \between I_{so})$. Rule GENMISMATCH shows a single private method p with body B_p; here we have two, $register$ and $notify$. The proof obligations for their bodies, which we omit, are in context Π_{so}, Δ_{so}. Two other premises in the rule hold because $Main \neq SO$ and the procedure context for the whole program is empty. The remaining premises are

$$\vdash bnd(SO)\ \mathsf{frm}\ I_{so} \quad \text{and} \quad Init \Rightarrow I_{so}. \tag{37}$$

For (37), it is easy to derive the frame judgment; and the implication is valid because I_{so} holds vacuously when $sopool = \varnothing$.

To verify (35), we use MISMATCH as shown in (38)–(40) below. The client C in MISMATCH is instantiated with Cli and is verified in context Δ_{so}, Δ_{oc}, to wit:

$$\begin{array}{l}\Delta_{so}, \Delta_{oc} \vdash_{Main} \\ \{Init\} \\ s := \mathsf{new}\ Subject(); x := \mathsf{new}\ ObsColl(s, \mathsf{null}) \\ \{true\} \\ [x, s, \mathsf{alloc}, sopool]\,.\end{array} \tag{38}$$

Procedure body B in MISMATCH is instantiated with $body(ObsColl)$ and is verified in context $\Delta_{so}, (\Delta_{oc} \between I_{oc})$, to wit:

$$\Delta_{so}, (\Delta_{oc} \between I_{oc}) \vdash_{OC} \{\, Q \wedge QS \wedge I_{oc} \,\}\ body(ObsColl)\ \{\, Q' \wedge I_{oc} \,\}\ [\varepsilon_c^+(s, oc)]\,. \tag{39}$$

The remaining premises of MISMATCH are

$$\vdash bnd(OC)\ \mathsf{frm}\ I_{oc} \quad \text{and} \quad Init \Rightarrow I_{oc} \tag{40}$$

and two others, which hold because $Main \neq OC$ and $SO \not\preceq OC$. It is easy to see that (40) holds.

Of the outstanding obligations (36), (38), and (39), we shall consider first (38) and then (39) before ending with (36).

Verification of Cli: Illustrating CTXINTRO *and* CTXINTROCALL. We will derive the following two judgments.

$$\begin{array}{l}\Delta_{so}\,,\ \Delta_{oc} \vdash_{Main} \\ \{Init\} \\ s := \mathsf{new}\ Subject() \\ \{\forall o \in s.O \cdot Obs(o, s, s.val) \wedge s \neq \mathsf{null} \\ \qquad\qquad \wedge\ s \in sopool \wedge sopool^{\blacklozenge}(obs, O) \subseteq sopool \wedge U(\mathsf{null}, ocpool)\} \\ [s, \mathsf{alloc}, sopool, \mathsf{fr}\,\{s\}]\end{array} \tag{41}$$

$$\begin{array}{l}\Delta_{so}\,,\ \Delta_{oc} \vdash_{Main} \\ \{\forall o \in s.O \cdot Obs(o, s, s.val) \wedge s \neq \mathsf{null} \\ \qquad\qquad \wedge\ s \in sopool \wedge sopool^{\blacklozenge}(obs, O) \subseteq sopool \wedge U(\mathsf{null}, ocpool)\} \\ x := \mathsf{new}\ ObsColl(s, \mathsf{null}) \\ \{true\} \\ [x, \mathsf{alloc}, s.(O, obs), sopool, \mathsf{fr}\,\{x\}]\end{array} \tag{42}$$

From (41) and (42) by SEQ and SUBEFF, we obtain (38).

Derivation of (41). CONSTRUCTOR and CONSEQ yield

$$\begin{array}{l}\Delta_{so} \vdash_{Main} \\ \{sopool^{\blacklozenge}(obs, O) \subseteq sopool\} \\ s := \mathsf{new}\ Subject() \\ \{s.O = \varnothing \wedge s \in sopool \wedge sopool^{\blacklozenge}(obs, O) \subseteq sopool\} \\ [s, \mathsf{alloc}, sopool, \mathsf{fr}\,\{s\}]\end{array}$$

Observe that $Init$ implies this precondition. By FRAME, because $\mathsf{rd}\, ocpool \not\asymp \mathsf{wr}\, s, \mathsf{alloc}, sopool \Leftrightarrow true$, we can conjoin $ocpool = \varnothing$ to get the postcondition $s.O = \varnothing \wedge s \in sopool \wedge sopool^{\blacklozenge}(obs, O) \subseteq sopool \wedge ocpool = \varnothing$, which implies the postcondition in (41).

It remains to add Δ_{oc} to the context to obtain (41). This is where we can use CTXINTRO because the client does not write within the boundary of module OC. We check the rule's premises: $mdl(ObsColl) = OC$ and $bnd(OC)$ ↯ $\mathsf{wr}\, s, \mathsf{alloc}, sopool, \mathsf{fr}\{s\} \Leftrightarrow true$.

Derivation of (42). CONSTRUCTOR and CONSEQ yield

$$
\begin{array}{l}
\Delta_{oc} \vdash_{Main} \\
\{\forall o \in s.O \cdot\ Obs(o, s, s.val) \\
\qquad\qquad \wedge\ s \neq \mathsf{null} \wedge s \in sopool \wedge sopool\text{`}(obs, O) \subseteq sopool \wedge U(\mathsf{null}, ocpool)\} \\
x := \mathsf{new}\ ObsColl(s, \mathsf{null}) \\
\{true\} \\
[x, \mathsf{alloc}, s.(O, obs), sopool, \mathsf{fr}\{x\}]
\end{array}
$$

To obtain (42), it remains to add Δ_{so} to the previous context. Because OC relies on another library, namely SO, we can add it by CTXINTROCALL: we check that the premise $OC \preceq SO$ of the rule holds.

