Regional Logic
for Local Reasoning about Global Invariants

Anindya Banerjee[1,*], David A. Naumann[2,**], and Stan Rosenberg[2,**]

[1] Kansas State University, Manhattan KS 66506 USA and
Microsoft Research, Redmond WA 98052 USA
[2] Stevens Institute of Technology, Hoboken NJ 07030 USA

Abstract. Shared mutable objects pose grave challenges in reasoning, especially for data abstraction and modularity. This paper presents a novel logic for error-avoiding partial correctness of programs featuring shared mutable objects. 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 modifies clause specifies write, read, and allocation effects using region expressions; this supports effect masking and 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: disciplines such as ownership are expressible but not hard-wired in the logic.

1 Introduction

The potential for interference between supposedly independent program phrases or components due to shared mutable objects is the bane of formal reasoning and static analysis of software. This paper charts new territory, combining two simple and well known ideas —regions and ghost state— in a new way. We formulate a logic that needs only classical, first-order assertions, though inductively defined predicates are compatible. The key novelty is "modifies" specifications expressed in terms of state-dependent region expressions. Together with judicious static analysis of the "footprints" of formulas, this makes it possible to achieve the kinds of modularity associated with ownership methodologies and separation logic, in a flexible way that is compatible with widely used specification languages and tools.

Various notions of regions have been used in static analysis to abstract sets of objects of interest [28]. Separation logic [23] owes its success in specifying and verifying pointer algorithms at least in part to its ability to manifest the "footprint" or region of heap relevant to a particular predicate (and thereby the footprint of a command). At a coarser level, ownership systems and separation logic ideas have been critical to advances in data abstraction [11,2], especially for object invariants [18,5].

In this paper, instead of abstracting from regions and expressing separation via logical connectives or ownership types, we make regions explicit in a way similar to work

---

of Amtoft et al [1]. Most importantly, we follow Kassios [14] in using regions to directly represent footprints. We augment a Java-like language with type **rgn** ranging over finite sets of (allocated) references. Following Kassios, we instrument programs with assignments to ghost variables and fields, so assertions can refer explicitly to regions. Whereas Kassios works in the setting of a relational refinement theory and higher order logic, we develop a Hoare logic using first class regions in the "modifies" clause, often the most useful part of a program specification. Asserting the disjointness of regions helps delimit effects and facilitate heap-local reasoning.

It is no surprise that it is possible to reason in terms close to the semantic model [8]. If one's aim is to prove functional correctness of, say, a garbage collector then at the very least, the specification involves reachability, inductive definitions, quantification over paths, etc. But to specify and prove weaker properties, e.g., that an application program does not stray beyond its intended resources, what we achieve is promising. Without the need for inductive predicates or quantification over predicates to hide all but their footprint, we reason directly in terms of footprints. In particular we get "frame rules" that account for modular reasoning about representation invariants.

Notions like ownership [11] support encapsulation of state on which a single object's invariant depends. A precursor to our work is the use of ghost state to encode ownership [16,22] in a way that allows transfer of objects between clients and abstractions (as in low level memory management and higher level OO design patterns like connection pools and layered I/O abstractions). Unlike ownership type systems or programming disciplines, and unlike static analyses using regions, we avoid commitment to a fixed use of regions. On the contrary, regions as ghost state can encode such disciplines but can also combine them in uniform or ad hoc ways.

A benefit of treating regions as ghost state is that it can be done using first-order specification languages based on classical logic with modest use of set theory. Thus it fits with mostly-automated tools based on verification condition generation and it fits with conventional means of program structuring such as scope-based encapsulation. In this foundational study we expose the issues and formalize the ideas in terms of a simple object-based language and Hoare-style proof system which we prove sound using a standard program semantics. There is a major difficulty: "modifies" specifications using region expressions dependent on mutable state are susceptible to a kind of interference: The effect of a command can alter the meaning of the effect specification of another command! This issue has appeared before, in Kassios' dissertation and in the work of Leino and Nelson [17]; we explicitly focus on modifies specifications and offer a novel and flexible solution.

Our first contribution is the logic: its rules and subsidiary judgements together with proof of soundness. Various subtleties made it difficult to correctly design the details of our logic, but we find most of the rules and soundness proofs to be elegant.

Our second contribution is to show how the logic serves as a basis for encapsulating object invariants and invariants for clusters of objects (peers, friends and beyond). Remarkably, our approach can be formalized by a second order frame rule like that of separation logic. Soundness of the second order frame rule in separation logic is challenging [23,7]. Our version is an admissible rule, but the technical result is the subject of another paper [21]. In this paper, we propose an approach to developing sound

and flexible disciplines for modular reasoning about invariants, inspired by the work of Kassios, the Boogie team, and many others.

Outline. Sec. 2 sketches an example to illustrate features of the logic. Sec. 3 formalizes an illustrative, object-based programming language and Sec. 4 presents the assertion language. Sec. 5 formalizes effects using regions and Sec. 6 gives a static analysis for the separation of a formula from a write effect. Sec. 7 defines correctness statements and gives the proof rules and soundness theorem. Sec. 8 wraps up the running example and Sec. 9 applies the logic to modular reasoning about invariants. Sec. 10 discusses related work. More examples and proofs are in the online appendix [3].

2   A Small Example

We give a step-by-step correctness proof of a command acting on variable x of type *Node*. A *Node* has three fields: *item* of type **int** and *left*, *rt* of type *Node*. The command sets the *item* field of x's *left* node to zero. The precondition is $P \wedge Q$ and the postcondition is Q where

$$
\begin{aligned}
P &\ \widehat{=}\ x \neq \textbf{null} \wedge x.left \in r_1 \wedge x.rt \in r_2 \wedge r_1 \,\#\, r_2 \wedge closed \\
closed &\ \widehat{=}\ r_1\text{-}left \subseteq r_1 \wedge r_1\text{-}rt \subseteq r_1 \wedge r_2\text{-}left \subseteq r_2 \wedge r_2\text{-}rt \subseteq r_2 \\
Q &\ \widehat{=}\ \forall x \colon Node \in r_2 \mid x.item > 0
\end{aligned}
$$

The specification uses two region variables, r_1 and r_2. Precondition P expresses that x is non-null and the object denoted by $x.left$ is in r_1 (and $x.rt$ is in r_2). Furthermore, regions r_1, r_2 are disjoint which in our syntax is denoted by $r_1 \,\#\, r_2$. Regions are finite sets of object references, of any type. Since null is not a reference, $x.left \in r_1$ implies $x.left \neq \textbf{null}$. Formula *closed* says that both r_1 and r_2 are closed under both *left* and *rt*: If $o \in r_1$ and $o.left \neq \textbf{null}$ then $o.left \in r_1$, etc.

In general, for region expressions G, G' and field name f, the formula $G\text{-}f \subseteq G'$ says that the f-image of region G is contained in G'. For reasons discussed later, "$G\text{-}f$" is not a region expression.

In summary, the precondition P states that the left "subtree" of x is in r_1 and the right "subtree" of x is in r_2. (But these "subtrees" need not be trees, nor even dags.)

The formula Q plays the role of an invariant. It says that for any node o in r_2, o's item field is positive. Finally, since the command writes to the *item* field, we will show that its write effect is at most the *item* field of objects residing in r_1, denoted by $\textbf{wr}\, r_1\text{-}item$. The complete correctness statement is

$$
\{\, P \wedge Q \,\}\ \textbf{var}\ y \colon Node\ \textbf{in}\ y := x.left;\, y.item := 0\ \textbf{end}\ \{\, Q \,\}\, [\textbf{wr}\, r_1\text{-}item] \qquad (1)
$$

One can prove a stronger postcondition but this is enough for expository purposes.

We now turn to the proof. Here is a specification for the first assignment:

$$
\{\, x \neq \textbf{null} \,\}\ y := x.left\ \{\, y = x.left \,\}\, [\textbf{wr}\, y]
$$

This is a *small* specification in that it only mentions entities that are relevant to the assignment. Since $y := x.left$ only writes y, and since neither P nor Q mention y, their

truth value is not changed by the assignment. So we conjoin $P \wedge Q$ to both pre- and post-condition:

$$\{P \wedge Q\} \; y := x.left \; \{y = x.left \wedge P \wedge Q\} \, [\mathbf{wr}\, y]$$

This step is an instance of the *Frame* rule (Fig. 9), cf. [23]. We will be more precise later, but for now we say that $x, r_1, r_2, \langle x \rangle.left, r_1.left, r_2.left, \langle x \rangle.rt, r_1.rt, r_2.rt$ constitute a "frame" of P because modifications to the frame may affect the truth value of P, but no other modifications can. Notice that y, the variable modified by the command, is not in the frame. Similarly, $r_2, r_2.item$ constitute the frame of Q. Again, y is absent. This separation between the write effect of the command and the frames licenses conjoining $P \wedge Q$ to the pre- and postconditions above.

Recall that $x.left \in r_1$ implies $x.left \neq \mathbf{null}$ which together with $y = x.left$ implies that $y \neq \mathbf{null}$ and $y \in r_1$. The last assertion implies a weaker one: $\langle y \rangle \subseteq r_1$, where $\langle y \rangle$ denotes the singleton region iff y is non-null, and otherwise the empty region. Thus by the standard rule of Consequence we get

$$\{P \wedge Q\} \; y := x.left \; \{P \wedge y \neq \mathbf{null} \wedge \langle y \rangle \subseteq r_1 \wedge Q\} \, [\mathbf{wr}\, y] \qquad (2)$$

Here is a small specification for $y.item := 0$:

$$\{y \neq \mathbf{null}\} \; y.item := 0 \; \{y.item = 0\} \, [\mathbf{wr}\, \langle y \rangle.item]$$

The write effect, $\mathbf{wr}\, \langle y \rangle.item$, records the fact that the *item* field of the singleton region, $\langle y \rangle$, may have changed. Now we would like to conjoin Q to pre/post of the above specification, so it could be sequenced with (2). However, doing so appears to be unsound, since Q reads *item* whereas the command writes *item*. We shall refine the above specification to obtain a stronger precondition, which will imply that Q is separated from the write. First, we use the Frame rule to conjoin $P \wedge \langle y \rangle \subseteq r_1$, whose frame is clearly separated from $\langle y \rangle.item$ because a write to $y.item$ leaves y, r_1, and r_2 unchanged and *item* is not in the frame of P. This yields

$$\{y \neq \mathbf{null} \wedge P \wedge \langle y \rangle \subseteq r_1\} \; y.item := 0 \; \{y.item = 0 \wedge P \wedge \langle y \rangle \subseteq r_1\} \, [\mathbf{wr}\, \langle y \rangle.item]$$

Now, to apply the Frame rule to Q we need that $r_2.item$ is separated from the write of $\langle y \rangle.item$. It suffices to show that $\langle y \rangle$ is a region disjoint from r_2. This follows from $\langle y \rangle \subseteq r_1$ and the fact that $P \Rightarrow r_1 \# r_2$. So Frame yields

$$\{y \neq \mathbf{null} \wedge P \wedge \langle y \rangle \subseteq r_1 \wedge Q\} \; y.item := 0 \; \{y.item = 0 \wedge P \wedge \langle y \rangle \subseteq r_1 \wedge Q\} \, [\mathbf{wr}\, \langle y \rangle.item]$$

whence by the rule of Consequence we obtain

$$\{y \neq \mathbf{null} \wedge P \wedge \langle y \rangle \subseteq r_1 \wedge Q\} \; y.item := 0 \; \{Q\} \, [\mathbf{wr}\, \langle y \rangle.item] \qquad (3)$$

Now we are ready to sequence (2) followed by (3). But what should the effects be? Simply unioning the effects $\mathbf{wr}\, y$ and $\mathbf{wr}\, \langle y \rangle.item$ is unsound because effects are interpreted in the *pre-state*: the y in $\mathbf{wr}\, y.item$ does not have the same meaning as in the pre-state of the sequence, because y is modified by the first command. So the write to

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

$$T ::= \mathbf{int} \mid K \mid \mathbf{rgn}$$

$$E ::= x \mid c \mid \mathbf{null} \mid E \oplus E \qquad \text{where } c \text{ is in } \mathbb{Z} \text{ and } \oplus \text{ is in } \{=, +, -, *, >, \ldots\}$$

$$G ::= x \mid x.f \mid \langle E \rangle \mid \mathbf{emp} \mid \mathbf{alloc} \mid G \otimes G \qquad \text{where } \otimes \text{ is in } \{\cup, \cap, -\}$$

$$F ::= E \mid G$$

$$C ::= x := F \mid x := \mathbf{new}\, K \mid x := x.f \mid x.f := F$$
$$\quad \mid \mathbf{if}\, x\, \mathbf{then}\, C\, \mathbf{else}\, C \mid \mathbf{while}\, x\, \mathbf{do}\, C \mid C\,;\, C \mid \mathbf{var}\, x : T\, \mathbf{in}\, C\, \mathbf{end}$$

Fig. 1. Programming language. We confuse category names with typical elements (e.g., T).

the *item* field must be recorded by some other expression. Precondition $\langle y \rangle \subseteq r_1$ of (3) implies that $\mathbf{wr}\,\langle y \rangle \text{\textbullet}\mathit{item}$ is a sub-effect of $\mathbf{wr}\, r_1 \text{\textbullet}\mathit{item}$, so by weakening we obtain

$$\{\, y \neq \mathbf{null} \wedge P \wedge \langle y \rangle \subseteq r_1 \wedge Q \,\}\ y.item := 0\ \{\, Q \,\}\ [\mathbf{wr}\, r_1 \text{\textbullet}\mathit{item}]$$

Now we can apply the rule of sequential composition to obtain

$$\{\, P \wedge Q \,\}\ y := x.left; y.item := 0\ \{\, Q \,\}\ [\mathbf{wr}\, y, \mathbf{wr}\, r_1 \text{\textbullet}\mathit{item}]$$

The rule for local blocks lets us remove the effect $\mathbf{wr}\, y$ and conclude the proof of (1).

