A Logical Analysis of Framing for Specifications with Pure Method Calls

Anindya Banerjee [⋆1] and David A. Naumann [⋆⋆2]

[1] IMDEA Software Institute, Madrid, Spain
[2] Stevens Institute of Technology, Hoboken, USA

Abstract. For specifying and reasoning about object-based programs it is often attractive for contracts to be expressed using calls to pure methods. It is useful for pure methods to have contracts, including read effects to support local reasoning based on frame conditions. This leads to puzzles such as the use of a pure method in its own contract. These ideas have been explored in connection with verification tools based on axiomatic semantics, guided by the need to avoid logical inconsistency, and focusing on encodings that cater for first order automated provers. This paper adds pure methods and read effects to region logic, a first-order program logic that features frame-based local reasoning and a proof rule for linking of clients with modules to achieve end-to-end correctness by modular reasoning. Soundness is proved with respect to a conventional operational semantics and using the extensional (i.e., relational) interpretation of read effects.

1 Introduction

In reasoning about programs, a frame condition is the part of a method's contract that says what part of the state may be changed by an invocation of the method. Frame conditions make it possible to retain a global picture while reasoning locally: If predicate Q can be asserted at some point in a program where method m is called, Q still holds after the call provided that the locations on which Q depends are disjoint from the locations that may be written according to m's frame condition. This obvious and familiar idea is remarkably hard to formalize in a way that is useful for sound reasoning about programs acting on dynamically allocated mutable objects (even sequential programs, to which we confine attention here). One challenge is to precisely describe the writable state in case it involves heap allocated objects. Another challenge is to determine what part of such state may be read by Q (its 'footprint'). For reasons of abstraction, Q may be expressed in terms of named functions. To hide information about data representation, the function definitions may not be visible in the client program where m is called. This paper provides a foundational theory addressing these and related challenges.

Consider a class Cell with instances holding an integer value, used in the following client code.

```
method get(): int
method set(v: int) ensures self.get() = v
var c, d: Cell; c:= new Cell; d:= new Cell; c.set(5); d.set(6); assert c.get() = 5;
```

⋆ Currently on leave at the US National Science Foundation.
⋆⋆ Partially supported by US NSF award CNS-1228930.

```
class Cell { private val: int; ghost footp: rgn;
  pure method get(): int ensures self.get() = result
                         reads self.footp`any
  method set(v: int) ensures self.get() = v
                     writes self.footp`any //(we elide read effects for impure methods)
                     ensures ∀o: Cell . o ≠self ⇒o.footp #self.footp }
```

Fig. 1. Example: Cell. Type **rgn** (for 'region') means sets of object references and footp`**any** denotes a set of locations, namely all fields of objects in footp.

The goal is to prove the assertion by reasoning that the state read by c.get() is disjoint from the state written by d.set(6). Suppose the internal representation of Cell objects consists of an integer field val. The specifications could say set writes **self**.val and get reads **self**.val. Then the frame condition of d.set(6) would allow the postcondition of the call c.set(5), i.e., the predicate c.get()= 5, to be framed over the call d.set(6), yielding the assertion. But such specifications expose the internal representation. It would preclude, for example, an alternative implementation that uses, instead of integer field val, a pointer to a character string that represents the number using 0s and 1s.

Better specifications appear in Fig. 1, using ghost state to describe the 'footprint' of each cell, and postconditions from which the client can deduce disjointness of the representations of c and d. Use of ghost state for footprints is a key part of the 'dynamic frames' approach [8], and in addition to explicit disjointness conditions it supports separation reasoning based on freshness.

The example illustrates another challenging issue: one method (get) is used in the specification of others (get,set). Here is an example of calling a pure method in a frame condition: Instead of the ghost field footp one might choose to define a region-valued method footpm. If footpm is only used in specifications, one may argue it should be defined as part of the mathematical theory in which reasoning is carried out (though its read effect would still be useful). But there are practical benefits to using programmed methods in specifications, which can be justified provided that they are ***pure*** in the sense of having no effects other than reading.

Use of pure methods, especially ones in the program rather than part of the ambient mathematical theory, poses challenges. One is how to model such specifications without inconsistency. For example, care must be taken for sound treatment of specifications like **method** f(x: **int**): **int ensures result** = f(x)+1. Recursion aside, one may also wonder about soundness of using a pure method in its own postcondition, e.g., get in Fig. 1, or in its own frame condition: e.g., the read effect of footpm might be footpm()`**any** (making it 'self-framing'). Another issue is that the specifications of get and set are abstract, in the sense that they are consistent with many interpretations of the function get (e.g., get could return **self**.val+7). Client code should respect the abstraction, i.e., be correct with respect to any interpretation, whereas the expected implementations of get and set are only correct with respect to the interpretation that returns **self**.val.

The issues discussed so far have been addressed in prior work, especially in the context of verification-condition generation (VC-gen); see Sec. 7. However, most of the VC-gen work takes axiomatic semantics for granted rather than defining and proving

soundness with respect to operationally grounded program semantics; the focus is on methodological considerations and on encodings that work effectively with SMT-based theorem provers. In these works, hypotheses are encoded as axioms, and linking of separately verified methods is implicit in the implementation of the VC-gen. The intricacies of dealing with heap structure, framing, purity, and self-framing frame conditions have led to soundness bugs in implemented verification systems (see [6]).

This paper provides a foundational account, by way of a conventional logic of programs that caters for SMT-provers by reasoning about framing using ghost state and FOL, and that is proved sound with respect to a standard operational semantics. Our account focuses on a *proof rule for linking* the implementation of an interface (i.e., collection of method specifications) with a client that relies on that interface.

The approach we take is motivated by two additional challenges. The first is information hiding, in the sense that implementations rely on invariants on module-internal data structures, but these invariants do not appear in the interface specification [7]. As a contrived example, representing the integer cell using a string might have the invariant that only 0 and 1 characters appear, without leading zeros. The invariant might be exploited by a method getAsString, but it has no place in the interface specification of method get which returns an integer. An alternative to hiding is to rely on abstraction: a predicate whose definition is opaque in the interface can be defined internally to be the invariant [15,11,9].

The second additional challenge arises from the practical need to use programmed methods that are only ***observationally pure*** in the sense that they do have side effects but these effects are benevolent [7] and not observable to clients. There are many examples, including memoization, lazy initialization, and path compression in Union-Find structures. These may involve allocation of fresh objects and mutation of existing ones.

Strong encapsulation is critical both for hiding of invariants [7,14,1] and for observational purity [13]. Both involve linking a client with the implementation of a module, where that implementation is verified against specifications different from those used by clients of the module —hiding invariants and hiding effects. In prior work we developed ***region logic*** (RL), a Hoare logic for sequential object-based programs, using standard FOL for assertions. By contrast with separation logic and permission-based systems, in RL separation is expressed as disjointness of explicit footprints, following the approach of dynamic frames. A benefit is that the verifier does not need to support separating conjunction; it comes at the cost of more verbose specifications. The language features expressions that denote ***regions***, i.e., sets of object references. The logic provides a frame rule for local reasoning, based on frame conditions of methods and a subsidiary judgment for framing of formulas (Fig. 10). In addition to ordinary frame conditions, the logic formalizes encapsulation boundaries for modules, again in the manner of dynamic frames. This supports a second-order frame rule for linking method implementations to clients, hiding invariants [7,14].

In ongoing work, we have extended RL to a relational version, akin to [3,20,12] but featuring a proof rule for representation independence. We plan to use this as basis for a proof system that allows use of observationally pure methods in specifications, which relies on relational consequences of encapsulation [7,13]. The problem is that general relational reasoning depends on read effects in frame conditions, a non-trivial if not

earthshaking extension of RL. It deserves to be studied and presented in isolation from the complications needed for encapsulation and information hiding.

This paper builds on RLI [2] and RLII [1], extending RL with pure method calls in specifications and read effects in frame conditions. This involves adding read effects to frame conditions for commands and for pure and impure methods.

