A Logical Analysis of Framing for Specifications with Pure Method Calls

—- December 18, 2015 —-

Anindya Banerjee, IMDEA Software Institute, Madrid, Spain
and David A. Naumann, Stevens Institute of Technology, New Jersey, USA
and Mohammad Nikouei, Stevens Institute of Technology, New Jersey, USA

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 article 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 an extensional (that is, relational) interpretation of read effects. The approach is evaluated via machine-checked verification of examples. The developments in this article can (a) guide the implementations of linking as used in modular verifiers and (b) serve as basis for studying observationally pure methods and encapsulation in the setting of relational region logic.

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 article provides a theory, in the form of a logic, addressing these and related challenges.

Consider a class Cell, each instance of which holds an integer value, and these methods.

method get(): **int**
method set(v: **int**) **ensures** **self**.get() = v

Consider the following client code.

var c, d: Cell; c := new Cell; d := new Cell; c.set(5); d.set(6); **assert** c.get() = 5;

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

The second and third authors acknowledge partial support by US NSF award CNS-1228930.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

```
class Cell {
  private val: int;
  ghost foot: rgn;
  pure method get(): int
       requires I
       ensures self.get() = result ∧ I
       reads self.foot‘any
  method set(v: int)
       requires I
       ensures self.get() = v
       writes self.foot‘any //(in examples, read effects are omitted for impure methods)
}
```

Fig. 1. Example: Cell. Define $\mathsf{I} \mathrel{\widehat{=}} \forall x, y : Cell \cdot \ x \in x.foot \wedge (x = y \vee (x.foot \# y.foot \wedge x \notin y.foot))$.

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), that is, the predicate c.get() = 5, to be framed over the call d.set(6), yielding the assertion. But such specifications expose the internal representation. They 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 Figure 1, using ghost state[1] 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 [Kassios 2011], and in addition to explicit disjointness conditions it supports separation reasoning based on freshness. Our specification is based on a type **rgn** for 'region'. A ***region*** is a set of object references. The value of field **self**.foot is thus a set of references and **self**.foot‘val denotes the locations of the val fields of those objects. For example, in a state where the value of **self**.foot is the set $\{p, q\}$, then the set of locations denoted by **self**.foot‘val is the set $\{(p, \mathsf{val}), (q, \mathsf{val})\}$. In the sequel we write such pairs as p.val and q.val.

The keyword **any** abstracts over field names: the notation **self**.foot‘**any** denotes all fields of those objects. In a frame condition, it is the l-value —locations— that are denoted.[2]

In the rest of this article we will use the term "frame condition" loosely, to encompass both read and write effects.

The example illustrates another challenging issue: One method is used in the specification of others, in this case get in the postcondition of set. There are practical benefits to using programmed methods in specifications, and this can be justified provided that they are ***pure*** in the sense of having no effects other than reading. The idea is that such a method is computing a function and can be used as such in reasoning.

Pure methods are not only useful in pre- and post-conditions, they are also useful in frame conditions. For example, instead of the ghost field foot one might choose to define a region-valued method footpm.

Use of pure methods, especially ones in the program rather than part of the ambient mathematical theory, raises issues. One is how to reason using such specifications without inconsistency. For example, naïve use of the specification

method f(x: **int**): **int ensures result** = f(x)+1

[1]Mutable instrumentation used for reasoning but not affecting non-ghost state.

[2]Some other works based on dynamic frames express location sets in other ways (for example, Smans et al. [2010] and Bao et al. [2015]), but we follow Banerjee et al. [2013] in describing location sets in terms of fields (or data groups) and reference sets.

could lead to inconsistencies like $\mathsf{f}(0) = \mathsf{f}(0)+1$. How can we soundly reason about a pure method that is used in its own postcondition, for example, get in Figure 1, or in its own frame condition: for example, the read effect of footpm might be footpm()`**any** (making it 'self-framing'). Another issue is that specifications like those of get and set are abstract, in the sense that they are consistent with many interpretations of the function get (for example, get could return **self**.val+7 as long as set stores v−7). Client code should respect the abstraction, that is, be correct with respect to any interpretation. On the other hand, a given implementation (for example, implementing get and set by returning/setting **self**.val) is correct only if we interpret its specifications the right way.

The issues discussed so far have been addressed in prior work, especially in the context of verification-condition generation (***VC-gen***); see Section 9. 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. Their 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 (as reported in [Heule et al. 2013]).

This article provides a foundational account, by way of a conventional program logic that caters for SMT-provers by reasoning about framing using ghost state and standard first order logic, 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 (that is, collection of method specifications) with a client that relies on that interface.

1.1. Our approach

The approach we take is motivated by two additional challenges that we outline for motivation, but which are not directly addressed in this article. 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. 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. Hiding is useful for modularity but difficult to achieve soundly, so the state of the art is to rely on abstraction: a predicate whose definition is opaque in the interface can be defined internally to be the invariant [Parkinson and Bierman 2005; Nanevski et al. 2007; Krishnaswami et al. 2010].

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' 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 [Hoare 1972; O'Hearn et al. 2009] and for observational purity [Hoare 1972; Naumann 2007]. 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. Prior work developed ***region logic*** (RL) [Banerjee et al. 2013], a Hoare logic for sequential object-based programs, using standard first order logic (FOL) for assertions: the logic supports reasoning via explicit footprints captured in frame conditions. RL provides a frame rule for local reasoning, based on frame conditions of methods and a subsidiary judgment for framing of formulas. The frame rule expresses that a predicate continues to hold after a method call provided the locations on which it depends are disjoint (separated) from the locations that are writable according to the frame condition. In addition to ordinary frame conditions, the logic formalizes encapsulation boundaries for modules, by expressing separation between hidden state of a module and client visible state, so that hidden mod-

ule invariants are not falsified by client interaction. This idea is captured in a second-order frame rule for linking method implementations to clients, hiding invariants [Banerjee and Naumann 2013].

Prior work on RL does not allow pure methods in formulas. Nor does it track read effects of commands. However, such effects are needed for bodies of pure methods used in formulas. Read effects of commands are a relational property and one approach would be to use a general relational logic [Benton 2004; Yang 2007; Nanevski et al. 2013]. In ongoing work we adapt RL to a relational version, 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 [Hoare 1972; Naumann 2007]. However, read effects suffice to support pure methods in pre/post/frame specifications. Moreover, read effects raise some interesting problems in a relatively simple form, solutions of which are needed for a general relational logic. So in this article we study a "unary" logic with read effects, for both pure and impure methods, as a step towards the separate concerns of information hiding and observational purity.

This article builds on RLI [Banerjee et al. 2013] and RLII [Banerjee and Naumann 2013], extending RL with pure method calls in specifications and read effects in frame conditions of commands.

1.2. Outline and contributions

Section 2 introduces the programming language and specifications, as well as the judgment of correctness under hypotheses. The latter is written

$$\Phi; \psi \vdash C : P \rightsquigarrow Q\,[\varepsilon] \tag{1}$$

Please ignore ψ for now; that part of the judgment is discussed in due course. Judgment (1) 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. This conclusion is under hypothesis Φ, a list of method specifications. What's new in this article are read effects in ε and Φ, and pure methods used in $\Phi, P, C, Q, \varepsilon$ and specified in Φ.

Section 3 takes the first step towards defining semantics, by defining the semantics of expressions and formulas parameterized on the interpretation of pure methods. This is needed to dodge a potential circularity which we now explain.

There are two approaches to semantics of the judgment (1). The first goes by quantifying over all correct implementations of the procedures specified by Φ, that is, considering behavior of C when linked with any correct implementation. Such semantics appears in many guises in the literature.[3] Our transition semantics uses an environment for let-bound methods; calling a let-bound method results in execution of the body found in the method environment. So the first approach could be realized by considering all environments that also provide bodies for procedures in the hypothesis context. In this approach, the proof of soundness for the linking rule is almost immediate.

The second approach goes by using nondeterminacy to represent a single 'worst implementation' of each procedure, akin to the 'specification statement' used in axiomatic semantics. The second approach facilitates proof of second order frame rules, and is used for that purpose by O'Hearn et al. [2009] (for denotational semantics) and in RLII [Banerjee and Naumann 2013] (for transition semantics). Although first order framing is the focus of prior work on pure methods, second order framing is an essential part of modular accounts of encapsulation for heap structure.

Both approaches require, in effect, that C never calls a method of Φ outside its precondition. This becomes explicit in our proof of the linking rule. For pure methods, both approaches also raise some technical issues. The first issue is that for pure method calls in formulas, conventional semantics requires determinate values. Thus the second approach seems inappropriate for pure methods. The first approach, on the other hand, leads to a potential circularity: The approach says to interpret (1) by quantifying over correct implementations, but correctness is defined with respect to the meaning of specifications —and methods occur in the specifications.

[3] An example using denotational semantics is in the notion of "modular correctness" in [Leavens and Naumann 2015, Section 6], where the two approaches are compared.

The issues are both addressed by combining the first approach with the second appoach. We quantify not over implementations but over possible interpretations, that is, possible denotations of the implementations. For pure m, an interpretation $\varphi(m)$ applies to a state and an argument value, and returns a value. This can be used as its value in formulas and also as the value returned by a call in code. For impure m, $\varphi(m)$ applies to a state and an argument value, and can return a set of states; call to m takes a single step, nondeterministically choosing any of those states. The semantics of a correctness judgment (1) quantifies over all φ such that $\varphi(m)$ conforms to the specification $\Phi(m)$ for each m in $dom\,\Phi$. For pure methods, this is similar to an axiomatic semantics where $\varphi(m)$ is an uninterpreted function constrained by $\Phi(m)$.

To avoid circularity, Section 3 defines the semantics of expressions and formulas in terms of an arbitrary 'candidate interpretation' φ that is not required to satisfy any specifications. This serves to define, in Section 5, what it means for a candidate interpretation to satisfy its specifications, and thus to define the semantics of (1).

Section 4 formalizes an extensional semantics of read effects, adapting the standard relational notion of dependency. For deterministic programs and partial correctness this has a simple form sometimes called termination-insensitive noninterference. For C to read only locations X means the following. Consider execution of C from each of two states σ, σ' that agree on the values in those locations. If both executions terminate, the corresponding final states τ, τ' agree on any locations written or freshly allocated by C.

For impure methods, one might want to retain the second approach to semantics of (1). This requires possibly nondeterministic programs, and thus a possibilistic property for read effects, in $\forall\exists$ form: For all σ, σ' and all τ reached by C from σ, there exists τ' reached by C from σ' such that τ agrees with τ'. The conference version of this article [Banerjee and Naumann 2014] uses that semantics for read effects of commands and impure methods. Unfortunately it is broken. In checking the details of the proof of the linking rule, we were unable to complete the argument, because the $\forall\exists$ property would require moving back and forth between premiss and conclusion of the rule in an unsound way.

The solution in this article leverages quantification over interpretations, which is needed anyway to deal with pure methods. For impure m, an interpretation $\varphi(m)$ is required to be "quasi-deterministic", meaning that the only nondeterminacy is due to allocation. Owing to quasi-determinacy, we can avoid the existential quantifier and use the simpler semantics of read effects.

One might wonder why the allocator needs to be nondeterministic. The reason is that an allocator depends on hidden state that is not visible at the level of source code. Thus a faithful model of such an allocator must also work relative to arbitrary allocation behavior. This requires nondeterminacy, which we handle using bijective renamings ("refperms" in the sequel) in a standard way. As a consequence interpretations must cater to nondeterminacy as well.

Read effects have several applications including compiler optimizations [Benton et al. 2007] and observational purity. In this article we focus on their use for framing of formulas that call pure methods, for which purpose, we do not need read effects for bodies of impure methods. However, in this article we use the single judgment form (1) for all commands. This loses no generality, because there is a maximally permissive read effect. We also use a semantics of read effect that is slightly more general than needed to validate the Frame rule. These decisions lay groundwork for future work on the other applications.

Section 5 completes the semantics of the correctness judgment (1), suitably instantiating the interpretation of pure methods as motivated by the rule for linking. For C verified under hypothesis Φ that specifies pure method p called in C and/or used in the specification of C, linking discharges the hypothesis as explained below. Using the notion of correct interpretation, we also define what it means for a judgment to be healthy in the sense that its formulas and effect expressions do not depend on pure methods outside the preconditions of those methods. Healthy formulas satisfy the usual two-valued semantics of FOL, which justifies their use in SMT-based verifiers. Healthiness is formulated in terms of definedness predicates derived syntactically from formulas as in prior work on VC generation.

Section 6 defines two subsidiary judgments used in the proof rules. The subeffect judgment expresses that one effect is subsumed by another. The framing judgment expresses a bound on the footprint or read effect of a formula: its semantics is that the formula is not falsified by state updates that are outside its footprint —and that is the essence of framing. The footprint of a formula is derived from the read effects specified for the pure methods on which the formula may depend. What is important about the subsidiary judgments is their semantics, which is amenable to direct checking using an SMT prover. However, following RLI we also give proof rules for deriving the subsidiary judgments.

Section 6.3 explicates a notion we call 'framed reads'. In RLI we observed that the framing judgment for formulas naturally yields read effects that are self-framing. This property is not required for the proof rules, but in the literature it is shown to facilitate reasoning about separation for freshly allocated objects (Section 9). For read effects of commands, however, it turns out that this property is important.

Section 7 gives selected proof rules for the program correctness judgment, a worked example, and the main theorem: soundness of the rules. We also show necessity of framed reads for soundness of the sequence and iteration rules. The soundness proofs are intricate, especially for the linking rule, because it is proved directly in terms of small-step operational semantics. Small-step semantics is essential for the FOL-based form of dynamic frames and encapsulation used in RLII. The overall structure of the soundness proof for linking is close to the original proof in RLII, though details differ.

Section 8 evaluates the suitability of our approach for use in SMT-based tools. We report on the verification of the Cell (Figure 1) and Composite (Figure 2) implementations, together with their clients, using the Why3 verification system.[4] Section 9 discusses related work and Section 10 concludes. Appendix A provides some additional proofs.

To facilitate review, an index of definitions is included.

1.3. Explaining the interpretation ψ in the correctness judgment form (1)

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 Φ which includes a specification for m. But reasoning about B can use a particular interpretation for m. For example, a client of Cell should be correct with respect to any interpretation, including the one where get returns **self**.val+7. By contrast, the expected implementations of get and set are correct only with respect to the interpretation that returns **self**.val. 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, where pure methods are usually restricted to a simple form in order to ensure that the derived interpretation is not inconsistent [Darvas and Müller 2006; Smans et al. 2010]. Here we treat such interpretations semantically, in order to focus on their use rather than how to obtain them. And we impose no restrictions beyond purity of effect.

The role of ψ in (1) is to provide what we call a 'partial candidate interpretation' for zero or more of the pure methods in Φ. The semantics of (1) quantifies over all interpretations φ that satisfy Φ, as explained above, but which in addition agree with ψ where it is defined. That is, $\psi(m) = \varphi(m)$ if $\psi(m)$ is defined. This role is defined in the linking rule, which we sketch here in simplified form. For this discussion we elide effects, and we consider a single method m, so the rule looks like this:

$$\frac{\begin{array}{c}\Theta \equiv m : (x{:}T, \mathsf{res}{:}U) R \rightsquigarrow S \qquad \Phi, \Theta; \vdash C : P \rightsquigarrow Q \\ \Phi, \Theta; \psi \vdash B : R \rightsquigarrow S \qquad \mathit{dom}\, \psi = \{m\} \qquad \psi \models \Phi, \Theta \qquad \Phi; \psi \models R \wedge S \Rightarrow \mathsf{res} = m(x)\end{array}}{\Phi; \vdash \mathsf{let}\ m(x{:}T, \mathsf{res}{:}U) = B\ \mathsf{in}\ C : P \rightsquigarrow Q} \tag{2}$$

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 Φ, Θ which include the specification Θ of m

[4] Why3 is at `why3.lri.fr`. Our case studies are at `www.cs.stevens.edu/~naumann/pub/readRLWhy3.tar` .

as well as an ambient library Φ. The partial candidate is empty in the judgment for C, which means that C is correct with respect to any interpretation φ of all the procedures in Φ, Θ. The verification condition for B also has hypothesis Φ, 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 procedures in Φ, but only the fixed interpretation ψ of m.

The use of ψ is in the rule of consequence and other rules involving validity of assertions: the relevant formulas should be entailed by the specifications in Φ and also any available definitions (that is, the interpretation ψ) for pure methods.

Reasoning under inconsistent hypotheses leads to vacuous conclusions. If the hypotheses in (1) include an unsatisfiable specification for some pure method, there will exist no interpretations and so by definition the judgment is vacuously true. There is no need for a separate consistency check, though such a check could be helpful to flag errors early, because the linking rule requires there to be an interpretation for a pure method. That is the role of the conditions $\psi \models \Phi, \Theta$ and $\mathit{dom}\ \psi = \{m\}$ in (2). It says that under the assumptions Φ, the partial candidate ψ does satisfy its specification Θ. It cannot be written $\psi \models \Theta$, because specification Θ may refer to pure methods in Φ.

One way to go wrong is to choose for ψ an interpretation of m that is not consistent with behavior of B. In that case, the judgment for B will not be provable: the side condition $R \wedge S \Rightarrow \mathsf{res} = m(x)$ in (2) ensures that the value computed by B in variable res coincides with that given by interpretation ψ. (Variable res is used in the operational semantics as the return value from a method call.) On the other hand, a good way to establish $\psi \models \Phi, \Theta$ is to define ψ based on a correct implementation B —provided that it is correct in the sense of total correctness. This has been explored well in prior work, see Section 9.

1.4. Summary of contributions

— We provide a logic for object-based programs with dynamic allocation, featuring state dependent expressions in frame conditions that include both read and write effects for commands.
— We provide semantics for hypothetical judgments where specifications (pre, post, and frame) can refer to pure methods that can also be called in code. The key notion of partial interpretation accounts for axioms used in prior work on verification conditions for pure methods.
— We provide semantics for read effects of commands. A key finding is the necessity of 'framed reads' to enable state-dependent read effects to be composed in sequence just as write effects are.
— We provide proof rules, including a frame rule and rules for linking of pure and impure methods with their clients. Soundness is proved in detail, and directly in terms of a standard small step semantics.

Read effects of impure commands are not necessary for the frame rule, but are included as an important step towards observationally pure methods in specifications.

A preliminary version of this article appeared in a conference [Banerjee and Naumann 2014] but major changes have been made to the semantics and core definitions. The examples have now been fully verified using SMT provers and detailed proofs are provided.

2. PROGRAMS, SPECIFICATIONS, AND DEFINEDNESS

Figure 2 illustrates features of our programming and specification notations, by way of the Composite pattern, a well-known verification challenge problem [Robby et al 2008; Rosenberg et al. 2010; Bobot and Filliâtre 2012]. A Comp is the root of a tree, nodes of which are accessible to clients. Here is an example client:

```
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.Size() over the call b.add(c). The frame condition of add says it is allowed to write **self**.chrn, x.parent, and the size field of the ancestors of **self**. In method set (Figure 1) we use **'any** to abstract from field names, but here size is appropriate to make visible in the interface. That is the purpose of the specpublic annotation [Leavens et al.

