

Regional Logic for Local Reasoning about Global Invariants

Anindya Banerjee

Kansas State University, Manhattan, KS, USA
ab@cis.ksu.edu

David A. Naumann Stan Rosenberg

Stevens Institute of Tech., Hoboken, NJ, USA
[naumann|rosenbe]@cs.stevens.edu

Abstract

Shared mutable objects pose grave challenges in reasoning, especially for data abstraction and modularity. Current solutions rely on higher order predicates, non-standard logical connectives, reachability predicates, whole program analyses and restrictive disciplines. Among other shortcomings, such solutions can be difficult to deploy using standard decision procedures and other tools. This paper presents a novel logic for error-avoiding partial correctness of programs featuring shared mutable objects. Soundness is proved using a standard program semantics. The logic provides heap-local reasoning about mutation and separation, via novel use of ghost fields and variables of type ‘region’ (finite sets of object references). Intuitionistic separating conjunction can be encoded. A new form of modifies clause specifies write and allocation effects using region expressions; this supports effect masking and a frame rule that allows shared reads. The logic facilitates heap-local reasoning about object invariants: disciplines such as transferable ownership are expressible but not hard-wired in the logic. As a case study in data abstraction, a hypothetical (higher-order) frame rule is adapted from separation logic and proved sound in a strong sense: it is admissible, under suitable conditions.

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 that will make it possible to achieve the kinds of modularity associated with separation logic but using widely used specification languages, decision procedures, tools, etc, and avoiding the costs of transitive closure, higher order predicates, etc., while achieving vastly more flexibility.

Various notions of regions have been used in static analysis to abstract sets of objects of interest. Separation logic [23] owes its stunning 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, separation logic ideas have been critical to advances in data abstraction and information hiding [6] but the logic per se has difficulties in addressing data abstraction [23, 17].

This paper takes a step backwards, in a sense, encoding separation explicitly rather than abstracting from it via logical connectives. But it is a step forward in that footprints are made first-class. Our encoding is exceedingly simple: we augment a Java-like language with type `rgn` ranging over finite sets of (allocated) references. We instrument programs with assignments to ghost variables and fields, so assertions can refer explicitly to regions, as can the “modifies clause” that is often the most useful part of a program specification. Asserting the disjointness of regions helps delimit effects and facilitate heap-local reasoning, e.g., strong updates without need to globally propagate alias information.

In one light, what we get is yet another way to write excruciatingly intricate proofs about small pointer programs. It is no surprise that it is possible to reason in terms close to the semantic model [10]. If one’s aim is to prove functional correctness of, say, a garbage collector then our approach is not obviously better. Such proofs are inherently costly; 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 impressive. Without the need for inductive predicates that traverse data structure, 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 [13, 1] 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, 21] 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

[copyright notice will appear here]

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 semantics of regions. On the contrary, regions as ghost state can encode such disciplines but can also combine them in uniform or ad hoc ways.

There is a major difficulty: “modifies” specifications using region expressions dependent on mutable state are susceptible to a new kind of interference. The effect of a command can alter the meaning of the effect specification of another command! Our first major technical contribution is to uncover this problem and its solution. 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. But in this foundational study we expose the issues and formulate the solution in terms of a Hoare-style proof system which we prove sound using a standard program semantics.

Our second major contribution is to put the logic to work for data abstraction. Our ultimate goal is to integrate heap-modularity for dynamically instantiated clusters of interdependent objects with layered abstractions composed using sophisticated type based module systems. Here, we focus on the “hypothetical” or “higher order” frame rule from separation logic, which distills an essential aspect of local reasoning for a static module (i.e., single-instance abstraction). We prove our version sound, by simple syntactic means: it is an admissible rule, for well-asserted programs.

Outline. In the rest of this Introduction we sketch some examples to illustrate features of the logic. Sec. 2 formalizes a conventional semantics for an illustrative programming language and Sec. 3 presents the assertion language including a simple static analysis of the frame or footprint of a predicate. Sec. 4 formalizes effects using regions and gives a static analysis for independence of a formula from an effect. Sec. 5 defines correctness statements (Hoare triples plus an effects clause); the proof rules are given and proved sound. Sec. 6 investigates a second-order frame rule. Sec. 7 discusses what has been achieved, sketches related work, and outlines future work.

Overview. In the rest of this section we sketch some highlights from a proof that *in situ* list reversal is *in situ*. It involves objects of type *Node* with field *next*. Let $x, y : \text{Node}$ and $r : \text{rgn}$. The loop invariant includes

$$\text{inv} : \quad x \in r \wedge y \in r \wedge r.\text{next} \subseteq r$$

The last conjunct, $r.\text{next} \subseteq r$, says that for every object o in r , $o.\text{next}$ is in r .

As in separation logic, there are “small axioms” for assignments, that mention only the directly affected state. For example, the field access axiom yields

$$\{x \neq \text{null}\} \text{tmp} := x.\text{next} \{ \text{tmp} = x.\text{next} \} [\text{wr tmp}]$$

The frame rule, like Hoare’s invariance rule, allows a formula to be conjoined to the pre- and post-conditions provided the command does not interfere with it. We have that *inv* is independent from the effect wr tmp , written $\text{inv} \# \text{wr tmp}$, so the rule gives

$$\{x \neq \text{null} \wedge \text{inv}\} \text{tmp} := x.\text{next} \{ \text{tmp} = x.\text{next} \wedge \text{inv} \} [\text{wr tmp}]$$

The frame rule in separation logic requires that the framed predicate depend only on heap objects that are neither read nor written by the command. By tracking write effects, our frame rule can express that nothing shared is written, while allowing reads of shared state.

Next we consider a code fragment that updates field *next*, potentially interfering with $r.\text{next} \subseteq r$. The idea is to zoom in on an updated object so as to allow strong update. The field update axiom gives

$$\{x \neq \text{null}\} x.\text{next} := y \{x.\text{next} = y\} [\text{wr } \langle x \rangle.\text{next}]$$

where the effect $\text{wr } \langle x \rangle.\text{next}$ says that field *next* can be updated for any object in region $\langle x \rangle$, which is the singleton region of x . Define

$$\theta_1 : \quad r' = r - \langle x \rangle \wedge x \in r \wedge y \in r$$

where r' is a fresh variable. Since *next* is not in θ_1 , we have $\theta_1 \# \text{wr } \langle x \rangle.\text{next}$ so the frame rule yields

$$\{x \neq \text{null} \wedge \theta_1\} x.\text{next} := y \{x.\text{next} = y \wedge \theta_1\} [\text{wr } \langle x \rangle.\text{next}]$$

Now we can use frame again to add $r'.\text{next} \subseteq r$: this is independent from effect $\text{wr } \langle x \rangle.\text{next}$, because from $r' = r - \langle x \rangle$ it follows that r' is disjoint from $\langle x \rangle$. The formal instantiation of the side condition of rule [Frame] in this case looks like $\theta_1 \vdash r'.\text{next} \subseteq r \# \text{wr } \langle x \rangle.\text{next}$ and the result is

$$\{x \neq \text{null} \wedge \theta_1 \wedge r'.\text{next} \subseteq r\} x.\text{next} := y \{x.\text{next} = y \wedge \theta_1 \wedge r'.\text{next} \subseteq r\} [\text{wr } \langle x \rangle.\text{next}]$$

Finally we get the desired postcondition using the ordinary consequence rule:

$$\{x \neq \text{null} \wedge \theta_1 \wedge r'.\text{next} \subseteq r\} x.\text{next} := y \{r.\text{next} \subseteq r\} [\text{wr } \langle x \rangle.\text{next}]$$

(using $r'.\text{next} \subseteq r$, $r = r' \cup x$, $x.\text{next} = y$, and $y \in r$). Observe that we achieve reasoning in the manner of separation logic, less beautiful in appearance but using more elementary means.

Effects such as $\text{wr } \langle x \rangle.\text{next}$ and $\text{wr } r.\text{next}$ depend on state (since x and r are mutable variables). This leads to a difficulty with sequential composition. Several straightforward proof steps yield the following correctness statement:

$$\begin{aligned} & \{x \neq \text{null} \wedge \text{inv}\} \\ & \text{tmp} := x.\text{next}; r' := r - \langle x \rangle; x.\text{next} := y; y := x \\ & \{y = x \wedge \text{inv} \wedge \langle \text{tmp} \rangle \subseteq r\} \\ & [\text{wr tmp}, r', y, \text{wr } \langle x \rangle.\text{next}] \end{aligned}$$

But the loop body has one further assignment, $x := \text{tmp}$, to follow the command above. This interferes with the effect $\text{wr } \langle x \rangle.\text{next}$! The sequence rule imposes the condition that the first command’s effect, $\text{wr tmp}, r', y, \text{wr } \langle x \rangle.\text{next}$ should be *immune* from the second command’s effect, $\text{wr } x$, and it is not. So the write to this object’s *next* field must be recorded using some other region expression. Us-

$$\begin{aligned}
x &\in \text{VarName} & f &\in \text{FieldName} & k &\in \text{ProcName} \\
T &::= \text{int} \mid N \mid \text{rgn} & & & & \text{where } N \in \text{ClassName} \\
E &::= x \mid c \mid \text{null} \mid E \oplus E & & & & \text{where } c \in \mathbb{Z}, \oplus \in \{+, -, =\} \\
R &::= x \mid x.f \mid \langle E \rangle \mid \text{emp} \mid R \otimes R & & & & \text{where } \otimes \in \{\cup, \cap, -\} \\
F &::= E \mid R \\
C &::= x := F \mid x := \text{new } N \mid x := x.f \mid x.f := F \\
& \mid \text{if } x \text{ then } C \text{ else } C \mid \text{while } x \text{ do } C \mid C ; C \\
& \mid \text{var } x : T \text{ in } C \text{ end} \mid x += R \mid x.f += R
\end{aligned}$$

Figure 1. Programming language. As usual we confuse category names with typical elements (e.g., T). Typical elements of VarName are x, y, r . Typical field names are f, g, rf, rg . In Sec. 6, $C ::= k$ is added.

ing $x \in r$ and a rule of sub-effecting, the effect can be rewritten to $\text{wr } tmp, r', y, \text{wr } r.nxt$, which is immune from $\text{wr } x$, so the sequence rule can be used to yield $\{x \neq \text{null} \wedge \text{inv}\}$
 $tmp := x.nxt; r' := r - \langle x \rangle; x.nxt := y; y := x; x := tmp$
 $\{\text{inv}\}$
 $[\text{wr } tmp, r', y, x, \text{wr } r.nxt]$
 (where we also use consequence on the postcondition).

Notice that we could do the effect subsumption already for $x.nxt := y$ alone, but then the effect $\text{wr } r.nxt$ would have prevented framing to introduce $r'.nxt \subseteq r$.

Finally, the local variable blocks can be introduced, so that tmp and r' are scoped inside the loop and their effects can be forgotten; the loop's effect is just $\text{wr } y, x, \text{wr } r.nxt$.

2. Programming language

This section presents the language for which we formalize a 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 $\text{class } N \{ \bar{f} : \bar{T} \}$ introduces a type name N ; values of this type are **null** and references to mutable objects with typed fields $\bar{f} : \bar{T}$. (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. Unlike Java but as usual in separation logics, expressions 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. Primitive $x := \text{new } N$ allocates a fresh object of type N .

In addition to **int** and reference types there is an unusual type **rgn** with values ranging over finite sets of references. There is at least one class declaration, for the distinguished name **Obj**. For simplicity we omit general inheritance but retain it in a very restricted form adequate for our purposes. The subtyping relation \leq is the smallest reflexive relation such that $N \leq \text{Obj}$ for all class names N . All classes have the fields declared in **Obj**. The reason to include this minimal form of subtyping is to reconcile regions, which are untyped sets of references, with types.

There is no need for syntax to distinguish between ghost fields and variables (i.e., those that instrument the program, or sometimes just its assertions, to facilitate reasoning) and ordinary ones. Such use of auxiliary state dates back to the 1970s [26, 24]. However, there is no way for region expressions to influence control flow or the value of non-region fields/variables. Region expressions do include a form that reads one step into the heap, namely $x.f$ when f has type **rgn**; regions do not have fields so $x.f.g$ would then not be well formed. Region expressions include set operations like subtraction ($-$). The *singleton region* $\langle E \rangle$ is, roughly, the singleton set containing the reference denoted by E ; but if E is null then $\langle E \rangle$ is empty.

The primitive command $x += R$ is equivalent to $x := x \cup R$ but is included because it admits a more precise effect; similarly for $x.f += R$.

Since there are no binding constructs for region expressions, and field identifiers are considered disjoint from variable identifiers, we can define $\text{Vars}(R)$, the variables that occur in R , by $\text{Vars}(rv) = \{rv\}$, $\text{Vars}(x.rf) = \{x\}$, $\text{Vars}(\langle E \rangle) = \text{Vars}(E)$, and otherwise as the union of variables of subexpressions.

Typing There is an ambient class table comprised of a well formed collection of class declarations. We write $\text{fields}(N)$ for the field declarations $\bar{f} : \bar{T}$ of class N . The judgement $\Gamma \vdash F : T$ says that region or ordinary expression F has type T , and $\Gamma \vdash C$ 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 R and those of other type. The rule for singleton regions enforces that E is a reference type. Field dereferencing is allowed only for fields of type **rgn**.

$$\frac{\Gamma \vdash E : N}{\Gamma \vdash \langle E \rangle : \text{rgn}} \quad \frac{\Gamma(x) = N \quad (rf : \text{rgn}) \in \text{fields}(N)}{\Gamma \vdash x.rf : \text{rgn}}$$

Recall that metavariable N 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 $\frac{\Gamma \vdash F : T'}{\Gamma, x : T \vdash F : T'}$ and it cannot be used with x that is in $\text{dom}(\Gamma)$, because the comma in $\Gamma, x : T$ union of disjoint partial functions.

