Local Reasoning for Global Invariants, Part I: Region Logic

ANINDYA BANERJEE, IMDEA Software Institute and Kansas State University
DAVID A. NAUMANN and STAN ROSENBERG, Stevens Institute of Technology

Dedicated to the memory of Stephen L. Bloom (1940–2010).

Shared mutable objects pose grave challenges in reasoning, especially for information hiding and modularity. This article presents a novel technique for reasoning about error-avoiding partial correctness of programs featuring shared mutable objects, and investigates the technique by formalizing a logic. Using a first-order assertion language, the logic provides heap-local reasoning about mutation and separation, via ghost fields and variables of type "region" (finite sets of object references). A new form of frame condition specifies write, read, and allocation effects using region expressions; this supports a frame rule that allows a command to read state on which the framed predicate depends. Soundness is proved using a standard program semantics. The logic facilitates heap-local reasoning about object invariants, as shown here by examples. Part II of this article extends the logic with second-order framing which formalizes the hiding of data invariants.

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., Naumann, D. A., and Rosenberg, S. 2013. Local reasoning for global invariants, Part I: Region logic. *J. ACM* 60, 3, Article 18 (June 2013), 56 pages.
DOI: http://dx.doi.org/10.1145/2485982

1. INTRODUCTION

Typed, object-oriented languages like Java are widely used. One reason for the popularity of typed object-oriented languages is their strong support for modularity in the

This is an expanded and revised version of a paper originally appearing in *European Conference on Object-Oriented Programming*, 2008.
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 and S. Rosenberg were supported in part by US NSF grants CNS-0627338, CRI-0708330, CCF-0429894, and CCF-0915611. D. A. Naumann was partially supported 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 and S. Rosenberg, Stevens Institute of Technology, Castle Point on Hudson, Hoboken, NJ 07030-5991; email: naumann@cs.stevens.edu.

DOI: http://dx.doi.org/10.1145/2485982

form of the "class" and "package" (module) constructs, which provide encapsulation boundaries by means of scoping and typing. Because objects are mutable and dynamically allocated, there is another dimension of potential modularity: *Local reasoning* [O'Hearn et al. 2001] aims to focus on the relatively small number of objects or locations pertinent to the execution of a particular program phrase or the specification thereof. Local reasoning is difficult to achieve because the local structure of the heap, and the locality properties of programs, are not explicitly represented by program syntax or conventional logic formulas. Furthermore, sharing of mutable objects can and often does compromise the encapsulation that programmers intend to provide using classes and packages.

This two-part article addresses the challenge of modular reasoning about programs written in such languages, especially procedure-modular reasoning and its automation using theorem provers. Procedure-modular reasoning simply means: (a) each procedure body is verified separately, and (b) in verifying a procedure body, reasoning about any procedure call in the body is done in terms of the specification rather than the implementation of the called procedure. For modular reasoning, data invariants over encapsulated state play a critical role—which is sufficiently well known to be a standard topic in programming textbooks. Yet sound modular reasoning is difficult to achieve, in part because data invariants are "global", in two senses.

Invariants are global in a temporal sense: an invariant is a state predicate intended to hold in many states of a computation. At the least, a package invariant holds whenever control enters or leaves the package, that is, on calls and returns of its public procedures [Hoare 1972]. For state shared among concurrent threads, invariants hold when control may be transferred between threads. Concurrency is not a central concern in this paper. But many object-oriented programs are structured in ways that involve "reentrant callbacks" [Barnett et al. 2004] which, like concurrency, yields control flow that is not in accord with the hierarchical structure of program modules. To cope with reentrancy and local reasoning, various disciplines have been proposed, including some deployed as special type systems. Like type systems in general, these disciplines hinge on "all-states" invariants. The most successful such discipline is ownership (e.g., Müller [2002], Clarke and Drossopoulou [2002], and Boyapati et al. [2003]), which is widely applicable; yet many design patterns involve local object structures for which ownership is not applicable.

Invariants are also global in a "spatial" sense (where space refers to memory or state). An obvious informal notion of class invariant is a formula that denotes a condition on instance fields. Local reasoning would then focus on maintaining the invariants of those objects directly involved in the execution of a particular program phrase. However, the intention is that the object invariant holds for all instances of the class—a spatially global condition. Furthermore, the invariant for an object often refers to other objects, ranging from those that serve as its internal representation to those that are peers in heterogeneous clusters visible to clients of an API. The granularity of useful locality varies widely and does not simply correspond to the syntactic structure of classes and packages. Notions like ownership, permissions, and capabilities have been proposed to express, enforce, and exploit locality for invariants—and these notions are themselves based on all-states invariants that impose structural constraints on the entire heap.

In addition to local and modular reasoning, automation is also critical for scalability. In view of recent advances in automated theorem proving, especially first order SAT Modulo Theories (SMT) provers, we seek means to formulate specifications of procedures, and class/package invariants, in first order logic with a minimum of novel logical or notational machinery. We address the standard embodiment of modularity for invariants, namely hiding them: They do not appear in specifications of the

procedures exported by an interface, but may be used for reasoning about implementations within the module, as famously articulated by Hoare [1972]. We address locality in a granularity-neutral way: An invariant may pertain to a single object, or an object and its internal representation, or a heterogeneous cluster of client-accessible objects. Such clusters arise in many design patterns and application programs. Locality is embodied in frame conditions: the part of a procedure specification that designates what part of the state is susceptible to change, together with frame-based reasoning that "all else is unchanged". A command's frame condition is sometimes called its "footprint" and often specified by an explicit "modifies clause".

The key ideas that we investigate are inspired by separation logic [O'Hearn et al. 2001], in which specifications implicitly designate local footprints on which a command acts. Our approach is to reason about footprints explicitly, as sets of locations expressed in terms of sets of objects (references) and their fields. Use of explicit footprints for frame conditions was pioneered by Kassios, who dubbed it "dynamic framing" [Kassios 2011]. To cater for reasoning by SMT solvers, we express footprints using mutable *auxiliary* fields and variables (commonly known as *ghost* state). This minimizes the need for inductively defined predicates in specifications, and it entirely avoids two key features by which separation logic achieves elegant and concise specifications: the "separating conjunction" of the assertion language and the implicit specification of frame conditions via a special interpretation of preconditions in Hoare triples. The price to be paid in our approach is the need for programmers or tools to annotate programs with ghost instrumentation.

Contributions and Outline. One of our contributions is to develop the idea of dynamic framing in the form of conventional—but stateful!—frame conditions for procedures. A second contribution is a novel notion of all-steps frame condition, the "dynamic boundary", that specifies stateful encapsulation boundaries and supports hiding of invariants. This two-part article explores these ideas by formalizing and using a Hoare logic with explicit frame conditions. We call it "region logic" due to the key role of what we call *regions*: sets of object references. Part II of the article [Banerjee and Naumann 2013] focuses on reasoning under hypotheses, that is, assumed specifications of procedures, and on the hiding of package invariants. Part I lays the foundation by formalizing a Hoare logic with frame conditions that may involve ghost state, for a programming notation similar to Java source code. Hoare logic is a convenient setting in which to address the sequential composition of stateful frame conditions, which is an issue because effects can have effects. (In other words, a command can interfere with a frame condition, as we explain.) Formalization of a logic also facilitates comparison with other work, in particular separation logic. Our logic is proved sound and examples are given to illustrate the ideas and indicate the range of applicability. Let us emphasize, however, a virtue of stateful frame conditions: They do not require a really new logic, but rather make effective use of ordinary logic and program annotations.

Automated verification tools typically embody proof rules at the level of procedures and modules but not at the level of individual commands in procedure bodies. Rather, verification conditions are generated. But Hoare logic models, and even guides the design of, the axiomatic semantics embodied by verification conditions. Elsewhere we explore the automation of our specifications, including a pragma for framing as well as automated checking of the separation judgment used in our frame rule [Banerjee et al. 2008a; Rosenberg 2011; Rosenberg et al. 2010].

In more detail, Part I is organized as follows. Section 2 sketches an example to illustrate features of the logic, including frame conditions for commands as well as framing and local reasoning for global invariants. Section 3 formalizes the programming language and Section 4 presents the assertion language, essentially sorted first-order

logic with a region sort equipped with set theoretic operations. Section 5 formalizes effects using regions. Section 6 investigates the separation of a formula's footprint from the write effect of a command. Section 7 defines correctness statements for commands, gives the proof rules including a frame rule, and proves soundness for the logic. Section 8 presents worked examples involving reasoning with loops and allocation. Section 9 discusses related work at some length, including the important precursors mentioned above and also our prototype verification tool based on the logic. Section 10 concludes Part I.

It is in procedure specifications that frame conditions play their most important role, as emphasized by examples in Part I. However, procedures are not included in the logic formalized in Part I. This lets us focus on other aspects of the development, laying groundwork for Part II. Inheritance is not considered in either part, as it is an orthogonal issue and our approach is compatible with the standard techniques of supertype abstraction and behavioral subtyping [Leavens and Naumann 2013; Liskov and Wing 1994].

Part II of this article is concerned with information hiding. We extend the programming language to include procedures and a simple notion of package, termed "module" therein. Correctness statements are generalized to judgments with hypotheses that specify procedures. We propose to expose in each module interface a "dynamic boundary", essentially a frame condition that must be respected by clients and which prevents them from interfering with internal invariants. We develop proof rules for the hiding of module invariants, derived from a "second order frame rule" like that of separation logic [O'Hearn et al. 2009]. Hiding means that an invariant is assumed (and must be reestablished) by implementations of a module's procedures, but is not mentioned in interface specifications of those procedures.

Dynamic boundaries designate an all-steps notion of frame condition. To define that, a *transition semantics* (i.e., small-step operational semantics) is used for the semantics of correctness judgments. The semantics is unusual in that it involves a sort of big-step semantics for calls to procedures for which specifications are provided as hypotheses. The semantics provides an elementary basis on which to prove soundness of our second-order frame rule—but it comes at the cost of validating all of the proof rules with respect to transition semantics. Direct use of transition semantics to validate proof rules for loops (in Part I) and for the linking of procedure implementations requires somewhat intricate reasoning. These detailed proofs are one difference between Part II and the conference version thereof [Naumann and Banerjee 2010]. Another difference is that we derive from the second order frame rule not only a rule for linking hierarchical modules (as in O'Hearn et al. [2009]) but also a rule for mutual linking of modules with intermodule callbacks.

Although procedures are not formalized in Part I, it would not be difficult to extend the logic of Part I to include procedures and hypothetical judgments, using a pre/post denotational semantics (as in O'Hearn et al. [2009]).[1] It is for semantics of dynamic boundaries that we need small steps.

2. BACKGROUND AND OVERVIEW OF PARTS I AND II

This section sketches a toy example, based on the Composite design pattern [Gamma et al. 1995]. It serves to illustrate issues in reasoning about separation and data invariants (for which reason it was the challenge problem for a workshop [Robby et al 2008])

[1]In the conference version [Banerjee et al. 2008c] we did use denotational semantics. Aside from what is mentioned in the article, the other differences between Part I and the conference version are that here we give a full set of proof rules, both syntax-directed and structural, and we give additional examples.

```
class Comp {
  private lt, rt : Comp;    // children, if any
  specpublic size : int := 1;    // number of descendants
  specpublic parent : Comp;
  ghost desc : rgn;

  method add(x : Comp)
    requires x ≠ null ∧ x.parent = null ∧ ...
    ensures ... ∧ "x is attached as child of self"
    modifies lt, rt, x.parent, "size and desc field of every ancestor of self"
  {
    x.parent := self;
    if lt = null then lt := x; else if rt = null then rt := x; else diverge; fi;
    var p : Comp := self;
    while p ≠ null do p.size := p.size + x.size; p := p.parent; od
    foreach o in ancestors(self) do o.desc := o.desc ∪ x.desc; od
  }
  ...    // other methods
}
```

Fig. 1. Initial sketch of an embodiment of the Composite design pattern. The frame condition "*size* and *desc* field of every ancestor of self" is formalized in Eq. (6).

and our approach to these issues. Another example, based on the Observer pattern, is developed in more detail in Section 3.

The code in Figure 1 declares a class of "Composite" objects. Fields lt and rt have type $Comp$, values of which are the improper reference, null, and references to allocated objects of type $Comp$. Each Composite references a parent as well as a maximum of two children. (The limit of two is one of several crude features chosen for simplicity.) Method add is used to add a child to a $Comp$ designated by the implicit receiver parameter, self. Note that "self." is usually elided, for example, in the code of add the test $lt = \mathsf{null}$ abbreviates $\mathsf{self}.lt = \mathsf{null}$.

The modifier specpublic is used in specification languages such as JML [Leavens et al. 2003] to designate private (or protected) fields that are made visible in public specifications like those given here. Like many specifications in practice, those shown in the figure provide useful information that falls short of full functional specification. That will still be the case after we revise the example to flesh out the ellipses and correct some flaws.

The intended use of field $size$ is to hold the number of descendants. Later, we formalize the invariant $size = 1 + sz(lt) + sz(rt)$, where the pure function sz is defined by

$$sz(p : Comp) \;\widehat{=}\; \text{if } p = \mathsf{null} \text{ then } 0 \text{ else } p.size \tag{1}$$

The purpose of the loop in the body of add is to maintain this invariant for all $Comp$s.

Frame Conditions and Separation. A frame condition (often termed "modifies clause") is part of a method's specification that lists parts of the state that may be changed by the execution of the method. Note that a frame condition pertains to all heap locations but its expression mentions only the heap locations that may be modified. In this sense, a frame condition abbreviates a postcondition to the effect that every location has its initial value except those mentioned. This form of specification embodies a significant design decision: The modifiable locations are designated

independent of the conditions under which they may be modified (e.g., for method add, field rt is in fact only written when $lt \neq \mathsf{null}$).

Succinct frame conditions facilitate reasoning about "obvious reasons" for absence of interference between program phrases—in particular frame conditions help avoid the case explosion that arises with naïve reasoning about potential aliasing. For example, imagine a client program where the composite objects represent hierarchically organized visible elements in a graphical user interface window. The client could be an editor for musical compositions, with one window that displays standard score notation and another window for setup of audio effect processing. An operation on one window might "obviously" not interfere with the other, nor with the state of the underlying application data.

The use of frame conditions does not preclude finer information, which can be expressed in postconditions.[2] For example, one conjunct of the ensures clause of add could be $\mathsf{old}(lt) \neq \mathsf{null} \Rightarrow lt = \mathsf{old}(lt)$ (where old expressions are interpreted in the initial state). (More realistically, the postcondition would say that the children of self are its previous children plus c. For that matter, the representation of children might be encapsulated.)

There are two benefits from using conventional frame conditions, that is, those that designate modifiable locations unconditionally. One benefit is to facilitate framing, that is, reasoning that an assertion is unaffected by a command, as in the frame rule discussed later in this overview. Frame rules resemble the axiom or rule of Invariance.[3] For framing to be sound, it suffices for the modifies clause to conservatively approximate all locations that may be written. The other benefit is that unconditional effects compose in a simple manner: the effect of a composed program is often the union of its effects. The expressive effects we explore in the sequel are especially good for framing and mostly, but not always, compose by union.

As an example use of frame conditions, suppose the client program has variables b, c, d of type $Comp$ and variable $audio$ that points to an object with boolean field $acyclic$. Imagine that $audio.acyclic$ indicates the absence of feedback loops in the audio processing setup. Obviously, this value is not changed by the invocation $b.add(c)$, because the frame condition in Figure 1 says fields $lt, rt, size$ are modifiable, not field $acyclic$. Such simple reasoning is sound in languages where distinct field names have distinct locations, as in Java and the language of this article.

In reasoning about the call $b.add(c)$, one might naïvely say the value of $d.size$ remains unchanged because d is not b or c, but this is incorrect: variable d may well have the same value as b, that is, reference the same object. Nor would it be sufficient to require precondition $d \neq b$; the program may write the $size$ field of many objects. What is needed is a precise way to designate the ancestors of self. Then, it can be specified that add writes the $size$ only of ancestors. The client can reason that $d.size$ is not changed by $b.add(c)$ if d is not an ancestor of b.

Recall from Section 1 that one of our goals in this work is to be able to specify such spatial frame conditions with a minimum of logical and notational machinery, in order to support reasoning based on automated first-order theorem provers. To define ancestors is easy enough using the transitive closure of $parent$, but there is another way which avoids the logical complexity of transitive closure. Our approach leans heavily

[2]The form of frame conditions we explore in this article can be extended to express conditional effects, using conditionals in region expressions, but that strays from the main points.

[3]See Apt et al. [2009] or Apt [1981], where these are attributed to Gorelick [1975] and Harel [1979]. Invariance has a side condition that refers directly to the program text, whereas frame rules rely only on specifications.

on regions. In our usage, a *region* is a set of references, possibly also containing null.[4] There is no implication that regions partition the heap. A *location*, in our terminology, is a reference paired with a field name. A set of modifiable locations is designated by a region expression together with a field name. The frame condition of add will include $ancestors(\mathsf{self})`size$; this designates the $size$ fields of objects in the region $ancestors(\mathsf{self})$, using notation explained later.

For expository purposes, we define ancestors in terms of descendants. Our reasoning about descendants is carried out using *ghost state*, that is, mutable auxiliary state that is deleted by the compiler but serves for reasoning about partial correctness. Ghost state must not influence the "underlying computation", that is, it does not appear in branching conditions or assignments to non-ghost variables or fields. In examples, ghost declarations and commands appear in gray boxes.

Field $desc$ in class $Comp$ is ghost state intended to hold the set of (references to) descendants of self. Our use of ghost state relies heavily on invariants.

Localized Global Invariants. Field $size$ is non-ghost data that will be used by the program, for example, there could be a public method $getSize$ that relies on $size$ rather than a time-consuming traversal of the tree. The invariant mentioned at the beginning of Section 2 should hold for every $Comp$, that is, the program maintains $\forall o : Comp \cdot ok(o)$ where o ranges over currently allocated references of type $Comp$ (which excludes null) and ok is defined by

$$ok(p : Comp) \;\widehat{=}\; p \neq \mathsf{null} \wedge p.size = 1 + sz(p.lt) + sz(p.rt). \tag{2}$$

This is a pure first-order, nonrecursive condition, with an interesting property: if it holds for every composite then the structure is acyclic and moreover each $size$ is indeed the count of descendants. To formalize and prove this property requires induction—which is worthwhile in that it convinces us the specification is interesting. On the other hand, the formulation (2) facilitates purely first order reasoning about the invariant being preserved by the code.

The condition $\forall o : Comp \cdot ok(o)$ is brutally global in quantifying over all nonnull references of type $Comp$. It could be made slightly less global by restricting o to range over some designated pool of objects of interest. Global conditions in specifications, or as data invariants, might seem inimical to local reasoning and frame conditions. What matters, however, is not whether some reasoning is expressed in terms of global conditions—but rather, whether there are effective ways to factor out parts as they are relevant. Owing to the form of the definition of $ok(p)$, the invariant (2) admits local reasoning quite handily. Suppose that before an invocation $b.add(c)$ we have $\forall o : Comp \cdot ok(o)$. The code falsifies ok at b, then restores it but falsifies ok at $b.parent$, and so forth. The loop maintains as invariant that all composites except p are ok, until p becomes null. A relevant factorization of this global invariant is obtained by splitting it into the equivalent formula $ok(p) \wedge (\forall o : Comp \cdot o \neq p \Rightarrow ok(o))$. Later, we show how such splitting is used in frame-based local reasoning.

As another example, one can imagine a programmer declaring as an object invariant the condition $(lt = \mathsf{null} \vee lt.parent = \mathsf{self}) \wedge (rt = \mathsf{null} \vee rt.parent = \mathsf{self})$ to capture the linking on which the implementation of add relies. The intent is for this to hold of all relevant objects; let us re-state it as such:

$$\forall o, p : Comp \cdot o.parent = p \iff p.lt = o \vee p.rt = o \tag{3}$$

One benefit of global invariants in this style is that they offer granularity-neutral treatment of invariants. The examples so far center on a single object and those

[4] In Banerjee et al. [2008c], regions cannot contain null.

directly reached from its fields. In more complicated situations, the natural granularity may involve a cooperating cluster of objects, with each cluster more or less separate from other clusters. For example, each effect processing setup in our music editor may be represented by a number of interdependent objects, but each setup is entirely separate from the others. The Observer example studied later has a natural granularity: a subject and its observers. (In its proof we split off a conjunct for one cluster just like we split off $ok(p)$ above.) Complete separation is not always in evidence: an observer may also play the role of subject, observed by others. This is akin to the dependencies of a Composite on both its children and parents.

Given (3), it becomes possible to enforce that each $desc$ field holds the set of descendants. Consider the following condition on a composite o:

$$o \in o.desc \wedge (o.parent = \mathsf{null} \vee o \in o.parent.desc) \wedge o.desc\text{`}desc \subseteq o.desc \tag{4}$$

The second conjunct can be read as "o is its parent's descendant, if it has a parent". The third conjunct expresses that $desc$ is transitively closed, using our *field image* notation, $\text{`}desc$, explained in this article. It is equivalent to this formula: $\forall p \in o.desc \cdot p.desc \subseteq o.desc$. The idea is to find conditions that, if satisfied by all composites, are sufficiently strong to ensure that $o.desc$ is the set of o's descendants—while avoiding inductive predicates or functions—just as (2) ensures that the $size$ fields are accurate. (Sufficient conditions are explored in Rosenberg et al. [2010]; the illustrative ones given here are not enough.) With descendants in hand, we can define

$$ancestors(p) = \{o \in \mathsf{alloc} \mid p \in o.desc\}. \tag{5}$$

The special variable alloc contains the set of allocated object references.

Image expressions like $o.desc\text{`}desc$ play an important role in frame conditions. The r-value of this expression is the union of all $p.desc$ where p ranges over elements of the region $o.desc$. The l-value of this expression is a set of locations, namely the locations of the $desc$ fields of objects in $o.desc$. Similarly, the l-value of $ancestors(\mathsf{self})\text{`}size$ is a set of locations, namely the $size$ fields of ancestors of self—just what we want in the frame condition for add.

The price we pay for this precise frame condition is that the $desc$ field must be updated to maintain (4). The update is for reasoning, not execution, so it may as well be expressed succinctly; see the foreach command in Figure 1. Like any update, the update of $desc$ is included in the frame condition.

The expression $ancestors(\mathsf{self})\text{`}size$ in the frame condition is different from the expression $\{x\}\text{`}parent$ there. (In the frame condition, $x.parent$ is sugar for $\{x\}\text{`}parent$. We will come back to this point in more detail in Section 3 where we will be precise on when such sugaring is permissible.) Note that $ancestors(\mathsf{self})$ is *stateful*: its r-value following a call is different from its r-value in the pre-state of a call because, by definition (5), $ancestors(\mathsf{self})$ depends on the field $desc$ and $desc$ is mutable. By contrast, x is a value parameter; in a postcondition it refers to the argument value (and so within the procedure implementation it is not assignable).

To see the impact of a stateful frame condition, let $b, c, d : Comp$ point to distinct objects and let c, d be roots, that is, $ancestors(c) = \{c\}$ and $ancestors(d) = \{d\}$. Consider the sequence $b.add(c);\ c.add(d)$. It attaches c as child of b,[5] which gives c new ancestors. According to the frame condition, the call $c.add(d)$ changes the $desc$ fields of the current ancestors of c. The frame condition interprets the region expression $ancestors(\mathsf{self})$ in the pre-state, and its interpetation in the pre-state of the second call to add is $\{c\} \cup ancestors(b)$; this is different from the interpretation of $ancestors(\mathsf{self})$ in the pre-state of the first call, where it is simply $\{c\}$.

[5]Or diverges if b already has two children.

As a design choice that caters for automation, we do not use sets of sets. Rather, the image of a field of type rgn is flattened, as described above for $o.desc`desc$. In general, if $f$ is a field of reference type and $G$ is a region expression, then the r-value of $G`f$ is the set of v such that $v = o.f$ for some $o \in G$—that it is the usual image of a set under a relation. However, if f is of type rgn, then $G`f$ is the union of the f-values.

Abstraction and Information Hiding. There is an inconsistency in the declarations of Figure 1. Fields rt and lt appear in the frame condition, which is part of the public interface used by client code outside class $Comp$, but these fields are declared *private*. (That is, they are visible only within code of $Comp$,and they are an implementation detail, so not suitable to be specpublic.) There is an established way to avoid exposing internal fields in interfaces. A *data group* is a public field name designated as an abstraction of some private (or protected) fields [Leino 1998; Leino et al. 2002]. We revise our example by declaring

$$\text{private } lt, rt : Comp \text{ in datagroup } chldrn;$$

We revise the frame condition to use self.$chldrn$ in place of self.lt and self.rt. The final version is

$$\text{modifies } chldrn, x.parent, ancestors(\text{self})`size, ancestors(\text{self})`desc. \tag{6}$$

Within the scope of lt, rt, the definition of $chldrn$ must be visible: in reasoning about the implementation of some method in $Comp$, we need to know that add interferes with lt, rt. Data groups are particularly useful in connection with subclassing, but this topic is beyond our scope. So too are mechanisms such as model fields for abstraction in pre- and postconditions (see Section 9).

An extreme case of abstraction pertains to representation invariants, that is, those that pertain entirely to internal state—and so can be hidden entirely. As an example, suppose we want client programs to have access to the size of a composite. Our declaration does not allow client code to refer to field $size$ but there could be a public method $getSize$. It might have no specification beyond its suggestive name, or it might even be specified in terms of the recursive definition of the size of a tree. But the code would rely on field $size$, and thus on invariant $\forall o : Comp \cdot ok(o)$. This is an example of an invariant that pertains to state that should be encapsulated so clients cannot interfere with it. As articulated by Hoare [1972], the hiding of internal invariants creates a mismatch: the invariants should not appear at all in the public interface, but should be required as precondition and ensured as postcondition for the purpose of checking the implementation of add and any other method of $Comp$. The ellipses in the pre- and postcondition of add (Figure 1) would then be read as empty or as the invariant.

Hoare's mismatch is sound insofar as encapsulation prevents clients from interfering with module internals. A popular concept for encapsulation of shared mutable objects is ownership, which imposes hierarchical structure on the heap and restricts updates (or access) to ensure that an invocation on some receiver o can only update locations in objects transitively owned by o (e.g., Clarke and Drossopoulou [2002] and Aldrich and Chambers [2004]). One facet of ownership is that it serves to encapsulate invariants that depend only on owned objects. Another facet is that ownership provides a kind of implicit frame condition for clients: the owner is always allowed to update its owned objects, so that the owned objects need not appear in the frame condition. However, ownership is not enough as witnessed by the composite pattern.

The composite pattern has been raised as a specification challenge [Leavens et al. 2007; Robby et al 2008] because, despite the hierarchical structure of objects, it does not fit ownership: clients may have references to internals. For example, a client

program could have variables b, c of type $Comp$, with c a descendant of b. Ownership systems disallow this, or prevent the client from doing useful things with c.

One objective of our work is to provide sound and flexible means to hide internal invariants at convenient granularities including but not limited to those for which ownership is suited. Part II of this article develops a "second order frame rule" that accounts for the hiding of an invariant within a module, where a module consists of one or more classes. This account uses a straightforward notion of hypothetical correctness: a command is verified under assumptions that comprise some procedure specifications. The form of our second order frame rule, adapted from the "hypothetical frame rule" introduced by O'Hearn et al. [2009], caters for a single invariant formula such as the global invariants discussed previously. The key novelty is to include, in a module interface, a kind of frame condition, which we call the *dynamic boundary*, that designates the encapsulation boundary within which invariants are hidden. In Part I of this article, we lay the groundwork by formalizing in region logic the key ingredients of framing, namely, regions, separation, and frame conditions.

A Logic with Stateful Frame Conditions. Elsewhere, we present experiments with this approach to framing and invariants, including verification of the Composite implementation and non-trivial clients using an automated verifier based on axiomatic semantics [Banerjee et al. 2008a; Rosenberg 2011; Rosenberg et al. 2010]. Here, our goal is to investigate the approach by formalizing it as a Hoare logic.

Verification tools are typically compositional at the level of methods and modules. For a clear understanding of compositionality of effects in sequenced commands as well as in mismatched method specifications, a logic is an appropriate formalization. Within commands, verifiers are often based on verification conditions [Floyd 1967]—weakest preconditions or symbolic execution—rather than directly implementing compositional proof rules. However, such rules bring to light the technical issues, disentangled from issues such as heap model and avoidance of redundancy in weakest precondition formulas.

One pleasant feature of compositionality at the level of commands is that, for atomic commands, we can use "small axioms" [O'Hearn et al. 2001] that focus on the local part of the state. For example, the axiom for field update can be instantiated for the command $p.size := p.size + c.size$ as follows:

$$\{\, p \neq \mathsf{null} \wedge y = p.size + c.size \,\}\ p.size := p.size + c.size\ \{\, p.size = y \,\}\ [\mathsf{wr}\ p.size]\,.$$

Variable y serves to snapshot an initial value. Note that for brevity the frame condition is written in square brackets. The tag, wr, indicates a write effect; the tag is often omitted, as writes are the most oft-occurring effect.