```
class Comp {
  specpublic chrn: listOf(Comp); // list of children
  specpublic parent: Comp; //
  specpublic size: int := 1; // number of descendants

  method add(x: Comp) //add x to the list of children of self
    requires x.parent = null ∧ x ∉ self.anc() // (first conjunct implies x not null)
    ensures h.parent[x] = self
    writes self.chrn, x.parent, self.anc()‘size

  pure method getSize(): int
    reads size ensures result = size

  pure method anc(): rgn // get ancestors of self
    reads self.anc()‘parent
    ensures result = self.anc()
}
```

Fig. 2. Composite example, adapted from RLI.

2006] in Figure 2: chrn, size, and parent can appear in interface specifications but are private in the sense that client code can neither read nor write these fields. By contrast, we do not really want to expose field chrn. A good solution would be to use a data group [Leino et al. 2002] to abstract from it. However, the data group **any** is not appropriate in this case, because it would encompass **self**.parent and **self**.size which are not written by method add. The frame condition would be less precise using **self**.anc()‘**any**. In order to avoid formalizing data groups in this article, we simply make **self**.chrn specpublic. (See RLI for more discussion of this facet of information hiding.)

In order to reason using the frame rule (Figure 14), 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 for framing 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 add. The function $\cdot/\!/\cdot$ generates the separator formula and is defined by recursion on syntax. 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 Figure 11. In the example, we compute $\varepsilon /\!/ (\mathsf{rd}\, i, d, d.size)$, where ε is the write effect of add. The computed formula is the disjointness $\{\mathsf{d}\} \# \mathsf{b.anc()}$, which says the singleton region $\{d\}$ is disjoint from the set of ancestors; equivalently, $d \notin \mathsf{b.anc()}$. It needs to hold following the elided part of the example client above.

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. (See Lemma 6.6).

In this article we are concerned with pure methods that are implemented and used in code. In the case of anc, the implementation iteratively or recursively traverses parent pointers. The chosen specification avoids the use of descendants, in contrast to RLI or [Rosenberg et al. 2010].

2.1. Programs

Figure 3 gives the grammar of programs, revised from RLII to allow method calls in expressions. 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

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

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

Fig. 3. Programming language, highlighting additions to RLII. Take careful note of categories E, G, F.

short for procedure. Distinct classes have distinct field names.[5] For expository clarity methods have exactly one parameter. The letters T, U, V are used for types and B, C, D for commands.

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 $m(x)$ is its call. For expository purposes we omit the general form, which binds several pure and impure procedures together, allowing mutual recursion.

Typing contexts, ranged over by Γ, are finite maps, written in conventional form. For example, $x : T, m : (y{:}U), p : (y{:}U, \mathsf{res}{:}V)$ declares state variable x, impure method m, and pure method p. The judgment $\Gamma \vdash E : T$ means that E is ***syntactically well-formed*** (***swf***) and has type T. The judgment $\Gamma \vdash C$ means that command C is swf. Most of the typing rules are straightforward and omitted. For expressions, here are three rules of note:

$$\frac{\Gamma \vdash F : T \qquad \Gamma(m) = (x{:}T, \mathsf{res}{:}U)}{\Gamma \vdash m(F) : U} \qquad\qquad \frac{\Gamma \vdash E : K}{\Gamma \vdash \{E\} : \mathsf{rgn}}$$

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

Notice that the distinguished name res appears as a parameter in the declaration of a pure method, but there is no corresponding argument in the method call —rather, the call has a result and so can be used as an expression. See also the form $y := m(z)$ in Figure 4. To extend our formalization to pure methods of two arguments, the parameters would look like $(x{:}T, x'{:}T', \mathsf{res}{:}U)$ and a call would look like $m(v, v')$.

For commands, some typing rules are in Figure 4. The rules are designed to restrict assignments so that there are only two ways method calls occur in commands: $m(z)$ for impure m and $y := m(z)$ for pure m. This loses no generality but streamlines the formalization. (For example, it avoids the need to define small-step semantics of method calls in expressions.) An additional restriction on method bodies B, that they are let-free (as in RLII), simplifies the transition semantics.[6]

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

[5] Owing to the simple model of classes, the notation $G\text{`}\mathsf{any}$ can be defined as shorthand for $G\text{`}\overline{f}$ where $\overline{f}$ is the list of all field names. In a richer model with visibility restrictions, one would use a notion like data groups [Leino et al. 2002].

[6] Its consequence is that we are not fully modeling a module system, because any library in scope for a procedure is also in scope for its clients.

$$\frac{\Gamma(m) = (x{:}T)}{\Gamma, z{:}T \vdash m(z)} \qquad \frac{\Gamma(m) = (x{:}T, \mathsf{res}{:}U)}{\Gamma, y{:}U, z{:}T \vdash y := m(z)} \qquad \frac{\Gamma \vdash F : T \qquad F \text{ is call-free}}{\Gamma, x{:}T \vdash x := F}$$

$$\frac{(f{:}T) \in \mathit{Fields}(\Gamma x)}{\Gamma, y{:}T \vdash x.f := y} \qquad \frac{\Gamma, m : (x{:}T), x : T \vdash B \qquad \Gamma, m : (x{:}T) \vdash C \qquad B \text{ is let-free}}{\Gamma \vdash \mathsf{let}\ m(x{:}T) = B\ \mathsf{in}\ C}$$

$$\frac{\Gamma, m : (x{:}T, \mathsf{res}{:}U), x : T, \mathsf{res} : U \vdash B \qquad \Gamma, m : (x{:}T, \mathsf{res}{:}U) \vdash C \qquad B \text{ is let-free}}{\Gamma \vdash \mathsf{let}\ m(x{:}T, \mathsf{res}{:}U) = B\ \mathsf{in}\ C}$$

$$\frac{\Gamma \vdash C \qquad \Gamma \vdash D \qquad \Gamma \vdash E : \mathsf{int}}{\Gamma \vdash \mathsf{if}\ E\ \mathsf{then}\ C\ \mathsf{else}\ D} \qquad \frac{\Gamma \vdash C \qquad \Gamma \vdash E : \mathsf{int}}{\Gamma \vdash \mathsf{while}\ E\ \mathsf{do}\ C}$$

Fig. 4. Selected typing rules for commands.

in G. In case $f : \mathsf{rgn}$, the value of $G\text{`}f$ is the union of the f-values. Typing rules ensure there are no dangling references in reachable states.[7] Aside from allocation and dereference, the only operation on references is equality test.[8]

2.2. Specifications

The syntax of formulas is standard and unchanged from RLI (Section 4.2), except that now the expressions include method calls, for example the points-to predicate $x.f = p(y)$.

$$P \ ::= \ E = E \mid x.f = E \mid G \subseteq G \mid (\forall x : K \in G \cdot P) \mid (\forall x : \mathsf{int} \cdot P) \mid P \wedge P \mid P \vee P \mid \neg P$$

The formula $\forall x : K \in G \cdot P$ quantifies over all non-null references of type K in G. For disjointness of regions it is convenient to write $G \# H$ for $G \cap H \subseteq \{\mathsf{null}\}$. Note that for f of type region, there is no primitive $x.f = G$; but that can be desugared as $\{x\}\text{`}f \subseteq G \wedge G \subseteq \{x\}\text{`}f$.

We write $\Gamma \vdash P$ to express that P is swf in context Γ, and omit the straightforward typing rules.

Effects are given by

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

We abbreviate a compound effect $\mathsf{wr}\, x, \mathsf{rd}\, x$ as $\mathsf{rw}\, x$. Effects must be swf for the context Γ in which they occur: $\mathsf{rd}\, x$ and $\mathsf{wr}\, x$ are swf if $x \in \mathit{dom}(\Gamma)$; $\mathsf{rd}\, G\text{`}f$, and $\mathsf{wr}\, G\text{`}f$ are swf if $\Gamma \vdash G : \mathsf{rgn}$. By contrast with the typing rule for $G\text{`}f$ as an expression, we need no restriction on the type of f because the effect refers to its l-value. 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.

The function $\mathit{writes}(\varepsilon)$ discards all but the write effects, for example, $\mathit{writes}(\mathsf{wr}\, x, \mathsf{rd}\, y, \mathsf{wr}\, z) = \mathsf{wr}\, x, \mathsf{wr}\, z$. Similarly, $\mathit{reads}(\varepsilon)$ is the read effects in ε.

Specifications for impure methods take the form $(x{:}T)R \rightsquigarrow S\,[\eta]$ and for pure methods the form $(x{:}T, \mathsf{res}{:}U)R \rightsquigarrow S\,[\eta]$. The parameter, x, is passed by value. Here R is the precondition, S the postcondition, and η the effects. For these specifications to be swf in context Γ, η must not include $\mathsf{wr}\, x$ or $\mathsf{rd}\, x$. Moreover, R and η must be typable in $\Gamma, x{:}T$. Postcondition S must be typable in

[7] A fine point: To avoid complications in the substitutions used in some proof rules, we require that in any call $m(z)$ of an impure method, the variable z does not occur free in the relevant specifications. Similarly, in a call $y := m(z)$ of a pure method, we require that y, z do not occur free in the relevant specifications. This minor technicality is formalized in RLI/II by partitioning the set of variable names into so-called Locals and others; that way the restriction can be expressed without reference to specifications. For clarity in this article we simply ignore the issue.

[8] Dereference occurs in the command $x := y.f$ and $x.f := y$. It can also occur due to image expressions, in the form $x := F$ where $x : \mathsf{rgn}$. For example $x := \{y\}\text{`}f$ reads $y.f$ in states where $y \neq \mathsf{null}$.

$\Gamma, x{:}T$, for the impure form, or $\Gamma, x{:}T, \mathsf{res}{:}U$, for the pure form. Finally, for the pure method there must be no write effects in η.

It is standard in Hoare logic to disallow writes to the parameter, in order for postconditions to refer to initial parameter values. Although the body of a pure method will write res, the semantics is a return value, not an observable mutation of state. As a design choice, we require that the specification not include $\mathsf{rd}\, x$ (for parameter x), though it may include, for example, $\mathsf{rd}\,\{x\}\text{`}f$; and for an impure method the specification may include $\mathsf{wr}\,\{x\}\text{`}f$. In the semantics the argument value is handled specially. In the proof rules for method call, read effects are included for the argument expression. In the proof rules for linking, the premise for the method body does include $\mathsf{rd}\, x$.

For simplicity, we do not provide for specification-only variables (logical constants) in specifications. A sound formalization of specification-only variables has been worked out in RLII, and should carry over to the present setting. For the sake of simplicity, in this article we do not formalize specification-only variables.

Typically, the postcondition for a pure method m is simply that $\mathsf{res} = m(x)$, and for practical purposes that could be left implicit. In this foundational study, we found that it is only in linking that we need that form of postcondition. For flexibility, we formulate this as a side condition in the rule for linking pure methods. The condition is that the equation should follow from the pre and post conditions, that is, $R \wedge S \Rightarrow \mathsf{res} = m(x)$ should be valid. See PURELINK in Figure 14.

As mentioned in Section 1, for purposes of ordinary framing there is no need to track read effects of impure methods. However, they are needed for commands used in the body of a pure method. (They are also needed for reasoning about observational purity and data abstraction.) For simplicity the formalism in this article makes no distinction, that is, it tracks read effects for all methods and commands. This loses no generality, because in any context Γ there is a read effect that imposes no restriction: $\mathsf{rd}\, \mathit{vars}(\Gamma), \mathsf{rd}\,\mathsf{alloc}\text{`}\mathsf{any}$. The distinguished variable alloc is explained in the next Section.

A ***method context*** Φ 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 that is ***method-free***, that is, $\mathit{dom}\,\Gamma \subseteq \mathit{VarName}$. Moreover, specifications in Φ are allowed to refer to any of the pure methods in Φ; the specification of p may have calls to p in its postcondition and effect, or p and m may refer mutually to each other —subject to the restriction that calls in preconditions of pure methods must exhibit acyclic dependency. To make this restriction precise, we define a relation $\prec_\Phi$ on names of pure methods: $m' \prec_\Phi m$ iff m' occurs in the precondition of $\Phi(m)$, for m, m' specified in Φ as pure methods.

Definition 2.1 (*syntactically well-formed context*). A context Φ ***is swf in*** Γ provided that

— Γ is method-free
— the transitive closure, $\prec_\Phi^+$, is irreflexive, and
— each specification is swf in the context $\Gamma, \mathit{sigs}(\Phi)$.

We use comma to combine disjoint contexts, and *sigs* extracts the types of methods. For example, let Φ_0 be $m : (x{:}T)R \rightsquigarrow S\,[\eta],\ p : (y{:}V, \mathsf{res}{:}U)P \rightsquigarrow Q\,[\varepsilon]$. Then $\mathit{sigs}(\Phi_0)$ is $m : (x{:}T),\ p : (y{:}V, \mathsf{res}{:}U)$.

At this point we have all but one of the ingredients to define what it means for a correctness judgment (1) to be swf. What is missing is the partial candidate interpretation ψ, to be defined in Section 3.

2.3. Definedness

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 Figure 5. The idea is that in states where $\mathit{df}(P, \Phi)$ holds, evaluation of P does not depend on values of pure methods outside their preconditions.

We use the notation P^x_E for capture-avoiding syntactic substitution.

$$
\begin{array}{lcl}
\mathit{df}(x,\Phi) & = & \mathsf{true} \\
\mathit{df}(c,\Phi) & = & \mathsf{true} \\
\mathit{df}(\mathsf{null},\Phi) & = & \mathsf{true} \\
\mathit{df}(E_1 \oplus E_2,\Phi) & = & \mathit{df}(E_1,\Phi) \wedge \mathit{df}(E_2,\Phi) \text{ where } \oplus \text{ is in } \{=,+,\ldots\} \\
\mathit{df}(G\text{`}f,\Phi) & = & \mathit{df}(G,\Phi) \\
\mathit{df}(G_1 \otimes G_2,\Phi) & = & \mathit{df}(G_1,\Phi) \wedge \mathit{df}(G_2,\Phi) \text{ where } \otimes \text{ is in } \{\cup,\cap,\setminus\} \\
\mathit{df}(m(F),\Phi) & = & \mathit{df}(P^x_F,\Phi) \wedge \mathit{df}(F,\Phi) \wedge P^x_F \text{ where } \Phi(m) = (x:T,\mathsf{res}:U)P \rightsquigarrow Q\,[\varepsilon] \\
\\
\mathit{df}(E_1 = E_2,\Phi) & = & \mathit{df}(E_1,\Phi) \wedge \mathit{df}(E_2,\Phi) \\
\mathit{df}(x.f = E,\Phi) & = & x \neq \mathsf{null} \Rightarrow \mathit{df}(E,\Phi) \\
\mathit{df}(G_1 \subseteq G_2,\Phi) & = & \mathit{df}(G_1,\Phi) \wedge \mathit{df}(G_2,\Phi) \\
\mathit{df}(\forall x:\mathsf{int}\cdot P,\Phi) & = & \forall x:\mathsf{int}\cdot \mathit{df}(P,\Phi) \\
\mathit{df}(\forall x:K \in G\cdot P,\Phi) & = & \mathit{df}(G,\Phi) \wedge \forall x:K \in G\cdot \mathit{df}(P,\Phi) \\
\mathit{df}(P_1 \wedge P_2,\Phi) & = & \mathit{df}(P_1,\Phi) \wedge (P_1 \Rightarrow \mathit{df}(P_2,\Phi)) \\
\mathit{df}(P_1 \vee P_2,\Phi) & = & \mathit{df}(P_1,\Phi) \wedge (\neg P_1 \Rightarrow \mathit{df}(P_2,\Phi)) \\
\mathit{df}(\neg P,\Phi) & = & \mathit{df}(P,\Phi) \\
\\
\mathit{df}(\varepsilon,\Phi) & = & \mathit{df}(G_0,\Phi) \wedge \ldots \wedge \mathit{df}(G_n,\Phi) \\
 & & \text{where } G_0 \ldots G_n \text{ are the regions with } \mathsf{rd}\ G_i\text{`}f \text{ or } \mathsf{wr}\ G_i\text{`}f \text{ in } \varepsilon
\end{array}
$$

Fig. 5. Definedness formulas for expressions, formulas, and effects for swf method context Φ.

Although the clause for $\mathit{df}(m(F),\Phi)$ refers to a method specification that may refer to another pure method in its precondition, *df* is well-defined, owing to the requirement that $\prec^+_\Phi$ is irreflexive (and $\mathit{dom}\,\Phi$ is finite, so this is well founded). It is straightforward to show that if $\Gamma \vdash P$ then its definedness formula is well-formed in the same context, that is, $\Gamma \vdash \mathit{df}(P,\Phi)$.

For example, let Φ have specification[9] $\mathit{div6by} : (x{:}\mathsf{int},\mathsf{res}{:}\mathsf{int})x \neq 0 \rightsquigarrow \mathsf{res} = 6/x\,[\,]$. Let P be $y \neq 0 \wedge \mathit{div6by}(y) > 3$. Then $\mathit{df}(P,\Phi)$ is valid because

$$
\begin{array}{ll}
 & \mathit{df}(P,\Phi) \\
= & \mathit{df}(y \neq 0,\Phi) \wedge (y \neq 0 \Rightarrow \mathit{df}(\mathit{div6by}(y) > 3,\Phi)) \\
= & y \neq 0 \Rightarrow \mathit{df}(\mathit{div6by}(y) > 3,\Phi) \\
= & y \neq 0 \Rightarrow \mathit{df}(\mathit{div6by}(y),\Phi) \\
= & y \neq 0 \Rightarrow \mathit{df}(y \neq 0,\Phi) \wedge y \neq 0 \\
= & y \neq 0 \Rightarrow y \neq 0
\end{array}
$$

A definedness formula may itself include calls to pure methods. For example, if Φ also has $m : (x{:}\mathsf{int},\mathsf{res}{:}\mathsf{int})x \neq 0 \wedge \mathit{div6by}(x) > 5 \rightsquigarrow \mathit{true}\,[\,]$ then $\mathit{df}(m(y) = m(y),\Phi)$ has conjuncts including $y \neq 0$ and $\mathit{div6by}(y) > 5$.

An expression or formula is considered well-formed if its definedness formula is valid, in addition to it being swf (see Definition 5.5). To define validity, we need semantics.

3. SEMANTICS OF EXPRESSIONS, FORMULAS, AND PROGRAMS

Recall that we aim to interpret hypothetical correctness judgments by quantifying over all interpretations φ that conform to the hypotheses. To define what it means for $\varphi(m)$ to conform, 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 φ.

[9]Note that the method is allowed to read x and write res, but these effects are not supposed to be included in the specification so its effect is empty.

3.1. Preliminaries

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. We refrain from giving a complete concrete representation but instead describe the interface. The store is a type-respecting assignment of values to the variables in Γ. Note that if σ is a Γ-state then it is a $\mathit{vars}(\Gamma)$-state, where *vars* drops methods and retains only variables. Letters σ, τ, υ range over states.

We are generally concerned with contexts Γ that include the distinguished variable $\mathsf{alloc} : \mathsf{rgn}$. Updates of alloc are built into the program semantics so that alloc holds the set of all allocated references and does not contain *null*.

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, $[\sigma \,|\, x{:}\, v]$ to update variable x to value v, $[\sigma + x{:}\, v]$ to extend σ with additional variable x, and $[\![\Gamma]\!]$ for the set of Γ-states. In contexts where $\sigma(x)$ cannot be null, abbreviate $\sigma(\sigma(x).f)$ as $\sigma(x.f)$. 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, the faulting outcome $\lightning$ is used for runtime errors (null-dereference), and also to signal precondition violations (described later). These are not considered to be values or states. In RLII, $\lightning$ is written *fault* and a notational distinction is made between runtime error and precondition violation.

As basis for semantics of expressions and formulas, we define the notion of candidate Γ-interpretation, for a given typing context Γ. A candidate interpretation θ is, roughly, a mapping on the 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, $\theta(m)(\sigma, t)$ is a U-value or $\lightning$. If $\Gamma(m) = (x : T)$ then $\theta(m)$ is a (total) function such that for any T-value t and state σ, $\theta(m)(\sigma, t)$ is a set of states possibly including $\lightning$. These conditions are made precise below, by giving $\theta(m)$ a dependent type.

For states σ, τ, to express that τ is possible after σ we say τ ***succeeds*** σ, and write $\sigma \hookrightarrow \tau$, provided that $\sigma(\mathsf{alloc}) \subseteq \tau(\mathsf{alloc})$ and σ is compatible with τ in the sense that $\mathit{Type}(o, \sigma) = \mathit{Type}(o, \tau)$ for all $o \in \sigma(\mathsf{alloc})$.

Definition 3.1 (*candidate interpretation, partial candidate*).