The grammar is slightly delicate. Ordinary expressions E , which will be of some class type N or else **int**, do not include field dereference. Region expressions R include field dereference in the special case $x.f$ only. This still leaves the ambiguity that our notation $x := y.f$ could be interpreted as field read or as assignment of a region expression—and

the semantics is different¹ since null-dereference is not an error in case of region fields. The typing rules eliminate the ambiguity:

$$\frac{\Gamma(y) = N \quad (f : T) \in \text{fields}(N) \quad T \neq \mathbf{rgn} \quad T \leq \Gamma(x)}{\Gamma \vdash x := y.f}$$

$$\frac{\Gamma \vdash F : T \quad T \leq \Gamma(x)}{\Gamma \vdash x := F}$$

$$\frac{\Gamma(x) = N \quad (f : T) \in \text{fields}(N) \quad \Gamma \vdash F : T' \quad T' \leq T}{\Gamma \vdash x.f := F}$$

Semantics. We use a straightforward denotational semantics where commands denote deterministic state transformers, which fits well with pre/post specifications. Some details are adapted from Rosenberg et al. [15], because their model has been encoded in PVS, including hooks to add types like **rgn** and operations on ghost state, and we plan to machine-check the results of this paper.

We assume given a set Ref of reference values, disjoint from the integers, and a distinguished value nil not in Ref or \mathbb{Z} . The values denoted by a reference type N include nil as well as all references that have been allocated for objects of type N . A *state* for context Γ has the form (ρ, h, s) where ρ is the reference typing, a partial function mapping the allocated references to their types; heap h maps each allocated reference to its object state (i.e., map from field names to values); and the *store* s maps each variable x in Γ to its value. Throughout the paper, all states are assumed to be well formed in the sense that, in (ρ, h, s) if $\rho(o)$ is some class N then o is in $\text{dom}(h)$ and $h(o)$ is a record with exactly the fields of N and moreover every field or variable value has the right type relative to ρ . The values of type N are all $o \in \text{dom}(\rho)$ such that $\rho(o) = N$, together with the distinguished value nil . The values of type **rgn** are all finite subsets of $\text{dom}(\rho)$. 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 $\llbracket \Gamma \vdash C \rrbracket \sigma$, for Γ -state σ , is either \perp (fault), \perp (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 boring complications about definedness).² The compound commands like sequence and loops are strict

¹ This design decision keeps the program semantics true to Java-like languages but streamlines reasoning about regions in assertions.

² The CoreJML work [15] models exceptions in full generality, whereas here errors are the only exception and cannot be handled. For a recent discussion of definedness issues in partial correctness logic, see [12].

$$\begin{aligned} \llbracket r \rrbracket \sigma &= \sigma(r) \\ \llbracket x.rf \rrbracket \sigma &= \sigma(x.rf) \text{ if } \sigma(x) \neq \text{nil}, \text{ else } \emptyset \\ \llbracket \langle E \rangle \rrbracket \sigma &= \{ \llbracket E \rrbracket \sigma \} \text{ if } \llbracket E \rrbracket \sigma \neq \text{nil}, \text{ else } \emptyset \\ \llbracket R_1 \cup R_2 \rrbracket \sigma &= \llbracket R_1 \rrbracket \sigma \cup \llbracket R_2 \rrbracket \sigma \\ \llbracket R_1 \cap R_2 \rrbracket \sigma &= \llbracket R_1 \rrbracket \sigma \cap \llbracket R_2 \rrbracket \sigma \\ \llbracket \mathbf{emp} \rrbracket \sigma &= \emptyset \\ \llbracket R_1 - R_2 \rrbracket \sigma &= \llbracket R_1 \rrbracket \sigma - \llbracket R_2 \rrbracket \sigma \end{aligned}$$

Figure 2. Semantics of region expressions (eliding $\Gamma \vdash \dots : \mathbf{rgn}$).

in \perp as well as in \perp . To be precise, the semantics is defined by induction on typing derivations.

We let σ range over states, and use the following abbreviations for readability. For state $\sigma = (\rho, h, s)$, write $\sigma(x)$ for $s(x)$ —variable lookup
 $\sigma(x.f)$ for $h(s(x))(f)$ —field of object referenced by a variable
 $\sigma(o.f)$ for $h(o)(f)$ —field of literal reference
 $\text{alloc}(\sigma)$ for $\text{dom}(\rho)$ —set of all allocated references
 $\text{type}(o, \sigma)$ for $\rho(o)$ —type of an allocated reference
 $\text{update}(\sigma, x, v)$ for (ρ, h, s') where s' overrides s to map x to v
 $\text{extend}(\sigma, x, v)$ for (ρ, h, s')
 where s' extends s to map x to value v (for $x \notin \text{dom}(s)$)
 $\text{update}(\sigma, o.f, v)$ for (ρ, h', s)
 where h' overrides h to map field f of reference o to v .

Note that metavariables x, y, z range over variable names, whereas we use o, p, q for elements of Ref . For ordinary expressions E we define $\llbracket \Gamma \vdash x : T \rrbracket \sigma = \sigma(x)$, $\llbracket \Gamma \vdash c : \mathbf{int} \rrbracket \sigma = c$, etc. The semantic definitions are straightforward and omitted; we also write just $\llbracket E \rrbracket$ sometimes. Note that $\llbracket E \rrbracket \sigma$ is always a value (of appropriate type), never \perp or \perp ; moreover it only depends on the store, not the heap. As an example, the semantics of simple assignment is

$$\llbracket \Gamma \vdash x := E \rrbracket \sigma = \text{update}(\sigma, x, \llbracket E \rrbracket \sigma)$$

Semantics for commands is routine and omitted. 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 **rgn**. As mentioned earlier, that case is actually parsed as $x := R$ and uses the semantics of region expressions given in Fig. 2.

Although we do not use a partial-heap semantics, we do need the straightforward property that if a fresh variable is added to the state space of command C then C acts as the identity on that variable: $\llbracket \Gamma, x : T \vdash C \rrbracket \sigma = \sigma'$ iff $\llbracket \Gamma \vdash C \rrbracket (\sigma - x) = \sigma'_1$ where $\sigma - x$ is σ with x dropped and $\sigma' = \text{extend}(\sigma'_1, x, \sigma(x))$

3. Assertion language

The assertion language includes first order quantifiers. If the bound variable is a reference type N , quantification is over currently allocated objects, as is usual and most useful [25, 11]. There are also atomic formulas for inclusion and

disjointness of regions. The formula $R_1.f \subseteq R_2$ says that for every object in R_1 , if it has field f with non-null value then that value is in R_2 . The formula $x.f = E$ is like the points-to predicate in object-oriented separation logic [8]; it says that x is non-null and the value of its f field is E . Here is the grammar.³

$$\theta ::= E = E \mid x.f = E \mid R \subseteq R \mid R \# R \mid \mathbf{type}(N, R) \\ \mid R.f \subseteq R \mid R.f \# R \mid \forall x: T \cdot \theta \mid \theta \wedge \theta \mid \neg\theta$$

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 $E \in R \hat{=} E \neq \mathbf{null} \wedge \langle E \rangle \subseteq R$. We cannot write $x.f \in R_2$ because $x.f$ is not an expression (unless $f: \mathbf{rgn}$, but regions are not nested). But it is safe to abbreviate $E.f \in R \hat{=} E.f \neq \mathbf{null} \wedge \langle E \rangle.f \subseteq R$. Another convenient feature is the ability to refer to all fields, as in $R.\mathbf{any} \subseteq R$; we use it in examples but omit the formalization.

One might expect that the set inclusions and set operations can be eliminated using quantifiers, e.g., $R.f \subseteq R'$ would be equivalent to $\forall x, y: \mathbf{Obj} \cdot x \in R \wedge x.f = y \Rightarrow y \in R'$. Technically this does not work because we adopt a typing rule for $x.f = y$ that requires x to have some type N that declares field f . (One can easily imagine variations on the assertion language, e.g., typed regions, or else dropping the type restriction on $x.f = y$ and using a semantics like that for $R_1.f = R_2$.) Without the possibility of desugaring the region formulas, we actually need to introduce further variations like $R_1.f \# R_2.g$, but this should not obtrude in the sequel. Atomic formulas involving regions admit convenient rules for expressing independence from effects. They are also useful for automation using decision procedures for fragments of set theory—an important feature of our logic but beyond the scope of this paper.

The judgement $\Gamma \vdash \theta$ expresses well formedness. We omit most typing rules since they are straightforward. Note the following:

$$\frac{\Gamma \vdash x: N \quad (f: T) \in \mathbf{fields}(N) \quad \Gamma \vdash F: T}{\Gamma \vdash x.f = F}$$

$$\frac{\Gamma \vdash R: \mathbf{rgn} \quad \Gamma \vdash R': \mathbf{rgn} \quad (f: N) \in \mathbf{fields}(N')}{\Gamma \vdash R.f \subseteq R'}$$

$$\frac{\Gamma, x: T \vdash \theta \quad T \neq \mathbf{rgn}}{\Gamma \vdash \forall x: T \cdot \theta} \quad \frac{\Gamma \vdash \theta}{\Gamma, x: T \vdash \theta}$$

The first rule ensures that x is of class type and the second rule (like that for $\Gamma \vdash R.f \# R'$) ensures that f is of class type. The third disallows quantification over regions. As with expressions (and commands, though we omitted the rule), the context for a formula can be extended. To streamline the treatment of local variables and quantifiers, we as-

³ A boring technicality: an integer expression like $x > 3$ (which evaluates to 1 or 0) can be lifted to the formula $(x > 3) \neq 0$.

sume 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 $\Gamma \vdash \theta$ is given as a satisfaction relation, written $\sigma \models^\Gamma \theta$ 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. This use of the term “valid” is appropriate because we consider a fixed model rather than a class of models.

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

$$x.f = y \Rightarrow x \neq \mathbf{null} \\ x = \mathbf{null} \Leftrightarrow \langle x \rangle = \emptyset \text{ and } x \neq \mathbf{null} \Leftrightarrow x \in \langle x \rangle \\ x = \mathbf{null} \Rightarrow x.rf \# R \\ x \in R_1 \wedge R_1.f \subseteq R_2 \Rightarrow \langle x \rangle.f \subseteq R_2 \\ R \# (R_1 \cup R_2) \Leftrightarrow R \# R_1 \wedge R \# R_2 \\ x \in R_1 \wedge y \in R_2 \wedge R_1 \# R_2 \Rightarrow x \neq y \\ x.f = y \Rightarrow \langle x \rangle.f \subseteq \langle y \rangle \quad (\text{for } f, y: N)$$

The last can be strengthened to $x.f = y \Rightarrow \langle x \rangle.f = \langle y \rangle$ using an evident syntax sugar for the consequent.

If f is in $\mathbf{fields}(\mathbf{Obj})$, like the owner field in Boogie [16], then a simple form of ownership separation is expressed by this valid formula: $R.f \subseteq \langle x \rangle \wedge R'.f \subseteq \langle y \rangle \wedge x \neq y \Rightarrow R \# R'$, provided we maintain invariant that all f fields are non-null.

Making footprints explicit. The examples hinted that the “footprint” of an assertion of interest may be made explicit in a ghost field, say $x.rep$. Kassios [14] expresses the fact that the footprint of θ is some region R using a state predicate “ R frames θ ”, the definition of which involves second order quantification. In his examples, this predicate is used as a global invariant (an axiom, in his terms). We eschew higher order logic but formalize rules for proving that “ R frames θ ” is valid.

Given context Γ , formula θ , region expression R , and list of variables \bar{x} such that $\bar{x} \subseteq \mathbf{dom}(\Gamma)$ and θ, R are well formed in Γ , we write $R; \bar{x} \mathbf{frames} \theta$ for the judgement that says truth of θ can only be changed by updates to \bar{x} , updates to objects in R , or new objects added to R . Rules for this judgement appear in Fig. 4. It is a simple read effect analysis.⁴ The semantic property is connected with independence, in Sec. 4.2.

On separating conjunction. In separation logic, the separated conjunction $\theta_1 * \theta_2$ says that θ_1 and θ_2 are both true, and moreover their truth is supported by disjoint regions of the heap. In “classical” separation logic, it says in addition that these disjoint regions cover the entire heap; the intuitionistic

⁴ One can imagine a more precise notion like $\bar{x}, \mathbf{al} R_0, R_1.f, R_2.g, \dots \mathbf{frames} \theta$, meaning that θ may be falsified by updates of the variables, allocation into R_0 , writes of f -fields of pre-existing objects in R_1 , etc. But to investigate the idea we avoid this complication.

$\sigma \models E_1 = E_2$	iff	$\llbracket E_1 \rrbracket \sigma = \llbracket E_2 \rrbracket \sigma$
$\sigma \models x.f = E$	iff	$\sigma(x) \neq \text{nil}$ and $\sigma(x.f) = \llbracket E \rrbracket \sigma$
$\sigma \models R_1 \# R_2$	iff	$\llbracket R_1 \rrbracket \sigma \cap \llbracket R_2 \rrbracket \sigma = \emptyset$
$\sigma \models R_1 \subseteq R_2$	iff	$\llbracket R_1 \rrbracket \sigma \subseteq \llbracket R_2 \rrbracket \sigma$
$\sigma \models R_1.f \subseteq R_2$	iff	for all $o \in \llbracket R_1 \rrbracket \sigma$ with $(f : N) \in \text{fields}(\text{type}(o, \sigma))$, if $\sigma(o.f) \neq \text{nil}$ then $\sigma(o.f) \in \llbracket R_2 \rrbracket \sigma$
$\sigma \models R_1.f \# R_2$	iff	for all $o \in \llbracket R_1 \rrbracket \sigma$ with $(f : N) \in \text{fields}(\text{type}(o, \sigma))$, $\sigma(o.f)$ is not in $\llbracket R_2 \rrbracket \sigma$
$\sigma \models \text{type}(N, R)$	iff	$\text{type}(o, \sigma) \leq N$ for all $o \in \llbracket R \rrbracket \sigma$
$\sigma \models \theta_1 \wedge \theta_2$	iff	$\sigma \models \theta_1$ and $\sigma \models \theta_2$
$\sigma \models \neg \theta$	iff	$\sigma \not\models \theta$
$\sigma \models^\Gamma \forall x : N \cdot \theta$	iff	$\text{extend}(\sigma, x, o) \models^{\Gamma, x : N} \theta$ for all o in $\text{alloc}(\sigma)$ with $\text{type}(o, \sigma) \leq N$
$\sigma \models^\Gamma \forall x : \text{int} \cdot \theta$	iff	$\text{extend}(\sigma, x, v) \models^{\Gamma, x : \text{int}} \theta$ for all $v \in \mathbb{Z}$
$\sigma \models^{\Gamma, x : T} \theta$	iff	$\sigma - x \models^\Gamma \theta$

Figure 3. Semantics of formulas. Note that for $R_1.f \subseteq R_2$ and $R_1.f \# R_2$, the quantification is over reference type fields $f : N$ only. The last line is for the context-extension typing rule; notation $\sigma - x$ removes variable x from the state.

$$\frac{R \cup \langle y \rangle; \bar{x}, y \text{ frames } \theta}{R; \bar{x} \text{ frames } (\forall y : N \cdot y \in R \Rightarrow \theta)}$$

$$\frac{\text{Vars}(E) \subseteq \bar{x} \quad y \in \bar{x}}{\langle y \rangle; \bar{x} \text{ frames } y.f = E}$$

$$\frac{\text{Vars}(R_1, R_2) \subseteq \bar{x} \quad \text{pivots}(R_1, R_2) = \{y_1, \dots, y_i\}}{R_1 \cup \langle y_1 \rangle \cup \dots \cup \langle y_i \rangle; \bar{x}, y_1, \dots, y_i \text{ frames } R_1.f \subseteq R_2}$$

$$\frac{R; \bar{x} \text{ frames } \theta_1 \quad R; \bar{x} \text{ frames } \theta_2}{R; \bar{x} \text{ frames } \theta_1 \wedge \theta_2}$$

$$\frac{R; \bar{x} \text{ frames } \theta}{R; \bar{x} \text{ frames } \neg \theta} \qquad \frac{R; \bar{x} \text{ frames } \theta}{R \cup R'; \bar{x}, x \text{ frames } \theta}$$

Figure 4. Framing rules. Here $\text{pivots}(R)$ is defined to be the set of x such that $x.rf$ occurs in R for some rf . The rule for $R_1.f \# R_2$ is like that for $R_1.f \subseteq R_2$.

version of $*$ allows there to be objects outside the footprint of θ_1 and θ_2 . We can encode the intuitionistic reading. Suppose we have $Q; \bar{x} \text{ frames } \theta_1$ and $R; \bar{x} \text{ frames } \theta_2$. Then $\theta_1 * \theta_2$ amounts to $\theta_1 \wedge \theta_2 \wedge Q \# R$.