Verification of Body of ObsColl: Illustrating CTXINTROIN. Here we are interested in showing (39). This will follow from applying SEQ to (43) and (44). Note that $\varepsilon_c^+(s, oc) = \varepsilon_o^+(s), \mathsf{wr}\, \mathsf{self}.(next, prev), oc.prev\text{`}next, oc.prev, ocpool$.

$$
\begin{array}{l}
\Delta_{so}\,,\, (\Delta_{oc} \circledcirc I_{oc}) \vdash_{OC} \\
\{Q \wedge QS \wedge I_{oc}\} \\
super(s);\ \mathsf{if}\ oc = \mathsf{null}\ \ldots \\
\{P'(s) \wedge Y = ocpool \wedge \mathsf{self}.next.prev = \mathsf{self} \wedge \mathsf{self}.prev.next = \mathsf{self} \wedge I_{oc}\} \\
[\varepsilon_o^+(s), \mathsf{wr}\, \mathsf{self}.(next, prev), oc.prev\text{`}next, oc.prev]
\end{array}
\tag{43}
$$

$$
\Delta_{so}\,,\, (\Delta_{oc} \circledcirc I_{oc}) \vdash_{OC} \{\ldots\}\ ocpool := ocpool \cup \{\mathsf{self}\}\ \{\ Q' \wedge I_{oc}\ \}\ [ocpool] \tag{44}
$$

Derivation of (44). The postcondition in (39) is $Q' \wedge I_{oc}$. So by BACKWARDSASSIGN

$$
\vdash_{OC} \{\ (Q' \wedge I_{oc})^{ocpool}_{ocpool \cup \{\mathsf{self}\}}\ \}\ ocpool := ocpool \cup \{\mathsf{self}\}\ \{\ Q' \wedge I_{oc}\ \}\ [ocpool]\,. \tag{45}
$$

We are done if we show that the precondition in (44) (indicated by the ellipses) implies $(Q' \wedge I_{oc})^{ocpool}_{ocpool \cup \{\mathsf{self}\}}$ in (45). It remains to add the contexts. We can use CTXINTROIN to add $\Delta_{oc} \circledcirc I_{oc}$: the command denotes updates within module OC and we are adding OC's procedures to the context. We can add Δ_{so} by CTXINTRO since the command is outside this module and so respects its dynamic boundary. Indeed, the precondition implies $bnd(SO)$ ↯ $\mathsf{wr}\, ocpool$ as needed by the rule's premise.

Derivation of (43). This will follow by SEQ on the judgments of $super(s)$ and the conditional. Here, we again highlight the use of CTXINTROIN in the proof of $super(s)$. We will consider $super(s)$ to be a (nonconstructor) call to method $Observer(s)$. So the effects will be $\varepsilon_o^+(s)$ rather than $\varepsilon_o(s)$. By CALL we get $\Delta_{so} \vdash_{OC} \{\ P(s)\ \}\ super(s)\ \{\ P'(s)\ \}[\varepsilon_o^+(s)]$. By CTXINTROIN, we can add context $\Delta_{oc} \circledcirc I_{oc}$ because $super(s)$ is code within OC and we are adding procedures of OC itself. Indeed, $mdl(ObsColl) = OC$ as required by the rule's premise.

Verification of Observer's Body. (36) can be derived by applying SEQ, FRAME and CONSEQ in a routine manner on the three assignments in $body(Observer)$. Here, we provide some highlights of the development there.

Derivation for $\mathsf{self}.sub := u$. The main idea is to start from rule FIELDUPD and obtain

$$\vdash_{SO} \{\, true \,\}\ \mathsf{self}.sub := u\ \{\, \mathsf{self}.sub = u \,\}\ [\mathsf{self}.sub]\,.$$

To add the contexts $\Pi_{so}, (\Delta_{so} \between I_{so})$, we use CTXINTROIN: for each such method m, $mdl(m) = SO$. The rest of the derivation is routine.

Derivation for $u.register(\mathsf{self})$. Recall that Π_{so} contains the specification of $register$. We write this as $\{R\}register(s)\{R'\}[\omega]$ where R, R', ω are as in Figure 4. Let $R0 = R^{\mathsf{self},s}_{u,\mathsf{self}}$, $R0' = R'^{\mathsf{self},s}_{u,\mathsf{self}}$, and $\omega' = \omega^{\mathsf{self},s}_{u,\mathsf{self}}$. Then, CALL yields $\Pi_{so} \vdash_{SO} \{\, R0 \,\}\ u.register(\mathsf{self})\ \{\, R0' \,\}\ [\omega']$. We can add the context $\Delta_{so} \between I_{so}$ by CTXINTROIN. Again, as in the previous case, we are adding methods in SO itself.

Derivation for $sopool := sopool \cup \{\mathsf{self}\}$. We have by BACKWARDSASSIGNMENT and CTXINTROIN

$$\Pi_{so}, (\Delta_{so} \between I_{so}) \vdash_{SO} \{\, (P'(u) \wedge I_{so})^{sopool}_{sopool \cup \{\mathsf{self}\}} \,\}\ sopool := sopool \cup \{\mathsf{self}\}\ \{P'(u) \wedge I_{so}\}[sopool].$$

The rest of the development proceeds similarly to that of the assignment to $ocpool$ in the verification of $body(ObsColl)$.

9. CALLBACKS BETWEEN MODULES

In Section 8, we saw how rule MISMATCH can be used to link one module in the context of another module on which it relies. The rules SOF and LINK, from which MISMATCH is derived, are not restricted to hierarchically nested modules. They support modular reasoning about separate but interdependent modules as we show in this section.

This rule links a client with two interdependent modules M and N, each with its own hidden invariant and dynamic boundary. The client command B is in module L. We consider the special case of a single procedure in each of M and N. In particular, suppose that

$$\Theta \text{ is } \{Q\}m(x:T)\{Q'\}[\varepsilon_m] \quad \text{and} \quad \Delta \text{ is } \{R\}n(z:U)\{R'\}[\varepsilon_n]\,,$$

where $mdl(m) = M$ and $mdl(n) = N$. The rule allows the implementation, C, of m to call n, and also the implementation, D, of n to call m. The rule is

$$\text{MISMATCHX}\ \frac{\begin{array}{c}\Xi, \Delta, \Theta \vdash^{\Gamma}_{L} \{\, P \,\}\ B\ \{\, P' \,\}\ [\varepsilon] \\ \Xi, (\Delta \between I), (\Theta \between I) \vdash^{\Gamma, x:T}_{M} \{\, Q \wedge I \,\}\ C\ \{\, Q' \wedge I \,\}\ [\varepsilon_m] \\ \Xi, (\Delta \between J), (\Theta \between J) \vdash^{\Gamma, z:U}_{N} \{\, R \wedge J \,\}\ D\ \{\, R' \wedge J \,\}\ [\varepsilon_n] \\ \vdash bnd(M)\ \mathsf{frm}\ I \qquad \vdash bnd(N)\ \mathsf{frm}\ J \qquad P \Rightarrow I \wedge J \\ mdl(p) \not\preceq M \text{ and } mdl(p) \not\preceq N \text{ for all } p \text{ in } \Xi \qquad L, M, N \text{ are distinct}\end{array}}{\Xi \vdash^{\Gamma}_{L} \{\, P \,\}\ \mathsf{letrec}\ m(x:T) = C;\ n(z:U) = D\ \mathsf{in}\ B\ \{\, P' \,\}\ [\varepsilon]}.$$

Here, the client B is verified using procedures Δ, Θ, and those of Ξ. The implementation C of m relies on invariant I and the implementation D of n relies on invariant J; both implementations may use ambient library Ξ. The idea is that the conditions $\vdash bnd(M)\ \mathsf{frm}\ I$, $P \Rightarrow I$, and correctness of C are checked in the context of module M; the conditions $\vdash bnd(N)\ \mathsf{frm}\ J$, $P \Rightarrow J$, and correctness of D are checked in the context of N. More to the point: P will conjoin some client-specific conditions with $Init0$ and $Init1$, with $Init0 \Rightarrow I$ and $Init1 \Rightarrow J$ proved in the appropriate contexts.