For a typing context Γ, a ***candidate*** Γ***-interpretation*** θ is a mapping from the method names in Γ such that

(pure) if $\Gamma(m) = (x : T, \mathsf{res} : U)$ then $\theta(m)$ is a function of type
$(\sigma \in [\![\Gamma]\!]) \times [\![T]\!]\sigma \to ([\![U]\!]\sigma \cup \{\lightning\})$
(impure) if $\Gamma(m) = (x : T)$ then $\theta(m)$ is a function of type
$(\sigma \in [\![\Gamma]\!]) \times [\![T]\!]\sigma \to \mathbb{P}([\![\Gamma]\!] \cup \{\lightning\})$ such that $\sigma \hookrightarrow \tau$ for all σ, t, τ with $\tau \in \theta(m)(\sigma, t)$

For method context Φ that is swf in typing context Γ, a ***candidate*** Φ***-interpretation*** is just a candidate $(\Gamma, \mathit{sigs}(\Phi))$-interpretation.

A ***partial candidate*** Φ***-interpretation***, or just ***partial candidate***, is θ defined on some of the pure methods in Φ, satisfying condition (pure) for m on which θ is defined.

Note that a candidate Φ-interpretation is defined on methods in Φ and acts on Γ-states.

To avoid confusion, please note that a candidate Γ-interpretation is a mapping on the method names in $\mathit{dom}\,\Gamma$ which acts on Γ-states, which are $\mathit{vars}(\Gamma)$-states. The term "candidate Φ-interpretation" elides the typing context Γ for Φ; and in that case the interpretation is defined on methods in Φ acting on Γ-states since for Φ to be swf in Γ implies that Γ is method-free.

Definition 3.2 (*(swf) correctness judgment*). A ***correctness judgment*** takes the form $\Phi; \psi \vdash^{\Gamma} C : P \rightsquigarrow Q\,[\varepsilon]$ where ψ is a partial candidate for Φ. The judgment is swf iff Φ is swf in Γ and C, P, Q, ε are all swf in $\Gamma, \mathit{sigs}(\Phi)$. We often elide Γ.[10]

[10] In RLII, methods are allowed in Γ in this situation and in subsequent definitions like Definition 5.2. This is a technicality that facilitates proof of the program linking rule, the premise judgment of which may be applied to sub-traces involving let-bound methods and intermediate states with extended typing contexts. RLII is extremely careful about typing of such

$$
\begin{array}{lcl}
[\![E_1 \oplus E_2]\!]_\theta\sigma & = & \textit{let } v_1 = [\![E_1]\!]_\theta\sigma \textit{ in let } v_2 = [\![E_2]\!]_\theta\sigma \textit{ in } v_1 \oplus v_2 \\
[\![m(F)]\!]_\theta\sigma & = & \textit{let } v = [\![F]\!]_\theta\sigma \textit{ in } \theta(m)(\sigma, v) \\
[\![\{E\}]\!]_\theta\sigma & = & \textit{let } v = [\![E]\!]_\theta\sigma \textit{ in } \{v\} \\
[\![\varnothing]\!]_\theta\sigma & = & \varnothing \\
[\![G_1 \otimes G_2]\!]_\theta\sigma & = & \textit{let } X_1 = [\![G_1]\!]_\theta\sigma \textit{ in let } X_2 = [\![G_2]\!]_\theta\sigma \textit{ in } X_1 \otimes X_2 \\
[\![G\text{`}f]\!]_\theta\sigma & = & \textit{let } X = [\![G]\!]_\theta\sigma \textit{ in } \{\sigma(o.f) \mid o \in X \wedge o \neq \textit{null} \wedge \textit{Type}(o,\sigma) = \textit{DeclClass}(f)\} \\
 & & \text{if } f : K \text{ for some } K \\
 & = & \textit{let } X = [\![G]\!]_\theta\sigma \textit{ in } \bigcup\{\sigma(o.f) \mid o \in X \wedge o \neq \textit{null} \wedge \textit{Type}(o,\sigma) = \textit{DeclClass}(f)\} \\
 & & \text{if } f : \mathsf{rgn}
\end{array}
$$

Fig. 6. Semantics of selected program and region expressions, for state σ and candidate interpretation θ. The $\lightning$-strict let-binder is used: '*let* $v = X$ *in* Y' denotes $\lightning$ if X denotes $\lightning$. Here $\oplus$ is in $\{=, +, \ldots\}$ and $\otimes$ is in $\{\cup, \cap, \setminus\}$.

$$
\begin{array}{lcl}
[\![E_1 = E_2]\!]_\theta\sigma & = & \textit{let } v_1 = [\![E_1]\!]_\theta\sigma \textit{ in let } v_2 = [\![E_2]\!]_\theta\sigma \textit{ in if } v_1 = v_2 \textit{ then } true \textit{ else } false \\
[\![x.f = E]\!]_\theta\sigma & = & \textit{if } \sigma(x) = \textit{null} \textit{ then } false \textit{ else let } v = [\![E]\!]_\theta\sigma \textit{ in} \\
 & & \qquad \textit{if } \sigma(x.f) = v \textit{ then } true \textit{ else } false \\
[\![G_1 \subseteq G_2]\!]_\theta\sigma & = & \textit{let } X_1 = [\![G_1]\!]_\theta\sigma \textit{ in let } X_2 = [\![G_2]\!]_\theta\sigma \textit{ in if } X_1 \subseteq X_2 \textit{ then } true \textit{ else } false \\
[\![\Gamma \vdash \forall x : \mathsf{int} \cdot P]\!]_\theta\sigma & = & \lightning \text{ if } [\![\Gamma, x : \mathsf{int} \vdash P]\!]_\theta[\sigma + x\colon v] = \lightning \text{ for some } v \in \mathbb{Z} \\
 & = & true \text{ if } [\![\Gamma, x : \mathsf{int} \vdash P]\!]_\theta[\sigma + x\colon v] = true \text{ for all } v \in \mathbb{Z} \\
 & = & false \text{ otherwise} \\
[\![\Gamma \vdash \forall x : K \in G \cdot P]\!]_\theta\sigma & = & \lightning \text{ if } [\![G]\!]_\theta\sigma = \lightning \text{ or } [\![\Gamma, x : K \vdash P]\!]_\theta[\sigma + x\colon o] = \lightning \\
 & & \qquad \text{for some } o \text{ in } ([\![G]\!]_\theta\sigma) \setminus \{\textit{null}\} \text{ with } \textit{Type}(o,\sigma) = K \\
 & = & true \text{ if } [\![\Gamma, x : K \vdash P]\!]_\theta[\sigma + x\colon o] = true \\
 & & \qquad \text{for all } o \text{ in } ([\![G]\!]_\theta\sigma) \setminus \{\textit{null}\} \text{ with } \textit{Type}(o,\sigma) = K \\
 & = & false \text{ otherwise} \\
[\![P_1 \wedge P_2]\!]_\theta\sigma & = & \textit{let } b_1 = [\![P_1]\!]_\theta\sigma \textit{ in if } b_1 = false \textit{ then } false \textit{ else let } b_2 = [\![P_2]\!]_\theta\sigma \textit{ in } b_2 \\
[\![P_1 \vee P_2]\!]_\theta\sigma & = & \textit{let } b_1 = [\![P_1]\!]_\theta\sigma \textit{ in if } b_1 = true \textit{ then } true \textit{ else let } b_2 = [\![P_2]\!]_\theta\sigma \textit{ in } b_2 \\
[\![\neg P]\!]_\theta\sigma & = & \textit{let } b = [\![P]\!]_\theta\sigma \textit{ in } not\ b
\end{array}
$$

Fig. 7. Formulas: three-valued semantics, $[\![\Gamma \vdash P]\!]_\theta\sigma \in \{true, false, \lightning\}$ where σ ranges over Γ-states. Typing context is elided in most cases.

3.2. Semantics of expressions and formulas

The denotation of an expression F in candidate Γ-interpretation θ and state σ is written $[\![F]\!]_\theta\sigma$ and defined straightforwardly, see Figure 6. Of course it depends on θ only for pure methods. The second line in Figure 6 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, the 3-valued semantics of formulas is defined in Figure 7. The satisfaction relation $\models^\Gamma_\theta$ is defined by

$$\sigma \models^\Gamma_\theta P \quad \text{iff} \quad [\![P]\!]_\theta\sigma = \mathsf{true} \tag{3}$$

Later we show that when the definedness formulas hold, the usual 2-valued clauses hold for $\models^\Gamma_\theta$ (see Figure 8 and Lemma 5.3).

Strictly speaking, the semantic definitions go by induction on typing derivations. For clarity we elide typing contexts when they can be inferred from context. Here is why that is safe to do. The typing rules admit addition of extra variables, for example, if $\Gamma \vdash E : T$ and $x \notin \mathit{dom}\,\Gamma$ then $\Gamma, x : U \vdash E : T$. Furthermore, for $\Gamma, x\colon U$-state σ we have $[\![\Gamma, x : U \vdash E : T]\!]_\theta\sigma = [\![\Gamma \vdash E : T]\!]_\theta(\sigma\restriction x)$.

If $\varphi(m) = \theta(m)$ for all pure methods m, then

$$\sigma \models_\varphi P \text{ iff } \sigma \models_\theta P \quad \text{for all } \sigma, P \tag{4}$$

configurations, at the cost of extra generality of definitions and lemmas such as Definition 4.3 which may be applied to intermediate configurations. Here we gloss over this uninteresting fine point.

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

Fig. 8. Two-valued semantics of formulas.These clauses hold when $\sigma \models_\theta \mathit{df}(P, \Phi)$ (Lemma 5.3).

because the semantics of formulas does not depend on impure methods.

Implicit coercion. In the semantics of expressions and commands, candidate interpretations are applied to states with more variables than the ones in scope for method context Φ. For clarity we implicitly coerce the interpretations to such states, as follows. Suppose Φ is swf in Γ and θ is a candidate Φ-interpretation. So each $\theta(m)$ acts on Γ-states (that is, elements of $[\![\Gamma]\!]$). Suppose $\Gamma' \supseteq \Gamma$, declaring additional variables xs. If m is pure then for $\sigma \in [\![\Gamma']\!]$ define $\theta(m)(\sigma, v) = \theta(m)(\sigma \restriction xs, v)$. Here $\sigma \restriction xs$ has the same heap as σ but the store is defined only on $\mathit{dom}\,\Gamma$. This coercion is implicitly used in the semantic clause for $m(F)$ in Figure 6, and in the transition rules for $y := m(z)$ in Figure 9.

For impure m, which returns a state, the coercion is slightly more complicated. Let us write $\sigma + s$ for a Γ'-state where s is the valuation of the extra variables xs and σ is a Γ-state. Define

$$\theta(m)(\sigma + s, v) = \{\tau + s \mid \tau \in \theta(m)(\sigma, v)\} \cup \{\lightning \mid \lightning \in \theta(m)(\sigma, v)\}$$

The extra variables remain, unchanged, in the non-$\lightning$ case. This coercion is implicitly used in the transition rules for $m(z)$ in Figure 9.

Transition semantics. The transition relation depends on a candidate interpretation θ, for calls of pure and impure methods specified in the context. The transition relation $\overset{\theta}{\longmapsto}$ is defined in Figure 9, for arbitrary candidate interpretation θ.

The transition semantics is defined for configurations of the form $\langle C, \sigma, \mu \rangle$ where μ is a ***method environment***, that is, a mapping from method names to bodies of the form $(x{:}\, T \,.\, C)$ and $(x{:}\, T, \mathsf{res}{:}\, U \,.\, C)$. This caters for streamlined notation but requires that we disallow re-declaration of method names. The transition rules for let use the notation for extension of a mapping; this works because, by an invariant due to typing, the bound method cannot already be in the 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 $y := m(z)$ and there is some extra bookkeeping to assign the final value of res (or rather, a fresh instance thereof) to y. Note that in addition to the designated variable res, we use names res' as ordinary variables.

The control state in a configuration can be an ***extended command***, that is, possibly containing end-markers. The end-marker $\mathsf{elet}(m)$ causes m to be removed from the method environment and $\mathsf{ecall}(x, \ldots)$ causes some local variables to be removed from the state. In this article we gloss over fine points concerning extended commands, in particular typing of intermediate configurations, for which see RLII.

The transition semantics for pure method call $y := m(z)$ takes a step that assigns to y a value that could also be written as $[\![m(z)]\!]_\theta \sigma$ (see Figure 6). The transition semantics of a call $m(z)$, for impure m in θ, takes a single step to a final state (or $\lightning$) given by $\theta(m)$.

In proofs later, we rely on several straightforward properties of the transition semantics. For example, if $\Gamma \vdash C$ and θ is a candidate interpretation of Γ then for any σ and suitable μ, $\langle C, \sigma, \mu \rangle$ has at least one successor under $\overset{\theta}{\longmapsto}$ unless C is a call to an impure method. This can be checked

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

$$\frac{\theta(m)(\sigma,\sigma(z)) = v \qquad v \neq \lightning}{\langle y := m(z),\,\sigma,\,\mu\rangle \overset{\theta}{\longmapsto} \langle \mathsf{skip},\,[\sigma \,|\, y\colon v],\,\mu\rangle} \qquad \frac{\theta(m)(\sigma,\sigma(z)) = \lightning}{\langle y := m(z),\,\sigma,\,\mu\rangle \overset{\theta}{\longmapsto} \lightning}$$

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

$$\frac{\mu(m) = (x\colon T.C) \qquad x' \notin \mathit{Dom}(\sigma) \qquad C' = C^{x}_{x'}}{\langle m(z),\,\sigma,\,\mu\rangle \overset{\theta}{\longmapsto} \langle C';\mathsf{ecall}(x'),\,[\sigma + x'\colon \sigma(z)],\,\mu\rangle}$$

$$\frac{\mu(m) = (x\colon T, \mathsf{res}\colon U.C) \qquad x' \notin \mathit{Dom}(\sigma) \qquad \mathsf{res}' \notin \mathit{Dom}(\sigma) \qquad C' = C^{x,\mathsf{res}}_{x',\mathsf{res}'}}{\langle y := m(z),\,\sigma,\,\mu\rangle \overset{\theta}{\longmapsto} \langle C';\, y := \mathsf{res}';\mathsf{ecall}(x',\mathsf{res}'),\,[\sigma + x'\colon \sigma(z)],\,\mu\rangle}$$

$$\langle \mathsf{let}\ m(x\colon T) = B\ \mathsf{in}\ C,\,\sigma,\,\mu\rangle \overset{\theta}{\longmapsto} \langle C;\mathsf{elet}(m),\,\sigma,\,[\mu + m\colon (x\colon T.B)]\rangle$$

$$\langle \mathsf{let}\ m(x\colon T, \mathsf{res}\colon U) = B\ \mathsf{in}\ C,\,\sigma,\,\mu\rangle \overset{\theta}{\longmapsto} \langle C;\mathsf{elet}(m),\,\sigma,\,[\mu + m\colon (x\colon T, \mathsf{res}\colon U.B]\rangle$$

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

Fig. 9. Selected transition rules, using candidate interpretation θ. $\mathit{New}(\sigma, o, K, \overline{v})$ extends σ by adding o to alloc and by mapping o to a K-record with field values $\overline{v}$ and type K. Requires $o \notin \sigma(\mathsf{alloc})$.

by inspection of the transition rules. In case of an impure method call $m(z)$, it is possible that σ satisfies the precondition but the set $\theta(m)(\sigma,\sigma(z))$ is empty.

4. SEMANTICS OF EFFECTS

This section lays groundwork for defining, in Section 5, correct interpretations and the semantics of correctness judgments.

A ***location*** is either a variable name x or a heap location[11] comprised of a reference o and field name f. We write $o.f$ for such pairs. For any state σ, define the set of all locations by

$$\mathit{locations}(\sigma) = \mathit{Dom}(\sigma) \cup \{o.f \mid o \in \sigma(\mathsf{alloc}) \wedge f \in \mathit{Fields}(\mathit{Type}(o,\sigma))\}$$

Define $\mathit{rlocs}(\sigma,\theta,\varepsilon)$, the locations designated in σ by read effects of ε, by

$$\mathit{rlocs}(\sigma,\theta,\varepsilon) = \{x \mid \varepsilon \text{ contains } \mathsf{rd}\,x\} \cup \{o.f \mid \varepsilon \text{ contains } \mathsf{rd}\,G\text{`}f \text{ with } o \in [\![G]\!]_\theta\sigma\}$$

For write effects, define *wlocs mutatis mutandis*. Note that $\mathit{rlocs}(\sigma,\theta,\varepsilon) = \mathit{rlocs}(\sigma,\theta,\mathit{reads}(\varepsilon))$ and likewise for *wlocs*. Here θ is any candidate interpretation of the typing context, left implicit, for ε and σ.

Write effects constrain what locations may be updated, between an initial and a final state.

Definition 4.1 (*allows change*, $\sigma{\to}\tau \models_\theta \varepsilon$). Let effect ε be swf in Γ, let σ and τ be Γ-states and let θ be a candidate interpretation (for some Φ that is swf in Γ). Say ε ***allows change from*** σ ***to*** τ ***under*** θ, written $\sigma{\to}\tau \models_\theta \varepsilon$, iff $\sigma \hookrightarrow \tau$ and

[11] In RLI/II the term 'location' is used differently: it means heap location.

(a) for every y in $dom(\Gamma)$, either $\sigma(y) = \tau(y)$ or y is in $\mathit{wlocs}(\sigma, \theta, \varepsilon)$
(b) for every $o \in \sigma(\mathsf{alloc})$ and every f in $\mathit{Fields}(\mathit{Type}(o, \sigma))$, either $\sigma(o.f) = \tau(o.f)$ or $o.f$ is in $\mathit{wlocs}(\sigma, \theta, \varepsilon)$.

In (b), region expressions in ε are interpreted in the initial state because frame conditions need only report writes to fields of pre-existing objects and not freshly allocated objects.

Define $\mathit{written}(\sigma, \tau)$ to be $\{x \mid \sigma(x) \neq \tau(x)\} \cup \{o.f \mid \sigma(o.f) \neq \tau(o.f)\}$. Then $\sigma \to \tau \models_\theta \varepsilon$ iff $\sigma \hookrightarrow \tau$ and $\mathit{written}(\sigma, \tau) \subseteq \mathit{wlocs}(\sigma, \theta, \varepsilon)$.

Read effects and allowed dependence. 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 $\mathit{Ref} \setminus \{\mathit{null}\}$. 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

— $dom(\pi) \subseteq \sigma(\mathsf{alloc})$ and $\mathit{rng}\,(\pi) \subseteq \sigma'(\mathsf{alloc})$
— $\pi(p) = p'$ implies $\mathit{Type}(p, \sigma) = \mathit{Type}(p', \sigma')$ for all p, p'

Define $p \overset{\pi}{\sim} p'$ to mean $\pi(p) = p'$ or $p = \mathit{null} = p'$. Extend $\overset{\pi}{\sim}$ to a relation on integers by $i \overset{\pi}{\sim} j$ iff $i = j$. For reference sets X, Y, define $X \overset{\pi}{\sim} Y$ iff $Y = \pi(X)$, where $\pi(X)$ is the direct image of X. That is, π forms a bijection between X and Y. The image of refperm π on location set W is written $\pi(W)$ and defined for variables and heap locations by

$$x \in \pi(W) \quad \text{iff} \quad x \in W \qquad\qquad o.f \in \pi(W) \quad \text{iff} \quad (\pi^{-1}(o)).f \in W \tag{5}$$

Definition 4.2 (*agreement on a location set,* Lagree). For a set W of locations, define

$$\begin{aligned} \mathit{Lagree}(\sigma, \sigma', \pi, W) \quad \text{iff} \quad & \forall x \in W \cdot \sigma(x) \overset{\pi}{\sim} \sigma'(x) \wedge \\ & \forall (o.f) \in W \cdot o \in dom(\pi) \wedge \sigma(o.f) \overset{\pi}{\sim} \sigma'(\pi(o).f) \end{aligned}$$

Agreement on locations has a kind of symmetry:

$$\mathit{Lagree}(\sigma, \sigma', \pi, W) \text{ implies } \mathit{Lagree}(\sigma', \sigma, \pi^{-1}, \pi(W)) \text{ for all } \sigma, \sigma', \pi, W \tag{6}$$

A critical notion is agreement on read effects, for which symmetry turns out to be more delicate.

Definition 4.3 (*agreement on read effects*).

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

Agreement on some $\mathsf{rd}\, G\text{`}f$ modulo $\pi$ implies that $\sigma(G) \subseteq \mathit{dom}\,\pi$. However, it is important to note that agreement on $\mathsf{rd}\, G\text{`}f$ does not imply $[\![G]\!]_\theta \sigma \overset{\pi}{\sim} [\![G]\!]_\theta \sigma'$. For example, let G be the singleton $\{x\}$ of reference variable x and consider states where $\sigma(x) = o$, $\sigma'(x) = o'$, $\sigma(o.f) \overset{\pi}{\sim} \sigma'(\pi(o).f)$ but $\pi(o) \neq o'$.

A related issue is that agreement on read effects does not in general exhibit a symmetry property like (6). Consider these states, written in suggestive notation:

$$\sigma = [\mathsf{alloc}{:}\{o, p\},\ r{:}\{o\},\ o.f{:}3,\ p.f{:}4] \quad \sigma' = [\mathsf{alloc}{:}\{o, p\},\ r{:}\{o, p\},\ o.f{:}3,\ p.f{:}5] \tag{7}$$

We have $\mathit{Agree}(\sigma, \sigma', id, \mathsf{rd}\, r\text{`}f)$, with $id$ the identity relation on $\{o, p\}$, but we do not have $\mathit{Agree}(\sigma', \sigma, id, \mathsf{rd}\, r\text{`}f)$. We return to this in connection with framed reads in Section 6.3.

The semantics of read effects is a relational property of two initial states σ, σ' and two final states τ, τ'. Define

$$\begin{array}{l} \mathit{freshRefs}(\sigma,\tau) = \tau(\mathsf{alloc}) \setminus \sigma(\mathsf{alloc}) \\ \mathit{freshLocs}(\sigma,\tau) = \{p.f \,|\, p \in \mathit{freshRefs}(\sigma,\tau) \wedge f \in \mathit{Fields}(\mathit{Type}(p,\tau))\} \end{array}$$

Definition 4.4 (*allowed dependence,* $\sigma,\sigma' \Rightarrow \tau,\tau' \models_\theta \varepsilon$).
We say ε ***allows dependence*** from τ,τ' to σ,σ', and write $\sigma,\sigma' \Rightarrow \tau,\tau' \models_\theta \varepsilon$, iff for all π if $\mathit{Agree}(\sigma,\sigma',\varepsilon,\pi,\theta)$ then there is $\rho \supseteq \pi$ such that $\mathit{Lagree}(\tau,\tau',\rho,\mathit{freshLocs}(\sigma,\tau) \cup \mathit{written}(\sigma,\tau))$.

Here it is in a diagram.

$$\begin{array}{ccc} \sigma & \longmapsto & \tau \\ \Big| \; \mathit{Agree}(\sigma,\sigma',\varepsilon,\pi,\varphi) & & \Big| \; \mathit{Lagree}(\tau,\tau',\rho,\mathit{freshLocs}(\sigma,\tau) \cup \mathit{written}(\sigma,\tau)) \\ \sigma' & \longmapsto & \tau' \end{array}$$

5. SEMANTICS OF CORRECTNESS JUDGMENTS

This section completes the semantic definitions for program correctness judgments.

Recall that for syntactic substitution we use the notation P^x_E. In addition, for clarity we also use substitution of values, even references —though strictly speaking the syntax does not (and should not) include reference literals.[12] This is only done in certain contexts, for which we define the following abbreviations. If $\Gamma, x : T \vdash P$ and $\sigma \in [\![\Gamma]\!]$ and v is a value in $[\![T]\!]\sigma$, we write