Our logic includes a so-called "frame" rule inspired by that of separation logic [O'Hearn et al. 2001; Reynolds 2002]. The purpose is to conjoin a formula to both pre- and postcondition, if the command cannot interfere with the value of the formula. For example, a procedure call rule yields $\{\, Q \,\}\ b.add(c)\ \{\, Q' \,\}\ [\varepsilon]$ where ε is the frame condition and Q, Q' the pre- and postcondition given by the specification, all instantiated for b and c. In particular, from (6), ε is the list of write effects

$$b.chldrn,\ c.parent,\ ancestors(b)\text{`}size,\ ancestors(b)\text{`}desc.$$

In a proof, we might choose to strengthen Q to some P that implies d is not in the ancestors of b. From $\{\, P \,\}\ b.add(c)\ \{\, Q' \,\}\ [\varepsilon]$ we want to derive the following, by framing the formula $d.size = 10$.

$$\{\, P \wedge d.size = 10 \,\}\ b.add(c)\ \{\, Q' \wedge d.size = 10 \,\}\ [\varepsilon]\,. \tag{7}$$

This is given by our frame rule, with two provisos written $P \vdash \delta$ frm $(d.size = 10)$ and $P \wedge d.size = 10 \Rightarrow \delta \between \varepsilon$, where δ is the read effect $\mathsf{rd}\, d, \mathsf{rd}\, d.size$. The "framing judgment"

$$P \vdash \delta \text{ frm } d.size = 10$$

says that in P-states, the value of formula $d.size = 10$ depends only on the values of d and $d.size$. Here $d.size$ is a sugared form of $\{d\}`size$. The operator $\between$ generates a conjunction of region disjointness formulas, given a list of read effects and a list of write effects. In this example the separator $\delta \between \varepsilon$ compares the relevant read effect $ancestors(b)`size$ and the relevant write effect $d.size$. The comparison yields the single disjointness formula, $\{d\} \# ancestors(b)$, which is equivalent to $d \notin ancestors(b)$. (The operator $\between$ is defined in Section 6.2.)

As another example, suppose some method in class $Comp$ needs to swap two non-null children, say using temporary t in the command $C \mathrel{\widehat{=}} t := lt; lt := rt; rt := t$. We want to show that C maintains the invariant, that is, this judgment:

$$\{\, \forall o : Comp \cdot ok(o) \,\}\ C\ \{\, \forall o : Comp \cdot ok(o) \,\}\ [t, \mathsf{self}.lt, \mathsf{self}.rt]\,.$$

By reasoning about assignments (proof rules in Figure 16) and the definition of $ok(o)$, we can show

$$\{\, ok(\mathsf{self}) \,\}\ C\ \{\, ok(\mathsf{self}) \,\}\ [t, \mathsf{self}.lt, \mathsf{self}.rt]\,. \tag{8}$$

Now we use the frame rule to conjoin to (8) $\forall o : Comp \cdot o \neq \mathsf{self} \Rightarrow ok(o)$, which yields pre- and postcondition $ok(\mathsf{self}) \wedge (\forall o : Comp \cdot o \neq \mathsf{self} \Rightarrow ok(o))$, which simplifies to $\forall o : Comp \cdot ok(o)$. (Formally, we use the rule of consequence for that simplification.) To use the frame rule, we find read effects that frame the formula $\forall o : Comp \cdot o \neq \mathsf{self} \Rightarrow ok(o)$. These are as follows (using $\setminus$ for set subtraction):

$$\begin{array}{l} \mathsf{rd}\,\mathsf{alloc}, \mathsf{rd}\,\mathsf{self}, \\ \mathsf{rd}\,(\mathsf{alloc} \setminus \{\mathsf{self}\})`rt,\ \mathsf{rd}\,(\mathsf{alloc} \setminus \{\mathsf{self}\})`lt, \\ \mathsf{rd}\,(\mathsf{alloc} \setminus \{\mathsf{self}\})`size,\ \mathsf{rd}\,(\mathsf{alloc} \setminus \{\mathsf{self}\})`lt`size,\ \mathsf{rd}\,(\mathsf{alloc} \setminus \{\mathsf{self}\})`rt`size \end{array} \tag{9}$$

(In Section 6, we show how the read effects are obtained.) Next, we must show the separation condition. Note that there are no variables in common between the write effects and the read effects. However, fields lt, rt occur in both. (Field $size$ is read but not written.) Hence, the separation condition is

$$\{\mathsf{self}\} \# (\mathsf{alloc} \setminus \{\mathsf{self}\}),$$

which is equivalent to $\mathsf{self} \notin (\mathsf{alloc} \setminus \{\mathsf{self}\})$.

To emphasize this reasoning idiom, we sketch another example. Our music editing application may have a number of windows open at once. Consider an editing operation that applies to audio setups. The application maintains invariant $\forall w : AudioWindow \cdot valid(w)$ where $valid$ imposes conditions on the audio setup and its rendering in the window. To reason about an operation, $edit$, one would focus on the body of $edit$, which updates fields of self and some other objects. These would falsify $valid(\mathsf{self})$ and then for non-trivial reasons restore it. Having established that the body of $edit$ preserves $valid(\mathsf{self})$, we would frame[6] the formula $\forall w : AudioWindow \cdot w = \mathsf{self} \vee valid(w)$ and then check that it is separate from the writes of $edit$.

An interesting but annoying feature of the logic is that stateful effects are subject to interference. This may explain why the simple approach we explore here had not

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

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

(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	$::= \mathsf{skip} \mid x := F \mid x := \mathsf{new}\ K \mid x := x.f \mid x.f := F$
		$\mid \mathsf{if}\ E\ \mathsf{then}\ C\ \mathsf{else}\ C \mid \mathsf{while}\ E\ \mathsf{do}\ C \mid C\,;C \mid \mathsf{var}\ x : T\ \mathsf{in}\ C$

Fig. 2. Programming language. We confuse category names with typical elements (e.g., E). The *atomic commands* are skip and the various forms of assignment.

been pursued long before the work of Kassios [2011]. Consider this subsequence in an unfolding of the loop body of add.

$$p := p.\mathit{parent};\ p.\mathit{size} := p.\mathit{size} + x.\mathit{size}; \tag{10}$$

What is the effect? The effect of the assignment to $p.\mathit{size}$, in isolation, can be expressed by $\mathsf{wr}\, p.\mathit{size}$, because the initial value of p is indeed the object whose size is updated. But for the displayed sequence, it is not the initial value of p but rather of $p.\mathit{parent}$ whose size gets updated. In our terminology, the effect $\mathsf{wr}\, p.\mathit{size}$ is not *immune* from the effect, $\mathsf{wr}\, p$, of the first assignment. The proof rule for sequencing combines effects of the commands in sequence, but imposes an immunity condition on them. To verify the displayed sequence, we first use a rule of state-dependent effect subsumption, ascribing to $p.\mathit{size} := p.\mathit{size} + x.\mathit{size}$ the effect $\mathsf{wr}\ \mathit{ancestors}(\mathsf{self})\text{`}\mathit{size}$, which is sound owing to an invariant of the loop: $p \in \mathit{ancestors}(\mathsf{self})$. The effect $\mathsf{wr}\ \mathit{ancestors}(\mathsf{self})\text{`}\mathit{size}$ is immune from update of p. Immunity is the topic of Section 6.3.

One way to implement these ideas in an automated verifier is to desugar frame conditions to their semantics as postconditions. Then nothing explicit needs to be done about immunity. The effect of assignments on expressions in frame conditions is calculated just like the effect of assignments on expressions in other formulas. In our experiments using an SMT solver we also found that the prover is able to handle quantified global invariants without assistance, at least those like ok that refer concretely to the state. On the other hand, in case of inductively defined predicates (encoded as uninterpreted functions plus axioms) and predicates expressed using procedural abstraction, some explicit framing hints are valuable [Rosenberg et al. 2010]. A hint generates a framing judgment as additional verification condition, as we discuss in Section 6 where we also provide a deductive system for framing judgments.

3. PROGRAMMING LANGUAGE

This section presents an illustrative language for which we formalize the programming logic.

3.1. Syntax

A program consists of a command C in the context of some class declarations. The grammar for types, commands etc. is in Figure 2. A class declaration $\mathsf{class}\ K\ \{\ \overline{T}\ \overline{f}\ \}$ introduces a type name K; values of this type are *null* and references to mutable objects with typed fields $\overline{f} : \overline{T}$. Here and throughout, identifiers with an overline range over lists. In addition to int and reference types, there is type rgn with values ranging over finite sets of references. Field names are globally unique, by assumption, so we can say $f : T$ to abbreviate that there is a class that declares field $f : T$.

There is no need in this article for syntax to distinguish between ghost fields and variables (i.e., those that instrument the program, or sometimes just its assertions, to facilitate reasoning [Owicki and Gries 1976; Reynolds 1981]) and ordinary ones.

Indeed, region expressions cannot influence control flow or the value of nonregion fields/variables, so they can only serve as ghosts for reasoning. Ghost state of other types is also useful. In Section 8.3, we consider the elimination of ghost instrumentation.

Program expressions (E in Figure 2) do not depend on the heap: $y.f$ is not an expression but rather part of the atomic command $x := y.f$ for reading a field. The restriction to heap independent expressions serves, as in separation logic, to simplify rules for reasoning about assignments: we need not worry about chains of dereferences, such as $x.f.g$, in program expressions and the possibility of faults arising out of dereferencing *null* when, for example, either x or $x.f$ denotes *null*. Another consequence of the restriction is that an ordinary assignment, $x := F$, can never fault. It also helps simplify the framing rules (Figure 15).

Region expressions (G in Figure 2) of the form $G`f$ (read "$G$'s image under $f$") are restricted to fields $f$ of some class type $K$ or type rgn. If $f$ is of class type, then $G`f$ denotes the set of f-images of all non-null references in G's denotation; but if $f : \mathsf{rgn}$, then $G`f$ denotes the union of the f-images. This restriction pertains to the use of regions for their r-value. It lets us use a single region type; otherwise, we would need int sets too. In effect, which refer to the l-value, f can have any type.

The example in Section 2 uses set comprehension notation to define the region $ancestors(\mathsf{self})$; this we omit in the formalization, but see Remark 6.3 in Section 6. In effect, we include one special case: The region expression G/K denotes the subset of G of references of type K (including null if it is in G).

There are no binding constructs for region expressions. Field identifiers are considered disjoint from variable identifiers. We define $Vars(G)$, the variables that occur in G, by $Vars(x) = \{x\}$, $Vars(\{E\}) = Vars(E)$, $Vars(G`f) = Vars(G)$, and otherwise as the union of variables of subexpressions.

The following is an important note on the notations $x.f$ and $\{x\}`f$.

Remark 3.1. As mentioned in Section 2, we often use $x.f$ as syntax sugar for $\{x\}`f$, but only in the following cases.

— When f is of type rgn and $\{x\}`f$ occurs in an assertion or effect and is interpreted for its r-value. For $f : \mathsf{rgn}$, this disallows an assignment $y := x.f$ but permits an assertion $b \in x.f$ (which is false when $x = \mathsf{null}$) or an effect $\mathsf{wr}\, x.f`g$ (which refers to an empty set of locations when $x = \mathsf{null}$).
— When f is interpreted for its l-value in an effect (and f has any type).

Other uses risk ambiguity or confusion. For example, for $f : K$ the putative expression $x.f$ would have type K whereas $\{x\}`f$ has type rgn. Also, in case $f : \mathsf{rgn}$ and $y : \mathsf{rgn}$ the syntax allows both commands $y := x.f$ and $y := \{x\}`f$, which differ when x is null.[7]

3.2. Typing

There is an ambient class table comprising a well-formed collection of class declarations. We write $\mathit{Fields}(K)$ for the field declarations $\overline{f} : \overline{T}$ of class K. A field name f uniquely determines the class, $\mathit{DeclClass}(f)$, that declares it; so $\mathit{DeclClass}(f) = K$ iff there is some T such that $(f : T)$ is in $\mathit{Fields}(K)$. Note that int is separated from reference types: there is no pointer arithmetic but references can be tested for equality.

A *typing context*, Γ, is a finite mapping of variable names to types. It is well formed if its domain at least contains the distinguished variable, alloc, with $\Gamma(\mathsf{alloc}) = \mathsf{rgn}$; consequently alloc cannot appear in non-ghost code. In the sequel, we will only consider

[7]For practical use, sugaring in this case is likely to be harmless because the annotation may imply $x \neq \mathsf{null}$. In this paper we err on the side of precision.

$$\Gamma \vdash \varnothing : \mathsf{rgn} \qquad \frac{\Gamma y = \mathsf{rgn}}{\Gamma \vdash y : \mathsf{rgn}} \qquad \frac{\Gamma \vdash E : K}{\Gamma \vdash \{E\} : \mathsf{rgn}}$$

$$\frac{\Gamma \vdash G : \mathsf{rgn} \qquad (f : K') \text{ or } (f : \mathsf{rgn}) \text{ is in } \mathit{Fields}(K)}{\Gamma \vdash G\text{`}f : \mathsf{rgn}} \qquad \frac{\Gamma \vdash G : \mathsf{rgn}}{\Gamma \vdash G/K : \mathsf{rgn}}$$

$$\frac{\Gamma \vdash G_1 : \mathsf{rgn} \qquad \Gamma \vdash G_2 : \mathsf{rgn} \qquad \otimes \text{ is in } \{\cup, \cap, \setminus\}}{\Gamma \vdash G_1 \otimes G_2 : \mathsf{rgn}}$$

Fig. 3. Typing rules for region expressions.

$$\frac{\Gamma \vdash F : T \qquad T = \Gamma x \qquad x \not\equiv \mathsf{alloc}}{\Gamma \vdash x := F} \qquad \frac{(f : T) \in \mathit{Fields}(\Gamma y) \qquad T = \Gamma x \qquad x \not\equiv \mathsf{alloc}}{\Gamma \vdash x := y.f}$$

$$\frac{(f : T) \in \mathit{Fields}(\Gamma x) \qquad T = \Gamma y}{\Gamma \vdash x.f := y} \qquad \frac{K = \Gamma x}{\Gamma \vdash x := \mathsf{new}\ K} \qquad \frac{\Gamma \vdash C \qquad \Gamma \vdash D}{\Gamma \vdash C\,;D}$$

$$\frac{\Gamma \vdash E : \mathsf{int} \qquad \Gamma \vdash C \qquad \Gamma \vdash D}{\Gamma \vdash \mathsf{if}\ E\ \mathsf{then}\ C\ \mathsf{else}\ D} \qquad \frac{\Gamma \vdash E : \mathsf{int} \qquad \Gamma \vdash C}{\Gamma \vdash \mathsf{while}\ E\ \mathsf{do}\ C} \qquad \frac{\Gamma, x : T \vdash C}{\Gamma \vdash \mathsf{var}\ x : T\ \mathsf{in}\ C} \qquad \Gamma \vdash \mathsf{skip}$$

Fig. 4. Typing rules for commands.

well-formed contexts. 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 $dom(\Gamma)$.

The judgment $\Gamma \vdash F : T$ says that in context Γ, region or program expression F has type T. We omit the typing rules for program expressions since they are standard. The typing rules for region expressions appear in Figure 3. The typing rules make some distinctions between region expressions G and program expressions E. In particular, whether variable x is a program expression or a region expression can only be determined by way of its typing. The typing rule for singleton region $\{E\}$ enforces that E is of reference type. The typing rule for region dereference, $G\text{`}f$, checks that f is declared to be of reference type or type rgn.

The judgment $\Gamma \vdash C$ says C is a well-formed command in context Γ; furthermore there is no assignment to variable alloc. The typing rules for commands appear in Figure 4; typing prevents "field not defined" errors. For brevity, we omit a Boolean type: The guard for an if- or while-command has type int and any nonzero value is interpreted as true.

Our treatment of typing is intended to ensure coherence of definitions in subsequent sections, and in particular to streamline the formulation of the proof rules for correctness, with a minimum of fuss. Because the language is first order, we can treat bindings rather simply, as the following result shows.

LEMMA 3.2 (CONTEXT EXTENSION FOR COMMANDS). *The following rule is admissible:*

$$\frac{\Gamma \vdash C \qquad x \textit{ does not occur bound in } C}{\Gamma, x : T \vdash C}$$

That is, if the premise is derivable then so is the conclusion, given the proviso. Furthermore, if x occurs bound in C, then $\Gamma, x : T \vdash C$ is not derivable.

PROOF. There is a typing rule for each construct, and no other typing rules. The proof of admissibility goes by induction on the typing derivation of $\Gamma \vdash C$. The only

Notation	Explanation
$Type(o, \sigma)$	type of an allocated reference o
$Dom(\sigma)$	domain of the store (i.e., the local variables)
$\sigma(x)$	variable lookup; if x is alloc returns set of all allocated references
$\sigma(o.f)$	field lookup (requires $o \in \sigma(\mathsf{alloc})$ and f in $Fields(Type(o, \sigma))$)
$\sigma \restriction x$	σ with variable x removed ("tossed") from the domain of the store
$[\sigma \mid x : v]$	overrides σ to map x to v (requires $x \in Dom(\sigma)$)
$[\sigma \mid o.f : v]$	overrides σ to map field f of o to v (requires $o \in \sigma(\mathsf{alloc})$)
$Extend(\sigma, x, v)$	extends σ to map x to value v (requires $x \notin Dom(\sigma)$)
$New(\sigma, o, K, \overline{v})$	extends σ by adding o to alloc and by mapping o to a K-record with field values $\overline{v}$ and type K (requires $o \notin \sigma(\mathsf{alloc})$)
$Fresh(\sigma)$	a non-empty set of non-null references that are not allocated in σ (*Fresh* is an arbitrary function with this property)

Fig. 5. Operations on state σ. Here x ranges over variable names; o is an element of *Ref*; v is a value in σ (i.e., an integer, *null*, or an element of $\sigma(\mathsf{alloc})$).

binding construct is var and it removes the bound variable from the context. That is also why the last part of the lemma holds. □

Remark 3.3. It is straightforward to add subtyping based on subclasses, but to keep the focus on the main novelties, we refrain from formalizing it. In the presence of subtyping, it would be convenient to replace the untyped region type by types of the form $\mathsf{rgn}(K)$; such regions hold references to objects of type K and its subtypes, so untyped regions are retained as $\mathsf{rgn}(Object)$. The distinguished variable alloc could be replaced by a family of variables, so alloc_K would hold the references of type K and its subtypes.

Untyped regions are an important means of abstraction: an interface can expose that there are some objects, without revealing their types.

3.3. Semantics

The semantics is based on conventional program states. We abstract from the concrete representation of states and merely assume the operations in Figure 5 are available.

We assume given a set *Ref* of reference values including a distinguished value, *null*. A Γ-*state* has a global heap and a store. The store assigns values to the variables in Γ and to the variable $\mathsf{alloc} : \mathsf{rgn}$ which is special in that its updates are built into the language semantics as follows: newly allocated references are added and there are no other updates to alloc. In a well-formed state, alloc holds the set of all allocated references and does not contain *null*.

The heap maps each allocated reference to its (immutable) type and field values. The values denoted by class type K (Figure 2) are *null* as well as all allocated references of type K. The values of type rgn are sets of allocated references and *null*; in other words, in a given state σ, regions are subsets of $\sigma(\mathsf{alloc}) \cup \{\mathit{null}\}$. Heaps have no dangling references: there is no "dispose" operation or garbage collection.

We emphasize that a *well-formed* Γ-*state* σ satisfies

$$\sigma(x) \in \sigma(\mathsf{alloc}) \cup \{\mathit{null}\} \text{ for all } x \in dom(\Gamma) \text{ of class type}$$
$$\sigma(r) \subseteq \sigma(\mathsf{alloc}) \cup \{\mathit{null}\} \text{ for all } r \text{ with } \Gamma(r) = \mathsf{rgn}$$

and similarly for object fields. Henceforth, by *state* we always mean well-formed state.

The semantics of program expressions E in state σ, written $\sigma(E)$, is straightforward and omitted. Note that $\sigma(E)$ is always a value (of appropriate type), never *fault*;

$$
\begin{array}{lcl}
\sigma(\{E\}) & = & \{\,\sigma(E)\,\} \\
\sigma(\varnothing) & = & \varnothing \\
\sigma(G_1 \cup G_2) & = & \sigma(G_1) \cup \sigma(G_2) \\
\sigma(G_1 \cap G_2) & = & \sigma(G_1) \cap \sigma(G_2) \\
\sigma(G_1 \setminus G_2) & = & \sigma(G_1) \setminus \sigma(G_2) \\
\sigma(G/K) & = & \{o \mid o \in \sigma(G) \wedge (o = null \vee Type(o,\sigma) = K)\} \\
\sigma(G\text{`}f) & = & \{\sigma(o.f) \mid o \in \sigma(G) \wedge o \neq null \wedge Type(o,\sigma) = DeclClass(f)\} \\
 & & \quad \text{if } f : K \text{ for some } K \\
 & = & \bigcup\{\sigma(o.f) \mid o \in \sigma(G) \wedge o \neq null \wedge Type(o,\sigma) = DeclClass(f)\} \quad \text{if } f : \mathsf{rgn}
\end{array}
$$

Fig. 6. Semantics of region expressions.

moreover, it only depends on the store, not the heap.[8] We now consider the semantics of region expressions G in state σ, written $\sigma(G)$. The semantics of a region variable y is $\sigma(y)$ (see Figure 5). The semantics for other region expressions appear in Figure 6. The meaning of singleton region $\{E\}$ is $\{\sigma(E)\}$, that is, the singleton set containing the value of E. Because E is heap-independent, E is guaranteed to have a value and there is no possibility of a null-pointer dereference. In a given state region expression, $G\text{`}f$ denotes one of two things. If $f$ has class type, then $G\text{`}f$ is the set containing the f-images of all non-*null* references of G that have field f; but if $f : \mathsf{rgn}$, then $G\text{`}f$ denotes the union of the f-images. The meaning of $G_1 \cup G_2$ is the union of the meanings of G_1, G_2, etc.

The transition semantics of commands appears in Figure 7. The semantics operates on configurations of the form $\langle C, \sigma\rangle$ where command C, the *control state*, is typable in some context Γ and σ is a Γ-state. Here, C is an *extended command* that may include the atomic $\mathsf{evar}(x)$, which does not occur in source programs but is used in the semantics to mark the end of the scope of a local variable. Well formedness is preserved by the transition relation, in the following sense: If $\langle C, \sigma\rangle \longmapsto \langle C', \sigma'\rangle$, then there is some Γ' that extends Γ (for local variables) such that $\Gamma' \vdash C'$ and σ' is a Γ'-state. Variable alloc is suitably updated by the transition rule for new, cf. the New function in Figure 5. As in many works (e.g., Apt et al. [2009]) we consider a control state of the form skip; D to be identical to D. This avoids the need for a separate rule for the terminating step of C in a sequence $C; D$.

Every reachable configuration has a successor unless the command is skip. A terminating computation ends in a configuration of the form $\langle \mathsf{skip}, \sigma\rangle$ or else *fault*. If $\langle C, \sigma\rangle \longmapsto^* \langle \mathsf{skip}, \sigma'\rangle$ then σ' is a Γ-state (owing to nesting local variable blocks). We assume the set $VarName$ is infinite, as otherwise the transition for local variable block could get stuck (once we add recursive procedures, in Part II).

The semantics in Figure 7 is parameterized by the allocator *Fresh* that appears in Figure 5. Thus our results encompass deterministic allocators (when the set determined by $\mathit{Fresh}(\sigma)$, for some state σ, is a singleton) as well as the maximally nondeterministic one used in separation logic. There is no deallocation so the domain of the heap only grows, and once allocated the type of a reference never changes.

The following property of semantics can be proved by induction on the length of the computation sequence.

LEMMA 3.4. *For $\Gamma \vdash C$, let σ be a Γ-state, with $x \notin dom(\Gamma)$. Let v be a value of type T. Let τ be the $(\Gamma, x : T)$-state such that $\tau = \mathit{Extend}(\sigma, x, v)$. Then:*

(a) $\langle C, \tau\rangle \longmapsto^*$ *fault iff* $\langle C, \sigma\rangle \longmapsto^*$ *fault*
(b) $\langle C, \sigma\rangle \longmapsto^* \langle \mathsf{skip}, \sigma'\rangle$ *iff* $\langle C, \tau\rangle \longmapsto^* \langle \mathsf{skip}, \mathit{Extend}(\sigma', x, v)\rangle$

[8]The only faults are null dereference in field access and update commands, since we consider programs that satisfy usual Java-style typing rules. We assume for simplicity that arithmetic operators are error-avoiding.

$$\langle x := F,\ \sigma\rangle \longmapsto \langle \mathsf{skip},\ [\sigma \mid x : \sigma(F)]\rangle \qquad \frac{\sigma(y) = null}{\langle x := y.f,\ \sigma\rangle \longmapsto fault}$$

$$\frac{\sigma(y) = o \qquad o \neq null}{\langle x := y.f,\ \sigma\rangle \longmapsto \langle \mathsf{skip},\ [\sigma \mid x : \sigma(o.f)]\rangle} \qquad \frac{\sigma(x) = null}{\langle x.f := F,\ \sigma\rangle \longmapsto fault}$$

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

$$\frac{o \in Fresh(\sigma) \qquad Fields(K) = \overline{f} : \overline{T} \qquad \sigma_1 = New(\sigma, o, K, default(\overline{T}))}{\langle x := \mathsf{new}\ K,\ \sigma\rangle \longmapsto \langle \mathsf{skip},\ [\sigma_1 \mid x : o]\rangle}$$

$$\frac{\langle C,\ \sigma\rangle \longmapsto \langle C',\ \sigma'\rangle}{\langle C\,;D,\ \sigma\rangle \longmapsto \langle C'\,;D,\ \sigma'\rangle} \qquad \frac{\langle C,\ \sigma\rangle \longmapsto fault}{\langle C\,;D,\ \sigma\rangle \longmapsto fault}$$

$$\frac{\sigma(E) \neq 0}{\langle \mathsf{if}\ E\ \mathsf{then}\ C\ \mathsf{else}\ D,\ \sigma\rangle \longmapsto \langle C,\ \sigma\rangle} \qquad \frac{\sigma(E) = 0}{\langle \mathsf{if}\ E\ \mathsf{then}\ C\ \mathsf{else}\ D,\ \sigma\rangle \longmapsto \langle D,\ \sigma\rangle}$$

$$\frac{\sigma(E) = 0}{\langle \mathsf{while}\ E\ \mathsf{do}\ C,\ \sigma\rangle \longmapsto \langle \mathsf{skip},\ \sigma\rangle} \qquad \frac{\sigma(E) \neq 0}{\langle \mathsf{while}\ E\ \mathsf{do}\ C,\ \sigma\rangle \longmapsto \langle C; \mathsf{while}\ E\ \mathsf{do}\ C,\ \sigma\rangle}$$

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

$$\langle \mathsf{evar}(x),\ \sigma\rangle \longmapsto \langle \mathsf{skip},\ \sigma \upharpoonright x\rangle$$

Fig. 7. Small step operational semantics, $\longmapsto$, for commands. In the transition rule for var, the notation $C^x_{x'}$ denotes substitution of x' for x in C.

In the lemma, if T is a reference type, K, then v should be in $\sigma(\mathsf{alloc})$ with $\mathit{Type}(v, \sigma) = K$. Note also that any outcome from τ can be written in the form $\mathit{Extend}(\sigma', x, \_)$, that is, $\mathit{Extend}(\sigma', x, v)$ for some v.

4. ASSERTION LANGUAGE

This section formalizes an assertion language. In brief, it is first order logic with region image expressions and a "points-to" predicate. We begin with an example that will be used through the rest of the article.

4.1. Example: An Implementation of the Observer Pattern

Figure 8 provides code for the Observer pattern. We notice several methods in the figure, including constructor methods $Subject()$ and $Observer(s)$. Although we defer formalizing methods to Part II [Banerjee and Naumann 2013], in this article, we will be considering bodies of these methods as commands in context. For example, the context for the body of *register* is $\mathsf{alloc} : \mathsf{rgn}, \mathsf{self} : Subject, b : Observer$; the context for *update* is $\mathsf{alloc} : \mathsf{rgn}, \mathsf{self} : Subject, n : \mathsf{int}$; and the context for the constructor $Observer(s)$ is $\mathsf{alloc} : \mathsf{rgn}, \mathsf{self} : Observer, s : Subject$.

We now give an explanation of the implementation of the pattern. An object $s : Subject$ has a pointer to a list of observers rooted at obs and all observers of s

```
class Subject {
  obs : Observer; val : int; ghost O : rgn;
  Subject() { obs := null; val := 0; O := ∅; }
  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) { // for use by 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); }
  notify() { // for use by Subject
    cache := sub.get(); } }
```

Fig. 8. Observer pattern.

reside in region O. The current value of s's internal state is in the field val. An object $o : Observer$ has a pointer, sub, to the subject to which it belongs and has a field, $cache$, which contains what o believes to be the current value of its subject. The $nxto$ field points to the next element in the list of observers.

Method $register$ is intended to be used only by code in class $Observer$ and not by clients. It inserts an object o to the front of the subject's list of observers and notifies o of its current state. The method also performs a ghost field update: it adds o to region field O.