3   Programming Language

This section presents an illustrative language for which we formalize the programming logic. A program consists of a command C in the context of some class declarations. The grammar for commands etc. is in Fig. 1. A class declaration $\mathbf{class}\, K\, \{\, \overline{T}\, \overline{f}\, \}$ introduces a type name K; values of this type are $\mathbf{null}$ and references to mutable objects with typed fields $\overline{f} : \overline{T}$. Here and throughout, identifiers with an overline range over lists. As in Java, an assignment $x := y.f$ implicitly dereferences the value in y and reads field f in the heap. Equality test, written $=$, is for reference equality.

In addition to $\mathbf{int}$ and reference types, there is type $\mathbf{rgn}$ with values ranging over finite sets of references (excluding null). Region expressions include set operations like subtraction $(-)$. Region expressions cannot influence control flow or the value of non-region fields/variables, so they can only serve as ghosts for reasoning.

Ordinary expressions (E in Fig. 1) do not depend on the heap: $y.f$ is not an expression but rather part of the primitive command $x := y.f$ for reading a field. There is also a primitive formula for reading a field (see Fig. 3). This restriction serves, as in separation logic, to simplify rules for reasoning about assignments. We also gain some simplification in the framing rules to come (Fig. 7). Region expressions (G in Fig. 1) do include a form that reads a single step into the heap, namely $x.g$ when g has type $\mathbf{rgn}$ (in which case $x.g.f$ is not well formed). The form $x.g$ is essential for our purposes, but allowing multi-step heap dependence would cause complications, e.g., in the framing rules. This is why the form "$G \text{\textbullet} f$" which appears in effects and assertions is not a region expression. Of course syntax sugars can be added for practical purposes, with derived rules.

There is an ambient class table comprising a well formed collection of class declarations. We write fields(K) for the field declarations $\overline{f} : \overline{T}$ of class K. The judgement $\Gamma \vdash F : T$ says that region or ordinary expression F has type T in context Γ that assigns types to variable names. Similarly, $\Gamma \vdash C$ says C is a well formed command. For programs we assume the standard rules that prevent "field not defined" errors. For brevity we omit a boolean type. The guard for an if- or while-command has type **int**; the semantics interprets any non-zero value as true.

We omit most of the rules since they are straightforward, but note that **int** is separated from reference types: there is no pointer arithmetic. The typing rules make some distinctions between region expressions G and those of other type. The rule for singleton region $\langle E \rangle$ enforces that E is of reference type. Field dereferencing in expressions is allowed only for fields of type **rgn**, as per: $\dfrac{(g : \textbf{rgn}) \in \mathsf{fields}(K)}{\Gamma, x : K \vdash x.g : \textbf{rgn}}$. Recall that metavariable K ranges over class names; only classes have fields. Here and throughout the paper, rules are only permitted to be instantiated when the consequent as well as the antecedent are well formed. For example, the rule for context extension is $\dfrac{\Gamma \vdash F : T'}{\Gamma, x : T \vdash F : T'}$ and it cannot be used with x that is in dom(Γ), because the comma in $\Gamma, x : T$ denotes the union of disjoint partial functions.

The concrete syntax $x := y.f$ has an ambiguity that is resolved by typing: If $f : \textbf{rgn}$ then it is read as $x := F$ with $F \equiv y.f$ —and in case y is null, the value is the empty region, for convenience in reasoning. If f is any other type, it is read as a primitive command —and of course there is a runtime error if y is null.

Semantics. We use a straightforward denotational semantics where commands denote deterministic state transformers, which fits well with pre/post specifications. The details are adapted and simplified from a machine-checked semantics of CoreJML encoded in PVS and including hooks to add types like **rgn** and operations on ghost state [15].

We are given a set, Ref, of reference values and a distinguished value, null, not in Ref or $\mathbb{Z}$ or 2^{Ref}. The values denoted by a reference type K include null as well as all references that have been allocated for objects of type K, and the values of type **rgn** are finite sets of allocated references (of any type).

A *state* for context Γ has the form (r, h, s) where: r is a *ref context*, i.e., a partial function mapping the allocated references to their types; h is a *heap* that maps each allocated reference to its object state (i.e., map from field names to values); and s is a *store* that maps each variable x in Γ to its value. Throughout the paper, states are assumed to be well typed and have no dangling references. The semantics is parameterized on the allocator, i.e., a deterministic function of the state that yields fresh references but is otherwise arbitrary.

Following separation logic and program verifiers like ESC/Java, correctness judgements specify error-free partial correctness. So we use a denotational semantics in which $[\![\Gamma \vdash C]\!]\sigma$, for Γ-state σ, is either $\Uparrow$ (fault), $\bot$ (divergence), or a Γ-state σ' (normal termination). The only faults are null dereference, since we consider programs that satisfy usual Java-style typing rules and therefore there are no dangling references (and we assume the arithmetic operators are error-avoiding, to avoid complications about definedness). The compound commands like sequence and loops are strict in $\Uparrow$ as well

$$
\begin{aligned}
[\![y]\!]\sigma &= \sigma(y) & [\![G_1 \cup G_2]\!]\sigma &= [\![G_1]\!]\sigma \cup [\![G_2]\!]\sigma \\
[\![x.g]\!]\sigma &= \sigma(x.g) \text{ if } \sigma(x) \neq \mathsf{null}, \text{ else } \varnothing & [\![G_1 \cap G_2]\!]\sigma &= [\![G_1]\!]\sigma \cap [\![G_2]\!]\sigma \\
[\![\langle E \rangle]\!]\sigma &= \{\, [\![E]\!]\sigma \,\} \text{ if } [\![E]\!]\sigma \neq \mathsf{null}, \text{ else } \varnothing & [\![G_1 - G_2]\!]\sigma &= [\![G_1]\!]\sigma - [\![G_2]\!]\sigma \\
[\![\mathbf{alloc}]\!]\sigma &= \mathsf{alloc}(\sigma) & [\![\mathbf{emp}]\!]\sigma &= \varnothing
\end{aligned}
$$

Fig. 2. Semantics of region expressions (eliding $\Gamma \vdash \ldots : \mathbf{rgn}$). Here y and g have type $\mathbf{rgn}$.

as in $\bot$. The semantics of loops is given by fixpoint as usual, where we order outcomes by $\bot \leq \Cap$ and $\bot \leq \sigma$ for any state σ (but distinct states are incomparable and not comparable with $\Cap$).

We use the following abbreviations, for state $\sigma = (r, h, s)$

$\sigma(x)$ for $s(x)$ —variable lookup

$\sigma(x.f)$ for $h(s(x))(f)$ —field of object referenced by variable x

$\sigma(o.f)$ for $h(o)(f)$ —field of reference value o

$\mathsf{alloc}(\sigma)$ for $\mathsf{dom}(r)$ —set of all allocated references

$\mathsf{type}(o, \sigma)$ for $r(o)$ —type of an allocated reference

$\mathsf{update}(\sigma, x, v)$ for (r, h, s'), where s' overrides s to map x to v

$\mathsf{extend}(\sigma, x, v)$ for (r, h, s'), where s' extends s to map x to value v, for $x \notin \mathsf{dom}(s)$

$\mathsf{update}(\sigma, o.f, v)$ for (r, h', s), where h' overrides h to map field f of o to v.

Here metavariables x, y, z range over variable names, whereas we use o, p, q for elements of Ref. We write $\mathsf{fields}(o, \sigma)$ for $\mathsf{fields}(\mathsf{type}(o, \sigma))$. Less obviously, we write $f \in \mathsf{refields}(o, \sigma)$ to say that f is of some reference type, i.e., there is K such that $(f : K) \in \mathsf{fields}(o, \sigma)$.

Semantics for expressions and commands are routine and omitted. Note that $[\![E]\!]\sigma$ depends on the store but not the heap and is always a value (of appropriate type), never $\Cap$ or $\bot$. As one would expect, $x.f := E$ faults if x is **null**, and the same for $x := y.f$ —except in case f has type $\mathbf{rgn}$. As mentioned earlier, that case is actually parsed as $x := G$ and uses the semantics of region expressions given in Fig. 2.

4   Assertion Language

Fig. 3 gives the grammar for assertions. Quantification is over **int** and reference types only, and in the latter case a bounding region is required as in $(\forall x : K \in \mathbf{alloc} \mid P)$ where the quantification is over all allocated objects as is usual [25,9] and important for certain global invariants. There are also atomic formulas for inclusion and disjointness of regions. The formula $G_1 \cdot f \subseteq G_2$ says that for every object in G_1, if it has field f with non-null value then that value is in G_2. The formula $x.f = E$ says that x is non-null and the value of its f field is E. The semantics is two-valued and classical. So *false* can be defined as $1 = 0$, $\vee$ and $\exists$ by DeMorgan, etc. Another syntax sugar is $E \in G \;\widehat{=}\; E \neq \mathbf{null} \wedge \langle E \rangle \subseteq G$. Officially, we cannot write "$x.f \in G$" because $x.f$ is not an expression (unless $f : \mathbf{rgn}$, but then $\in$ does not make sense). But it is safe to abbreviate $x.f \in G \;\widehat{=}\; x.f \neq \mathbf{null} \wedge \langle x \rangle \cdot f \subseteq G$. Another convenient feature is the ability to refer to all fields, as in $G \cdot \mathsf{any} \subseteq G$; we use it in examples but omit the formalization. Finally, we abbreviate $x \,\mathbf{is}\, K$ for $\mathbf{type}(K, \langle x \rangle)$.

$$P ::= E = E \mid x.f = F \mid G \subseteq G \mid G \,\#\, G \mid \textbf{type}(K, G)$$
$$\mid G\text{\textbullet}f \subseteq G \mid G\text{\textbullet}f \,\#\, G \mid (\forall x : \textbf{int} \mid P) \mid (\forall x : K \in G \mid P) \mid P \wedge P \mid \neg P$$

$$\sigma \models x.f = F \qquad \text{iff } \sigma(x) \neq \text{null and } \sigma(x.f) = [\![F]\!]\sigma$$
$$\sigma \models G_1 \,\#\, G_2 \qquad \text{iff } [\![G_1]\!]\sigma \cap [\![G_2]\!]\sigma = \varnothing$$
$$\sigma \models G_1\text{\textbullet}f \subseteq G_2 \qquad \text{iff } \sigma(o.f) = \text{null or } \sigma(o.f) \in [\![G_2]\!]\sigma$$
$$\text{for all } o \in [\![G_1]\!]\sigma \text{ with } f \in \text{refields}(o, \sigma)$$
$$\sigma \models G_1\text{\textbullet}f \,\#\, G_2 \qquad \text{iff } \sigma(o.f) \notin [\![G_2]\!]\sigma \quad \text{for all } o \in [\![G_1]\!]\sigma \text{ with } f \in \text{refields}(o, \sigma)$$
$$\sigma \models \textbf{type}(K, G) \qquad \text{iff type}(o, \sigma) \leq K \text{ for all } o \in [\![G]\!]\sigma$$
$$\sigma \models^{\Gamma} \forall x : \textbf{int} \mid P \qquad \text{iff extend}(\sigma, x, v) \models^{\Gamma, x : \textbf{int}} P \quad \text{for all } v \in \mathbb{Z}$$
$$\sigma \models^{\Gamma} \forall x : K \in G \mid P \text{ iff extend}(\sigma, x, o) \models^{\Gamma, x : K} P \quad \text{for all } o \text{ in } [\![G]\!]\sigma \text{ with type}(o, \sigma) \leq K$$

Fig. 3. Formulas: grammar and selected semantics. The boolean connectives are standard.