Other encodings of separation are possible in our logic. For example, let *left* and *right* be two global variables (of type **Obj**) and let $\text{left} \neq \text{right} \wedge \text{left} \neq \text{null} \wedge \text{right} \neq \text{null}$ be invariant. (One might think of *left* and *right* as permissions or capabilities —this aspect of our logic will be explored elsewhere.) Suppose every object has ghost field *where* : **Obj**. Define relativized versions of the atomic formulas, e.g.,

$$\text{points}(x, f, y, wh) \hat{=} x.\text{where} = wh \wedge x.f = y$$

$$\text{within}(R, f, Q, wh) \hat{=} R.\text{where} \subseteq \langle wh \rangle \wedge R.f \subseteq Q$$

Suppose θ_1 and θ_2 do not depend on the heap except by using *points* and *within* with *wh* a free variable throughout. Then we can encode something like $\theta_1 * \theta_2$ by $\theta_1 / wh \rightarrow \text{left} \wedge \theta_2 / wh \rightarrow \text{right}$.

These encodings do not capture separating conjunction exactly, since we do not have “imprecise” predicates for which the supporting part of the heap is not uniquely determined. (Thus we avoid conundra caused by the implicit existential quantifier in $*$.)

4. Effects, independence, and immunity

4.1 Effects

An *effect set* is a comma-separated list $\bar{\varepsilon}$ of *effects*, ε , with grammar

$$\varepsilon ::= \text{al } R \mid \text{wr } x \mid \text{wr } R.f \mid \text{wr}^+ x \mid \text{wr}^+ R.f$$

The idea is that $\text{al } R$ allows allocation of new objects that are in R , $\text{wr } x$ allows update of variable x , $\text{wr } R.f$ allows update of the f field of objects in R . The forms $\text{wr}^+ x$ and $\text{wr}^+ R.f$ are for $x : \text{rgn}$ and $f : \text{rgn}$; they say that the update is allowed only by adding references to the region.

DEFINITION 4.1 (well formed effect). An effect ε is *well formed* in Γ iff one of the following holds:

- ε is $\text{al } R$ and R is well formed in Γ
- ε is $\text{wr } x$ and $x \in \text{dom}(\Gamma)$
- ε is $\text{wr } R.f$ and R is well formed in Γ
- ε is $\text{wr}^+ x$ and $(x : \text{rgn}) \in \Gamma$
- ε is $\text{wr}^+ R.f$ and f is a field of type rgn

The fat dot in $\text{wr } R.f$ is intended to be reminiscent of field access yet distinct since it refers to field f of potentially many objects in R . For example, R can have the form $x.rf$ and then we have effect $\text{wr } x.rf.f$; but $\text{wr } x.rf.f$ is not well formed.⁵

⁵Uniformity suggests a fat dot in the syntax of formulas like $R.f \subseteq R'$ but there we saw less chance for confusion.

We say σ' extends σ provided $\text{alloc}(\sigma) \subseteq \text{alloc}(\sigma')$ and $\text{type}(o, \sigma) = \text{type}(o, \sigma')$ for all $o \in \text{alloc}(\sigma)$. The semantics has the property that σ' extends σ whenever $\sigma' = \llbracket C \rrbracket \sigma$.

DEFINITION 4.2 (allows transition). Let effect set $\bar{\varepsilon}$ be well formed in Γ and let σ, σ' be Γ -states. We say $\bar{\varepsilon}$ allows transition from σ to σ' , written $\sigma \rightarrow \sigma' \models \bar{\varepsilon}$, iff σ' extends σ and the following all hold:

- (a) for every y in $\text{dom}(\Gamma)$ we have either
 1. $\sigma(y) = \sigma'(y)$,
 2. $\text{wr } y$ is in $\bar{\varepsilon}$, or
 3. $\text{wr}^+ y$ is in $\bar{\varepsilon}$ and $\sigma(y) \subseteq \sigma'(y)$
- (b) for every $o \in \text{alloc}(\sigma)$ and every $f \in \text{fields}(\text{type}(o, \sigma))$, either
 1. $\sigma(o.f) = \sigma'(o.f)$,
 2. there is $\text{wr } R.f$ in $\bar{\varepsilon}$ such that $o \in \llbracket R \rrbracket \sigma'$, or
 3. there is $\text{wr}^+ R.f$ in $\bar{\varepsilon}$ such that $o \in \llbracket R \rrbracket \sigma'$ and $\sigma(o.f) \subseteq \sigma'(o.f)$,
- (c) for every o in $\text{alloc}(\sigma') - \text{alloc}(\sigma)$, there is some $\text{al } R$ in $\bar{\varepsilon}$ such that $o \in \llbracket R \rrbracket \sigma'$.

Note that in (a3) y has type **rgn** by well-formedness of $\text{wr}^+ y$.

Sub-effects. An effect set $\bar{\varepsilon}, \varepsilon$, with ε added to $\bar{\varepsilon}$, allows at least the effects allowed by $\bar{\varepsilon}$. In the case of an effect like $\text{wr } R.f$, there is also the possibility of more liberal effect $\text{wr } R'.f$ in case $R \subseteq R'$. Since regions can be state-dependent, such inclusions are state-dependent, so we use a judgement $\theta \vdash \bar{\varepsilon} \leq \bar{\varepsilon}'$ to express that, when the post-state satisfies θ , the effects allowed by $\bar{\varepsilon}$ are included in those allowed by $\bar{\varepsilon}'$. The inductive definition is given by Fig. 5. Since effects are treated as sets, ε, ε is the same as ε .

LEMMA 4.3 (sub-effect). If $\theta \vdash \bar{\varepsilon}_1 \leq \bar{\varepsilon}_2$ and $\sigma \rightarrow \sigma' \models \bar{\varepsilon}_1$ and $\sigma' \models \theta$ then $\sigma \rightarrow \sigma' \models \bar{\varepsilon}_2$.

4.2 Independence

The *independence judgement* $\theta' \vdash \theta \# \varepsilon$ is intended to express that the truth of θ is not affected by changing the state to one where θ' holds, with the change admitted by effect ε . This is used in the frame rule, to express that θ is not falsified by a command that does not touch the footprint of θ .

DEFINITION 4.4 (independence). An ordinary expression E is *independent from* effect ε , written $E \# \varepsilon$, iff for all $x \in \text{Vars}(E)$, ε is not $\text{wr } x$ or $\text{wr}^+ x$. A region expression R is *independent from* effect ε , written $R \# \varepsilon$, iff the following both hold:

- if ε has the form $\text{wr } x$ or $\text{wr}^+ x$ then $x \notin \text{Vars}(R)$
- if ε has the form $\text{wr } R'.rf$ or $\text{wr}^+ R'.rf$ then field rf does not occur in R

$$R_1 \subseteq R_2 \vdash \text{wr } R_1.f \leq \text{wr } R_2.f \qquad \text{true} \vdash \bar{\varepsilon} \leq \bar{\varepsilon}, \varepsilon$$

$$R_1 \subseteq R_2 \vdash \text{wr}^+ R_1.f \leq \text{wr}^+ R_2.f$$

$$R_1 \subseteq R_2 \vdash \text{al } R_1 \leq \text{al } R_2$$

$$\text{true} \vdash \text{wr}^+ R.f \leq \text{wr } R.f \qquad \text{true} \vdash \text{wr}^+ r \leq \text{wr } r$$

$$\text{true} \vdash \text{wr } R_1.f, \text{wr } R_2.f \leq \text{wr } (R_1 \cup R_2).f$$

$$\text{true} \vdash \text{al } R_1, \text{al } R_2 \leq \text{al } R_1 \cup R_2 \qquad \text{true} \vdash \bar{\varepsilon} \leq \bar{\varepsilon}$$

$$\frac{\theta \vdash \bar{\varepsilon}_1 \leq \bar{\varepsilon}_2 \quad \theta \vdash \bar{\varepsilon}_2 \leq \bar{\varepsilon}_3}{\theta \vdash \bar{\varepsilon}_1 \leq \bar{\varepsilon}_3}$$

$$\frac{\theta' \Rightarrow \theta \quad \theta \vdash \bar{\varepsilon} \leq \bar{\varepsilon}'}{\theta' \vdash \bar{\varepsilon} \leq \bar{\varepsilon}'} \qquad \frac{\theta \vdash \bar{\varepsilon}_1 \leq \bar{\varepsilon}_2}{\theta \vdash \varepsilon, \bar{\varepsilon}_1 \leq \varepsilon, \bar{\varepsilon}_2}$$

Figure 5. Selected sub-effect rules. We write \leq to abbreviate two inclusion rules. Union rules for wr^+ – are omitted.

Expression E is independent from effect set $\bar{\varepsilon}$, written $E \# \bar{\varepsilon}$, iff it is independent from each ε in $\bar{\varepsilon}$, and *mutatis mutandis* for R .

Finally, we say θ is *independent from $\bar{\varepsilon}$ modulo θ'* , written $\theta' \vdash \theta \# \bar{\varepsilon}$, if and only if $\theta' \vdash \theta \# \varepsilon$ for each ε in $\bar{\varepsilon}$, where $\theta' \vdash \theta \# \varepsilon$ is defined by rules⁶ in Figs. 6 and 7. \square

For example, region expression $x.rf$ is not independent from $\text{wr } R.rf$. On the other hand, we have $x.rf \# R \vdash x.rf.f \leq R' \# \text{wr } R.f$ (for any R').

The second rule for \forall can be used to justify reasoning about quantifications over “owned” objects, e.g., $\forall y: N \cdot y \in x.rep \Rightarrow \dots$ where $x.rep$ is a region field containing objects owned by x . This could be pinned down further if there is an *own* field: $x.rep.own \subseteq \langle x \rangle$.

The judgement $R; \bar{x}$ **frames** θ says that θ is independent from the complement of R , in all states. Using complement in such a way would force us to reason non-locally about the entire program state. But when something is known about the complement, a connection can be made, by the rules in Fig.7.

Suppose $Q; \bar{x}$ **frames** θ . If $\theta' \vdash Q \# R$, where R is the union of the regions in $\bar{\varepsilon}$, and $\bar{x} \# \bar{\varepsilon}$, then the rules of Fig. 7 yield $\theta' \vdash \theta \# \bar{\varepsilon}$.

LEMMA 4.5 (independence). Suppose $\sigma \rightarrow \sigma' \models \bar{\varepsilon}$.

- if $E \# \bar{\varepsilon}$ then $\llbracket E \rrbracket \sigma = \llbracket E \rrbracket \sigma'$;
- if $R \# \bar{\varepsilon}$ then $\llbracket R \rrbracket \sigma = \llbracket R \rrbracket \sigma'$; and

⁶As usual, the rules may be instantiated only when each $\theta' \vdash \theta \# \varepsilon$ is well formed in the sense that there is a context Γ in which θ, θ' , and ε are well formed. For $\forall x: T \cdot \theta$, this means that x cannot occur in ε or in θ' .

$$\begin{array}{c}
\frac{E \# \varepsilon \quad E' \# \varepsilon}{true \vdash E = E' \# \varepsilon} \quad \frac{R \# \varepsilon}{true \vdash \mathbf{type}(N, R) \# \varepsilon} \\
\\
\frac{R \# \varepsilon \quad R' \# \varepsilon}{true \vdash R \# R' \# \varepsilon} \quad \frac{R \# \varepsilon \quad R' \# \varepsilon}{true \vdash R \subseteq R' \# \varepsilon} \\
\\
\frac{x \# \varepsilon \quad E \# \varepsilon \quad \text{if } \varepsilon \text{ writes } R.f \text{ then } \theta' \Rightarrow \langle x \rangle \# R}{\theta' \vdash x.f = E \# \varepsilon} \\
\\
\frac{\text{if } \varepsilon \text{ writes } R.f \text{ then } \theta' \Rightarrow R \# R_1 \vee R.f \subseteq R_2 \quad R_1 \# \varepsilon \quad R_2 \# \varepsilon}{\theta' \vdash R_1.f \subseteq R_2 \# \varepsilon} \\
\\
\frac{\text{if } \varepsilon \text{ writes } R.f \text{ then } \theta' \Rightarrow R \# R_1 \vee R.f \# R_2 \quad R_1 \# \varepsilon \quad R_2 \# \varepsilon}{\theta' \vdash (R_1.f \# R_2) \# \varepsilon} \\
\\
\frac{\theta' \vdash^{\Gamma, x:T} \theta \# \varepsilon \quad \text{either } T \equiv \mathbf{int} \text{ or } \varepsilon \text{ is not of the form } \mathbf{al} R}{\theta' \vdash^{\Gamma} \forall x : T \cdot \theta \# \varepsilon} \\
\\
\frac{\theta' \vdash^{\Gamma, x:T} \theta \# \mathbf{al} R' \quad \theta' \Rightarrow R \# R'}{\theta' \vdash^{\Gamma} (\forall x : T \cdot x \in R \Rightarrow \theta) \# \mathbf{al} R'} \quad \frac{\theta' \vdash \theta \# \varepsilon}{\theta' \vdash \neg \theta \# \varepsilon} \\
\\
\frac{\theta' \vdash \theta_1 \# \varepsilon \quad \theta' \wedge \theta_1 \vdash \theta_2 \# \varepsilon}{\theta' \vdash \theta_1 \wedge \theta_2 \# \varepsilon} \quad \frac{\theta' \vdash^{\Gamma} \theta \# \varepsilon}{\theta' \vdash^{\Gamma, x:T} \theta \# \varepsilon} \\
\\
\frac{\theta' \vdash \theta \# \varepsilon_2 \quad \theta'' \Rightarrow \theta' \quad \theta' \vdash \varepsilon_1 \leq \varepsilon_2}{\theta'' \vdash \theta \# \varepsilon_1}
\end{array}$$

Figure 6. Selected rules for independence. An antecedent “if ε writes $R.f$ then θ ” means that if ε is either $\mathbf{wr} R.f$ or $\mathbf{wr}^+ R.f$, for some R , then validity of θ (for that R) is required. The rule for $R_1.f \# R_2$ is like that for $R_1.f \subseteq R_2$.