Method $update$ updates the current state of the subject and notifies all observers of the subject. Notification is the job of $notify$: it is called on a newly allocated observer from $register$ but more interestingly, also called from within the $update$ method. Like $register$, the method $notify$ is intended to be used only by code in class $Subject$ and not by clients. The call to $notify$ results in a callback to the $Subject$'s get method. The assignment to self.$cache$ in the body of $notify$ can be seen as sugar for the command

$$\mathsf{var}\ t : Subject\ \mathsf{in}\ \mathsf{var}\ u : \mathsf{int}\ \mathsf{in}\ t := \mathsf{self}.sub; u := t.get(); \mathsf{self}.cache := u;$$

where t, u are local variables.

The particular usages of $register$ and $notify$ described here could be formalized by putting the classes $Subject$ and $Observer$ together in a package and considering the methods to be package-scoped. In Part II, we formalize this kind of scoping by considering the two classes to comprise a module.

4.2. Syntax and Semantics of Assertions

Figure 9 gives the grammar for assertions. Quantification is over int and class types only. In the latter case a bounding region is required as in $(\forall x : K \in pool \cdot P)$ where the quantification is over all currently allocated references of type K in region $pool$. Quantifying over allocated objects is usual [Calcagno et al. 2003; Pierik and de Boer 2005b] and important for certain global invariants. There are also formulas for inclusion of regions. For example, $G`f \subseteq H$ is an instance of the syntax $G \subseteq G$ in Figure 9. It says that the $f$-image of every nonnull object in $G$ is in $H$. A convenient feature is the ability to refer to all fields, as in $G`\mathsf{any} \subseteq G$. Here, any is like a data group [Leino

$$
\begin{array}{rl}
P & ::= \; E = E \mid x.f = E \mid G \subseteq G \quad \text{(atomic formulas)} \\
& \quad \mid P \wedge P \mid P \vee P \mid \neg P \\
& \quad \mid (\forall x : \mathsf{int} \cdot P) \mid (\forall x : K \in G \cdot P) \mid (\exists x : \mathsf{int} \cdot P) \mid (\exists x : K \in G \cdot P)
\end{array}
$$

$$
\begin{array}{ll}
\sigma \models E_1 = E_2 & \text{iff } \sigma(E_1) = \sigma(E_2) \\
\sigma \models x.f = E & \text{iff } \sigma(x) \neq \mathit{null} \text{ and } \sigma(x.f) = \sigma(E) \\
\sigma \models G_1 \subseteq G_2 & \text{iff } \sigma(G_1) \subseteq \sigma(G_2) \\
\sigma \models^{\Gamma} \forall x : \mathsf{int} \cdot P & \text{iff } \mathit{Extend}(\sigma, x, v) \models^{\Gamma, x:\mathsf{int}} P \quad \text{for all } v \in \mathbb{Z} \\
\sigma \models^{\Gamma} \forall x : K \in G \cdot P & \text{iff } \mathit{Extend}(\sigma, x, o) \models^{\Gamma, x:K} P \\
 & \quad \text{for all } o \text{ in } \sigma(G) \setminus \{\mathit{null}\} \text{ with } \mathit{Type}(o, \sigma) = K \\
\sigma \models P_1 \wedge P_2 & \text{iff } \sigma \models P_1 \text{ and } \sigma \models P_2 \\
\sigma \models \neg P & \text{iff } \sigma \not\models P
\end{array}
$$

Fig. 9. Formulas: grammar and semantics. Semantics of $\vee$ and $\exists$ is by de Morgan duality.

Syntax Sugar	Explanation
$E \in G$	$\{E\} \subseteq G$
$G = G'$	$G \subseteq G' \wedge G' \subseteq G$
$G \# G'$	$G \cap G' \subseteq \{\mathit{null}\}$
$x.f.g = E$	$\exists y : K \in \mathsf{alloc} \cdot x.f = y \wedge y.g = E$ (where $f : K$)
$\mathsf{type}(K, G)$	$G \subseteq G/K$

Fig. 10. Syntax sugar for formulas. See also Remark 3.1.

1998] in that it stands for all fields.[9] The formula $x.f = E$ is like the points-to predicate in object-oriented separation logic [Parkinson and Bierman 2005]; it says that x is non-null and the value of its f field is E. The semantics is two-valued and classical. We omit the various mathematical types needed for practical application; they can be treated just like int.[10]

Several syntax sugars are defined in Figure 10. Note that in formulas, if $f : \mathsf{rgn}$ then $x.f = G$ can be used as sugar for $\{x\}`f = G$ without ambiguity. But please keep in mind the important Remark 3.1 at the end of Section 3.1.

Rules for the well-formedness judgment $\Gamma \vdash P$ are mostly straightforward, and omitted, but we note the following:

$$
\frac{(f : T) \in \mathit{Fields}(\Gamma x) \qquad \Gamma \vdash E : T}{\Gamma \vdash x.f = E} \qquad\qquad \frac{\Gamma, x : K \vdash P \qquad \Gamma \vdash G : \mathsf{rgn}}{\Gamma \vdash \forall x : K \in G \cdot P}.
$$

The first rule is only applicable for $T \not\equiv \mathsf{rgn}$, because program expressions E cannot be typed as regions (recall Section 3.2). The comparison of region expressions for equality can be done by employing the abbreviation $G = G'$ as in, for example, formulas $x.g = r$ where g and r have type rgn. The second rule disallows quantification over regions and demands that the bound variable x not appear in the bound, G, of the quantification. This facilitates framing and loses no generality.

A minor technicality is that we do not include a rule of context extension for formulas (nor for expressions). As in the case of commands—viz. Lemma 3.2—context extension

[9]We do not formalize data groups, but an ordinary data group would be declared to abstract a specific set of fields. One can treat any as the extreme case of all fields in the entire program.

[10]A boring technicality: An integer expression like $x > 3$ (which evaluates to 1 or 0) can be lifted to the formula $(x > 3) \neq 0$, which, of course, abbreviates $\neg((x > 3) = 0)$.

is admissible in this sense: if $\Gamma \vdash P$ and x does not occur bound in P then $\Gamma, x : T \vdash P$ is derivable. Moreover if x does occur bound in P then $\Gamma, x : T \vdash P$ is not derivable. In brief, a variable cannot occur both free and bound in a formula. Nor can a variable binding be shadowed. This property is sometimes known as *hygiene*.

The semantics of a well-formed formula $\Gamma \vdash P$ is given as a satisfaction relation, written $\sigma \models^{\Gamma} P$ and defined for all Γ-states σ. The definition is in Figure 9. In most cases we elide Γ since it is unchanged throughout. For the semantics of $\forall$, recall that the bound variable ranges over nonnull, allocated references; note the use of *Extend*, which is well defined owing to the hygiene property remarked above.

A formula in context Γ is called *valid* iff it is true in all well-formed states. This use of the term "valid" is appropriate because we consider a fixed model rather than a class of models.

In case we have $\Gamma \vdash P$ and also $\Gamma, x : T \vdash P$, the semantics is unchanged:

$$\sigma \models^{\Gamma, x:T} P \text{ iff } \sigma \restriction x \models^{\Gamma} P \quad \text{for any } (\Gamma, x : T)\text{-state } \sigma. \tag{11}$$

We use Reynolds' notation for substitution in formulas, writing $P/x{\rightarrow}F$ for substitution of F for x in P. For our purposes, substitution is capture-rejecting: we consider $P/x{\rightarrow}F$ to be meaningless if a variable in F would be captured by a binding in P. This only matters in connection with rule SUBST in Figure 17. It is straightforward to show these properties about substitution and update of expressions and formulas:

$$\sigma(G/x{\rightarrow}F) \;=\; [\sigma \mid x : \sigma(F)]\,(G) \tag{12}$$

$$\sigma \models^{\Gamma} P/x{\rightarrow}F \;\text{ iff }\; [\sigma \mid x : \sigma(F)] \models^{\Gamma} P \tag{13}$$

for all σ, x, F, G, P such that $G/x{\rightarrow}F$ and $P/x{\rightarrow}F$ are well formed in Γ.

The assertion language includes integers and we are reasoning about a standard interpretation, so any rules we give will be incomplete according to Gödel. For later reference, we mention some valid formulas that highlight features of the assertion language.

Remark 4.1. Because states are well formed in the sense defined in Section 3.3, the following are valid.

$$\mathsf{null} \notin \mathsf{alloc} \wedge G \setminus \{\mathsf{null}\} \subseteq \mathsf{alloc} \quad \text{(for any region expression } G\text{)}$$
$$x = \mathsf{null} \vee x \in \mathsf{alloc} \quad \text{(for } x \text{ declared as } x : K\text{)}$$

The following formulas are also valid (using some syntax sugars, cf. Figure 10, Remark 3.1).

$E \in \{E\}$

$x.f = y \Rightarrow x \neq \mathsf{null}$

$x = \mathsf{null} \Rightarrow \{x\}\text{`}f \;\#\; G$

$x = \mathsf{null} \Rightarrow x.f \;\#\; G$ (only well-formed for $f : \mathsf{rgn}$; the consequent abbreviates $\{x\}\text{`}f \;\#\; G$)

$x.f.g = E \Rightarrow x \neq \mathsf{null} \wedge x.f \neq \mathsf{null}$

(the antecedent abbreviates $\exists y : K \in \mathsf{alloc} \cdot x.f = y \wedge y.g = E$, where $f : K$)

$x \in G_1 \wedge G_1\text{`}f \subseteq G_2 \Rightarrow \{x\}\text{`}f \subseteq G_2$ (consequent can be written $x.f \subseteq G_2$ only if $f : \mathsf{rgn}$)

$G \;\#\; (G_1 \cup G_2) \Leftrightarrow G \;\#\; G_1 \wedge G \;\#\; G_2$

$x \in G \Leftrightarrow x \in G/K$ (for x declared as $x : K$)

$G/K \;\#\; G/L$ (if K and L are incomparable classes: $K \not\leq L$ and $L \not\leq K$)

$x \in G_1 \wedge y \in G_2 \wedge G_1 \;\#\; G_2 \Rightarrow x \neq y \vee (x = \mathsf{null} \wedge y = \mathsf{null})$

Method	Pre-condition	Post-condition
$Subject()$	$true$	$\mathsf{self}.val = 0 \wedge \mathsf{self}.O = \varnothing$
$update(n)$	$\forall o \in \mathsf{self}.O \cdot Obs(o, \mathsf{self}, \_)$	$\forall o \in \mathsf{self}.O \cdot Obs(o, \mathsf{self}, n)$
$get()$	$true$	$res = val$
$register(b)$	$b \neq \mathsf{null} \wedge b \notin \mathsf{self}.O$ $\wedge\ b.sub = \mathsf{self} \wedge SubH(\mathsf{self})$ $\wedge\ \forall o \in \mathsf{self}.O \cdot Obs(o, \mathsf{self}, \mathsf{self}.val)$	$b \in \mathsf{self}.O \wedge SubH(\mathsf{self})$ $\wedge\ \forall o \in \mathsf{self}.O \cdot Obs(o, \mathsf{self}, \mathsf{self}.val)$
$Observer(s)$	$\forall o \in s.O \cdot Obs(o, s, s.val)$ $\wedge\ s \neq \mathsf{null} \wedge \mathsf{self} \notin s.O$	$\forall o \in s.O \cdot Obs(o, s, s.val)$ $\wedge\ \mathsf{self} \in s.O$
$notify()$	$\mathsf{self}.sub \neq \mathsf{null}$	$\mathsf{self}.cache = \mathsf{self}.sub.val$

Fig. 11. Specifications for Observer pattern. Methods *register* and *notify* are not part of the public interface.

It is tempting to use $\in$ notation like $x.f \in G$ as sugar for $\{x\}\text{`}f \subseteq G$ but we rarely do because there is a potential confusion. This would be true in case $x$ is null, as $\{x\}\text{`}f$ is empty if x is null. By contrast, the points-to predicate $x.f = E$ is false when x is null.

4.3. Specifications for the Observer Pattern

Inductive Predicates. An important feature of our logic is that inductive predicates can often be avoided, as emphasized in Section 2. Nonetheless, inductive predicates are compatible with the logic. In fact, regions provide a convenient way to define inductive predicates over possibly cyclic structures. The recursive predicate $List(b, r)$ defined here expresses that b points to null-terminated list and that the region r is exactly the set of all nodes of the list. Our running example involves a subject together with its list of Observers. Thus, variable b and field $nxto$ have type $Observer$.

$$List(b : Observer,\ r : \mathsf{rgn}) \ \widehat{=}\ (b = \mathsf{null} \Rightarrow r = \varnothing) \wedge (b \neq \mathsf{null} \Rightarrow b \in r \wedge List(b.nxto, r \setminus \{b\}))$$

Note that "$b.nxto$" is not in the syntax of expressions (Figure 2); we let $List(b.nxto, r \setminus \{b\})$ abbreviate the formula

$$\forall p : Observer \in \{b\}\text{`}nxto \cdot List(p, r \setminus \{b\}).$$

The $List$ predicate involves an explicit region, r, for the "footprint" of the list. The recursion is well founded with respect to the subset order on regions, because regions are finite and because sublists lie in a smaller region, $r \setminus \{b\}$. When using $List$ in a framing judgment we will see (Section 6) that the footprint of $List$ will be explicit. In separation logic, a similar definition of the $List$ predicate would be defined using separating conjunction, with well-foundedness based on another parameter like a mathematical list.

We do not formalize recursively defined predicates; indeed, we do not formalize predicate definitions, though we use them in examples.

The Specifications. To specify the methods of Figure 8 we use the predicates $SubH$ and Obs (adapted from Parkinson [2007]):

$$SubH(s) \ \widehat{=}\ List(s.obs, s.O)$$
$$Obs(b, s, v) \ \widehat{=}\ b.cache = v \wedge b.sub = s$$

$SubH(s)$ is an invariant of $Subject$. It says that all observers of s are in a list whose nodes comprise region $s.O$. The invariant $Obs(b, s, v)$ is an invariant of $Observer$. It says that b is an observer of subject s and that b's view of s's internal state is v.

With these definitions, the method specifications in Figure 11 are mostly self-explanatory. Note the presence of dereference chains, for example, $\mathsf{self}.cache =$

self.$sub.val$, in the postcondition of $notify$. This formula can be rendered in our assertion language as

$$\forall s \in \{\mathsf{self}\}`sub \cdot \forall v : \mathsf{int} \cdot s.val = v \wedge \mathsf{self}.cache = v.$$

We will continue to use sugared versions of field access in the examples and leave desugaring to the reader. The "_" in $Obs(o, \mathsf{self}, \_)$ abbreviates an existential quantification.

The given specifications are suitable for use by clients. As we will see in Part II, the implementations rely on a data invariant that needs to be conjoined to the pre- and postconditions of the public specifications (but not to the specifications of $register$ or $notify$). Parkinson uses a predicate $SubObs$ in the specifications of the public methods. This predicate is a conjunction of the predicates $SubH$ and Obs. We factor out $SubH$ so we can treat it as a hidden invariant (in Part II).

The implementations have been checked using the VERL tool [Rosenberg 2011; Rosenberg et al. 2010]; the software distribution includes example clients. Of course, the specifications need to be augmented with frame conditions, which are given in Section 5.1.

5. EFFECTS

Effects are used in frame conditions and also for framing of formulas. This section presents basic semantic notions concerning effects; these are used throughout both Part I and Part II of the article. In the overview Section 2, frame conditions consist of write effects, and the frame of a formula consists of read effects. Despite these differing uses, we formalize a single syntactic category of effects. There is nothing essential about this design choice and it can be confusing, but it does streamline some formulations. Collapsing both kinds of effects also caters for using read effects for commands, as needed for extensions of the logic to concurrency, for reasoning about program transformations, and for pure method calls and model fields in assertions. The addition of read effects for commands requires only modest changes to the proof rules in Section 7. However, the semantics of read effects for commands is intricate, as we remark here; so to streamline this article, we omit them.

Effects are given by the grammar

$$\varepsilon ::= \varepsilon, \varepsilon \mid (empty) \mid \mathsf{rd}\, x \mid \mathsf{rd}\, G`f \mid \mathsf{wr}\, x \mid \mathsf{wr}\, G`f \mid \mathsf{fr}\, G.$$

The latter five forms are called *atomic effects*, so an effect amounts to a list of atomic effects.[11] Sometimes we abuse notation and treat an effect as a set of atomic effects. Besides ε, we also use identifiers δ and η for effects and tend to use δ to indicate that read effects are of interest.

The idea is that $\mathsf{rd}\, x$ allows variable x to be read, $\mathsf{rd}\, G`f$ allows read of the f field of objects in G, $\mathsf{wr}\, x$ allows update of variable x. For the distinguished region variable, alloc, that holds the set of all allocated references and is automatically updated by the allocator, wr alloc allows allocation and rd alloc allows dependence on the set of allocated objects. The effect $\mathsf{wr}\, G`f$ designates a set of l-values at which updates are to be allowed; these l-values are locations (o, f) where $o \in G$ is an object with field f. For this purpose, we interpret G in the initial state. By contrast, $\mathsf{fr}\, G$ says that all elements of G in the final state are freshly allocated.

As emphasized in Section 2, some effects are state dependent. For example, for field $g : \mathsf{rgn}$, the effect $\mathsf{wr}\, \mathsf{self}.g`f$ permits update of f field of any reference currently in self.g.

[11] Different from other syntactic categories, we do not use overlines to indicate lists of atomic effects. We did so in [Banerjee et al. 2008c]. And there we used the term "effect" for atomic effect and "effect set" for effect.

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

Fig. 12. Effects for the Observer Pattern. For this example, we do not need freshness effects.

In accord with remarks in Section 3.1, an effect of the form wr $x.f$ abbreviates wr $\{x\}$`f (*mutatis mutandis* for rd $x.f$). In case x is null, this is well defined and designates the empty set of locations. In examples, we also allow f to be a data group [Leino 1998], to abstract over concrete fields. In particular, the effect wr self.any says that any fields of self may be written.

A command may have a freshness effect, written fr G. 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. Freshness effects are not essential, because they can be expressed via pre- and postconditions. But they are convenient in reasoning about sequenced commands, to mask updates to fresh objects. For example, consider the sequence $x :=$ new $Comp; x.size := 0$ in using class $Comp$ from Section 2, By itself, the field update $x.size := 0$ has effect wr $\{x\}$`$size$. But in the prestate of the sequence, $\{x\}$ cannot possibly contain the updated object. Indeed, no pre-existing object is updated.

We omit tags wr and rd in lists of effects of the same kind, for example, wr $y, \{y\}$`$f$ abbreviates wr $y$, wr $\{y\}$`f. Note that the elements of the list are separated by a comma as in $y, \{y\}$`f above. In this article, we do not use read effects in frame conditions for commands; instead, a command may read anything. So in frame conditions of commands we drop the wr tag but retain fr. Finally, we use the abbreviation, for example, wr $x.(f, g)$ to denote wr $x.f, x.g$.

5.1. Effects for the Observer Pattern

Figure 12 shows the effects we choose to specify for the methods in the running example. Consider, for example, the effects wr $s.val$ and wr $s.O$`$cache$ of $s.update(n)$. Recall that $s.O$ is a region. The effect wr $s.val$ licenses the write of the $val$ field of $s$, and wr $s.O$`$cache$ licenses the write of the $cache$ field of any object in the set $s.O$. Furthermore, $s.O$ is *stateful*: its interpretation may change from state to state owing to assignment to s and/or $s.O$. However, owing to the absence of effect wr $s.O$ we know that the O field of s is not mutated by $s.update(n)$.

The effect for $notify$ records that a call to it will result in the writing of an observer's $cache$ field. The effect for $register$ records that O was written when an observer b was added to the existing list of observers of a subject. It also takes into account the effect of $notify$. The effect for $update$ records that the subject's val field is updated and also takes into account the effects accrued as a result of calling $notify$. The body of constructor $Subject$ is verified with respect to effects wr self.(obs, val, O). Similarly the body of constructor $Observer$ is verified with respect to effects wr self.$(sub, nxto, cache)$ in addition to the effects mentioned in Figure 12. However, the effects on self for the constructors get masked because we are only concerned with the write effects of objects already allocated in the pre-state of the constructor method, and constructors are only invoked with new. This explains the empty effects of $Subject$ and $Observer$ in Figure 12.

As a matter of interface design, it seems wise to include wr alloc in the effects of most methods, because even if the current implementation does no allocation alternative

implementations might. In a full-fledged concrete syntax, wr alloc should probably be implicitly the default, but that would be a needless complication in this article. For the sake of brevity, our examples only use wr alloc where it is necessary.

5.2. Syntax and Semantics of Effects

Effects must be well formed (wf) for the context Γ in which they occur: rd x and wr x are wf if $x \in dom(\Gamma)$; rd G`$f$, wr $G$`f, and fr G are wf if G is wf in Γ. The empty effect is wf in any context and ε, η is wf if ε and η are. The subeffect rules given later allow effects to be put in a normal form where there is at most one freshness effect, at most one read of given field, etc. Note that in contrast to the typing of region expressions G`$f$, in effects wr $G$`f and rd G`f the field f can have any type.

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 $\sigma(\mathsf{alloc}) \subseteq \sigma'(\mathsf{alloc})$. The semantics has the property that the type of an allocated reference never changes: $\langle C, \sigma\rangle \longmapsto^* \langle C', \sigma'\rangle$ implies $\sigma \hookrightarrow \sigma'$. We adopt notation from Amtoft et al. [2006] in the following.

Definition 5.1 (*Changes Allowed by Write and Freshness Effects*). Let effect ε be well formed in Γ and let σ, σ' be Γ'-states for some $\Gamma' \supseteq \Gamma$. We say ε *allows change from* σ *to* σ', written

$$\sigma \rightarrow \sigma' \models \varepsilon$$

iff $\sigma \hookrightarrow \sigma'$ and the following all hold:

(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 such that o is in $\sigma(G)$
(c) for each G such that fr G is in ε, we have $\sigma'(G) \subseteq \sigma'(\mathsf{alloc}) \setminus \sigma(\mathsf{alloc})$.

In Part I, this definition is only instantiated with $\Gamma' = \Gamma$. The extra generality is only needed in Part II of this article, where it is used in the semantics of procedure calls.

The definition is formulated for effects in general, but it "ignores" read effects. That is, read effects in ε have no bearing on whether $\sigma \rightarrow \sigma' \models \varepsilon$ or not.

Recall that we always assume alloc $\in dom(\Gamma)$, whence by Definition 5.1(a), if $\sigma'(\mathsf{alloc}) \neq \sigma(\mathsf{alloc})$ then wr alloc is in ε. Note also that wr G`f in Definition 5.1(b) refers to writes of f fields of objects in the pre-state, σ. However, fr G in Definition 5.1(c) refers to freshly allocated objects that are present in the post-state, σ', but absent in σ.

Definition 5.2 (*Agreement on Read Effects*). Let ε 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 be a Γ''-state. Say that σ and τ *agree on* ε, written $\mathit{Agree}(\sigma, \tau, \varepsilon)$, provided $\sigma \asymp \sigma'$ and moreover the following hold:

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

For (b), note that $o \in \sigma(G) \cap \tau(\mathsf{alloc})$ implies $o \neq \mathit{null}$.

In Part I, this definition is only instantiated with $\Gamma' = \Gamma'' = \Gamma$. The extra generality is only needed in Part II of this article.

The condition $\sigma \asymp \sigma'$ says the states must agree on types of all objects in common. Usually[12] we consider states such that $\sigma \hookrightarrow \sigma'$, in which case Definition 5.2(b) implies agreement on f for all $o \in \sigma(G)$, because then we have $o \in \sigma'(\mathsf{alloc})$ and $\mathit{Type}(o, \sigma) =$

[12]In fact, throughout Part I and Part II, but not in the relational version of the logic, under development.

Type(o,σ') hence $f \in$ *Fields*(*Type*$(o,\sigma'))$. But even then, it need not be the case that $o \in \sigma'(G)$, for example, in case G is a variable r such that $\sigma(r) \neq \sigma'(r)$. For a given effect, one might expect agreement to be a symmetric relation on states, but that is not the case owing to the interpretation of stateful effects in the first state (i.e., σ in (b)).

In most situations, an effect of interest will be *self-framing* in the following sense: some of the included effects ensure that the others do not change their intepretation. For example, in the framing analysis of Section 6, the effects of assertion $x.f = 0$ are $\mathsf{rd}\, x, \mathsf{rd}\, \{x\}\text{`}f$. Two states could agree on $\mathsf{rd}\, x\text{`}f$ even though they disagree on x and thus on the interpretation of $\{x\}$. But this is disallowed by $\mathsf{rd}\, x$. In the work of Kassios [2011], self-framing is important because frames are expressed using model fields, defined in terms of state variables. We are using explicitly updated ghost state rather than model fields, so for our purposes there is no need to formalize the notion of self-framing effects.

We do not formalize the use of read effects in frame conditions for commands, because their semantics involves additional complications and for our purposes in Part I and Part II they can be omitted. Informally, the semantics of a frame condition with read effects δ is as follows: When the command is run twice, from two initial states that agree on δ and both satisfy the precondition, if both terminate normally then the final states agree on the updatable locations. To make this precise involves the use of renaming relations on references, to allow for differing allocation behavior; agreement then becomes "agree modulo renaming". Moreover case distinctions need to be made for faulting and divergence.

5.3. Subeffects

Effects that are convenient and precise in a local context may not be meaningful in another context. So we often need to subsume an effect by a weaker one. As an example, following Eq. (10) in Section 2 we mentioned that the command $p.size := p.size + x.size$ has precise effect $\mathsf{wr}\, p.size$ that can be subsumed by $\mathsf{wr}\, ancestors(\mathsf{self})\text{`}size$, under the precondition $p \in ancestors(\mathsf{self})$. For an effect of the form $\mathsf{wr}\, G\text{`}f$, there is the possibility of more liberal effect $\mathsf{wr}\, H\text{`}f$ in case $G \subseteq H$.

Since regions can be state-dependent, inclusions like these are state-dependent. So we define the *subeffect judgment* to have the form $P \vdash \varepsilon_1 \leq \varepsilon_2$. It is intended to mean that under precondition P, the "bigger" effect ε_2 is more permissive than ε_1. For example, $p \in ancestors(\mathsf{self}) \vdash \mathsf{wr}\, p.size \leq \mathsf{wr}\, ancestors(\mathsf{self})\text{`}size$.

Definition 5.3 (*Valid Subeffect*). The judgment $P \vdash \varepsilon \leq \eta$ is *valid*, written $P \models \varepsilon \leq \eta$, iff the following hold for all σ, σ' such that $\sigma \models P$:

(valid write and freshness subeffect) $\sigma \rightarrow \sigma' \models \varepsilon$ implies $\sigma \rightarrow \sigma' \models \eta$.
(valid read subeffect) *Agree*(σ, σ', η) implies *Agree*$(\sigma, \sigma', \varepsilon)$.

In Figure 13, we provide syntactic rules for subeffecting. We abbreviate $true \vdash \varepsilon \leq \eta$ by omitting $true$.

LEMMA 5.4 (SUBEFFECT SOUNDNESS). *Suppose* $P \vdash \varepsilon \leq \eta$ *is derivable by rules in Figure 13. Then* $P \models \varepsilon \leq \eta$.

PROOF. By induction on a derivation of $P \vdash \varepsilon \leq \eta$, using that each rule is sound, that is, preserves validity. We consider two representative cases.

(Write Subeffect) $G_1 \subseteq G_2 \vdash \mathsf{wr}\, G_1\text{`}f \leq \mathsf{wr}\, G_2\text{`}f$. Assume $\sigma \rightarrow \sigma' \models \mathsf{wr}\, G_1\text{`}f$ and $\sigma \models G_1 \subseteq G_2$, to show $\sigma \rightarrow \sigma' \models \mathsf{wr}\, G_2\text{`}f$. We have $\sigma \hookrightarrow \sigma'$ and $\sigma(G_1) \subseteq \sigma(G_2)$. Because $\sigma \rightarrow \sigma' \models \mathsf{wr}\, G_1\text{`}f$, for every $o \in \sigma(\mathsf{alloc})$ and every $f \in$ *Fields*(*Type*$(o,\sigma))$, either $\sigma(o.f) = \sigma'(o.f)$ or $o \in \sigma(G_1)$. But if $o \in \sigma(G_1)$, then $o \in \sigma(G_2)$. Hence, $\sigma \rightarrow \sigma' \models \mathsf{wr}\, G_2\text{`}f$

$$G_1 \subseteq G_2 \vdash \mathsf{wr}\ G_1\text{`}f \leq \mathsf{wr}\ G_2\text{`}f \qquad G_1 \subseteq G_2 \vdash \mathsf{rd}\ G_1\text{`}f \leq \mathsf{rd}\ G_2\text{`}f$$

$$\frac{\mathit{DeclClass}(f) = K}{\vdash \mathsf{wr}\ G\text{`}f \leq \mathsf{wr}\,(G/K)\text{`}f} \qquad \frac{\mathit{DeclClass}(f) = K}{\vdash \mathsf{rd}\ G\text{`}f \leq \mathsf{rd}\,(G/K)\text{`}f}$$

$$\vdash \mathsf{wr}\ G_1\text{`}f, G_2\text{`}f \leqslant \mathsf{wr}\,(G_1 \cup G_2)\text{`}f \qquad \vdash \mathsf{rd}\ G_1\text{`}f, G_2\text{`}f \leqslant \mathsf{rd}\,(G_1 \cup G_2)\text{`}f$$

$$\frac{P \vdash \varepsilon_1 \leq \varepsilon_2 \qquad P \vdash \varepsilon_2 \leq \varepsilon_3}{P \vdash \varepsilon_1 \leq \varepsilon_3} \qquad \frac{P' \Rightarrow P \qquad P \vdash \varepsilon \leq \eta}{P' \vdash \varepsilon \leq \eta} \qquad \frac{P \vdash \varepsilon_1 \leq \varepsilon_2}{P \vdash \varepsilon_1, \eta \leq \varepsilon_2, \eta}$$

$$\vdash \varepsilon \leq \varepsilon \qquad \vdash \varepsilon, \eta \leq \eta, \varepsilon \qquad \frac{\eta \text{ is a write or read effect}}{\vdash \varepsilon \leq \varepsilon, \eta} \qquad \vdash \mathsf{fr}\ G, \varepsilon \leq \varepsilon$$

$$\mathit{false} \vdash \varepsilon \leq \eta$$

Fig. 13. Subeffect rules. We write $\leqslant$ to abbreviate two inclusion rules.

because 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 exists $\mathsf{wr}\ G_2\text{`}f$ such that $o \in \sigma(G_2)$.