Outline and contributions. Sec. 2 introduces the programming language and specifications, as well as the judgment of correctness under hypotheses. The latter is written $\Delta \vdash C : P \rightsquigarrow Q\,[\varepsilon]$. It says that under precondition P command C does not fault; if it terminates its final state satisfies Q and the computation's effects are allowed by ε. Moreover this conclusion is under hypothesis Δ, a list of method specifications. What's new in this paper is read effects in ε and Δ, and pure methods used in $\Delta, P, C, Q, \varepsilon$, specified in Δ. Sec. 3 takes the first step towards defining semantics, sketching two ways to interpret the hypotheses and pointing out a potential circularity. To dodge this circularity, semantics of expressions and formulas is parameterized on the interpretation of pure methods. Sec. 4 formalizes an extensional semantics of read effects; this is used to define correct interpretations of pure method specifications and to define the denotation of impure method specifications. The latter is like a specification statement, and is used in the operational semantics of programs; its first order semantics is justified by a closure property that is our first technical result. Sec. 5 completes the semantics of the correctness judgment, suitably instantiating the interpretation of pure methods as motivated by the rule for linking. For C verified under hypothesis Δ that specifies pure method p called in C and/or used in the specification of C, linking discharges the hypothesis. If the specification of p is unsatisfiable, it is not possible to instantiate the interpretation as required by the linking rule. This shows that a separate satisfiability check is not needed in a tool that correctly verifies the linked program, though the check maybe be helpful to flag problems early.

Sec. 6 gives selected proof rules and states the main result, soundness of the rules. Proofs and technical details that we gloss over can be found in the full version of the paper. The proofs are intricate because we work directly with small-step operational semantics, yet this is essential for the use of dynamic frames to provide flexible encapsulation of modules in RLII. But the proofs are elementary and do not involve fixpoints.

Sec. 7 briefly discusses related work. For future work, the next steps towards observational purity are (a) to extend the logic with second order framing, as in RLII but with hiding of effects, and (b) to add weak purity which allows allocation though not other effects (this is not hard but does add a few complications). Another step is to add read effects and pure methods to our prototype SMT-based verifier for RL [16,17], which already provides limited support for pure function definitions with framing, based on a version of Leino's Dafny. As a first step, we have successfully checked versions of the Cell example by manual encoding in Why3, using SMT-provers only.

2 Programs, specifications, and correctness judgments

Fig. 2 illustrates features of our programming and specification notations, by way of the Composite pattern, a well-known verification challenge problem [16,4]. A Comp is the root of a tree, nodes of which are accessible to clients. Here is an example client:

```
class Comp {
  private children: listOf(Comp);
  specpublic parent: Comp; // (private but visible in specifications)
  specpublic size: int := 1; // number of descendants
  ghost desc: rgn; // set of descendants

  method addChild(x: Comp)
    requires x ≠ null ∧ x.parent = null ∧ ...
    ensures // x is added as a child self
    writes children, x.parent, ancestors(self)`size, ancestors(self)`desc
  pure method getSize(): int
    reads size ensures result = size
  pure method ancestors(p: Comp): rgn
    reads ancestors(p)`desc, ancestors(p)`parent
    ensures result = { o ∈ alloc | type(o,Comp) ∧ p ∈ o.desc } }
```

Fig. 2. Composite example, adapted from RLI. Ghost code maintains the invariant that desc is the set of descendants.

```
var b, c, d: Comp; var i: int; ... i := d.getSize(); b.add(c); assert i = d.getSize();
```

To prove the assertion we want to frame the formula i = d.getSize() over the call b.add(c). The frame condition of addChild says it is allowed to write **self**.children, x.parent, and the size and desc fields of the ancestors of **self**. In method set (Fig. 1) we use `**any** to abstract from field names, but here both size and desc are appropriate to make visible in the interface. (See RLI for more discussion of this facet of information hiding.) The frame condition would be less precise using ancestors(**self**)`**any**.

In order to reason using the frame rule (Fig. 10), we establish a subsidiary judgment written $\vdash \mathsf{rd}\, i, d, d.size \;\mathsf{frm}\; i = d.getSize()$ which says the formula $i = d.getSize()$ depends only on the values of i, d, and $d.size$. The rules let us establish this judgment based on the specification of $getSize$. The frame rule also requires us to establish validity of a so-called ***separator formula***. This formula is determined from the frame of the formula and from the write effect of addChild. The function $\cdot/\!\!/\cdot$ generates the separator formula and is defined by recursion on syntax.[3] In the example, we compute $\varepsilon \,/\!\!/\, (\mathsf{rd}\, i, d, d.size)$, where ε is the write effect of addChild. The formula is the disjointness {d} # ancestors(b), which says the singleton region $\{d\}$ is disjoint from the set of ancestors. It needs to hold following the elided part of the example client. In general, $\eta \,/\!\!/\, \varepsilon$ is a formula which implies that the locations writable according to ε are disjoint from the locations readable according to η.

Fig. 3 gives the grammar of programs, revised from RLII to allow method calls in expressions. We assume given a fixed collection of classes. A class has a name and some typed fields. We do not formalize dynamic dispatch or even associate methods with classes; so the term 'method' is just short for procedure. For expository clarity methods have exactly one parameter (plus res for pure methods).

[3] Please note that $\cdot/\!\!/\cdot$ is not syntax in the logic; it's a function in the metalanguage that is used to obtain formulas from effects; see Sec. 6.

$m, p \in \mathit{MethName}$ $\quad x, y \in \mathit{VarName}$ $\quad f, g \in \mathit{FieldName}$ $\quad K \in \mathit{DeclaredClassNames}$

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

Fig. 3. Programming language, highlighting additions to RLII [1].

The linking construct $\mathsf{let}\ m(x{:}T, \mathsf{res}{:}U) = C\ \mathsf{in}\ C'$ designates that m is pure, with return type U, as indicated by the distinguished variable name res. It binds x, res, and m in C, and m in C'. Calls of m are expressions and pass a single argument. The body C is executed in a state with both x and res, the latter initialized to the default value for type U. The final value of res is the value of the call expression. The linking construct $\mathsf{let}\ m(x{:}T) = C\ \mathsf{in}\ C'$ designates that m is impure; command $m(y)$ depicts its call.

Typing contexts, ranged over by Γ, are finite maps, written in conventional form. The judgment $\Gamma \vdash E : T$ means that E is well-formed and has type T. The typing rules straightforward. A command C is ***well-formed in context*** Γ provided that it is typable, i.e., $\Gamma \vdash C$, and in addition method call expressions $m(F)$ occur only in assignments $x := m(F)$ to a simple variable and with F free of method calls.

Values of type K are references to objects of class K (including the improper reference **null**). Value of type rgn are sets of references of any type. If $\Gamma \vdash G : \mathsf{rgn}$ then $\Gamma \vdash G\text{`}f : \mathsf{rgn}$ for any field name $f$ of region or reference type. In case $f : K$, the value of $G\text{`}f$ is the set of f-values of objects in G. In case $f : \mathsf{rgn}$, the value of $G\text{`}f$ is the union of the f-values. Aside from allocation and dereference (in the command forms $x := y.f$ and $y.f := F$), the only operation on references is equality test.

The syntax of formulas is standard and unchanged from RLI (Sec. 4.2), except that now the expressions include method calls, as in the points-to predicate $x.f = E$ and region containment $G \subseteq G'$.

$$P ::= E = E \mid x.f = E \mid G \subseteq G \mid (\forall x : K \in G.\, P) \mid P \wedge P \mid \neg P$$

The formula $\forall x : K \in G.\, P$ quantifies over all non-null references of type K in G. For disjointness of regions it is convenient to write $G \# H$ for $G \cap H \subseteq \{\mathsf{null}\}$.