- if $\theta' \vdash \theta \# \bar{\varepsilon}$ and $\sigma' \models \theta'$ then we have $\sigma \models \theta$ iff $\sigma' \models \theta$.

The rules are not complete, i.e., the semantic property does not imply the syntactic one. For example, $x = x$ is not independent from $\mathbf{wr} x$ though obviously its value is not affected by updating x . There is no need for completeness in this sense. Effects themselves are conservative in that a command may not have its advertised effect: a simple example is $x := x$ with $\mathbf{wr} x$, but also $x := x + y$ in states where $y = 0$. However, there are rules to connect assertional reasoning with effects (see Fig. 10).

Derived rules for independence. One useful rule is for implication.

$$\frac{\theta' \vdash \theta_1 \# \varepsilon \quad \theta' \wedge \theta_1 \vdash \theta_2 \# \varepsilon}{\theta' \vdash \theta_1 \Rightarrow \theta_2 \# \varepsilon}$$

$$\begin{array}{c}
\text{FR IND VAR} \quad \frac{y \notin \bar{x} \quad R; \bar{x} \text{ frames } \theta}{true \vdash \theta \# \mathbf{wr} y} \\
\\
\text{FR IND FLD} \quad \frac{R; \bar{x} \text{ frames } \theta \quad \theta' \Rightarrow R \# R'}{\theta' \vdash \theta \# \mathbf{wr} R'.f} \\
\\
\text{FR IND AL} \quad \frac{R; \bar{x} \text{ frames } \theta \quad \theta' \Rightarrow R \# R'}{\theta' \vdash \theta \# \mathbf{al} R'}
\end{array}$$

Figure 7. Frame independence rules (\mathbf{wr}^+ versions omitted).

Two others are for sugared formulas.

$$\begin{array}{c}
\frac{R_1 \# \varepsilon \quad R_2 \# \varepsilon}{\text{if } \varepsilon \text{ is } \mathbf{wr} R.f \text{ or } \mathbf{wr}^+ R.f \text{ then } \theta \Rightarrow R \# R_1 \vee R.f \# R_2.g} \\
\frac{\text{if } \varepsilon \text{ is } \mathbf{wr} R.g \text{ or } \mathbf{wr}^+ R.g \text{ then } \theta \Rightarrow R \# R_2 \vee R.f \# R_1.f}{\theta \vdash R_1.f \# R_2.g \# \varepsilon} \\
\\
\frac{x \# \varepsilon \quad \text{if } \varepsilon \text{ is } \mathbf{wr} R.f \text{ or } \mathbf{wr}^+ R.f \text{ then } \theta \Rightarrow \langle x \rangle \# R}{y \# \varepsilon \quad \text{if } \varepsilon \text{ is } \mathbf{wr} R.g \text{ or } \mathbf{wr}^+ R.g \text{ then } \theta \Rightarrow \langle y \rangle \# R} \\
\theta \vdash x.f = y.g \# \varepsilon
\end{array}$$

4.3 Immunity

A critical consequence of our decision to allow state-dependent region expressions in effects is that it introduces a novel form of interference, not between program parts or assertions but rather between effects.

The effects of one command in a sequence can invalidate the effects of its predecessor, as illustrated in the introduction. In reasoning about a sequence $C_1; C_2$ of commands, a correctness statement for C_1 includes effects $\bar{\varepsilon}$ that are interpreted in the final state of C_1 , and it is possible for C_2 to include updates of region variables/fields that alters the interpretation of $\bar{\varepsilon}$. The example in Sec. 1 also showed how the detailed effects of C_1 can be subsumed, by sub-effecting, into effects described in a more globally-meaningful way (often using the \mathbf{wr}^+ –). We define a notion of “immunity” to express that the resulting effect set is not interfered with by C_2 . An immunity condition is imposed by the rules for sequence and for **while**.

Immunity for an effect set is defined in terms of immunity for region expressions. Immunity for a region expression ensures that its value cannot decrease by the allowed updates.⁷

DEFINITION 4.6 (θ, ε -**immune**). Region expression R is said to be $\theta, \bar{\varepsilon}$ -immune iff the following all hold:

1. R has no occurrence of the subtraction operator
2. for every x with $\mathbf{wr} x$ in $\bar{\varepsilon}$, we have $x \notin \mathit{Vars}(R)$

⁷This notion is incomparable to independence. R can be independent from ε even if subtraction occurs in R , and on the other hand immunity allows non-decreasing writes.

3. for every R' such that $\mathbf{wr} R'.f$ is in $\bar{\varepsilon}$, and every sub-expression $x.rf$ of R , the formula $\theta \Rightarrow \langle x \rangle \# R'$ is valid.

Effect set $\bar{\varepsilon}_1$ is said to be $\theta, \bar{\varepsilon}_2$ -immune iff R is $\theta, \bar{\varepsilon}_2$ -immune, for every R such that $\mathbf{al} R$, $\mathbf{wr} R.f$, or $\mathbf{wr}^+ R.f$ (for some f) occurs in $\bar{\varepsilon}_1$. \square

We abbreviate $\text{true}, \bar{\varepsilon}_2$ -immune as $\bar{\varepsilon}_2$ -immune.

Let us consider some examples. First, $\mathbf{wr} x$ is $\mathbf{wr} x$ -immune but $\mathbf{wr} \langle x \rangle.f$ is not $\mathbf{wr} x$ -immune. If r has type \mathbf{rgn} then $\mathbf{wr} r.f$ is $\mathbf{wr}^+ r$ -immune although it is not $\mathbf{wr} r$ -immune.

LEMMA 4.7 Suppose R is $\theta, \bar{\varepsilon}$ -immune. Then $\llbracket R \rrbracket \sigma \subseteq \llbracket R \rrbracket \sigma'$ for any σ, σ' such that $\sigma \rightarrow \sigma' \models \bar{\varepsilon}$ and $\sigma' \models \theta$.

Proof: By structural induction on R .

Case $R \equiv r$: Thus $\mathbf{wr} r \notin \bar{\varepsilon}$ by immunity. It is possible that $\mathbf{wr}^+ r \in \bar{\varepsilon}$ (since r has type \mathbf{rgn}), but then Def. 4.2(a3) allows only $\sigma(r) \subseteq \sigma'(r)$, so $\llbracket r \rrbracket \sigma \subseteq \llbracket r \rrbracket \sigma'$.

Case $R \equiv x.rf$: Again, $\mathbf{wr} x \notin \bar{\varepsilon}$ by immunity, so $\sigma(x) = \sigma'(x)$. Let $p = \sigma(x)$. If there is R' with $\mathbf{wr} R'.f$ in $\bar{\varepsilon}$ then by immunity we have $\theta \Rightarrow \langle x \rangle \# R'$, hence $p \notin \llbracket R' \rrbracket \sigma'$ and the effect does not allow update of $p.rf$. If $\mathbf{wr}^+ R'.f$ is in $\bar{\varepsilon}$ then it is possible for $p.rf$ to change, but by Def. 4.2(b3) we have $\sigma(p.rf) \subseteq \sigma'(p.rf)$ as required.

Case $R \equiv \mathbf{emp}$ is immediate.

Cases $R \equiv R_1 \cup R_2$ and $R \equiv R_1 \cap R_2$: by induction and monotonicity of the set operation.

Case $R \equiv R_1 - R_2$: disallowed by immunity.

Case $R \equiv \langle E \rangle$: by well-formedness, E is not of region type and contains no region-type variables; so by immunity $\bar{\varepsilon}$ contains no $\mathbf{wr} x$ for $x \in \text{Vars}(E)$ and thus $\llbracket E \rrbracket \sigma = \llbracket E \rrbracket \sigma'$. \square

LEMMA 4.8 (immunity). Suppose $\bar{\varepsilon}_1$ is $\theta, \bar{\varepsilon}_2$ -immune. Suppose $\sigma \rightarrow \sigma_1 \models \bar{\varepsilon}_1$ and $\sigma_1 \rightarrow \sigma' \models \bar{\varepsilon}_2$ and $\sigma' \models \theta$. Then $\sigma \rightarrow \sigma' \models \bar{\varepsilon}_1, \bar{\varepsilon}_2$.

Proof: Under the hypotheses, we prove $\sigma \rightarrow \sigma' \models \bar{\varepsilon}_1, \bar{\varepsilon}_2$ by cases on the parts of Def. 4.2, and in the short version of the paper consider just one part. Part (a): Consider any x such that $\sigma(x) \neq \sigma'(x)$.

- Case $\sigma_1(x) \neq \sigma'(x)$: There are two effects that allow this. If $\mathbf{wr} x$ is in $\bar{\varepsilon}_2$ then $\sigma \rightarrow \sigma' \models \bar{\varepsilon}_1, \bar{\varepsilon}_2$. If $\mathbf{wr}^+ x$ is in $\bar{\varepsilon}_2$ (so x has type \mathbf{rgn}) then $\sigma_1(x) \subseteq \sigma'(x)$; if also $\sigma(x) \subseteq \sigma_1(x)$ then we are done, but if not then $\mathbf{wr} x$ is in $\bar{\varepsilon}_1$.
- Case $\sigma_1(x) = \sigma'(x)$: If $\mathbf{wr} x$ is in $\bar{\varepsilon}_1$ then $\sigma \rightarrow \sigma' \models \bar{\varepsilon}_1, \bar{\varepsilon}_2$. If $\mathbf{wr}^+ x$ is in $\bar{\varepsilon}_1$ then we have $\sigma(x) \subseteq \sigma_1(x) = \sigma'(x)$ hence $\sigma \rightarrow \sigma' \models \bar{\varepsilon}_1, \bar{\varepsilon}_2$.

Part (b): Consider any $p \in \text{alloc}(\sigma)$ and f such that $\sigma(p.f) \neq \sigma'(p.f)$. There are two subcases.

Case $\sigma_1(p.f) \neq \sigma'(p.f)$: So by $\sigma_1 \rightarrow \sigma' \models \bar{\varepsilon}_2$ we have either $\mathbf{wr} R.f$ or $\mathbf{wr}^+ R.f$ in $\bar{\varepsilon}_2$ such that $p \in \llbracket R \rrbracket \sigma'$. If it is $\mathbf{wr} R.f$ then it allows $\sigma(p.f) \neq \sigma'(p.f)$. Otherwise, for $\mathbf{wr}^+ R.f$ we need $\sigma(p.f) \subseteq \sigma'(p.f)$. We have $\sigma_1(p.f) \subseteq \sigma'(p.f)$; if not also $\sigma(p.f) \subseteq \sigma_1(p.f)$ then there is $\mathbf{wr} R'.f$ in $\bar{\varepsilon}$ such that $p \in \llbracket R' \rrbracket \sigma_1$ but then by Lemma 4.7 we get $p \in \llbracket R \rrbracket \sigma'$.

Case $\sigma_1(p.f) = \sigma'(p.f)$: So by $\sigma \rightarrow \sigma_1 \models \bar{\varepsilon}_1$ we have either $\mathbf{wr} R.f$ or $\mathbf{wr}^+ R.f$ in $\bar{\varepsilon}_1$ with $p \in \llbracket R \rrbracket \sigma_1$. By Lemma 4.7 we get $p \in \llbracket R \rrbracket \sigma'$. For the case of $\mathbf{wr} R.f$ this is enough. If it is $\mathbf{wr}^+ R.f$ in $\bar{\varepsilon}_1$ then what is allowed is $\sigma(p.f) \subseteq \sigma_1(p.f)$ and again we get $\sigma_1(p.f) \subseteq \sigma'(p.f)$. \square

Part (c): Consider any $p \in \text{alloc}(\sigma')$ such that $p \notin \text{alloc}(\sigma)$.

- Case $p \in \text{alloc}(\sigma_1)$: then there is $\mathbf{al} R$ in $\bar{\varepsilon}_1$ with $p \in \llbracket R \rrbracket \sigma_1$ (by allows (c)), so by immunity of R and Lemma 4.7 we have $p \in \llbracket R \rrbracket \sigma'$.
- Case $p \notin \text{alloc}(\sigma_1)$: then there is $\mathbf{al} R$ in $\bar{\varepsilon}_2$ with $p \in \llbracket R \rrbracket \sigma'$ and we are done. \square

Def. 4.6 is designed to give minimal, easily checked conditions sufficient for the semantic property (Lemma 4.8) needed in the proof rules for sequence and iteration. There is a stronger condition that is often satisfied. Call effect set $\bar{\varepsilon}$ *robust* provided that

- for each $\mathbf{wr} x$ in $\bar{\varepsilon}$, there is no region expression R in $\bar{\varepsilon}$ with $x \in \text{Vars}(R)$ and x is not of type \mathbf{rgn} ,
- every region expression in $\bar{\varepsilon}$ is subtraction-free, and
- if $\mathbf{wr} R.f$ occurs in $\bar{\varepsilon}$ then f is not of type \mathbf{rgn} .