(Read Subeffect) $G_1 \subseteq G_2 \vdash \mathsf{rd}\ G_1\text{`}f \leq \mathsf{rd}\ G_2\text{`}f$. Assume $\mathit{Agree}(\sigma, \sigma', \mathsf{rd}\ G_2\text{`}f)$ and $\sigma \models G_1 \subseteq G_2$, to show $\mathit{Agree}(\sigma, \sigma', \mathsf{rd}\ G_1\text{`}f)$. Consider any $o \in \sigma(G_1)$ with $\mathit{DeclClass}(f) = \mathit{Type}(o, \sigma)$. Then, $o \in \sigma(G_2)$. Now from $\mathit{Agree}(\sigma, \sigma', \mathsf{rd}\ G_2\text{`}f)$ and $\sigma(G_1) \subseteq \sigma(G_2)$, we have $\sigma(o.f) = \sigma'(o.f)$. Hence, $\mathit{Agree}(\sigma, \sigma', \mathsf{rd}\ G_1\text{`}f)$. □

By contrast with the rules for read and write, we do not have $G_1 \subseteq G_2 \vdash \mathsf{fr}\ G_1 \leq \mathsf{fr}\ G_2$, because the antecedent is interpreted in the pre-state whereas freshness effects refer to regions in the post-state. Reasoning about freshness based on region inclusion is provided by rule FrSub, derived in Section 7.2 using rules in Figure 17.

Earlier we mentioned that data groups could be added, with appropriate adaptation of Definitions 5.1(b) and 5.2(b). For example, Definitions 5.1(b) would allow modification of $o.f$ if there is a write effect $\mathsf{wr}\ G\text{`}d$ where o is in G and f is in the data group d. The only addition to the logic would be the subeffect rule

$$\frac{f \text{ in data group } d}{\vdash G\text{`}f \leq G\text{`}d},$$

and means to declare that a field is in a data group. In examples, we assume that every field is in the data group any.

6. FRAMING AND SEPARATORS

This section formalizes the two key elements of the Frame rule, which is found in Figure 17 together with other rules for program correctness. The first element is the *framing judgment*, $P \vdash^{\Gamma} \delta\ \mathsf{frm}\ Q$. Recall from Section 2 the intended meaning is roughly that Q depends only on the state read according to δ, or δ covers the "footprint" of Q in P-states.

Definition 6.1 (*Frame Validity*). Judgment $P \vdash^{\Gamma} \delta\ \mathsf{frm}\ Q$ is *valid*, written $P \models^{\Gamma} \delta\ \mathsf{frm}\ Q$, iff for all Γ-states σ, σ', if $\mathit{Agree}(\sigma, \sigma', \delta)$ and $\sigma \models^{\Gamma} P \wedge Q$ then $\sigma' \models^{\Gamma} Q$.

Often we elide the context Γ in a framing judgment, as it is usually clear from context.

The other critical element in the Frame rule is the *separator*. This is given by an operator, $\mathbin{/.}$, which is applied to the read effects, say δ, of a formula and the write effects,

$$
\begin{array}{lcl@{\qquad}lcl}
\mathit{ftpt}(x) & = & \mathsf{rd}\, x & \mathit{ftpt}(E = E') & = & \mathit{ftpt}(E), \mathit{ftpt}(E') \\
\mathit{ftpt}(G\text{`}f) & = & \mathsf{rd}\, G\text{`}f, \mathit{ftpt}(G) & \mathit{ftpt}(G_1 \subseteq G_2) & = & \mathit{ftpt}(G_1), \mathit{ftpt}(G_2) \\
\mathit{ftpt}(\varnothing) & = & \varnothing & \mathit{ftpt}(x.f = F) & = & \mathsf{rd}\, x, x.f, \mathit{ftpt}(F) \\
\mathit{ftpt}(\{E\}) & = & \mathit{ftpt}(E) & & & \\
\mathit{ftpt}(G/K) & = & \mathit{ftpt}(G) & & & \\
\mathit{ftpt}(G_1 \odot G_2) & = & \mathit{ftpt}(G_1), \mathit{ftpt}(G_2) \ \text{ for } \odot \text{ in } \{\cup, \cap, \setminus\} & & &
\end{array}
$$

Fig. 14. Footprints of region expressions and atomic assertions (definition of *ftpt*).

ε, of a command. Their separator, $\delta \between \varepsilon$, is a conjunction of region disjointness formulas sufficient to ensure that if the changes from σ to σ' are allowed by the write effects ε, then σ, σ' agree on the read effects δ. This serves to establish the agreement condition required by Definition 6.1. The definition of $\between$ is in Section 6.2.

Frame validity is a first-order condition that is amenable to automated checking by an SMT solver, as we have demonstrated in experiments using VERL [Rosenberg et al. 2010]. More interesting is the inference problem: given P and Q, find δ such that $P \models \delta\ \mathsf{frm}\ Q$. In Section 6.1, we provide a proof system for frame judgments. This has conceptual interest, providing for a comprehensive program logic—though if explicit proof representation is not of interest, one could as well treat frame judgments semantically just as many program logics treat first-order validities. The rules also serve as basis for inference heuristics that are used by the VERL implementation, to minimize the need for user-provided frames even for recursively defined predicates.

6.1. Deductive Framing

We give rules for deriving framing judgments. First, we define the function, *ftpt*, to compute the precise "footprint" of program expressions, region expressions and atomic assertions. The footprint consists of all read effects needed to evaluate a given expression or atomic assertion. Next, we give rules for deriving a framing judgment, $P \vdash \delta\ \mathsf{frm}\ Q$. The section ends with a soundness result for derivable framing judgments. The most interesting aspect is negation.

Footprints of Expressions and Atomic Assertions. For any program expression E, define its read effect, $\mathit{ftpt}(E)$, as follows:

$$\mathit{ftpt}(E) = \mathsf{rd}\, x, \ldots, \mathsf{rd}\, z \quad \text{where } x, \ldots, z \text{ comprise } \mathit{Vars}(E).$$

Figure 14 defines the *ftpt* function for region expressions and for atomic assertions.

As an example, note that $\mathit{ftpt}(x = x)$ is not $\varnothing$ but rather $\mathsf{rd}\, x$. This is in accord with the semantics, in which x is evaluated even though this particular equality expression always denotes true. Framing judgments can be used to more precisely determine the state on which an expression's value depends.

Note that $\mathit{ftpt}(G/K)$ is just $\mathit{ftpt}(G)$; indeed there is no syntax for the effect of reading the type of an object. The semantics has the property that the type of an allocated reference never changes and this is embodied in the definition of agreement (Definition 5.2).

Rules for Derivable Framing Judgments. Figure 15 specifies (mostly) syntax-directed rules for the framing judgment $P \vdash \delta\ \mathsf{frm}\ Q$. The rules ensure frame validity (Lemma 6.5). Rule FRMPROJCTX reflects the semantics (Definition 6.1), which says Q is not falsified, though it may be truthified. This asymmetry complicates the treatment of negation as explained here.

The rules FRMFTPT and FRMFTPTNEG say that $\mathit{ftpt}(P)$ frames P as well as $\neg P$ when P is an atomic assertion. Rule FRMCONJ for framing a conjunction $Q_1 \wedge Q_2$

$$\textsc{FrmFtpt}\ \frac{P \text{ is atomic}}{\mathit{true} \vdash \mathit{ftpt}(P)\ \mathsf{frm}\ P} \qquad \textsc{FrmFtptNeg}\ \frac{P \text{ is atomic}}{\mathit{true} \vdash \mathit{ftpt}(P)\ \mathsf{frm}\ \neg P}$$

$$\textsc{FrmSub}\ \frac{R \vdash \delta\ \mathsf{frm}\ Q \qquad R \vdash \delta \leq \delta' \qquad P \Rightarrow R}{P \vdash \delta'\ \mathsf{frm}\ Q} \qquad \textsc{FrmDisj}\ \frac{P \vdash \delta\ \mathsf{frm}\ Q_1 \qquad P \vdash \delta\ \mathsf{frm}\ Q_2}{P \vdash \delta\ \mathsf{frm}\ Q_1 \vee Q_2}$$

$$\textsc{FrmConj}\ \frac{P \vdash \delta\ \mathsf{frm}\ Q_1 \qquad P \wedge Q_1 \vdash \delta\ \mathsf{frm}\ Q_2}{P \vdash \delta\ \mathsf{frm}\ Q_1 \wedge Q_2} \qquad \textsc{FrmEq}\ \frac{Q_1 \Leftrightarrow Q_2 \qquad P \vdash \delta\ \mathsf{frm}\ Q_1}{P \vdash \delta\ \mathsf{frm}\ Q_2}$$

$$\textsc{FrmProjCtx}\ \frac{P \wedge Q \vdash \delta\ \mathsf{frm}\ Q}{P \vdash \delta\ \mathsf{frm}\ Q} \qquad \textsc{Frm}\forall_{\mathsf{int}}\ \frac{P \vdash^{\Gamma, x:\mathsf{int}} \delta, \mathsf{rd}\, x\ \mathsf{frm}\ Q}{P \vdash^{\Gamma} \delta\ \mathsf{frm}\ \forall x : \mathsf{int} \cdot Q}$$

$$\textsc{Frm}\forall\ \frac{P \vdash \mathit{ftpt}(G) \leq \delta \qquad P \wedge x \in G \vdash^{\Gamma, x:K} \delta, \mathsf{rd}\, x\ \mathsf{frm}\ Q}{P \vdash^{\Gamma} \delta\ \mathsf{frm}\ \forall x : K \in G \cdot Q}$$

Fig. 15. Rules for the framing judgment. Context Γ is elided in rules where the context is the same in every judgment of the rule. Rules $\textsc{Frm}\exists_{\mathsf{int}}$ and $\textsc{Frm}\exists$ are similar to $\textsc{Frm}\forall_{\mathsf{int}}$ and $\textsc{Frm}\forall$; hence elided.

with δ allows Q_1 to be used as hypothesis in showing that δ frames Q_2. This is sound because in a state where Q_1 is false, the conjunction's value is independent of the value of Q_2. The rule is very helpful in subsuming local effects by more global effects. For example, suppose $\delta = \mathsf{rd}\, b, p, r, r\text{`}nxt$ and we wish to establish that the formula $b \in r \wedge p = b.nxt$ is framed by δ. It is clear that $b \in r$ is framed by δ. But $p = b.nxt$ is framed by $\mathsf{rd}\, b, \mathsf{rd}\, b.nxt$, and $\mathsf{rd}\, b.nxt$ is not in δ. However, because of $b \in r$ we have $\mathsf{rd}\, b.nxt \leq \mathsf{rd}\, r\text{`}nxt$ using the second rule in Figure 13. Note that $\wedge$ is commutative—it has standard semantics. Rule FrmConj can be used for either conjunct, owing to rule FrmEq which allows use of a valid equivalence $Q_1 \Leftrightarrow Q_2$. For example, FrmEq allows the empty frame for the predicate $x = x$ even though $\mathit{ftpt}(x = x)$ is $\mathsf{rd}\, x$.

Rule FrmEq is not concerned with framing of an equivalence, but rather use of logical equivalence, so it is a little akin to FrmSub. In FrmEq, it is not sound to weaken $Q_1 \Leftrightarrow Q_2$ to $P \Rightarrow (Q_1 \Leftrightarrow Q_2)$. For example, let $P \mathrel{\widehat{=}} x = y$, $Q_1 \mathrel{\widehat{=}} x.f = \mathsf{null}$, and $Q_2 \mathrel{\widehat{=}} y.f = \mathsf{null}$. Although $x = y \Rightarrow (x.f = \mathsf{null} \Leftrightarrow y.f = \mathsf{null})$ is valid and $x = y \vdash \mathsf{rd}\, x, x.f\ \mathsf{frm}\ x.f = \mathsf{null}$, it is not the case that $x = y \vdash \mathsf{rd}\, x, x.f\ \mathsf{frm}\ y.f = \mathsf{null}$.

Rule FrmFtptNeg says that $\mathit{ftpt}(P)$ frames $\neg P$ when P is an atomic assertion. There is an obvious general rule for negation, along the lines of FrmDisj, but it is not sound. This is discussed further in Remark 6.6. In order to frame $\neg P$ where P is not atomic, one can use the negation normal form, say Q, of $\neg P$. In this form, negation is only applied to atomic assertions. Rule FrmEq says that a frame of Q frames $\neg P$ because $Q \Leftrightarrow \neg P$.

For rule Frm$\forall$, suppose that under assumption P the quantification $\forall x : K \in G \cdot Q$ is to be framed by some δ. By well-formedness, δ cannot mention $\mathsf{rd}\, x$. However, within Q, x might appear by itself, or as $x.f$, $x.g.h$, etc., in which case, Q needs to be framed by $\mathsf{rd}\, x, \mathsf{rd}\, x.f$, $\mathsf{rd}\, x.g.h$ etc. Owing to the additional assumption $x \in G$ in the second premise of the rule, $\mathsf{rd}\, x.f$ can be subsumed by $\mathsf{rd}\, G\text{`}f$, which can appear in δ.

The read effects of the formula $\forall o : \mathit{Comp} \cdot o \neq \mathsf{self} \Rightarrow \mathit{ok}(o)$, given by (9), are obtained as follows. Recall that o ranges over allocated objects. By FrmEq, it is enough to find the read effects of the equivalent formula $\forall o : \mathit{Comp} \in \mathsf{alloc} \setminus \{\mathsf{self}\} \cdot \mathit{ok}(o)$. To

obtain the read effects of $ok(o)$, we must unfold the definition of ok, according to (2). This yields

$$o \neq \mathsf{null} \wedge o.size = 1 + sz(o.lt) + sz(o.rt)$$

The read effect for $o \neq \mathsf{null}$ is $\mathsf{rd}\ o$, by way of FRMFTPTNEG and FRMFTPT. For the second conjunct, we must unfold the definition of sz, according to (1). Then, by FRMFTPT, the read effect is $\mathsf{rd}\ o, \mathsf{rd}\ o.size, \mathsf{rd}\ o.lt, \mathsf{rd}\ o.lt`size, \mathsf{rd}\ o.rt, \mathsf{rd}\ o.rt`size$, which by FRMSUB and FRMCONJ is the read effect of the conjunction. Having thus obtained the read effects for the body of the quantifier, we get the read effects (9) for the whole formula by applying FRM$\forall$.

Our running example involves the recursively defined predicate $List$; such predicates are not addressed by the framing rules given here. But on the basis of semantics one can show the following (omitting antecedent $true$):

$$\vdash \mathsf{rd}\ b, r, r`nxto\ \mathsf{frm}\ List(b, r)$$
$$\vdash \mathsf{rd}\ b, s, v, \{b\}`cache, \{b\}`sub\ \mathsf{frm}\ Obs(b, s, v)$$
$$\vdash \mathsf{rd}\ s.obs, s.O, s.O`nxto\ \mathsf{frm}\ SubH(s)$$

Recall that $SubH(s) = List(s.obs, s.O)$.

LEMMA 6.2 (FOOTPRINT AGREEMENT). *For any states, σ, σ', for any expression F, suppose $Agree(\sigma, \sigma', ftpt(F))$. Then, $\sigma(F) = \sigma'(F)$.*

PROOF. Straightforward structural induction on F. □

Remark 6.3. In Section 2, Eq. (5) defines the ancestors of p to be the set $\{o \in \mathsf{alloc} \mid p \in o.desc\}$. The use of set comprehension in region expressions would also provide an expressive form of conditional effects, for example, a command might write $\{x \mid P(x) \wedge Q\}`f$, where Q is some condition on the initial state, not involving x. One could add comprehensions by extending the grammar with $G ::= \{x : T \in G \mid P\}$, but since formulas (category P) refer to region expressions this would make regions and formulas mutually recursive. As a consequence, we could not use the simple *ftpt* function but would rather have an inductively defined framing relation for expressions just as for formulas. This is not technically difficult.

In the Composite example, most expressions involving $ancestors$ occur in effects. But there is one occurrence in an assertion, namely the loop invariant in procedure add. The formula is $p \neq \mathsf{null} \Rightarrow p \in ancestors(\mathsf{self})$ and it has valid frame $\mathsf{rd}\ p, \mathsf{self}, \mathsf{alloc}`desc$.

Frame Soundness. Now we prove that any derivable framing judgment is valid. Before that, we prove a lemma on agreements under state extension.

LEMMA 6.4. *Let $\sigma_1 = Extend(\sigma, x, o)$ and $\sigma_1' = Extend(\sigma', x, o)$. (For this to be meaningful, we assume x does not occur in states σ, σ' but reference o is allocated.) Then*

(a) $Agree(\sigma, \sigma', \mathsf{rd}\ y)$ implies $Agree(\sigma_1, \sigma_1', \mathsf{rd}\ y)$ for y distinct from x
(b) $Agree(\sigma, \sigma', \mathsf{rd}\ G`f)$ implies $Agree(\sigma_1, \sigma_1', \mathsf{rd}\ G`f)$ for all G, f such that x does not occur in G

PROOF. Part (a) is immediate from definitions. For part (b), consider any G and f with x not in G, and suppose $Agree(\sigma, \sigma', \mathsf{rd}\ G`f)$. Thus, $\sigma \asymp \sigma'$ and since $\asymp$ does not involve variables we have $\sigma_1 \asymp \sigma_1'$. To prove the main condition for $Agree(\sigma_1, \sigma_1', \mathsf{rd}\ G`f)$, suppose $o \in \sigma_1(G)$ and $o \neq \mathsf{null}$ and $f \in Fields(Type(o, \sigma_1))$—to show $\sigma_1(o.f) = \sigma_1'(o.f)$. Because x is not in G, we have $\sigma(G) = \sigma_1(G)$ so $o \in \sigma(G)$ and also

$f \in \mathit{Fields}(\mathit{Type}(o,\sigma))$ because $\sigma_1 = \mathit{Extend}(\sigma, x, o)$. Thus, by $\mathit{Agree}(\sigma,\sigma',\mathsf{rd}\ G\text{`}f)$, we have $\sigma(o.f) = \sigma'(o.f)$ and thus $\sigma_1(o.f) = \sigma(o.f) = \sigma'(o.f) = \sigma_1'(o.f)$. □

LEMMA 6.5 (FRAME SOUNDNESS). *Every derivable framing judgment is valid.*

PROOF. By induction on a derivation of a framing judgment $P \vdash \delta\ \mathsf{frm}\ Q$, using that each rule is sound. We proceed to prove soundness of the rules. (FRMFTPT) and (FRMFTPTNEG) follow immediately by Lemma 6.2, the semantics of the atomic assertions, and the definition of *ftpt*.

(FRMSUB) Suppose $P \vdash \delta'\ \mathsf{frm}\ Q$ because $R \vdash \delta\ \mathsf{frm}\ Q$ and $R \vdash \delta \le \delta'$ and $P \Rightarrow R$. Our assumptions are that $\mathit{Agree}(\sigma,\sigma',\delta')$ and $\sigma \models P \wedge Q$. Hence, $\sigma \models R$. By Lemma 5.4, we obtain $\mathit{Agree}(\sigma,\sigma',\delta)$. Now using $\sigma \models Q \wedge R$, by validity of premises we get $\sigma' \models Q$.

(FRMDISJ) Suppose $\mathit{Agree}(\sigma,\sigma',\delta)$ and $\sigma \models P \wedge (Q_1 \vee Q_2)$. Using $\sigma \models P$, by validity of the premises we have: $\sigma \models Q_1$ implies $\sigma' \models Q_1$ and also $\sigma \models Q_2$ implies $\sigma' \models Q_2$. Hence, $\sigma' \models Q_1 \vee Q_2$.

(FRMCONJ) Suppose $P \models \delta\ \mathsf{frm}\ Q_1 \wedge Q_2$ because $P \models \delta\ \mathsf{frm}\ Q_1$ and $P \wedge Q_1 \models \delta\ \mathsf{frm}\ Q_2$. Assume that $\mathit{Agree}(\sigma,\sigma',\delta)$ and $\sigma \models P \wedge Q_1 \wedge Q_2$. By validity of premises, we obtain $\sigma' \models Q_1$ and $\sigma' \models Q_2$. Hence, $\sigma' \models Q_1 \wedge Q_2$.

(FRMEQ) Assume $Q_1 \Leftrightarrow Q_2$ is valid, $P \vdash \delta\ \mathsf{frm}\ Q_1$, $\sigma \models P \wedge Q_2$, and $\mathit{Agree}(\sigma,\sigma',\delta)$. We have $\sigma \models P \wedge Q_1$ on account of the validity of $Q_1 \Leftrightarrow Q_2$. So by validity of premises, $\sigma' \models Q_1$, which proves $\sigma' \models Q_2$.

(FRMPROJCTX) Immediate from assumptions $\sigma \models P \wedge Q$, $\mathit{Agree}(\sigma,\sigma',\delta)$ and validity of premise.

(FRM$\forall_{\mathsf{int}}$) Assume $P \models \delta, \mathsf{rd}\,x\ \mathsf{frm}\ Q$, $\mathit{Agree}(\sigma,\sigma',\delta)$, $\sigma \models P \wedge \forall x : \mathsf{int} \cdot Q$. By semantics (see Figure 9), $\sigma \models^{\Gamma} \forall x : \mathsf{int} \cdot Q$ iff $\mathit{Extend}(\sigma, x, v) \models^{\Gamma, x:\mathsf{int}} Q$ for all $v \in \mathbb{Z}$. Let $\sigma_1 = \mathit{Extend}(\sigma, x, v)$ and let $\sigma_1' = \mathit{Extend}(\sigma', x, v)$, for some $v \in \mathbb{Z}$. Then it remains to show that $\sigma_1' \models Q$. By well formedness, x does not occur in P, δ, and since $\sigma \models P$ and $\mathit{Agree}(\sigma,\sigma',\delta)$, we obtain $\sigma_1 \models P$ and $\mathit{Agree}(\sigma_1,\sigma_1',\delta)$. Since $\sigma_1(x) = \sigma_1'(x)$, we have that σ_1, σ_1' agree on $\mathsf{rd}\,x$ (using Definition 5.2(1)). Because $\sigma_1 \models P \wedge Q$, we can appeal to the validity of the premise to obtain $\sigma_1' \models Q$.

(FRM$\forall$) We assume validity of the premises, that is:

$$P \models \mathit{ftpt}(G) \le \delta \tag{14}$$

$$P \wedge x \in G \models^{\Gamma, x:K} \delta,\ \mathsf{rd}\,x\ \mathsf{frm}\ Q. \tag{15}$$

To show the conclusion

$$P \models^{\Gamma} \delta\ \mathsf{frm}\ \forall x : K \in G \cdot Q \tag{16}$$

consider any σ, σ' such that $\sigma \models^{\Gamma} P \wedge \forall x : K \in G \cdot Q$ and $\mathit{Agree}(\sigma,\sigma',\delta)$. We must show $\sigma' \models^{\Gamma} \forall x : K \in G \cdot Q$. To that end, we consider arbitrary $p \in \sigma'(G) \setminus \{\mathit{null}\}$ such that $\mathit{Type}(p,\sigma') = K$, and must show

$$\sigma_1' \models^{\Gamma, x:K} Q \quad \text{where } \sigma_1' = \mathit{Extend}(\sigma', x, p) \tag{17}$$

As we are assuming $\mathit{Agree}(\sigma,\sigma',\delta)$, we can use (14) to get $\mathit{Agree}(\sigma,\sigma',\mathit{ftpt}(G))$ (by Lemma 5.4), whence by Lemma 6.2 we get $\sigma(G) = \sigma'(G)$. By definition, $\mathit{Agree}(\sigma,\sigma',\delta)$ implies $\sigma \asymp \sigma'$, hence $\mathit{Type}(p,\sigma) = \mathit{Type}(p,\sigma')$ for all $p \in \sigma(G) \setminus \{\mathit{null}\}$ (noting that $\sigma(G) \setminus \{\mathit{null}\} \cap \sigma'(\mathsf{alloc}) = \sigma(G) \setminus \{\mathit{null}\}$ from $\sigma(G) = \sigma'(G)$). As a consequence we can use our assumption that $\sigma \models^{\Gamma} P \wedge \forall x : K \in G \cdot Q$ to obtain the following.

$$\sigma_1 \models^{\Gamma, x:K} Q \quad \text{where } \sigma_1 = \mathit{Extend}(\sigma, x, p) \tag{18}$$

Now $\sigma_1(G) = \sigma(G)$ because x does not occur in G owing to well formedness of (16). Hence from $\sigma \models^{\Gamma} P$ and preceding considerations about p we get the following.

$$\sigma_1 \models^{\Gamma, x:K} P \wedge x \in G \tag{19}$$

From $\mathit{Agree}(\sigma, \sigma', \delta)$, we get $\mathit{Agree}(\sigma_1, \sigma_1', \delta)$ using that x does not occur in δ owing to well formedness of (16). Hence, $\mathit{Agree}(\sigma_1, \sigma_1', (\delta, \mathsf{rd}\, x))$ because $\sigma_1(x) = \sigma_1'(x)$. So we can instantiate (15) with σ_1, σ_1', whence from (19) and (18) we conclude (17).

The proofs of FRM$\exists_{\mathsf{int}}$ and FRM$\exists$ are similar to their $\forall$ counterparts and elided. □

Remark 6.6. As an alternative to rule FRMFTPTNEG in Figure 15, let us consider this general negation rule FRMNEG: $\dfrac{P \vdash \delta\ \mathsf{frm}\ Q}{P \vdash \delta\ \mathsf{frm}\ \neg Q}$. It is not sound for the semantics given by Definition 6.1. For example, the judgment

$$x = 0 \vdash \mathit{empty}\ \mathsf{frm}\ x = 1 \tag{20}$$

is valid because no state satisfies $x = 0 \wedge x = 1$. However, $x = 0 \vdash \mathit{empty}\ \mathsf{frm}\ x \neq 1$ is not valid: In a state where $x = 0$, it allows setting $x := 1$ that falsifies $x \neq 1$. Judgment (20) does not, however, satisfy this slightly stronger version of validity:

Suppose $P \vdash \delta\ \mathsf{frm}\ Q$ and $\mathit{Agree}(\sigma, \sigma', \delta)$ and $\sigma \models P$. Then $\sigma \models Q$ iff $\sigma' \models Q$.

Let us call this "two-way semantics". It differs from Definition 6.1 by using "iff" instead of an implication, and (20) shows that the two semantics are different.

Rule FRMNEG is sound for two-way semantics. In fact all of the framing rules are sound for two-way semantics with the exception of FRMPROJCTX. (It can be made sound by adding another premise: $P \wedge \neg Q \vdash \delta\ \mathsf{frm}\ Q$.) The advantage of Definition 6.1 is that it is exactly the property needed for the frame rule. (A similar "Frame Property" is fundamental to separation logic [O'Hearn et al. 2009].) An advantage of two-way semantics is that rule FRMNEG allows disjunction and existentials to be treated as syntax sugar (by de Morgan), and framing need not be done via negation normal form.

As for rule FRMPROJCTX, we need it to derive (20), which can be done as follows.

$\mathit{false} \vdash \mathsf{rd}\, x\ \mathsf{frm}\ x = 1$	by FRMFTPT then FRMSUB
$\mathit{false} \vdash \mathit{empty}\ \mathsf{frm}\ x = 1$	by FRMSUB, using $\mathit{false} \vdash \mathsf{rd}\, x \leq \mathit{empty}$
$x = 0 \wedge x = 1 \vdash \mathit{empty}\ \mathsf{frm}\ x = 1$	by FRMSUB, using $x = 0 \wedge x = 1 \Leftrightarrow \mathit{false}$
$x = 0 \vdash \mathit{empty}\ \mathsf{frm}\ x = 1$	by FRMPROJCTX

This derivation is the only use we encountered for the subeffect $\mathit{false} \vdash \mathsf{rd}\, x \leq \mathit{empty}$.