Specifications. ***Effects*** are given by $\varepsilon ::= \mathsf{rd}\, x \mid \mathsf{rd}\, G\text{`}f \mid \mathsf{wr}\, x \mid \mathsf{wr}\, G\text{`}f \mid \mathsf{fr}\, G \mid \varepsilon, \varepsilon \mid (\mathit{empty})$. Effects must be ***syntactically well-formed*** (***swf***) for the context Γ in which they occur: $\mathsf{rd}\, x$ and $\mathsf{wr}\, x$ are swf if $x \in \mathit{dom}(\Gamma)$; $\mathsf{rd}\, G\text{`}f$, $\mathsf{wr}\, G\text{`}f$, and $\mathsf{fr}\, G$ are swf if G is swf in Γ. In particular, if G is a call $m(F)$ to a pure method, then it must be that $\Gamma \vdash m(F) : \mathsf{rgn}$. The freshness effect $\mathsf{fr}\, G$ says the value of G in the final state contains only (but not necessarily all) references that were not allocated in the initial state. Later we use the term 'well-formed', without qualification, to mean in addition that the expressions do not depend on pure methods invoked outside their preconditions.

Specifications for impure methods take the form $(x{:}T)R \rightsquigarrow S\,[\eta]$ and for pure methods the form $(x{:}T, \mathsf{res}{:}U)R \rightsquigarrow S\,[\eta]$: x is the parameter (passed by value), R the

$$
\begin{array}{ll}
\mathit{df}(m(F),\Delta) & = \mathit{df}(P_F^x,\Delta) \wedge \mathit{df}(F,\Delta) \wedge P_F^x \text{ where } \Delta(m) = (x:T, \mathsf{res}:U)P \rightsquigarrow Q\,[\varepsilon] \\
\mathit{df}(x.f = E,\Delta) & = x \neq \mathsf{null} \Rightarrow \mathit{df}(E,\Delta) \\
\mathit{df}(G_1 \subseteq G_2,\Delta) & = \mathit{df}(G_1,\Delta) \wedge \mathit{df}(G_2,\Delta) \\
\mathit{df}(\forall x : K \in G.\, P,\Delta) & = \mathit{df}(G,\Delta) \wedge \forall x : K \in G.\, \mathit{df}(P,\Delta) \\
\mathit{df}(P_1 \wedge P_2,\Delta) & = \mathit{df}(P_1,\Delta) \wedge (P_1 \Rightarrow \mathit{df}(P_2,\Delta))
\end{array}
$$

Fig. 4. Definedness formulas for expressions and formulas (selected), for swf method context Δ.

precondition, S the postcondition, and η the effects. For these specifications to be swf in context Γ, η must not include $\mathsf{wr}\,x$. (This is standard in Hoare logic; postconditions refer to initial parameter values.) Moreover, R must be typable in $\Gamma, x{:}T$. Both S and η must be typable in $\Gamma, x{:}T$, for the impure form or $\Gamma, x{:}T, \mathsf{res}{:}U$, for the pure form. Finally, for the pure method there can be no write effects in η. Although the body of a pure method will write res, the semantics is a return value, not an observable mutation of state. In this paper, there's no need for impure methods to have read effects, but they will be needed for reasoning about data abstraction and observational purity. In any context Γ, there is a read effect that imposes no restriction: $\mathsf{rd}\,\mathit{vars}(\Gamma), \mathsf{rd}\,\mathsf{alloc}\text{`}\mathsf{any}$.

A ***method context*** Δ is a finite map from method names to specifications. We are interested in specifications that may refer to global variables declared in some typing context Γ. Moreover, specifications in Δ are allowed to refer to any of the pure methods in Δ; the specification of p may have calls to p in its post-condition and effect, or p and m may refer mutually to each other —subject to the restriction that calls in preconditions must exhibit acyclic dependency. To make this restriction precise, we define a relation $\prec_\Delta$ on method names: $m \prec_\Delta m'$ iff m' occurs in the precondition of $\Delta(m)$. Now we can define what it means for a context Δ to be swf in Γ. First, the transitive closure, $\prec_\Delta^+$, is irreflexive. Second, the domains of Γ and Δ are disjoint and each specification is swf in the context $\mathit{vars}(\Gamma), \mathit{sigs}(\Delta)$. Here *sigs* extracts the types of methods. For example, let Δ_0 be $m : (x{:}T)R \rightsquigarrow S\,[\eta],\ p : (y{:}V, \mathsf{res}{:}U)P \rightsquigarrow Q\,[\varepsilon]$. Then $\mathit{sigs}(\Delta_0)$ is $m : (x{:}T),\ p : (y{:}V, \mathsf{res}{:}U)$. Also *vars* discards method declarations.

A ***correctness judgment*** takes the form $\Delta \vdash^\Gamma C :\ P \rightsquigarrow Q\,[\varepsilon]$. It is swf iff Δ is swf in Γ and C, P, Q, ε are all swf in $\mathit{vars}(\Gamma), \mathit{sigs}(\Delta)$. We often elide Γ.

Sound proof rules for correctness judgments prevent a pure method from being applied outside its precondition, to avoid the need to reason about undefined or faulty values. As is common in VC-generation, we use ***definedness formulas***, see Fig. 4. The idea is that in states where $\mathit{df}(P,\Delta)$ holds, evaluation of P does not depend on values of pure methods outside their preconditions. Although the clause for $\mathit{df}(m(F),\Delta)$ refers to a method specification that may refer to another pure method in its precondition, *df* is well-defined, owing to the requirement that $\prec_\Delta^+$ is irreflexive (and $\mathit{dom}\,\Delta$ is finite).

An expression or formula will be considered well-formed if its definedness formula is valid, in addition to it being swf. To define validity, we proceed to semantics.

3 Semantics of expressions and formulas

There are two approaches to semantics of a judgment $\Delta \vdash C :\ P \rightsquigarrow Q\,[\varepsilon]$. The first goes by quantifying over all correct implementations of the procedures specified by Δ. The second goes by using nondeterminacy to represent a 'worst implementation' of each

procedure, akin to the 'specification statement' used in axiomatic semantics. The second avoids a quantification and has been found to be quite effective [14,1]; we use it for impure methods (and in so doing show how the specification statement can include read effects). However, for pure method calls in formulas conventional semantics requires determinate values, so we use the first approach for pure methods.[4]

The transition semantics uses an environment for let-bound methods. A call to such m results in execution of the body found in the method environment. By contrast, if m is declared in Δ then its call is a single step in accord with its specification. If m is impure, the step goes to any state allowed by the specification $\Delta(m)$; we describe this by a relation $[\![\Delta(m)]\!]$ (Def. 2). If m is pure, we need a determinate result value but no change of state. So we use a function $\theta(m)$ to provide this value. The semantics of a correctness judgment (Def. 5) quantifies over all θ such that $\theta(m)$ conforms to the specification $\Delta(m)$ for each m in $\mathit{dom}\,\Delta$ (Def. 3). This is similar to an axiomatic semantics where $\theta(m)$ is an uninterpreted function constrained by $\Delta(m)$.

To define what it means for $\theta(m)$ to conform, and to define $[\![\Delta(m)]\!]$, we need semantics of expressions, formulas, and effects —and these depend on the meaning of pure method calls. To break this circularity, we define in this section a notion of candidate interpretation, and define the semantics of formulas and expressions with respect to any candidate interpretation θ.

We assume given an infinite set *Ref* of reference values including a distinguished 'improper reference' *null*. A ***Γ-state*** is comprised of a global heap and a store. The store is a type-respecting assignment of values to the variables in Γ and to the variable $\mathsf{alloc} : \mathsf{rgn}$ which is special. Updates of alloc are built into the program semantics so that alloc holds the set of all allocated references. We write $\sigma(x)$ for the value of variable x in state σ, $\sigma(o.f)$ to look up field f of object o in the heap, $\mathit{Dom}(\sigma)$ for the variables of σ, and $[\![\Gamma]\!]$ for the set of Γ-states. We write $[\![T]\!]\sigma$ for the set of values of type T in state σ. Thus $[\![\mathsf{int}]\!]\sigma = \mathbb{Z}$ and $[\![K]\!]\sigma = \{\mathit{null}\} \cup \{o | o \in \sigma(\mathsf{alloc}) \wedge \mathit{Type}(o, \sigma) = K\}$. Besides states, we use the faulting outcome $\lightning$ for runtime errors (null-dereference), and also to signal precondition violations as described later.