The reader may care to check that if $\bar{\varepsilon}$ is robust then it is $\bar{\varepsilon}$ -immune.

5. Program correctness

A *correctness statement* takes the form $\{\theta\} C \{\theta'\} [\bar{\varepsilon}]$. The intended meaning is that from any initial state that satisfies θ , C does not fault (terminate with error), and if it terminates then the final state satisfies θ' . Moreover any allocation and update effects are allowed by $\bar{\varepsilon}$ (Def. 4.2). The statement is *well-formed in Γ* provided that θ, θ', C , and $\bar{\varepsilon}$ are well-formed in Γ .

The notation \vdash^Γ is used for provability of statements that are well formed in Γ , so the proof system derives judgements of the form $\vdash^\Gamma \{\theta\} C \{\theta'\} [\bar{\varepsilon}]$. The semantics is used to define *valid* correctness statements, for which we use notation \models^Γ . The proof rules do not include well-formedness conditions but rather are to be instantiated only with well formed statements. We often omit Γ . In a proof rule, this means all judgements are for the same Γ .

DEFINITION 5.1 (**validity**). Let $\{\theta\} C \{\theta'\} [\bar{\varepsilon}]$ be well-formed in Γ . The correctness statement is *valid*, written $\models^\Gamma \{\theta\} C \{\theta'\} [\bar{\varepsilon}]$, if and only if $\llbracket \Gamma \vdash C \rrbracket \models^\Gamma \{\theta\} - \{\theta'\} [\bar{\varepsilon}]$.

$$\begin{array}{c}
\text{ALLOC} \quad \frac{\text{fields}(N) = \bar{f} : \bar{T}}{\vdash \{ \text{true} \} x := \mathbf{new} N \{ x \neq \text{null} \wedge \langle x \rangle \# r \wedge x.\bar{f} = \overline{\text{default}(\bar{T})} \wedge \mathbf{type}(N, \langle x \rangle) \} [\mathbf{wr} x, \mathbf{al} \langle x \rangle]} \\
\text{ASSIGN} \quad \frac{y \neq x}{\vdash \{ x = y \} x := F \{ x = (F/x \rightarrow y) \} [\mathbf{wr} x]} \qquad \text{FIELDACC} \quad \frac{z \neq x}{\vdash \{ y \neq \mathbf{null} \wedge z = y \} x := y.f \{ x = z.f \} [\mathbf{wr} x]} \\
\text{FIELDUPD} \quad \vdash \{ x \neq \mathbf{null} \} x.f := F \{ x.f = F \} [\mathbf{wr} \langle x \rangle.f] \\
\text{SEQ} \quad \frac{\vdash \{ \theta \} C_1 \{ \theta_1 \} [\bar{\varepsilon}_1] \quad \vdash \{ \theta_1 \} C_2 \{ \theta' \} [\bar{\varepsilon}_2] \quad \bar{\varepsilon}_1 \text{ is } \theta', \bar{\varepsilon}_2\text{-immune}}{\vdash \{ \theta \} C_1 ; C_2 \{ \theta' \} [\bar{\varepsilon}_1, \bar{\varepsilon}_2]} \\
\text{IF} \quad \frac{\vdash \{ \theta \wedge x \neq 0 \} C_1 \{ \theta' \} [\bar{\varepsilon}] \quad \vdash \{ \theta \wedge x = 0 \} C_2 \{ \theta' \} [\bar{\varepsilon}]}{\vdash \{ \theta \} \mathbf{if} x \mathbf{then} C_1 \mathbf{else} C_2 \{ \theta' \} [\bar{\varepsilon}]} \qquad \text{WHILE} \quad \frac{\vdash \{ \theta \wedge x \neq 0 \} C \{ \theta \} [\bar{\varepsilon}] \quad \bar{\varepsilon} \text{ is } \theta, \bar{\varepsilon}\text{-immune}}{\vdash \{ \theta \} \mathbf{while} x \mathbf{do} C \{ \theta \wedge x = 0 \} [\bar{\varepsilon}]} \\
\text{VAR} \quad \frac{\vdash^{\Gamma, x:T} \{ \theta \wedge x = \text{default}(T) \} C \{ \theta' \} [\mathbf{wr} x, \bar{\varepsilon}]}{\vdash^{\Gamma} \{ \theta \} \mathbf{var} x : T \mathbf{in} C \mathbf{end} \{ \theta' \} [\bar{\varepsilon}]} \qquad \text{ASSIGN POS} \quad \frac{r \neq x}{\vdash \{ x = r \} x += R \{ x = r \cup (R/x \rightarrow r) \} [\mathbf{wr}^+ x]} \\
\text{FIELDUPD POS} \quad \vdash \{ x \neq \mathbf{null} \wedge x.rf = r \} x.rf += R \{ x.rf = r \cup R \} [\mathbf{wr}^+ \langle x \rangle.rf]
\end{array}$$

Figure 8. Proof rules and axioms for correctness.

Here we use the notation $\{ \theta \} - \{ \theta' \} [\bar{\varepsilon}]$ for a specification. And for any state transformer t (of type Γ) we define $t \models^{\Gamma} \{ \theta \} - \{ \theta' \} [\bar{\varepsilon}]$ iff for all Γ -states σ, σ' such that $\sigma \models \theta$ we have $t(\sigma) \neq \perp$ and if $t(\sigma) = \sigma'$ then $\sigma' \models \theta'$ and $\sigma \rightarrow \sigma' \models \bar{\varepsilon}$. \square

Proof rules and axioms. Fig. 8 gives the syntax-directed proof rules and axioms.⁸

The axioms for variable assignment and field update are “small” in the sense that they describe the local effect only. Small axioms are elegant and admit simple proofs of soundness.

Axiom [FieldUpd] is a little surprising. Following separation logic, the programming and assertion languages disallow expressions that dereference fields; but one step of dereferencing is allowed here since F in the rule can be of the form $x.f$ in the case that $f : \mathbf{rgn}$.

In rule [Var] the context is made explicit since the antecedent’s context is different from the consequent’s context. Because rules may only be instantiated so that all the judgments are well formed, x cannot occur in θ, θ' , or $\bar{\varepsilon}$.

⁸ An axiom may have “side conditions”, displayed as antecedents, but does not have an antecedent correctness statement. For example, axiom [Alloc] expresses freshness by the postcondition $\langle x \rangle \# R$ where R can be any region expression not containing x .

Axiom [Alloc] uses a region variable in its postcondition in order to express freshness. Note that variable r is distinct from x for reasons of typing: $r : \mathbf{rgn}$ and $x : N$. This technique is also used in the example specifications for *init* and *listcopy*. To see how it works consider for example the sequence $x := \mathbf{new} N; y := \mathbf{new} N$, for $x \neq y$. By the axiom and [Conseq] we have

$$\{ \text{true} \} x := \mathbf{new} N \{ \langle x \rangle \# r \} [\mathbf{wr} x, \mathbf{al} \langle x \rangle]$$

Using the axiom for $y := \mathbf{new} N$ and then [Subst] of $r \cup \langle x \rangle$ for r we get the following, where we abbreviate $\bar{\varepsilon}_y \equiv \mathbf{wr} y, \mathbf{al} \langle y \rangle$:

$$\{ \text{true} \} y := \mathbf{new} N \{ \langle y \rangle \# r \cup \langle x \rangle \} [\bar{\varepsilon}_y]$$

Then by framing, we get

$$\{ \langle x \rangle \# r \} y := \mathbf{new} N \{ \langle y \rangle \# r \cup \langle x \rangle \wedge \langle x \rangle \# r \} [\bar{\varepsilon}_y]$$

Using [Conseq] we can rewrite the postcondition as

$$\{ \langle x \rangle \# r \} y := \mathbf{new} N \{ x \neq y \wedge r \# (\langle x \rangle \cup \langle y \rangle) \} [\bar{\varepsilon}_y]$$

By [Seq], writing S for $x := \mathbf{new} N; y := \mathbf{new} N$, we get

$$\{ \text{true} \} S \{ x \neq y \wedge r \# (\langle x \rangle \cup \langle y \rangle) \} [\mathbf{wr} x, \mathbf{al} \langle x \rangle, \bar{\varepsilon}_y]$$

Fig. 9 gives the structural rules. Rule [SubEff] loosens the effect clause; it is used in the example of Sec. 1 to weaken

$$\begin{array}{c}
\text{SUB EFF} \quad \frac{\vdash \{\theta\} C \{\theta'\} [\bar{\varepsilon}] \quad \theta' \vdash \bar{\varepsilon} \leq \bar{\varepsilon}'}{\vdash \{\theta\} C \{\theta'\} [\bar{\varepsilon}']} \\
\text{FRAME} \quad \frac{\vdash \{\theta\} C \{\theta'\} [\bar{\varepsilon}] \quad \theta' \vdash \theta_1 \# \bar{\varepsilon}}{\vdash \{\theta \wedge \theta_1\} C \{\theta' \wedge \theta_1\} [\bar{\varepsilon}]} \\
\text{CONJ} \quad \frac{\vdash \{\theta_1\} C \{\theta'_1\} [\bar{\varepsilon}] \quad \vdash \{\theta_2\} C \{\theta'_2\} [\bar{\varepsilon}]}{\vdash \{\theta_1 \wedge \theta_2\} C \{\theta'_1 \wedge \theta'_2\} [\bar{\varepsilon}]} \\
\text{CONSEQ} \quad \frac{\vdash \{\theta_1\} C \{\theta'_1\} [\bar{\varepsilon}] \quad \theta_2 \Rightarrow \theta_1 \quad \theta'_1 \Rightarrow \theta'_2}{\vdash \{\theta_2\} C \{\theta'_2\} [\bar{\varepsilon}]} \\
\text{EXIST} \quad \frac{\vdash^{\Gamma, x:T} \{\theta\} C \{\theta'\} [\bar{\varepsilon}]}{\vdash^{\Gamma} \{\exists x: T \cdot \theta\} C \{\theta'\} [\bar{\varepsilon}]} \\
\text{CONTEXT} \quad \frac{\vdash^{\Gamma} \{\theta\} C \{\theta'\} [\bar{\varepsilon}]}{\vdash^{\Gamma, x:T} \{\theta\} C \{\theta'\} [\bar{\varepsilon}]} \\
\text{SUBST} \quad \frac{F \# \bar{\varepsilon} \quad \vdash \{\theta\} C \{\theta'\} [\bar{\varepsilon}] \quad C \text{ does not read } x \quad \bar{\varepsilon} \text{ is } \mathbf{wr} \ x\text{-immune}}{\vdash \{\theta/x \rightarrow F\} C \{\theta'/x \rightarrow F\} [\bar{\varepsilon}]}
\end{array}$$

Figure 9. Structural rules for correctness.

a specifications to get immunity for rule [Seq]. It is also needed to get the two antecedents in [If] or in [Conj] to match up.

In a case like $\mathbf{wr} \ R.f, \mathbf{wr} \ R'.f \leq \mathbf{wr} \ (R \cup R').f$ where sub-effecting holds independent of context, the use of [Sub Eff] can be commuted with other proof rules. For example, an application to a judgement for some S_1 could be deferred to apply for some $S_1; S_2$ following application of [Seq]. However, in case the sub-effecting $\varepsilon_1 \leq \varepsilon_2$ depends on an antecedent predicate θ , it could be that θ holds following S_1 but not S_2 in which case the rules could not be commuted.

Rule [Exist] is typical in Hoare logics. Some authors prefer an existential elimination rule that also quantifies the postcondition, but this can be achieved by using [Conseq] first, by $\theta \Rightarrow \exists x: T \cdot \theta$.

In [Subst] we use Reynolds' notation, writing $\theta/x \rightarrow F$ for substitution of F for x in θ . In accord with our convention on well formed rule instantiations, the result of substitution must be well formed here, e.g., $(x.rf \subseteq r)/x \rightarrow \mathbf{null}$ is not. The condition “ C does not read x ” has an easy syntactic definition for the language considered here, but becomes an issue in Sec. 6.1 where we add procedures to the language.

The remaining rules for program correctness are in Fig. 10. These allow elimination of effects based on assertional reasoning. Rule [No Assign] uses $y \# \mathbf{wr} \ x$ to ensure

$$\begin{array}{c}
\text{NO ASSIGN} \quad \frac{\vdash \{\theta\} C \{\theta'\} [\mathbf{wr} \ x, \bar{\varepsilon}] \quad \theta \vee \theta' \Rightarrow x = y \quad y \# \mathbf{wr} \ x, \bar{\varepsilon}}{\vdash \{\theta\} C \{\theta'\} [\bar{\varepsilon}]} \\
\text{NO ALLOC} \quad \frac{\vdash \{\theta\} C \{\theta'\} [\mathbf{al} \ R, \bar{\varepsilon}] \quad R \# \bar{\varepsilon}}{\vdash \{\theta\} C \{\theta'\} [\bar{\varepsilon}]} \\
\text{NO UPDATE} \quad \frac{\vdash \{\theta\} C \{\theta'\} [\mathbf{wr} \ \langle x \rangle.f, \bar{\varepsilon}] \quad x \# \bar{\varepsilon} \quad y \# \bar{\varepsilon} \quad \theta \vee \theta' \Rightarrow x.f = y}{\vdash \{\theta\} C \{\theta'\} [\bar{\varepsilon}]}
\end{array}$$

Figure 10. Effect elimination (masking) rules. Omits similar rules for $\mathbf{wr}^+ \ R.f$ and $\mathbf{wr}^+ \ r$.

that y is distinct from x . Rule [No Update] does not have that issue since y is a variable, not a field. One might hope to generalize [No Update] to region effect $\mathbf{wr} \ R.f$ but it is not clear how to express the initial-final equality for an arbitrary set of objects. For rule [No Alloc], one might guess that an equation $r = R$ would be used in pre and post, like in [No Assign], to say that the value of R hasn't changed. But this is achieved by $R \# \bar{\varepsilon}$.

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

Effect elimination rules are necessary for completeness of the logic. For example, consider this valid triple: $\vdash \{\mathbf{true}\} x := x \{\mathbf{true}\} []$ with empty effect. It can be derived using the assignment axiom, [Exist], [Conseq], and [No Assign].