The contrived nature of the example hints that for practical purposes there may be no reason not to use two-way semantics. Both semantics are amenable to direct checking by a first order prover. Indeed, the current version of VERL [Rosenberg et al. 2010] uses two-way semantics. The main reason for that design choice is that, following Dafny, VERL treats framing in terms of expressions of any type, not just predicates. For an expression X of, say, integer type, it can be useful to say that the value of X is preserved by execution of command C. Of course, this can be expressed at the level of predicates, in the one-way semantics, using a specification-only variable y: one takes Q in Definition 6.1 to be the predicate $y = X$. Indeed, this technique can be used to encode two-way semantics in one-way: taking Q in Definition 6.1 to be $y \iff Q$ gives us the two-way semantics for Q, provided that y is not mutable.[13]

[13]Officially, our formulas do not include propositional variables, but one can use, for example, $y > 0 \iff Q$.

6.2. Separators

Given effects δ and ε, we define the *separator* formula $\delta \not\prec\!\!\cdot\ \varepsilon$ to be a conjunction of certain disjointnesses.[14] In a state where $\delta \not\prec\!\!\cdot\ \varepsilon$ holds, nothing that the read effects in δ allow to be read can be written according to the write effects in ε. Note that ε (respectively, δ) may contain read (respectively, write) effects but these do not influence the separator.

Definition 6.7 (*Separator*). Define separators by recursion on the effects:

$$\begin{array}{lll}
\mathsf{rd}\ G_1\text{`}f \not\prec\!\!\cdot\ \mathsf{wr}\ G_2\text{`}g & = & \text{if } f \equiv g \text{ then } G_1 \,\#\, G_2 \text{ else } \mathit{true} \\
\mathsf{rd}\ y \not\prec\!\!\cdot\ \mathsf{wr}\ x & = & \text{if } x \equiv y \text{ then } \mathit{false} \text{ else } \mathit{true} \\
\delta \not\prec\!\!\cdot\ \varepsilon & = & \mathit{true} \quad \text{for all other pairs of atomic effects} \\
\delta \not\prec\!\!\cdot\ \varepsilon & = & \mathit{true} \quad \text{in case } \delta \text{ or } \varepsilon \text{ is empty} \\
(\varepsilon, \delta) \not\prec\!\!\cdot\ \eta & = & (\varepsilon \not\prec\!\!\cdot\ \eta) \wedge (\delta \not\prec\!\!\cdot\ \eta) \\
\delta \not\prec\!\!\cdot\ (\varepsilon, \eta) & = & (\delta \not\prec\!\!\cdot\ \varepsilon) \wedge (\delta \not\prec\!\!\cdot\ \eta).
\end{array}$$

In Section 2, we considered the separator of read effects $\mathsf{rd}\ d, \mathsf{rd}\ d.\mathit{size}$ and write effects $\mathsf{wr}\ c.\mathit{parent}, \mathsf{wr}\ b.\mathit{lt}, \mathsf{wr}\ b.\mathit{rt}, \mathsf{wr}\ \mathit{ancestors}(b)\text{`}\mathit{size}$, which, once the true conjuncts are dropped, is $\{d\} \,\#\, \mathit{ancestors}(b)$.

Recall that in effects we write, for example, $x.f$ as syntax sugar for $\{x\}\text{`}f$. Thus, $\mathsf{rd}\ x.f \not\prec\!\!\cdot\ \mathsf{wr}\ y.f$ desugars to $\mathsf{rd}\ \{x\}\text{`}f \not\prec\!\!\cdot\ \mathsf{wr}\ \{y\}\text{`}f$ which by definition is $\{x\} \,\#\, \{y\}$. It is equivalent to $x \neq y \vee (x = \mathsf{null} \wedge y = \mathsf{null})$.

LEMMA 6.8 (SEPARATOR AGREEMENT). *Let δ and ε be effects that are well-formed in Γ (whence $\delta \not\prec\!\!\cdot\ \varepsilon$ will be well-formed in Γ). Let $\Gamma' \supseteq \Gamma$. Let σ and τ be Γ'-states. Suppose $\sigma \rightarrow \tau \models \varepsilon$ and $\sigma \models \delta \not\prec\!\!\cdot\ \varepsilon$. Then Agree$(\sigma, \tau, \delta)$.*

The use of Γ' in Lemma 6.8 makes it general enough so that it can be used unchanged in Part II of the article. For the purposes of Part I, $\Gamma' = \Gamma$.

PROOF. To show *Agree*(σ, τ, δ), note first that the hypothesis implies $\sigma \hookrightarrow \tau$, hence $\sigma \asymp \tau$. It remains to consider conditions (a) and (b) in Definition 5.2.

(a) Consider any $\mathsf{rd}\ x$ in δ. Since $\sigma \models \delta \not\prec\!\!\cdot\ \varepsilon$, there is no $\mathsf{wr}\ x$ in ε (using the definition of $\delta \not\prec\!\!\cdot\ \varepsilon$). So by $\sigma \rightarrow \tau \models \varepsilon$ we have $\sigma(x) = \tau(x)$. The argument holds even for the case that x is alloc.

(b) Consider any $\mathsf{rd}\ G\text{`}f$ in $\delta$. Consider any $p \in \sigma(G)$; we must show $\sigma(p.f) = \tau(p.f)$. By definition, the formula $\delta \not\prec\!\!\cdot\ \varepsilon$ has a conjunct $G \,\#\, G'$ for any $\mathsf{wr}\ G'\text{`}f$ in ε. Because $\sigma \models \delta \not\prec\!\!\cdot\ \varepsilon$, we get $p \notin \sigma(G')$ for any $\mathsf{wr}\ G'\text{`}f$ in ε, so by $\sigma \rightarrow \tau \models \varepsilon$ we have $\sigma(p.f) = \tau(p.f)$. □

Remark 6.9. In separation logic, the *separating conjunction* $P * Q$ says that P and Q are both true and their truth is supported by disjoint regions of the heap. We can approximate the intuitionistic version that allows there to be objects outside the footprint of P and Q. Suppose $\varepsilon\ \mathsf{frm}\ P$ and $\eta\ \mathsf{frm}\ Q$. Obtain η' from η by discarding reads of variables and replacing each region read $\mathsf{rd}\ G\text{`}f$ by $\mathsf{wr}\ G\text{`}f$. Then, the separation logic formula $P * Q$ says something like $P \wedge Q \wedge (\varepsilon \not\prec\!\!\cdot\ \eta')$. There is a significant difference, however. The semantics of $*$ is that there exists a partition of the heap, and there may be more than one partition in case P and Q do not have unique footprints (are not "precise"). Our use of explicit footprints and ghost variables

[14] In the conference version, we used the $\star$ symbol, to allude to separating conjunction, although our operation pertains to effects not formulas and is not symmetric. Here, we choose an asymmetric symbol that can be written in ascii as %.

can be seen as skolemizing the existential implicit in $*$, since ε and η determine specific sets of locations (asserted to be disjoint by $\varepsilon \not\asymp \eta'$).

For example, let r be a region variable and define $P(r) \mathrel{\widehat{=}} \forall x : Node \in r \cdot x.item \leq 0$ and $Q(r) \mathrel{\widehat{=}} \forall x : Node \in r \cdot x.item \geq 0$. Let $R \mathrel{\widehat{=}} P(r) \wedge Q(\mathsf{alloc} \setminus r)$. Then we have

both $\{R\}\ x := \mathsf{new}\ K; x.n := 0\ \{R\}\ [x, r]$

and $\{R\}\ x := \mathsf{new}\ K; x.n := 0; r := r \cup \{x\}\ \{R\}\ [x, r]$

But the reasoner must choose between these two commands. Issues with nondeterminacy could arise if we allowed bound region variables, for example, $\exists r \cdot P(r) \wedge Q(\mathsf{alloc} \setminus r)$. Precise predicates are discussed further in the related work section of Part II of this article.

6.3. Immunity

Recall from the discussion of Eq. (10) that the proof rule for sequential composition (in Figure 16) must prevent interference between the effects of field updates and assignments. This is done using a notion of immunity that we now formalize.

Definition 6.10 (P/ε-*immune*). Region expression G is said to be *immune from* ε *under* P, or P/ε-immune, iff this formula is valid:

$$P \Rightarrow \mathit{ftpt}(G) \not\asymp \varepsilon$$

Effect η is P/ε-immune provided that for all G, f such that $\mathsf{wr}\ G\text{`}f$ occurs in η, it is the case that G is P/ε-immune.

For example, $\mathsf{wr}\,\mathsf{alloc}\text{`}f$ is $P/\varepsilon$-immune provided $\mathsf{wr}\,\mathsf{alloc}$ is not in $\varepsilon$. Also, $\mathsf{wr}\,x$ is $true/\mathsf{wr}\,x$-immune (vacuously), but $\mathsf{wr}\,\{x\}\text{`}f$ is not $true/\mathsf{wr}\,x$-immune because $\mathit{ftpt}(\{x\}) \not\asymp \mathsf{wr}\,x = \mathit{false}$ by Definition 6.7.

The idea is that if $\{P\}\ C_1\ \{P_1\}[\varepsilon]$ and $\{P_1\}\ C_2\ \{P'\}[\eta]$ are valid, and η is P/ε-immune, then ε, η is a valid effect for the sequence $C_1; C_2$. The proof rule for sequence must also mask writes of fresh objects, as mentioned near the beginning of Section 5.

LEMMA 6.11. *Let G be P/ε-immune. Then, $\sigma(G) = \sigma'(G)$ for any σ, σ' such that $\sigma \rightarrow \sigma' \models \varepsilon$ and $\sigma \models P$.*

PROOF. Since G is P/ε-immune, $P \Rightarrow \mathit{ftpt}(G) \not\asymp \varepsilon$. So by $\sigma \models P$, we have $\sigma \models \mathit{ftpt}(G) \not\asymp \varepsilon$. Then, from $\sigma \rightarrow \sigma' \models \varepsilon$, we have by Lemma 6.8 that $\mathit{Agree}(\sigma, \sigma', \mathit{ftpt}(G))$. Then, by Lemma 6.2, we $\sigma(G) = \sigma'(G)$. □

The following somewhat technical result describes the effects of a succession of states: writes to fresh objects can be dropped and effects can be joined when suitably immune. The result is only used for the soundness proofs of the rules SEQ and WHILE in the sequel.

LEMMA 6.12 (EFFECT TRANSFER). *Suppose the following hold.*

— $\sigma_0 \models P_0$ *and* $\sigma_1 \models P_1$.
— $\sigma_0 \rightarrow \sigma_1 \models \varepsilon_1$.
— $\sigma_1 \rightarrow \sigma_2 \models \varepsilon_2, \mathsf{wr}\ H\text{`}\overline{f}$.
— ε_2 *is* P_0/ε_1-*immune*.
— *for every* G *such that* ε_1 *contains* $\mathsf{fr}\ G$, *we have* G *is* $P_1/(\varepsilon_2, \mathsf{wr}\ H\text{`}\overline{f})$-*immune*.
— $\sigma_1(H) \cap \sigma_0(\mathsf{alloc}) = \varnothing$.

Then $\sigma_0 \rightarrow \sigma_2 \models \varepsilon_1, \varepsilon_2$.

PROOF. For each condition required by Definition 5.1 for the conclusion $\sigma_0 \rightarrow \sigma_2 \models \varepsilon_1, \varepsilon_2$, we show that it follows from the assumptions.

For condition (a) of Definition 5.1, consider any x with $\sigma_0(x) \neq \sigma_2(x)$. We have either $\sigma_0(x) \neq \sigma_1(x)$ or $\sigma_1(x) \neq \sigma_2(x)$ (or both). So $\mathsf{wr}\, x$ is in either ε_1 or ε_2, according to assumptions $\sigma_0 \rightarrow \sigma_1 \models \varepsilon_1$ and $\sigma_1 \rightarrow \sigma_2 \models \varepsilon_2, \mathsf{wr}\, H\text{`}\overline{f}$.

For condition (b) of Definition 5.1, consider any f and any $p \in \sigma_0(\mathsf{alloc})$ with $\sigma_0(p.f) \neq \sigma_2(p.f)$. Then, we have one or both of the following cases.

— $\sigma_0(p.f) \neq \sigma_1(p.f)$. Then, by assumption $\sigma_0 \rightarrow \sigma_1 \models \varepsilon_1$, there is some G with $\mathsf{wr}\, G\text{`}f$ in $\varepsilon_1$ and $p \in \sigma_0(G)$. Then, $\mathsf{wr}\, G\text{`}f$ licenses the update as required by $\sigma_0 \rightarrow \sigma_2 \models \varepsilon_1, \varepsilon_2$.

— $\sigma_1(p.f) \neq \sigma_2(p.f)$. Then, by assumption $\sigma_1 \rightarrow \sigma_2 \models \varepsilon_2, \mathsf{wr}\, H\text{`}\overline{f}$, there are two subcases to consider: If there is $G$ with $\mathsf{wr}\, G\text{`}f$ in ε_2 and $p \in \sigma_1(G)$, then because ε_2 is P_0/ε_1-immune we have by Lemma 6.11 that $\sigma_0(G) = \sigma_1(G)$ (using $\sigma_0 \models P_0$ and $\sigma_0 \rightarrow \sigma_1 \models \varepsilon_1$). So $\mathsf{wr}\, G\text{`}f$ licenses the update as required for $\sigma_0 \rightarrow \sigma_2 \models \varepsilon_1, \varepsilon_2$. The other subcase would be that p is in $\sigma_1(H)$ and f is in the list $\overline{f}$; but by assumption $\sigma_1(H) \cap \sigma_0(\mathsf{alloc}) = \varnothing$, this would imply $p \notin \sigma_0(\mathsf{alloc})$—which contradicts the assumption that $p \in \sigma_0(\mathsf{alloc})$.

For condition (c) of Definition 5.1, first consider any $\mathsf{fr}\, G$ in ε_1. We have

$$
\begin{array}{lll}
& \sigma_2(G) & \\
= & \sigma_1(G) & \text{by Lemma 6.11, as } G \text{ is } P_1/(\varepsilon_2, \mathsf{wr}\, H\text{`}\overline{f})\text{-immune and } \sigma_1 \models P_1 \\
\subseteq & \sigma_1(\mathsf{alloc}) \setminus \sigma_0(\mathsf{alloc}) & \text{by assumption } \sigma_0 \rightarrow \sigma_1 \models \varepsilon_1 \text{ and } \mathsf{fr}\, G \text{ in } \varepsilon_1 \\
\subseteq & \sigma_2(\mathsf{alloc}) \setminus \sigma_0(\mathsf{alloc}) & \text{by } \sigma_1(\mathsf{alloc}) \subseteq \sigma_2(\mathsf{alloc}) \text{ from assum. } \sigma_1 \rightarrow \sigma_2 \models \varepsilon_2, \mathsf{wr}\, H\text{`}\overline{f}
\end{array}
$$

so $\mathsf{fr}\, G$ is justified in $\sigma_0 \rightarrow \sigma_2 \models \varepsilon_1, \varepsilon_2$. Now consider any $\mathsf{fr}\, G$ in ε_2; we have

$$
\begin{array}{lll}
& \sigma_2(G) & \\
\subseteq & \sigma_2(\mathsf{alloc}) \setminus \sigma_1(\mathsf{alloc}) & \text{by } \sigma_1 \rightarrow \sigma_2 \models \varepsilon_2, \mathsf{wr}\, H\text{`}\overline{f} \text{ and } \mathsf{fr}\, G \text{ in } \varepsilon_2 \\
\subseteq & \sigma_2(\mathsf{alloc}) \setminus \sigma_0(\mathsf{alloc}) & \text{by } \sigma_0(\mathsf{alloc}) \subseteq \sigma_1(\mathsf{alloc}) \text{ from assumption } \sigma_0 \rightarrow \sigma_1 \models \varepsilon_1
\end{array}
$$

□

7. PROGRAM CORRECTNESS

A *correctness judgment* takes the form $\vdash^{\Gamma} \{P\}\ C\ \{P'\}\ [\varepsilon]$ and is *well formed* provided that P, P', C, and ε are well formed in Γ. In the sequel, we will only consider well-formed correctness judgments. The intended meaning of the correctness judgment is that from any initial state that satisfies P, C's execution does not *fault*, and if the execution terminates then the final state satisfies P'. Moreover, any changes are allowed by ε (Definition 5.1). We will present a proof system for correctness judgments.

Definition 7.1 (*Derivability*). A correctness judgment $\vdash^{\Gamma} \{P\}\ C\ \{P'\}\ [\varepsilon]$ is *derivable* iff it can be obtained using the proof rules in Figures 16 and 17. In instantiating the proof rules both premises and conclusions must be well formed.

In a correctness judgment, we often omit Γ. In a proof rule, this means all judgments are for the same Γ. In rule VAR, the context is made explicit since the premises context is different from the conclusion's context. Because rules may only be instantiated so that all the judgments are well formed, the x in rule VAR cannot occur in P, P', or ε.

For the substitution rule, in Figure 17, we need the following.

Assumption 7.2. A set $\mathit{SpecOnlyVar} \subseteq \mathit{VarName} \setminus \{\mathsf{alloc}\}$ 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.

The need to disallow $\mathsf{wr}\, x$ only arises in Part II of this article. Given that a specification-only variable x cannot occur in code, the effect $\mathsf{wr}\, x$ would be pointless.

$$\textsc{Alloc}\ \frac{Fields(K) = \overline{f} : \overline{T}}{\vdash^{\Gamma} \{\, true \,\}\ x := \mathsf{new}\ K\ \{\, \mathsf{type}(K, \{x\}) \wedge x.\overline{f} = default(\overline{T}) \,\}\ [x, \mathsf{alloc}, \mathsf{fr}\,\{x\}]}$$

$$\textsc{Assign}\ \frac{y \not\equiv x}{\vdash \{\, x = y \,\}\ x := F\ \{\, x = (F/x{\rightarrow}y) \,\}\ [x]}$$

$$\textsc{FieldAcc}\ \frac{z \not\equiv x}{\vdash \{\, y \neq \mathsf{null} \wedge z = y \,\}\ x := y.f\ \{\, x = z.f \,\}\ [x]}$$

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

$$\textsc{Seq}\ \frac{\begin{array}{c}\vdash \{\, P \,\}\ C_1\ \{\, P_1 \,\}\ [\varepsilon_1, \mathsf{fr}\ G] \qquad \vdash \{\, P_1 \,\}\ C_2\ \{\, P' \,\}\ [\varepsilon_2, G`\overline{f}] \\ \varepsilon_1 \text{ is } \mathsf{fr}\text{-free} \qquad \varepsilon_2 \text{ is } P/\varepsilon_1\text{-immune} \qquad G \text{ is } P_1/(\varepsilon_2, \mathsf{wr}\ G`\overline{f})\text{-immune}\end{array}}{\vdash \{\, P \,\}\ C_1\,;\, C_2\ \{\, P' \,\}\ [\varepsilon_1, \varepsilon_2, \mathsf{fr}\ G]}$$

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

$$\textsc{While}\ \frac{\begin{array}{c}\vdash \{\, P \wedge E \neq 0 \,\}\ C\ \{\, P \,\}\ [\varepsilon, H`\overline{f}] \\ \varepsilon \text{ is } \mathsf{fr}\text{-free} \qquad P \Rightarrow H\ \#\ g \qquad \varepsilon \text{ is } P/\varepsilon\text{-immune} \qquad \mathsf{wr}\ g \notin \varepsilon\end{array}}{\vdash \{\, P \wedge g = \mathsf{alloc} \,\}\ \mathsf{while}\ E\ \mathsf{do}\ C\ \{\, P \wedge E = 0 \,\}\ [\varepsilon]}$$

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

Fig. 16. Correctness rules and axioms for commands.

Note that we still allow such x to occur in effects like $\mathsf{wr}\,\{x\}`f$ and $\mathsf{wr}\,x`f$ (for x of reference and region type, respectively).

Definition 7.3 (*Validity*). A correctness judgment $\vdash^{\Gamma} \{P\}\ C\ \{P'\}\ [\varepsilon]$ is *valid*, written $\models^{\Gamma} \{\, P \,\}\ C\ \{\, P' \,\}\ [\varepsilon]$, if and only if for all Γ-states σ such that $\sigma \models P$ the following hold:

(Safety) It is not the case that $\langle C,\ \sigma\rangle \longmapsto^{*}$ fault.
(Post) $\sigma' \models P'$, for all σ' such that $\langle C,\ \sigma\rangle \longmapsto^{*} \langle \mathsf{skip},\ \sigma'\rangle$
(Effect) $\sigma{\rightarrow}\sigma' \models \varepsilon$, for all σ' such that $\langle C,\ \sigma\rangle \longmapsto^{*} \langle \mathsf{skip},\ \sigma'\rangle$

Instead of "Effect", it is tempting to use the term "Frame", but this might cause confusion because we include freshness conditions here.

Although the assertion language does not include quantification over region variables, Definition 7.3 effectively quantifies (universally) over all variables in scope, that is, in $dom(\Gamma)$. For example, suppose $\Gamma y = K$, $\Gamma x = K$, $\Gamma r = \mathsf{rgn}$, and ε does not include $\mathsf{wr}\ r$. Then, the judgment $\vdash^{\Gamma} \{x \in r\}\ C\ \{y \in r\}\ [\varepsilon]$ says that the final value of y equals the initial value of x, because r ranges over all regions including $\{x\}$.

7.1. Proof Rules

Figure 16 gives the syntax-directed proof rules and axioms for commands while Figure 17 gives structural rules. The axioms for variable assignment, field access, field update and allocation are "small" [O'Hearn et al. 2001] in the sense that they describe

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

$$\textsc{SubEff}\ \frac{\vdash \{\,P\,\}\ C\ \{\,P'\,\}\,[\varepsilon] \qquad P \vdash \varepsilon \leq \varepsilon'}{\vdash \{\,P\,\}\ C\ \{\,P'\,\}\,[\varepsilon']} \qquad\qquad \textsc{ExtendCtx}\ \frac{\vdash^{\Gamma} \{\,P\,\}\ C\ \{\,P'\,\}\,[\varepsilon]}{\vdash^{\Gamma,x:T} \{\,P\,\}\ C\ \{\,P'\,\}\,[\varepsilon]}$$

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

$$\textsc{Conseq}\ \frac{\vdash \{\,P_1\,\}\ C\ \{\,P_1'\,\}\,[\varepsilon] \qquad P_2 \Rightarrow P_1 \qquad P_1' \Rightarrow P_2'}{\vdash \{\,P_2\,\}\ C\ \{\,P_2'\,\}\,[\varepsilon]}$$

$$\textsc{Subst}\ \frac{\vdash \{\,P\,\}\ C\ \{\,P'\,\}\,[\varepsilon] \qquad (P/x{\to}F) \Rightarrow \mathit{ftpt}(F)\ \not\cdot\ (\varepsilon/x{\to}F) \qquad x \text{ is specification-only}}{\vdash \{\,P/x{\to}F\,\}\ C\ \{\,P'/x{\to}F\,\}\,[\varepsilon/x{\to}F]}$$

$$\textsc{Exist}\ \frac{\vdash^{\Gamma,x:K} \{\,x \in G \wedge P\,\}\ C\ \{\,P'\,\}\,[\varepsilon]}{\vdash^{\Gamma} \{\,\exists x:K \in G \cdot P\,\}\ C\ \{\,P'\,\}\,[\varepsilon]} \qquad\qquad \textsc{ExistRegion}\ \frac{\vdash^{\Gamma,x:\mathsf{rgn}} \{\,x = F \wedge P\,\}\ C\ \{\,P'\,\}\,[\varepsilon]}{\vdash^{\Gamma} \{\,P/x{\to}F\,\}\ C\ \{\,P'\,\}\,[\varepsilon]}$$

$$\textsc{PostToFr}\ \frac{\vdash \{\,P\,\}\ C\ \{\,P'\,\}\,[\varepsilon] \qquad P \Rightarrow r = \mathsf{alloc} \qquad P' \Rightarrow G \cap (r \cup \{\mathsf{null}\}) = \varnothing \qquad \mathsf{rd}\ r\ \not\cdot\ \varepsilon}{\vdash \{\,P\,\}\ C\ \{\,P'\,\}\,[\varepsilon, \mathsf{fr}\ G]}$$

$$\textsc{FrToPost}\ \frac{\vdash \{\,P\,\}\ C\ \{\,P'\,\}\,[\varepsilon, \mathsf{fr}\ G] \qquad \mathsf{rd}\ r\ \not\cdot\ \varepsilon}{\vdash \{\,P\,\}\ C\ \{\,P' \wedge G \cap r = \varnothing\,\}\,[\varepsilon, \mathsf{fr}\ G]}$$

$$\textsc{VarMask}\ \frac{\vdash \{\,P\,\}\ C\ \{\,P'\,\}\,[x, \varepsilon] \qquad P \vee P' \Rightarrow x = y \qquad \mathsf{rd}\ y\ \not\cdot\ x, \varepsilon}{\vdash \{\,P\,\}\ C\ \{\,P'\,\}\,[\varepsilon]}$$

$$\textsc{FieldMask}\ \frac{\vdash \{\,P\,\}\ C\ \{\,P'\,\}\,[\{x\}\text{`}f, \varepsilon] \qquad P \vee P' \Rightarrow x.f = y \qquad \mathsf{rd}\ x\ \not\cdot\ \varepsilon \qquad \mathsf{rd}\ y\ \not\cdot\ \varepsilon}{\vdash \{\,P\,\}\ C\ \{\,P'\,\}\,[\varepsilon]}$$

Fig. 17. Structural rules.

the local effect only. For example, the effect of rule FIELDUPD licenses the update of field f of the object referenced by x. The effect of rule ALLOC accounts for a newly allocated object. Small axioms are elegant and admit simple proofs of soundness.

In rule SEQ, the effect $\mathsf{fr}\ G$ ensures that elements of G are allocated during C_1's execution, so their updates by C_2 can be ignored. The condition ε_2 is P/ε_1-immune is necessary for composing the effects of C_1 and C_2 (see the discussion in Section 6.3). As an example to show that the condition "G is $P_1/(\varepsilon_2, \mathsf{wr}\ H\text{`}\overline{f})$-immune" is necessary in rule SEQ, let *rep* be a field of type rgn. Take $G$ to be $x.rep$, $P_1$ to be $y \in x.rep$ and $C_2$ to be $y.rep := \varnothing$. The effect of $C_2$ is $\mathsf{wr}\ y.rep$, which using rule SUBEFF can be subsumed to $\mathsf{wr}\ x.rep\text{`}rep$. Now instantiate ε_2 as $\varnothing$, and $\overline{f}$ as the single field *rep*. Note that $x.rep$ is not immune from $\mathsf{wr}\ x.rep\text{`}rep$, since the separator $\mathsf{rd}\ x.rep\ \not\cdot\ \mathsf{wr}\ x.rep\text{`}rep$ is $\{x\}\ \#\ x.rep$, which is not implied by P_1.

In rule WHILE, P is the loop invariant. The effect $\mathsf{wr}\ H\text{`}\overline{f}$ accounts for writes to fields of freshly allocated objects in the loop body.

The condition $P \Rightarrow H\ \#\ g$, where g snapshots the set of allocated references in the pre-state of the loop, states that objects in H did not exist in the pre-state of the loop. Like rule SEQ, rule WHILE may seem insufficiently general, in that it requires use of g. But if there are no writes to fresh objects, the list $\overline{f}$ can be taken to be empty, and g can be eliminated using rules EXIST and CONSEQ. See the derived rule SIMPLESEQ in the sequel. Examples with loops and allocation appear in Section 8.

We now discuss some of the structural rules (Figure 17). Rule FRAME has the form discussed in Section 2. Here, we comment on the antecedents in the side conditions. As an example, let

$$Q \;\widehat{=}\; r\text{`}f \subseteq r \wedge r \subseteq s \wedge x = y \qquad \text{and} \qquad P \;\widehat{=}\; x \neq \mathsf{null} \wedge y \notin s.$$

The judgment $\vdash \{\, P \,\}\; x.f := x \;\{\, true \,\}\; [x.f]$ is valid and easily proved. The judgment $\vdash \{\, P \wedge Q \,\}\, x.f := x \;\{\, Q \,\}\; [x.f]$ is also valid and we would like to prove it just by Frame. Let $\delta \;\widehat{=}\; \mathsf{rd}\, r, s, x, y, s\text{`}f$. We can derive $Q \vdash \delta\ \mathsf{frm}\ Q$, using framing rule FRMSUB with subeffect $r \subseteq s \vdash \mathsf{rd}\, r\text{`}f \leq \mathsf{rd}\, s\text{`}f$; then FRMPROJCTX and FRMSUB yield $P \vdash \delta\ \mathsf{frm}\ Q$. We also have $\models P \wedge Q \Rightarrow \delta\ \not\cdot\ x.f$, because $\delta\ \not\cdot\ \mathsf{wr}\, x.f$ is equivalent to $s \,\#\, \{x\}$ which is equivalent to $x \in s \Rightarrow x = \mathsf{null}$. But it is not the case that $\models P \Rightarrow \delta\ \not\cdot\ \mathsf{wr}\, x.f$, which shows the need for Q in the antecedent.