Figure 11 in Section 6 is a toy example that illustrates intermodule callbacks in the context of an ambient module. We declare the import relation to be $OD \preceq EV$,

$EV \preceq OD$, $OD \preceq Lib$, $EV \preceq Lib$, and $Main \preceq EV$ (closed reflexively and transitively). The main module is for the following judgment:

$$\Xi_q \vdash_{Main} \{\, x \neq y \,\}\ \mathsf{letrec}\ \ldots = \ldots\ \mathsf{in}\ even(2)\ \{\, true \,\}\ [x.val, y.val, x.f, y.f]\,,$$

where Ξ_q is the specification for Lib, a single procedure $\{true\}q(c:Ctr)\{true\}[c.f]$. It can be derived using rule MISMATCHX instantiated with $L, M, N, \Xi := Main, EV, OD, \Xi_q$.

Derivation. Rule MISMATCHX can be derived as follows. For C, we instantiate SOF, using side conditions $\vdash bnd(N)$ frm J, $M \neq N$, and $\forall p \in \Xi \cdot mdl(p) \not\preceq N$, as follows:

$$\frac{\Xi, (\Delta \mathbin{\textcircled{\scriptsize /}} I), (\Theta \mathbin{\textcircled{\scriptsize /}} I) \vdash_M^{\Gamma, x:T} \{\, Q \wedge I \,\}\ C\ \{\, Q' \wedge I \,\}\ [\varepsilon_m]}{\Xi\,, (\Delta \mathbin{\textcircled{\scriptsize /}} I \mathbin{\textcircled{\scriptsize /}} J)\,, (\Theta \mathbin{\textcircled{\scriptsize /}} I \mathbin{\textcircled{\scriptsize /}} J) \vdash_M^{\Gamma, x:T} \{\, Q \wedge I \wedge J \,\}\ C\ \{\, Q' \wedge I \wedge J \,\}\ [\varepsilon_m]}.$$

For D, we use side conditions $\vdash bnd(M)$ frm I, $N \neq M$, and $\forall p \in \Xi \cdot mdl(p) \not\preceq M$, instantiating SOF as follows:

$$\frac{\Xi, (\Delta \mathbin{\textcircled{\scriptsize /}} J), (\Theta \mathbin{\textcircled{\scriptsize /}} J) \vdash_N^{\Gamma, z:U} \{\, R \wedge J \,\}\ D\ \{\, R' \wedge J \,\}\ [\varepsilon_n]}{\Xi, (\Delta \mathbin{\textcircled{\scriptsize /}} J \mathbin{\textcircled{\scriptsize /}} I), (\Theta \mathbin{\textcircled{\scriptsize /}} J \mathbin{\textcircled{\scriptsize /}} I) \vdash_N^{\Gamma, z:U} \{\, R \wedge J \wedge I \,\}\ D\ \{\, R' \wedge J \wedge I \,\}\ [\varepsilon_n]}.$$

Using commutativity of $\wedge$, specifically $J \wedge I \equiv I \wedge J$, and the definition of $\mathbin{\textcircled{\scriptsize /}}$, we can normalize to get $\Delta \mathbin{\textcircled{\scriptsize /}} I \wedge J$ and $\Theta \mathbin{\textcircled{\scriptsize /}} I \wedge J$ in both conclusions. (We return to this later.)

For D, we also use CONSEQ to commute J and I in the conclusion, so we reach

$$\Xi, (\Delta \mathbin{\textcircled{\scriptsize /}} I \mathbin{\textcircled{\scriptsize /}} J), (\Theta \mathbin{\textcircled{\scriptsize /}} I \mathbin{\textcircled{\scriptsize /}} J) \vdash_N^{\Gamma, z:U} \{\, R \wedge I \wedge J \,\}\ D\ \{\, R' \wedge I \wedge J \,\}\ [\varepsilon_n]\,.$$

For the client B, from the premise $\Xi, \Delta, \Theta \vdash_L \{\, P \,\}\ B\ \{\, P' \,\}\ [\varepsilon]$ and side conditions we can use SOF to get

$$\Xi, (\Delta \mathbin{\textcircled{\scriptsize /}} I), (\Theta \mathbin{\textcircled{\scriptsize /}} I) \vdash_L \{\, P \wedge I \,\}\ B\ \{\, P' \wedge I \,\}\ [\varepsilon]$$

and again use SOF to get

$$\Xi, (\Delta \mathbin{\textcircled{\scriptsize /}} I \mathbin{\textcircled{\scriptsize /}} J), (\Theta \mathbin{\textcircled{\scriptsize /}} I \mathbin{\textcircled{\scriptsize /}} J) \vdash_L \{\, P \wedge I \wedge J \,\}\ B\ \{\, P' \wedge I \wedge J \,\}\ [\varepsilon]\,.$$

By definition of $\mathbin{\textcircled{\scriptsize /}}$ we have $\Delta \mathbin{\textcircled{\scriptsize /}} I \mathbin{\textcircled{\scriptsize /}} J, \Theta \mathbin{\textcircled{\scriptsize /}} I \mathbin{\textcircled{\scriptsize /}} J = (\Delta, \Theta) \mathbin{\textcircled{\scriptsize /}} I \wedge J$, so we can write the preceding judgments for B, C, and D as follows:

$$\begin{aligned}
&\Xi, ((\Delta, \Theta) \mathbin{\textcircled{\scriptsize /}} I \wedge J) \vdash_L^{\Gamma} \{\, P \wedge I \wedge J \,\}\ B\ \{\, P' \wedge I \wedge J \,\}\ [\varepsilon]\\
&\Xi, ((\Delta, \Theta) \mathbin{\textcircled{\scriptsize /}} I \wedge J) \vdash_M^{\Gamma, x:T} \{\, Q \wedge I \wedge J \,\}\ C\ \{\, Q' \wedge I \wedge J \,\}\ [\varepsilon_m]\\
&\Xi, ((\Delta, \Theta) \mathbin{\textcircled{\scriptsize /}} I \wedge J) \vdash_N^{\Gamma, z:U} \{\, R \wedge I \wedge J \,\}\ D\ \{\, R' \wedge I \wedge J \,\}\ [\varepsilon_n]\,.
\end{aligned}$$