The well-formedness judgement $\Gamma \vdash P$ has straightforward rules, but note:

$$\frac{\Gamma \vdash G : \textbf{rgn} \qquad \Gamma \vdash G' : \textbf{rgn} \qquad (f : K) \in \text{fields}(K')}{\Gamma \vdash G\text{\textbullet}f \subseteq G'} \qquad \qquad \frac{\Gamma, x : K \vdash P \qquad \Gamma \vdash G : \textbf{rgn}}{\Gamma \vdash \forall x : K \in G \mid P}$$

The first rule (like that for $\Gamma \vdash G\text{\textbullet}f \,\#\, G'$) ensures that f is of class type. The second disallows quantification over regions and demands that the bound variable x not appear in the bound, G, of the quantification where G is a region expression. This facilitates framing.

To streamline the treatment of local variables and quantifiers, we assume a hygiene condition: no identifier should occur both bound and free in any context, nor bound more than once.

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 Fig. 3. In most cases we elide Γ since it is unchanged throughout. A formula in context Γ is called *valid* iff it is true in all states.

Example. The recursive predicate $List(o, r)$ defined below expresses that o 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 o and field nxt have type $Observer$.

$$List(o : Observer, \; r : \textbf{rgn}) \;\; \hat{=} \;\; (\langle o \rangle.nxt = \textbf{emp} \Rightarrow r = \langle o \rangle) \wedge$$
$$(\langle o \rangle.nxt \neq \textbf{emp} \Rightarrow o \in r \wedge List(o.nxt, r - \langle o \rangle))$$

Note that in the case that o is null, $\langle o \rangle$ is empty and so is $\langle o \rangle.nxt$. If o is non-null but its nxt field is null, then $\langle o \rangle.nxt$ is empty and $\langle o \rangle$ is the singleton set $\{o\}$.

We do not formalize recursively defined predicates. There is no difficulty with $List$ since its occurrences on the right side are in monotonic positions (with respect to the subset ordering on state-sets); such recursions have least fixpoints in the complete lattice of state-sets. (A small complication is that $List$ has parameters.) Note that "$o.nxt$" is not in the syntax for expressions; we write $List(o.nxt, r - \langle o \rangle)$ to abbreviate the formula $o.nxt \in r \wedge \forall p : Observer \in r \mid p = o.nxt \Rightarrow List(p, r - \langle o \rangle)$.

```
class Subject{                              class Observer{
   Observer obs; int val; rgn O;               Subject sub; int cache;
                                               Observer nxt;
   Subject(){
      self.obs := null; self.val := 0; self.O := emp;}   Observer(Subject s){
                                                  self.sub := s; s.register(self);}
   void register(Observer o){
      self.add(o); o.notify();}               void notify(){
                                                  self.cache := self.sub.get();}
   void update(int n){
      self.val := n; Observer o := self.obs;  int val(){return self.cache;}
      while (o! = null){o.notify(); o := o.nxt;}}   }

   int get(){return self.val;}

   void add(Observer o){
      self.O := self.O∪⟨o⟩;
      o.nxt := self.obs; self.obs := o;
}
```

Fig. 4. Subject/Observer implementation

5   Effects

The "frames" described in Sec. 2 are formalized as read effects. For commands, we focus in this paper on write effects. But the full logic includes read effects for commands; these are useful for reasoning about method calls in assertions as well as for program transformations. An *effect set* is a comma-separated list $\overline{\varepsilon}$ of *effects*, ε, with grammar

$$\varepsilon ::= \mathbf{rd}\, x \mid \mathbf{rd}\, G\text{\textbf{.}}f \mid \mathbf{wr}\, x \mid \mathbf{wr}\, G\text{\textbf{.}}f \mid \mathbf{rd\, alloc} \mid \mathbf{wr\, alloc} \mid \mathbf{fr}\, G$$

The idea is that $\mathbf{rd}\, x$ allows variable x to be read, $\mathbf{rd}\, G\text{.}f$ allows read of the f field of objects in G, $\mathbf{wr}\, x$ allows update of variable x, $\mathbf{wr}\, G\text{.}f$ allows update of the f field of objects in G. The region expression **alloc** is like a variable that holds the set of all allocated references and is automatically updated by the allocator, so $\mathbf{wr\, alloc}$ allows allocation and $\mathbf{rd\, alloc}$ allows dependence on the set of allocated objects. Finally, $\mathbf{fr}\, G$ says that all elements of G in the final state are freshly allocated.

Freshness is used to mask updates to fresh objects in sequences. For example, consider the sequence $x := \mathbf{new}\ Node; x.rt := 0$ in using class $Node$ from Sec. 2, By itself, the field update has effect $\mathbf{wr}\,\langle x\rangle\text{.}rt$. But in the pre-state of the sequence, $\langle x\rangle$ cannot possibly contain the updated object. Indeed, no pre-existing object is updated. In the proof rules, the effect of $x := \mathbf{new}\ Node$ includes $\mathbf{fr}\,\langle x\rangle$ which by the sequence rule (in Fig. 8) annihilates the write effect.

Before delving into the technicalities of effects, we give, in Fig. 4, code that we will use as a running example, $Subject/Observer$. Fig. 5 gives the specifications. Note that $notify$ is called on an observer from within the $update$ method but this results in a callback to the $Subject$'s get method via $\mathbf{self}.sub.get()$.

Method	Pre-condition	Post-condition
Subject()	true	$SubObs(\mathbf{self}, 0)$
register(o)	$o \neq \mathbf{null} \wedge SubObs(\mathbf{self}, val) \wedge o \notin O$	$SubObs(\mathbf{self}, val) \wedge o \in O$
update(n)	$SubObs(\mathbf{self}, val)$	$SubObs(\mathbf{self}, n)$
get	$Sub(\mathbf{self}, val)$	$Sub(\mathbf{self}, val) \wedge res = val$
add(o)	$o \neq \mathbf{null} \wedge Sub(\mathbf{self}, val) \wedge o \notin O$	$Sub(\mathbf{self}, val) \wedge o \in O$
Observer(s)	$SubObs(s, s.val)$	$SubObs(s, s.val) \wedge \mathbf{self} \in s.O$
notify	$Sub(sub, sub.val)$ $\wedge Obs(\mathbf{self}, sub, cache)$	$Sub(sub, sub.val)$ $\wedge Obs(\mathbf{self}, sub, sub.val)$
val	$SubObs(sub, sub.val) \wedge \mathbf{self} \in sub.O$	$SubObs(sub, sub.val) \wedge res = sub.val$

Method	Effects
Subject()	
register(o)	$\mathbf{wr}\, \mathbf{self}.O\cdot nxt, \langle \mathbf{self} \rangle \cdot O, \langle o \rangle \cdot nxt, \langle o \rangle \cdot cache, \langle \mathbf{self} \rangle \cdot obs$
update(n)	$\mathbf{wr}\, \langle \mathbf{self} \rangle \cdot val, \mathbf{self}.O\cdot cache$
get	
add(o)	$\mathbf{wr}\, \mathbf{self}.O\cdot nxt, \langle \mathbf{self} \rangle \cdot O, \langle o \rangle \cdot nxt, \langle \mathbf{self} \rangle \cdot obs$
Observer(s)	$\mathbf{wr}\, s.O\cdot nxt, \langle s \rangle \cdot O, \langle s \rangle \cdot obs$
notify	$\mathbf{wr}\, \langle \mathbf{self} \rangle \cdot cache$
val	

Fig. 5. Specifications for Subject/Observer example based on Parkinson [24]

For the specifications of the methods of Fig. 4, the predicates $SubObs$, Sub, Obs are used (following Parkinson [24]). The predicate $List$ used in Sub is defined in Sec. 4.

$$SubObs(s, v) \mathrel{\widehat{=}} Sub(s, v) \wedge \forall o : Observer \in s.O \mid Obs(o, s, v)$$
$$Sub(s, v) \mathrel{\widehat{=}} s.val = v \wedge List(s.obs, s.O)$$
$$Obs(o, s, v) \mathrel{\widehat{=}} o.cache = v \wedge o.sub = s$$

$SubObs$ is an invariant for the entire aggregate structure comprising an instance of $Subject$ together with its $Observer$s. $SubObs$ holds when the subject's invariant, Sub, holds and for each observer in the subject's list of observers, that observer's invariant, Obs, holds. The invariant $Sub(s, v)$ says that the current internal state of subject s is v and all observers of s are in a list whose nodes lie in region $s.O$. The invariant $Obs(o, s, v)$ says that o is an observer of subject s and that o's view of s's internal state is v.

With the above definitions, the method specifications in Fig. 5 are self-explanatory so we move on to explaining the effects. The effect for $notify$ records that a call to it will result in the writing of an observer's $cache$. The effect for add records that O was written to when an observer o was added to the existing list of observers of a subject. The effect for $register$ takes into account the effects of add and $notify$. 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 constructor for $Subject$ has no effects that need be recorded; we are only concerned with the write effects of pre-existing objects. Similarly, note the absence of effects $\mathbf{wr}\, \langle \mathbf{self} \rangle \cdot sub$, $\mathbf{wr}\, \langle \mathbf{self} \rangle \cdot nxt$ and $\mathbf{wr}\, \langle \mathbf{self} \rangle \cdot cache$ in the effects of the constructor for $Observer$.

Technicalities. Effects must be well formed (wf) for the context Γ in which they occur: **rd** x and **wr** x are wf if $x \in \mathrm{dom}(\Gamma)$; **rd** $G.f$, **wr** $G.f$, and **fr** G are wf if G is wf in Γ. We say σ' *extends* σ provided $\mathrm{alloc}(\sigma) \subseteq \mathrm{alloc}(\sigma')$ and $\mathrm{type}(o, \sigma) = \mathrm{type}(o, \sigma')$ for all $o \in \mathrm{alloc}(\sigma)$. The semantics has the property that σ' extends σ whenever $\sigma' = [\![C]\!]\sigma$.

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

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

Definition 2 (agreement on read effects). Let $\overline{\varepsilon}$ be an effect set and σ, σ' be states such that σ' extends σ. Say that σ and σ' *agree on* $\overline{\varepsilon}$, written $\sigma \sim_{\overline{\varepsilon}} \sigma'$, provided the following hold:

(a) for all **rd** x in $\overline{\varepsilon}$, we have $\sigma(x) = \sigma'(x)$
(b) if **rd alloc** in $\overline{\varepsilon}$ then $\mathrm{alloc}(\sigma) = \mathrm{alloc}(\sigma')$
(c) for all **rd** $G.f$ in $\overline{\varepsilon}$, for all $o \in [\![G]\!]\sigma$ with $f \in \mathrm{fields}(o, \sigma)$, we have $\sigma(o.f) = \sigma'(o.f)$

For Def. 2(c), note that because σ' extends σ, we have $o \in \mathrm{alloc}(\sigma')$ and $\mathrm{type}(o, \sigma) = \mathrm{type}(o, \sigma')$, hence $f \in \mathrm{fields}(o, \sigma')$. But it need not be the case that $o \in [\![G]\!]\sigma'$. Were we to consider **alloc** as a variable, (b) would be subsumed by (a); but it is not an ordinary variable.

Often, as discussed following Eqn. (3) in Sec. 2, we need to subsume an effect by a weaker one when the effect refers to a local variable in a different context. An effect set $\overline{\varepsilon}, \varepsilon$, with ε added to $\overline{\varepsilon}$, allows at least the effects allowed by $\overline{\varepsilon}$. In the case of an effect like **wr** $G.f$ there is also the possibility of more liberal effect **wr** $G'.f$ in case $G \subseteq G'$. Since regions can be state-dependent, inclusions like the above are state-dependent, so we use a judgement $P \vdash \overline{\varepsilon} \leq \overline{\varepsilon}'$ to express that the writes/reads in a bigger effect are more permissive. Subsumption for freshness effects is treated separately, since such effects are interpreted in the post-state. Rules for sub-effecting are defined in Fig. 6. Note that the relation is reflexive, since in the weakening rule $\vdash \overline{\varepsilon} \leq \overline{\varepsilon}, \varepsilon$ one may choose ε to be some element of the set $\overline{\varepsilon}$.