Rule SUBEFF loosens the effect clause. It can be used to weaken a specification to get immunity for rule SEQ and to get the two premises in IF or in CONJ to match up.

Rule EXIST is typical in Hoare logics. Some authors prefer an existential elimination rule that also quantifies the postcondition, but this can be achieved by using CONSEQ first, by $P \Rightarrow \exists x : T \cdot P$. The rule is stated for x of class type K; there is a similar rule for $x : \mathsf{int}$ but without the bounding region G. Rule EXISTREGION is so named because it embodies the idiomatic use of EXIST, namely in combination with CONSEQ to eliminate a variable. It is needed only because we eschew quantified variables of region type. Owing to the hygiene property of derivable typing judgments, no variable in F can be bound in P, so there is no issue of capture in the substitution. Owing to well-formedness of the conclusion, x does not occur in C, P', ε, or F. Similarly, in EXIST x does not occur in C, P', or ε.

Rule EXTENDCTX adds a variable to the state space. Because the premise must be well formed, x does not occur free in it. Because the consequent must be well formed, x does not occur bound in P, P', or C—recall Lemma 3.2 and the related remarks about admissibility of context extension for formulas.

For Rule SUBST, recall that we write $P/x{\rightarrow}F$ for substitution of F for x in P. In accord with our convention on well-formed rule instantiations, the result of substitution must be well formed here, for example, $(x.f = y)/x{\rightarrow}\mathsf{null}$ is not. Moreover, because the consequent must be well formed and substitution is not defined if capture would occur, no variable bound in P is free in F. Note that the definition of substitution distributes over the atomic effects of an effect ε, changing any occurrences of x in region expressions. There is no useful reason for ε to contain $\mathsf{wr}\, x$, but no harm comes of it. (In such cases, the conclusion judgment would be ill-formed unless F is just a variable.)

The rules POSTTOFR and FRTOPOST manipulate freshness effects. The first rule snapshots the allocated references in the prestate of C in variable r; this variable is not written by C. The command's post-condition implies that all fresh objects may be allocated in region G—the fact that G and r are disjoint accounts for freshness. Thus, it is sound to add the effect $\mathsf{fr}\, G$ to C's effects.

Rule FRTOPOST is the reverse of POSTTOFR. If $\mathsf{fr}\, G$ is in the effects of C then fresh objects must have been allocated in G. Hence G is disjoint from the set of all references that were allocated in C's prestate and snapshotted by r; hence, it is sound to conjoin $G \cap r = \varnothing$ to the post-condition P'.

Rules VARMASK and FIELDMASK drop write effects. In VARMASK, the separator $\mathsf{rd}\, y\ \not\cdot\ \mathsf{wr}\, x, \varepsilon$ expresses that y is not written and is distinct from x, so from validity of $P \vee P' \Rightarrow x = y$ we have that x is initially and finally equal to the value of y, which is unchanged. These rules are needed to prove, for example, $\{\, true \,\}\; x := x \;\{\, true \,\}\; []$ and $\{\, x.f = 0 \,\}\; x.f := 0 \;\{\, true \,\}\; []$ which have empty effects.

7.2. Derived Rules

Various derived rules are important for constructing proofs by hand. To prove completeness of the logic or for use in an automated verifier, weakest-precondition or strongest-postcondition formulations are needed; but this is beyond the scope of this article.

Using SUBST, one can derive the following rule from ASSIGN.

$$\textsc{SimpleAssign}\ \frac{x \notin \mathit{Vars}(F)}{\vdash \{\, \mathit{true} \,\}\ x := F\ \{\, x = F \,\}\ [x]}$$

For field access, using SUBST with y for z in rule FIELDACC, we get the following.

$$\textsc{SimpleFieldAccess}\ \frac{x \not\equiv y}{\vdash \{\, y \neq \mathsf{null} \,\}\ x := y.f\ \{\, x = y.f \,\}\ [x]}$$

The backwards assignment axiom can also be derived.

$$\textsc{BackwardsAssign}\ \vdash \{\, P/x{\rightarrow}E \,\}\ x := E\ \{\, P \,\}\ [x]$$

Given any P, x, E, choose variable x' distinct from x and let $E' \equiv E/x{\rightarrow}x'$ so $x \notin \mathit{Vars}(E')$. By ASSIGN, we have $\vdash \{\, x = x' \,\}\ x := E\ \{\, x = E' \,\}\ [x]$. Now x does not occur in $P/x{\rightarrow}E'$ so we can use it with FRAME to get

$$\vdash \{\, P/x{\rightarrow}E' \wedge x = x' \,\}\ x := E\ \{\, P/x{\rightarrow}E' \wedge x = E' \,\}\ [x]$$

By predicate calculus (equality substitution), $P/x{\rightarrow}E' \wedge x = x'$ is equivalent to $P/x{\rightarrow}E \wedge x = x'$ and also $P/x{\rightarrow}E' \wedge x = E'$ is equivalent to $P \wedge x = E'$. So by CONSEQ, we get $\vdash \{\, P/x{\rightarrow}E \wedge x = x' \,\}\ x := E\ \{\, P \wedge x = E' \,\}\ [x]$, whence by CONSEQ again we get

$$\vdash \{\, P/x{\rightarrow}E \wedge x = x' \,\}\ x := E\ \{\, P \,\}\ [x]\,.$$

Now x' doesn't occur in the command or postcondition, so by EXIST, we get

$$\vdash \{\, (\exists x' \cdot P/x{\rightarrow}E \wedge x = x') \,\}\ x := E\ \{\, P \,\}\ [x]$$

and by the one-point rule of predicate calculus, using that x' doesn't occur in $P/x{\rightarrow}E$, this precondition is equivalent to $P/x{\rightarrow}E$. So a final use of CONSEQ yields BACKWARDSASSIGN.

The following expresses freshness in terms of an arbitrary region variable r in scope. Note that r is not written (it is distinct from x for reasons of typing). So we can derive from ALLOC the axiom

$$\textsc{Alloc2}\ \frac{\mathit{Fields}(K) = \overline{f} : \overline{T}}{\vdash \{\, \mathit{true} \,\}\ x := \mathsf{new}\ K\ \{\, x \notin r \wedge \mathsf{type}(K, \{x\}) \wedge x.\overline{f} = \mathit{default}(\overline{T}) \,\}\ [x, \mathsf{alloc}]}\,,$$

using FRTOPOST, CONSEQ, and SUBEFF.

Here is a sequence rule for cases where there are no writes to fresh objects.

$$\textsc{SimpleSeq}\ \frac{\vdash \{\, P \,\}\ C_1\ \{\, P_1 \,\}\ [\varepsilon_1] \qquad \vdash \{\, P_1 \,\}\ C_2\ \{\, P' \,\}\ [\varepsilon_2] \qquad \varepsilon_1 \text{ is } \mathsf{fr}\text{-free} \qquad \varepsilon_2 \text{ is } P/\varepsilon_1\text{-immune}}{\vdash \{\, P \,\}\ C_1\,;\, C_2\ \{\, P' \,\}\ [\varepsilon_1, \varepsilon_2]}$$

To derive it, we begin with premise $\vdash^{\Gamma} \{\, P \,\}\ C_1\ \{\, P_1 \,\}\ [\varepsilon_1]$ where we make the typing context explicit. Choosing fresh r, we obtain

$$\vdash^{\Gamma, r:\mathsf{rgn}} \{\, P \wedge r = \mathsf{alloc} \,\}\ C_1\ \{\, P_1 \,\}\ [\varepsilon_1]$$

using EXTENDCTX and CONSEQ. Now, by POSTTOFR (with $G := \varnothing$), we get

$$\vdash^{\Gamma, r:\mathsf{rgn}} \{\, P \wedge r = \mathsf{alloc} \,\}\ C_1\ \{\, P_1 \,\}\ [\varepsilon_1, \mathsf{fr}\,\varnothing]\,.$$

From premise $\vdash^{\Gamma}\ \{\,P_1\,\}\ C_2\ \{\,P'\,\}\ [\varepsilon_2]$ we get $\vdash^{\Gamma,r:\mathsf{rgn}}\ \{\,P_1\,\}\ C_2\ \{\,P'\,\}\ [\varepsilon_2]$ by ExtendCtx. To combine these judgments, we use Seq instantiated with $G := \varnothing$ and $\overline{f}$ the empty sequence, noting that $\varnothing$ is P_1/ε_2-immune. This yields

$$\vdash^{\Gamma,r:\mathsf{rgn}}\ \{\,P \wedge r = \mathsf{alloc}\,\}\ C_1\,;\,C_2\ \{\,P'\,\}\ [\varepsilon_1, \varepsilon_2, \mathsf{fr}\,\varnothing]\,.$$

We obtain the conclusion of SimpleSeq using ExistRegion (with $F := \mathsf{alloc}$) to drop r, along with Conseq and SubEff (the latter using $\vdash\ \varepsilon_1, \varepsilon_2, \mathsf{fr}\,\varnothing\ \leq\ \varepsilon_1, \varepsilon_2$, see Figure 13).

From the freshness rules, one can derive

$$\textsc{FrSub}\ \frac{\vdash\ \{\,P\,\}\ C\ \{\,P'\,\}\ [\varepsilon, \mathsf{fr}\,G] \qquad P' \Rightarrow G' \subseteq G}{\vdash\ \{\,P\,\}\ C\ \{\,P'\,\}\ [\varepsilon, \mathsf{fr}\,G']}$$

as follows. Assume $\vdash^{\Gamma}\ \{\,P\,\}\ C\ \{\,P'\,\}\ [\varepsilon, \mathsf{fr}\,G]$ and $P' \Rightarrow G' \subseteq G$. Choose fresh variable r not in $dom(\Gamma)$ and so by ExtendCtx we have $\vdash^{\Gamma,r:\mathsf{rgn}}\ \{\,P\,\}\ C\ \{\,P'\,\}\ [\varepsilon, \mathsf{fr}\,G]$ and then by Conseq

$$\vdash^{\Gamma,r:\mathsf{rgn}}\ \{\,P \wedge r = \mathsf{alloc}\,\}\ C\ \{\,P'\,\}\ [\varepsilon, \mathsf{fr}\,G]\,.$$

By well formedness of the assumption, r does not occur anywhere, so $\mathsf{rd}\,r\ \not\cdot\ \varepsilon$ is true. Thus, by FrToPost, we get $\vdash^{\Gamma,r:\mathsf{rgn}}\ \{\,P \wedge r = \mathsf{alloc}\,\}\ C\ \{\,P' \wedge G \cap r = \varnothing\,\}\ [\varepsilon, \mathsf{fr}\,G]$. Now apply SubEff, using $\vdash \mathsf{fr}\,G, \varepsilon \leq \varepsilon$, to get $\vdash^{\Gamma,r:\mathsf{rgn}}\ \{\,P \wedge r = \mathsf{alloc}\,\}\ C\ \{\,P' \wedge G \cap r = \varnothing\,\}\ [\varepsilon]$. Because $P' \Rightarrow G' \subseteq G$, we get by Conseq

$$\vdash^{\Gamma,r:\mathsf{rgn}}\ \{\,P \wedge r = \mathsf{alloc}\,\}\ C\ \{\,P' \wedge G' \cap r = \varnothing\,\}\ [\varepsilon]\,.$$

So, by PostToFr, we get $\vdash^{\Gamma,r:\mathsf{rgn}}\ \{\,P \wedge r = \mathsf{alloc}\,\}\ C\ \{\,P' \wedge G' \cap r = \varnothing\,\}\ [\varepsilon, \mathsf{fr}\,G']$ whence by another use of Conseq we get $\vdash^{\Gamma,r:\mathsf{rgn}}\ \{\,P \wedge r = \mathsf{alloc}\,\}\ C\ \{\,P'\,\}\ [\varepsilon, \mathsf{fr}\,G']$. Finally, by ExistRegion, we obtain the conclusion of FrSub.

In a similar manner, one can also show

$$\textsc{FrUnion}\ \frac{\vdash\ \{\,P\,\}\ C\ \{\,P'\,\}\ [\varepsilon, \mathsf{fr}\,G_1, \mathsf{fr}\,G_2]}{\vdash\ \{\,P\,\}\ C\ \{\,P'\,\}\ [\varepsilon, \mathsf{fr}\,(G_1 \cup G_2)]}\,.$$

7.3. Soundness

Theorem 7.4. *Any judgment* $\vdash^{\Gamma}\ \{P\}\ C\ \{P'\}\ [\varepsilon]$ *that is derivable by the rules in Figures 16 and 17, as well as the rules in Figures 13 and 15, is valid.*

Proof. By induction on the derivation and by cases on the last rule used. In each case, we show the proof rule is sound, that is, derives valid conclusions from valid premises (if it has premises) when its side conditions hold. The case of Frame also uses frame soundness (Lemma 6.5), the case of SubEff also uses subeffect soundness (Lemma 5.4), and many cases rely on validities such as $\mathsf{null} \notin \mathsf{alloc}$, cf. Remark 4.1.

For each rule in Figures 16 or 17, we show the three properties in Definition 7.3 for its conclusion, assuming the side conditions and validity of the premises.

(Alloc). Here $C\ \widehat{=}\ x := \mathsf{new}\ K$. It is easy to see that Safety holds by semantics: $\langle C,\ \sigma\rangle \longmapsto \langle \mathsf{skip},\ \sigma'\rangle$ where, with $o \in \mathit{Fresh}(\sigma)$, $\sigma'\ \widehat{=}\ [\sigma_1 \mid x : o]$ and $\sigma_1 = \mathit{New}(\sigma, o, K, \mathit{default}(\overline{T}))$ (as in Figure 7). So it is not the case that $\langle C,\ \sigma\rangle \longmapsto^{*} \mathit{fault}$.

The Post condition to show is $\sigma' \models x\,\mathsf{is}\,K \wedge x.\overline{f} = \mathit{default}(\overline{T})$. Both are immediate from the semantics: $\sigma'(x) = o$, $\mathit{Type}(o, \sigma') = K$, and $\sigma'(o.\overline{f}) = \mathit{default}(\overline{T})$.

Finally, we show Effect. The only variables that are updated are x and alloc. From Figure 5, elements of $\mathit{Fresh}(\sigma)$ are nonnull and not in $\sigma(\mathsf{alloc})$; by semantics, o is in $\sigma'(\mathsf{alloc})$. So we have $\sigma{\rightarrow}\sigma' \models \mathsf{wr}\, x, \mathsf{alloc}, \mathsf{fr}\, \{x\}$.

(ASSIGN). Here, $C \mathrel{\widehat{=}} x := F$. Consider any σ such that $\sigma \models x = y$. So $\sigma(x) = \sigma(y)$. By semantics, the only transition is $\langle x := F,\, \sigma\rangle \longmapsto \langle \mathsf{skip},\, \sigma'\rangle$ where $\sigma' = [\sigma \mid x : \sigma(F)]$. Safety is immediate. For Effect, we appeal directly to Definition 5.1(a). For Post, we must show $\sigma' \models x = F/x{\rightarrow}y$. Note that $\sigma'(y) = \sigma(y)$ by Effect and $y \not\equiv x$, hence $\sigma'(y) = \sigma(y) = \sigma(x)$. Observe

$$\begin{aligned}
& \sigma'(F/x{\rightarrow}y) \\
= \; & [\sigma' \mid x : \sigma'(y)]\,(F) && \text{by substitution lemma Eq. (13)} \\
= \; & [\sigma' \mid x : \sigma(x)]\,(F) && \text{by } \sigma'(y) = \sigma(x) \text{ noted above} \\
= \; & \sigma(F) && \text{by def } \sigma' \text{ and def } [- \mid - : -] \\
= \; & \sigma'(x) && \text{by def } \sigma'
\end{aligned}$$

(FIELDUPD). Here, $C \mathrel{\widehat{=}} x.f := F$. Suppose $\sigma \models x \neq \mathsf{null} \wedge y = F$. Then $\sigma(x) = o$ for some o and $o \neq \mathsf{null}$. By semantics, it is not the case that $\langle C,\, \sigma\rangle \longmapsto \mathit{fault}$, establishing Safety. Instead $\langle C,\, \sigma\rangle \longmapsto \langle \mathsf{skip},\, \sigma'\rangle$ where $\sigma' \mathrel{\widehat{=}} [\sigma \mid o.f : \sigma(F)]$. To establish Post we must show $\sigma' \models x.f = y$.

By semantics of assertions (Figure 9), $\sigma' \models x.f = y$ iff $\sigma'(x) \neq \mathsf{null}$ and $\sigma'(x.f) = \sigma'(y)$. Because neither x nor y is modified by the field update, this is equivalent to $\sigma(x) \neq \mathsf{null}$ and $\sigma(x.f) = \sigma(y)$. We get $\sigma(x) \neq \mathsf{null}$ from $\sigma \models x \neq \mathsf{null}$. And we get $\sigma(x.f) = \sigma(y)$ from $\sigma \models y = F$ and the definition of $[\sigma \mid x.f : \sigma(F)]$, which establishes Post.

Finally, to show Effect, we must show $\sigma{\rightarrow}\sigma' \models \mathsf{wr}\, x.f$. For this we appeal directly to Definition 5.1(b).

(SEQ). Here, $C \mathrel{\widehat{=}} C_1; C_2$. Consider any σ such that $\sigma \models P$.

For Safety, observe that if $\langle C_1; C_2,\, \sigma\rangle \longmapsto^* \mathit{fault}$ then by semantics either $\langle C_1,\, \sigma\rangle \longmapsto^* \mathit{fault}$ or there is σ_1 with $\langle C_1,\, \sigma\rangle \longmapsto^* \langle \mathsf{skip},\, \sigma_1\rangle$ and $\langle C_2,\, \sigma_1\rangle \longmapsto^* \mathit{fault}$. The first is ruled out by validity of premise $\vdash \{\, P \,\}\; C_1 \;\{\, P_1 \,\}\; [\varepsilon_1, \mathsf{fr}\, G]$ and $\sigma \models P$. The second is ruled out because by the first premise we have $\sigma_1 \models P_1$ and then the second premise disallows fault.

For Post, suppose we have $\langle C_1; C_2,\, \sigma\rangle \longmapsto^* \langle \mathsf{skip},\, \sigma'\rangle$ for some σ'. By semantics, there is some σ_1 with $\langle C_1,\, \sigma\rangle \longmapsto^* \langle \mathsf{skip},\, \sigma_1\rangle$ and $\langle C_2,\, \sigma_1\rangle \longmapsto^* \langle \mathsf{skip},\, \sigma'\rangle$. By the first premise, we have $\sigma_1 \models P_1$, so, by the second premise, we get $\sigma' \models P'$.

For Effect, we must show $\sigma{\rightarrow}\sigma' \models \varepsilon_1, \varepsilon_2, \mathsf{fr}\, G$; this can be obtained by instantiating Lemma 6.12 with $H := G, \sigma_0 := \sigma, \sigma_2 := \sigma', \varepsilon_1 := (\varepsilon_1, \mathsf{fr}\, G)$ and $P_0 := P$; so it suffices to check these conditions required by the Lemma.

— $\sigma \models P$ and $\sigma_1 \models P_1$ from the preceding paragraphs.
— $\sigma{\rightarrow}\sigma_1 \models \varepsilon_1, \mathsf{fr}\, G$, which holds by validity of the first premise of SEQ.
— $\sigma_1{\rightarrow}\sigma' \models \varepsilon_2, \mathsf{wr}\, G\overline{`f}$, which holds by the second premise.
— ε_2 is $P/(\varepsilon_1, \mathsf{fr}\, G)$-immune, which amounts to the condition in rule SEQ that ε_2 is P/ε_1-immune.
— G is $P_1/(\varepsilon_2, \mathsf{wr}\, G\overline{`f})$-immune, which is a condition in SEQ (note that ε_1 is fr-free)
— $\sigma_1(G) \cap \sigma(\mathsf{alloc}) = \varnothing$. For this, note that by $\sigma{\rightarrow}\sigma_1 \models \varepsilon_1, \mathsf{fr}\, G$, we have $\sigma_1(G) \subseteq \sigma_1(\mathsf{alloc}) \setminus \sigma(\mathsf{alloc})$. So the intersection is empty.

(WHILE). Here C in the theorem takes the form $\mathsf{while}\ E\ \mathsf{do}\ C$. For brevity define $D \mathrel{\widehat{=}} \mathsf{while}\ E\ \mathsf{do}\ C$. We begin with a technical result on loops in small-step semantics, which decomposes computations of D into a series of iterations like this:

$$\langle D,\, \sigma_0\rangle \longmapsto \overbrace{\langle C; D,\, \sigma_0\rangle \longmapsto^* \langle D,\, \sigma_1\rangle} \longmapsto \overbrace{\langle C; D,\, \sigma_1\rangle \longmapsto^* \langle D,\, \sigma_2\rangle} \longmapsto^* \ldots$$

LEMMA. *For any state* σ_0 *we have*

(a) *If* $\langle D,\ \sigma_0\rangle \longmapsto^*$ *fault then there exists* $n \geq 0$ *and sequence of states* $\sigma_1, \ldots, \sigma_n$ *such that* $\langle C,\ \sigma_n\rangle \longmapsto^*$ *fault and*

$$\langle C,\ \sigma_{i-1}\rangle \longmapsto^* \langle \mathsf{skip},\ \sigma_i\rangle \quad \textit{and} \quad \sigma_{i-1}(E) \neq 0 \quad \textit{for all } i,\ 0 < i \leq n \tag{21}$$

(b) *If* $\langle D,\ \sigma_0\rangle \longmapsto^* \langle \mathsf{skip},\ \sigma'\rangle$ *there exists* $n \geq 0$ *and states* $\sigma_1, \ldots, \sigma_n$ *such that* $\sigma' = \sigma_n$, $\sigma_n(E) = 0$*, and (21) holds.*

Proof of the lemma is left to the reader.

To prove soundness of WHILE, suppose $\sigma_0 \models P \wedge g = \mathsf{alloc}$.

For Safety, suppose $\langle D,\ \sigma_0\rangle \longmapsto^*$ *fault*. Then by the Lemma part (a), we have a series of iterations and states σ_i, such that $\langle C,\ \sigma_n\rangle \longmapsto^*$ *fault*. Using the first premise of WHILE and assumption $\sigma_0 \models P$ we have by induction on i that $\sigma_n \models P$. And now the premise says it is not the case that $\langle C,\ \sigma_n\rangle \longmapsto^*$ *fault*, a contradiction.

For Post, suppose $\langle D, \sigma_0\rangle \longmapsto^* \langle \mathsf{skip}, \sigma'\rangle$ and let n and the states σ_i be as in the Lemma part (b). By induction on i, using the first premise, we have $\sigma_n \models P$. And by (b), we have $\sigma_n(E) = 0$.

For Effect, we again consider the iterates and prove by induction on i that $\sigma_0 \rightarrow \sigma_i \models \varepsilon$. The base case, $i = 0$, is immediate. For the inductive step, suppose $\langle C,\ \sigma_{i-1}\rangle \longmapsto^* \langle \mathsf{skip},\ \sigma_i\rangle$, noting that $\sigma_{i-1} \models P \wedge E \neq 0$ from preceding assumptions. By induction, we have $\sigma_0 \rightarrow \sigma_{i-1} \models \varepsilon$ and by premise of the rule we have $\sigma_{i-1} \rightarrow \sigma_i \models \varepsilon, \mathsf{wr}\, H\text{`}\overline{f}$. We obtain $\sigma_0 \rightarrow \sigma_i \models \varepsilon$ by instantiating Lemma 6.12 with $P_0 := P \wedge g = \mathsf{alloc}$, $P_1 := P \wedge E \neq 0$, $\varepsilon_1 := \varepsilon$, $\varepsilon_2 := \varepsilon$, $\sigma_1 := \sigma_{i-1}$, and $\sigma_2 := \sigma_i$. It remains to check the condition of Lemma 6.12:

— $\sigma_0 \models P \wedge g = \mathsf{alloc}$ and $\sigma_{i-1} \models P \wedge E \neq 0$, which we have from above
— $\sigma_0 \rightarrow \sigma_{i-1} \models \varepsilon$ which we have by induction.
— $\sigma_{i-1} \rightarrow \sigma_i \models \varepsilon, \mathsf{wr}\, H\text{`}\overline{f}$, which we have from $\langle C,\ \sigma_{i-1}\rangle \longmapsto^* \langle \mathsf{skip},\ \sigma_i\rangle$ and the first premise, using $\sigma_{i-1} \models P$ (proved previously) and $\sigma_{i-1}(E) \neq 0$ by (21).
— ε is $(P \wedge g = \mathsf{alloc})/\varepsilon$-immune, which follows from the rule's condition that ε is P/ε-immune
— for every $\mathsf{fr}\, G$ in ε we have G is $(P \wedge E \neq 0)/(\varepsilon, \mathsf{wr}\, H\text{`}\overline{f})$-immune: this holds by the rule's condition that ε is fr-free.
— $\sigma_{i-1}(H) \cap \sigma_0(\mathsf{alloc}) = \varnothing$, which we have because

$$\begin{array}{lll} & \sigma_{i-1}(H) \cap \sigma_0(\mathsf{alloc}) & \\ = & \sigma_{i-1}(H) \cap \sigma_0(g) & \text{by } \sigma_0 \models P \wedge g = \mathsf{alloc} \\ = & \sigma_{i-1}(H) \cap \sigma_{i-1}(g) & \text{by } \sigma_0(g) = \sigma_{i-1}(g) \text{ because } \sigma_0 \rightarrow \sigma_{i-1} \models \varepsilon \text{ and } \mathsf{wr}\, g \notin \varepsilon \\ = & \varnothing & \text{by } \sigma_{i-1} \models P \text{ and } P \Rightarrow H \,\#\, g \text{ and } \mathsf{null} \notin \sigma_0(\mathsf{alloc})\ . \end{array}$$

(FRAME). Suppose $\sigma \models P \wedge Q$. Our assumptions are $\models \{P\}\ C\ \{P'\}\ [\varepsilon]$, $P \models \delta\ \mathsf{frm}\ Q$, and validity of $P \wedge Q \Rightarrow \delta \between \varepsilon$. We must show $\models \{P \wedge Q\}\ C\ \{P' \wedge Q\}\ [\varepsilon]$. Any σ that satisfies $P \wedge Q$ satisfies P, so Safety is a direct consequence of $\models \{P\}\ C\ \{P'\}\ [\varepsilon]$. Moreover, consider any terminating computation $\langle C,\ \sigma\rangle \longmapsto^* \langle \mathsf{skip},\ \sigma'\rangle$. Then, Effect is immediate: $\sigma \rightarrow \sigma' \models \varepsilon$, again from the premise. We also have $\sigma' \models P'$, so to show Post it remains to show $\sigma' \models Q$. Now we use $\sigma \models P \wedge Q$ and the side condition $P \wedge Q \Rightarrow \delta \between \varepsilon$ to first conclude $\sigma \models \delta \between \varepsilon$. Which, together with $\sigma \rightarrow \sigma' \models \varepsilon$ (Effect), yields $\mathit{Agree}(\sigma, \sigma', \delta)$ (Lemma 6.8). Now we appeal to the definition of $P \models \delta\ \mathsf{frm}\ Q$ (Definition 6.1): from $\mathit{Agree}(\sigma, \sigma', \delta)$ and $\sigma \models P \wedge Q$ we obtain $\sigma' \models Q$.

(EXTENDCTX). By well-formedness of the premise, x does not occur in C or in the specification. For any $(\Gamma, x : T)$-state σ such that $\sigma \models^{\Gamma, x:T} P$, we have $\sigma \restriction x \models^{\Gamma} P$ and

conversely for final states and P', see Eq. (11) in Section 4.2. Any computation from $\langle C,\, \sigma\rangle$ yields a computation from $\langle C,\, \sigma\lceil x\rangle$ with exactly the same configurations except x is tossed. Moreover in the computation from $\langle C,\, \sigma\rangle$, every state τ has $\tau(x) = \sigma(x)$. Thus, validity of the conclusion follows easily from validity of the premise.