For a typing context Γ, a ***candidate Γ-interpretation*** θ is a mapping from the pure method names in Γ such that if $\Gamma(m) = (x : T, \mathsf{res} : U)$ then $\theta(m)$ is a function such that for any T-value t and state σ, $\theta(\sigma, t)$ is a U-value or $\lightning$. To be precise, $\theta(m)$ has the dependent type $(\sigma \in [\![\Gamma]\!]) \times [\![T]\!]\sigma \to ([\![U]\!]\sigma \cup \{\lightning\})$. A ***candidate Δ-interpretation*** is just a candidate $\mathit{sigs}(\Delta)$-interpretation.

The denotation of an expression F in candidate Γ-interpretation θ and state σ is written $[\![F]\!]_\theta\sigma$ and defined straightforwardly, see Fig. 5. The second line in the figure is for application $m(F)$ of a pure method: evaluate F to get a value v, then apply the function $\theta(m)$ to the pair (σ, v). Using the semantics for expressions we define the 3-valued semantics of formulas in Fig 6. We also define $\sigma \models^\Gamma_\theta P$ iff $[\![P]\!]_\theta\sigma = \mathsf{true}$. Fig. 7 shows that when the definedness formulas hold, the usual 2-valued clauses hold.

[4] This does not preclude nondeterminacy modulo an equivalence relation, which is especially important for 'weakly pure' methods that return freshly allocated references [13]. For VCs this is explored in [10].

$$
\begin{array}{ll}
[\![E_1 + E_2]\!]_\theta\sigma & = \mathit{let}\ v_1 = [\![E_1]\!]_\theta\sigma\ \mathit{in}\ \mathit{let}\ v_2 = [\![E_2]\!]_\theta\sigma\ \mathit{in}\ v_1 + v_2 \\
[\![m(F)]\!]_\theta\sigma & = \mathit{let}\ v = [\![F]\!]_\theta\sigma\ \mathit{in}\ \theta(m)(\sigma, v) \\
[\![\{E\}]\!]_\theta\sigma & = \mathit{let}\ v = [\![E]\!]_\theta\sigma\ \mathit{in}\ \{v\} \\
[\![G\text{`}f]\!]_\theta\sigma & = \mathit{let}\ X = [\![G]\!]_\theta\sigma\ \mathit{in}\ \{\sigma(o.f) \mid o \in X \wedge o \neq \mathit{null} \wedge \mathit{Type}(o,\sigma) = \mathit{DeclClass}(f)\} \\
 & \quad \text{if } f : K \text{ for some } K \\
 & = \mathit{let}\ X = [\![G]\!]_\theta\sigma\ \mathit{in}\ \bigcup\{\sigma(o.f) \mid o \in X \wedge o \neq \mathit{null} \wedge \mathit{Type}(o,\sigma) = \mathit{DeclClass}(f)\} \\
 & \quad \text{if } f : \mathsf{rgn}
\end{array}
$$

Fig. 5. Semantics of selected program and region expressions, for state σ and candidate interpretation θ. We use the $\lightning$-strict let-binder, i.e., '*let* $v = X$ *in* Y' denotes $\lightning$ if X denotes $\lightning$.

$$
\begin{array}{ll}
[\![E_1 = E_2]\!]_\theta\sigma & = \mathit{let}\ v_1 = [\![E_1]\!]_\theta\sigma\ \mathit{in}\ \mathit{let}\ v_2 = [\![E_2]\!]_\theta\sigma\ \mathit{in}\ \mathit{if}\ v_1 = v_2\ \mathit{then}\ \mathit{true}\ \mathit{else}\ \mathit{false} \\
[\![x.f = E]\!]_\theta\sigma & = \mathit{if}\ \sigma(x) = \mathit{null}\ \mathit{then}\ \mathit{false}\ \mathit{else}\ \mathit{let}\ v = [\![E]\!]_\theta\sigma\ \mathit{in} \\
 & \qquad \mathit{if}\ \sigma(x.f) = v\ \mathit{then}\ \mathit{true}\ \mathit{else}\ \mathit{false} \\
[\![x.f = E]\!]_\theta\sigma & = \mathit{let}\ v = [\![E]\!]_\theta\sigma\ \mathit{in}\ \mathit{if}\ \sigma(x) \neq \mathit{null}\ \text{and}\ \sigma(x.f) = v\ \mathit{then}\ \mathit{true}\ \mathit{else}\ \mathit{false} \\
[\![G_1 \subseteq G_2]\!]_\theta\sigma & = \mathit{let}\ X_1 = [\![G_1]\!]_\theta\sigma\ \mathit{in}\ \mathit{let}\ X_2 = [\![G_2]\!]_\theta\sigma\ \mathit{in}\ \mathit{if}\ X_1 \subseteq X_2\ \mathit{then}\ \mathit{true}\ \mathit{else}\ \mathit{false} \\
[\![\Gamma \vdash \forall x : K \in G.\, P]\!]_\theta\sigma & = \lightning \text{ if } [\![G]\!]_\theta\sigma = \lightning \text{ or } [\![\Gamma, x : K \vdash P]\!]_\theta \mathit{Extend}(\sigma, x, o) = \lightning \\
 & \qquad \text{for some } o \text{ in } ([\![G]\!]_\theta\sigma)\backslash\{\mathit{null}\} \text{ with } \mathit{Type}(o,\sigma) = K \\
 & = \mathit{true} \text{ if } [\![\Gamma, x : K \vdash P]\!]_\theta \mathit{Extend}(\sigma, x, o) = \mathit{true} \\
 & \qquad \text{for all } o \text{ in } ([\![G]\!]_\theta\sigma)\backslash\{\mathit{null}\} \text{ with } \mathit{Type}(o,\sigma) = K \\
 & = \mathit{false}\ \text{otherwise}
\end{array}
$$

Fig. 6. Formulas: three-valued semantics, $[\![\Gamma \vdash P]\!]_\theta\sigma \in \{\mathit{true}, \mathit{false}, \lightning\}$. Typing context is elided in most cases.

4 Semantics of effects and programs

Effects. A ***location*** is either a variable name x or a heap location comprised of a reference o and field name f. We write $o.f$ for such pairs. Define $\mathit{rlocs}(\sigma, \theta, \varepsilon)$, the locations designated by read effects of ε, in σ, by $\mathit{rlocs}(\sigma, \theta, \varepsilon) = \{x \mid \varepsilon \text{ contains } \mathsf{rd}\, x\} \cup \{o.f \mid \varepsilon \text{ contains } \mathsf{rd}\, G\text{`}f \text{ with } o \in [\![G]\!]_\theta\sigma\}$. Define *wlocs* similarly, for write effects.

Write effects constrain what locations are allowed to change between one state and another. We say ε ***allows change from*** σ ***to*** τ ***under*** θ, written $\sigma \to \tau \models_\theta \varepsilon$, provided (a) if y changed value (i.e., $\tau(y) \neq \sigma(y)$) then $\mathsf{wr}\, y$ is in ε; (b) if $o.f$ changed value then there is $\mathsf{wr}\, G\text{`}f$ in ε such that $o \in [\![G]\!]_\theta\sigma$; and (c) if $\mathsf{fr}\, G$ is in ε then elements of G in τ are fresh. Reads are ignored, so $\sigma \to \tau \models_\theta \varepsilon$ iff $\sigma \to \tau \models_\theta \mathit{writes}(\varepsilon)$. In (b), region expressions G are interpreted in the initial state because frame conditions need only report writes to fields of pre-existing objects and not freshly allocated objects.

Read effects constrain what locations an outcome can depend on. Dependency is expressed by considering two initial states that agree on the locations deemed readable. Agreement needs to take into account variation in allocation, as two states may have isomorphic pointer structure but differently chosen references.