Using substitution one can derive $\vdash \{\mathbf{true}\} x := F \{x = F\} [\mathbf{wr} \ x]$ from [Assign] provided that $x \notin \text{Vars}(F)$. For field access, using [Subst] with y for z in rule [FieldAcc] we get

$\vdash \{y \neq \mathbf{null}\} x := y.f \{x = y.f\} [\mathbf{wr} \ x]$ provided $x \neq y$. The backwards assignment axiom can also be derived:

$\vdash \{\theta/x \rightarrow E\} x := E \{\theta\} [\mathbf{wr} \ x]$. Let us see how. Given any θ, x, E , choose variable x' distinct from x and let $E' \equiv E/x \rightarrow x'$ so $x \notin \text{Vars}(E')$. By [Assign] we have

$$\vdash \{x = x'\} x := E \{x = E'\} [\mathbf{wr} \ x]$$

Now x does not occur in $\theta/x \rightarrow E'$ so we can use it with Frame to get

$$\vdash \{\theta/x \rightarrow E' \wedge x = x'\} x := E \{\theta/x \rightarrow E' \wedge x = E'\} [\mathbf{wr} \ x]$$

By predicate calculus (equality substitution), $\theta/x \rightarrow E' \wedge x = x'$ is equivalent to $\theta/x \rightarrow E \wedge x = x'$ and also $\theta/x \rightarrow E' \wedge x = E'$ is equivalent to $\theta \wedge x = E'$. So by [Conseq] we get

$$\vdash \{\theta/x \rightarrow E \wedge x = x'\} x := E \{\theta \wedge x = E'\} [\mathbf{wr} \ x]$$

and by weakening the postcondition we get

$$\vdash \{ \theta/x \rightarrow E \wedge x = x' \} x := E \{ \theta \} [\mathbf{wr} x]$$

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

$$\vdash \{ (\exists x' \cdot \theta/x \rightarrow E \wedge x = x') \} x := E \{ \theta \} [\mathbf{wr} x]$$

and by the one-point rule of predicate calculus, using that x' doesn't occur in $\theta/x \rightarrow E$, this precondition is equivalent to $\theta/x \rightarrow E$ so a final use of [Conseq] yields

$$\vdash \{ \theta/x \rightarrow E \} x := E \{ \theta \} [\mathbf{wr} x]$$

Soundness. The program logic is comprised of the rules in Figs. 8, 9, and 10.

THEOREM 5.2 If $\vdash \{ \theta \} C \{ \theta' \} [\bar{\varepsilon}]$ then $\models \{ \theta \} C \{ \theta' \} [\bar{\varepsilon}]$, for any $C, \theta, \theta', \bar{\varepsilon}$.

Proof: By induction on the derivation of $\vdash \{ \theta \} C \{ \theta' \} [\bar{\varepsilon}]$. This boils down to showing soundness for each rule, i.e., if the side conditions hold and the antecedent correctness statements are valid then the consequent is valid. For brevity we focus on the case of normal termination; the arguments for fault-avoidance are straightforward. Recall that $\sigma, \sigma', \sigma_1$ range over proper states (non- \perp).

Case [Frame]: Suppose $\models \{ \theta \} C \{ \theta' \} [\bar{\varepsilon}]$ and $\theta' \vdash \theta_1 \# \bar{\varepsilon}$. To prove $\models \{ \theta \wedge \theta_1 \} C \{ \theta' \wedge \theta_1 \} [\bar{\varepsilon}]$, suppose $\sigma \models \theta \wedge \theta_1$ and $\llbracket C \rrbracket \sigma = \sigma'$. By $\models \{ \theta \} C \{ \theta' \} [\bar{\varepsilon}]$ we have $\sigma' \models \theta'$. Using $\sigma \models \theta_1$ and $\theta' \vdash \theta_1 \# \bar{\varepsilon}$, independence Lemma 4.5 yields $\sigma' \models \theta_1$.

Case [Seq]: Suppose $\models \{ \theta \} C_1 \{ \theta_1 \} [\bar{\varepsilon}_1]$ and $\models \{ \theta_1 \} C_2 \{ \theta' \} [\bar{\varepsilon}_2]$ and $\bar{\varepsilon}_1$ is $\theta', \bar{\varepsilon}_2$ -immune. Let σ be any Γ -state such that $\sigma \models \theta$. Suppose $\llbracket C_1 \rrbracket \sigma = \sigma_1$ and $\llbracket C_2 \rrbracket \sigma_1 = \sigma'$. By validity of the antecedent correctness statements we get $\sigma' \models \theta'$; moreover $\sigma \rightarrow \sigma_1 \models \bar{\varepsilon}_1$ and $\sigma_1 \rightarrow \sigma' \models \bar{\varepsilon}_2$. Using that $\bar{\varepsilon}_1$ is $\theta', \bar{\varepsilon}_2$ -immune, Lemma 4.8 yields $\sigma \rightarrow \sigma' \models \bar{\varepsilon}_1, \bar{\varepsilon}_2$.

Case [While]: Suppose $\models \{ \theta \wedge x \neq 0 \} C \{ \theta \} [\bar{\varepsilon}]$ and $\bar{\varepsilon}$ is $\theta, \bar{\varepsilon}$ -immune. To show $\models \{ \theta \} \mathbf{while} x \mathbf{do} C \{ \theta \wedge x = 0 \} [\bar{\varepsilon}]$, consider any initial state σ . The argument for $x = 0$ in the postcondition is direct from semantics. We claim that for any $n \geq 0$, if the loop iterates n times from σ to some state σ' then $\sigma' \models \theta$ and $\sigma \rightarrow \sigma' \models \bar{\varepsilon}$. The base case is easy; in particular, there are no effects. For the induction step, suppose $n > 1$ so the first $n - 1$ steps lead to some σ_1 . By induction we have $\sigma_1 \models \theta$ and $\sigma \rightarrow \sigma_1 \models \bar{\varepsilon}$. By assumption about C we get $\sigma' \models \theta$ and $\sigma_1 \rightarrow \sigma' \models \bar{\varepsilon}$. So by Lemma 4.8 we get $\sigma \rightarrow \sigma' \models \bar{\varepsilon}, \bar{\varepsilon}$ which is the same as $\sigma \rightarrow \sigma' \models \bar{\varepsilon}$.

Case [Conj] is very easy. Suppose $\models \{ \theta_1 \} C \{ \theta'_1 \} [\bar{\varepsilon}]$ and $\models \{ \theta_2 \} C \{ \theta'_2 \} [\bar{\varepsilon}]$. To prove $\models \{ \theta_1 \wedge \theta_2 \} C \{ \theta'_1 \wedge \theta'_2 \} [\bar{\varepsilon}]$, consider any σ such that $\sigma \models \theta_1 \wedge \theta_2$. If $\sigma' = \llbracket C \rrbracket \sigma$ then by validity of the first antecedent we get $\sigma' \models \theta_1$ and

by the second $\sigma' \models \theta_2$. Finally, by either the first or second we get $\sigma \rightarrow \sigma' \models \bar{\varepsilon}$.

Case [Subst]: It is straightforward to show the key lemma about substitution and update:

$$\sigma \models \theta/x \rightarrow F \text{ iff } \text{update}(\sigma, x, \llbracket F \rrbracket \sigma) \models \theta \quad (1)$$

for all σ, x, F, θ . From $\mathbf{wr} x$ -immunity of $\bar{\varepsilon}$, we know $\bar{\varepsilon}$ has neither $\mathbf{wr} x$ nor $\mathbf{wr}^+ x$, so together with the antecedent that C does not read x we have that

$$\llbracket C \rrbracket (\text{update}(\sigma, x, v)) = \text{update}(\llbracket C \rrbracket \sigma, x, v) \quad (2)$$

for any σ and any value v (of the type T of x , which is unrestricted). Moreover, C diverges (resp. faults), from σ iff it diverges (resp. faults) from $\text{update}(\sigma, x, v)$.

To prove the consequent, $\models \{ \theta/x \rightarrow F \} C \{ \theta'/x \rightarrow F \} [\bar{\varepsilon}]$, consider any σ such that $\sigma \models \theta/x \rightarrow F$, and suppose $\sigma' = \llbracket C \rrbracket \sigma$. From $\sigma \models \theta/x \rightarrow F$ by (1) we get $\text{update}(\sigma, x, \llbracket F \rrbracket \sigma) \models \theta$. Let $\tau = \llbracket C \rrbracket (\text{update}(\sigma, x, \llbracket F \rrbracket \sigma))$, so by validity of the antecedent correctness statement, i.e., $\models \{ \theta \} C \{ \theta' \} [\bar{\varepsilon}]$, we get $\tau = \text{update}(\sigma', x, \llbracket F \rrbracket \sigma)$. By $F \# \bar{\varepsilon}$ and independence Lemma 4.5 we have $\llbracket F \rrbracket \sigma' = \llbracket F \rrbracket \sigma$, hence $\tau = \text{update}(\sigma', x, \llbracket F \rrbracket \sigma')$. So from $\tau \models \theta'$ by (1) we get $\sigma' \models \theta'/x \rightarrow F$. Having shown the postcondition we turn to the effect. From above we have $\text{update}(\sigma, x, \llbracket F \rrbracket \sigma) \rightarrow \text{update}(\sigma', x, \llbracket F \rrbracket \sigma) \models \bar{\varepsilon}$. To prove $\sigma \rightarrow \sigma' \models \bar{\varepsilon}$ as required, observe that $\sigma(x) = \sigma'(x)$ and all the other effects of $\sigma \rightarrow \sigma' \models \bar{\varepsilon}$ are present in $\text{update}(\sigma, x, \llbracket F \rrbracket \sigma) \rightarrow \text{update}(\sigma', x, \llbracket F \rrbracket \sigma)$.

Case [No Assign]: To show $\models \{ \theta \} C \{ \theta' \} [\bar{\varepsilon}]$, suppose $\sigma \models \theta$ and $\llbracket C \rrbracket \sigma = \sigma'$. By hypothesis, $\sigma' \models \theta'$ so it remains to show $\sigma \rightarrow \sigma' \models \bar{\varepsilon}$, given that $\sigma \rightarrow \sigma' \models \mathbf{wr} x, \bar{\varepsilon}$. It is enough to consider part (a) of Def. 4.2 since for field update and allocation the effect set is unchanged. So consider the case that there is some variable z with $\sigma(z) \neq \sigma'(z)$. By hypothesis $y \# \mathbf{wr} x, \bar{\varepsilon}$ and Lemma 4.5 we have $\sigma(y) = \sigma'(y)$. Thus by $\theta \vee \theta' \Rightarrow x = y$ we get $\sigma(x) = \sigma(y) = \sigma'(y) = \sigma'(x)$ so z is not x . It must be some other variable, and thus $\mathbf{wr} z$ or $\mathbf{wr}^+ z$ is in $\bar{\varepsilon}$.

Case [No Alloc]: To show $\models \{ \theta \} C \{ \theta' \} [\bar{\varepsilon}]$ it suffices to show $\sigma \rightarrow \sigma' \models \bar{\varepsilon}$, given that $\sigma \rightarrow \sigma' \models \mathbf{al} R, \bar{\varepsilon}$. Suppose $\sigma \models \theta$ and $\llbracket C \rrbracket \sigma = \sigma'$. Suppose there is $o \in \text{alloc}(\sigma')$ but $o \notin \text{alloc}(\sigma)$. By hypothesis $\models \{ \theta \} C \{ \theta' \} [\mathbf{al} R, \bar{\varepsilon}]$ there is some R' such that either $R' \equiv R$ or $\mathbf{al} R'$ is in $\bar{\varepsilon}$. By $R \# \bar{\varepsilon}$ and independence Lemma 4.5 we have $\llbracket R \rrbracket \sigma = \llbracket R \rrbracket \sigma'$ so $o \notin \llbracket R \rrbracket \sigma'$ (since by semantics $\llbracket R \rrbracket \sigma \subseteq \text{alloc}(\sigma)$). Thus R' is distinct from R hence $\mathbf{al} R'$ is in $\bar{\varepsilon}$.

Case [No Update]: To show $\models \{ \theta \} C \{ \theta' \} [\bar{\varepsilon}]$, suppose $o \in \text{alloc}(\sigma)$ and $\sigma(o.f) \neq \sigma'(o.f)$. By $y \# \bar{\varepsilon}$ and Lemma 4.5 we have $\sigma(y) = \sigma'(y)$. By $x \# \bar{\varepsilon}$ we have $\sigma(x) = \sigma'(x)$. So if $o = \sigma'(x)$ —so that $\mathbf{wr} \langle x \rangle.f$ licenses the field update—then $\sigma'(o.f) = \sigma'(x.f) = \sigma'(y) = \sigma(y) = \sigma(x.f) = \sigma(o.f)$, contradicting the assumption that $o = \sigma'(x)$. Thus there must be some other region R with

wr $R.f$ or wr⁺ $R.f$ in $\bar{\varepsilon}$. □

6. Framing representation invariants

Clients of a module should be oblivious to its internal representation. The module’s procedures maintain a representation invariant that may depend on the state of heap data structures and module-scoped global variables. Such procedures can assume the invariant because, for reasons of encapsulation, the client cannot falsify the invariant between calls to module procedures. Of course this ideal story is often compromised, since heap encapsulation is difficult to achieve. In this section we consider the second-order or “hypothetical” frame rule [23] that embodies such reasoning in a simple form, without distracting clutter formalizing the module as such.

6.1 Hypothetical judgements

We extend the grammar of commands by $C ::= k$ where $k \in ProcName$. We escalate to hypothetical judgements of the form

$$\Delta \vdash^\Gamma \{ \theta \} C \{ \theta' \} [\bar{\varepsilon}_C; \bar{\varepsilon}_M]$$

where Δ is a set of *assumed* specifications for command names and $\{ \theta \} C \{ \theta' \} [\bar{\varepsilon}_C; \bar{\varepsilon}_M]$ is the *conclusion*. (We use *antecedent* and *consequent* for proof rules.) An assumption for command k takes the form $\{ \delta \} k \{ \delta' \} [\bar{\varepsilon}]$. The judgement displayed above is well formed provided $\delta, \delta', \bar{\varepsilon}$ is well formed in Γ (for each assumption) and the conclusion is also well formed in Γ .

The effect is now partitioned, as indicated by the semicolon in $\bar{\varepsilon}_C; \bar{\varepsilon}_M$, where $\bar{\varepsilon}_C$ are the effects of explicit updates and allocations in C and $\bar{\varepsilon}_M$ are the effects from procedure calls. The identifier $\bar{\varepsilon}_M$ hearkens to “module” but in general $\bar{\varepsilon}_C$ and $\bar{\varepsilon}_M$ may write the same variables and objects.