(SUBST). Restricting x to be a specification-only variable ensures that it does not occur in C, although x is part of the state space.[15] Because C does not write x, we have

$$\langle C,\, \sigma\rangle \longmapsto^* \langle \mathsf{skip},\, \sigma'\rangle \text{ implies } \sigma(x) = \sigma'(x) \text{ for all } \sigma, \sigma' \tag{22}$$

Because C neither reads nor writes x—nor depends on it via the allocator, owing to Assumption 7.2—the set of outcomes from a state σ is the same as the set of outcomes from a state that agrees with σ except for the value of x. Thus, for any value v and states σ, σ', we have

$$\langle C,\, \sigma\rangle \longmapsto^* \mathit{fault} \text{ iff } \langle C,\, [\sigma \mid x : v]\,\rangle \longmapsto^* \mathit{fault} \tag{23}$$

$$\langle C,\, \sigma\rangle \longmapsto^* \langle \mathsf{skip},\, \sigma'\rangle \text{ iff } \langle C,\, [\sigma \mid x : v]\,\rangle \longmapsto^* \langle \mathsf{skip},\, [\sigma' \mid x : v]\,\rangle \tag{24}$$

To prove soundness of the rule, suppose the premise judgment is valid:

$$\models \{\, P \,\}\ C\ \{\, P' \,\}\ [\varepsilon] \tag{25}$$

To prove the conclusion, consider any σ such that $\sigma \models P/x{\rightarrow}F$. Our obligations are

(Safety). It is not the case that $\langle C,\, \sigma\rangle \longmapsto^* \mathit{fault}$.
(Post). $\sigma' \models P'/x{\rightarrow}F$, for all σ' with $\langle C,\, \sigma\rangle \longmapsto^* \langle \mathsf{skip},\, \sigma'\rangle$.
(Effect). $\sigma{\rightarrow}\sigma' \models \varepsilon/x{\rightarrow}F$, for all σ' with $\langle C,\, \sigma\rangle \longmapsto^* \langle \mathsf{skip},\, \sigma'\rangle$.

For brevity, define $\tau = [\sigma \mid x : \sigma(F)]$. From $\sigma \models P/x{\rightarrow}F$ by Eq. (13) in Section 4, we get $\tau \models P$. Thus, using (25), we have that $\langle C,\, \tau\rangle$ does not fault. Hence, it is not the case that $\langle C,\, \sigma\rangle \longmapsto^* \mathit{fault}$ owing to (23), which establishes Safety.

For Post, consider any σ' such that $\langle C,\, \sigma\rangle \longmapsto^* \langle \mathsf{skip},\, \sigma'\rangle$. By (24), we have $\langle C,\, \tau\rangle \longmapsto^* \langle \mathsf{skip},\, \tau'\rangle$ where $\tau' = [\sigma' \mid x : \sigma(F)]$. By (25) and preceding definitions, we have $\tau' \models P'$. Later, we will show

$$\sigma(F) = \sigma'(F), \tag{26}$$

which implies that $\tau' = [\sigma' \mid x : \sigma'(F)]$. Hence, $[\sigma' \mid x : \sigma'(F)] \models P'$, which by (13) is equivalent to $\sigma' \models P'/x{\rightarrow}F$. This completes the proof of Post.

To show Effect, let ε' be the set of all effects in ε other than freshness effects. We now prove that $\sigma{\rightarrow}\sigma' \models \varepsilon'/x{\rightarrow}F$; later we use this to deal with freshness effects in ε. By (25) and the previous definitions, we have $\tau{\rightarrow}\tau' \models \varepsilon$, which by definition of τ, τ' is equivalent to

$$[\sigma \mid x : \sigma(F)] {\rightarrow} [\sigma' \mid x : \sigma(F)] \models \varepsilon. \tag{27}$$

Now we prove $\sigma{\rightarrow}\sigma' \models \varepsilon'/x{\rightarrow}F$ by cases in Definition 5.1:

(a) We have $\sigma'(x) = \sigma(x)$ by (22). If $\sigma(y) \neq \sigma'(y)$ for some variable y, distinct from x, then $[\sigma \mid x : \sigma(F)]\,(y) \neq [\sigma' \mid x : \sigma(F)]\,(y)$, so by (27) we have $\mathsf{wr}\, y \in \varepsilon'$ and thus $\mathsf{wr}\, y \in \varepsilon'/x{\rightarrow}F$.
(b) Suppose $o \in \sigma(\mathsf{alloc})$ and $\sigma(o.f) \neq \sigma'(o.f)$. Then $o \in [\sigma \mid x : \sigma(F)]\,(\mathsf{alloc})$ and $[\sigma \mid x : \sigma(F)]\,(o.f) \neq [\sigma' \mid x : \sigma'(F)]\,(o.f)$ (recall that alloc is not a specification-only variable, so x is not alloc). Thus, by (27) there is some $\mathsf{wr}\, G\text{`}f$ in $\varepsilon'$ with $o \in [\sigma \mid x : \sigma(F)]\,(G)$. Hence, by (12) in Section 4, we have $o \in \sigma(G/x{\rightarrow}F)$ so this update is licensed by the effect $\mathsf{wr}\,(G/x{\rightarrow}F)\text{`}f$ which is in $\varepsilon'/x{\rightarrow}F$.

[15] In the conference version of this article [Banerjee et al. 2008c], the restriction on x is expressed by the conditions $\mathsf{rd}\, x \notin \varepsilon$ and $\mathsf{wr}\, x \notin \varepsilon$. This turned out to be unsound.

There are no freshness effects in ε' or $\varepsilon'/x{\rightarrow}F$, by definition of ε'.

Now we complete the proof of Effect by considering the freshness effects.

(c) Consider any $\mathsf{fr}\,H$ in $\varepsilon/x{\rightarrow}F$, that is, any $\mathsf{fr}\,G/x{\rightarrow}F$ such that $\mathsf{fr}\,G$ is in ε. We must show $\sigma'(G/x{\rightarrow}F) \subseteq \sigma'(\mathsf{alloc}) \setminus \sigma(\mathsf{alloc})$, that is, $p \in \sigma'(\mathsf{alloc}) \setminus \sigma(\mathsf{alloc})$ for all $p \in \sigma'(G/x{\rightarrow}F)$. Observe

$$\begin{array}{lll} & p \in \sigma'(G/x{\rightarrow}F) & \\ \equiv & p \in [\sigma' \mid x : \sigma'(F)]\,(G) & \text{by (12)} \\ \equiv & p \in [\sigma' \mid x : \sigma(F)]\,(G) & \text{by (26)} \\ \Rightarrow & p \in [\sigma' \mid x : \sigma(F)]\,(\mathsf{alloc}) \setminus [\sigma \mid x : \sigma(F)]\,(\mathsf{alloc}) & \text{by } \mathsf{fr}\,G \in \varepsilon,\ (27) \\ \Rightarrow & p \in \sigma'(\mathsf{alloc}) \setminus \sigma(\mathsf{alloc}) & \text{by def } [- \mid - : -]. \end{array}$$

The last step uses that x is not alloc (by Assumption 7.2).

It remains to prove (26). By the assumption $\sigma \models P/x{\rightarrow}F$ and the rule's side condition $(P/x{\rightarrow}F) \Rightarrow \mathit{ftpt}(F)\ \mathbin{\prime\!\!\prime}\ (\varepsilon/x{\rightarrow}F)$, we have $\sigma \models \mathit{ftpt}(F)\ \mathbin{\prime\!\!\prime}\ (\varepsilon/x{\rightarrow}F)$. But freshness effects have no bearing on separators, so $\mathit{ftpt}(F)\ \mathbin{\prime\!\!\prime}\ (\varepsilon'/x{\rightarrow}F) \equiv \mathit{ftpt}(F)\ \mathbin{\prime\!\!\prime}\ (\varepsilon/x{\rightarrow}F)$ where, as above, ε' is ε but without freshness effects. Previously, we proved $\sigma{\rightarrow}\sigma' \models \varepsilon'/x{\rightarrow}F$ so using Lemmas 6.8 and 6.2 we obtain (26).

(EXIST). Suppose σ is a Γ-state that satisfies $\exists x : T \in G \cdot P$. Consider any p that witnesses the existential. That is, let σ_p be the $(\Gamma, x : T)$-state with p for x, that is, $\sigma_p = \mathit{Extend}(\sigma, x, p)$, and suppose $\sigma_p \models x \in G \wedge P$. Note that C is well formed in context Γ so it neither reads nor writes x.

By validity of the premise, we have that $\langle C,\ \sigma_p\rangle$ does not fault. Therefore, by Lemma 3.4(a), $\langle C,\ \sigma\rangle$ does not fault either thus establishing Safety. Furthermore, by Lemma 3.4(b), if $\langle C,\ \sigma_p\rangle \longmapsto^* \langle \mathsf{skip},\ \sigma'_p\rangle$— where $\sigma'_p = \mathit{Extend}(\sigma', x, p)$—then $\langle C,\ \sigma\rangle \longmapsto^* \langle \mathsf{skip},\ \sigma'\rangle$. Again, by validity of the premise, $\sigma'_p \models P'$ and $\sigma_p{\rightarrow}\sigma'_p \models \varepsilon$. Because P' does not mention x, we obtain $\sigma' \models P'$, establishing Post. And, because ε does not mention x, we have by Definition 5.1, that $\sigma{\rightarrow}\sigma' \models \varepsilon$, establishing Effect.

(POSTTOFR). Safety follows by validity of the premise and, with assumption $\langle C, \sigma\rangle \longmapsto^* \langle \mathsf{skip}, \sigma'\rangle$, so does Post. With the assumption in hand, it remains to establish Effect, that is, $\sigma{\rightarrow}\sigma' \models \varepsilon, \mathsf{fr}\,G$. By validity of premise we get $\sigma{\rightarrow}\sigma' \models \varepsilon$ so it remains to show $\sigma{\rightarrow}\sigma' \models \mathsf{fr}\,G$, that is, the inclusion $\sigma'(G) \subseteq \sigma'(\mathsf{alloc}) \setminus \sigma(\mathsf{alloc})$. From $\sigma \models P$ and $P \Rightarrow r = \mathsf{alloc}$, we have $\sigma(r) = \sigma(\mathsf{alloc})$. Because $\mathsf{rd}\,r\ \mathbin{\prime\!\!\prime}\ \varepsilon$, we have $\mathsf{wr}\,r \notin \varepsilon$, so from $\sigma{\rightarrow}\sigma' \models \varepsilon$ we get $\sigma'(r) = \sigma(r) = \sigma(\mathsf{alloc})$. From $\sigma' \models P'$ and $P' \Rightarrow G \cap (r \cup \{\mathsf{null}\}) = \varnothing$ we get $\mathsf{null} \notin \sigma'(G)$ and we get $\sigma'(G) \cap \sigma'(r) = \varnothing$, which implies $\sigma'(G) \cap \sigma(\mathsf{alloc}) = \varnothing$. By semantics, we have $\sigma'(G) \subseteq \sigma'(\mathsf{alloc}) \cup \{\mathsf{null}\}$ and then, using $\mathsf{null} \notin \sigma'(G)$, we get $\sigma'(G) \subseteq \sigma'(\mathsf{alloc}) \setminus \sigma(\mathsf{alloc})$.

(FIELDMASK). Suppose $\sigma \models P$. Safety and Post follow by validity of premise, so that $\sigma' \models P'$, where $\langle C, \sigma\rangle \longmapsto^* \langle \mathsf{skip}, \sigma'\rangle$. It remains to show Effect, that is, $\sigma{\rightarrow}\sigma' \models \varepsilon$, for which we can assume $\sigma{\rightarrow}\sigma' \models \mathsf{wr}\,\{x\}\text{`}f, \varepsilon$. It is enough to consider part (b) of Definition 5.1. Suppose $o \in \mathsf{alloc}(\sigma)$, $f \in \mathit{Fields}(\mathit{Type}(o, \sigma))$ and $\sigma(o.f) \neq \sigma'(o.f)$. If $o \neq \sigma(x)$, then there exists a region $G$ such that $\mathsf{wr}\,G\text{`}f \in \varepsilon$ such that $o \in \sigma(G)$. If $o = \sigma(x)$, then we reach a contradiction: $\sigma \models \mathsf{rd}\,y\ \mathbin{\prime\!\!\prime}\ \varepsilon$ and $\sigma \models \mathsf{rd}\,x\ \mathbin{\prime\!\!\prime}\ \varepsilon$ yield $\sigma(y) = \sigma'(y)$ and $\sigma(x) = \sigma'(x)$, respectively; we get $\sigma(o.f) = \sigma'(o.f)$ owing to $\sigma \models x.f = y$ and $\sigma' \models x.f = y$, which are given by premise $P \vee P' \Rightarrow x.f = y$. □

8. EXAMPLES WITH LOOP AND ALLOCATION

This section presents detailed proofs of contrived examples that illustrate novel aspects of the proof rules, in particular immunity and freshness effects, on which earlier examples did not touch.

Our approach to invariants relies heavily on ghost state, which is supposed to serve only for specifying and reasoning about the "underlying program" from which ghost instrumentation has been removed. The proofs in this section illustrate both the use of specification-only variables and the use of ghost variables that are updated as part of the program annotation. To justify the latter, Section 8.3 discusses a proof rule for elimination of ghost variables.

8.1. Allocation in a Loop

Consider the following commands, in context $\Gamma = \mathsf{alloc} : \mathsf{rgn}, r : \mathsf{rgn}, n : \mathit{Node}$:

$$C \;\widehat{=}\; r := \varnothing;\ \mathsf{while\ 1\ do}\ B$$
$$B \;\widehat{=}\; n := \mathsf{new}\ \mathit{Node};\ r := r \cup \{n\};$$

The loop does not terminate but serves our expository purpose. We shall prove

$$\vdash^{\Gamma} \{\, \mathit{true} \,\}\ C\ \{\, \mathit{true} \,\}\ [r, n, \mathsf{alloc}, \mathsf{fr}\, r] \tag{28}$$

Let $\Delta \;\widehat{=}\; \Gamma, s : \mathsf{rgn}$. Variable s is used to snapshot the initial value of alloc. To save space, we write $\hat{s}$ to abbreviate $s \cup \{\mathsf{null}\}$. Later we derive

$$\vdash^{\Delta} \{\, r \cap \hat{s} = \varnothing \,\}\ B\ \{\, r \cap \hat{s} = \varnothing \,\}\ [r, n, \mathsf{alloc}] \tag{29}$$

From (29), rule WHILE (instantiated with $g := s$, $\overline{f}$ empty, and $H := \varnothing$) yields

$$\vdash^{\Delta} \{\, r \cap \hat{s} = \varnothing \wedge s = \mathsf{alloc} \,\}\ \mathsf{while\ 1\ do}\ B\ \{\, r \cap \hat{s} = \varnothing \wedge 1 = 0 \,\}\ [r, n, \mathsf{alloc}]$$

(The immunity condition in WHILE holds vacuously and no fields are written.) Now we can use POSTTOFR (using $r := s$, $G := r$ in that rule) to get

$$\vdash^{\Delta} \{\, r \cap \hat{s} = \varnothing \wedge s = \mathsf{alloc} \,\}\ \mathsf{while\ 1\ do}\ B\ \{\, r \cap \hat{s} = \varnothing \wedge 1 = 0 \,\}\ [r, n, \mathsf{alloc}, \mathsf{fr}\, r]$$

and by CONSEQ on the above we have

$$\vdash^{\Delta} \{\, r \cap \hat{s} = \varnothing \wedge s = \mathsf{alloc} \,\}\ \mathsf{while\ 1\ do}\ B\ \{\, \mathit{true} \,\}\ [r, n, \mathsf{alloc}, \mathsf{fr}\, r] \tag{30}$$

Leaving (30) aside, we use SIMPLEASSIGN to get $\vdash^{\Delta} \{\, \mathit{true} \,\}\ r := \varnothing\ \{\, r = \varnothing \,\}\ [r]$ (this is one of the derived rules in Section 7.2). Hence, by FRAME (framing $s = \mathsf{alloc}$ by $\mathsf{rd}\, s, \mathsf{rd}\, \mathsf{alloc}$) we get

$$\vdash^{\Delta} \{\, s = \mathsf{alloc} \,\}\ r := \varnothing\ \{\, r = \varnothing \wedge s = \mathsf{alloc} \,\}\ [r]$$

Now, by CONSEQ using $r = \varnothing \wedge s = \mathsf{alloc} \Rightarrow r \cap \hat{s} = \varnothing \wedge s = \mathsf{alloc}$, we get

$$\vdash^{\Delta} \{\, s = \mathsf{alloc} \,\}\ r := \varnothing\ \{\, r \cap \hat{s} = \varnothing \wedge s = \mathsf{alloc} \,\}\ [r]\,. \tag{31}$$

By SIMPLESEQ, from (31) and (30), we get

$$\vdash^{\Delta} \{\, s = \mathsf{alloc} \,\}\ C\ \{\, \mathit{true} \,\}\ [r, n, \mathsf{alloc}, \mathsf{fr}\, r]\,.$$

Apropos the side conditions of SIMPLESEQ, note in particular that $\mathsf{wr}\, r, n, \mathsf{alloc}, \mathsf{fr}\, r$ is $(\mathsf{wr}\, r/(s = \mathsf{alloc}))$-immune. Now, by EXISTREGION we reach judgment (28).

It remains to derive (29). By ALLOC2 and CONSEQ, we infer

$$\vdash^{\Delta} \{\, \mathit{true} \,\}\ n := \mathsf{new}\ \mathit{Node}\ \{\, n \neq \mathsf{null} \wedge n \notin s \,\}\ [n, \mathsf{alloc}]\,,$$

whence by FRAME we have

$$\vdash^{\Delta} \{\, r \cap \hat{s} = \varnothing \,\}\ n := \mathsf{new}\ \mathit{Node}\ \{\, r \cap \hat{s} = \varnothing \wedge n \neq \mathsf{null} \wedge n \notin s \,\}\ [n, \mathsf{alloc}]\,.$$

Now we have by CONSEQ from the above

$$\vdash^{\Delta} \{\, r \cap \hat{s} = \varnothing \,\}\ n := \mathsf{new}\ \mathit{Node}\ \{\, (r \cup \{n\}) \cap \hat{s} = \varnothing \,\}\ [n, \mathsf{alloc}] \tag{32}$$

We now consider the assignment $r := r \cup \{n\}$. We have by BACKWARDSASSIGN

$$\vdash^{\Delta} \{ (r \cup \{n\}) \cap \hat{s} = \varnothing \}\ r := r \cup \{n\}\ \{ r \cap \hat{s} = \varnothing \}\ [r] \tag{33}$$

Now applying SEQ to (32) and (33) we have (29).

8.2. Updating Objects Allocated in a Loop

Consider the following:

$$\begin{aligned} B &\mathrel{\widehat{=}} n := \mathsf{new}\ \mathit{Node}; n' := n;\ r := \{n\};\ p := n; \\ C &\mathrel{\widehat{=}} n := \mathsf{new}\ \mathit{Node}; p.\mathit{nxt} := n; p := n;\ r := r \cup \{n\}; \\ \Gamma &\mathrel{\widehat{=}} n : \mathit{Node}, n' : \mathit{Node}, p : \mathit{Node}, r : \mathsf{rgn}, \mathsf{alloc} : \mathsf{rgn} \\ \Delta &\mathrel{\widehat{=}} \Gamma, s : \mathsf{rgn} \end{aligned}$$

We shall prove

$$\vdash^{\Gamma} \{ \mathit{true} \}\ B; \mathsf{while}\ 1\ \mathsf{do}\ C\ \{ \mathit{true} \}\ [n, n', p, r, \mathsf{alloc}, \mathsf{fr}\, r] \tag{34}$$

This program would be useful if we add to B an assignment $\mathit{root} := n'$ following the first assignment. Then, it would create a linked list of nodes starting at root. Of course, another guard condition would be needed too.

Let $V \mathrel{\widehat{=}} p \neq \mathsf{null} \wedge p \in r \wedge (r \setminus \{n'\}) \cap \mathsf{alloc} = \varnothing \wedge n' \in r$. First, we show

$$\vdash^{\Gamma} \{ \mathit{true} \}\ B\ \{ V \}\ [n, n', \mathsf{alloc}, r, p, \mathsf{fr}\, \{n'\}]\,. \tag{35}$$

Later, we show

$$\vdash^{\Gamma} \{ V \}\ \mathsf{while}\ 1\ \mathsf{do}\ C\ \{ n' \in r \}\ [n, n'.\mathit{nxt}, p, r, \mathsf{alloc}, \mathsf{fr}\, (r \setminus \{n'\})]\,. \tag{36}$$

Then, by SEQ on (35) and (36), and using FRUNION, we have

$$\vdash^{\Gamma} \{ \mathit{true} \}\ B; \mathsf{while}\ 1\ \mathsf{do}\ C\ \{ n' \in r \}\ [n, n', p, r, \mathsf{alloc}, \mathsf{fr}\, r \cup \{n'\}]\,,$$

whence by FRSUB and CONSEQ we have (34).

We first tackle (35). By ALLOC2 and CONSEQ, we have

$$\vdash^{\Delta} \{ s = \mathsf{alloc} \}\ n := \mathsf{new}\ \mathit{Node}\ \{ n \neq \mathsf{null} \wedge n \notin s \}\ [n, \mathsf{alloc}]\,. \tag{37}$$

By SIMPLEASSIGN and FRAME, we have

$$\vdash^{\Delta} \{ n \neq \mathsf{null} \wedge n \notin s \}\ n' := n\ \{ n' = n \wedge n \neq \mathsf{null} \wedge n \notin s \}\ [n']\,. \tag{38}$$

By SIMPLESEQ on (37) and (38), we get

$$\vdash^{\Delta} \{ s = \mathsf{alloc} \}\ n := \mathsf{new}\ \mathit{Node}; n' := n\ \{ n' = n \wedge n \neq \mathsf{null} \wedge n \notin s \}\ [n, n', \mathsf{alloc}]$$

so by CONSEQ, we have

$$\vdash^{\Delta} \{ s = \mathsf{alloc} \}\ n := \mathsf{new}\ \mathit{Node}; n' := n\ \{ n' = n \wedge n \neq \mathsf{null} \wedge n' \notin s \}\ [n, n', \mathsf{alloc}]\,.$$

Now by POSTTOFR, with $G := \{n'\}$ and $r := s$ in the rule, and noting that $\{n'\} \cap (s \cup \{\mathsf{null}\}) = \varnothing$,

$$\vdash^{\Delta} \{ s = \mathsf{alloc} \}\ n := \mathsf{new}\ \mathit{Node}; n' := n\ \{ n' = n \wedge n \neq \mathsf{null} \wedge n' \notin s \}\ [\varepsilon]\,,$$

where $\varepsilon \mathrel{\widehat{=}} \{n, n', \mathsf{alloc}, \mathsf{fr}\, \{n'\}\}$. By CONSEQ, we have

$$\vdash^{\Delta} \{ s = \mathsf{alloc} \}\ n := \mathsf{new}\ \mathit{Node}; n' := n\ \{ n' = n \wedge n \neq \mathsf{null} \}\ [\varepsilon]$$

and by EXISTREGION, we get

$$\vdash^{\Gamma} \{ \mathit{true} \}\ n := \mathsf{new}\ \mathit{Node}; n' := n\ \{ n' = n \wedge n \neq \mathsf{null} \}\ [\varepsilon]\,. \tag{39}$$

By SIMPLEASSIGN and FRAME

$$\vdash^{\Gamma} \{\, n' = n \wedge n \neq \mathsf{null} \,\}\ r := \{n\}\ \{\, n' = n \wedge n \neq \mathsf{null} \wedge r = \{n\} \,\}\ [r]\,,$$

whence by CONSEQ we get

$$\vdash^{\Gamma} \{\, n' = n \wedge n \neq \mathsf{null} \,\}\ r := \{n\}\ \{\, n \neq \mathsf{null} \wedge n \in r \wedge r = \{n'\} \,\}\ [r]\,. \tag{40}$$

Now from (39) and (40), we have by SEQ

$$\vdash^{\Gamma} \{\, \mathit{true} \,\}\ n := \mathsf{new}\ \mathit{Node}; n' := n; r := \{n\}\ \{\, n \neq \mathsf{null} \wedge n \in r \wedge r = \{n'\} \,\}\ [r, \varepsilon]\,. \tag{41}$$

By SIMPLEASSIGN and FRAME, we have

$$\vdash^{\Gamma} \{\, n \neq \mathsf{null} \wedge n \in r \wedge r = \{n'\} \,\}\ p := n\ \{\, p = n \wedge n \neq \mathsf{null} \wedge n \in r \wedge r = \{n'\} \,\}\ [p]\,,$$

whence by CONSEQ, we get

$$\vdash^{\Gamma} \{\, n \neq \mathsf{null} \wedge n \in r \wedge r = \{n'\} \,\}\ p := n\ \{\, p \neq \mathsf{null} \wedge p \in r \wedge r = \{n'\} \,\}\ [p]\,. \tag{42}$$

Now by SEQ on (41) and (42), we have

$$\vdash^{\Gamma} \{\, \mathit{true} \,\}\ B\ \{\, p \neq \mathsf{null} \wedge p \in r \wedge r = \{n'\} \,\}\ [\, r, p, \varepsilon]$$

whence we get (35) by CONSEQ (using that $r = \{n'\} \Rightarrow (r \setminus \{n'\}) \cap \mathsf{alloc} = \varnothing$ and $r = \{n'\} \Rightarrow n' \in r$).

We now tackle (36) for which the loop invariant, in context Δ, is

$$P \mathrel{\widehat{=}} p \neq \mathsf{null} \wedge p \in r \wedge (r \setminus \{n'\}) \cap \hat{s} = \varnothing \wedge n' \in r.$$

Here, as in Section 8.1, we write $\hat{s}$ to abbreviate $s \cup \{\mathsf{null}\}$. For C, we will show that

$$\vdash^{\Delta} \{\, P \,\}\ C\ \{\, P \,\}\ [n, \mathsf{alloc}, r.\mathit{nxt}, p, r]\,. \tag{43}$$

From (43), by SUBEFFECT (using $P \Rightarrow r \subseteq (r \setminus \{n'\}) \cup \{n'\}$), we have

$$\vdash^{\Delta} \{\, P \,\}\ C\ \{\, P \,\}\ [n, \mathsf{alloc}, (r \setminus \{n'\}).\mathit{nxt}, n'.\mathit{nxt}, p, r]\,.$$

Then, by WHILE, noting that the immunity condition holds and also that $P \Rightarrow (r \setminus \{n'\}) \cap s = \varnothing$, we get

$$\vdash^{\Delta} \{\, P \wedge s = \mathsf{alloc} \,\}\ \mathsf{while}\ 1\ \mathsf{do}\ C\ \{\, P \wedge 1 = 0 \,\}\ [n, n'.\mathit{nxt}, p, r, \mathsf{alloc}]\,.$$

By POSTTOFR (using $G := r \setminus \{n'\}$ and $r := s$ in the rule), we get

$$\vdash^{\Delta} \{\, P \wedge s = \mathsf{alloc} \,\}\ \mathsf{while}\ 1\ \mathsf{do}\ C\ \{\, P \wedge 1 = 0 \,\}\ [n, n'.\mathit{nxt}, p, r, \mathsf{alloc}, \mathsf{fr}\,(r \setminus \{n'\})]\,.$$

Then, by CONSEQ followed by EXISTREGION on this we get (36).

It remains to establish (43). Let $Q \mathrel{\widehat{=}} n \neq \mathsf{null} \wedge n \notin r \wedge n \notin s$. By ALLOC2 (twice), CONJ, and CONSEQ, we have $\vdash^{\Delta} \{\, \mathit{true} \,\}\ n := \mathsf{new}\ \mathit{Node}\ \{\, Q \,\}\ [n, \mathsf{alloc}]$, whence, by FRAME, we have

$$\vdash^{\Delta} \{\, P \,\}\ n := \mathsf{new}\ \mathit{Node}\ \{\, Q \wedge P \,\}\ [n, \mathsf{alloc}]\,. \tag{44}$$

By FIELDUPD, we have

$$\vdash^{\Delta} \{\, p \neq \mathsf{null} \,\}\ p.\mathit{nxt} := n\ \{\, p.\mathit{nxt} = n \,\}\ [p.\mathit{nxt}]\,,$$

whence, by FRAME and CONSEQ, we have

$$\vdash^{\Delta} \{\, Q \wedge P \,\}\ p.\mathit{nxt} := n\ \{\, Q \wedge (r \setminus \{n'\}) \cap \hat{s} = \varnothing \wedge n' \in r \,\}\ [p.\mathit{nxt}]\,. \tag{45}$$

By SIMPLESEQ on (44) and (45) and by SUBEFFECT (because $P \Rightarrow p \in r$), we have

$$\vdash^{\Delta} \{\, P \,\}\ n := \mathsf{new}\ \mathit{Node}; p.\mathit{nxt} := n\ \{\, Q \wedge (r \setminus \{n'\}) \cap \hat{s} = \varnothing \wedge n' \in r \,\}\ [n, \mathsf{alloc}, r.\mathit{nxt}]\,.$$

Let $R \mathrel{\widehat{=}} n \neq \mathsf{null} \wedge n \notin r \wedge (r \cup \{n\} \setminus \{n'\}) \cap \hat{s} = \varnothing \wedge n' \in r \cup \{n\}$. Then, by CONSEQ we have

$$\vdash^{\Delta} \{\, P \,\}\ n := \mathsf{new}\ \mathit{Node}; p.\mathit{nxt} := n\ \{\, R \,\}\ [n, \mathsf{alloc}, r.\mathit{nxt}]\,. \tag{46}$$

By SIMPLEASSIGN and FRAME, we have

$$\vdash^{\Delta} \{\, R \,\}\ p := n\ \{\, p = n \wedge R \,\}\ [p] \tag{47}$$

Now by SIMPLESEQ on (46) and (47) we have

$$\vdash^{\Delta} \{\, P \,\}\ n := \mathsf{new}\ \mathit{Node}; p.\mathit{nxt} := n; p := n\ \{\, p = n \wedge R \,\}\ [n, \mathsf{alloc}, r.\mathit{nxt}, p]$$

Let $W \mathrel{\widehat{=}} p \in r \cup \{n\} \wedge (r \cup \{n\} \setminus \{n'\}) \cap \hat{s} = \varnothing \wedge p \neq \mathsf{null} \wedge n' \in r \cup \{n\}$. Then, by CONSEQ on this equation, using $n \in r \cup \{n\}$, $n \in \mathsf{alloc} \Rightarrow n \neq \mathsf{null}$ and $p = n$, we have

$$\vdash^{\Delta} \{\, P \,\}\ n := \mathsf{new}\ \mathit{Node}; p.\mathit{nxt} := n; p := n\ \{\, W \,\}\ [n, \mathsf{alloc}, r.\mathit{nxt}, p] \tag{48}$$

By BACKWARDSASSIGN, we have

$$\vdash^{\Delta} \{\, W \,\}\ r := r \cup \{n\}\ \{\, P \,\}\ [r]\,. \tag{49}$$

By SEQ on (48) and (49), we get (43).