Lemma 1 (write sub-effect). Suppose $P \vdash \overline{\varepsilon}_1 \leq \overline{\varepsilon}_2$ and $\overline{\varepsilon}_1$ allows transition from σ to σ'. If $\sigma \models P$ then $\overline{\varepsilon}_2$ allows transition from σ to σ'.

Lemma 2 (read sub-effect). Suppose $P \vdash \overline{\varepsilon}_1 \leq \overline{\varepsilon}_2$ and σ and σ' agree on $\overline{\varepsilon}_2$. If $\sigma \models P$ then σ, σ' agree on $\overline{\varepsilon}_1$.

$$G_1 \subseteq G_2 \vdash \mathbf{wr}\, G_1{\cdot}f \leq \mathbf{wr}\, G_2{\cdot}f \qquad G_1 \subseteq G_2 \vdash \mathbf{rd}\, G_1{\cdot}f \leq \mathbf{rd}\, G_2{\cdot}f \qquad true \vdash \overline{\varepsilon} \leq \overline{\varepsilon}, \varepsilon$$

$$true \vdash \mathbf{wr}\, G_1{\cdot}f, \mathbf{wr}\, G_2{\cdot}f \lesseqgtr \mathbf{wr}\,(G_1 \cup G_2){\cdot}f \qquad true \vdash \mathbf{rd}\, G_1{\cdot}f, \mathbf{rd}\, G_2{\cdot}f \lesseqgtr \mathbf{rd}\,(G_1 \cup G_2){\cdot}f$$

$$\frac{P \vdash \overline{\varepsilon}_1 \leq \overline{\varepsilon}_2 \qquad P \vdash \overline{\varepsilon}_2 \leq \overline{\varepsilon}_3}{P \vdash \overline{\varepsilon}_1 \leq \overline{\varepsilon}_3} \qquad \frac{P' \Rightarrow P \qquad P \vdash \overline{\varepsilon} \leq \overline{\varepsilon}'}{P' \vdash \overline{\varepsilon} \leq \overline{\varepsilon}'} \qquad \frac{P \vdash \varepsilon_1 \leq \varepsilon_2}{P \vdash \varepsilon_1, \overline{\varepsilon} \leq \varepsilon_2, \overline{\varepsilon}}$$

Fig. 6. Selected sub-effect rules. We write $\lesseqgtr$ to abbreviate two inclusion rules.

6   Framing and Separators

This section defines a judgement, $P \vdash \overline{\varepsilon}$ **frm** P', that says the truth or falsity of predicate P' depends only on the state read according to $\overline{\varepsilon}$, i.e., $\overline{\varepsilon}$ covers the footprint of P' in P-states. This is one of the two critical ingredients in the Frame rule (Sec. 7). The other is the notion of *separator*, which applies to the read effects of a formula and the write effects of a command. Their separator is a conjunction of region disjointness formulas sufficient to ensure that in any transition from state σ to σ' allowed by the write effects, σ, σ' agree on the read effects.

6.1   Framing

First, we define a syntax-directed analysis that computes a precise "footprint" of ordinary expressions, region expressions and primitive assertions. Intuitively, the footprint is all reads needed to evaluate a given expression or primitive assertion whereas the frame of an assertion is an over-approximation of these requisite reads. We want $true \vdash \mathrm{ftpt}(P)$ **frm** P to hold for any primitive assertion P.

For any expression F, define the set of read effects of F, written $\mathrm{ftpt}(F)$, as follows: If F is an ordinary expression, E, define $\mathrm{ftpt}(E) = \{\mathbf{rd}\, x \mid x \in \mathit{Vars}(E)\}$. For a region expression G, define $\mathrm{ftpt}(G)$ as:

$$
\begin{array}{llll}
\mathrm{ftpt}(x) & = \{\mathbf{rd}\, x\} & \mathrm{ftpt}(x.g) & = \{\mathbf{rd}\, x, \mathbf{rd}\, \langle x \rangle{\cdot}g\} \\
\mathrm{ftpt}(\mathbf{alloc}) & = \{\mathbf{rd}\, \mathbf{alloc}\} & \mathrm{ftpt}(\mathbf{emp}) & = \varnothing \\
\mathrm{ftpt}(G_1 \cup G_2) & = \mathrm{ftpt}(G_1) \cup \mathrm{ftpt}(G_2) & \mathrm{ftpt}(G_1 \cap G_2) & = \mathrm{ftpt}(G_1) \cup \mathrm{ftpt}(G_2) \\
\mathrm{ftpt}(G_1 - G_2) & = \mathrm{ftpt}(G_1) \cup \mathrm{ftpt}(G_2) & \mathrm{ftpt}(\langle E \rangle) & = \mathrm{ftpt}(E)
\end{array}
$$

For primitive assertions P, define $\mathrm{ftpt}(P)$ as follows:

$$
\begin{array}{ll}
\mathrm{ftpt}(E = E') & = \mathrm{ftpt}(E) \cup \mathrm{ftpt}(E') \\
\mathrm{ftpt}(x.f = F) & = \{\mathbf{rd}\, x, \mathbf{rd}\, \langle x \rangle{\cdot}f\} \cup \mathrm{ftpt}(F) \\
\mathrm{ftpt}(G_1 \subseteq G_2) & = \mathrm{ftpt}(G_1 \,\#\, G_2) = \mathrm{ftpt}(G_1) \cup \mathrm{ftpt}(G_2) \\
\mathrm{ftpt}(G_1{\cdot}f \subseteq G_2) & = \mathrm{ftpt}(G_1{\cdot}f \,\#\, G_2) = \mathrm{ftpt}(G_1) \cup \mathrm{ftpt}(G_2) \cup \{\mathbf{rd}\, G_1{\cdot}f\}
\end{array}
$$

Fig. 7 specifies the judgement $P \vdash \overline{\varepsilon}$ **frm** P'. The rule for framing a conjunction $P_1 \wedge P_2$ with $\overline{\varepsilon}$ allows P_1 to be used as hypothesis in showing that $\overline{\varepsilon}$ frames P_2. This is sound because in a state where P_1 is false, the conjunction's value is independent from the

$$\frac{P \text{ is primitive}}{true \vdash \mathsf{ftpt}(P) \ \textbf{frm} \ P}$$

$$\frac{P \vdash \overline{\varepsilon}_1 \ \textbf{frm} \ P' \qquad P \vdash \overline{\varepsilon}_1 \leq \overline{\varepsilon}_2 \qquad Q \Rightarrow P}{Q \vdash \overline{\varepsilon}_2 \ \textbf{frm} \ P'}$$

$$\frac{P_1 \Leftrightarrow P_2 \qquad P \vdash \overline{\varepsilon} \ \textbf{frm} \ P_1}{P \vdash \overline{\varepsilon} \ \textbf{frm} \ P_2}$$

$$\frac{P \vdash \overline{\varepsilon} \ \textbf{frm} \ P_1 \qquad P \wedge P_1 \vdash \overline{\varepsilon} \ \textbf{frm} \ P_2}{P \vdash \overline{\varepsilon} \ \textbf{frm} \ P_1 \wedge P_2}$$

$$\frac{P \vdash \overline{\varepsilon}, \mathbf{rd}\, x \ \textbf{frm} \ P'}{P \vdash \overline{\varepsilon} \ \textbf{frm} \ \forall x\!:\!\textbf{int} \mid P'}$$

$$\frac{\mathsf{ftpt}(G) \subseteq \overline{\varepsilon} \qquad P \wedge x \in G \vdash \overline{\varepsilon}, \mathbf{rd}\, x, \mathbf{rd}\, \langle x \rangle . \overline{f} \ \textbf{frm} \ P'}{P \vdash \overline{\varepsilon}, \mathbf{rd}\, G . \overline{f} \ \textbf{frm} \ \forall x\!:\!K \in G \mid P'}$$

$$\frac{\begin{array}{cc} \mathsf{ftpt}(G) \subseteq \overline{\varepsilon} & \mathsf{ftpt}(G') \subseteq \overline{\varepsilon} \\ P \Rightarrow \forall x \in G \mid x.g \subseteq G' & P \wedge x \in G \vdash \overline{\varepsilon}, \mathbf{rd}\, x, \mathbf{rd}\, \langle x \rangle . \overline{f}, \mathbf{rd}\, x.g . \overline{g} \ \textbf{frm} \ P' \end{array}}{P \vdash \overline{\varepsilon}, \mathbf{rd}\, G . \overline{f}, \mathbf{rd}\, G' . \overline{g} \ \textbf{frm} \ \forall x\!:\!K \in G \mid P'}$$

Fig. 7. Inductive definition of the frames judgement

value of P_2. It is very helpful in subsuming local effects by more global effects. For example, suppose $\overline{\varepsilon} = \mathbf{rd}\, o, \mathbf{rd}\, p, \mathbf{rd}\, r, \mathbf{rd}\, r.nxt$ and we wish to establish that $\overline{\varepsilon}$ frames the formula $o \in r \wedge p = o.nxt$. It is clear that $\overline{\varepsilon}$ frames $o \in r$. But the frame of $p = o.nxt$ must include $\langle o \rangle.nxt$, and this is missing from $\overline{\varepsilon}$. However, because of $o \in r$ we have $\mathbf{rd}\, \langle o \rangle.nxt \leq \mathbf{rd}\, r.nxt$ using the second rule in Fig. 6. Note that $\wedge$ is commutative —it has standard semantics. The rule for $\wedge$ can be use for either conjunct, owing to the rule to its left which allows use of a valid $P_1 \Leftrightarrow P_2$.

To frame a quantification $\forall x\!:\!K \in G \mid P'$ in context P, observe that because P' might refer to x, we are likely to need $\mathbf{rd}\, x, \mathbf{rd}\, \langle x \rangle.\overline{f}$ and $x.g.\overline{g}$ (i.e., the read effects of the pivot field $x.g$) to frame P'. The frame of the quantification cannot mention x. However, read effects $\mathbf{rd}\, \langle x \rangle.\overline{f}$ may be subsumed by $\mathbf{rd}\, G.\overline{f}$ because $x \in G$. Similarly if we are able to establish that for all x the pivot expressions $x.g$ are all bounded by the region G', the effect $x.g.\overline{g}$ can be subsumed by $G'.\overline{g}$. The first rule in Fig. 7 for quantification of a reference variable $(x\!:\!K)$ applies when there are no pivot regions, the second when P' uses only a single pivot region, $x.g$. The generalization to multiple pivots is straightforward but notationally messy.

For our running examples one can derive the following:

$$true \vdash \mathbf{rd}\, o, r, r.nxt \ \textbf{frm} \ List(o, r)$$
$$true \vdash \mathbf{rd}\, o, s, v, \langle o \rangle.cache, \langle o \rangle.sub \ \textbf{frm} \ Obs(o, s, v)$$
$$true \vdash \mathbf{rd}\, s, v, \langle s \rangle.val, \langle s \rangle.obs, \langle s \rangle.O, s.O.nxt \ \textbf{frm} \ Sub(s, v)$$
$$true \vdash \mathbf{rd}\, s, v, \langle s \rangle.val, \langle s \rangle.obs, \langle s \rangle.O, s.O.nxt, s.O.cache, s.O.sub \ \textbf{frm} \ SubObs(s, v)$$

Lemma 3 (footprint agreement). For any states, σ, σ', for any expression F, suppose that σ, σ' agree on $\mathsf{ftpt}(F)$. Then $[\![F]\!]\sigma = [\![F]\!]\sigma'$.

Lemma 4 (frame agreement). For any σ, σ', any predicates P, P', and any set of effects $\overline{\varepsilon}$, suppose $P \vdash \overline{\varepsilon} \ \textbf{frm} \ P'$ and $\sigma \models P$ and $\sigma \sim_{\overline{\varepsilon}} \sigma'$. Then $\sigma \models P'$ iff $\sigma' \models P'$.