To these three judgments we apply rule LINK. It requires $M \notin \Xi$ and $N \notin \Xi$, both of which follow from $\forall p \in \Xi \cdot mdl(p) \not\preceq M$. We get

$$\Xi \vdash \{\, P \wedge I \wedge J \,\}\ \mathsf{letrec}\ m(x:T) = C;\ n(z:U) = D\ \mathsf{in}\ B\ \{\, P' \wedge I \wedge J \,\}\ [\varepsilon]\,.$$

Finally, to reach the conclusion of rule MISMATCHX we use its side condition $P \Rightarrow I \wedge J$ and the rule of consequence.

A Fine Point. In the derivation, we replace specification $\Delta \mathbin{\textcircled{\scriptsize /}} J \wedge I$ by $\Delta \mathbin{\textcircled{\scriptsize /}} I \wedge J$, which by definition of the syntactic operation $\mathbin{\textcircled{\scriptsize /}}$ simply means replacing $\{R \wedge J \wedge I\}n(z:U)\{R' \wedge J \wedge I\}[\varepsilon_n]$ by $\{R \wedge I \wedge J\}n(z:U)\{R' \wedge I \wedge J\}[\varepsilon_n]$. Strictly speaking, to derive MISMATCHX we would need a proof rule that allows a procedure specification to be replaced by one with logically equivalent pre- and postconditions.

More generally, it is sound to strengthen a specification in a hypothetical judgment. To formulate a general rule, one could introduce a refinement relation $\sqsupseteq$ on specifications, so that $\{Q\}m(x:T)\{Q'\}[\varepsilon] \sqsupseteq \{Q_1\}m(x:T)\{Q_1'\}[\varepsilon_1]$ would imply that any implementation that satisfies $\{Q\}m(x:T)\{Q'\}[\varepsilon_1]$ also satisfies $\{Q_1\}m(x:T)\{Q_1'\}[\varepsilon]$. Then, lift $\sqsupseteq$ so that $\Delta \sqsupseteq \Delta_1$ means that Δ and Δ_1 specify the same procedures and for each procedure, the specification in Δ refines the one in Δ_1. The proof rule is then

$$\frac{\Delta_1 \vdash \{\,P\,\}\ B\ \{\,P'\,\}\ [\varepsilon] \qquad \Delta \sqsupseteq \Delta_1}{\Delta \vdash \{\,P\,\}\ B\ \{\,P'\,\}\ [\varepsilon]}.$$

Reasoning about refinement of specifications on the right side of $\vdash$ is the purpose of structural rules such as SUBST and FRAME (and others like CONSEQ, SUBEFF, and EXIST in Part I). The paper by Hoare [1971] on procedures introduced a rule of "adaptation" that is an alternative way of supporting such reasoning. As emphasized by Naumann [2001], adaptation rules effectively approximate refinement of specifications by validities involving the pre- and postconditions. For derivation of MISMATCHX, it would suffice for $\sqsupseteq$ to be approximated as $(Q_1 \Rightarrow Q) \wedge (Q' \Rightarrow Q_1') \wedge \varepsilon = \varepsilon_1$, but for completeness one could use conditions that better approximate refinement.

10. RELATED WORK

It is notoriously difficult to achieve encapsulation in the presence of shared, dynamically allocated mutable objects [Leavens et al. 2007; O'Hearn et al. 2009]. Most current tools for automated software verification either do not support the hiding of invariants (e.g., Jahob [Zee et al. 2008], jStar [Distefano and Parkinson 2008], Krakatoa [Filliâtre and Marché 2007], VCC [Cohen et al. 2009]), do not treat object invariants soundly (e.g., ESC/Java [Flanagan et al. 2002]) or at best offer soundness for restricted situations where a hierarchical structure can be imposed on the heap (e.g., Spec# [Barnett et al. 2005]). Some of these tools do achieve significant automation, especially by using SMT solvers [Kroening and Strichman 2008]. The KeY tool [Beckert et al. 2007] supports hiding as in JML and provides support for sound reasoning about class invariants; more on that below.

Hiding is easy to encode in an axiomatic semantics—it is just Hoare's mismatch, phrased in terms of assert and assume statements. Some verifiers provide hiding, by enforcing specific encapsulation disciplines through some combination of type checking and extra verification conditions. For example, the Boogie methodology [Leino and Müller 2004] used by Spec# stipulates intermediate assertions (in all code) that guarantees an all-states ownership invariant. Another version of Spec# [Smans et al. 2010] 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.

Methodologies. A number of methodologies have been proposed for ownership-based hiding of invariants (e.g., Müller et al. [2006]). Drossopoulou et al. [2008] 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, for example, *visible state semantics* requires all invariants to hold on all public method call/return boundaries; other proposals require invariants to hold more often [Leino and Müller 2004].

Ownership type systems are a way to enforce hierarchical ownership with the benefit of uniformity and a fixed semantics of when invariants hold. Recent work on ownership

has addressed the need for clusters without a single dominating owner. Cameron et al. [2007] give a helpful survey of ownership systems. They adapt ownership types to a system of "boxes" (clusters) that describes rather than restricts program structure. Thus, it does not ensure encapsulation, but they provide and prove sound an effect system for disjointness of boxes. Müller and Rudich [2007] extend Universe Types, which provides encapsulation and has been adopted by JML for invariants, to solve the difficult problem of ownership transfer.

The KeY tool [Beckert et al. 2007] supports variants of visible state semantics for invariants, complemented by special accessibility and reachability predicates to specify encapsulation properties that subsume ownership [Roth 2005, 2006]. A recent version of KeY [Schmitt et al. 2010] provides dynamic frames for procedures, much in the form we use here (see Part I for discussion); the connection with invariants has evidently not been addressed to date.

The difficulty of generalizing ownership to fit important design patterns led Parkinson and Bierman [2005] to pursue abstraction instead of hiding, via second order assertions in separation logic; this has been implemented in jStar [Distefano and Parkinson 2008]. The main idea is to permit client reasoning using "abstract predicates"—predicates whose concrete definitions are not visible to the client. Parkinson and Bierman [2008] consider abstract predicates in a language with inheritance and dynamic dispatch. Parkinson [2007] articulates the case for specifications at the level of object clusters and shows an example specification of the Observer pattern that uses abstract predicates. In this article, we have alluded to a limited form of abstraction with data groups (cf. Remark 6.2).