Let π range over ***partial bijections*** on Ref. We write $\pi(p) = p'$ to express that π is defined on p and has value p'. A ***refperm*** from σ to σ' is partial bijection π such that

- $\mathit{dom}(\pi) \subseteq \sigma(\mathsf{alloc}) \cup \{\mathit{null}\}$ and $\mathit{rng}\,(\pi) \subseteq \sigma'(\mathsf{alloc}) \cup \{\mathit{null}\}$
- $\pi(\mathit{null}) = \mathit{null}$
- $\pi(p) = p'$ implies $\mathit{Type}(p, \sigma) = \mathit{Type}(p', \sigma')$ for all proper references p, p'

$\sigma \models_\theta E_1 = E_2$ iff $[\![E_1]\!]_\theta\sigma = [\![E_2]\!]_\theta\sigma$
$\sigma \models_\theta x.f = E$ iff $\sigma(x) \neq \mathit{null}$ and $\sigma(x.f) = [\![E]\!]_\theta\sigma$
$\sigma \models_\theta G_1 \subseteq G_2$ iff $[\![G_1]\!]_\theta\sigma \subseteq [\![G_2]\!]_\theta\sigma$
$\sigma \models^\Gamma_\theta \forall x : K \in G.\, P$ iff $\mathit{Extend}(\sigma, x, o) \models^{\Gamma, x:K}_\theta P$
for all o in $([\![G]\!]_\theta\sigma)\backslash\{\mathit{null}\}$ with $\mathit{Type}(o,\sigma) = K$
$\sigma \models_\theta P_1 \wedge P_2$ iff $\sigma \models_\theta P_1$ and $\sigma \models_\theta P_2$
$\sigma \models_\theta \neg P$ iff $\sigma \not\models_\theta P$

Fig. 7. Two-valued semantics.These clauses hold when $\sigma \models_\theta \mathit{df}(P, \Delta)$ (Lemma 6).

Define $p \overset{\pi}{\sim} p'$ to mean $\pi(p) = p'$. We extend $\overset{\pi}{\sim}$ to a relation on integers by $i \overset{\pi}{\sim} j$ iff $i = j$. For reference sets X, Y we define $X \overset{\pi}{\sim} Y$ iff $\pi(X) \supseteq Y$ and $X \subseteq \pi^{-1}(Y)$ (where $\pi(X)$ is the direct image of X). That is, π forms a bijection between X and Y.

Define $\mathit{freshLocs}(\sigma, \tau) = \{p.f \,|\, p \in \mathit{freshRefs}(\sigma, \tau) \wedge f \in \mathit{Fields}(\mathit{Type}(p, \tau))\}$ where $\mathit{freshRefs}(\sigma, \tau) = \tau(\mathsf{alloc})\backslash\sigma(\mathsf{alloc})$. For a set W of variables and heap locations, define $\mathit{Lagree}(\sigma, \sigma', W, \pi)$ iff $\forall x \in W.\, \sigma(x) \overset{\pi}{\sim} \sigma'(x)$ and $\forall (o.f) \in W.\, o \in dom(\pi) \wedge \sigma(o.f) \overset{\pi}{\sim} \sigma'(\pi(o).f)$.

Definition 1 (agreement on read effects) Let ε be an effect that is swf in Γ. Consider states σ, σ'. Let π be a partial bijection. Let θ be a candidate interpretation (for some Δ that is swf in Γ). Say that σ and σ' ***agree on ε modulo π***, written $\mathit{Agree}(\sigma, \sigma', \varepsilon, \pi, \theta)$, iff $\mathit{Lagree}(\sigma, \sigma', \mathit{rlocs}(\sigma, \theta, \varepsilon), \pi)$. Define $\mathit{Agree}(\sigma, \sigma', \varepsilon, \theta) = \mathit{Agree}(\sigma, \sigma', \varepsilon, \pi, \theta)$ where π is the identity on $\sigma(\mathsf{alloc}) \cap \sigma'(\mathsf{alloc})$.

Programs. In the following we consider a method context Δ that is well-formed in some typing context Γ (often elided). For substitution we use the notation P^x_e. For clarity we use substitution notation in satisfaction statements, even though strictly speaking the syntax does not (and should not) include reference literals. If $\Gamma, x : T \vdash P$ and $\sigma \in [\![\Gamma]\!]$ and $v \in [\![T]\!]\sigma$, we may write $\sigma \models^\Gamma_\theta P^x_v$ to abbreviate $\mathit{Extend}(\sigma, x, v) \models^{\Gamma, x:T}_\theta P$.

The transition relation depends on a method context Δ. Configurations take the form $\langle C, \sigma, \mu\rangle$ where μ is a method environment. The call of a let-bound method m executes the body $\mu(m)$ with variables renamed to avoid clashes with the calling context. In case of a pure method the call takes the form $y := m(F)$ and there is some extra bookkeeping to assign the final value of res (or rather, a fresh instance thereof) to y.

The transition semantics for pure method call $y := m(F)$ takes a step that assigns to y the value $[\![m(F)]\!]_\theta\sigma$ (defined in Fig. 5). The transition semantics of a call $m(z)$, for impure m in Δ, takes a single step to a final state (or $\lightning$) that satisfies the specification $\Delta(m)$. Such states are described by the denotation $[\![\Delta(m)]\!]$ of the specification.

Definition 2 (Denotation of impure method spec) Let Δ be swf and let $(x{:}T)R \rightsquigarrow S\,[\eta]$ be in Δ. Let θ be a candidate interpretation of Δ and z a variable name. Then $[\![(x{:}T)R \rightsquigarrow S\,[\eta]]\!](\theta, z)$ is defined as follows, for any $\Gamma_1 \supseteq \Gamma$ and Γ_1-states σ, τ :

(i) $[\![(x{:}T)R \rightsquigarrow S\,[\eta]]\!](\theta, z)\sigma\lightning$ iff $\sigma \not\models R^x_z$
(ii) $[\![(x{:}T)R \rightsquigarrow S\,[\eta]]\!](\theta, z)\sigma\tau$ iff
(a) $\sigma \models^{\Gamma_1}_\theta R^x_z$ and $\tau \models^{\Gamma_1}_\theta S^x_z$ and $\sigma{\rightarrow}\tau \models_\theta \eta^x_z$ and

$$\frac{[\![\Delta(m)]\!](\theta,z)\,\sigma\,\tau}{\langle m(z),\,\sigma,\,\mu\rangle \overset{\Delta,\theta}{\longmapsto} \langle \mathsf{skip},\,\tau,\,\mu\rangle} \qquad \frac{[\![\Delta(m)]\!](\theta,z)\,\sigma\,\lightning}{\langle m(z),\,\sigma,\,\mu\rangle \overset{\Delta,\theta}{\longmapsto} \lightning}$$

$$\frac{[\![m(F)]\!]_\theta\sigma \neq \lightning \qquad \tau = [\sigma \,|\, x : [\![m(F)]\!]_\theta\sigma]}{\langle x := m(F),\,\sigma,\,\mu\rangle \overset{\Delta,\theta}{\longmapsto} \langle \mathsf{skip},\,\tau,\,\mu\rangle} \qquad \frac{[\![m(F)]\!]_\theta\sigma = \lightning}{\langle x := m(F),\,\sigma,\,\mu\rangle \overset{\Delta,\theta}{\longmapsto} \lightning}$$

Fig. 8. Transition rules for calls of impure and pure procedures in context Δ.

(b) for all σ', π, if $\mathit{Agree}(\sigma, \sigma', \eta^x_z, \pi, \theta)$ and $\sigma' \models^{\Gamma_1}_\theta R^x_z$ then there are τ', ρ with
- $\tau' \models^{\Gamma_1}_\theta S^x_z$ and $\sigma' \to \tau' \models_\theta \eta^x_z$
- $\rho \supseteq \pi$ and $\mathit{freshRefs}(\sigma', \tau') \subseteq \rho(\mathit{freshRefs}(\sigma, \tau))$
- $\mathit{Lagree}(\tau, \tau', X, \rho)$ where $X = \mathit{freshLocs}(\sigma, \tau) \cup \mathit{wlocs}(\sigma, \theta, \eta^x_z)$

It is item (ii)(b) that is new in this paper; the rest is from RLII. Note that X is defined by interpreting η in the initial state.