8.3. Ghost Elimination

In proofs by hand, we seldom find the need for ghost variables; it is often enough to introduce specification-only snapshot variables in specifications of subcommands, as in the examples of the preceding sections. But those examples also use ghost variable r in their specifications. Ghost state is not intended to be observable, so ultimately it is used to specify internal interfaces of program components but not in specifications of main program requirements. (But see Hofmann and Pavlova [2008].) The restrictions needed for sensible use of ghost state are well known, but it is worth reviewing the key ideas, which are embodied in the following proof rule:

$$\textsc{Aux}\ \frac{\vdash^{\Gamma} \{\, P \,\}\ \mathsf{var}\ \overline{v} : \overline{T}\ \mathsf{in}\ C\ \{\, P' \,\}\ [\overline{\varepsilon}] \qquad \mathit{auxil}(\overline{v}, C)}{\vdash^{\Gamma} \{\, P \,\}\ \mathit{erase}(\overline{v}, C)\ \{\, P' \,\}\ [\overline{\varepsilon}]}.$$

The side condition $\mathit{auxil}(\overline{v}, C)$ is defined to check that the variables in the list $\overline{v}$ only occur in assignments to variables in $\overline{v}$, that is, they are "auxiliary" in the sense defined by Owicki and Gries [1976]. Function erase is defined so that $\mathit{erase}(\overline{v}, C)$ replaces by skip every assignment to a variable in $\overline{v}$. (The use of skip ensures that the result is well formed.) By assignment we mean the grammatical forms $x := F$ and $x := y.f$ only. The precise definitions of auxil and erase are left to the reader.

As an example, for program C at the beginning of Section 8.1, we have $\mathit{auxil}(r, C)$. Furthermore, using rule SUBEFF we can drop the effect fr r from the specification (28). Using CONSEQ and then VAR, we can introduce local variable block for r, removing it from the typing context and making rule AUX applicable. This yields

$$\vdash \{\, \mathit{true} \,\}\ \mathsf{skip};\ \mathsf{while}\ 1\ \mathsf{do}\ n := \mathsf{new}\ \mathit{Node};\ \mathsf{skip}\ \{\, \mathit{true} \,\}\ [n, \mathsf{alloc}]\,.$$

Similarly, from (34), we can remove n' and r to obtain the program

$$n := \mathsf{new}\ \mathit{Node}; p := n; \mathsf{while}\ 1\ \mathsf{do}\ n := \mathsf{new}\ \mathit{Node}; p.\mathit{nxt} := n; p := n$$

with effects wr n, p, alloc.

Soundness of rule AUX rests on this fact: For any P-state σ, letting τ extend σ with the default initial values for $\overline{v}$, the outcomes of C from τ are in one-to-one correspondence with outcomes of $erase(\overline{v}, C)$ from σ.

We do not formalize the elimination of ghost fields, but their rationale is the same as for ghost variables: they do not influence the program's termination or other behavior as observed in predicates on non-ghost state. In verification tools, ghost state typically appears as special comments, ignored by the compiler but never erased per se.

9. RELATED WORK

Our approach originated in work on secure information flow analysis combining verification and type checking [Banerjee et al. 2008b]. For modular reasoning about information flow, Amtoft et al. [2006] introduce a relational logic that relies on abstract locations to abstract sets of concrete locations at a program point as commonly done in static analysis. In order to extend their work to declassification policies, which may depend on complex program state conditions, we needed to enrich the assertion language, which led to dropping their abstract interpretation of heap locations in favor of explicit regions.

Separation Logic. An important precursor to our work is separation logic [O'Hearn et al. 2001], a kind of Hoare logic in which a program's precondition implicitly specifies which preexisting heap cells may be read or written by the program. This heap footprint is the part of the heap necessary for the program to execute without memory faults in a partial-heap semantics. A special logical connective, separating conjunction, facilitates local reasoning as embodied in a frame rule. By contrast with our use of regions, heap footprints do not appear in separation logic formulas explicitly. Instead they are implicit in the semantics of the correctness judgment; this and the separating conjunction make possible an elegant frame rule with minimal side conditions. By contrast, our use of explicit footprints and ghost variables can be seen as skolemizing the existential implicit in separating conjunction (see Remark 6.9).

Because footprints are shadows of predicates, specifications in separation logic involve inductive predicates for traversing data structures, often together with quantification over predicates [Birkedal et al. 2005; Nanevski et al. 2006]. Higher order, induction, and separating conjunction all pose challenges for automated reasoning. However, exciting results have been achieved in automated static analysis using fragments of separation logic [Calcagno et al. 2011] as well as in interactive verification [Feng et al. 2008; Nanevski et al. 2010]. Although most work in separation logic is at the C level of abstraction, the jStar verifier [Distefano and Parkinson 2008] uses symbolic execution to implement a separation logic for sequential Java. The logic features abstraction by higher order predicates [Parkinson and Bierman 2005], as we discuss in Part II.

Methodologies. Several automated verifiers use intricate methodologies for reasoning about the heap in ordinary first-order logic. Our approach is inspired by the Boogie methodology [Barnett et al. 2004; Leino and Müller 2004; Naumann and Barnett 2006], which is explicitly based on all-states invariants that use ghost fields to express an ownership encapsulation régime. Boogie combines instance invariants into a global condition akin to our example $\forall p : Comp \in \mathsf{alloc} \cdot ok(p)$ in Section 2. A key feature of the methodology is the hiding of ownership-based invariants, for which Part II of this article offers a foundation and generalization (and discusses more related work).

Drossopoulou and Smith [2003] use regions denoted by ownership contexts to formulate a simple, type-based frame rule that resembles ours. Another type-based frame rule is used by Schlesinger et al. [2011] to reason about data integrity. Their work builds on that of Lahiri et al. [2011] who use linear types and explicit partial

heaps, dubbed "linear maps", in a classical assertion language for local reasoning in object-based programs. The use of linear maps involves novel program constructs including explicit transfer of objects between linear maps, which in their system are always disjoint. The resemblance to region logic seems clear and experimental comparisons may be interesting, for example, concerning the instrumentation burden that seems proportional to what we have seen in experiments with region logic.

Regions as disjoint sets of references is featured in the work of Marron et al. [2008] who formulate shape analyses where shapes of regions are characterized using standard graph-theoretic notions like trees and dags. In recent unpublished work Marron uses injectivity of fields to encode possibly transitive ownership: if region R' is a successor of another region R in the shape graph and R' has one incoming edge and this edge is injective then one can infer that each object in R' is owned by a unique object in R.

The idea to use regions and explicit ghost state to express frame conditions is directly inspired by the state-dependent effects of Kassios [2006, 2011], who worked out how it could be done effectively without global imposition of a fixed programming discipline like ownership. The state-dependent effects are termed *dynamic frames*, whence our term "dynamic boundary" (cf. Section 2). Kassios developed his ideas in the setting of a relational refinement theory and higher order logic; in particular, the framing of a formula is expressed directly as a second order predicate, quantifying over all global program states.

By contrast, we work out a Hoare logic based on first-order assertions. But there are similarities, for example, Metatheorem 5.4.1 in his thesis is related to our notion of immunity, as is the swinging pivots restriction of Leino and Nelson [2002, Section 8.3]. The latter grapples with stateful frame conditions of the form that we would write $\mathsf{wr}\,\{x\}$`f and justifies soundness of their treatment in ESC/Java [Flanagan et al. 2002] under somewhat restrictive conditions; the state of the art at that time was that this and other JML-based tools had various unsoundnesses in connection with frame conditions.

Like us, Smans et al. [2010] adapt Kassios' ideas to the setting of pre/post specifications and frame conditions. Where we work with sets of object references, they use sets of locations. (Of course, we can express finite location sets using singleton regions.) Moreover, their sets are given by pure functions. This entails reasoning about calls of (pure) methods in assertions. Smans et al.'s approach has been implemented in their VeriCool verifier and applied to examples like observer and iterator. This use of procedural abstraction has the benefit of abstracting from both object references and field names. It comes at the cost of reasoning about method calls in assertions (e.g., Rudich et al. [2008]). Our work is compatible with, but not dependent on, means of abstraction like pure methods in assertions [Naumann 2007; Rudich et al. 2008], model fields [Leino and Müller 2006], and data groups [Leino 1998].

Smans et al. [2012] avoid the need for a modifies clause somewhat in the manner of separation logic and instead infer frame conditions using special "access predicates", acc, with a permission-based semantics and special program constructs. (Their methodology is dubbed "implicit dynamic frames".) Every read/write of an expression $E.f$ is permitted by asserting $acc(E.f)$. Their assertion language includes the separating conjunction but the meaning of a formula $\phi_1 * \phi_2$ is that $\phi_1 \wedge \phi_2$ holds and the set of locations deemed accessible by the access predicates in ϕ_1 is disjoint from those of ϕ_2.

Verifiers. The Jahob system [Zee et al. 2008] features extensive use of sets for reasoning about data structures, using both automated and interactive theorem proving. Many of the specifications use a variable, $contents$, that holds the set of internal nodes

of a data structure and is defined using reachability. This is used not only in pre/post-conditions, but also in frame conditions. Many of the procedure specifications in the Jahob distribution include "modifies *contents*", which appears to mean that any field of any element of *contents* may be written. (Finer-grained frame conditions, which seem likely to be needed for client code, can be expressed using quantified postconditions, as the assertion language is that of higher-order logic.) The example modifies specifications resemble the use of JML's "model fields", which are given definitions in terms of concrete state: in a frame condition, mention of a model field is interpreted to allow modification of that part of the state. Making this precise and usable has proved to be difficult [Leino and Müller 2006; Leino and Nelson 2002].

The use of ghost state to encode inductive properties without induction has been fruitful in verifications using SMT solvers (e.g., Cohen et al. [2009], Hawblitzel and Petrank [2009], Zee et al. [2008]). The VeriCool verifier Smans et al. [2012] is SMT-based, as is the Chalice system [Leino and Müller 2009] that uses permissions for verification of concurrent code. In contrast to VeriCool's use of pure method calls in frame conditions, the Dafny tool [Leino 2010] uses ghost fields and ghost variables to specify effects. Dafny provides a parametric type $\mathsf{set}\langle K\rangle$ to denote sets of references of type K. The sets are therefore akin to typed regions and are used in modifies clauses: for example, $\mathsf{modifies}\ s$ denotes that any field of any reference in s may be modified. Thus, Dafny's effects are in the form we write as G`any in region logic. The Boogie tool [Barnett et al. 2005] has been used for experiments with region logic specifications of the Observer [Banerjee et al. 2008a] and Composite [Rosenberg et al. 2010] patterns. Boogie and the tools mentioned above interface with the Z3 SMT solver [de Moura and Bjørner 2008] to decide the generated verification conditions.

Our VERL tool described in Rosenberg [2011, Chapter 3] is based on Dafny and translates a program annotated with region logic specifications into the intermediate specification language Boogie2 [Leino 2008]. It supports the full generality of region logic assertions and effects: In contrast to Dafny, VERL supports effects of the form $\mathsf{wr}\ G$`f for any reference-typed, integer-typed or region-typed field f. VERL uses a syntax-directed algorithm to compute read effects of quantified formulas and expressions [Rosenberg 2011, Figures 3.18, 3.19] while Dafny requires programmer-annotated read effects. Based on these annotations, Dafny generates frame axioms for every declared function. These axioms are essentially instances of the frame validity condition (Definition 6.1 or its two-way counterpart in Remark 6.6). The frame "axioms" are a proof obligation to be checked, and are then provided as axioms for use by the verifier in proving assertions and postconditions. By contrast, VERL provides "localized framing" [Rosenberg et al. 2010] via a *preserves* pragma with which the programmer can indicate where the frame rule could be used and the expressions whose values must be proven to be preserved across a command.

The preserves pragma, $\mathsf{preserves}\ Q\ C$, causes VERL to infer a read effect for a framed formula Q using the syntax-directed part of the framing rules (of Figure 15). So the inferred read effects are correct by construction and satisfy frame validity (Definition 6.1). Instead of computing a separator and checking the condition $P \wedge Q \Rightarrow \delta \between \varepsilon$ in FRAME, VERL interprets the preserves pragma by directly checking that the command C does not interfere with the (inferred) read effects δ of Q. In other words, VERL directly checks that the pre-state and the post-state agree modulo δ. This allows VERL to *assume* that the meaning of Q remains the same before and after execution of C. (By contrast, Dafny would assert that, and try to prove it using frame axioms.) In our experiments [Rosenberg et al. 2010], the inferred read effects are always sufficient. However VERL could be extended to allow a programmer to declare finer read effects for Q. Validity of such a frame would be checked just like Dafny's checking of frame axioms.

Berdine et al. [2005a] implement the Smallfoot verifier to automate verification of programs whose assertions are in the symbolic heaps fragment [Berdine et al. 2005b] of separation logic. Smallfoot uses symbolic execution of formulas and encodes the effects of each command in the symbolic heap and (variable) store. The verifier, VeriFast [Jacobs et al. 2010], has been developed to verify single-threaded and multithreaded C and Java programs annotated with separation logic assertions. Like the tools inspired by Boogie, VeriFast too interfaces with the Z3 SMT solver to discharge verification conditions.

KeY [Beckert et al. 2007] is a highly automated interactive verification tool, based on a dynamic logic [Harel et al. 2000] and targeted at the JavaCard dialect of Java. The annotation language includes frame conditions for loops. Because regions are simply sets of references, the JML assertion language already includes almost all features of region logic assertions, and region images can be encoded using quantifiers. An extension of JML with dynamic frames has been designed and implemented in the KeY tool [Schmitt et al. 2010]. They include read effects in method specifications, to support reasoning about model fields and calls of pure methods in specifications. This extension seems to have been developed independently from our work but the treatment of frame conditions is quite similar.

Decision Procedures. Verifiers like Boogie, VERL, VeriCool, Dafny rely on axioms to reason about assertions. For example to reason about sets of references they must rely on axioms for sets. Such axioms are typically universally quantified. Axiomatization of region images of the form $G`f$ results in a $\forall\exists$ quantifier prefix. It is well known that reasoning about quantifiers is inherently hard and SMT solvers must rely on heuristics on how to instantiate quantifiers. A more complete approach is to use a dedicated decision procedure. For example, Jahob uses the BAPA procedure (Boolean Algebra and Presburger Arithmetic) to reason about sets and their cardinalities in its assertion language [Kuncak and Rinard 2007]. Suter et al. [2011] recently implemented an extension of the decision procedure for BAPA in Z3. Rosenberg's PhD dissertation investigates decision procedures for quantifier free region logic assertions and has developed a complete, tableau-based decision procedure for such assertions [Rosenberg 2011, Chapter 4]. This decision procedure extends the tableau-based decision procedure for the quantifier free language 2LST [Zarba 2003]—a two-sorted language of sets of elements with any number of constant, function and predicate symbols over the element sort in some theory T, provided as a parameter. Region logic's assertion language can be viewed as an extension of 2LST with a universal set (denoted by alloc) and images.

Miscellaneous Issues. We have not investigated completeness of region logic, except informally in the course of designing the rules and applying them in proofs. There are a number of considerations which make it questionable what is a good formulation of completeness. Pierik and de Boer [2005a] consider modular completeness with respect to given specifications of methods; their focus is subtyping. Pierik [2006] proves completeness of his logic for a language similar to ours, and Apt et al. [2012] prove completeness of their logic [Apt et al. 2009] by an elegant reduction of objects and inheritance to arrays and recursive procedures. These logics lack explicit frame conditions and other features of interest here. On the other hand, these works include logics in proof outline form. For direct use in interactive verification, proof outlines may be more convenient than trees of judgments.

We have not modeled garbage collection in this article. Consider a complete program $x := \mathsf{new}\ K; x := \mathsf{null}$. The postcondition of the program establishes that there exists no object of type K, at least once garbage is collected, but in our semantics there does exist an object of type K. A more practical example would be where an invariant

of the form $\forall x : K \in \mathsf{alloc} \cdot P$ is maintained; once a K-object becomes unreachable, what happens to the invariant? It might well not hold, for example, if it relates to other state that is updated. But in our experience, useful invariants quantify over some designated pool of objects, from which an object is likely to be removed before it becomes unreachable. Garbage collection has been studied in connection with separation logic [Calcagno et al. 2003].

10. CONCLUSION

The primary concern of this work is local and modular reasoning about object-based programs. In Part I, we have focused on local reasoning by making footprints explicit in frame conditions and expressing them as sets of references paired with their fields. The sets of references, termed regions, can appear as annotations in program text by way of mutable auxiliary fields and variables, often called ghost state. A practical upshot of reasoning about regions as ghost state is that this provides enough flexibility to encode programming disciplines such as ownership as well as disciplines embodied in various design patterns that organize the heap in uniform or ad hoc ways.

We have developed region logic, a Hoare logic with frame conditions, that allows local reasoning for object-based programs to be carried out with the help of region logic's FRAME rule. Assertions in region logic are formulas in first-order logic together with sets: in particular one can express quantified formulas in which the bound of the quantification is over a designated set of allocated references. The logic also features specification-only variables, for which there is a substitution rule; this is important for completeness in connection with procedures (which are added in Part II). The logic is amenable to automation using SMT provers as evidenced by our SMT-based verifier, VERL, and its application to case studies that involve challenging design patterns. Separately, Rosenberg's PhD dissertation [2011] has developed decision procedures for quantifier-free region logic formulas.

VERL incorporates some of the features that we omitted from our formalization but which are clearly desirable: data groups [Leino et al. 2002], typed regions (cf. Remark 3.3), region comprehension (cf. Remark 6.3), sequences, specification statements, for example, foreach known as "bulk updates" in Dafny. Other features worthy of theoretical study as well as inclusion in a tool are ghost parameters and injective images.

In ongoing work we are developing a relational version of region logic, along the lines of other relational Hoare logics like that of Benton [2004]. In the course of this work, we developed an extension of the logic that includes read effects for commands; this is straightforward. We also hope to gain theoretical and practical insight by finding embeddings between separation logic and region logic (for which Parkinson and Summers [2011] may be relevant). Other obvious topics for investigation are (relative) completeness of the logic, modeling of garbage collection, and extension of the logic to concurrent programs.

Information hiding is the focus of Part II of this article. The approach of Kassios, which we have explored here, was motivated by the hope that disciplines like ownership for the hiding of invariants could be embodied as reasoning patterns rather than hard-coded in a logic or verification system. Taken literally, Kassios' approach requires that the general reasoning behind soundness of a discipline like Boogie [Leino and Müller 2004] be re-produced as part of the verification for any particular program that uses the discipline. But the spirit of Kassios' work is that notions like ownership would be formalized as theorems that could be used in reasoning about many programs. One of our goals is to provide a more explicit framework in which disciplines can be formalized and validated as such, at the level of program annotations and without direct recourse to the underlying semantics. The goal motivates Part II of this article.

ACKNOWLEDGMENTS

Many people helped with suggestions and encouragement, including Mike Barnett, Sophia Drossopoulou, Manuel Fähndrich, Bart Jacobs, Rustan Leino, Peter Müller, Peter O'Hearn, Matthew Parkinson, Wolfram Schulte, Jan Smans, Alexander Summers, David Walker, Hongseok Yang, and several anonymous reviewers.

REFERENCES

Aldrich, J. and Chambers, C. 2004. Ownership domains: Separating aliasing policy from mechanism. In *Proceedings of the European Conference on Object-Oriented Programming*. 1–25.

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. 1981. Ten years of Hoare's logic, a survey, part I. *ACM Trans. Program. Lang. Syst. 3,* 4, 431–483.

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.

Banerjee, A. and Naumann, D. A. 2013. Local reasoning for global invariants, part II: Dynamic boundaries. *J. ACM*. To appear.

Banerjee, A., Barnett, M., and Naumann, D. A. 2008a. Boogie meets regions: A verification experience report. In *Verified Software: Theories, Tools, Experiments*. Lecture Notes in Computer Science, vol. 5295, 177–191.

Banerjee, A., Naumann, D. A., and Rosenberg, S. 2008b. Expressive declassification policies and modular static enforcement. In *Proceedings of the IEEE Symposium on Security and Privacy*. 339–353.

Banerjee, A., Naumann, D. A., and Rosenberg, S. 2008c. Regional logic for local reasoning about global invariants. In *Proceedings of the European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science, vol. 5142, 387–411.

Barnett, M., DeLine, R., Fähndrich, M., Leino, K. R. M., and Schulte, W. 2004. Verification of object-oriented programs with invariants. *J. Object Technol. 3,* 6, (Special Issue: ECOOP 2003 Workshop on Formal Techniques for Java-like Programs) 27–56.

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 2004), 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.

Benton, N. 2004. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. 14–25.

Berdine, J., Calcagno, C., and O'Hearn, P. W. 2005a. Smallfoot: Modular automatic assertion checking with separation logic. In *Formal Methods for Components and Objects*. Lecture Notes in Computer Science, vol. 4111, 115–137.

Berdine, J., Calcagno, C., and O'Hearn, P. W. 2005b. Symbolic execution with separation logic. In *Proceedings of the Asian Symposium on Programming Languages and Systems*. Lecture Notes in Computer Science, vol. 3780, 52–68.

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.

Boyapati, C., Liskov, B., and Shrira, L. 2003. Ownership types for object encapsulation. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. 213–223.

Calcagno, C., O'Hearn, P., and Bornat, R. 2003. Program logic and equivalence in the presence of garbage collection. *Theoret. Comput. Sci. 298,* 3, 557–581.

Calcagno, C., Distefano, D., O'Hearn, P. W., and Yang, H. 2011. Compositional shape analysis by means of bi-abduction. *J. ACM 58,* 6.

Clarke, D. and Drossopoulou, S. 2002. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications*. 292–310.

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 *Theorem Proving in Higher Order Logics*. Lecture Notes in Computer Science, vol. 5674, 23–42.

de Moura, L. M. and Bjørner, N. 2008. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science, vol. 4963, 337–340.

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.

Drossopoulou, S. and Smith, M. 2003. Cheaper reasoning with ownership types. In *Proceedings of the International Workshop on Aliasing, Confinement and Ownership*.

Feng, X., Shao, Z., Guo, Y., and Dong, Y. 2008. Combining domain-specific and foundational logics to verify complete software systems. In *Verified Software: Theories, Tools, Experiments*. Lecture Notes in Computer Science, vol. 5295, 54–69.

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 Language Design and Implementation*. 234–245.

Floyd, R. W. 1967. Assigning meanings to programs. In *Proceedings of the Symposia in Applied Mathematics 19*. American Mathematical Society, 19–32.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Gorelick, G. 1975. A complete axiomatic system for proving assertions about recursive and nonrecursive programs. Tech. rep. 75, Department of Computer Science, University Toronto.

Harel, D. 1979. *First-Order Dynamic Logic*. Lecture Notes in Computer Science, vol. 68, Springer.

Harel, D., Kozen, D., and Tiuryn, J. 2000. *Dynamic Logic*. MIT Press.

Hawblitzel, C. and Petrank, E. 2009. Automated verification of practical garbage collectors. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. 441–453.

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

Hofmann, M. and Pavlova, M. 2008. Elimination of ghost variables in program logics. In *Trustworthy Global Computing 2007*. Lecture Notes in Computer Science, vol. 4912, 1–20.

Jacobs, B., Smans, J., and Piessens, F. 2010. A quick tour of the VeriFast program verifier. In *Proceedings of the Asian Symposium on Programming Languages and Systems*. Lecture Notes in Computer Science, vol. 6461, 304–311. `http://people.cs.kuleuven.be/~bart.jacobs/verifast/`.

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. Aspects Comput. 23,* 3, 267–288.

Kuncak, V. and Rinard, M. C. 2007. Towards efficient satisfiability checking for Boolean algebra with Presburger arithmetic. In *Proceedings of the International Conference on Automated Deduction*. Lecture Notes in Computer Science, vol. 4603, 215–230.

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

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 *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. Aspects Comput. 19,* 2, 159–189.

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.

Leino, K. R. M. 1998. Data groups: Specifying the modification of extended state. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications*. 144–153.

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. 2010. DAFNY: An automatic program verifier for functional correctness. In *Proceedings of the International Conference on Logic for Programming Artificial Intelligence and Reasoning*. E. M. Clarke and A. Voronkov Eds., Lecture Notes in Computer Science, 348–370.

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 Notes in Computer Science, vol. 3924, 115–130.

Leino, K. R. M. and Müller, P. 2009. A basis for verifying multi-threaded programs. In *Proceedings of the European Symposium on Programming Languages and Systems*. Lecture Notes in Computer Science, vol. 5502, 378–393.

Leino, K. R. M. and Nelson, G. 2002. Data abstraction and information hiding. *ACM Trans. Program. Lang. Syst. 24,* 5, 491–553.

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 Languages, 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.

Marron, M., Méndez-Lojo, M., Hermenegildo, M. V., Stefanovic, D., and Kapur, D. 2008. Sharing analysis of arrays, collections, and recursive structures. In *Proceedings of the ACM Workshop on Program Analysis for Software Tools and Engineering*. 43–49.

Müller, P. 2002. *Modular Specification and Verification of Object-Oriented Programs*. Lecture Notes in Computer Science, vol. 2262.

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., 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. 2007. Observational purity and encapsulation. *Theoret. Comput. Sci. 376,* 3, 205–224.

Naumann, D. A. and Banerjee, A. 2010. Dynamic boundaries: Information hiding by second order framing with first order assertions. In *Proceedings of the Programming Languages and Systems, European Symposium on Programming*. 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.

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.

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 Summers, A. J. 2011. The relationship between separation logic and implicit dynamic frames. In *Proceedings of the European Symposium on Programming Languages and Systems*. Lecture Notes in Computer Science, vol. 6602, 439–458.

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. 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.

Reynolds, J. C. 1981. *The Craft of Programming*. Prentice-Hall.

Reynolds, J. C. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings of the IEEE Symposium on Logic in Computer Science*. 55–74.

Robby, Aldrich J. 2008. *Proceedings of the 7th International Workshop on Specification and Verification of Component Systems (SAVCBS)*. Tech. rep. CS-TR-08-07, School of Electrical Engineering and Computer Science, University of Central Florida.

Rosenberg, S. 2011. Region logic: Local reasoning for Java programs and its automation. Ph.D. thesis, Stevens Institute of Technology.

Rosenberg, S., Banerjee, A., and Naumann, D. A. 2010. Local reasoning and dynamic framing for the composite pattern and its clients. In *Verified Software: Theories, Tools, Experiments*. Lecture Notes in Computer Science, vol. 6217, 183–198. `http://www.cs.stevens.edu/~naumann/pub/VERL/`.

Rudich, A., Darvas, A., and Müller, P. 2008. Checking well-formedness of pure-method specifications. In *Formal Methods*. Lecture Notes in Computer Science, vol. 5014, 68–83.

Schlesinger, C., Pattabiraman, K., Swamy, N., Walker, D., and Zorn, B. 2011. Modular protections against non-control data attacks. In *Proceedings of the IEEE Computer Security Foundations Symposium*.

Schmitt, P. H., Ulbrich, M., and Weiß, B. 2010. Dynamic frames in Java dynamic logic. In *Formal Verification of Object-Oriented Software (FoVeOOS)* (*Revised Selected Papers*). Lecture Notes in Computer Science, vol. 6528, 138–152.

Smans, J., Jacobs, B., and Piessens, F. 2012. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst. 34,* 1, 2:1–2:58.

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. *Form. Aspects Comput. 22,* 3–4, 423–457.

Suter, P., Steiger, R., and Kuncak, V. 2011. Sets with cardinality constraints in satisfiability modulo theories. In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation*. Lecture Notes in Computer Science, vol. 6538, 403–418.

Zarba, C. G. 2003. Combining sets with elements. In *Verification: Theory and Practice*. Lecture Notes in Computer Science, vol. 2772, 762–782.

Zee, K., Kuncak, V., and Rinard, M. C. 2008. Full functional verification of linked data structures. In *Proceedings of the ACM Conference on Programming Languages Design and Implementation*. 349–361.

Received July 2011; revised November 2012; accepted March 2013