Often abstract predicates are used when a client needs to track phase changes of internal data structures [Thamsborg et al. 2012]. Consider functions *grant* and *revoke* that control granting of read access to some sensitive file in a database module. For reasoning purposes, a client needs to know whether a grant is successful. So the function *grant* can enable reading by adding an abstract predicate to the knowledge exposed at the interface about the module's internal state: this abstract predicate would appear in *grant*'s postcondition. Similarly *revoke* will omit the abstract predicate in its postcondition but will require it in its precondition: so *revoke* can only be called in states where the abstract predicate is in force, that is, after a corresponding *grant* is invoked.

For reasoning about functional correctness, the internal state of a module can be factored into two components: a fully encapsulated part and a client visible part. The former can be hidden and absent at the module interface, for clients do not need to reason about fully encapsulated state. This yields succinct specifications. To reason about properties of the latter one can expose information at the interface in several ways, including abstract predicates, model fields [Leino and Müller 2006], typestate [Bierhoff and Aldrich 2007], and in many case direct reference to concrete state assessible to client code (e.g., the *hasNext* method of an iterator). To cope with inheritance and dynamic dispatch, abstract predicates and model fields require some care with scoping issues; for the others, one can use the standard notion of behavioral subtyping [Leavens and Naumann 2013; Liskov and Wing 1994].

The Boogie methodology is generalized and extended to concurrency by Locally Checked Invariants (LCI) [Cohen et al. 2010] which is implemented in the VCC tool [Cohen et al. 2009]. To complement ownership, non-hierarchical dependency is tracked in ghost state (called "claims", generalizing the friendship discipline [Naumann and Barnett 2006]). In LCI, invariants are associated with types, which provide a form of abstraction. Abstract predicates also have been explored for modular reasoning about concurrent programs, in which context they are called concurrent abstract predicates [Dinsdale-Young et al. 2010]. It would be useful to explore invariant hiding in the

concurrency setting. LCI gives some indication of how this might be possible using first-order techniques along the lines we are exploring in this article.

Separation Logics. Separation logic (SL) is a major influence on our work. Our SOF rule is adapted from [O'Hearn et al. 2009], as is the example in Section 2.2. The SOF rule of SL relies on two critical features: the separating conjunction and the *tight interpretation* of a correctness judgment $\{P\}C\{Q\}$, which requires that C neither reads nor writes outside the footprint of P. These features yield great economy of expression (though it does not remove the annoyance of ordinary frame conditions for variables). But conflating read with write has consequences. To get shared reads, the semantics of separating conjunction can embody some notion of permissions [Bornat et al. 2005; Dockins et al. 2009], 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, so they can be abstracted but not hidden.

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 that is separated from the write effect of a command. In SL, the corresponding 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 ordinary rule of conjunction (cf. CONJ in Figure 15) 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 [O'Hearn et al. 2009].[23] The semantic analysis in O'Hearn et al. [2009] 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 such as the one of Birkedal et al. [2005]. They give higher order framing rules for call-by-name procedures in idealized Algol. Such a semantics was eschewed by O'Hearn et al. [2009] because the conjunction rule is patently sound in ordinary readings of Hoare triples. The later journal paper of Birkedal et al. [2006] allows the rule of conjunction when invariants are restricted to precise predicates. Dropping the conjunction rule facilitates modeling of higher order framing rules that capture something like visible state semantics for invariants even in programs using code pointers (e.g., Schwinghammer et al. [2010]). The metatheory underlying the Ynot tool for interactive verification [Malecha et al. 2010] uses a model that does not validate the conjunction rule [Petersen et al. 2008] while the rule is valid in Hoare type theory (HTT) without restriction to precise predicates [Nanevski et al. 2010]. HTT does not have a rule akin to SOF, nor higher order frame rules, but permits data abstraction by way of abstract predicates. Loss of the conjunction rule also occurs in the work of Amtoft et al. [2006] who were interested in reasoning about information flow.

Higher order separation logics and type theories offer elegant means to achieve data abstraction and strong functional specifications of interesting design patterns

[23] In separation logic, predicate I is *precise* iff $(I * \_)$ distributes over $\wedge$. In this article, our example invariants are all precise in the sense of having unique minimal footprints, but not all useful ones are, for example, "there exists a nonfull queue".

[Krishnaswami et al. 2009, 2010; Malecha et al. 2010] and algorithms (e.g., fast congruence closure in Nanevski et al. [2010]). 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.

Whereas the MISMATCH and SOF rules connect a module implementation to its client while hiding the module's invariant, the "anti-frame" rule of Pottier hides the invariant of a module, independent of the client. The idea is attractive, and potentially important for higher order programs, but it poses difficult semantic challenges [Schwinghammer et al. 2010].

Other Hoare Logics for Object-Oriented Programs. Some early works on reasoning about object-oriented programs are by von Oheimb and Nipkow [2002], Poetzsch-Heffter and Müller [1999], Pierik and de Boer [2005b]. As Parkinson and Bierman [2005] remark, "these logics do not have the framing properties of separation logic . . . Also they do not attempt to express abstraction." Nor do they address hiding. These works do address subtyping and inheritance, as do many subsequent works including Müller [2002], Parkinson and Bierman [2005, 2008], and Chin et al. [2008]. Region logic was developed with object-oriented programs in mind and the formalization is easily adapted to subtyping. Extant techniques for inheritance appear to be compatible but are beyond the scope of this article.

Lahiri et al. [2011] use linear types and explicit partial heaps, dubbed "linear maps", in a classical assertion language for local reasoning in object-based programs. The approach resembles aspects of region logic and aspects of separation logic, but is quite different in detail, including several novel program constructs and judgments. The paper formalizes and proves sound a logic. Hiding of module invariants is provided, in a way similar to Pottier's anti-frame rule. The development to date does not encompass callbacks between modules, and stored linear maps are needed to cope with nested data structures. Because the assertion language is first order logic with maps, verification conditions should be amenable to automated checking using SMT solvers.

The textbook of Apt et al. [2009] provides an elegant treatment of Hoare logics for object-based programs, encompassing both partial and total correctness. Specifications do not include frame conditions. This loss of modularity is mitigated by another limitation: every procedure specification is associated with the declaration of the procedure implementation, so one may check informally that the procedure does no unintended updates. But there is no explicit linking rule. Soundness of the proof rules, including method call and dynamic allocation, is proved beautifully by transforming programs using ordinary (mutually recursive) procedures. Completeness of the logic (omitting dynamic allocation) is also proved via this transformation [Apt et al. 2012].