With the exception of [Subst], all of the previous rules for correctness judgements are retained, but with Δ added to the left of the turnstile (the same Δ throughout the rule). The partitioning of effects is propagated; for example [Frame] becomes

$$\frac{\Delta \vdash \{ \theta \} C \{ \theta' \} [\bar{\varepsilon}_1; \bar{\varepsilon}_2] \quad \theta' \vdash \theta_1 \# \bar{\varepsilon}_1, \bar{\varepsilon}_2}{\Delta \vdash \{ \theta \wedge \theta_1 \} C \{ \theta' \wedge \theta_1 \} [\bar{\varepsilon}_1; \bar{\varepsilon}_2]}$$

Note that the independence is still for a single effect set, $\bar{\varepsilon}_1, \bar{\varepsilon}_2$. The axioms introduce effects on the client side, e.g. [FieldUpd] becomes

$$\Delta \vdash \{ x \neq \mathbf{null} \} x.f := F \{ x.f = F \} [\mathbf{wr} \langle x \rangle . f ;]$$

The issue with [Subst] is its condition “ C does not read x ” which, in case C is k , requires read effects in the specification of k , or some other static analysis. This is neither an insurmountable problem nor relevant to our main concerns; so we just revise [Subst] to say C neither mentions x nor

contains any procedure call. A new rule allows use of an assumption, recording the effect in the right place:

$$\text{PROC INTRO} \quad \Delta, \{ \theta \} k \{ \theta' \} [\bar{\varepsilon}] \vdash \{ \theta \} k \{ \theta' \} [; \bar{\varepsilon}]$$

The only other new rule, [Hyp Frame], is introduced in Sec. 6.2.

The semantics is revised straightforwardly. An *environment* for Δ is a function η that maps each k_i to a state transformer t_i for Γ -states. The semantics of commands is augmented with an environment, so it is now written $\llbracket \Gamma \vdash C \rrbracket^\eta$. Other than passing η around, nothing changes, but there is an additional clause for procedures: $\llbracket \Gamma \vdash k \rrbracket^\eta \sigma = \eta(k)(\sigma)$

Define $\eta \models \Delta$ iff for each $\{ \delta_i \} k_i \{ \delta'_i \} [\bar{\varepsilon}_i] \in \Delta$ we have

$$\eta(k_i) \models^\Gamma \{ \delta_i \} - \{ \delta'_i \} [\bar{\varepsilon}_i]$$

DEFINITION 6.1 (hypothetical validity). A hypothetical judgement $\Delta \vdash \{ \theta \} C \{ \theta' \} [\bar{\varepsilon}_1; \bar{\varepsilon}_2]$ is *valid* iff for all η , if $\eta \models \Delta$ then $\llbracket \Gamma \vdash C \rrbracket^\eta \models^\Gamma \{ \theta \} - \{ \theta' \} [\bar{\varepsilon}_1, \bar{\varepsilon}_2]$.

Note that the effect partitioner, “;”, has turned to a comma: semantically there is a single set of effects.

Soundness of rule [Proc Intro] is immediate from the definitions, and soundness of the hypothetical form of all the previous rules is by straightforward adaptation of the proof of Thm. 5.2, taking care with [Subst] as discussed above.

6.2 Hypothetical frame rule

Linking procedures to code, and proof obligations to procedure implementations, is standard. We omit this from our formalization, so in the proof system of this section the assumption set Δ is the same throughout a given proof.

Nonetheless, it is helpful to consider how procedures would be treated. Suppose the language is extended with bindings, written **let** k **be** C **in** B . A straightforward rule for this is as follows.

$$\text{PROC ELIM} \quad \frac{\Delta \vdash \{ \delta \} C \{ \delta' \} [\bar{\varepsilon}_k] \quad \Delta, \{ \delta \} k \{ \delta' \} [\bar{\varepsilon}_k] \vdash \{ \varphi \} B \{ \varphi' \} [\bar{\varepsilon}_B; \bar{\varepsilon}_M]}{\Delta \vdash \{ \varphi \} \mathbf{let} \ k \ \mathbf{be} \ C \ \mathbf{in} \ B \{ \varphi' \} [\bar{\varepsilon}_B; \bar{\varepsilon}_M]}$$

Note that $\bar{\varepsilon}_M$ reflects the effects from calls to k and other procedures in Δ . Even if only k is called, $\bar{\varepsilon}_M$ can differ a lot from $\bar{\varepsilon}_k$; there may be many calls, uses of ghost variables to re-express the effects, etc.

Generalization to multiple procedures and mutually recursive ones can be carried out along standard lines.⁹ But to verify the implementation of k we would like to exploit a representation invariant, say ψ , that does not appear in the interface specification. That is, we would like to change the first antecedent in [Proc Elim] to be $\Delta \vdash \{ \delta \wedge \psi \} C \{ \delta' \wedge \psi \} [\bar{\varepsilon}]$ while leaving the second unchanged. Soundness for this is not at all obvious owing to the mismatch between what is proved about C and assumed about k .

⁹See for example Yang et al [23], noting that some care is needed to avoid mistakes in quantifier scoping for recursive procedures [2, 22].

Yang et al [23] disentangle the essence of this mismatch from the separate issue of binding C to k . Their “hypothetical” or second order frame rule distills the essence of the mismatch. Here is our version.¹⁰ Look at the correctness judgements first; the side conditions will be explained in due course.

HYP FRAME

$$\frac{Q; \bar{x} \text{ frames } \psi \quad \bar{x} \# \bar{\varepsilon}_B \quad \text{regions}(\bar{\varepsilon}_B) = R_1, \dots, R_n \quad \gamma \Rightarrow Q \# R_1 \cup \dots \cup R_n \quad \{\delta\} k \{\delta'\} [\bar{\varepsilon}_k] \vdash \{\varphi\} B \{\varphi'\} [\bar{\varepsilon}_B; \bar{\varepsilon}_M]}{\{\delta \wedge \psi\} k \{\delta' \wedge \psi\} [\bar{\varepsilon}_k] \vdash \{\varphi \wedge \psi\} B \{\varphi' \wedge \psi\} [\bar{\varepsilon}_B; \bar{\varepsilon}_M]}$$

To streamline notation we restrict to the case of a single procedure. The rule says that, unbeknownst to the client reasoner, ψ is maintained.

One might expect the side condition $true \vdash \psi \# \bar{\varepsilon}_B$. This is questionable on grounds of modularity. It is also flat wrong. We need to arrange that ψ holds prior to each and every call to k ; they are at intermediate points within B , which have their own effects (and in which states $\bar{\varepsilon}_B$ could varying interpretations). We need B not to interfere with ψ at the points where k is invoked, not in the final state where $\bar{\varepsilon}_B$ is interpreted.

Many works on effects and regions treat regions as symbols that denote, in all states, disjoint portions of the heap and which implicitly grow via allocation. The cost of our more flexible treatment is that global separation assumptions are made explicit in the form of a commitment predicate, γ . It plays a role comparable to the global ownership invariant of ownership type systems. Later we add conditions to enforce this role.

6.3 Example

Consider two procedures $init$ and $copy$ on linked lists; $init$ creates a new list and $copy$ copies the newly created list into another list. The global variables are $xs, res : List$ and $r : \text{rgn}$. Class $List$ has fields $hd : Node$ and $rep : \text{rgn}$. Both procedures use the invariant

$$\psi : \langle xs \rangle .hd \subseteq xs.rep \wedge xs.rep \cdot next \subseteq xs.rep$$

Note that the regions mentioned in ψ are $\langle xs \rangle$ and $xs.rep$. Hence $Q; xs \text{ frames } \psi$ where $Q \equiv \langle xs \rangle \cup xs.rep$.

The spec of $init$ is $\{true\} init \{\delta'_{init}\} [\bar{\varepsilon}_{init}]$, where

$$\delta'_{init} : xs \neq null \wedge \langle xs \rangle \cup xs.rep \# r$$

$$\bar{\varepsilon}_{init} : \mathbf{al} \langle xs \rangle, \mathbf{al} xs.rep, \mathbf{wr} xs, \mathbf{wr} \langle xs \rangle \bullet \mathbf{any}, \mathbf{wr} xs.rep \bullet \mathbf{any}$$

The spec of $copy$ is $\{xs \neq null\} copy \{\delta'_{copy}\} [\bar{\varepsilon}_{copy}]$, where

$$\delta'_{copy} : res \neq null \wedge \langle res \rangle \cup res.rep \# r$$

¹⁰ We define $\text{regions}(\bar{\varepsilon})$ to extract the set of region expressions that occur at the top level in effects, i.e., it maps over $\bar{\varepsilon}$ the function sending $\mathbf{wr} R \bullet f$ to R , $\mathbf{al} R$ to R , $\mathbf{wr} x$ to \emptyset , etc. The rule uses $\bar{x} \# \bar{\varepsilon}_B$ to abbreviate independences for each variable in \bar{x} ; in practice some of these variables would have module scope.

$$\bar{\varepsilon}_{copy} : \mathbf{al} \langle res \rangle, \mathbf{al} res.rep, \mathbf{wr} res, \mathbf{wr} \langle res \rangle \bullet \mathbf{any}, \mathbf{wr} res.rep \bullet \mathbf{any}$$

The client program is

$$B : \quad \text{init}; x := \mathbf{new} List; x.hd := \mathbf{null}; copy$$

about which the reasoner will prove $\{true\} B \{\delta'_{copy}\} [\bar{\varepsilon}_B; \bar{\varepsilon}_M]$ where $\bar{\varepsilon}_B; \bar{\varepsilon}_M$ will become clear below. That is, we will take $\varphi \equiv true$ and $\varphi' \equiv \delta'_{copy}$ in [Hyp Frame].

Next, we consider the two assignments in turn. By [Alloc] followed by [Subst], with $\langle xs \rangle \cup xs.rep$ substituted for r we get:

$$\{true\} x := \mathbf{new} List \{\theta_1\} [\bar{\varepsilon}_1]; \quad (3)$$

where

$$\theta_1 : x \neq null \wedge \langle x \rangle \# (\langle xs \rangle \cup xs.rep)$$

$$\bar{\varepsilon}_1 : \mathbf{wr} x, \mathbf{al} \langle x \rangle$$

By [FieldUpd] and [Frame] we get

$$\vdash \{\theta_1\} x.hd := null \{\theta_2\} [\bar{\varepsilon}_2] \quad (4)$$

$$\theta_2 : x.hd = null \wedge \langle x \rangle \# (\langle xs \rangle \cup xs.rep)$$

$$\bar{\varepsilon}_2 : \mathbf{wr} \langle x \rangle .hd$$

Now, letting

$$\gamma : (xs.rep \cup \langle xs \rangle) \# \langle x \rangle$$

we proceed with

$$\begin{aligned} & \{true\} init \{xs \neq null\} [\bar{\varepsilon}_{init}] \quad \text{by [Conseq]} \\ & \{xs \neq null\} x := \mathbf{new} List \{\theta_1 \wedge xs \neq null\} [\bar{\varepsilon}_1] \\ & \quad \text{by (3, [Frame])} \\ & \{\theta_1 \wedge xs \neq null\} x.hd := null \{xs \neq null \wedge \gamma\} [\bar{\varepsilon}_2] \\ & \quad \text{by (4, [Conseq])} \\ & \{xs \neq null \wedge \gamma\} copy \{\delta'_{copy}\} [\bar{\varepsilon}_{copy}] \\ & \quad \text{by ([Frame], [Conseq])} \end{aligned}$$

At this stage, note that the overall effects of the client code are $\bar{\varepsilon}_1, \bar{\varepsilon}_2; \bar{\varepsilon}_{init}, \bar{\varepsilon}_{copy}$. We have $\text{regions}(\bar{\varepsilon}_1, \bar{\varepsilon}_2) = \{\langle x \rangle\}$. To show the antecedent of [Hyp Frame], note that $xs \# \mathbf{wr} x$ because xs does not occur in $\mathbf{wr} x$.

Note that γ is a postcondition of the sequence $x := \mathbf{new} List; x.hd := null$, at which point we want it to be sound to assert ψ . This will provide a “robust decomposition” of the proof, to be defined later. In fact γ holds after the first assignment as well, affording a different robust decomposition.

6.4 Soundness

Rule [Hyp Frame] is unsound without additional restrictions. The rule of Yang et al is also unsound without restrictions on the predicates involved: roughly, the invariant should be “precise” [23] in the sense that its footprint is uniquely determined (as are all footprints in our setting). Expressing this condition is not easy. Our additional restriction is rather different, in fact we restrict the structure of the proof of the

antecedent! (In Sec. 7 we point out why this is natural in more practical reasoning scenarios.)

We are going to justify [Hyp Frame] by showing its *admissibility*, i.e., its use can be eliminated, by transforming the proof of the antecedent. We would like to prove admissibility simply conjoining ψ to every assertion throughout the proof of the antecedent judgement of [Hyp Frame], so that proof becomes a proof of the consequent once we insert suitable instances of the ordinary [Frame] rule following axioms. The hope is that the independence of ψ from the relevant effects will follow from the side conditions on the rule. But how? The small axioms do not allow conjoining arbitrary predicates; effect elimination rules and effect subsumption rules allow intermediate effects to differ from the top level $\bar{\varepsilon}_B$; and the latter’s interpretation at intermediate states could differ from its interpretation in the final state. The key observation is that such delicate effects are almost always kept local in the proof, with local effects getting subsumed by ones of immunity.

In order to specify encapsulation —separation assumptions— using state-dependent effects, we need a notion of “commitment” on the part of the client. Commitments of this form are used, for example, in the Boogie discipline, and its assumption-commitment aspect is articulated in work on the friendship discipline [21].

To pursue this, without belaboring the details, we work with proofs as trees in the usual way: each node is an instance of a hypothetical judgement that is either

- a legal instance of an axiom
- a legal instance of a rule, in which there is a subtree for each of its antecedent judgements (including independence, subtyping, and “frames” judgements)

and any side conditions (disjointness of variables; validity of some formula) hold. We say *legal proof tree* for emphasis. Because the format for rules has consequent at the bottom, we say a node is “lower” if it is closer to the root; also “later”.