$$\sigma \models^{\Gamma}_{\theta} P^x_v \qquad \text{to abbreviate} \qquad [\sigma + x{:}\, v] \models^{\Gamma, x:T}_{\theta} P$$

If ε contains neither $\mathsf{wr}\, x$ nor $\mathsf{rd}\, x$ then $\sigma \to \tau \models_\theta \varepsilon^x_v$ abbreviates $[\sigma + x{:}\, v] \to [\tau + x{:}\, v] \models_\theta \varepsilon$. Finally, $\mathit{wlocs}(\sigma,\theta,\varepsilon^x_v)$ abbreviates $\mathit{wlocs}([\sigma + x{:}\, v],\theta,\varepsilon)$.

A correct candidate interpretation is one that satisfies its specifications.

Definition 5.1 (*context interpretation*). Let Φ be swf in Γ and let φ be a candidate Φ-interpretation. Say φ is a Φ-***interpretation*** iff for each m in $\mathit{dom}\,\Phi$

— If m has specification $(x{:}\,T, \mathsf{res}{:}\,U)P \rightsquigarrow Q\,[\varepsilon]$, then for any $\sigma \in [\![\Gamma]\!]$ and $v \in [\![T]\!]\sigma$,
  (a) $\varphi(m)(\sigma, v) = \lightning$ iff $\sigma \not\models_\varphi P^x_v$
  (b) if $\sigma \models_\varphi P^x_v$ then letting $w = \varphi(m)(\sigma, v)$ we have $\sigma \models_\varphi Q^{x,\mathsf{res}}_{v,w}$
  (c) if $\sigma \models_\varphi P^x_v$ then letting $w = \varphi(m)(\sigma, v)$ and $w' = \varphi(m)(\sigma', v')$ we have the following: for any $\sigma' \in [\![\Gamma]\!]$, $v' \in [\![T]\!]\sigma'$ with $\sigma' \models_\varphi P^x_{v'}$, and any refperm π from σ to σ', if $v \overset{\pi}{\sim} v'$ and $\mathit{Agree}([\sigma + x : v], [\sigma' + x : v'], \varepsilon, \pi, \varphi)$ then $w \overset{\pi}{\sim} w'$

— If m has specification $(x{:}\,T)P \rightsquigarrow Q\,[\varepsilon]$ then for any $\sigma \in [\![\Gamma]\!]$ and $v \in [\![T]\!]\sigma$,
  (d) $\lightning \in \varphi(m)(\sigma, v)$ iff $\sigma \not\models_\varphi P^x_v$
  (e) For all $\tau \in \varphi(m)(\sigma, v)$, if $\sigma \models_\varphi P^x_v$ then $\tau \models_\varphi Q^x_v$ and $\sigma \to \tau \models_\varphi \varepsilon^x_v$
  (f) For all $\tau, \sigma', \tau', v', \pi$, if
    — $v \overset{\pi}{\sim} v'$,
    — $\sigma \models_\varphi P^x_v$,
    — $\sigma' \models_\varphi P^x_{v'}$,
    — $\tau \in \varphi(m)(\sigma, v)$,
    — $\tau' \in \varphi(m)(\sigma', v')$,
    — $\mathit{Agree}([\sigma + x : v], [\sigma' + x : v'], \varepsilon, \pi, \varphi)$
    then there is $\rho \supseteq \pi$ with $\mathit{Lagree}(\tau, \tau', \rho, \mathit{freshLocs}(\sigma, \tau) \cup \mathit{written}(\sigma, \tau))$.

[12] Not only do (non-null) reference literals not make sense in a Java-like language, we must not allow reference literals in formulas, as otherwise we would lose the agreement lemmas or else need to include literal values in read effects.

Apropos (a) and (d), the negated satisfaction is equivalent to saying $[\![P]\!]_\varphi[\sigma + x{:}\,v]$ is $\lightning$ or false, as per (3). Apropos (c), (e), and (f), recall that a swf specification does not include $\mathsf{wr}\,x$ or $\mathsf{rd}\,x$ for its parameter x, so it is safe to use the substitution abbreviations.[13]

Note that the definition makes sense even if pure m occurs in its own specification, or in the specification of some other pure m' in Φ for which the specification refers to m.

To define validity of correctness judgments, the last ingredient is to connect the partial candidate ψ in the judgment with the context interpretations over which the judgment quantifies, as sketched in Section 1. We say φ ***extends*** ψ if $\varphi(m) = \psi(m)$ for every m on which ψ is defined. Representing maps by their graphs, this amounts to $\psi \subseteq \varphi$.

Definition 5.2 (*valid judgment*). A swf correctness judgment $\Phi; \psi \vdash^\Gamma C : P \rightsquigarrow Q\,[\varepsilon]$ is ***valid*** iff the following conditions hold for all Φ-interpretations φ such that φ extends ψ, and all states σ such that $\sigma \models_\varphi^{\Gamma, \mathit{sigs}(\Phi)} P$.

(Safety) It is not the case that $\langle C,\, \sigma,\, \_\rangle \overset{\varphi}{\longmapsto}{}^* \lightning$.

(Post) $\tau \models_\varphi Q$ for every τ with $\langle C,\, \sigma,\, \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip},\, \tau,\, \_\rangle$

(Write Effect) $\sigma \rightarrow \tau \models_\varphi \varepsilon$ for every τ with $\langle C,\, \sigma,\, \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip},\, \tau,\, \_\rangle$

(Read Effect) For all $\tau, \sigma', \tau', \pi$, if $\langle C,\, \sigma,\, \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip},\, \tau,\, \_\rangle$ and $\langle C,\, \sigma',\, \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip},\, \tau',\, \_\rangle$ and $\sigma' \models_\varphi P$ then $\sigma, \sigma' \Rightarrow \tau, \tau' \models_\theta \varepsilon$.

Because the judgment is swf, Γ is method-free hence the only relevant method environment is the empty one, written $\_$. In (Post) and (Read Effect) we omit $\Gamma, \mathit{sigs}(\Phi)$ from $\models_\varphi$. In (Read Effect), note that the final states should agree on any location that is written, and on any freshly allocated locations. The significance of this is evident, for example, in the soundness proofs for sequence and while.

It is possible for the specification of a pure method to be unsatisfiable, and thus for there to be no Φ-interpretations. If the partial candidate interpretation ψ is defined for some m but does not satisfy $\Phi(m)$ then there are no Φ-interpretations that extend ψ. In both cases, the judgment is semantically valid, though of no use because the vacuous hypothesis cannot be discharged by linking with any method implementations. For practical purposes one would want to check early for non-satisfying $\psi(m)$, and for unsatisfiable specifications.

For impure methods, in case σ satisfies the precondition and no state satisfies the postcondition the code can diverge. As ours is a partial correctness logic, such an implementation can be correctly linked. In this situation a candidate interpretation $\varphi(m)$ can have $\varphi(m)(\sigma, v) = \varnothing$.

Healthiness and well-formed correctness judgments. The definitions up to this point apply even if pure methods are called outside their precondition. However, a specification or correctness judgment that involves a pure method called outside its precondition is unlikely to capture an intuitively meaningful requirement. For understandable proof rules, and to stay within FOL for assertions, we will disallow such specifications and correctness judgments. That is the purpose of Definition 5.5 to follow.

LEMMA 5.3 (TWO-VALUED SEMANTICS OF FORMULAS). (a) If φ is a Φ-interpretation and $\sigma \models_\varphi \mathit{df}(P, \Phi)$ then $[\![P]\!]_\varphi \sigma$ is not $\lightning$. (b) For any σ and any Φ-interpretation φ, if $\sigma \models_\varphi \mathit{df}(P, \Phi)$ then the condition $\sigma \models_\varphi P$ satisfies the usual defining clause, see Figure 8.

PROOF. For part (a), a similar lemma for expressions is needed as follows.

Lemma A. If φ is a Φ-interpretation and $\sigma \models_\varphi \mathit{df}(F, \Phi)$ then $[\![F]\!]_\varphi \sigma$ is not $\lightning$.

[13] Enabling these abbreviations is the main reason we decided to omit $\mathsf{wr}\,x$ and $\mathsf{rd}\,x$ from specifications and instead add the effects explicitly in the method call and method linking proof rules. Note that we do not use a substitution abbreviation for agreement, which involves two parallel states.

The proof of lemma A goes induction on F, using the definitions in Figures 5 and 6. For the base cases x, c, null, $\varnothing$, we have $df(F,\Phi) = \mathsf{true}$. And, $[\![x]\!]_\varphi\sigma = \sigma(x)$, $[\![c]\!]_\varphi\sigma = c$, $[\![\mathsf{null}]\!]_\varphi\sigma = \mathsf{null}$ and $[\![\varnothing]\!]_\varphi\sigma = \varnothing$ (so none of them is $\lightning$).

For case $m(F)$, we have

$$df(m(F),\Phi) = df(P^x_F,\Phi) \wedge df(F,\Phi) \wedge P^x_F,$$

where $\Phi(m) = (x : T, \mathsf{res} : U)P \rightsquigarrow Q\,[\varepsilon]$. Thus $\sigma \models_\varphi df(F,\Phi)$ and $\sigma \models_\varphi P^x_F$. By induction hypothesis, $[\![F]\!]_\varphi\sigma$ is not $\lightning$. Let $v = [\![F]\!]_\varphi\sigma$. Then, we have $[\![m(F)]\!]_\varphi\sigma = \varphi(m)(\sigma, v)$ so by Definition 5.1 (a), it is not $\lightning$.

The other cases in the proof of the Lemma A are straightforward.

Having proved Lemma A, we proceed to show part (a), i.e., $\sigma \models_\varphi df(P,\Phi)$ implies $[\![P]\!]_\varphi\sigma \neq \lightning$. Go by induction on P, using the definitions in Figures 5 and 7.

For base case $x.f = E$, we have

$$df(x.f = E,\Phi) = (x \neq \mathsf{null} \Rightarrow df(E,\Phi)).$$

Thus $\sigma(x) \neq \mathsf{null} \Rightarrow \sigma \models_\varphi df(E,\Phi)$. By Lemma A, if $\sigma(x) \neq \mathsf{null}$, then $[\![E]\!]_\varphi\sigma$ is not $\lightning$. On the other hand, by semantics (Figure 7), if $\sigma(x) = \mathsf{null}$ then $[\![x.f = E]\!]_\varphi\sigma$ is false. Otherwise, let $v = [\![E]\!]_\varphi\sigma$ then $[\![x.f = E]\!]_\varphi\sigma$ is true or false according to whether $\sigma(x.f) = v$, and not $\lightning$.

Case $\Gamma \vdash \forall x : K \in G \cdot P$. By Figure 5, we have

$$df(\forall x : K \in G \cdot P,\Phi) = df(G,\Phi) \wedge (\forall x : K \in G \cdot df(P,\Phi)).$$

Thus $\sigma \models_\varphi df(G,\Phi)$. By Lemma A, $[\![G]\!]_\varphi\sigma$ is not $\lightning$. Also, we have $[\sigma + x\colon o] \models_\varphi df(P,\Phi)$, for all $o \in ([\![G]\!]_\varphi\sigma) \setminus \{\mathsf{null}\}$ with $\mathit{Type}(o,\sigma) = K$. By induction hypothesis, $[\![P]\!]_\varphi[\sigma + x\colon o]$ is not $\lightning$, for all $o \in ([\![G]\!]_\varphi\sigma) \setminus \{\mathsf{null}\}$ with $\mathit{Type}(o,\sigma) = K$. So by semantics Figure 7, $[\![\Gamma \vdash \forall x : K \in G \cdot P]\!]_\varphi\sigma$ is not $\lightning$.

Case $P_1 \wedge P_2$. By Figure 5, we have

$$df(P_1 \wedge P_2,\Phi) = df(P_1,\Phi) \wedge (P_1 \Rightarrow df(P_2,\Phi)).$$

Thus $\sigma \models_\varphi df(P_1,\Phi)$ and $\sigma \models_\varphi (P_1 \Rightarrow df(P_2,\Phi))$. By induction hypothesis on P_1, $[\![P_1]\!]_\varphi\sigma \neq \lightning$. If $[\![P_1]\!]_\varphi\sigma$ is false, then $[\![P_1 \wedge P_2]\!]_\varphi\sigma = \mathit{false}$, and thus not $\lightning$. If $[\![P_1]\!]_\varphi\sigma$ is true, then $\sigma \models_\varphi P_1$. Hence $\sigma \models_\varphi df(P_2,\Phi)$ and by the induction hypothesis on P_2, $[\![P_2]\!]_\varphi\sigma \neq \lightning$. Thus $[\![P_1 \wedge P_2]\!]_\varphi\sigma = [\![P_2]\!]_\varphi\sigma$ and hence not $\lightning$.

For part (b) of the lemma, a straightforward case analysis shows that when the definedness condition holds, the clause in Figure 8 is equivalent to the definition in Figure 7. □

Definition 5.4 ($\Phi;\psi$-*valid formula*). Let Γ be a typing context and let Φ be a specification context that is swf in Γ. Let ψ be a partial candidate for Φ. Let P be a formula that is swf in $\Gamma, \mathit{sigs}(\Phi)$. Then P is $\Phi;\psi$-***valid***, written $\Phi;\psi \models P$, if and only if $\sigma \models_\varphi P$ for all states σ and all Φ-interpretations φ that extend ψ.

Note that φ includes impure methods if Φ does, but they have no bearing on validity of the formula, cf. (4).

The term "$\Phi;\psi$-valid" elides dependency on Γ, which should be provided by context. If Φ contains an unsatisfiable specification, or $\psi(m)$ does not satisfy $\Phi(m)$ for some m, then every P is $\Phi;\psi$-valid, as there are no Φ-interpretations.

In a VCgen setting, the proof obligations include definedness conditions on the specifications. In the logic, that is manifest by the stipulation that proof rules are only instantiated with healthy judgments.

Definition 5.5 (*healthy, well-formed*). Let Γ and Φ satisfy the conditions of Definition 5.4, and ψ be a partial candidate for Φ. A formula P that is swf is ***healthy for*** ψ iff $df(P,\Phi)$ is $\Phi;\psi$-valid. A swf impure method specification $(x\colon T)P \rightsquigarrow Q\,[\eta]$ is ***healthy*** (with respect to Γ,Φ,ψ) iff the three formulas $df(P,\Phi)$, $df(Q,\Phi)$, and $P \Rightarrow df(\eta,\Phi)$ are $\Phi;\psi$-valid. A swf pure method specification

$(x{:}T, \mathsf{res}{:}U)P \rightsquigarrow Q\,[\eta]$ is ***healthy*** if $df(P,\Phi)$, $P \Rightarrow df(Q,\Phi)$, and $P \Rightarrow df(\eta,\Phi)$ are $\Phi;\psi$-valid. A swf correctness judgment $\Phi;\psi \vdash^{\Gamma} C : P \rightsquigarrow Q\,[\eta]$ is ***healthy*** iff the specifications in Φ are healthy for ψ and the three formulas $df(P,\Phi)$, $df(Q,\Phi)$, and $P \Rightarrow df(\eta,\Phi)$ are $\Phi;\psi$-valid.

The term ***well-formed*** means swf and healthy.

The definitions to this point are intricate but elementary. 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. SUBEFFECTS, FRAMING OF FORMULAS, AND SEPARATOR FORMULAS

The proof system for correctness judgments relies on several concepts which are covered in this section. Section 6.1 is about the subeffect judgment, which allows to weaken an effect or change the way it is expressed. Section 6.2 covers several notions. The framing judgment delimits the read effect of a formula. Separator formulas play a key role in the FRAME rule for programs. Separator formulas are also used in the notion of immunity which enables something like the FRAME rule for effects, and appears in the proof rules for command sequences and loops. Immunity, separator formulas, and the framing judgment are adapted from RLI. Section 6.3 defines 'framed reads', a notion which plays a role similar to immunity but for read effects.

Concerning the subeffect and framing judgments, we start with their semantics, which is amenable to direct checking using an SMT prover. Then, following RLI, proof rules are given for deriving the judgments syntactically. In turn, these rules refer to context-validity of first order formulas, cf. Definition 5.4.

6.1. Subeffect

For an effect of the form $\mathsf{wr}\,G\text{`}f$ there is the possibility of more liberal effect $\mathsf{wr}\,H\text{`}f$ in case $G \subseteq H$. Since region expressions are state-dependent and context-interpretation dependent, so are inclusions like the above.

For method context Φ and partial pure Φ-interpretation ψ we define the ***subeffect judgment*** to have the form

$$P;\Phi;\psi \vdash \varepsilon_1 \leq \varepsilon_2$$

Such a judgment is ***healthy*** iff $df(P,\Phi)$, $P \Rightarrow df(\varepsilon_1,\Phi)$ and $P \Rightarrow df(\varepsilon_2,\Phi)$ are all $\Phi;\psi$-valid. A healthy subeffect judgment is intended to mean that under precondition P, the 'bigger' effect ε_2 is more permissive than ε_1. Impure methods may be present in Φ but those are irrelevant here.

Definition 6.1 (*valid subeffect*). A well-formed subeffect judgment is ***valid***, written $P;\Phi;\psi \models \varepsilon \leq \eta$, if for all Φ-interpretations φ that extend ψ, and all states σ with $\sigma \models_\varphi P$, we have $\mathit{rlocs}(\sigma,\varphi,\varepsilon) \subseteq \mathit{rlocs}(\sigma,\varphi,\eta)$ and $\mathit{wlocs}(\sigma,\varphi,\varepsilon) \subseteq \mathit{wlocs}(\sigma,\varphi,\eta)$.

LEMMA 6.2 (SUBEFFECTS ALLOW CHANGE AND DEPENDENCY). If $P;\Phi;\psi \models \varepsilon \leq \eta$ then the following hold for all Φ-interpretations φ that extend ψ and states $\sigma,\sigma',\tau,\tau'$ such that $\sigma \models_\varphi P$ and $\sigma' \models_\varphi P$:

(allowed change) $\sigma{\to}\tau \models_\varphi \varepsilon$ implies $\sigma{\to}\tau \models_\varphi \eta$.
(agreement) $\mathit{Agree}(\sigma,\sigma',\eta,\pi,\varphi)$ implies $\mathit{Agree}(\sigma,\sigma',\varepsilon,\pi,\varphi)$
(allowed dependency) $\sigma,\sigma'{\Rightarrow}\tau,\tau' \models_\varphi \eta$ implies $\sigma,\sigma'{\Rightarrow}\tau,\tau' \models_\varphi \varepsilon$

The first two parts are immediate from definitions. To show the third part, (allowed dependency), suppose $\sigma,\sigma'{\Rightarrow}\tau,\tau' \models_\varphi \eta$ and consider any $\sigma,\sigma',\tau,\tau'$ such that $\mathit{Agree}(\sigma,\sigma',\eta,\pi,\varphi)$. By agreement we have $\mathit{Agree}(\sigma,\sigma',\varepsilon,\pi,\varphi)$ so we can use $\sigma,\sigma'{\Rightarrow}\tau,\tau' \models_\varphi \varepsilon$ to get the conclusion that $\mathit{Lagree}(\tau,\tau',\mathit{freshLocs}(\sigma,\tau) \cup \mathit{written}(\sigma,\tau))$.

In Figure 10 we provide syntactic rules for subeffecting for well-formed subeffect judgments.

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

$$true; \Phi; \psi \vdash \mathsf{wr}\, G_1\text{`}f, G_2\text{`}f \lesseqgtr \mathsf{wr}\,(G_1 \cup G_2)\text{`}f$$

$$true; \Phi; \psi \vdash \mathsf{rd}\, G_1\text{`}f, G_2\text{`}f \lesseqgtr \mathsf{rd}\,(G_1 \cup G_2)\text{`}f \qquad \frac{P; \Phi; \psi \vdash \varepsilon_1 \leq \varepsilon_2 \qquad P; \Phi; \psi \vdash \varepsilon_2 \leq \varepsilon_3}{P; \Phi; \psi \vdash \varepsilon_1 \leq \varepsilon_3}$$

$$true; \Phi; \psi \vdash \varepsilon \leq \varepsilon, \eta \qquad true; \Phi; \psi \vdash \varepsilon \leq \varepsilon \qquad true; \Phi; \psi \vdash \varepsilon, \eta \leq \eta, \varepsilon$$

$$\frac{P; \Phi; \psi \vdash \varepsilon_1 \leq \varepsilon_2}{P; \Phi; \psi \vdash \varepsilon_1, \eta \leq \varepsilon_2, \eta} \qquad \frac{\Phi; \psi \models P' \Rightarrow P \qquad P; \Phi; \psi \vdash \varepsilon \leq \eta}{P'; \Phi; \psi \vdash \varepsilon \leq \eta}$$

Fig. 10. Selected rules for well-formed subeffect judgments. We write $\lesseqgtr$ to abbreviate two inclusion rules and elide Φ and ψ in the rules except the one where they are used.

LEMMA 6.3 (SUBEFFECT SOUNDNESS). If $P; \Phi; \psi \vdash \varepsilon \leq \eta$ is derivable by rules in Figure 10 then the judgment is valid.

The proof goes by showing that each rule is sound, and is straightforward.

6.2. Framing and separator formulas

The ***framing judgment*** has the form

$$P; \Phi; \psi \vdash^{\Gamma} \eta\ \mathsf{frm}\ Q$$

and is swf under evident conditions. It means that in P-states, the formula Q depends only on the part of the state delimited by η. The judgment is ***healthy*** iff the formulas $df(P, \Phi)$, $P \Rightarrow df(\eta, \Phi)$, and $P \Rightarrow df(Q, \Phi)$ are $\Phi; \psi$-valid. Often we elide the context Γ in a framing judgment, as it is usually clear from context.

Definition 6.4 (*frame validity*). A well-formed framing judgment is ***valid***, written $P; \Phi; \psi \models^{\Gamma} \eta\ \mathsf{frm}\ Q$, iff for all Φ-interpretations φ that extend ψ, all Γ-states σ, τ and refperms π, if $Agree(\sigma, \sigma', \eta, \pi, \varphi)$, and $\sigma \models^{\Gamma}_{\varphi} P \wedge Q$, then $\tau \models^{\Gamma}_{\varphi} Q$.

In this article, it would suffice to define frame validity in terms of the identity refperm on $\sigma(\mathsf{alloc})$, as that suffices for its use in the frame rule. The extra generality has little cost and is of interest in the setting of relational logic.

A verifier can check framing judgments in terms of the validity property (see 8 and [Rosenberg et al. 2010]), but our logic includes rules to derive framing judgments. A basic rule allows to infer, for atomic formula P, the judgment $true; \Phi; \psi \vdash ftpt(P, \Phi)\ \mathsf{frm}\ P$ concerning a precise footprint computed by function *ftpt* which is defined in Figure 12. In the Figure, notation ε^x_F means syntactic substitution.

LEMMA 6.5 (FOOTPRINT AGREEMENT). For any states, σ, σ', for any expression F, for any refperm π from σ to σ', for any method context Φ, and for any Φ-interpretation φ, suppose $df(F, \Phi)$ is valid and $Agree(\sigma, \sigma', ftpt(F, \Phi), \pi, \varphi)$. Then $[\![F]\!]_\varphi \sigma \overset{\pi}{\sim} [\![F]\!]_\varphi \sigma'$.