For completeness, the logic(s) of Apt et al. [2009] have structural rules like exists-introduction, conjunction, and substitution.[24] Specifications take the form originated by Hoare, in which specification-only variables are implicitly quantified (and scoped) over pre-post specifications. It is not easy to formulate and prove a sound substitution rule and mistakes have appeared in the literature. For this reason, many tools and some logics (e.g., Nanevski et al. [2008]) eschew the use of specification-only variables in favor of old-expressions in postconditions. O'Hearn et al. [2009] explicitly omit them, for the sake of simplifying their soundness proof and analysis of second order framing.

Soundness of our rule SUBST in Figure 15 hinges on the correct semantics of context procedure calls (Figure 9). The naïve semantics described by the transition rule (10) (in Section 4) is erroneous in its treatment of specification-only variables; it fails to have the properties in Lemma 4.10, which are needed to prove soundness of rule SUBST,

[24]Our rules are similar, except that their INVARIANCE rule is generalized by our FRAME rule.

and indeed the rule is unsound for that semantics. Remarkably, the semantics using (10) does validate all of our other proof rules.[25]

The reasoning embodied collectively by the structural rules mentioned above can alternatively be provided by a rule of Adaptation [Hoare 1971; Olderog 1983]. Pierik and de Boer [2004] provide an adaptation rule for object-oriented programs. It would be interesting to investigate an adaptation rule for region logic.

One way to cope with substitution and specification-only variables is to embed Hoare logic in an ambient logic, treating them as meta-variables. An influential example of this approach is the Specification Logic of Reynolds [1982]. Nipkow [2002] takes that approach using an interactive theorem prover and an adaptation rule based on that of Kleymann [1999].

11. CONCLUSION

In this article, we explore a novel interface specification feature: the *dynamic boundary* that 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 that is hidden from clients. 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.

For practical purposes, dynamic boundaries should only be used where scoping and parameterization mechanisms are inadequate, that is, for dynamically allocated objects. For expository purposes, we formalized a rudimentary notion of module as group of procedures; in examples we use dynamic boundaries even for static variables, to avoid the need for a full fledged module system.

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 [Berdine et al. 2007; Leavens et al. 2007; Malecha et al. 2010]. 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 [Dietl and Müller 2005; Grothoff et al. 2007] as well as less restrictive disciplines that impose proof obligations on clients, e.g., ownership transfers that are "in the eye of the asserter" [O'Hearn et al. 2009].

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, for example, 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 Section 2.2).

[25]The proofs for SOF and CALL are slightly simpler using (10). Notably, the proofs of LINK and its supporting recursion Lemma 7.2 are almost the same, but slightly simpler and no longer needing Lemma 4.10.

Because the dynamic boundary of a module encapsulates state on which depends an invariant hidden within that module, the boundaries of distinct modules must be disjoint in a given state. In such situations, it is possible to hide information using suitable forms of rely-guarantee reasoning [Jones 1983], as explored for this purpose by Naumann and Barnett [2006], Leino and Schulte [2007], and Cohen et al. [2010]. It is an interesting question as to how such techniques relate to the approach to hiding in this article.

Finally it remains to be seen how the approach explored here can be extended to address the last of the needs listed in Section 1, namely more advanced programming features such as inheritance, concurrency [Filipovic et al. 2010], code pointers, and parametric polymorphism.

ACKNOWLEDGMENTS

Many people helped with suggestions and encouragement, including Lennart Beringer, Lars Birkedal, Sophia Drossopoulou, Bart Jacobs, Neel Krishnaswami, Gary Leavens, Peter Müller, Aleksandar Nanevski, Peter O'Hearn, Matthew Parkinson, Stan Rosenberg, Jan Smans, Jacob Thamsborg, Hongseok Yang, and several anonymous reviewers.

REFERENCES

Amtoft, T., Bandhakavi, S., and Banerjee, A. 2006. A logic for information flow in object-oriented programs. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. 91–102.

Apt, K. R., de Boer, F. S., and Olderog, E.-R. 2009. *Verification of Sequential and Concurrent Programs* 3rd Ed. Springer.

Apt, K. R., de Boer, F. S., Olderog, E.-R., and de Gouw, S. 2012. Verification of object-oriented programs: A transformational approach. *J. Comput. Syst. Sci. 78,* 3, 823–852.

Back, R.-J. and von Wright, J. 1998. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science, Springer-Verlag.

Banerjee, A., Naumann, D. A., and Rosenberg, S. 2013. Local reasoning for global invariants, Part I: Region logic. *J. ACM*, To appear.

Barnett, M., Leino, K. R. M., and Schulte, W. 2005. The Spec# programming system: An overview. In *Proceedings of the International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*. Revised Selected Papers. Lecture Notes in Computer Science, vol. 3362, 49–69.

Beckert, B., Hähnle, R., and Schmitt, P. H. 2007. *Verification of Object-Oriented Software: The KeY Approach*. Lecture Notes in Computer Science, vol. 4334, Springer-Verlag.

Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P. W., Wies, T., and Yang, H. 2007. Shape analysis for composite data structures. In *Proceedings of the Computer Aided Verification*. Lecture Notes in Computer Science, vol. 4590, 178–192.

Bierhoff, K. and Aldrich, J. 2007. Modular typestate checking of aliased objects. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications*. 301–320.

Birkedal, L., Torp-Smith, N., and Yang, H. 2005. Semantics of separation-logic typing and higher-order frame rules. In *Proceedings of the IEEE Symposium on Logic in Computer Science*. 260–269.

Birkedal, L., Torp-Smith, N., and Yang, H. 2006. Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. *Log. Meth. Comput. Sci. 2,* 5:1, 1–33.

Bornat, R., Calcagno, C., O'Hearn, P. W., and Parkinson, M. J. 2005. Permission accounting in separation logic. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. 259–270.

Cameron, N. R., Drossopoulou, S., Noble, J., and Smith, M. J. 2007. Multiple ownership. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications*. 441–460.

Chin, W.-N., David, C., Nguyen, H. H., and Qin, S. 2008. Enhancing modular OO verification with separation logic. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. 87–99.

Cohen, E., Dahlweid, M., Hillebrand, M. A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., and Tobies, S. 2009. VCC: A practical system for verifying concurrent C. In *Proceedings of the Theorem Proving in Higher Order Logics*. Lecture Notes in Computer Science, vol. 5674, 23–42.