A state σ may have no successor because the specification is unsatisfiable at σ. Unsatisfiability may be due to the postcondition, but it can also happen that τ satisfies the postcondition but not the read effect. The specification $(x{:}\mathsf{Cell})\mathit{true} \rightsquigarrow y = x.\mathit{val}\,[\mathsf{wr}\,y]$ is unsatisfiable: y cannot be set to $x.\mathit{val}$ without reading $\{x\}\text{`}\mathit{val}$ or having a stronger precondition like $y = x.\mathit{val}$.

Although specifications include read effects —a relational property— the denotation of a specification need not be defined as an extreme solution to constraints including that relational property. The elementary definition above has the property that any τ' that satisfies the conditions in (ii) is a possible successor of σ', i.e., the denotation is closed in the sense that it includes the pair σ', τ'. This is made precise in Thm. 9. The condition $\mathit{freshRefs}(\sigma', \tau') \subseteq \rho(\mathit{freshRefs}(\sigma, \tau))$ in (ii)(b) was not immediately obvious but is crucial for Thm. 9.

With all the ingredients in hand, the transition semantics can be defined; see Fig. 8.

5 Semantics of correctness judgments

To link a client C with implementation B of a method m used by C we want C to be correct for all interpretations of the method context. But reasoning about B can use a particular interpretation for m. Such an interpretation might be provided directly, as a mathematical definition provided by the programmer, or it might be derived from the code as it is in work on VC generation for pure methods [5]. Here we treat such interpretations semantically. To that end, we generalize the correctness judgment form to $\Delta; \theta \vdash^\Gamma C : P \rightsquigarrow Q\,[\varepsilon]$. For this to be swf, θ should be a candidate interpretation of some subset of Δ, and $\Delta \vdash^\Gamma C : P \rightsquigarrow Q\,[\varepsilon]$ should be swf as defined in Sec. 2. The original form is essentially the special case where θ is the empty function. The generalized correctness judgment is important for the linking rule, which we introduce here in abridged form. We consider a single method specification $\Theta \equiv m : (x{:}T, \mathsf{res}{:}U)\,Q \rightsquigarrow Q'$, we elide effects, and the partial interpretation of the

ambient library Δ is empty.

$$\frac{\Delta,\Theta;\varnothing \vdash C : P \rightsquigarrow P' \qquad \Delta,\Theta;\theta \vdash B : Q \rightsquigarrow Q' \qquad \mathit{dom}\,\theta = \{m\} \qquad \theta \models \Delta,\Theta}{\Delta;\varnothing \vdash \mathsf{let}\ m(x{:}T,\mathsf{res}{:}U) = B\ \mathsf{in}\ C : P \rightsquigarrow P'} \tag{1}$$

A client C is linked with the implementation B of a pure procedure m. The verification condition for C is under the hypothesis of some specifications Δ,Θ which include the specification Θ of m. The rule may only be instantiated with swf judgments, so Δ is swf (as it appears in the conclusion) and the larger method context Δ,Θ is also swf.

According to the semantics to follow, the judgment for C means that it is correct with respect to any interpretation φ of all the pure procedures in Δ,Θ. The verification condition for B also has hypothesis Δ,Θ for procedures that may be called in B or used in its specification, and B must be correct with respect to any interpretation of the pure procedures in Δ, but fixed interpretation θ of m. The rule requires that in fact θ is an interpretation of Θ, meaning that $\theta(m)$ satisfies the specification of m. Because this specification may refer to pure methods in the ambient context Δ, satisfaction is expressed as $\theta \models \Delta,\Theta$. This is defined in terms of the following.

Definition 3 (context interpretation) Let Δ be swf in Γ and let θ be a candidate Δ-interpretation. (Note that $\mathit{dom}\,\theta = \mathit{dom}\,\Delta$.) Say θ is a ***Δ-interpretation*** iff the following holds for each $m : (x{:}T,\mathsf{res}{:}U)P \rightsquigarrow Q\,[\varepsilon]$ in Δ. For any $\sigma \in [\![\Gamma]\!]$ and $v \in [\![T]\!]\sigma$,

(a) $\theta(m)(\sigma,v) = \lightning$ iff $\sigma \not\models_\theta P^x_v$

Furthermore, if $\sigma \models_\theta P^x_v$ then letting $w = \theta(m)(\sigma,v)$ we have

(b) $\sigma \models_\theta Q^{x,\mathsf{res}}_{v,w}$
(c) for any $\sigma' \in [\![\Gamma]\!]$, $v' \in [\![T]\!]\sigma'$ with $\sigma' \models_\theta P^x_{v'}$, and any refperm π from σ to σ', if $v \overset{\pi}{\sim} v'$ and $\mathit{Agree}(\sigma,\sigma',\varepsilon,\pi,\theta)$ then $w \overset{\pi}{\sim} w'$ where $w' = \theta(m)(\sigma',v')$

Definition 4 (partial context interpretation) Let Δ be swf and Δ,Θ be swf. Let θ be a candidate interpretation of Θ. We say θ is a ***partial interpretation*** of Δ,Θ, written $\theta \models \Delta,\Theta$, provided that for any Δ-interpretation δ, the candidate $\delta \cup \theta$ is a (Δ,Θ)-interpretation.[5]

Definition 5 (valid judgment) A swf correctness judgment $\Delta;\theta \vdash^\Gamma C :\ P \rightsquigarrow Q\,[\varepsilon]$ is ***valid*** iff the following conditions hold for all Γ-environments μ, all Δ-interpretations δ such that $\theta \subseteq \delta$, and all states σ such that $\sigma \models^{\Gamma,\mathit{sigs}(\Delta)}_\delta P$.

(Safety) It is not the case that $\langle C,\sigma,\mu\rangle \overset{\Delta,\delta}{\longmapsto}{}^* \lightning$.
(Post) $\tau \models_\delta Q$ for every τ with $\langle C,\sigma,\mu\rangle \overset{\Delta,\delta}{\longmapsto}{}^* \langle \mathsf{skip},\tau,\mu\rangle$
(Effect) $\sigma{\to}\tau \models_\delta \varepsilon$ for every τ with $\langle C,\sigma,\mu\rangle \overset{\Delta,\delta}{\longmapsto}{}^* \langle \mathsf{skip},\tau,\mu\rangle$

[5] Under these conditions, if the specifications in Θ refer to methods in Δ, Θ is not swf on its own, and then it is not meaningful to call θ a Θ-interpretation.

(Read Effect) for any τ such that $\langle C, \sigma, \mu\rangle \overset{\Delta,\delta}{\longmapsto}{}^{*} \langle \mathsf{skip}, \tau, \mu\rangle$, and any σ', π, τ such that $\mathit{Agree}(\sigma, \sigma', \varepsilon, \pi, \delta)$ and $\sigma' \models^{\Gamma}_{\delta} P$, there are τ', ρ such that

$\langle C, \sigma', \mu\rangle \overset{\Delta,\delta}{\longmapsto}{}^{*} \langle \mathsf{skip}, \tau', \mu\rangle$ and $\rho \supseteq \pi$ and $\mathit{freshLocs}(\sigma', \tau') \subseteq \rho(\mathit{freshLocs}(\sigma, \tau))$ and $\mathit{Lagree}(\tau, \tau', X, \rho)$ where $X = \mathit{freshLocs}(\sigma, \tau) \cup \mathit{wlocs}(\sigma, \delta, \varepsilon)$.

In case $\Delta(m)$ is unsatisfiable (except possibly by divergence), no Δ-interpretation exists. Then the judgment holds but the hypotheses cannot be discharged by linking because there is no way to instantiate θ in rule (1).

The definitions up to this point apply even if pure methods are called outside their precondition. For understandable proof rules, and to stay within FOL for assertions, we will disallow such specifications and correctness judgments.

Lemma 6 (two-valued semantics of formulas) If θ is a Δ-interpretation and $\sigma \models_\theta \mathit{df}(P, \Delta)$ then $[\![P]\!]_\theta \sigma$ is not $\lightning$. And for any σ and any Δ-interpretation θ, if $\sigma \models_\theta \mathit{df}(P, \Delta)$ then $\sigma \models_\theta P$ satisfies the usual defining clause, see Fig. 7.