Proof. By induction on a derivation of $P \vdash \overline{\varepsilon} \ \textbf{frm} \ P'$. We consider the case for conjunction. Suppose $P \models \overline{\varepsilon} \ \textbf{frm} \ P_1 \wedge P_2$ because $P \models \overline{\varepsilon} \ \textbf{frm} \ P_1$ and $P \wedge P_1 \models \overline{\varepsilon} \ \textbf{frm} \ P_2$.

Assume that σ, σ' agree on $\overline{\varepsilon}$ and $\sigma \models P$. By induction on judgement of P_1 we obtain $\sigma \models P_1$ iff $\sigma' \models P_1$. Case $\sigma \models P_1$: Then $\sigma \models P \wedge P_1$. Hence by induction on judgement of P_2 we obtain $\sigma \models P_2$ iff $\sigma' \models P_2$. Thus $\sigma \models P_1 \wedge P_2$ and $\sigma' \models P_1 \wedge P_2$. Case $\sigma \not\models P_1$: Then $\sigma' \not\models P_1$. Hence, in either case, $\sigma \models P_1 \wedge P_2$ iff $\sigma' \models P_1 \wedge P_2$.   □

6.2   Separators

Given effect sets $\overline{\varepsilon}_r$ and $\overline{\varepsilon}_w$, we define the *separator* formula $\overline{\varepsilon}_r \star \overline{\varepsilon}_w$ to be a conjunction of certain disjointnesses. In a state where $\overline{\varepsilon}_r \star \overline{\varepsilon}_w$ holds, nothing that the read effects in $\overline{\varepsilon}_r$ allow to be read can be written according to the write effects in $\overline{\varepsilon}_w$. Note that $\overline{\varepsilon}_w$ (resp. $\overline{\varepsilon}_r$) may contain read (resp. write) effects but these do not influence the separator.

Definition 3 (separator). Define separator $\overline{\varepsilon}_r \star \overline{\varepsilon}_w$ by recursion on the effect sets:

$$
\begin{aligned}
\mathbf{rd}\, G_1 \bullet f \,\star\, \mathbf{wr}\, G_2 \bullet g &= \text{ if } f \equiv g \text{ then } G_1 \,\#\, G_2 \text{ else } \textit{true} \\
\mathbf{rd}\, y \,\star\, \mathbf{wr}\, x &= \text{ if } x \equiv y \text{ then } \textit{false} \text{ else } \textit{true} \\
\mathbf{rd}\,\mathbf{alloc} \,\star\, \mathbf{wr}\,\mathbf{alloc} &= \textit{false} \\
\varepsilon \,\star\, \varepsilon' &= \textit{true} \quad \text{otherwise (for all other single effects)} \\
(\varepsilon, \overline{\varepsilon}) \,\star\, \overline{\varepsilon}_1 &= (\varepsilon \star \overline{\varepsilon}_1) \wedge (\overline{\varepsilon} \star \overline{\varepsilon}_1) \\
\varepsilon \,\star\, (\varepsilon', \overline{\varepsilon}) &= (\varepsilon \star \varepsilon') \wedge (\varepsilon \star \overline{\varepsilon})
\end{aligned}
$$

In Sec. 2, to get Eqn. (3) we needed to use a separator. We can now restate that condition as $\langle y \rangle \subseteq r_1 \wedge r_1 \,\#\, r_2 \Rightarrow \mathbf{rd}\, r_2 \bullet item \star \mathbf{wr}\, \langle y \rangle \bullet item$. By definition of $\star$, it is immediate that the consequent is equal to $r_2 \,\#\, \langle y \rangle$ so the implication is valid.

Lemma 5 (separator agreement). Consider any effect sets $\overline{\varepsilon}_1$ and $\overline{\varepsilon}_2$. Suppose $\sigma \rightarrow \sigma' \models \overline{\varepsilon}_2$ and $\sigma \models \overline{\varepsilon}_1 \star \overline{\varepsilon}_2$. Then $\sigma \sim_{\overline{\varepsilon}_1} \sigma'$.

On separating conjunction. In separation logic, $P_1 * P_2$ says that P_1 and P_2 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_1 and P_2. Suppose $\overline{\varepsilon}_1 \;\mathbf{frm}\; P_1$ and $\overline{\varepsilon}_2 \;\mathbf{frm}\; P_2$. Obtain $\overline{\varepsilon}_2'$ from $\overline{\varepsilon}_2$ by discarding reads of variables and replacing each region read $\mathbf{rd}\, G \bullet f$ by $\mathbf{wr}\, G \bullet f$. Then the separation logic formula $P_1 * P_2$ amounts to $P_1 \wedge P_2 \wedge (\overline{\varepsilon}_1 \star \overline{\varepsilon}_2')$.

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_1 and P_2 do not have unique semantic footprints. Our use of explicit footprints and ghost variables can be seen as skolemizing the existential implicit in $*$, since $\overline{\varepsilon}_1 \star \overline{\varepsilon}_2'$ typically refers to regions involving ghost variables assigned in the (instrumented) program. For example, define $P(r) \mathrel{\widehat{=}} \forall x : Node \in r \mid x.item \leq 0$ and $Q(r) \mathrel{\widehat{=}} \forall x : Node \in r \mid x.item \geq 0$. Let $R \mathrel{\widehat{=}} P(r) \wedge Q(\mathbf{alloc} - r)$. Then we have both $\{\, R \,\}\, x := \mathbf{new}\, K; x.n := 0 \,\{\, R \,\}\, [\mathbf{wr}\, x, r]$ and $\{\, R \,\}\, x := \mathbf{new}\, K; x.n := 0; r := r \cup \langle x \rangle \,\{\, R \,\}\, [\mathbf{wr}\, x, r]$. But the reasoner must choose between these two commands. Issues with nondeterminacy could arise if we allowed bound region variables, e.g., $\exists r \mid P(r) \wedge Q(\mathbf{alloc} - r)$. Such matters are discussed further in [21].

6.3   Immunity

Recall from Sec. 2 the sequence $y := x.left; y.item := 0$. The individual effects, write of y and write of $\langle y \rangle.item$, cannot just be unioned to give the effect of the sequence, because write effects are interpreted in the pre-states (Def. 1(b)). The y in $\mathbf{wr}\,\langle y \rangle.item$ is not the same y as in the pre-state of the entire composition. The proof rule for sequential composition in Fig 8 must therefore ensure that the effect of the field update is *immune* from (or does not interfere with) the effect of the assignment. In this particular case we saw that the effect $\mathbf{wr}\,\langle y \rangle.item$ can be subsumed by a bigger effect, $\mathbf{wr}\,r_1.item$. The footprint of the region r_1 in the write effect is separate from the footprint of y and this permits the combined write effects to be $\mathbf{wr}\,y, \mathbf{wr}\,r_1.item$.

Definition 4 ($P/\overline{\varepsilon}$-immune). Region expression G is said to be $P/\overline{\varepsilon}$-immune provided $P \Rightarrow \mathsf{ftpt}(G) \star \overline{\varepsilon}$ is valid. Effect set $\overline{\varepsilon}_2$ is $P/\overline{\varepsilon}_1$-immune provided that for all G, f such that $\mathbf{wr}\,G.f$ occurs in $\overline{\varepsilon}_2$, it is the case that G is $P/\overline{\varepsilon}_1$-immune. $\square$

For example, **alloc** is $P/\overline{\varepsilon}$-immune provided $\mathbf{wr}\,\mathbf{alloc}$ is not in $\overline{\varepsilon}$. Also, $\mathbf{wr}\,x$ is $true/\mathbf{wr}\,x$-immune (vacuously), but $\mathbf{wr}\,\langle x \rangle.f$ is not $true/\mathbf{wr}\,x$-immune because $\mathsf{ftpt}(\langle x \rangle) \star \mathbf{wr}\,x = false$ by Def. 3.

The key property of immunity is that if $\{\,P\,\}\,C_1\,\{\,P_1\,\}\,[\overline{\varepsilon}_1]$ and $\{\,P_1\,\}\,C_2\,\{\,P'\,\}\,[\overline{\varepsilon}_2]$ are valid, and $\overline{\varepsilon}_2$ is $P/\overline{\varepsilon}_1$-immune, then $\overline{\varepsilon}_1, \overline{\varepsilon}_2$ is a valid effect for the sequence $C_1; C_2$. This is part of the proof of soundness for the [Seq] rule, see Thm. 1.

Lemma 6. Let G be $P/\overline{\varepsilon}$-immune. Then $[\![G]\!]\sigma = [\![G]\!]\sigma'$ for any σ, σ' such that $\sigma \to \sigma' \models \overline{\varepsilon}$ and $\sigma \models P$.

Proof. Since G is $P/\overline{\varepsilon}$-immune, $P \Rightarrow \mathsf{ftpt}(G) \star \overline{\varepsilon}$. So by $\sigma \models P$, we have $\sigma \models \mathsf{ftpt}(G) \star \overline{\varepsilon}$. Then from $\sigma \to \sigma' \models \overline{\varepsilon}$ we have by Lemma 5 that σ, σ' agree on $\mathsf{ftpt}(G)$. Then by Lemma 3 we have $[\![G]\!]\sigma = [\![G]\!]\sigma'$. $\square$

7   Program Correctness

A *correctness statement* takes the form $\{\,P\,\}\,C\,\{\,P'\,\}\,[\overline{\varepsilon}]$. The intended meaning is that from any initial state that satisfies P, C does not fault (terminate with error), and if it terminates then the final state satisfies P'. Moreover any allocation and update effects are allowed by $\overline{\varepsilon}$ (Def. 1). The statement is *well-formed in Γ* provided that P, P', C, and $\overline{\varepsilon}$ are well-formed in Γ.

The notation $\vdash^\Gamma$ is used for provability of statements that are well formed in Γ, so the proof system derives judgements of the form $\vdash^\Gamma \{\,P\,\}\,C\,\{\,P'\,\}\,[\overline{\varepsilon}]$. The semantics is used to define *valid* correctness statements, for which we use notation $\models^\Gamma$.

Definition 5 (validity). For state transformer φ of type Γ, define $\varphi \models^\Gamma \{\,P\,\} - \{\,P'\,\}\,[\overline{\varepsilon}]$ iff for all Γ-states σ, σ' such that $\sigma \models P$ we have $\varphi(\sigma) \neq \text{\Lightning}$ and if $\varphi(\sigma) = \sigma'$ then $\sigma' \models P'$ and $\sigma \to \sigma' \models \overline{\varepsilon}$.

Let $\{\,P\,\}\,C\,\{\,P'\,\}\,[\overline{\varepsilon}]$ be well-formed in Γ. The correctness statement is *valid*, written $\models^\Gamma \{\,P\,\}\,C\,\{\,P'\,\}\,[\overline{\varepsilon}]$, if and only if $[\![\Gamma \vdash C]\!] \models^\Gamma \{\,P\,\} - \{\,P'\,\}\,[\overline{\varepsilon}]$. $\square$

ALLOC
$$\dfrac{\mathsf{fields}(K)=\overline{f}:\overline{T}}{\vdash\{\mathit{true}\}\ x:=\mathbf{new}\ K\ \{x\,\mathbf{is}\,K\wedge x.\overline{f}=\overline{\mathit{default}(\overline{T})}\}\,[\mathbf{wr}\,x,\mathbf{wr\,alloc},\mathbf{fr}\,\langle x\rangle]}$$

FIELDACC
$$\dfrac{z\not\equiv x}{\vdash\{y\neq\mathbf{null}\wedge z=y\}\ x:=y.f\ \{x=z.f\}\,[\mathbf{wr}\,x]}$$

FIELDUPD $\vdash\{x\neq\mathbf{null}\wedge y=F\}\ x.f:=F\ \{x.f=y\}\,[\mathbf{wr}\,\langle x\rangle\boldsymbol{\cdot}f]$

SEQ
$$\dfrac{\begin{array}{c}\vdash\{P\}\ C_1\ \{P_1\}\,[\overline{\varepsilon}_1,\mathbf{fr}\,G]\\[2pt]\vdash\{P_1\}\ C_2\ \{P'\}\,[\overline{\varepsilon}_2,\mathbf{wr}\,\overline{G\boldsymbol{\cdot}f}]\qquad\overline{\varepsilon}_1\ \text{is}\ \mathbf{fr}\text{-free}\qquad\overline{\varepsilon}_2\ \text{is}\ P/\overline{\varepsilon}_1\text{-immune}\\[2pt]G\ \text{is}\ P_1/(\overline{\varepsilon}_2,\mathbf{wr}\,\overline{G\boldsymbol{\cdot}f})\text{-immune}\qquad P_1\Rightarrow\overline{G}_i\subseteq G\ \text{for every}\ \mathbf{wr}\,\overline{G}_i\boldsymbol{\cdot}f_i\end{array}}{\vdash\{P\}\ C_1\,;C_2\ \{P'\}\,[\overline{\varepsilon}_1,\overline{\varepsilon}_2,\mathbf{fr}\,G]}$$