DEFINITION 6.2 (robust decomposition). Let t be a legal proof tree for an instance of the antecedent correctness judgement in [Hyp Frame]. A subtree u of t is called *k-free* provided that the command at the root of u does not invoke the procedure k . A *robust decomposition of t for γ* is a set D of disjoint subtrees of t such that

1. each subtree in D is *k-free*,
2. every leaf in t is a leaf of some u in D , except for leaves that are instances of [Proc Intro] for k
3. every occurrence of an effect-elimination rule in t appears in one of the subtrees in D
4. every occurrence of [Subst] is either for some variable not in \bar{x} or it is in one of the subtrees in D
5. for each subtree u in D , let the root of u be $\Delta \vdash \{\theta\} C \{\theta'\} [\bar{\varepsilon};]$ (note that the module effect will be

empty). Then $\theta' \Rightarrow \gamma$ is valid and $\gamma \vdash \varepsilon \leq \bar{\varepsilon}_B$ for every ε in $\bar{\varepsilon}$ that is not a write to a local variable declared in B

The critical condition is the last one: it connects the local effect with the final effect of the main program B at the root of t . Typically γ is globally invariant.

Note that the designated subtrees might be proofs for a single assignment command or for a composite command, though without calls to the procedures k of interest. In our simple example above, there are no uses of effect elimination. There is a robust decomposition with one subtree for the two assignments, and another with separate subtrees for the two assignments.

THEOREM 6.3 (admissibility of Hyp Frame). Consider a proof of

$$\{\delta \wedge \psi\} k \{\delta' \wedge \psi\} [\bar{\varepsilon}_k] \vdash \{\varphi \wedge \psi\} B \{\varphi' \wedge \psi\} [\bar{\varepsilon}; \bar{\varepsilon}_M] \quad (5)$$

that concludes with use of [Hyp Frame], for some γ, Q, \bar{x} (and otherwise does not use [Hyp Frame]). Let t be the proof tree of the antecedent judgement. If there is a robust decomposition of t for γ , then there is a proof of (5) that does not use [Hyp Frame].

Corollary: Any hypothetical judgement that is provable in the system that includes [Hyp Frame] is provable in the system without it. **Proof of corollary:** by induction on nesting of the uses of [Hyp Frame], using the Theorem.

Proof: Let D be the designated subtrees for robust decomposition of t and let $\Delta_k \triangleq \{\delta \wedge \psi\} k \{\delta' \wedge \psi\} [\bar{\varepsilon}_k]$. Let $t1$ be the structure obtained from t by conjoining ψ with every pre- and post-condition in every judgement in t excluding those in the subtrees in D ; even the roots of those subtrees are left unchanged. Note that the conclusion of $t1$ is (5). Moreover, each leaf that introduces a procedure k by [Proc Intro] now matches the assumption in (5), with ψ . The judgements remain well formed, because ψ is well formed in the same global context as command B . But $t1$ is far from being a legal proof.

The main idea is to insert instances of [Frame] after the roots of the designated subtrees. Let $t2$ be the structure obtained from $t1$ by inserting those instances of [Frame]. To show that they are legal, consider an instance

$$\frac{\Delta_k \vdash \{\theta\} C \{\theta'\} [\bar{\varepsilon};] \quad \theta' \vdash \psi \# \bar{\varepsilon}}{\Delta_k \vdash \{\theta \wedge \psi\} C \{\theta' \wedge \psi\} [\bar{\varepsilon};]}$$

The module part of the effect is empty since by robustness there are no calls to k . Using hypothesis $\bar{x} \# \bar{\varepsilon}_B$ and $\gamma \Rightarrow Q \# R_1 \cup \dots \cup R_n$ from [Hyp Frame], we get $\gamma \vdash \psi \# \bar{\varepsilon}_B$ using independence rules (as remarked just before Lemma 4.5). By robustness we have $\gamma \vdash \bar{\varepsilon} \leq \bar{\varepsilon}_B$ and $\theta' \Rightarrow \gamma$, whence $\theta' \vdash \bar{\varepsilon} \leq \bar{\varepsilon}_B$ by the consequence rule for sub-effects. So by the sub-effect rule for independence (last rule in Fig. 6) we get $\gamma \vdash \psi \# \bar{\varepsilon}$ as needed for this instance of [Frame].

Having shown that these new instances of [Frame] in t_2 are legal, it remains to show that the changed rule instances are legal. None of the effect elimination rules [No Assign], [No Alloc], [No Update] need be considered, because in a robust proof they are inside subtrees that are left unchanged. The case of [If] is easy. For [Seq], suppose this is the instance in the original proof t , lumping the effects together for readability:

$$\frac{\Delta_k \vdash \{\theta\} C_1 \{\theta_1\} [\bar{\varepsilon}_1] \quad \Delta_k \vdash \{\theta_1\} C_2 \{\theta'\} [\bar{\varepsilon}_2] \quad \bar{\varepsilon}_1 \text{ is } \theta', \bar{\varepsilon}_2\text{-immune}}{\Delta_k \vdash \{\theta\} C_1 ; C_2 \{\theta'\} [\bar{\varepsilon}_1, \bar{\varepsilon}_2]}$$

Since the intermediate assertion is strengthened to $\theta_1 \wedge \psi$ in both subproofs, the assertions match perfectly. But the side condition is now that $\bar{\varepsilon}_1$ is $(\theta' \wedge \psi), \bar{\varepsilon}_2$ -immune. In Def. 4.6, θ' appears only on the left of an implication so it is straightforward to show that strengthening to $\theta' \wedge \psi$ also satisfies the definition.

The argument for [While] is similar to that for [Seq]. For most of the structural rules, the argument is straightforward, e.g., the assertions are unchanged in [Sub Eff] and the sub-effect judgement $\theta' \vdash \bar{\varepsilon} \leq \bar{\varepsilon}'$ yields $\theta' \wedge \psi \vdash \bar{\varepsilon} \leq \bar{\varepsilon}'$ by rule [SubEff Conseq]. The cases of [Var], [Conj], [Conseq], and [Context] are left to the reader. For [Exist], note that by hygiene the bound variable is not in the frame of ψ . For [Subst], note that if it is outside a designated subtree then the substituted variable is not in \bar{x} (by robustness) so the substitution distributes over $\wedge \psi$ without changing ψ . The remaining case is [Frame] for some predicate θ_1 . Consider an instance

$$\frac{\vdash \{\theta\} C \{\theta'\} [\bar{\varepsilon}] \quad \theta' \vdash \theta_1 \# \bar{\varepsilon}}{\vdash \{\theta \wedge \theta_1\} C \{\theta' \wedge \theta_1\} [\bar{\varepsilon}]}$$

in the original proof t . With ψ conjoined everywhere, the side condition becomes $\theta' \wedge \psi \vdash \theta_1 \# \bar{\varepsilon}$ which can be obtained from the original side condition by using the consequence rule that appears last in Fig. 6. \square

7. Discussion

The genesis of this work is our ongoing exploration of a hybrid approach to secure information flow analysis that combines verification and type checking [7]. We have been using a relational logic that lets us specify fine grained declassification policies (like gradual release [5]) and we have an end-to-end knowledge based semantics. In particular, Amtoft et al [3] achieve modular reasoning about information flow using regions; extension of their work to declassification policies, however, is not easy, particularly because such policies may depend on complex program state conditions [7]. Thus we needed to extend Amtoft et al. to a full assertion language. This necessitated dropping their abstract interpretation of heap locations in favor of explicit regions. This allows the use of existing tools to verify security policies [20].

Indeed, we can also address the difficult problem of declassification of a data structure for which the state of the art is to clone the data structure to prevent attacks via escaped pointers [4]. Instead, all that is necessary is the declassification of the root pointer of the data structure, given its provable isolation. This is akin to transferring the root pointer of the data structure thereby exposing encapsulated objects. We were thus led to considering heap-based encapsulation and the verification of encapsulated data structures.

State of the art verifiers use intricate reasoning about the heap, often based on variants of ownership régimes. Such régimes, although amenable to automation, can have ad hoc soundness proofs. Kassios [14] points out that such reasoning can, in general, be performed through disciplined use of ghost state, thereby opening the door to integrated use of multiple disciplines. 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. One shortcoming of his work is that the notion of “frames” is a second order predicate, quantifying over all global program states. Our logic achieves the benefits of the idea but in an entirely first order way. Another shortcoming of Kassios’s work is that it seems to require the general reasoning behind soundness of a discipline like Boogie [16] be re-produced as part of the verification for any particular program that uses the discipline. Our hypothetical frame rule, [HypFrame], shows that general patterns of data abstraction can be formalized as such and proved sound once and for all, by reduction to a foundational logic, rather than building them in as primitive features of the logic or resorting to semantic arguments [16, 21].

Our proof rules involve not only conventional correctness statements but also novel judgements for independence, immunity, and sub-effects. But it is evident from the rules for these judgements that they are decidable modulo validity of assertions: In addition to the usual implications in the rule of consequence, there are disjointness conditions like $\theta' \Rightarrow \langle x \rangle \# R$ in the independence rules (Fig. 6) and region containments in the rules for sub-effects (Fig. 5). So the logic gives rise to verification conditions that are decidable modulo validity of assertions in our rather tame assertion language. Nevertheless, a thorough study of verification conditions is underway.

An interesting feature of the particular version of [HypFrame] we presented is that it requires the proof tree of the client to be “robustly decomposable”. The intuition is that the designated subtrees are the part of the client code where the invariant is suspended; the invariant holds for all other parts of the proof tree. This is similar in spirit to Boogie’s “expose” block, and by using syntax in that manner it could be made into a proper proof rule.

Ultimately one hopes for some programming constructs that adapt notions like ML modules to design patterns that involve clusters of interacting objects that maintain

invariants dependent on multiple shared objects (iterators, database connections, etc).

Existing approaches. The influence of separation logic on our work is clear. The separating conjunction hides the heap and expresses separation in the heap implicitly. The semantics of the separating conjunction has an implicit existential quantification that poses technical challenges and conundra. Because footprints are mere shadows of predicates, specifications require full functional descriptions using inductive definitions, or else quantification over predicates; neither is ideal for automated verification.

In higher order separation logics [9] and Hoare type theory [18, 19], one can quantify predicates to get multi-instance abstractions at the cost of intricate semantics, and a theory that cannot be adapted directly to automated verifiers. The work of Mijajlovic and Yang [17] on data refinement has some complications regarding nondeterminacy of allocation; and the fully nondeterministic allocator is needed for the soundness of the frame rule.

Amtoft et al. use regions in assertions; these regions are tied to an underlying abstract interpretation. However, in contrast to our logic, rule [Conj] is unsound in the system of Amtoft et al. Kassios reasons directly about regions and shows how the reasoning behind the ad hoc disciplines used in verifiers can be expressed using ordinary logic and ghost state.

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.

The proof rules of our logic are formulated in a way that shows how reasoning works. However, a verifier will likely not apply such rules directly but will rather transform the code and generate verification conditions. We therefore plan to implement and experiment with the logic by first translating into BoogiePL. For detailed verification of programs, we plan to investigate inductive definitions.

Another avenue to explore is to consider our logic (or some embedding thereof) 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)

Our semantics, and the definitions based on it, assume there is no garbage collection (GC). We plan to investigate GC using a semantics in which objects are never removed from the heap but have a ghost field indicating they have been collected, provided they are not reachable via ordinary variables and fields, since they could well remain reachable via ghost fields. This could facilitate dealing with the issues about quantification [11].

Finally, we plan to machine check the core logic and abstraction rules like [Hyp Frame] building on an existing model of JML in PVS [15].

References

- [1] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *ECOOP*, pages 1–25, 2004.
- [2] P. America and F. de Boer. Proving total correctness of recursive procedures. *Information and Computation*, 84(2):129–164, 1990.
- [3] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *POPL*, 2006. Extended version available as KSU CIS-TR-2005-1.
- [4] A. Askarov and A. Sabelfeld. Secure implementation of cryptographic protocols: A case study of mutual distrust. In *ESORICS*, 2005.
- [5] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *IEEE Symposium on Security and Privacy*, 2007.
- [6] A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM*, 52(6):894–960, Nov. 2005.
- [7] A. Banerjee, D. A. Naumann, and S. Rosenberg. Towards a logical account of declassification. In *PLAS*, 2007.
- [8] G. Bierman and M. Parkinson. Separation logic and abstraction. In *POPL*, pages 247–258, 2005.
- [9] L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. *Logical Methods in Computer Science*, 2(5:1):1–33, 2006.
- [10] R. Bornat. Proving pointer programs in Hoare logic. In *MPC*, 2000.
- [11] C. Calcagno, P. O’Hearn, and R. Bornat. Program logic and equivalence in the presence of garbage collection. *Theoretical Comput. Sci.*, 298(3):557–581, 2003.
- [12] P. Chalin. A sound assertion semantics for the dependable systems evolution verifying compiler. In *International Conference on Software Engineering*, pages 23–33, 2007.
- [13] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, pages 292–310, Nov. 2002.
- [14] I. T. Kassios. Dynamic framing: Support for framing, dependencies and sharing without restriction. In *Formal Methods*, number 4085 in LNCS, pages 268–283, 2006.
- [15] G. T. Leavens, D. A. Naumann, and S. Rosenberg. Preliminary definition of core JML. Technical Report CS Report 2006-07, 2006.
- [16] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 491–516, 2004.
- [17] I. Mijajlović and H. Yang. Data refinement with low-level pointer operations. In *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 19–36, 2005.
- [18] A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable ads in Hoare type theory. In *ESOP*, 2007.

- [19] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare type theory. In *ICFP*, 2006.
- [20] D. A. Naumann. From coupling relations to mated invariants for secure information flow. In *European Symposium on Research in Computer Security (ESORICS)*, number 4189 in LNCS, pages 279–296, 2006.
- [21] D. A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. *Theoretical Comput. Sci.*, 365:143–168, 2006.
- [22] T. Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In J. Bradfield, editor, *Computer Science Logic (CSL 2002)*, volume 2471 of LNCS, pages 103–119, 2002.
- [23] P. O’Hearn, H. Yang, and J. Reynolds. Separation and information hiding. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 268–280, 2004.
- [24] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Inf.*, 6, 1976.
- [25] C. Pierik and F. S. de Boer. A proof outline logic for object-oriented programming. *Theoretical Comput. Sci.*, 343:413–442, 2005.
- [26] J. C. Reynolds. *The Craft of Programming*. Prentice-Hall, 1981.