Definition 7 Let Γ be a typing context and let Δ be a specification context that is swf in Γ. Let P be a formula that is swf in $\mathit{vars}(\Gamma), \mathit{sigs}(\Delta)$. Then P is ***Δ-valid***, written $\Delta \models P$, if and only if $\sigma \models_\theta P$ for all Δ-interpretations θ and all states σ.

Definition 8 (healthy, well-formed) Let Γ and Δ satisfy the conditions of Def. 7. A formula P that is swf is ***healthy*** iff $\mathit{df}(P, \Delta)$ is valid. A swf specification $P \rightsquigarrow Q\,[\eta]$ is ***healthy*** (with respect to Γ, Δ) iff the three formulas $\mathit{df}(P, \Delta)$, $P \Rightarrow \mathit{df}(Q, \Delta)$, and $P \Rightarrow \mathit{df}(\eta, \Delta)$ are Δ-valid. A swf correctness judgment $\Delta; \theta \vdash^{\Gamma} C : P \rightsquigarrow Q\,[\eta]$ is ***healthy*** iff the three formulas $\mathit{df}(P, \Delta)$, $P \Rightarrow \mathit{df}(Q, \Delta)$, and $P \Rightarrow \mathit{df}(\eta, \Delta)$ are Δ-valid. The term ***well-formed*** means swf and healthy.

The definitions to this point are intricate but elementary; in particular, there are no fixpoints. But by contrast with axiomatic semantics, correctness is directly grounded in a conventional operational semantics. The one unconventional element is that transition semantics depends on method context. The ultimate confirmation that we *are* reasoning about program behavior is soundness of the linking rule, which can be used to discharge all hypotheses.

6 Proof system

The ***framing judgment*** has the form $P; \Delta \vdash \eta\ \mathsf{frm}\ Q$ and is swf under evident conditions. It means that in P-states, Q reads within the read effect η. The judgment is ***healthy*** iff the formulas $\mathit{df}(P, \Delta)$, $P \Rightarrow \mathit{df}(\eta, \Delta)$, and $P \Rightarrow \mathit{df}(Q, \Delta)$ are all Δ-valid. The judgment is ***valid***, written $P; \Delta \models^{\Gamma} \eta\ \mathsf{frm}\ Q$, iff for all Γ-states σ, σ', refperms π, and Δ-interpretations θ, if $\mathit{Agree}(\sigma, \sigma', \eta, \pi, \theta)$, and $\sigma \models^{\Gamma}_{\theta} P \wedge Q$, then $\sigma' \models^{\Gamma}_{\theta} Q$.

A verifier can check framing judgments in terms of the validity property, but our logic includes rules to derive framing judgments. A basic rule allows to infer, for atomic formula P, the judgment $\mathit{true}; \Delta \vdash \mathit{ftpt}(P, \Delta)\ \mathsf{frm}\ P$ concerning a precise footprint computed by function *ftpt* which is defined in Fig. 9. For non-atomic formulas there are

$$
\begin{array}{lll}
\mathit{ftpt}(x,\Delta) & = \mathsf{rd}\, x & \mathit{ftpt}(E = E',\Delta) \;\; = \mathit{ftpt}(E,\Delta), \mathit{ftpt}(E',\Delta) \\
\mathit{ftpt}(G\text{`}f,\Delta) & = \mathsf{rd}\, G\text{`}f, \mathit{ftpt}(G,\Delta) & \mathit{ftpt}(G_1 \subseteq G_2,\Delta) = \mathit{ftpt}(G_1,\Delta), \mathit{ftpt}(G_2,\Delta) \\
\mathit{ftpt}(\varnothing,\Delta) & = \varnothing & \mathit{ftpt}(x.f = F,\Delta) \;\; = \mathsf{rd}\, x, x.f, \mathit{ftpt}(F,\Delta) \\
\mathit{ftpt}(m(F),\Delta) & = \mathit{reads}(\varepsilon_F^x), \mathit{ftpt}(F,\Delta) & \text{for } \Delta(m) = (x : T, \mathsf{res} : U)P \rightsquigarrow Q\,[\varepsilon]
\end{array}
$$

Fig. 9. Footprints of region expressions and atomic assertions well-formed in Δ.

syntax-directed rules, e.g., the rule for conjunction allows to infer $P;\Delta \vdash \varepsilon\ \mathsf{frm}\ Q_1 \wedge Q_2$ from $P;\Delta \vdash \varepsilon\ \mathsf{frm}\ Q_1$ and $P \wedge Q_1;\Delta \vdash \varepsilon\ \mathsf{frm}\ Q_2$. There are also subsidiary rules for subsumption of effects and for logical manipulation of P. These rules are adapted in a straightforward way from RLI (Sec. 6.1).

The point of establishing $P;\Delta \vdash \eta\ \mathsf{frm}\ Q$ is that code that writes outside η cannot falsify Q. This is expressed in the frame rule by computing, from the frame η of Q and the frame condition ε of the code, a ***separator formula*** which is a conjunction of region disjointness formulas describing states in which writes allowed by ε cannot affect the value of a formula with read effect η. We define the separator formula as $\eta \mathbin{/\!\cdot\!/} \varepsilon$, using function $\cdot/\!\cdot\!/\cdot$ which recurses on syntax (see RLI Sec. 6.2). For example, $\mathsf{rd}\, G\text{`}f \mathbin{/\!\cdot\!/} \mathsf{wr}\, H\text{`}g$ is true, and $\mathsf{rd}\, G\text{`}f \mathbin{/\!\cdot\!/} \mathsf{wr}\, H\text{`}f$ is the disjointness formula $G \# H$. Also, $\mathsf{rd}\, x \mathbin{/\!\cdot\!/} \mathsf{wr}\, y$ is simply false, if x and y are the same variable, and true otherwise. Writes on the left and reads on the right are ignored, so $\eta \mathbin{/\!\cdot\!/} \varepsilon$ is the same as $\mathit{reads}(\eta) \mathbin{/\!\cdot\!/} \mathit{writes}(\varepsilon)$.

The key property of a separator is to establish the agreement to which frame validity refers. To be precise, suppose $\sigma{\rightarrow}\tau \models_\theta \varepsilon$ and $\sigma \models_\theta \eta \mathbin{/\!\cdot\!/} \varepsilon$. Then $\mathit{Agree}(\sigma,\tau,\eta,\mathit{id},\theta)$, where id is the identity on $\sigma(\mathsf{alloc})$.

An effect ε is called ***self-framing*** in method context Δ provided that for every $\mathsf{rd}\, G\text{`}f$ or $\mathsf{wr}\, G\text{`}f$ in ε, $\mathit{ftpt}(G,\Delta)$ is in ε. Such effects arise, for example, in case a method refers to itself in its frame condition. So are effects obtained using the *ftpt* function and most of the framing rules. For self-framing ε, if $\mathit{Agree}(\sigma,\sigma',\varepsilon,\pi,\theta)$ then $[\![G]\!]_\theta\sigma \overset{\pi}{\sim} [\![G]\!]_\theta\sigma'$ for any $\mathsf{rd}\, G\text{`}f$ or $\mathsf{wr}\, G\text{`}f$ in ε.

Theorem 9 (denotation closure). Suppose η is self-framing and θ is a Δ-interpretation. If $[\![(x{:}T)R \rightsquigarrow S\,[\eta]]\!](\theta,z)\sigma\tau$ then $[\![(x{:}T)R \rightsquigarrow S\,[\eta]]\!](\theta,z)\sigma'\tau'$, provided that σ',τ' satisfy the conditions in Def. 2(ii).

Proof rules for correctness judgments. Fig. 10 presents a few proof rules. They are to be instantiated only with well-formed premises and conclusions (Def. 8). The first, for field update, is a 'local axiom' that precisely describes the effect; it shows how read effects can easily be incorporated into the rules from RLI (Sec. 7.1) and RII (Sec. 7.1). Next is the frame rule, adapted from RLI/II by adding Δ to the side conditions. Then come rules for impure and pure method calls. For reasons of parsimony we make self-framing an explicit premise where needed for soundness. It turns out that in non-trivial provable judgements the frame conditions in the method context will be self-framing.