Separator formulas and immunity. The point of establishing $P; \Phi; \psi \vdash \eta\ \mathsf{frm}\ Q$ is that code that writes outside η cannot falsify Q. This is expressed in the frame rule (Figure 14) 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. There is a related notion for effects, called immunity, which

$$
\begin{array}{lcl}
\mathsf{rd}\, G\text{`}f \mathbin{/\!\cdot\!/} \mathsf{wr}\, G_2\text{`}g & = & \text{if } f \equiv g \text{ or } f \equiv \mathsf{any} \text{ or } g \equiv \mathsf{any} \text{ then } G_1 \# G_2 \text{ else } \mathit{true} \\
\mathsf{rd}\, y \mathbin{/\!\cdot\!/} \mathsf{wr}\, x & = & \text{if } x \equiv y \text{ then } \mathit{false} \text{ else } \mathit{true} \\
\delta \mathbin{/\!\cdot\!/} \varepsilon & = & \mathit{true} \quad \text{for all other pairs of atomic effects} \\
\delta \mathbin{/\!\cdot\!/} \varepsilon & = & \mathit{true} \quad \text{in case } \delta \text{ or } \varepsilon \text{ is empty} \\
(\varepsilon, \delta) \mathbin{/\!\cdot\!/} \eta & = & (\varepsilon \mathbin{/\!\cdot\!/} \eta) \wedge (\delta \mathbin{/\!\cdot\!/} \eta) \\
\delta \mathbin{/\!\cdot\!/} (\varepsilon, \eta) & = & (\delta \mathbin{/\!\cdot\!/} \varepsilon) \wedge (\delta \mathbin{/\!\cdot\!/} \eta)
\end{array}
$$

Fig. 11. The separator function $\cdot/\!/\cdot$ is defined by recursion on effects.

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

Fig. 12. Footprints of region expressions and atomic assertions well-formed in Φ.

enables a sort of framing of effects that is embodied in the proof rules for sequential composition and for while loops.

For any η, ε the separator formula $\eta \mathbin{/\!\cdot\!/} \varepsilon$ is defined in Figure 11 using function $\cdot/\!/\cdot$ which recurses on effects. The most interesting case is the first line: $\mathsf{rd}\, G\text{`}f \mathbin{/\!\cdot\!/} \mathsf{wr}\, H\text{`}f$ is the disjointness formula $G \# H$. We use the data group any to abstract from all field names, so $\mathsf{rd}\, G\text{`}\mathsf{any} \mathbin{/\!\cdot\!/} \mathsf{wr}\, H\text{`}f$ is $G \# H$ for any f. Writes on the left and reads on the right are ignored, so $\eta \mathbin{/\!\cdot\!/} \varepsilon$ is the same as $\mathit{reads}(\eta) \mathbin{/\!\cdot\!/} \mathit{writes}(\varepsilon)$. Note that $G \# H$ means the intersection of the regions contains at most null, which is not an allocated reference.

One can show by structural induction on effects that for any σ and any Φ-interpretation φ:

$$\sigma \models_\varphi \eta \mathbin{/\!\cdot\!/} \varepsilon \quad \text{iff} \quad \mathit{rlocs}(\sigma, \varphi, \eta) \cap \mathit{wlocs}(\sigma, \varphi, \varepsilon) = \varnothing \ .$$

The key property of a separator is to establish the agreement to which frame validity refers.

LEMMA 6.6 (SEPARATOR AGREEMENT). Consider any effects η and ε. Suppose $\sigma \rightarrow \tau \models_\psi \varepsilon$ and $\sigma \models_\psi \eta \mathbin{/\!\cdot\!/} \varepsilon$. Then $\mathit{Agree}(\sigma, \tau, \eta, \mathit{id}, \psi)$, where id is the identity on $\sigma(\mathsf{alloc})$.

The frame rule relies on separation to allow an assertion to be transferred from one point in control flow to a later one. The proof rules for sequence and While allow a write effect to be transferred, under a suitable notion of separation called immunity. We adapt the notion of immunity from RLI, simply by including the context for pure methods.

Definition 6.7 ($P; \Phi; \psi/\varepsilon$-*immune*). Region expression G is said to be ***immune from ε under*** P, Φ, ψ, written $P; \Phi; \psi/\varepsilon$-immune, iff this formula is $\Phi; \psi$-valid:

$$P \Rightarrow \mathit{ftpt}(G, \Phi) \mathbin{/\!\cdot\!/} \varepsilon$$

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

For example, $\mathsf{wr}\, x$ is $\mathit{true}; \Phi; \psi/\mathsf{wr}\, x$-immune vacuously. But $\mathsf{wr}\, \{x\}\text{`}f$ is not $\mathit{true}; \Phi; \psi/\mathsf{wr}\, x$-immune because the region expression $\{x\}$ in $\mathsf{wr}\, \{x\}\text{`}f$ is not: $\mathit{ftpt}(\{x\}, \Phi) \mathbin{/\!\cdot\!/} \mathsf{wr}\, x = \mathsf{rd}\, x \mathbin{/\!\cdot\!/} \mathsf{wr}\, x = \mathit{false}$. In contrast, $\mathsf{wr}\, \{x\}\text{`}f$ is $\mathit{true}; \Phi; \psi/\varepsilon$-immune provided $\mathsf{wr}\, x$ is not in ε.

LEMMA 6.8. Let G be $P, \Phi, \psi/\varepsilon$-immune and let φ be a Φ-interpretation that extends ψ. Then for any σ, σ' such that $\sigma \rightarrow \sigma' \models_\varphi \varepsilon$ and $\sigma \models_\varphi P$ we have $[\![G]\!]_\varphi \sigma = [\![G]\!]_\varphi \sigma'$.

LEMMA 6.9. Let η be $P, \Phi, \psi/\varepsilon$-immune and let φ be a Φ-interpretation that extends ψ. Then for any σ, σ' such that $\sigma \rightarrow \sigma' \models_\varphi \varepsilon$ and $\sigma \models_\varphi P$ we have $\mathit{rlocs}(\sigma, \varphi, \eta) = \mathit{rlocs}(\sigma', \varphi, \eta)$ and $\mathit{wlocs}(\sigma, \varphi, \eta) = \mathit{wlocs}(\sigma', \varphi, \eta)$.

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

$$\textsc{FrmDisj}\ \frac{P \vdash \varepsilon\ \mathsf{frm}\ Q_1 \qquad P \vdash \varepsilon\ \mathsf{frm}\ Q_2}{P \vdash \varepsilon\ \mathsf{frm}\ Q_1 \vee Q_2}$$

$$\textsc{FrmConj}\ \frac{P \vdash \varepsilon\ \mathsf{frm}\ Q_1 \qquad P \wedge Q_1 \vdash \varepsilon\ \mathsf{frm}\ Q_2}{P \vdash \varepsilon\ \mathsf{frm}\ Q_1 \wedge Q_2} \qquad \textsc{Frm}\forall_{\mathsf{int}}\ \frac{P \vdash^{\Gamma,x:\mathsf{int}} \varepsilon, \mathsf{rd}\,x\ \mathsf{frm}\ Q}{P \vdash^{\Gamma} \varepsilon\ \mathsf{frm}\ \forall x : \mathsf{int}\ \cdot\ Q}$$

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

$$\textsc{FrmEq}\ \frac{\Phi;\psi \models Q_1 \Leftrightarrow Q_2 \qquad P;\Phi;\psi \vdash \varepsilon\ \mathsf{frm}\ Q_1}{P;\Phi;\psi \vdash \varepsilon\ \mathsf{frm}\ Q_2} \qquad \textsc{FrmProjCtx}\ \frac{P \wedge Q \vdash \varepsilon\ \mathsf{frm}\ Q}{P \vdash \varepsilon\ \mathsf{frm}\ Q}$$

$$\textsc{FrmSub}\ \frac{R;\Phi;\psi \vdash \varepsilon\ \mathsf{frm}\ Q \qquad R;\Phi;\psi \models \varepsilon \leq \varepsilon' \qquad \Phi;\psi \models P \Rightarrow R}{P;\Phi;\psi \vdash \varepsilon'\ \mathsf{frm}\ Q}$$

Fig. 13. Rules for the framing judgment. Typing context Γ is elided in rules where the context is the same in every judgment of the rule. Context Φ, and Φ-interpretation ψ, are elided in rules where they are the same.

Framing rules. Figure 13 specifies (mostly) syntax-directed rules for the framing judgment $P;\Phi;\psi \vdash \varepsilon\ \mathsf{frm}\ Q$. The *ftpt* function is used for atomic formulas. For non-atomic formulas there are syntax-directed rules, for example, the rule for conjunction allows to infer $P;\Phi;\psi \vdash \varepsilon\ \mathsf{frm}\ Q_1 \wedge Q_2$ from $P;\Phi;\psi \vdash \varepsilon\ \mathsf{frm}\ Q_1$ and $P \wedge Q_1;\Phi;\psi \vdash \varepsilon\ \mathsf{frm}\ Q_2$. There are also subsidiary rules for subsumption of effects and for logical manipulation of P. The latter means that $P;\Phi;\psi \vdash \varepsilon\ \mathsf{frm}\ Q$ may be inferred from $R;\Phi;\psi \vdash \varepsilon\ \mathsf{frm}\ Q$ if $\Phi;\psi \models P \Rightarrow R$. These rules are adapted in a straightforward way from RLI.

As it happens, the framing rules preserve well-formedness, so it is enough to say the axioms must be instantiated by well-formed judgments. But later in connection with correctness judgments we require not only the premises but also the conclusion of a rule instance to be well-formed judgments.

LEMMA 6.10 (FRAME SOUNDNESS). Every derivable framing judgment is valid.

The proof is by induction on a derivation of a framing judgment $P;\Phi;\psi \vdash \eta\ \mathsf{frm}\ Q$, using soundness of the rules. Soundness of the rules is proved using using Lemmas 6.3 and 6.5.

6.3. On the importance of framed reads

An effect ε is said to have ***framed reads*** in method context Φ provided that for every $\mathsf{rd}\,G\text{`}f$ in $\varepsilon$, its footprint $\mathit{ftpt}(G,\Phi)$ is in $\varepsilon$. For example, with $r$:rgn the effect $\mathsf{rd}\,r\text{`}f$ does not have framed reads, but it is a subeffect of $\mathsf{rd}\,r\text{`}f, \mathsf{rd}\,r$ which does.

This section begins with remarks about the significance of framed reads, then proceeds to prove key properties of effects with framed reads. The section concludes with examples showing the necessity of framed reads for these properties.

In this article, the property of having framed reads is important for soundness of the proof rules for sequence and iteration: it allows transfer of a read effect from one control point to another.

From a methodological perspective, it seems advisable for all judgments and specifications to have framed reads, and that could be added to our definition of well formed judgments. In this foundational study, we choose not to do that, but instead to use explicit side conditions in the rules where framed reads are necessary.

For ε that has framed reads, if $\mathit{Agree}(\sigma,\sigma',\varepsilon,\pi,\theta)$ then $[\![G]\!]_\theta\sigma \overset{\pi}{\sim} [\![G]\!]_\theta\sigma'$ for any $\mathsf{rd}\,G\text{`}f$ in ε (by Lemma 6.5). Two additional properties are important. First, although agreement is defined in an asymmetric way, referring to the left state for interpretation of the readable locations, a kind of symmetry holds in case of framed reads.

LEMMA 6.11 (AGREEMENT SYMMETRY). Let Φ be a method context and φ be a Φ-interpretation. Suppose ε has framed reads and $\mathit{df}(\varepsilon,\Phi)$ is $\Phi;\varphi$-valid. Consider any states σ,σ' and any refperm π such that $\mathit{Agree}(\sigma,\sigma',\varepsilon,\pi,\varphi)$. Then

(a) $\mathit{rlocs}(\sigma',\varphi,\varepsilon) = \pi(\mathit{rlocs}(\sigma,\varphi,\varepsilon))$,
(b) $\mathit{Agree}(\sigma',\sigma,\varepsilon,\pi^{-1},\varphi)$.

PROOF. (a) For variables the equality follows immediately by definition of *rlocs* and definition (5). For heap locations the argument is by mutual inclusion. To show $\mathit{rlocs}(\sigma',\varphi,\varepsilon) \subseteq \pi(\mathit{rlocs}(\sigma,\varphi,\varepsilon))$, let $o.f \in \mathit{rlocs}(\sigma',\varphi,\varepsilon)$. By definition of *rlocs*, there exists region G such that ε contains $\mathsf{rd}\,G\text{`}f$ and $o \in [\![G]\!]_\varphi\sigma'$. Since ε has framed reads, ε contains $\mathit{ftpt}(G,\Phi)$, hence from $\mathit{Agree}(\sigma,\sigma',\varepsilon,\pi,\varphi)$ by Lemma 6.5 we get $[\![G]\!]_\varphi\sigma \overset{\pi}{\sim} [\![G]\!]_\varphi\sigma'$. Thus $o \in \pi([\![G]\!]_\varphi\sigma)$. So, we have $o.f \in \pi(\mathit{rlocs}(\sigma,\varphi,\varepsilon))$. Proof of the reverse inclusion is similar.

(b) For variables this is straightforward. For heap locations, consider any $o.f \in \mathit{rlocs}(\sigma',\varphi,\varepsilon)$. From (a), we have $\pi^{-1}(o).f \in \mathit{rlocs}(\sigma,\varphi,\varepsilon)$. From $\mathit{Agree}(\sigma,\sigma',\varepsilon,\pi,\varphi)$, we get $\sigma(\pi^{-1}(o).f) \overset{\pi}{\sim} \sigma'(o.f)$. Thus we have $\sigma'(o.f) \overset{\pi^{-1}}{\sim} \sigma(\pi^{-1}(o).f)$. □

The second important property is that for a pair of states σ,σ' that are in "symmetric" agreement, that transition to a pair τ,τ' forming an allowed dependence, the transition preserves agreement on any set of locations whatsoever.

LEMMA 6.12 (PRESERVATION OF AGREEMENT). Let Φ be a method context and φ be a Φ-interpretation. Suppose $\sigma,\sigma' \Rightarrow \tau,\tau' \models_\varphi \varepsilon$ and $\sigma',\sigma \Rightarrow \tau',\tau \models_\varphi \varepsilon$. Suppose $\mathit{Agree}(\sigma,\sigma',\varepsilon,\pi,\varphi)$ and $\mathit{Agree}(\sigma',\sigma,\varepsilon,\pi^{-1},\varphi)$. Let ρ be any refperm $\rho \supseteq \pi$ for which

$$\mathit{Lagree}(\tau,\tau',\rho,\mathit{freshLocs}(\sigma,\tau)\cup\mathit{written}(\sigma,\tau)) \tag{8}$$

Then for any set of locations W in σ, if $\mathit{Lagree}(\sigma,\sigma',\pi,W)$ then $\mathit{Lagree}(\tau,\tau',\rho,W)$.

Note that existence of such ρ is a consequence of $\sigma,\sigma' \Rightarrow \tau,\tau' \models_\varphi \varepsilon$.

PROOF. Using $\mathit{Agree}(\sigma',\sigma,\varepsilon,\pi^{-1},\varphi)$ we appeal to $\sigma',\sigma \Rightarrow \tau',\tau \models_\varphi \varepsilon$ to obtain refperm $\rho' \supseteq \pi^{-1}$ such that

$$\mathit{Lagree}(\tau',\tau,\rho',\mathit{freshLocs}(\sigma',\tau')\cup\mathit{written}(\sigma',\tau')) \tag{9}$$

Now suppose W is a set of locations in σ such that $\mathit{Lagree}(\sigma,\sigma',\pi,W)$. We show that $\mathit{Lagree}(\tau,\tau',\rho,W)$.

For $x \in W$, either $x \in \mathit{written}(\sigma,\tau)$ or $\sigma(x) = \tau(x)$.

— If $x \in \mathit{written}(\sigma,\tau)$ then from from (8), we have $\tau(x) \overset{\rho}{\sim} \tau'(x)$.
— If $\sigma(x) = \tau(x)$, we claim that $\sigma'(x) = \tau'(x)$. It follows that from $\mathit{Lagree}(\sigma,\sigma',\pi,W)$ we have $\tau(x) = \sigma(x) \overset{\pi}{\sim} \sigma'(x) = \tau'(x)$.
We prove the claim by contradiction. If it does not hold then $x \in \mathit{written}(\sigma',\tau')$. By (9) this implies $\tau'(x) \overset{\rho'}{\sim} \tau(x) = \sigma(x) \overset{\pi}{\sim} \sigma'(x)$. Then, since $\rho' \supseteq \pi^{-1}$, we would have $\sigma'(x) = \pi(\pi^{-1}(\tau'(x))) = \tau'(x)$, which is a contradiction.

For $o.f \in W$, either $o.f \in \mathit{written}(\sigma,\tau)$ or $\sigma(o.f) = \tau(o.f)$.

— If $o.f \in \mathit{written}(\sigma,\tau)$ then from (8), we have $\tau(o.f) \overset{\rho}{\sim} \tau'(\rho(o).f)$.

— If $\sigma(o.f) = \tau(o.f)$, we claim that $\sigma'(\pi(o).f) = \tau'(\pi(o).f)$. It follows that from $\mathit{Lagree}(\sigma, \sigma', \pi, W)$ we have $\tau(o.f) = \sigma(o.f) \overset{\pi}{\sim} \sigma'(\pi(o).f) = \tau'(\pi(o).f)$.
The claim $\sigma'(\pi(o).f) = \tau'(\pi(o).f)$ is proved by contradiction. If it does not hold then $\pi(o).f \in \mathit{written}(\sigma', \tau')$. By (9) this implies $\tau'(\pi(o).f) \overset{\rho'}{\sim} \tau(\rho'\pi(o).f) = \tau(o.f) = \sigma(o.f) \overset{\pi}{\sim} \sigma'(\pi(o).f)$. Then, since $\rho' \supseteq \pi^{-1}$, we would have $\sigma'(\pi(o).f) = \pi(\pi^{-1}(\tau'(\pi(o).f))) = \tau'(\pi(o).f)$, hence $\sigma'(\pi(o).f) = \tau'(\pi(o).f)$, which is a contradiction.

This completes the proof of $\mathit{Lagree}(\tau, \tau', \pi, W)$ for heap locations. □

We conclude this section by showing necessity of the conditions in Lemmas 6.11 and 6.12.

Example 6.13. This example shows that allowed dependence is not symmetric, in general, and that symmetric instances of allowed dependence are necessary for the preservation of agreement Lemma 6.12. Consider the typing context

$\Gamma \mathrel{\widehat{=}} \mathsf{alloc{:}rgn}, r{:}\mathsf{rgn}, x{:}K$

We also consider type K with $\mathit{Fields}(K) = \{f : \mathsf{int}\}$. In this example we suppose that method context and its candidate interpretation are empty and we omit them. Now consider following four states, written in suggestive notation, and implicitly giving reference o type K.

$$\sigma \mathrel{\widehat{=}} [\mathsf{alloc{:}}\{o\}, r{:}\varnothing, o.f{:}3] \quad \sigma' \mathrel{\widehat{=}} [\mathsf{alloc{:}}\{o\}, r{:}\{o\}, o.f{:}3] \quad \tau \mathrel{\widehat{=}} \sigma \quad \tau' \mathrel{\widehat{=}} [\sigma' \mid o.f{:}4]$$

Consider the effect $\mathsf{rw}\, r\text{`}f$. We have $\mathit{rlocs}(\sigma, \mathsf{rw}\, r\text{`}f) = \varnothing$ and $\mathit{rlocs}(\sigma', \mathsf{rw}\, r\text{`}f) = \{o.f\}$. So we get $\mathit{Agree}(\sigma, \sigma', \mathsf{rw}\, r\text{`}f)$ and $\mathit{Agree}(\sigma', \sigma, \mathsf{rw}\, r\text{`}f)$ (for the identity refperm on $\{o\}$). We also have $\mathit{written}(\sigma, \tau) = \mathit{freshLocs}(\sigma, \tau) = \varnothing$. Thus we get $\mathit{Lagree}(\tau, \tau', id, \mathit{freshLocs}(\sigma, \tau) \cup \mathit{written}(\sigma, \tau))$. But we have $\mathit{written}(\sigma', \tau') = \{o.f\}$ and $\tau(o.f) \neq \tau'(o.f)$. Thus we don't have $\mathit{Lagree}(\tau', \tau, id, \mathit{freshLocs}(\sigma', \tau') \cup \mathit{written}(\sigma', \tau'))$. This shows that allowed dependence is not symmetric: $\sigma, \sigma' \Rightarrow \tau, \tau' \models \mathsf{rd}\, r\text{`}f$ but not $\sigma', \sigma \Rightarrow \tau', \tau \models \mathsf{rd}\, r\text{`}f$. Finally, let $W = \{o.f\}$; then we have $\mathit{Lagree}(\sigma, \sigma', id, W)$ but not $\mathit{Lagree}(\tau, \tau', id, W)$. Agreement is not preserved.

Note that this example does not arise in any method interpretation, because an interpretation is required to satisfy the allowed dependency both ways around (i.e., the bound variables σ, σ' in Def. 5.1 can be instantiated by both σ, σ' and σ', σ above). The next example does not satisfy the preservation of agreement either. But since it has symmetric allowed dependence, it can be part of an interpretation. This is used in Section 7 to show the necessity of framed reads in the proof rules for sequence and iteration.

Example 6.14. This example builds on (7) which shows that agreement on effects is not symmetric. This example illustrates the necessity of symmetric agreement for preservation of agreements in Lemma 6.12.

Consider the typing context

$\Gamma \mathrel{\widehat{=}} \mathsf{alloc} : \mathsf{rgn}, r : \mathsf{rgn}, x : K, j : \mathsf{int}$

where K is a type with with $\mathit{Fields}(K) = \{f : \mathsf{int}\}$. Consider the method context

$\Phi \mathrel{\widehat{=}}\ m() : P \rightsquigarrow \mathsf{true}\,[\mathsf{rw}\, r\text{`}f]$

where the precondition is this very particular condition:

$P \mathrel{\widehat{=}}\ 1 \leq |r| \leq 2 \wedge (\forall a, b : K \in \mathsf{alloc} \cdot\ a.f = 3 \wedge b.f = 5 \Rightarrow r = \{a, b\})$

The effect of $m()$ does not have framed reads. Since Φ doesn't contain any pure methods we consider empty interpretation as a candidate interpretation for Φ and omit it. Consider distinct references o, p and the following states:

$$\sigma \mathrel{\widehat{=}} [\mathsf{alloc{:}}\{o,p\}, r{:}\{o\}, x{:}o, j{:}0, o.f{:}3, p.f{:}4] \qquad \sigma' \mathrel{\widehat{=}} [\mathsf{alloc{:}}\{o,p\}, r{:}\{o,p\}, x{:}o, j{:}0, o.f{:}3, p.f{:}5]$$
$$\tau \mathrel{\widehat{=}} \sigma \qquad \tau' \mathrel{\widehat{=}} [\sigma' \mid o.f{:}6]$$

Consider the effect $\mathsf{rw}\, r\text{`}f$. We have $\mathit{rlocs}(\sigma, \mathsf{rw}\, r\text{`}f) = \{o.f\}$ and $\mathit{rlocs}(\sigma', \mathsf{rw}\, r\text{`}f) = \{o.f, p.f\}$. Since $\sigma(o.f) = 3 = \sigma'(o.f)$, we have $\mathit{Agree}(\sigma, \sigma', \mathsf{rw}\, r\text{`}f)$ (eliding the identity refperm of $\{o, p\}$). But since $\sigma(p.f) \neq 5$, we do not have the symmetric agreement $\mathit{Agree}(\sigma', \sigma, \mathsf{rw}\, r\text{`}f)$.

Since $\mathit{written}(\sigma, \tau) = \mathit{freshLocs}(\sigma, \tau) = \varnothing$, we have $\mathit{Lagree}(\tau, \tau', id, \mathit{freshLocs}(\sigma, \tau) \cup \mathit{written}(\sigma, \tau))$. So we have $\sigma, \sigma' \Rightarrow \tau, \tau' \models \mathsf{rw}\, r\text{`}f$. We also have the symmetric instance of allowed dependence, i.e., $\sigma', \sigma \Rightarrow \tau', \tau \models \mathsf{rw}\, r\text{`}f$, because $\mathit{Agree}(\sigma', \sigma, \mathsf{rw}\, r\text{`}f)$ is false.

To show that preservation of agreement fails, we consider $W = \{o.f\}$. We have $\mathit{Lagree}(\sigma, \sigma', id, W)$, but not $\mathit{Lagree}(\tau, \tau', id, W)$.

Example 6.15. Although Example 6.14 seems contrived, the states $\sigma, \sigma', \tau, \tau'$ can in fact arise in programs. Here we give Φ-interpretation φ such that $\varphi(m)(\sigma) = \{\tau\}$ and $\varphi(m)(\sigma') = \{\tau'\}$. Consider the map defined for all υ by

$$\varphi(m)(\upsilon) = \begin{cases} \{\lightning\} & \neg P \\ \{\upsilon'\} & \exists q, t : K \cdot\ q \neq t \wedge \upsilon(r) = \{q, t\} \wedge \upsilon(q.f) = 3 \wedge \upsilon(t.f) = 5 \wedge \upsilon' = [\upsilon \mid q.f{:}6] \\ \{\upsilon\} & \text{otherwise} \end{cases}$$