Cohen, E., Moskal, M., Schulte, W., and Tobies, S. 2010. Local verification of global invariants in concurrent programs. In *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 6174, 480–494.

Dietl, W. and Müller, P. 2005. Universes: Lightweight ownership for JML. *J. Obj. Tech. 4*, 5–32.

Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M. J., and Vafeiadis, V. 2010. Concurrent abstract predicates. In *Proceedings of the European Conference on Object-Oriented Programming*. 504–528.

Distefano, D. and Parkinson, M. J. 2008. jStar: Towards practical verification for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications*. 213–226.

Dockins, R., Hobor, A., and Appel, A. W. 2009. A fresh look at separation algebras and share accounting. In *Proceedings of the Asian Symposium on Programming Languages and Systems*. Lecture Notes in Computer Science, vol. 5904, 161–177.

Drossopoulou, S., Francalanza, A., Müller, P., and Summers, A. J. 2008. A unified framework for verification techniques for object invariants. In *Proceedings of the European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science, vol. 5142, 412–437.

Filipovic, I., O'Hearn, P. W., Rinetzky, N., and Yang, H. 2010. Abstraction for concurrent objects. *Theoret. Comput. Sci. 411,* 51–52, 4379–4398.

Filliâtre, J.-C. and Marché, C. 2007. The Why/Krakatoa/Caduceus platform for deductive program verification (tool paper). In *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 4590, 173–177.

Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. 2002. Extended static checking for Java. In *Proceedings of the ACM Conference on Programming Languages Design and Implementation*. 234–245.

Grothoff, C., Palsberg, J., and Vitek, J. 2007. Encapsulating objects with confined types. *ACM Trans. Program. Lang. Syst. 29,* 6.

Harel, D., Pnueli, A., and Stavi, J. 1977. A complete axiomatic system for proving deductions about recursive programs. In *Proceedings of the Annual ACM Symposium on Theory of Computing*. 249–260.

He, J., Hoare, C. A. R., and Sanders, J. 1986. Data refinement refined (resumé). In *Proceedings of the European Symposium on Programming*. Lecture Notes in Computer Science, vol. 213, Springer, 187–196.

Hoare, C. A. R. 1971. Procedures and parameters: An axiomatic approach. In *Proceedings of the Symposium on Semantics of Algorithmic Languages*. E. Engeler Ed., Springer, 102–116.

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

Jones, C. B. 1983. Specification and design of (parallel) programs. In *Proceedings of the IFIP Congress*. 321–332.

Kassios, I. T. 2006. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *Formal Methods*, Lecture Notes in Computer Science, vol. 4085, 268–283.

Kassios, I. T. 2011. The dynamic frames theory. *Form. Asp. Comput. 23,* 3, 267–288.

Kleymann, T. 1999. Hoare logic and auxiliary variables. *Form. Asp. Comput. 11*, 541–566.

Krishnaswami, N. R., Aldrich, J., Birkedal, L., Svendsen, K., and Buisse, A. 2009. Design patterns in separation logic. In *Proceedings of the ACM Workshop on Types in Languages Design and Implementation*. 105–116.

Krishnaswami, N. R., Aldrich, J., and Birkedal, L. 2010. Verifying event-driven programs using ramified frame properties. In *Proceedings of the ACM Workshop on Types in Languages Design and Implementation*. 63–76.

Kroening, D. and Strichman, O. 2008. *Decision Procedures: An Algorithmic Point of View*. Springer.

Lahiri, S. K., Qadeer, S., and Walker, D. 2011. Linear maps. In *Proceedings of the ACM Workshop on Programming Languages meets Program Verification*. 3–14.

Leavens, G. T. and Müller, P. 2007. Information hiding and visibility in interface specifications. In *Proceedings of the International Conference on Software Engineering*. 385–395.

Leavens, G. T. and Naumann, D. A. 2013. Behavioral subtyping, specification inheritance, and modular reasoning. Tech. rep. CS-TR-13-03, Department of Computer Science, University of Central Florida.

Leavens, G. T., Cheon, Y., Clifton, C., Ruby, C., and Cok, D. R. 2003. How the design of JML accommodates both runtime assertion checking and formal verification. In *Proceedings of the Formal Methods for Components and Objects (FMCO'02)*. Lecture Notes in Computer Science, vol. 2852, Springer, 262–284.

Leavens, G. T., Leino, K. R. M., and Müller, P. 2007. Specification and verification challenges for sequential object-oriented programs. *Form. Asp. Comput. 19,* 2, 159–189.

Leino, K. R. M. 2008. This is Boogie 2. Manuscript KRML 178. http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf.

Leino, K. R. M. and Müller, P. 2004. Object invariants in dynamic contexts. In *Proceedings of the European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science, vol. 3086, 491–516.

Leino, K. R. M. and Müller, P. 2006. A verification methodology for model fields. In *Proceedings of the European Symposium on Programming Languages and Systems*. Lecture Science on Computer Science, vol. 3924, 115–130.

Leino, K. R. M. and Schulte, W. 2007. Using history invariants to verify observers. In *Proceedings of the European Symposium on Programming Languages and Systems*. Lecture Notes in Computer Science, vol. 4421, 80–94.

Leino, K. R. M., Poetzsch-Heffter, A., and Zhou, Y. 2002. Using data groups to specify and check side effects. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 246–257.

Liskov, B. H. and Wing, J. M. 1994. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst. 16,* 6, 254–280.

Malecha, G., Morrisett, G., Shinnar, A., and Wisnesky, R. 2010. Toward a verified relational database management system. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. 237–248.

Morgan, C. 1994. *Programming from Specifications* 2nd Ed. Prentice Hall.

Müller, P. 2002. *Modular Specification and Verification of Object-Oriented Programs*. Lecture Notes in Computer Science, vol. 2262, Springer.

Müller, P. and Rudich, A. 2007. Ownership transfer in universe types. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications*. 461–478.

Müller, P., Poetzsch-Heffter, A., and Leavens, G. T. 2006. Modular invariants for layered object structures. *Sci. Comput. Program. 62,* 3, 253–286.

Nanevski, A., Morrisett, G., and Birkedal, L. 2006. Polymorphism and separation in Hoare type theory. In *Proceedings of the International Conference on Functional Programming*. 62–73.

Nanevski, A., Morrisett, J. G., and Birkedal, L. 2008. Hoare type theory, polymorphism and separation. *J. Funct. Prog. 18,* 5–6, 865–911.

Nanevski, A., Vafeiadis, V., and Berdine, J. 2010. Structuring the verification of heap-manipulating programs. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. 261–274.

Naumann, D. A. 2001. Calculating sharp adaptation rules. *Inf. Process. Lett. 77*, 201–208.