We give an illustrative rule for linking a client with a method implementation. The general rule allows several pure and impure methods that may refer to each other in their specifications and code (of course, subject to the proviso concerning $\prec_\Delta^+$ in the definition of swf method context in Sec. 2). Predicate $\mathit{terminates}(Q,B)$ says that from any Q-state, B terminates (normally or abnormally). One premise is that partial (Δ,Θ)-

$$\Delta;\varnothing \vdash x.f := F :\ x \neq \mathsf{null} \wedge y = F \rightsquigarrow x.f = y\,[\mathsf{wr}\,x.f, \mathsf{rd}\,x, \mathit{ftpt}(F,\Delta)]$$

$$\textsc{Frame}\ \frac{\Delta;\theta \vdash C :\ P \rightsquigarrow Q\,[\varepsilon] \qquad P;\Delta \vdash \eta\ \mathsf{frm}\ R \qquad \Delta;\theta \models P \wedge R \Rightarrow \eta\ \mathit{\#}.\ \varepsilon}{\Delta;\theta \vdash C :\ P \wedge R \rightsquigarrow Q \wedge R\,[\varepsilon]}$$

$$\frac{\Delta;\theta \vdash C :\ P \rightsquigarrow Q\,[\varepsilon]}{\Delta;\theta \cup \theta' \vdash C :\ P \rightsquigarrow Q\,[\varepsilon]} \qquad \frac{\varepsilon \text{ is self-framing}}{m : (x{:}T)P \rightsquigarrow Q\,[\varepsilon];\varnothing \vdash m(z) :\ P^x_z \rightsquigarrow Q^x_z\,[\varepsilon^x_z]}$$

$$\frac{x \notin \mathit{Vars}(H) \cup FV(Q)}{m : (y{:}T, \mathsf{res}{:}U)P \rightsquigarrow Q\,[\varepsilon];\varnothing \vdash x := m(H) :\ P^y_H \rightsquigarrow Q^{y,\mathsf{res}}_{H,x}\,[\mathsf{wr}\,x, \mathit{ftpt}(H,\Delta), \varepsilon^y_H]}$$

$$\frac{\begin{array}{c}\Theta \text{ is } m : (x{:}T, \mathsf{res}{:}U)Q \rightsquigarrow Q'\,[\eta] \qquad \mathit{dom}\,\theta = \mathit{dom}\,\Theta \qquad \theta \models \Delta, \Theta \\ \Delta, \Theta;\delta \vdash^{\Gamma} C :\ P \rightsquigarrow P'\,[\varepsilon] \qquad \Delta, \Theta;\delta \cup \theta \vdash^{\Gamma, x:T, \mathsf{res}:U} B :\ Q \rightsquigarrow Q'\,[\mathsf{wr}\,\mathsf{res}, \mathsf{rd}\,x, \eta] \\ \eta \text{ is } \mathsf{wr}\text{-free and self-framing} \qquad \mathit{terminates}(Q, B)\end{array}}{\Delta;\delta \vdash^{\Gamma} \mathsf{let}\ m(x{:}T, \mathsf{res}{:}U) = B\ \mathsf{in}\ C :\ P \rightsquigarrow P'\,[\varepsilon]}$$

Fig. 10. Proof rules for field update, framing, interpreting, pure/impure calls, and linking.

interpetation θ is provided; it gives the *chosen* interpretation for m, to be used in verifying the body B. By contrast, the premise for C requires correctness with respect to *all* interpretations of m.

Theorem 10. Any derivable correctness judgment is valid.

7 Related work

We take the Cell example from the most closely related work, [18], where read effects of pure methods are specified using dynamic frames and methods may be self-framing. They define (and implement) a VC-generator including VCs that encode the semantics of read effects, albeit only for a pair of states in succession. (That avoids the need for refperms, and suffices for framing but not relational reasoning for data abstraction and encapsulation.) They give a detailed proof of soundness with respect to transition semantics, by showing that the VCs ensure a small-step invariant that implies correctness and fault-avoidance. Axioms are included (and proved sound) to exploit read effects for framing. Different from our work, the body of a pure method is required to be a single '**return** E' statement and E is visible to clients; and pure methods do not have postconditions. (Their implementation does include such postconditions.) Although VCs are generated modularly, we do not discern an explicit account of linking, or an easy adaptation to cater for hiding a pure method body or invariants from clients. As usual in practical systems, the syntax embeds specifications in programs, as opposed to judgments that ascribe properties to programs.

A number of earlier works point out the importance of read effects for pure methods and explore VC-generation, e.g. [5], explore weak purity which allows allocation, and shows consistency of a system of VCs (but not operational soundness). The analog of consistency, in our setting, is being able to discharge hypotheses in the linking rule.

Framing in separation logic encompasses read and write effects, implicitly in syntax but explicitly in the semantics (safety monotonicity, frame property [14]). Whereas self-framing is a property of effects, in our setting, it is a property of formulas in other settings. In separation logic, all assertions are effectively self-framing. The abstract predicates approach [15] to data abstraction has inspired several works that cater for SMT provers by using ghost instrumentation to encode intensional semantics of effects in terms of permissions. One provides a VC generator and sketches an argument for its operational soundness [6]. Another gives a detailed semantics and soundness proof for VCs that provide effective reasoning about recursively defined abstract predicates and abstraction functions [19]. The latter works have extensive pointers to related work.

References

1. A. Banerjee and D. A. Naumann. Local reasoning for global invariants, part II: Dynamic boundaries. *Journal of the ACM*, 60(3):19:1–19:73, 2013.
2. A. Banerjee, D. A. Naumann, and S. Rosenberg. Local reasoning for global invariants, part I: Region logic. *Journal of the ACM*, 60(3):18:1–18:56, 2013.
3. N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, 2004.
4. F. Bobot and J.-C. Filliâtre. Separation predicates: A taste of separation logic in first-order logic. In *ICFEM*, 2012.
5. A. Darvas and P. Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology (JOT)*, June 2006.
6. S. Heule, I. T. Kassios, P. Müller, and A. J. Summers. Verification condition generation for permission logics with abstract predicates and abstraction functions. In *ECOOP*, 2013.
7. C. A. R. Hoare. Proofs of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.
8. I. T. Kassios. The dynamic frames theory. *Formal Aspects of Comp.*, 23(3):267–288, 2011.
9. N. R. Krishnaswami, J. Aldrich, and L. Birkedal. Verifying event-driven programs using ramified frame properties. In *TLDI*, 2010.
10. K. R. M. Leino and P. Müller. Verification of equivalent-results methods. In *ESOP*, 2008.
11. A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable ADTs in Hoare type theory. In *ESOP*, 2007.
12. A. Nanevski, A. Banerjee, and D. Garg. Dependent type theory for verification of information flow and access control policies. *ACM Trans. Program. Lang. Syst.*, 35(2):6, 2013.
13. D. A. Naumann. Observational purity and encapsulation. *Theoretical Comput. Sci.*, 376(3):205–224, 2007.
14. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. *ACM Trans. Prog. Lang. Syst.*, 31(3):1–50, 2009.
15. M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *POPL*, 2005.
16. S. Rosenberg, A. Banerjee, and D. A. Naumann. Local reasoning and dynamic framing for the composite pattern and its clients. In *VSTTE*, 2010.
17. S. Rosenberg, A. Banerjee, and D. A. Naumann. Decision procedures for region logic. In *VMCAI*, 2012.
18. J. Smans, B. Jacobs, F. Piessens, and W. Schulte. Automatic verification of Java programs with dynamic frames. *Formal Aspects of Comp.*, 22(3-4):423–457, 2010.
19. A. J. Summers and S. Drossopoulou. A formal semantics for isorecursive and equirecursive state abstractions. In *ECOOP*, 2013.
20. H. Yang. Relational separation logic. *Theoretical Comput. Sci.*, 375(1-3):308–334, 2007.