Notice that the second clause is well defined owing to P. One can check that $\varphi(m)(\sigma) = \{\tau\}$ and $\varphi(m)(\sigma') = \{\tau'\}$. We show that φ is indeed a Φ-interpretation. Let $\upsilon \in [\![\Gamma]\!]$.

— We have $\lightning \in \varphi(m)(\upsilon)$ iff $\upsilon \not\models P$.
— For all $\kappa \in \varphi(m)(\upsilon)$, we have $\kappa \models \mathsf{True}$ and $\upsilon{\to}\kappa \models \mathsf{rw}\, r\text{`}f$, since $\mathit{written}(\upsilon, \kappa) \subseteq \mathit{wlocs}(\upsilon, \mathsf{rw}\, r\text{`}f)$.
— For all κ, υ', κ' and π, if $\upsilon \models P$, $\upsilon' \models P$, $\kappa \in \varphi(m)(\upsilon)$, $\kappa' \in \varphi(m)(\upsilon')$ and $\mathit{Agree}(\upsilon, \upsilon', \mathsf{rw}\, r\text{`}f, \pi)$, then there are two cases:
  (a) Suppose $|\upsilon(r)| = 1$, or $\upsilon(r) = \{q, t\}$, and $\{\upsilon(q.f), \upsilon(t.f)\} \neq \{3{,}5\}$. Then we have $\kappa = \varphi(m)(\upsilon) = \upsilon$. Thus $\mathit{written}(\upsilon, \kappa) = \mathit{freshLocs}(\upsilon, \kappa) = \varnothing$, so we have $\mathit{Lagree}(\kappa, \kappa', \pi, \mathit{freshLocs}(\upsilon, \kappa) \cup \mathit{written}(\upsilon, \kappa))$.
  (b) Suppose $\upsilon(r) = \{q, t\}$ where $\upsilon(q.f) = 3$ and $\upsilon(t.f) = 5$. So $\kappa = [\upsilon \mid q.f{:}6]$. Also, we have $\mathit{rlocs}(\upsilon, \mathsf{rw}\, r\text{`}f) = \{q.f, t.f\}$, $\mathit{written}(\upsilon, \kappa) = \{q.f\}$, and $\mathit{freshLocs}(\upsilon, \kappa) = \varnothing$. From $\mathit{Agree}(\upsilon, \upsilon', \mathsf{rw}\, r\text{`}f, \pi)$, we know that there are references $q' = \pi(q)$ and $t' = \pi(t)$ such that $\upsilon'(q'.f) = 3$ and $\upsilon'(t'.f) = 5$. Since $\upsilon' \models P$, we have $\upsilon'(r) = \{q', t'\}$. Thus $\kappa' = [\upsilon' \mid q'.f{:}6]$ by definition of $\varphi(m)$. So we have $\mathit{Lagree}(\kappa, \kappa', \pi, \mathit{freshLocs}(\upsilon, \kappa) \cup \mathit{written}(\upsilon, \kappa))$.

7. PROOF SYSTEM FOR PROGRAM CORRECTNESS

This section gives the proof system, works out an example, and proves soundness of the rules.

Besides correctness judgments, the rules involve side conditions: validity of formulas, subeffects, and framing judgments. The linking rule for pure methods features one other condition based on the following.

Definition 7.1 (*correct partial candidate*). Let Φ and Φ, Θ both be swf and Θ a specification of pure methods only. Let ψ be a partial candidate for Φ. Let θ be a candidate interpretation of Θ. We say θ is a ***correct partial candidate*** for $\Phi, \Theta; \psi$, written

$$\theta \models \Phi, \Theta; \psi$$

provided that for any Φ-interpretation φ that extends ψ, the candidate $\varphi \cup \theta$ is a (Φ, Θ)-interpretation.[14]

We tend to use comma for union of disjoint partial maps in the context of judgments, for example, Φ, Θ. In other contexts it sometimes seems more clear to use $\cup$.

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

$$\textsc{FieldUpd}\ \frac{y \not\equiv x}{\Phi;\vdash x.f := y :\ x \neq \mathsf{null} \rightsquigarrow x.f = y\ [\mathsf{wr}\, x.f, \mathsf{rd}\, x, \mathsf{rd}\, y]}$$

$$\textsc{Assign}\ \frac{y \not\equiv x}{\Phi;\vdash x := F :\ x = y \rightsquigarrow x = F_y^x\ [\mathsf{wr}\, x, \mathit{ftpt}(F, \Phi)]}$$

$$\textsc{Alloc}\ \frac{y \not\equiv x \qquad \mathit{Fields}(K) = \overline{f} : \overline{T} \qquad f : K'' \text{ is in } \mathit{Fields}(K') \qquad g' : \mathsf{rgn} \text{ is in } \mathit{Fields}(K')}{\Phi;\vdash^{\Gamma} x := \mathsf{new}\ K :\ r = \mathsf{alloc} \rightsquigarrow \begin{array}{l} x \neq y \wedge x \notin r \wedge \mathsf{alloc} = r \cup \{x\} \wedge x.\overline{f} = \mathit{default}(\overline{T}) \\ \wedge (\forall z : K' \in \mathsf{alloc}.\ x \neq z.f \wedge x \notin z.g') \end{array}\ [\mathsf{wr}\, x, \mathsf{rw}\, \mathsf{alloc}]}$$

$$\textsc{Frame}\ \frac{\Phi;\psi \vdash C :\ P \rightsquigarrow Q\ [\varepsilon] \qquad P;\Phi;\psi \models \eta\ \mathsf{frm}\ R \qquad \Phi;\psi \models P \wedge R \Rightarrow \eta\ \mathcal{N}.\ \varepsilon}{\Phi;\psi \vdash C :\ P \wedge R \rightsquigarrow Q \wedge R\ [\varepsilon]}$$

$$\textsc{Conseq}\ \frac{\begin{array}{c}\Phi;\psi \vdash C :\ P \rightsquigarrow Q\ [\varepsilon] \\ \Phi;\psi \models P_1 \Rightarrow P \qquad \Phi;\psi \models Q \Rightarrow Q_1 \qquad P_1;\Phi;\psi \models \varepsilon \leq \varepsilon_1\end{array}}{\Phi;\psi \vdash C :\ P_1 \rightsquigarrow Q_1\ [\varepsilon_1]}$$

$$\textsc{InterpIntro}\ \frac{\Phi;\psi \vdash C :\ P \rightsquigarrow Q\ [\varepsilon]}{\Phi;\psi,\psi' \vdash C :\ P \rightsquigarrow Q\ [\varepsilon]}$$

$$\textsc{ImpureCall} \quad m : (x{:}T)P \rightsquigarrow Q\ [\varepsilon];\vdash m(z) :\ P_z^x \rightsquigarrow Q_z^x\ [\varepsilon_z^x, \mathsf{rd}\, z]$$

$$\textsc{PureCall}\ \frac{y \not\equiv z \qquad y \notin FV(Q)}{m : (x{:}T, \mathsf{res}{:}U)P \rightsquigarrow Q\ [\varepsilon];\vdash y := m(z) :\ P_z^x \rightsquigarrow Q_{z,y}^{x,\mathsf{res}}\ [\mathsf{wr}\, y, \mathsf{rd}\, z, \varepsilon_z^x]}$$

$$\textsc{ImpureLink}\ \frac{\begin{array}{c}\Theta \text{ is } m : (x{:}T)R \rightsquigarrow S\ [\eta] \\ m \notin B \qquad \Phi;\psi \vdash^{\Gamma, x:T} B :\ R \rightsquigarrow S\ [\mathsf{rd}\, x, \eta] \qquad \Phi,\Theta;\psi \vdash^{\Gamma} C :\ P \rightsquigarrow Q\ [\varepsilon]\end{array}}{\Phi;\psi \vdash^{\Gamma} \mathsf{let}\ m(x{:}T) = B\ \mathsf{in}\ C :\ P \rightsquigarrow Q\ [\varepsilon]}$$

$$\textsc{PureLink}\ \frac{\begin{array}{c}\Theta \text{ is } m : (x{:}T, \mathsf{res}{:}U)R \rightsquigarrow S\ [\eta] \qquad m \notin B \\ \Phi,\Theta;\psi,\theta \vdash^{\Gamma, x:T, \mathsf{res}:U} B :\ R \rightsquigarrow S\ [\mathsf{wr}\, \mathsf{res}, \mathsf{rd}\, x, \eta] \qquad \Phi,\Theta;\psi \vdash^{\Gamma} C :\ P \rightsquigarrow Q\ [\varepsilon] \\ \mathit{dom}\, \theta = \mathit{dom}\, \Theta \qquad \Phi;\psi \models R \wedge S \Rightarrow \mathsf{res} = m(x) \qquad \theta \models \Phi,\Theta;\psi\end{array}}{\Phi;\psi \vdash^{\Gamma} \mathsf{let}\ m(x{:}T, \mathsf{res}{:}U) = B\ \mathsf{in}\ C :\ P \rightsquigarrow Q\ [\varepsilon]}$$

$$\textsc{Seq}\ \frac{\begin{array}{c}\Phi;\psi \vdash C_1 :\ P \rightsquigarrow P_1\ [\varepsilon_1] \\ \Phi;\psi \vdash C_2 :\ P_1 \rightsquigarrow Q\ [\varepsilon_2, \mathsf{wr}\, H\, \text{`}\overline{f}, \mathsf{rd}\, H\, \text{`}\overline{f}] \qquad \varepsilon_1 \text{ and } \varepsilon_2 \text{ have framed reads} \\ \Phi;\psi \vdash P_1 \Rightarrow H\#g \qquad \varepsilon_2 \text{ is } P;\Phi;\psi/\varepsilon_1\text{-immune} \qquad \mathsf{wr}\, g \notin \varepsilon_1\end{array}}{\Phi;\psi \vdash C_1; C_2 :\ P \wedge g = \mathsf{alloc} \rightsquigarrow Q\ [\varepsilon_1, \varepsilon_2]}$$

$$\textsc{While}\ \frac{\begin{array}{c}\Phi;\psi \vdash C :\ P \wedge (x \neq 0) \rightsquigarrow P\ [\varepsilon, \mathsf{wr}\, H\, \text{`}\overline{f}, \mathsf{rd}\, H\, \text{`}\overline{f}] \qquad \varepsilon \text{ has framed reads} \\ \varepsilon \text{ is } P;\Phi;\psi/(\varepsilon, \mathsf{wr}\, H\, \text{`}\overline{f})\text{-immune} \qquad \Phi;\psi \models P \Rightarrow H\#g \qquad \mathsf{wr}\, g \notin \varepsilon\end{array}}{\Phi;\psi \vdash \mathsf{while}\ x\ \mathsf{do}\ C :\ P \wedge g = \mathsf{alloc} \rightsquigarrow P \wedge (x = 0)\ [\varepsilon, \mathsf{rd}\, x]}$$

Fig. 14. Selected proof rules. The notation $y \not\equiv x$ indicates the variables are syntactically distinct.

7.1. The proof system

Figure 14 presents selected proof rules. They are to be instantiated only with well-formed premises and conclusions (Definition 5.5). To emphasize the point we make the following definitions.

A correctness judgment is ***derivable*** iff it is well-formed and can be inferred using the proof rules instantiated with well-formed premises and conclusion. A proof rule is ***sound*** if for any instance with well-formed premises and conclusion, the conclusion is valid if the premises are valid and the side conditions hold.

THEOREM 7.2. *Each rule in Figure 14 is sound.*

An immediate corollary is that every derivable correctness judgment is valid. The proof comprises Section 7.5.

Here are a few comments on the rules.

The first rule, 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 (Section 7.1) and RLII (Section 7.1). Note that the pointer x is read, in order to write the field $x.f$.

Rule ASSIGN is formulated using the *ftpt* function to compute the read effect.

Rule ALLOC is a little complicated. To explain it, consider the following simpler rule:

$$\text{ALLOC1}\ \frac{\mathit{Fields}(K) = \overline{f} : \overline{T}}{\Phi;\vdash^{\Gamma} x := \mathsf{new}\ K : r = \mathsf{alloc} \rightsquigarrow x \notin r \wedge \mathsf{alloc} = r \cup \{x\} \wedge x.\overline{f} = \mathit{default}(\overline{T})\ [\mathsf{wr}\, x, \mathsf{rw}\, \mathsf{alloc}]}$$

Rule ALLOC1 has a postcondition that each field f in the list $\overline{f}$ of fields has the default value for its type. The rule uses variable r, that is not written, to snapshot the initial value of alloc in order to express freshness by postcondition $x \in \mathsf{alloc}$. This technique is also used in the rules SEQ and WHILE, avoiding the need for freshness effects as in RLI/II.[15] Note that the command reads alloc. In RLI, the allocation rule is formulated using a freshness effect. It is there shown that a rule like ALLOC1 is derivable. That takes care of expressing freshness with respect to a pre-existing expression, and the rule of conjunction can be used to apply ALLOC1 several times to get freshness with respect to several expressions. Rule ALLOC1 is not sufficient, however, because it does not express freshness with respect to expressions in the scope of quantifiers. The general rule, ALLOC, includes a postcondition that expresses freshness of the new object with respect to existing fields of reference and region type.

Next is the frame rule, adapted from RLI/II by adding partial candidate ψ and adding context $\Phi;\psi$ for the side conditions.

For PURECALL, the variable condition $y \notin FV(Q)$ should actually follow from specifications being swf, using the distinction between local and global variables mentioned in Footnote 7.

We give two illustrative rules 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 —subject to the proviso concerning well founded dependency among pure method preconditions, see $\prec^{+}_{\Phi}$ in the definition of swf method context in Section 2.

As discussed in Section 1, one premise of PURELINK is that partial (Φ,Θ)-interpetation θ is provided; its purpose is to give 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.

The condition $m \notin B$ in both of the linking rules disallows recursion. The general form of linking would allow mutual recursion among a group of pure and impure methods linked simultaneously, as we do in the programming language syntax and semantics. We believe that linking with recursion is

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

[15] In RLI the postcondition also includes an assertion $\mathsf{type}(K,\{x\})$ that x has type K but in this article we refrain from including that among the primitive formulas.

sound and can be proved using the same argument as for the linking rule in RLII, but we have not checked the details.

Aside from recursion, simultaneous linking is also needed for multiple pure methods that share a data representation and so need to have compatible interpretations (cf. Definition 7.1).

Rules SEQ and WHILE use immunity conditions to ensure that the interpretation of effects is consistent between the relevant points of control flow. In RLI, immunity is needed in these rules to deal with write effects. Here, it is needed as well for write effects, but in addition we need effects to be read framed. This ensures that certain agreements that hold initially also hold after executing commands in sequence, including iteratively, despite the use of state-dependent expressions in effects. The relevant technical result is Lemma 6.12. Section 7.2 shows necessity of framed reads.

About WHILE, note that $\mathsf{rd}\, x$ may be in ε but need not be if the loop body doesn't read x.

Not shown is the proof rule for local variable blocks; it is adapted straightforwardly from RLI. The premise allows reading and writing the local variable; these effects do not appear in the conclusion. That is, it drops read and write effects on the local variable. This makes it possible for a pure method to traverse heap structure using loops.

Not all valid judgments are provable. Here is an example judgment that involves a read effect that is not observable:

$$\ldots \vdash x := 0; y := x : \mathsf{true} \rightsquigarrow \mathsf{true}\,[\mathsf{wr}\, x, \mathsf{wr}\, y, \mathsf{rd}\, x]$$

Dropping $\mathsf{rd}\, x$ yields a valid judgment, because the final values are independent from the initial value of x. This is not derivable in our proof system. The logic RLI includes 'masking' rules that remove from a frame condition a write effect if the postcondition says the written location is unchanged from its initial value. In a more general relational logic, it is possible to formulate masking rules for read effects.

7.2. Examples showing the need for reads to be framed

Example 7.3. This example shows that in a sequence of two commands, the effect of the judgment of the first command needs to have framed reads. Recall the specification in Example 6.14:

$\Phi \mathrel{\widehat{=}} m() : P \rightsquigarrow \mathsf{true}\,[\mathsf{rw}\, r\text{`}f]$

$P \mathrel{\widehat{=}} 1 \le |r| \le 2 \wedge (\forall a, b : K \cdot a.f = 3 \wedge b.f = 5 \Rightarrow r = \{a, b\})$

We use $\mathsf{m}()$ for the first command in sequence. The judgment $\Phi; \varnothing \vdash m() : P \rightsquigarrow \mathsf{true}\,[\mathsf{rw}\, r\text{`}f]$ is valid, as it is an instance of the proof rule IMPURECALL. The FRAME rule yields this valid judgment:

$$\Phi; \varnothing \vdash m() : P \wedge x \neq \mathsf{null} \rightsquigarrow \mathsf{x} \neq \mathsf{null}\,[\mathsf{rw}\, r\text{`}f]. \tag{10}$$

For the second command, define

$C_1 \mathrel{\widehat{=}} j := x.f; \mathsf{if}\ j = 3\ \mathsf{then}\ j := -1\ \mathsf{else}\ j := 1$

Consider the judgment

$$\Phi; \varnothing \vdash C_1 : x \neq \mathsf{null} \rightsquigarrow \mathsf{true}\,[\mathsf{rd}\, x.f, \mathsf{rd}\, x, \mathsf{rw}\, j]. \tag{11}$$

This can be derived using the proof rules; note in particular that the effect is read framed. Using rule SEQ on (10) and (11) —but ignoring the side condition that the first judgment has framed reads— would give us the judgment

$$\Phi; \varnothing \vdash m(); C_1 : P \wedge x \neq \mathsf{null} \rightsquigarrow \mathsf{true}\,[\mathsf{rw}\, r\text{`}f, \mathsf{rd}\, x.f, \mathsf{rd}\, x, \mathsf{rw}\, j].$$

We show that this judgment is invalid because the read effect property fails.

Consider the interpretation φ from Example 6.15. Let σ, σ' be these states from Example 6.14:

$\sigma \mathrel{\widehat{=}} [\mathsf{alloc}{:}\{o, p\}, r{:}\{o\}, x{:}o, j{:}0, o.f{:}3, p.f{:}4] \qquad \sigma' \mathrel{\widehat{=}} [\mathsf{alloc}{:}\{o, p\}, r{:}\{o, p\}, x{:}o, j{:}0, o.f{:}3, p.f{:}5]$

$\tau \mathrel{\widehat{=}} \sigma \qquad \tau' \mathrel{\widehat{=}} [\sigma' \mid o.f{:}6]$

We have $\mathit{Agree}(\sigma, \sigma', (\mathsf{rw}\, r\text{`}f, \mathsf{rd}\, x.f, \mathsf{rd}\, x, \mathsf{wr}\, j))$. The transitions for $m()$ lead respectively to states τ, τ', Let κ, κ' be the respective states after C_1 executes from τ, τ'. Then we have $j \in \mathit{written}(\sigma, \kappa)$, $\kappa(j) = -1$, and $\kappa'(j) = 1$. This contradicts $\mathit{Lagree}(\tau, \tau', \pi, \mathit{freshLocs}(\sigma, \tau) \cup \mathit{written}(\sigma, \tau))$.

In conclusion, without the requirement of framed reads for the first command in a sequence, we could derive invalid conclusions from valid premises.

Example 7.4. To show the necessity for the second command in a sequence to have framed reads, we begin by considering the judgment

$$\Phi; \varnothing \vdash x.f := 3 :\ P \wedge x \neq \mathsf{null} \rightsquigarrow P\, [\mathsf{rd}\, x, \mathsf{wr}\, x.f]. \tag{12}$$

This can be derived using FIELDUPD, FRAME, and CONSEQ, so it is valid. Using rule SEQ on (10) and (12) —but without the side condition that the effect $\mathsf{rw}\, r\text{`}f$ of (10) has framed reads— yields

$$\Phi; \varnothing \vdash x.f := 3; m() :\ P \wedge x \neq \mathsf{null} \rightsquigarrow \mathsf{true}\, [\mathsf{rw}\, r\text{`}f, \mathsf{rd}\, x, \mathsf{wr}\, x.f].$$

We show that this is invalid owing to its read effect. Starting from states

$$\upsilon \mathrel{\widehat{=}} [\mathsf{alloc}{:}\{o, p\}, r{:}\{o\}, x{:}o, o.f{:}0, p.f{:}4] \quad \text{and} \quad \upsilon' \mathrel{\widehat{=}} [\mathsf{alloc}{:}\{o, p\}, r{:}\{o\}, x{:}o, o.f{:}0, p.f{:}4]$$

we have $\mathit{Agree}(\upsilon, \upsilon', \mathit{id}, \mathsf{rw}\, r\text{`}f, \mathsf{rd}\, x)$. We also have transitions $\langle x.f := 3; m(),\ \upsilon,\ \_\rangle \overset{\varphi}{\longmapsto}{}^{*}$ $\langle m(),\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \tau,\ \_\rangle$ and $\langle x.f := 3; m(),\ \upsilon',\ \_\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle m(),\ \sigma',\ \_\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \tau',\ \_\rangle$. Here $\sigma, \sigma', \tau, \tau'$ are as in Example 7.3. Notice that $\mathit{written}(\upsilon, \tau) = \{o.f\}$ and $\mathit{freshLocs}(\upsilon, \tau) = \varnothing$. But since $\tau(o.f) = 3 \neq 6 = \tau'(o.f)$, we do not have $\mathit{Lagree}(\tau, \tau', \mathit{id}, \mathit{freshLocs}(\upsilon, \tau) \cup \mathit{written}(\upsilon, \tau))$.

In conclusion, without the requirement of framed reads for the second command in a sequence, we could derive invalid conclusions from valid premises.

7.3. Example: proof of a Cell client

To illustrate features of some of the rules, we sketch a proof of an example, namely the following client of the Cell example from Figure 15

$$d := \mathsf{new}\mathit{Cell}; \mathit{init}(d); \mathit{set}(c, 5); \mathit{set}(d, 4) : I \wedge c \neq \mathsf{null} \rightsquigarrow I \wedge \mathit{get}(c) = 5\ [\eta]$$

where

$$\eta \mathrel{\widehat{=}} \mathsf{rw}\, d, \mathsf{rw}\, \mathsf{alloc}, \mathsf{rw}\, c.\mathit{foot}\text{`}\mathsf{any}, \mathsf{rd}\, c.\mathit{foot}, \mathsf{rd}\, c$$

Recall that rw abbreviates a read and write, e.g., $\mathsf{rw}\, c.\mathit{foot}\text{`}\mathsf{any}$ abbreviates the effects $\mathsf{rd}\, c.\mathit{foot}\text{`}\mathsf{any}, \mathsf{wr}\, c.\mathit{foot}\text{`}\mathsf{any}$. The typing context is

$$\Gamma \mathrel{\widehat{=}} \mathsf{alloc} : \mathsf{rgn}, r : \mathsf{rgn}, c : \mathit{Cell}, d : \mathit{Cell}$$

The hypothesis context, Φ, is comprised of the three specifications in Figure 15. The partial candidate is empty. In a richer specification language, I would be declared as a class invariant —a public one, because it is useful to clients and does not expose the internal representation.

Method init serves as constructor for Cell. Its specification uses ghost variable r, declared in Γ, in an idiom to express freshness. It would be better for r to be a specification variable, so it could be instantiated in different ways as needed for more complicated clients, but we do not formalize those in this article. The implementation of init is simply $\mathit{self}.\mathit{foot} := \{\mathit{self}\}$, which suffices for our implementation of set. But the specification follows good practice, which is to allow allocation in constructors.

In the following, all judgments considered are under hypotheses Φ and in context Γ, so to save space we refrain from writing those or the $\vdash$.

The proof proceeds by using small axioms for the atomic commands, rule FRAME to adapt their specifications, and rule SEQ to combine the commands in sequence, working from the left.

Formulae I and its variant I_r are framed by $\mathsf{rd}\, \mathsf{alloc}, \mathsf{rd}\, \mathsf{alloc}\text{`}\mathit{foot}$ and $\mathsf{rd}\, r, \mathsf{rd}\, r\text{`}\mathit{foot}$, respectively.