VAR
$$\dfrac{\vdash^{\Gamma,x:T}\{P\wedge x=\mathit{default}(T)\}\ C\ \{P'\}\,[\mathbf{wr}\,x,\overline{\varepsilon}]}{\vdash^{\Gamma}\{P\}\ \mathbf{var}\ x:T\ \mathbf{in}\ C\ \mathbf{end}\ \{P'\}\,[\overline{\varepsilon}]}$$

Fig. 8. Selected correctness rules and axioms for commands

FRAME
$$\dfrac{\vdash\{P\}\ C\ \{P'\}\,[\overline{\varepsilon}_C]\qquad P\vdash\overline{\varepsilon}_Q\ \mathbf{frm}\ Q\qquad P\Rightarrow\overline{\varepsilon}_Q\star\overline{\varepsilon}_C}{\vdash\{P\wedge Q\}\ C\ \{P'\wedge Q\}\,[\overline{\varepsilon}_C]}$$

SUB EFF $\dfrac{\vdash\{P\}\ C\ \{P'\}\,[\overline{\varepsilon}]\qquad P\vdash\overline{\varepsilon}\le\overline{\varepsilon}'}{\vdash\{P\}\ C\ \{P'\}\,[\overline{\varepsilon}']}$ CONTEXT $\dfrac{\vdash^{\Gamma}\{P\}\ C\ \{P'\}\,[\overline{\varepsilon}]}{\vdash^{\Gamma,x:T}\{P\}\ C\ \{P'\}\,[\overline{\varepsilon}]}$

NO UPDATE $\dfrac{\vdash\{P\}\ C\ \{P'\}\,[\mathbf{wr}\,\langle x\rangle\boldsymbol{\cdot}f,\overline{\varepsilon}]\qquad\mathbf{rd}\,x\star\overline{\varepsilon}\qquad\mathbf{rd}\,y\star\overline{\varepsilon}\qquad P\vee P'\Rightarrow x.f=y}{\vdash\{P\}\ C\ \{P'\}\,[\overline{\varepsilon}]}$

Fig. 9. Selected structural rules. Rules of Consequence, Conjunction, etc. are as usual.

Fig. 8 gives selected syntax-directed proof rules and axioms. In axiom [FieldUpd], one step of dereferencing is allowed since F in the rule can be of the form $x.f$ in the case that $f:\mathbf{rgn}$. But if we allowed command $x.f:=y.f.g$, the rule would yield postcondition $x.f=y.f.g$ which is unsound due to possible sharing.

Fig. 9 gives selected structural rules. Rule [No Update] illustrates how assertional reasoning can be used to eliminate effects.

Theorem 1. If $\vdash\{P\}\ C\ \{P'\}\,[\overline{\varepsilon}]$ then $\models\{P\}\ C\ \{P'\}\,[\overline{\varepsilon}]$, for any $C,P,P',\overline{\varepsilon}$.

Proof. By induction on the derivation of $\vdash\{P\}\ C\ \{P'\}\,[\overline{\varepsilon}]$. This boils down to showing soundness for each rule. For brevity we focus on the case of normal termination. Recall that σ,σ',σ_1 range over proper states (non-$\Uparrow$).

Case [Frame]: To prove $\models\{P\wedge Q\}\ C\ \{P'\wedge Q\}\,[\overline{\varepsilon}_C]$, suppose $\sigma\models P\wedge Q$ and $[\![C]\!]\sigma=\sigma'$. Then $\sigma\models P$. From $\vdash\{P\}\ C\ \{P'\}\,[\overline{\varepsilon}]$ we get $\models\{P\}\ C\ \{P'\}\,[\overline{\varepsilon}]$ by induction, hence $\sigma'\models P'$ and $\sigma\to\sigma'\models\overline{\varepsilon}_C$. Using $\sigma\models P$ and $P\Rightarrow\overline{\varepsilon}_Q\star\overline{\varepsilon}_C$, we

get $\sigma \models \overline{\varepsilon}_Q \star \overline{\varepsilon}_C$. Now by Lemma 5 we can conclude that σ, σ' agree on $\overline{\varepsilon}_Q$. So we have $P \vdash \overline{\varepsilon}_Q$ **frm** Q and σ, σ' agree on $\overline{\varepsilon}_Q$ and $\sigma \models P$. Thus from Lemma 4 we can conclude that $\sigma \models Q$ iff $\sigma' \models Q$. But $\sigma \models Q$ because $\sigma \models P \wedge Q$. Hence $\sigma' \models Q$ and so $\sigma' \models P' \wedge Q$.

Case [Seq]: Let σ be any Γ-state such that $\sigma \models P$. Suppose $[\![C_1]\!]\sigma = \sigma_1$ and $[\![C_2]\!]\sigma_1 = \sigma'$. By validity of the antecedent correctness statements we get $\sigma_1 \models P_1$ and $\sigma' \models P'$; moreover $\sigma \to \sigma_1 \models \overline{\varepsilon}_1, \mathbf{fr}\, G$ and $\sigma_1 \to \sigma' \models \overline{\varepsilon}_2, \mathbf{wr}\, \overline{G.f}$. To prove $\sigma \to \sigma' \models \overline{\varepsilon}_1, \overline{\varepsilon}_2, \mathbf{fr}\, G$, we argue by cases on the parts of Def. 1.

Part (a): Consider any x such that $\sigma(x) \neq \sigma'(x)$. If $\sigma_1(x) \neq \sigma'(x)$ then $\mathbf{wr}\, x$ is in $\overline{\varepsilon}_2$, by $\sigma_1 \models P_1$ and $\sigma_1 \to \sigma' \models \overline{\varepsilon}_2, \mathbf{wr}\, \overline{G.f}$ (from above). If $\sigma_1(x) = \sigma'(x)$ then $\sigma(x) \neq \sigma_1(x)$ so then $\mathbf{wr}\, x$ is in $\overline{\varepsilon}_1$, by $\sigma \models P$ and $\sigma \to \sigma_1 \models \overline{\varepsilon}_1, \mathbf{fr}\, G$ (from above).

Part (b): Consider any $p \in \mathsf{alloc}(\sigma)$ and f such that $\sigma(p.f) \neq \sigma'(p.f)$.

- Case $\sigma_1(p.f) \neq \sigma'(p.f)$: Owing to $\sigma_1 \models P_1$ and $\sigma_1 \to \sigma' \models \overline{\varepsilon}_2, \mathbf{wr}\, \overline{G.f}$, we have one of two cases:
  - There is $\mathbf{wr}\, G'\!.f \in \overline{\varepsilon}_2$ such that $p \in [\![G']\!]\sigma_1$. By antecedent of [Seq], $\overline{\varepsilon}_2$ is $P/\overline{\varepsilon}_1$-immune, so G' is $P/\overline{\varepsilon}_1$-immune. Thus by Lemma 6, $p \in [\![G']\!]\sigma$. Thus this update is allowed in virtue of $\mathbf{wr}\, G'\!.f$.
  - There is i such that $p \in [\![\overline{G}_i]\!]\sigma_1$ and $\overline{f}_i$ is f. Since $\sigma_1 \models P_1$, antecedent $P_1 \Rightarrow \overline{G}_i \subseteq G$ of [Seq] yields $p \in [\![G]\!]\sigma_1$. And since we have $\sigma \to \sigma_1 \models \overline{\varepsilon}_1, \mathbf{fr}\, G$, we have that $[\![G]\!]\sigma_1 \subseteq \mathsf{alloc}(\sigma') - \mathsf{alloc}(\sigma)$, which contradicts the assumption that $p \in \mathsf{alloc}(\sigma)$ —so this case cannot happen.
- Case $\sigma_1(p.f) = \sigma'(p.f)$: Then $\sigma(p.f) \neq \sigma_1(p.f)$, so by $\sigma \to \sigma_1 \models \overline{\varepsilon}_1, \mathbf{fr}\, G$ there is some $\mathbf{wr}\, G'\!.f \in \overline{\varepsilon}_1$ with $p \in [\![G']\!]\sigma$.

Part (c): Consider any $p \in \mathsf{alloc}(\sigma')$ such that $p \notin \mathsf{alloc}(\sigma)$. Case $p \in \mathsf{alloc}(\sigma_1)$: then $\mathbf{wr}\, \mathbf{alloc}$ is in $\overline{\varepsilon}_1$. Case $p \notin \mathsf{alloc}(\sigma_1)$: then $\mathbf{wr}\, \mathbf{alloc}$ is in $\overline{\varepsilon}_2$.

Part (d): For any $\mathbf{fr}\, G'$ in $\overline{\varepsilon}_2$, we have $[\![G']\!]\sigma' \subseteq \mathsf{alloc}(\sigma') - \mathsf{alloc}(\sigma_1)$ from $\sigma_1 \to \sigma' \models \overline{\varepsilon}_2, \mathbf{wr}\, \overline{G.f}$. Since σ_1 extends σ (by semantics), we thus have $[\![G']\!]\sigma' \subseteq \mathsf{alloc}(\sigma') - \mathsf{alloc}(\sigma)$. Finally, since $\overline{\varepsilon}_1$ is $\mathbf{fr}$-free it remains to justify the final effect $\mathbf{fr}\, G$: Using the antecedent that G is $P_1/(\overline{\varepsilon}_2, \mathbf{wr}\, \overline{G.f})$-immune, and $\sigma_1 \models P_1$, we have by Lemma 6 and $\sigma_1 \to \sigma' \models \overline{\varepsilon}_2, \mathbf{wr}\, \overline{G.f}$ that $[\![G]\!]\sigma_1 = [\![G]\!]\sigma'$. By $\sigma \models P$ and $\sigma \to \sigma_1 \models \overline{\varepsilon}_1, \mathbf{fr}\, G$ we have $[\![G]\!]\sigma_1 \subseteq \mathsf{alloc}(\sigma_1) - \mathsf{alloc}(\sigma)$ and hence since σ' extends σ_1 we have $[\![G]\!]\sigma_1 \subseteq \mathsf{alloc}(\sigma') - \mathsf{alloc}(\sigma)$. Using $[\![G]\!]\sigma_1 = [\![G]\!]\sigma'$ we get $[\![G]\!]\sigma' \subseteq \mathsf{alloc}(\sigma') - \mathsf{alloc}(\sigma)$. □

Substitution and ghost elimination rules. In order to connect initial and final states, we often use variables that occur in pre- and post-conditions but not the program. Substitution for such variables is sound, but it takes a bit of work to formulate that they do not occur in the program. This is made straightforward by the inclusion of read effects in command specifications. Then we can formulate the substitution rule as follows. We use Reynolds' notation for substitution in formulas, writing $P/x{\to}F$ for substitution of F for x in P.

$$\frac{\vdash \{P\}\, C\, \{P'\}\, [\overline{\varepsilon}] \qquad (P/x{\to}F) \Rightarrow \mathsf{ftpt}(F) \star (\overline{\varepsilon}/x{\to}F) \qquad \mathbf{rd}\, x \notin \overline{\varepsilon} \qquad \mathbf{wr}\, x \notin \overline{\varepsilon}}{\vdash \{P/x{\to}F\}\, C\, \{P'/x{\to}F\}\, [\overline{\varepsilon}/x{\to}F]}$$

In accord with our convention on well formed rule instantiations, the result of substitution must be well formed here, e.g., $(x.g \subseteq r)/x{\to}\mathbf{null}$ is not. The proof uses routine

techniques but it requires the semantics of read effects of commands which we omit from this paper for brevity. There is also an auxiliary elimination rule (cf. Owicki-Gries), needed since local variables of type **rgn** are used as ghosts for reasoning.