Naumann, D. A. and Banerjee, A. 2010. Dynamic boundaries: Information hiding by second order framing with first order assertions. In *Proceedings of the European Symposium on Programming Languages and Systems*. Lecture Notes in Computer Science, vol. 6012, 2–22.

Naumann, D. A. and Barnett, M. 2004. Towards imperative modules: Reasoning about invariants and sharing of mutable state. In *Proceedings of the IEEE Symposium on Logic in Computer Science*. 313–323.

Naumann, D. A. and Barnett, M. 2006. Towards imperative modules: Reasoning about invariants and sharing of mutable state. *Theoret. Comput. Sci. 365*, 143–168.

Nipkow, T. 2002. Hoare logics for recursive procedures and unbounded nondeterminism. In *Proceedings of the Conference on Computer Science Logic*. Lecture Notes in Computer Science, vol. 2471, 103–119.

O'Hearn, P. W. and Tennent, R. D., Eds. 1997. *ALGOL-like Languages*. vol. 1 and vol. 2, Birkhäuser, Boston, Massachusetts.

O'Hearn, P. W., Reynolds, J. C., and Yang, H. 2001. Local reasoning about programs that alter data structures. In *Proceedings of the Conference on Computer Science Logic*. Lecture Notes in Computer Science, vol. 2142, 1–19.

O'Hearn, P. W., Yang, H., and Reynolds, J. C. 2004. Separation and information hiding. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. 268–280.

O'Hearn, P. W., Yang, H., and Reynolds, J. C. 2009. Separation and information hiding. *ACM Trans. Program. Lang. Syst. 31,* 3, 1–50.

Olderog, E.-R. 1983. On the notion of expressiveness and the rule of adaptation. *Theoret. Comput. Sci. 24*, 337–347.

Owicki, S. and Gries, D. 1976. An axiomatic proof technique for parallel programs I. *Acta Inf. 6*, 319–340.

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

Parkinson, M. J. and Bierman, G. M. 2005. Separation logic and abstraction. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. 247–258.

Parkinson, M. J. and Bierman, G. M. 2008. Separation logic, abstraction and inheritance. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. 75–86.

Petersen, R. L., Birkedal, L., Nanevski, A., and Morrisett, G. 2008. A realizability model for impredicative Hoare type theory. In *Proceedings of the European Symposium on Programming Languages and Systems*. Lecture Notes in Computer Science, vol. 4960, 337–352.

Pierik, C. 2006. Validation techniques for object-oriented proof outlines. Tech. rep. 2006-5, Universiteit Utrecht, SIKS Dissertation Series. ISBN 90-393-4217-2.

Pierik, C. and de Boer, F. S. 2004. Modularity and the rule of adaptation. In *Algebraic Methodology and Software Technology*. Lecture Notes in Computer Science, vol. 3116, 394–408.

Pierik, C. and de Boer, F. 2005a. On behavioral subtyping and completeness. In *Proceedings of the 7th ECOOP Workshop on Formal Techniques for Java-like Programs*. J. Vitek and F. Logozzo Eds.

Pierik, C. and de Boer, F. S. 2005b. A proof outline logic for object-oriented programming. *Theoret. Comput. Sci. 343*, 413–442.

Pierik, C., Clarke, D., and de Boer, F. S. 2005. Controlling object allocation using creation guards. In *Formal Methods*, Lecture Notes in Computer Science, vol. 3582, Springer, 59–74.

Poetzsch-Heffter, A. and Müller, P. 1999. A programming logic for sequential Java. In *Proceedings of the European Symposium on Programming Languages and Systems*. Lecture Notes in Computer Science, vol. 1576, 162–176.

Reynolds, J. C. 1981. *The Craft of Programming*. Prentice-Hall.

Reynolds, J. C. 1982. Idealized Algol and its specification logic. In *Tools and Notions for Program Construction*, D. Néel Ed., Cambridge University Press, 121–161.

Reynolds, J. C. 1998. *Theories of Programming Languages*. Cambridge University Press.

Rosenberg, S., Banerjee, A., and Naumann, D. A. 2010. Local reasoning and dynamic framing for the composite pattern and its clients. In *Proceedings of the Verified Software: Theories, Tools, Experiments*. Lecture Notes in Computer Science, vol. 6217, 183–198. (`http://www.cs.stevens.edu/~naumann/pub/VERL/`).

Roth, A. 2005. Specification and verification of encapsulation in Java programs. In *Proceedings of the Formal Methods for Open Object-Based Distributed Systems (FMOODS)*. M. Steffen and G. Zavattaro Eds., Lecture Notes in Computer Science, vol. 3535, 195–210.

Roth, A. 2006. Specification and verification of object oriented software components. Ph.D. thesis, Karlsruhe Institute of Technology.

Schmitt, P. H., Ulbrich, M., and Weiß, B. 2010. Dynamic frames in Java dynamic logic. In *Proceedings of the Formal Verification of Object-Oriented Software (FoVeOOS) – (Revised Selected Papers)*. Lecture Notes in Computer Science, vol. 6528, 138–152.

Schwinghammer, J., Yang, H., Birkedal, L., Pottier, F., and Reus, B. 2010. A semantic foundation for hidden state. In *Proceedings of the Foundations of Software Science and Computational Structures*. Lecture Notes in Computer Science, vol. 6014, 2–17.

Shaner, S. M., Leavens, G. T., and Naumann, D. A. 2007. Modular verification of higher-order methods with mandatory calls specified by model programs. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications*. 351–368.

Smans, J., Jacobs, B., Piessens, F., and Schulte, W. 2008. An automatic verifier for Java-like programs based on dynamic frames. In *Proceedings of the Fundamental Aspects to Software Engineering*. Lecture Notes in Computer Science, vol. 4961, Springer, 261–275.

Smans, J., Jacobs, B., Piessens, F., and Schulte, W. 2010. Automatic verification of Java programs with dynamic frames. *Formal Aspects of Computing 22,* 3–4, 423–457.

Thamsborg, J., Birkedal, L., and Yang, H. 2012. Two for the price of one: Lifting separation logic assertions. *Log. Meth. Comput. Sci. 8*, 3.

von Oheimb, D. and Nipkow, T. 2002. Hoare logic for NanoJava: Auxiliary variables, side effects, and virtual methods revisited. In *Formal Methods*, Lecture Notes in Computer Science, vol. 2391, 89–105.

Zee, K., Kuncak, V., and Rinard, M. C. 2008. Full functional verification of linked data structures. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 349–361.

Received July 2011; revised November 2012; accepted March 2013