```
pure method get(self: Cell): int
    requires self ≠ null ∧ I
    ensures get(self) = result ∧ I
    reads self.foot`any

method set(self: Cell, v: int)
    requires self ≠ null ∧ I
    ensures get(self) = v ∧ I
    reads self.foot`any
    writes self.foot`any

method init(self: Cell)
    requires self ≠ null ∧ I_r
    ensures (r\{self}) # self.foot ∧ I
    reads alloc, self.foot
    writes alloc, self.foot
```

$\mathsf{I} \mathrel{\widehat{=}} \forall \mathsf{x,y{:}Cell} \in \mathsf{alloc} \cdot \mathsf{x} \in \mathsf{x.foot} \wedge (\mathsf{x{=}y} \vee (\mathsf{x.foot} \# \mathsf{y.foot} \wedge \mathsf{x} \notin \mathsf{y.foot}))$
$\mathsf{I}_r \mathrel{\widehat{=}} \forall \mathsf{x,y{:}Cell} \in \mathsf{r} \cdot \mathsf{x} \in \mathsf{x.foot} \wedge (\mathsf{x{=}y} \vee (\mathsf{x.foot} \# \mathsf{y.foot} \wedge \mathsf{x} \notin \mathsf{y.foot}))$

Fig. 15. Specifications for methods of Cell, adapted from Figure 1, using syntax close to the formalization but allowing multiple parameters. The name **self** has no special semantics.

By ALLOC using that the default value for $foot$: rgn is $\varnothing$, we get

$$d := \mathsf{new}\, Cell : r = \mathsf{alloc} \rightsquigarrow \begin{array}{l} d \notin r \wedge \mathsf{alloc} = r \cup \{d\} \wedge d \neq c \wedge \\ d.foot = \varnothing \wedge (\forall b : Cell \in r \cdot \ d \notin b.foot) \end{array} [\mathsf{wr}\, d, \mathsf{rw}\, \mathsf{alloc}]$$

Aiming to use FRAME with I_r, since I_r is framed by $\mathsf{rd}\, r, \mathsf{rd}\, r\text{`}foot$, we compute the separator formula

$$\begin{array}{rcll} (\mathsf{rd}\, r, \mathsf{rd}\, r\text{`}foot) \mathbin{/\!\!\cdot} \mathsf{wr}\, d, \mathsf{wr}\, \mathsf{alloc} & = & (\mathsf{rd}\, r \mathbin{/\!\!\cdot} \mathsf{wr}\, d) \wedge (\mathsf{rd}\, r \mathbin{/\!\!\cdot} \mathsf{wr}\, \mathsf{alloc}) & \text{by definition of } /\!\!\cdot \\ & & \wedge (\mathsf{rd}\, r\text{`}foot \mathbin{/\!\!\cdot} \mathsf{wr}\, d) \wedge (\mathsf{rd}\, r\text{`}foot \mathbin{/\!\!\cdot} \mathsf{wr}\, \mathsf{alloc}) & \\ & = & \mathsf{true} \wedge \mathsf{true} \wedge \mathsf{true} \wedge \mathsf{true} & \text{by definition of } /\!\!\cdot \end{array}$$

Because $I_r \wedge r = \mathsf{alloc} \Rightarrow \mathsf{true} \wedge \mathsf{true} \wedge \mathsf{true} \wedge \mathsf{true}$ is valid, by FRAME we get

$$d := \mathsf{new}\, Cell : I_r \wedge r = \mathsf{alloc} \rightsquigarrow \begin{array}{l} I_r \wedge d \notin r \wedge \mathsf{alloc} = r \cup \{d\} \wedge d \neq c \wedge \\ d.foot = \varnothing \wedge (\forall b : Cell \in r \cdot \ d \notin b.foot) \end{array} [\mathsf{wr}\, d, \mathsf{rw}\, \mathsf{alloc}]$$

Again aiming to frame $c \neq \mathsf{null}$, noting $\mathsf{rd}\, c$ frames $c \neq \mathsf{null}$ and

$$\mathsf{rd}\, c \mathbin{/\!\!\cdot} (\mathsf{wr}\, d, \mathsf{rw}\, \mathsf{alloc}) = (\mathsf{rd}\, c \mathbin{/\!\!\cdot} \mathsf{wr}\, d) \wedge (\mathsf{rd}\, c \mathbin{/\!\!\cdot} \mathsf{wr}\, \mathsf{alloc}) = \mathsf{true} \wedge \mathsf{true}$$

by FRAME, we get

$$d := \mathsf{new}\, Cell : I_r \wedge r = \mathsf{alloc} \wedge c \neq \mathsf{null} \rightsquigarrow \begin{array}{l} I_r \wedge d \notin r \wedge d \neq c \wedge \\ \wedge \mathsf{alloc} = r \cup \{d\} \wedge d.foot = \varnothing \\ \wedge (\forall b : Cell \in r \cdot \ d \notin b.foot) \wedge c \neq \mathsf{null} \end{array} [\mathsf{wr}\, d, \mathsf{rw}\, \mathsf{alloc}]$$

By CONSEQ, using the validity of $I \Rightarrow I_r$ and $\mathsf{alloc} = r \cup \{d\} \Rightarrow d \neq \mathsf{null}$ (because $\mathsf{null} \notin \mathsf{alloc}$ in all states), we get

$$d := \mathsf{new}\, Cell : I \wedge r = \mathsf{alloc} \wedge c \neq \mathsf{null} \rightsquigarrow I_r \wedge d \notin r \wedge d \neq \mathsf{null} \wedge d \neq c \wedge c \neq \mathsf{null} [\mathsf{wr}\, d, \mathsf{rw}\, \mathsf{alloc}] \tag{13}$$

Now by IMPURECALL

$$init(d) : I_r \wedge d \neq \mathsf{null} \rightsquigarrow I \wedge (r \setminus \{d\} \# d.foot) [\mathsf{rw}\, \mathsf{alloc}, \mathsf{rw}\, d.foot, \mathsf{rd}\, d]$$

By FRAME of $(d \neq \mathsf{null} \wedge d \notin r \wedge d \neq c \wedge c \neq \mathsf{null})$, noting that $(\mathsf{rd}\, c, \mathsf{rd}\, d, \mathsf{rd}\, r)$ $\mathbin{\text{·/.}}$ $(\mathsf{wr}\, d.foot, \mathsf{wr}\, \mathsf{alloc}) = \mathsf{true} \wedge \mathsf{true} \wedge \mathsf{true} \wedge \mathsf{true} \wedge \mathsf{true} \wedge \mathsf{true}$, we get

$$init(d) : I_r \wedge d \neq \mathsf{null} \wedge d \notin r \wedge d \neq c \wedge c \neq \mathsf{null} \rightsquigarrow \begin{array}{l} I \wedge (r \setminus \{d\} \# d.foot) \\ \wedge d \neq \mathsf{null} \wedge d \notin r \\ \wedge d \neq c \wedge c \neq \mathsf{null} \end{array} \quad [\mathsf{rw}\, \mathsf{alloc}, \mathsf{rw}\, d.foot, \mathsf{rd}\, d]$$

Using CONSEQ, we can rewrite this judgment as

$$init(d) : I_r \wedge d \notin r \wedge d \neq \mathsf{null} \wedge d \neq c \wedge c \neq \mathsf{null} \rightsquigarrow \begin{array}{l} I \wedge d \notin r \wedge d \neq \mathsf{null} \\ \wedge d \neq c \wedge c \neq \mathsf{null} \\ \wedge (r \setminus \{d\} \# d.foot) \end{array} \quad [\mathsf{rd}\, d, \mathsf{rw}\, d.foot, \mathsf{rw}\, \mathsf{alloc}] \tag{14}$$

Next we use SEQ to combine (13) and (14) as follows. Let η_1 be the effect of (13), that is, $\eta_1 \mathrel{\widehat{=}} \mathsf{wr}\, d, \mathsf{rw}\, \mathsf{alloc}$. Then η_1 has framed reads and $\mathsf{wr}\, r \notin \eta_1$. Also, let $\eta_2 \mathrel{\widehat{=}} \mathsf{rw}\, \mathsf{alloc}, \mathsf{rd}\, d$. The effect of (14) can be written as $\eta_2, \mathsf{rw}\, d.foot$. Notice that η_2 is immune from η_1 under $I \wedge r = \mathsf{alloc} \wedge c \neq \mathsf{null}$ vacuously (since there is no region expression of the form $G\text{`}f$ in η_2) and η_2 has framed reads. Let $H_1 = \{d\}$ and $P_1 \mathrel{\widehat{=}} I \wedge d \notin r \wedge d \neq \mathsf{null} \wedge d \neq c \wedge c \neq \mathsf{null}$. We have $P_1 \Rightarrow H_1 \# r$ is valid. Thus all side conditions of SEQ are valid. By SEQ

$$d := \mathsf{new}\, Cell; init(d) : \; I \wedge r = \mathsf{alloc} \wedge c \neq \mathsf{null} \rightsquigarrow \begin{array}{l} I \wedge d \notin r \wedge d \neq \mathsf{null} \\ \wedge d \neq c \wedge c \neq \mathsf{null} \\ \wedge (r \setminus \{d\} \# d.foot) \end{array} \quad [\mathsf{rw}\, d, \mathsf{rw}\, \mathsf{alloc}] \tag{15}$$

By IMPURECALL for set, we get

$$set(c,5) : I \wedge c \neq \mathsf{null} \rightsquigarrow I \wedge get(c) = 5 [\mathsf{rw}\, c.foot\text{`}\mathsf{any}, \mathsf{rd}\, c]$$

Let $\eta_3 \mathrel{\widehat{=}} \mathsf{rw}\, c.foot\text{`}\mathsf{any}, \mathsf{rd}\, c.foot, \mathsf{rd}\, c$. We have $I \wedge c \neq \mathsf{null} \models \mathsf{rw}\, c.foot\text{`}\mathsf{any}, \mathsf{rd}\, c \leq \eta_3$. Aiming to frame $d \notin r \wedge d \neq \mathsf{null} \wedge d \neq c \wedge c \neq \mathsf{null} \wedge (r \setminus \{d\} \# d.foot)$, we have $(\mathsf{rd}\, c, \mathsf{rd}\, d, \mathsf{rd}\, r, \mathsf{rd}\, d.foot)$ $\mathbin{\text{·/.}}$ η_3. This is a conjunction of trues with $\{d\} \# c.foot$, which is implied by the conjunct $d \neq c$ of the above formula and the conjunct $d \notin c.foot$ of I. Thus by FRAME and CONSEQ, we get

$$set(c,5) : \begin{array}{l} I \wedge d \notin r \wedge d \neq \mathsf{null} \\ \wedge d \neq c \wedge c \neq \mathsf{null} \\ \wedge (r \setminus \{d\} \# d.foot) \end{array} \rightsquigarrow \begin{array}{l} I \wedge d \notin r \wedge d \neq \mathsf{null} \\ \wedge d \neq c \wedge get(c) = 5 \; [\eta_3] \\ \wedge (r \setminus \{d\} \# d.foot) \end{array} \tag{16}$$

Again we aim to use SEQ to compose (15) and (16). Let $\eta_4 \mathrel{\widehat{=}} \mathsf{rw}\, d, \mathsf{rw}\, \mathsf{alloc}$. Notice that both η_3 and η_4 have framed reads and $\mathsf{wr}\, r \notin \eta_4$. Let $H_2 = \varnothing$, $P_2 \mathrel{\widehat{=}} I \wedge d \notin r \wedge d \neq \mathsf{null} \wedge d \neq c \wedge get(c) = 5 \wedge (r \setminus \{d\} \# d.foot)$ and $P \mathrel{\widehat{=}} I \wedge r = \mathsf{alloc} \wedge c \neq \mathsf{null}$. Then $P_2 \Rightarrow H_2 \# r$ is valid. We also need to check that η_3 is immune from η_4 under P. The region expressions in η_3 are $c.foot$ and $\{c\}$. Thus we need to show the validity of $P \Rightarrow \mathit{ftpt}(\{c\}\text{`}foot, \Phi)$ $\mathbin{\text{·/.}}$ $\eta_4$ and $P \Rightarrow \mathit{ftpt}(\{c\}, \Phi)$ $\mathbin{\text{·/.}}$ $\eta_4$. It suffices to show the validity of the first implication. The validity of the second follows because $\mathit{ftpt}(\{c\}\text{`}foot, \Phi) = \mathsf{rd}\, \{c\}\text{`}foot, \mathit{ftpt}(\{c\}, \Phi) = \mathsf{rd}\, \{c\}\text{`}foot, \mathsf{rd}\, c$.

The validity of the first implication follows because its consequent reduces to a conjunction of trues. To wit, by definition of separator, we have

$$\begin{array}{rcl} \mathsf{rd}\, \{c\}\text{`}foot, \mathsf{rd}\, c \mathbin{\text{·/.}} (\mathsf{wr}\, d, \mathsf{wr}\, \mathsf{alloc}) & = & (\mathsf{rd}\, \{c\}\text{`}foot \mathbin{\text{·/.}} \mathsf{wr}\, d) \wedge (\mathsf{rd}\, c \mathbin{\text{·/.}} \mathsf{wr}\, d) \\ & & \wedge (\mathsf{rd}\, \{c\}\text{`}foot \mathbin{\text{·/.}} \mathsf{wr}\, \mathsf{alloc}) \wedge (\mathsf{rd}\, c \mathbin{\text{·/.}} \mathsf{wr}\, \mathsf{alloc}) \\ & = & \mathsf{true} \wedge \mathsf{true} \wedge \mathsf{true} \wedge \mathsf{true}, \end{array}$$

Thus by SEQ we get

$$d := \mathsf{new}\, Cell; init(d); set(c,5) : I \wedge r = \mathsf{alloc} \wedge c \neq \mathsf{null} \rightsquigarrow \begin{array}{l} I \wedge d \notin r \wedge d \neq \mathsf{null} \\ \wedge d \neq c \wedge get(c) = 5 \; [\eta_4, \eta_3] \\ \wedge (r \setminus \{d\} \# d.foot) \end{array} \tag{17}$$

Recall from definition of η that $\eta = \eta_4, \eta_3$.

By IMPURECALL for set, letting $\eta_5 \mathrel{\widehat{=}} \mathsf{rw}\, d.foot\text{`}\mathsf{any}, \mathsf{rd}\, d$, we get

$$set(d,4) : I \wedge d \neq \mathsf{null} \rightsquigarrow I \wedge get(d) = 4[\eta_5]$$

Aiming to frame $d \notin r \wedge d \neq c \wedge get(c) = 5 \wedge (r \setminus \{d\}\#d.foot)$, note that $(\mathsf{rd}\, c, \mathsf{rd}\, d, \mathsf{rd}\, r, \mathsf{rd}\, c.foot, \mathsf{rd}\, d.foot, \mathsf{rd}\, c.foot\text{`}\mathsf{any})$ $\cdot/\cdot$ $\eta_5$ is a conjunction of trues and $\mathsf{rd}\, c.foot\text{`}\mathsf{any}$ $\cdot/\cdot$ $\mathsf{wr}\, d.foot\text{`}\mathsf{any}$. The latter simplifies to $c.foot\#d.foot$. Because $I \wedge c \neq d \Rightarrow c.foot\#d.foot$ is valid, we can use FRAME to get

$$set(d,4) : \begin{array}{l} I \wedge d \notin r \wedge d \neq \mathsf{null} \\ \wedge d \neq c \wedge get(c) = 5 \\ \wedge (r \setminus \{d\}\#d.foot) \end{array} \rightsquigarrow \begin{array}{l} I \wedge d \notin r \wedge get(d) = 4 \\ \wedge d \neq c \wedge get(c) = 5 \\ \wedge (r \setminus \{d\}\#d.foot) \end{array} [\eta_5] \tag{18}$$

Now we check the side conditions of SEQ to compose (17) and (18). Let $H_3 = d.foot$ and $P_3 \mathrel{\widehat{=}} I \wedge d \notin r \wedge d \neq \mathsf{null} \wedge d \neq c \wedge get(c) = 5 \wedge (r \setminus \{d\}\#d.foot)$. Then $P_3 \Rightarrow H_3\#r$ is valid. Also, η has framed reads and $\mathsf{wr}\, r \notin \eta$. On the other hand $\eta_5 = \mathsf{rd}\, d, \mathsf{wr}\, H_3\text{`}\mathsf{any}, \mathsf{rd}\, H_3\text{`}\mathsf{any}$. Since $\mathsf{rd}\, d$ is P/η-immune, by SEQ we get

$$d := \mathsf{new}\, Cell; init(d); set(c,5); set(d,4) : I \wedge r = \mathsf{alloc} \wedge c \neq \mathsf{null} \rightsquigarrow \begin{array}{l} I \wedge d \notin r \wedge get(d) = 4 \\ \wedge d \neq c \wedge get(c) = 5 \\ \wedge (r \setminus \{d\}\#d.foot) \end{array} [\eta]$$

To finish the proof, from CONSEQ, using $I \wedge d \notin r \wedge get(d) = 4 \wedge d \neq c \wedge get(c) = 5 \wedge (r \setminus \{d\}\#d.foot) \Rightarrow I \wedge get(c) = 5$, we get

$$d := \mathsf{new}\, Cell; init(d); set(c,5); set(d,4) : I \wedge r = \mathsf{alloc} \wedge c \neq \mathsf{null} \rightsquigarrow I \wedge get(c) = 5[\eta]$$

In the example, variable r only serves to refer to the initial value of alloc. By a standard rule of Hoare logic (rule EXIST in RLI) the above judgment yields

$$d := \mathsf{new}\, Cell; init(d); set(c,5); set(d,4) : (\exists r : \mathsf{rgn} \cdot I \wedge r = \mathsf{alloc} \wedge c \neq \mathsf{null}) \rightsquigarrow I \wedge get(c) = 5[\eta]$$

Then by predicate calculus and rule CONSEQ, using that r is not free in I, we get

$$d := \mathsf{new}\, Cell; init(d); set(c,5); set(d,4) : I \wedge c \neq \mathsf{null} \rightsquigarrow I \wedge get(c) = 5[\eta]$$

7.4. Technical results needed for soundness of linking rules

The following definitions and technical results are adapted from RLII. They are only used in proving the linking rules.

The ***active command*** of a configuration $\langle C, \sigma, \mu \rangle$ is the command's redex, that is, the part that determines the transition. So $\mathit{Active}(C_1; C_2) = \mathit{Active}(C_1)$ and $\mathit{Active}(C) = C$ if there are no C_1, C_2 such that C is $C_1; C_2$.

In the following we consider intermediate configurations that may include local variables as well as end-markers for let-bound methods. This can be formalized by a notion of compatibility, as in RLII, but here we gloss over the details with the phrase "well formed for an extension of Γ".

The following says that if a configuration is not about to call m then the behavior is independent of whether m is in the environment or the context.

LEMMA 7.5 (INDEPENDENCE). Suppose Φ is swf in Γ and φ is a Φ-interpretation. Consider any $m, C, \sigma, \dot{\mu}, \varphi$ such that $m \in dom(\dot{\mu})$ and $\langle C, \sigma, \dot{\mu} \rangle$ is well formed for some extension of Γ. Let $\mu = \dot{\mu} \restriction m$ and suppose Θ specifies m and θ is a Θ-interpretation for m. Suppose C has no $\mathsf{elet}(m)$ (and note that by well-formedness of the configurations, C has no let binding of m). Suppose $\mathit{Active}(C)$ is not a call to m. Then for any $C', \sigma', \dot{\mu}'$ we have $\langle C, \sigma, \dot{\mu} \rangle \overset{\varphi}{\longmapsto} \langle C', \sigma', \dot{\mu}' \rangle$ if and only if $\langle C, \sigma, \mu \rangle \overset{\varphi \cup \theta}{\longmapsto} \langle C', \sigma', \mu' \rangle$ where $\mu' = \dot{\mu}' \restriction m$. Moreover $\langle C, \sigma, \dot{\mu} \rangle \overset{\varphi}{\longmapsto} \lightning$ iff $\langle C, \sigma, \mu \rangle \overset{\varphi \cup \theta}{\longmapsto} \lightning$.

A key part of the argument for linking m to its implementation goes by induction on the number of calls to m. The following notion helps in the formalization.

Definition 7.6 (m-*truncated*). A trace $\langle C, \sigma, \mu\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle D, \tau, \nu\rangle$ is called m-***truncated***, if D is either of the form $y := m(z); C'$ or the form $m(z); C'$, or the trace has no incomplete invocation of m.