8   Examples

We conclude the Subject/Observer example by verifying a simple client program using the specifications in Fig. 5. Method call and constructor rules are omitted from the technical formalization but are straightforward and we use them here. Here is the client program: $o := \textbf{new } Observer(s); s.update(n); i := o.val();$. Here is its specification: requires $SubObs(s, s.val)$, ensures $i = n$, effects $\overline{\varepsilon_1}, \overline{\varepsilon_2}, \textbf{wr } i$ where: $\overline{\varepsilon_1} = \textbf{wr alloc}, o, s.O\text{-}nxt, \langle s \rangle\text{-}O, \langle s \rangle\text{-}obs, \textbf{fr } \langle o \rangle$, and $\overline{\varepsilon_2} = \textbf{wr } \langle s \rangle\text{-}val, s.O\text{-}cache$. Our first step uses the (omitted) rule for allocation with constructor call. Informally, the rule says that we can use constructor's pre/postconditions by adapting them to the calling context. The effects are those of the constructor with **self** replaced by LHS of the assignment.

$$\{\, SubObs(s, s.val) \,\}\ o := \textbf{new } Observer(s)\ \{\, SubObs(s, s.val) \wedge o \in s.O \,\}\,[\overline{\varepsilon_1}] \quad (4)$$

The method call rule yields $\{\, SubObs(s, s.val) \,\}\ s.update(n)\ \{\, SubObs(s, n) \,\}\,[\overline{\varepsilon_2}]$. Now we can apply Frame to conjoin $o \in s.O$ since $true \vdash \textbf{rd } o, s, \langle s \rangle\text{-}O\ \textbf{frm } o \in s.O$ and $true \Rightarrow (\textbf{rd } o, s, \langle s \rangle\text{-}O) \star (\textbf{wr } \langle s \rangle\text{-}val, s.O\text{-}cache)$.

$$\{\, SubObs(s, s.val) \wedge o \in s.O \,\}\ s.update(n)\ \{\, SubObs(s, n) \wedge o \in s.O \,\}\,[\overline{\varepsilon_2}] \quad (5)$$

Next, we have by the method call rule, and [Conseq]

$$\{\, SubObs(o.sub, o.sub.val) \wedge o \in o.sub.O \,\}\ i := o.val()\ \{\, i = o.sub.val \,\}\,[\textbf{wr } i]$$

Let $Q \mathrel{\widehat{=}} o.sub = s \wedge s.val = n$. Framing the above with Q we obtain

$$\{\, SubObs(o.sub, o.sub.val) \wedge o \in o.sub.O \wedge Q \,\}\ i := o.val()\ \{\, i = o.sub.val \wedge Q \,\}\,[\textbf{wr } i]$$

First, $i = o.sub.val \wedge Q$ implies $i = n$. Next, we can show that the postcondition of (5) implies $SubObs(o.sub, o.sub.val) \wedge o \in o.sub.O \wedge Q$ by unfolding the definition of $SubObs$ using $o \in o.sub.O$. Thus we get by [Conseq]

$$\{\, SubObs(s, n) \wedge o \in s.O \,\}\ i := o.val()\ \{\, i = n \,\}\,[\textbf{wr } i] \quad (6)$$

Now using [Seq] on (4), (5), (6) we obtain the desired correctness judgement at the beginning of the section.

For examples that use separation without any inductive predicates, we have verified the standard list copy and in-situ reversal algorithms with respect to the following specifications. For *reversal*: requires $x \in r \wedge r\text{-}nxt = r$, ensures $res \in r \wedge r\text{-}nxt = r$, effect $\textbf{wr } r\text{-}nxt$. That is, r remains closed, the result is in r, and there is no allocation. Our specification for *copy* says the copy is disjoint from the original: requires $x \in r_1 \wedge r_1\text{-}nxt = r_1$, ensures $res \in r_2 \wedge r_2\text{-}nxt = r_2 \wedge r_1 \,\#\, r_2$, effect $\textbf{wr } r_2$, **alloc**.

9   Framing Module Invariants

The previous section focused on framing in the small: using effect specifications to reason about commands in terms of specifications of their constituent commands. This section addresses framing at a higher level, in particular, reasoning about invariants for encapsulated state [13]. The idea is that the implementation of an abstract data type can maintain an invariant that pertains to its encapsulated data representation, without exposing the invariant to clients (or subclasses). This creates a mismatch between the client's view of a method call, say $\{P/\textbf{self}{\rightarrow}x\}\ x.m()\ \{P'/\textbf{self}{\rightarrow}x\}\ [\ldots]$, using the specification P, P' of m, and the proof obligation for the implementation C_m of m: $\{P \wedge Inv(\textbf{self})\}\ C_m\ \{P' \wedge Inv(\textbf{self})\}\ [\ldots]$.

Sec. 9.1 considers ownership confinement, the idea that a client-visible object that represents, say, a Collection can encapsulate its internal representation as a region disjoint from the reps of other collections and from clients. If the invariant is framed by the owned reps, and disjointness is maintained as a confinement invariant, and client effects are disjoint from the reps, then the mismatched proof obligations [13] are sound.

We treat confinement as an explicit invariant that pertains to all instances of some class (or subclasses, though we refrain from emphasizing that dimension). We treat the object invariant in terms of a single invariant that pertains to all instances. This treatment allows an encapsulation discipline to be expressed on a per-module basis, rather than being globally imposed on all code. Moreover, it allows the mismatch to be formalized as a second order frame rule like that of separation logic [23], with a confinement invariant and client effect bound carried through the part of a proof in which an invariant is hidden. The rule is admissible [21], which amounts to saying that the mismatch can always be fixed by augmenting the proof with explicit uses of our ordinary Frame rule.

Sec. 9.2 returns to the Subject/Observer example, again in terms of a global invariant that describes disjointness of encapsulated islands, albeit not based on hierarchical ownership.

9.1   Ownership and Object Invariants

The following classes illustrate a scenario like a set represented by a linked list that may contain duplicate elements.

class $Coll$ {**rgn** rep; $Node\ lst$; **int** $size$;}   **class** $Node$ {$T\ item$; **int** len; $Node\ nxt$;}

The $size$ of a collection is part of its external interface. For the sake of an example, we choose an object invariant that relates $size$ to the internal representation:

$CollI(c) \; \hat{=} \; c \neq null \wedge c\,\textbf{is}\,Coll \wedge c.lst \in c.rep \wedge c.rep\text{\textbullet}nxt \subseteq c.rep \wedge c.lst.len \geq c.size$

Recall that $c.lst \in c.rep$ abbreviates $c.lst \neq \textbf{null} \wedge \langle c\rangle\text{\textbullet}lst \subseteq c.rep$. Field rep serves to delimit the encapsulated representation or owned objects. The following can be derived:

$$true \vdash \textbf{rd}\,c, \langle c\rangle\text{\textbullet}(rep, lst, size), c.rep\text{\textbullet}(len, nxt)\ \textbf{frm}\ \ CollI(c) \qquad (7)$$

Let $d: Collection$, to consider a call $d.m()$ that relies on invariant $CollI(d)$. Methodologies based on ownership can be described as a way to ensure that a

client command, say $\{P\}\ C\ \{P'\}\ [\overline{\varepsilon}]$, can be lifted by the Frame rule to $\{P \wedge CollI(d)\}\ C\ \{P' \wedge CollI(d)\}\ [\overline{\varepsilon}]$ because the frame of $CollI(d)$ is necessarily separate from the client effect. If specifications for all parts of the client code are thus lifted, they match the hidden precondition $CollI(d)$.

An object invariant is intended to apply *separately* to *each instance* of the class. It is easy to say it applies to each instance: $\forall c : Coll \in \mathbf{alloc}\ |\ CollI(c)$. To illustrate the flexibility of our logic, let us instead suppose there is a global region variable $CollS$ that holds some or all instances of $Coll$. (In practice this would be a static field of class $Coll$.) The "component" of interest to clients is the pool of objects in $CollS$. We want each of these collections to satisfy its invariant: $\forall c \in CollS\ |\ CollI(c)$. To frame this formula, suppose global variable $CollR$ holds the union of the reps of $CollS$, i.e.

$$CollC_0 \ \hat{=} \ \ \forall c \in CollS\ |\ c.rep \subseteq CollR$$

This serves to derive, from (7), a frame for $\forall c \in CollS\ |\ CollI(c)$, to wit

$$CollC_0 \vdash \mathbf{rd}\ CollS, CollR, CollS\text{-}(rep, lst, size), CollR\text{-}len\ \mathbf{frm}\ \forall c \in CollS\ |\ CollI(c)$$

Formula $CollC_0$ says that the module's internal representation objects are in $CollR$. We can express the "package confinement" condition that clients don't reach reps:

$$CollC_1 \ \hat{=} \ \ (\mathbf{alloc} - (CollS \cup CollR))\text{-}\mathsf{any} \# CollR$$

If package confinement holds at call sites, then we can use the Frame rule to lift client code to get it to match with

$$\{P \wedge \forall c \in CollS\ |\ CollI(c)\}\ d.m()\ \{P' \wedge \forall c \in CollS\ |\ CollI(c)\}\ [\ldots] \qquad (8)$$

(Doing this once and for all is the point of the second order frame rule [21].)

The precondition of (8) is certainly strong enough to verify the implementation of m, since it implies $CollI(d)$. The postcondition appears very strong. But recall that an object invariant is supposed to apply "separately" to each instance. Besides separating clients from representations, with $CollC_1$, we can also use an "island confinement" condition to say that distinct collections have disjoint reps:

$$CollC_2 \ \hat{=} \ \ \forall c, c' : Coll \in CollS\ |\ c = c' \vee c.rep \# c'.rep$$

Confinement conditions like $CollC_0$, $CollC_1$, and $CollC_2$ can be enforced by ownership type systems and other pointer analyses.[1] Such enforcement mechanisms typically ensure that confinement holds in all reachable states.[2]

---

[1] For example, if we represent an ownership hierarchy using a ghost field *owner*, and impose the dominator property of Ownership Types, then $CollC_2$ will be an easy consequence. If instead we allow some references, for read-only use as in Universe Types, then a weaker confinement condition holds; but the story can still play out, as we distinguish between read and write effects of commands.

[2] Making these conditions explicit in program annotations might be a high cost, compared with getting them as global invariants "for free" from a separate static analysis. On the other hand, making them explicit would be one way to show soundness of the result of a particular analysis, and also provide a means to work around restrictions due to approximations made by static analysis. For now we set that issue aside and simply assume they are all-states invariants.

Let us return to the problem of establishing the postcondition of (8) in the implementation of method m. To focus on **self**, we can rewrite $\forall c \in CollS \mid CollI(c)$ as $P_C \wedge CollI(\textbf{self})$ where

$$P_C \;\widehat{=}\; \forall c \in CollS - \langle \textbf{self} \rangle \mid CollI(c)$$

In light of the frame we found for $\forall c \in CollS \mid CollI(c)$, we can frame P_C as

$$CollC_0 \vdash \textbf{rd self}, CollS, CollR, (CollS - \langle \textbf{self} \rangle) \bullet (rep, lst, size), CollR \bullet len \;\; \textbf{frm} \;\; P_C$$

Further, we can remove **self**.$rep \bullet len$ from the effect **rd** $CollR \bullet len$ owing to the island confinement $CollC_2$.

In short, we should verify the implementation with respect to just $CollI(\textbf{self})$. Owing to confinement, the footprint of the implementation is disjoint from the frame of $\forall c \in CollS - \langle \textbf{self} \rangle \mid CollI(c)$, so the frame rule will yield $\forall c \in CollS \mid CollI(c)$.

Making confinement an all-states invariant is sufficient but not necessary. The confinement conditions are necessary at those points where the frame rule is used to lift a local correctness property, like preserving $CollI(\textbf{self})$, to a stronger property like preserving $\forall c \in CollS \mid CollI(c)$. Confinement may also be exploited for reasoning within the implementation, e.g., at outgoing method calls —more on that below.

In summary, we suggest that method implementations maintain module invariants, which may include module-wide conditions for resource management etc, together with object invariants in global form, like $\forall c \in CollS \mid CollI(c)$.

On reentrant callbacks. Through cyclic references, it is possible for some module operation to invoke a method that leads to a *re-entrant callback*. That is, an invocation on some object o at a point when another operation is already in progress and which may thus have temporarily falsified the invariant. In some cases, re-entrant callback can be shown to be impossible simply in virtue of the graph of which method implementations invoke which (disambiguated) methods. Another technique is the "visible state semantics" [18] in which invariants are required to hold on every method call boundary; so in particular the module operation is required to re-establish the object invariant before making any call. In some cases, pointer analysis can determine the absence of cyclic references by which a chain of calls can lead back to an instance. Using the Frame requires separation and thus prevents hiding invariants in cases where the effects of callbacks are not disjoint from the invariants' footprints.

Sometimes reentrant callbacks are desirable, of course. The Subject/Observer code is an example where the relevant methods belong together in a module and therefore hiding the invariant is not an issue. In cases where hiding is important, such as other Subject/Observer scenarios where the Observer cannot be expected to be responsible for the Subject's invariants, a typical solution is to designate the intended callback methods as not assuming the hidden invariant.

Hierarchical ownership. Coll relies on *Node* operations. If *Node* is in a different module, it may rely on invariants that are hidden from *Coll*. For example, if *Coll* requires that the actual number of values in the list is at most $lst.len$, then *Node* takes care of that when a new item is inserted. On the other hand, that will make $self.lst.len$ out of sync with $self.size$; restoring that is *Coll*'s responsibility.

Our idea is that a module, say the module for *Node*, has some confinement policy and its clients are checked for conformance. The code of *Node* can thereby rely on its object invariant. This can be hidden from *Coll*, which is a client of *Node*. In turn, by encapsulating its list nodes, possibly by a different discipline, *Coll* ensures that operations on nodes of $c.reps$ only happen when methods of *Coll* have control.

9.2   Beyond Ownership to Cluster Invariants

In this section let us say an *abstract component* is one or more *interface objects* that serve to represent some data abstraction, together with their representation objects which are meant for internal use. For example, a collection together with its iterators are an abstract component that provides a set with stateful enumerations. Ownership is suited to situations where there is a single interface object. Perhaps the most obvious way to express an invariant for this example is as a predicate that involves an instance of *Coll* together with all its associated *Iterators*. A number of techniques have been proposed to treat such a "cluster invariant" as one or more object invariants whose dependencies are constrained but not by ownership [22]. There are several reasons to try to reduce cluster invariants to object invariants. On the other hand, notions like peer dependencies and friendship are very limited in applicability. It is worthwhile to develop disciplines tailored to design patterns in wide use, such as Subject/Observer and Iterator. But it is also desirable to have a setting in which ad hoc reasoning patterns can be devised for very specific situations, yet still achieving modular and economic reasoning based on encapsulated invariants.

The client of a cluster with multiple interface objects has the potential to interfere with the invariant via those multiple handles, and thus the specifications of operations on particular interface objects —like adding to the collection, or advancing one of its iterators— must surely expose a holistic view of the cluster. In the paper [24] from which we borrow the Subject/Observer example, Parkinson uses a cluster invariant *SubObs* that appears in preconditions of methods of both Subject and Observer.

First, we imagine *Subject* and *Observer* are together in a module, with methods *register* and *add* given module scope while others are public. The module-scope methods are verified in the same context and without need to hide the invariant; it can simply be made explicit in their specifications. Now we factor apart the *Sub* predicate into a condition, *SubX*, suited to public (external) specifications, and another, *SubH*, suited to be a hidden invariant. (In a more realistic example, *val* would be a model field.)

$$SubX(s,v) \mathrel{\widehat{=}} s.val = v \qquad SubH(s,v) \mathrel{\widehat{=}} List(s.obs, s.O)$$
$$SubObsX(s,v) \mathrel{\widehat{=}} SubX(s,v) \wedge \forall o: Observer \in s.O \mid Obs(o,s,v)$$
$$SubObsH(s,v) \mathrel{\widehat{=}} SubH(s,v) \wedge SubX(s,v)$$

Whereas the X-versions of the invariant are left in the method specifications, the H-versions are used only for verification of the implementations of the methods of Subject and Observer. Specifically, a global invariant like this is used:

$$SOI \mathrel{\widehat{=}} \quad (\forall s: Subject \in \mathbf{alloc} \mid SubObsH(s, s.val))$$
$$\wedge (\forall o: Observer \in \mathbf{alloc} \mid o.sub \neq \mathbf{null} \Rightarrow o \in o.sub.O)$$

Given precondition SOI for a method of $Observer$, the verifier can instantiate the second conjunct with **self** for o, and the first conjunct with **self**.sub for s, to obtain the invariant for its cluster. Confinement invariants can be used to separate clusters so that the $Observer$ method is responsible for restoring the invariant, $SubObsH$, for its cluster but then restores SOI for all others simply by framing. In particular, the separation of clusters would look similar to $CollC_2$, though in this case the islands are not as isolated from clients since both $Subject$s and $Observer$s are accessible.

10   Discussion

The genesis of this work is our ongoing work on secure information flow analysis, combining verification and type checking [4]. We are developing a relational logic that lets us specify fine grained declassification policies. Amtoft et al [1] achieve precise, modular reasoning about information flow using regions. 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.

State of the art verifiers use intricate reasoning about the heap, often based on variants of ownership régimes. Our approach is inspired by the Boogie methodology [5,16], which is explicitly based on all-states invariants that use ghost fields to express an encapsulation régime (though not focused on confinement). Boogie also combines instance invariants into a global condition akin to our example $\forall c : Coll \in \mathbf{alloc} \mid CollI(c)$.

Another inspiration is the work of Kassios [14], showing how explicit use of ghost state can be effective without global imposition of a fixed programming discipline. Kassios uses a ghost field to hold the footprint of an object's invariant, a means to specify the footprint, and a means to specify that a procedure touches only that footprint. Kassios works directly with the semantics of "frames" as a second order predicate, quantifying over all global program states. By contrast, we work out a first order assertion-based logic. But there are similarities, e.g., Metatheorem 5.4.1 in his thesis is related to our notion of immunity, as is the swinging pivots restriction of Leino and Nelson [17, Sec. 8.3].

Smans et al [26] investigate Kassios' approach to framing in the setting of pre / post / modifies specifications, focusing on reasoning with pure method calls in assertions. Like us they use explicit regions in modifies specifications, expressed as pure methods. Their approach has been implemented in a prototype verifier and applied to examples like observer and iterator.

Smith [27] uses regions denoted by ownership contexts to formulate a simple, type based frame rule that resembles ours.

Recent work on ownership has addressed the need for clusters without a single dominating owner. Cameron et al [10] give a good survey of ownership systems. They adapt Ownership Types to a system of "boxes" (clusters) that describes rather than restricts program structure. Thus it does not ensure encapsulation, but they provide and prove sound an effect system for disjointness of boxes. Müller and Rudich [19] extend Universe Types, which provides encapsulation and has been adopted by JML for invariants, to solve the difficult problem of ownership transfer. Drossopoulou et al [12] provide a

general theory to account for a variety of invariant disciplines, focusing on visible state semantics. Ownership type systems cater for hierarchical ownership, with the benefit of uniformity and a fixed semantics of when invariants hold. We propose the use of second order framing with module-specific disciplines, in hopes of more flexible deployment of these and even earlier and simpler ownership systems.

The influence of separation logic on our work is clear. The separating conjunction hides the heap and expresses separation in the heap implicitly. Becauses footprints are shadows of predicates, specifications require full functional descriptions using inductive definitions, or else quantification over predicates. Parkinson's position paper [24] clearly articulates the case for specifications at the level of object clusters. In higher order separation logics [7] and Hoare type theory [20], one can quantify predicates to get multi-instance abstractions at the cost of intricate semantics and sometimes the loss of the [Conj] rule. Various works articulate the view that the second order frame rule pertains to static, single-instance modules, e.g., this motivated Parkinson's technique for hiding invariants by opaque naming, which can be understood as second order existentials [6].

Now we turn to future work. The Boogie discipline can be viewed as a proof outline logic; it would be interesting to show that the invariants specified in the discipline hold in that logic by using reasoning similar to the one developed in this paper. More generally, the logic may be of use in connecting and even unifying various disciplines for ownership and beyond. To that end we are investigating better means to abstract from field names than the crude "any" used here in examples. Another question is whether effects can be made conditional, or even subsumed in two-state postconditions, while retaining effective generation of framing conditions as in Sec. 6.

The proof rules of our logic are formulated in a way that shows how reasoning works. An automated verifier will likely not apply such rules directly but will rather transform the code and generate verification conditions. Experiments with the logic are underway, by translating into BoogiePL, and the third author is investigating decision procedures for quantifier-free assertions. Another avenue to explore is use of the logic as translation target from higher level static analyses; instead of metatheory to justify that analysis and its use, it just creates verification conditions (c.f. runtime verification).

References

1. Amtoft, T., Bandhakavi, S., Banerjee, A.: A logic for information flow in object-oriented programs. In: ACM Symposium on Principles of Programming Languages (POPL) (2006)
2. Banerjee, A., Naumann, D.A.: Ownership confinement ensures representation independence for object-oriented programs. Journal of the ACM 52(6), 894–960 (2005)
3. Banerjee, A., Naumann, D., Rosenberg, S.: Regional logic for local reasoning about global invariants, www.cs.stevens.edu/~naumann/pub/rllrgi.pdf

 4. Banerjee, A., Naumann, D., Rosenberg, S.: Towards a logical account of declassification. In: ACM Workshop on Programming Languages and Analysis for Security (2007)
 5. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology 3(6), 27–56 (2004)
 6. Bierman, G., Parkinson, M.: Separation logic and abstraction. In: ACM Symposium on Principles of Programming Languages (POPL), pp. 247–258 (2005)
 7. Birkedal, L., Torp-Smith, N., Yang, H.: Semantics of separation-logic typing and higher-order frame rules. In: IEEE Symp. on Logic in Computer Science (LICS) (2005)
 8. Bornat, R.: Proving pointer programs in Hoare logic. In: MPC (2000)
 9. Calcagno, C., O'Hearn, P., Bornat, R.: Program logic and equivalence in the presence of garbage collection. Theoretical Comput. Sci. 298(3), 557–581 (2003)
10. Cameron, N.R., Drossopoulou, S., Noble, J., Smith, M.J.: Multiple ownership. In: OOPSLA (2007)
11. Clarke, D., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effect. In: OOPSLA, pp. 292–310 (November 2002)
12. Drossopoulou, S., Francalana, A., Müller, P.: A unified framework for verification techniques for object invariants. In: FOOL (2008)
13. Hoare, C.A.R.: Proofs of correctness of data representations. Acta. Inf. 1, 271–281 (1972)
14. Kassios, I.T.: Dynamic framing: Support for framing, dependencies and sharing without restriction. In: Formal Methods: International Conference of Formal Methods Europe (2006)
15. Leavens, G.T., Naumann, D.A., Rosenberg, S.: Preliminary definition of core JML. Technical Report CS Report 2006-07, Stevens Institute of Technology (2006)
16. Leino, K.R.M., Müller, P.: Object invariants in dynamic contexts. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 491–516. Springer, Heidelberg (2004)
17. Leino, K.R.M., Nelson, G.: Data abstraction and information hiding. ACM Trans. Prog. Lang. Syst. 24(5), 491–553 (2002)
18. Müller, P.: Modular Specification and Verification of Object-Oriented Programs. In: Müller, P. (ed.) Modular Specification and Verification of Object-Oriented Programs. LNCS, vol. 2262. Springer, Heidelberg (2002)
19. Müller, P., Rudich, A.: Ownership transfer in Universe Types. In: ACM Conf. on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA) (2007)
20. Nanevski, A., Morrisett, G., Birkedal, L.: Polymorphism and separation in Hoare type theory. In: ICFP (2006)
21. Naumann, D.A.: An admissible second order frame rule in region logic. Technical Report CS Report 2008-02, Stevens Institute of Technology (2008)
22. Naumann, D.A., Barnett, M.: Towards imperative modules: Reasoning about invariants and sharing of mutable state. Theoretical Comput. Sci. 365, 143–168 (2006)
23. O'Hearn, P., Yang, H., Reynolds, J.: Separation and information hiding. In: ACM Symposium on Principles of Programming Languages (POPL), pp. 268–280 (2004)
24. Parkinson, M.: Class invariants: the end of the road. In: International Workshop on Aliasing, Confinement and Ownership (2007)
25. Pierik, C., de Boer, F.S.: A proof outline logic for object-oriented programming. Theoretical Comput. Sci. 343, 413–442 (2005)
26. Smans, J., Jacobs, B., Piessens, F., Schulte, W.: An automatic verifier for java-like programs based on dynamic frames. In: FASE (2008)
27. Smith, M., Drossopoulou, S.: Cheaper reasoning with ownership types. In: International Workshop on Aliasing, Confinement and Ownership (2003)
28. Tofte, M., Talpin, J.-P.: Implementation of the Typed Call-by-Value lambda-Calculus using a Stack of Regions. In: POPL (1994)