In RLII, because we consider recursive methods, we use the term ***topmost call*** to refer to a call to a method m that is not invoked (directly or indirectly) from m itself, though it may be from a chain of other method invocations. Here we retain the term, to facilitate comparison with RLII, but note that for the method m in the linking rules, all calls to m are topmost.

LEMMA 7.7 (DECOMPOSITION FOR PURE ENVIRONMENT METHODS). Suppose $\mu_0(m) = (x : T, \mathsf{res} : U.B)$ and $\langle C_0, \sigma_0, \mu_0\rangle$ is compatible with $\Phi;\varphi$, where $m \notin \mathit{dom}\,\varphi$. Suppose $\langle C_0, \sigma_0, \mu_0\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle D, \tau, \nu\rangle$. Then there is $n \geq 0$ and, for all i $(0 < i \leq n)$, there are configurations $\langle C_i, \sigma_i, \mu_i\rangle$, variables z_i, y_i, x_i and res_i, states τ_i, υ_i and $\dot\sigma_i$ such that for all i $(0 < i \leq n)$

(1) $\langle C_{i-1}, \sigma_{i-1}, \mu_{i-1}\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle y_i := m(z_i); C_i, \tau_i, \mu_i\rangle$ without any intermediate configurations in which the call to m is the active command
(2) $\langle y_i := m(z_i); C_i, \tau_i, \mu_i\rangle \overset{\varphi}{\longmapsto} \langle B^{x,\mathsf{res}}_{x_i,\mathsf{res}_i}; y_i := \mathsf{res}_i; \mathsf{ecall}(x_i, \mathsf{res}_i); C_i, \upsilon_i, \mu_i\rangle$
and $\upsilon_i = [\tau_i + x_i, \mathsf{res}_i{:}\,\tau_i(z_i), \mathit{default}(U)]$ (note that x_i and res_i are fresh parameter names.)
(3) $\langle B^{x,\mathsf{res}}_{x_i,\mathsf{res}_i}, \upsilon_i, \mu_i\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{skip}, \dot\sigma_i, \mu_i\rangle$ and hence by semantics
$\langle B^{x,\mathsf{res}}_{x_i,\mathsf{res}_i}; y := \mathsf{res}_i; \mathsf{ecall}(x_i, \mathsf{res}_i); C_i, \upsilon_i, \mu_i\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{ecall}(x_i, \mathsf{res}_i); C_i, \ddot\sigma_i, \mu_i\rangle$, where $\ddot\sigma_i = [\dot\sigma_i \mid y_i{:}\,\dot\sigma(\mathsf{res}_i)]$
(4) $\langle \mathsf{ecall}(x_i, \mathsf{res}_i); C_i, \ddot\sigma_i, \mu_i\rangle \overset{\varphi}{\longmapsto} \langle C_i, \sigma_i, \mu_i\rangle$ and $\sigma_i = \ddot\sigma_i \restriction x_i \restriction \mathsf{res}_i$
(5) $\langle C_n, \sigma_n, \mu_n\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle D, \tau, \nu\rangle$ without any completed invocations of m —but allowing a topmost call that is incomplete.

LEMMA 7.8 (DECOMPOSITION FOR PURE INTERPRETED METHODS). Suppose that μ is method environment such that $m \notin \mathit{dom}\,\mu$ and $\langle C_0, \sigma_0, \mu_0\rangle$ is compatible with $\Phi;\varphi$, where $m \in \mathit{dom}\,\varphi$. Also, suppose $\langle C_0, \sigma_0, \mu_0\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle D, \tau, \nu\rangle$. Then there is $n \geq 0$ and, for all i $(0 < i \leq n)$, there are configurations $\langle C_i, \sigma_i, \mu_i\rangle$, variables z_i and y_i and states τ_i such that for all i $(0 < i \leq n)$

(1) $\langle C_{i-1}, \sigma_{i-1}, \mu_{i-1}\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle y_i := m(z_i); C_i, \tau_i, \mu_i\rangle$ without any intermediate configurations in which m is the active command
(2) $\langle y_i := m(z_i); C_i, \tau_i, \mu_i\rangle \overset{\varphi}{\longmapsto} \langle C_i, \sigma_i, \mu_i\rangle$ and $\sigma_i = [\tau_i \mid y_i{:}\,\varphi(m)(\tau_i, \tau_i(z_i))]$
(3) $\langle C_n, \sigma_n, \mu_n\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle D, \tau, \nu\rangle$ without any completed invocations of m —but allowing a topmost call that is incomplete.

LEMMA 7.9 (CHANGE OF METHOD ENVIRONMENT). Consider states σ and τ, candidate interpretation φ and command C. Then we have $\langle C, \sigma, \_\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{skip}, \tau, \_\rangle$ iff $\langle C, \sigma, \mu\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{skip}, \tau, \mu\rangle$ for any method environment μ with domain disjoint from the domain of φ and containing no method bound by let in C.

7.5. Proof of soundness

We consider each rule in turn, making reference to the named conditions Safety etc. in Definition 5.2.

FIELDUPD: Consider any Φ-interpretation φ, any state σ, where $\sigma \models_\varphi x \neq \mathsf{null}$. Thus $\sigma(x) \neq \mathsf{null}$. By semantics it is not the case that $\langle x.f = y, \sigma, \_\rangle \overset{\varphi}{\longmapsto} \lightning$. Instead $\langle x.f = y, \sigma, \_\rangle \overset{\varphi}{\longmapsto} \langle \mathsf{skip}, \tau, \_\rangle$ where $\tau \mathrel{\widehat{=}} [\sigma \mid x.f{:}\,\sigma(y)]$.

For proving Post we must show $\tau \models x.f = y$. By semantics $\tau \models x.f = y$ iff $\tau(x) \neq \mathsf{null}$ and $\tau(x.f) = \tau(y)$. Since neither x nor y is modified by field update, $\tau(x) = \sigma(x) \neq \mathsf{null}$ and $\tau(y) = \sigma(y)$. Thus $\tau(x.f) = \tau(y)$.

Let $\varepsilon \mathrel{\widehat{=}} \mathsf{wr}\, x.f, \mathsf{rd}\, x, \mathsf{rd}\, y$. Notice that $\mathit{wlocs}(\sigma,\varphi,\varepsilon) = \{\sigma(x).f\}$. Definition 4.1 together with the fact that τ is only different from σ in value of $x.f$ shows Write Effect.

To prove Read Effect consider $\langle x.f = y,\ \sigma',\ \_\rangle \overset{\varphi}{\longmapsto} \langle \mathsf{skip},\ \tau',\ \_\rangle$. Suppose $\sigma' \models x \neq \mathsf{null}$ and $\mathit{Agree}(\sigma,\sigma',\varepsilon,\pi,\varphi)$, where π is a refperm. By semantics $\tau' = [\sigma' \,|\, x.f : \sigma'(y)]$. Since $\mathit{rlocs}(\sigma,\varphi,\varepsilon) = \{x,y\}$, from $\mathit{Agree}(\sigma,\sigma',\varepsilon,\pi,\varphi)$ we have $\sigma(x) \overset{\pi}{\sim} \sigma'(x)$ and $\sigma(y) \overset{\pi}{\sim} \sigma'(y)$. Thus $\tau(\tau(x).f) \overset{\pi}{\sim} \tau'(\tau'(x).f)$. Also, $\mathit{written}(\sigma,\tau) = \{\sigma(x).f\}$ and $\mathit{freshLocs}(\sigma,\tau) = \varnothing$. These show that $\mathit{Lagree}(\tau,\tau',\rho,\mathit{written}(\sigma,\tau) \cup \mathit{freshLocs}(\sigma,\tau))$, where $\rho = \pi$.

ALLOC: Consider any Φ-interpretation φ, any state σ, where $\sigma \models_\varphi r = \mathsf{alloc}$. By semantics it is not the case that $\langle x := \mathsf{new}\ K,\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto} \lightning$. Instead we have $\langle x := \mathsf{new}\ K,\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto} \langle \mathsf{skip},\ [\sigma_1 \,|\, x : o],\ \_\rangle$, where $o \notin \sigma(\mathsf{alloc})$, $\sigma_1 = \mathit{New}(\sigma, o, K, \mathit{default}(\overline{T}))$ and $\mathit{Fields}(K) = \overline{f} : \overline{T}$. Let $\tau = [\sigma_1 \,|\, x : o]$.

Since τ is an extension of σ by x and o, we have $\tau(x) \notin \sigma(\mathsf{alloc}) = \sigma(r)$. Also, r is not modified. Thus $\tau(r) = \sigma(r)$. So, we have $\tau(x) \notin \tau(r)$ and $\tau(\mathsf{alloc}) = \tau(r \cup \{x\})$. On other hand, $\sigma(y) \in \sigma(r)$ and r is not modified. So, $\tau(y) \neq \tau(x)$. From semantics as mentioned above, $x.\overline{f} = \mathit{default}(\overline{T})$. Consider any $z : K' \in \mathsf{alloc}$. First suppose that $\tau(x) \neq \tau(z)$. Since $\tau(z) \in \tau(\mathsf{alloc})$, we have $\tau(z) = \sigma(z)$. Thus $\tau(z.f) = \sigma(z.f) \in \sigma(r) = \tau(r)$ and $\tau(z.g) = \sigma(z.g) \subseteq \sigma(r) = \tau(r)$. On other hand $\tau(x) \notin \tau(r)$. Thus $\tau(x) \neq \tau(z.f)$ and $\tau(x) \notin \tau(z.g)$. Now suppose that $\tau(x) = \tau(z)$. Since $x.\overline{f} = \mathit{default}(\overline{T})$, we have $x \neq x.f$ and $x \notin x.g$. These arguments prove Post.

To prove Write Effect, we need to show that $\sigma{\rightarrow}\tau \models \mathsf{wr}\, x, \mathsf{rw}\, \mathsf{alloc}$. For any $z \in r$, we have $\tau(z) = \sigma(z)$. Also, we have $\tau(\mathsf{alloc}) = \tau(r \cup \{x\})$ and $\mathit{wlocs}(\sigma,\varphi,(\mathsf{wr}\, x, \mathsf{rw}\, \mathsf{alloc})) = \{x, \mathsf{alloc}\}$. For any $\sigma(o) \in \sigma(r)$ and $\mathit{Fields}(\mathsf{type}(o,\sigma))$, $\tau(o.f) \in \tau(r) = \sigma(r)$. Thus $\tau(o.f) = \sigma(o.f)$.

To prove Read Effect, consider two states σ', τ' and refperm π such that $\langle x := \mathsf{new}\ K,\ \sigma',\ \_\rangle \overset{\varphi}{\longmapsto} \langle \mathsf{skip},\ \tau',\ \_\rangle$ and $\mathit{Agree}(\sigma,\sigma',(\mathsf{wr}\, x, \mathsf{rw}\, \mathsf{alloc}),\pi,\varphi)$. We have $\mathit{rlocs}(\sigma,\varphi,(\mathsf{wr}\, x, \mathsf{rw}\, \mathsf{alloc})) = \{\mathsf{alloc}\}$. Thus $\sigma(\mathsf{alloc}) \overset{\pi}{\sim} \sigma'(\mathsf{alloc})$. Define $\rho = \pi \cup \{(\tau(x), \tau'(x))\}$. Then

$$\tau(\mathsf{alloc}) = \tau(r) \cup \{\tau(x)\} = \sigma(r) \cup \{\tau(x)\} \overset{\rho}{\sim} \sigma'(r) \cup \{\tau'(x)\} = \tau'(\mathsf{alloc}).$$

Also $\mathit{written}(\sigma,\tau) = \{x\}$ and $\mathit{freshLocs}(\sigma,\tau) = \{x.\overline{f}\}$. By definition of ρ we have $\tau(x.\overline{f}) \overset{\rho}{\sim} \tau(\rho(x).\overline{f})$. Thus we have $\mathit{Lagree}(\tau,\tau',\rho,\mathit{written}(\sigma,\tau) \cup \mathit{freshLocs}(\sigma,\tau'))$.

FRAME: We must show $\Phi;\psi \models C : P \wedge R \rightsquigarrow Q \wedge R\,[\varepsilon]$, assuming validity of premises. Consider any Φ-interpretation φ such that $\psi \subseteq \varphi$. Suppose $\sigma \models_\varphi P \wedge R$. All the conditions in Definition 5.2 except Post are immediate from the validity of the premise, which also yields $\tau \models_\varphi Q$. To show Post it remains to show $\tau \models_\varphi R$. Because φ is a Φ-interpretation, the premise $\Phi;\psi \models P \wedge R \Rightarrow \eta \mathbin{/.} \varepsilon$ yields $\sigma \models_\varphi \eta \mathbin{/.} \varepsilon$. Instantiating the premise for C with φ gives $\sigma{\rightarrow}\tau \models_\varphi \varepsilon$ (Write Effect), so by Lemma 6.6 we have $\mathit{Agree}(\sigma,\tau,\eta,id,\varphi)$ where id is the identity on $\sigma(\mathsf{alloc})$. Now we appeal to the definition of $P;\Phi;\varphi \models \eta\ \mathsf{frm}\ R$ (Definition 6.4). Hence from $\mathit{Agree}(\sigma,\tau,\eta,id,\varphi)$ and $\sigma \models_\varphi P \wedge R$ we obtain $\tau \models_\varphi R$.

CONSEQ: For all Φ-interpretation φ that extends ψ and all state σ such that $\sigma \models_\varphi P_1$, from $\Phi;\overline{\psi \models P_1} \Rightarrow P$, we have $\sigma \models_\varphi P$. Thus by validity of the premise, we conclude that transition from $\langle C,\ \sigma,\ \_\rangle$ via $\overset{\varphi}{\longmapsto}$ cannot fault. To prove post and safety, consider state τ with $\langle C,\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip},\ \tau,\ \_\rangle$. Again from first premise we have $\tau \models_\varphi Q$ and $\sigma{\rightarrow}\tau \models_\varphi \varepsilon$. From $\Phi;\psi \models Q \Rightarrow Q_1$, we get $\tau \models_\varphi Q_1$. Since $\Phi;\psi \models \varepsilon \leq \varepsilon_1$, by Lemma 6.2 (allowed change) we have $\sigma{\rightarrow}\tau \models_\varphi \varepsilon_1$. To prove read effect, consider states τ, σ', τ and refperm π such that $\sigma' \models_\varphi P$, $\mathit{Agree}(\sigma,\sigma',\varepsilon_1,\pi,\varphi)$, $\langle C,\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip},\ \tau,\ \_\rangle$ and $\langle C,\ \sigma',\ \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip},\ \tau',\ \_\rangle$. By Lemma 6.2 (agreement), we get $\mathit{Agree}(\sigma,\sigma',\varepsilon,\pi,\varphi)$. By read effect property of the premise, there is a refperm ρ such that $\rho \supseteq \pi$ and $\mathit{Lagree}(\tau,\tau',\rho,\mathit{written}(\sigma,\tau) \cup \mathit{freshLocs}(\sigma,\tau))$.

INTERPINTRO: Note that by well-formedness, the union ψ, ψ' is a partial candidate so if there is any m in the domain of both then $\psi(m) = \psi'(m)$. Any partial interpretation that extends $\psi \cup \psi'$ also extends ψ, so the conclusion follows directly from the premise by semantics of correctness judgment.

SEQ: We only show Read Effect, as proofs for the other conditions are straightforward adaptations of the soundness proof in RLI. Consider any Φ-interpretation φ that extends ψ and suppose for states $\sigma, \sigma', \tau, \tau'$, for refperm π, we have

$$\sigma \models_\varphi P, \quad \sigma' \models_\varphi P, \quad \mathit{Agree}(\sigma, \sigma', (\varepsilon_1, \varepsilon_2), \pi, \varphi), \tag{19}$$

and

$$\langle C_1; C_2, \sigma, \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip}, \tau, \_\rangle \quad \text{and} \quad \langle C_1; C_2, \sigma', \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip}, \tau', \_\rangle.$$

We must show that there is a refperm ρ such that $\rho \supseteq \pi$ and

$$\mathit{Lagree}(\tau, \tau', \rho, \mathit{written}(\sigma, \tau) \cup \mathit{freshLocs}(\sigma, \tau)) \tag{20}$$

To show the agreement (20), observe that from $\sigma \models P$, $\sigma' \models P$, by semantics and validity of the first and second premises of the rule, there are states σ_1 and σ_1' such that

$$\langle C_1, \sigma, \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip}, \sigma_1, \_\rangle \quad \text{and} \quad \sigma_1 \models_\varphi P_1 \quad \text{and} \quad \langle C_2, \sigma_1, \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip}, \tau, \_\rangle,$$

and

$$\langle C_1, \sigma', \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip}, \sigma_1', \_\rangle \quad \text{and} \quad \sigma_1' \models_\varphi P_1 \quad \text{and} \quad \langle C_2, \sigma_1', \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip}, \tau', \_\rangle.$$

From (19) we have $\mathit{Agree}(\sigma, \sigma', \varepsilon_1, \pi, \varphi)$, so using the Read Effect property of the first premise we get some refperm $\rho_1 \supseteq \pi$ such that

$$\mathit{Lagree}(\sigma_1, \sigma_1', \rho_1, Y), \text{ where } Y = \mathit{written}(\sigma, \sigma_1) \cup \mathit{freshLocs}(\sigma, \sigma_1). \tag{21}$$

From the first premise, we also know that $\sigma{\to}\sigma_1 \models_\varphi \varepsilon_1$. Since ε_2 is $P, \Phi, \psi/\varepsilon_1$-immune, from Lemma 6.9, we have $\mathit{rlocs}(\sigma_1, \varphi, \varepsilon_2) = \mathit{rlocs}(\sigma, \varphi, \varepsilon_2)$. Hence $\mathit{rlocs}(\sigma_1, \varphi, \varepsilon_2) \subseteq \mathit{rlocs}(\sigma, \varphi, (\varepsilon_1, \varepsilon_2))$. Thus from (19), we have $\mathit{Lagree}(\sigma, \sigma', \pi, \mathit{rlocs}(\sigma_1, \varphi, \varepsilon_2))$. From validity of the first premise we have both $(\sigma, \sigma') \Rightarrow (\sigma_1, \sigma_1') \models_\varphi \varepsilon_1$ and $(\sigma', \sigma) \Rightarrow (\sigma_1', \sigma_1) \models_\varphi \varepsilon_1$. Since ε_1 has framed reads, using Lemma 6.12, we get

$$\mathit{Lagree}(\sigma_1, \sigma_1', \rho_1, \mathit{rlocs}(\sigma_1, \varphi, \varepsilon_2)). \tag{22}$$

Furthermore, since $\Phi, \psi \models P \Rightarrow H\#g$ and $\mathsf{wr}\, g \notin \varepsilon_1$, we have $\sigma_1(H) \subseteq \mathit{freshRefs}(\sigma, \sigma_1)$. Thus

$$\mathit{rlocs}(\sigma_1, \varphi, \mathsf{rd}\, H\, \text{`}\overline{f}) \subseteq \mathit{freshLocs}(\sigma, \sigma_1) \subseteq Y$$

and so, using (21) and (22), we have

$$\mathit{Lagree}(\sigma_1, \sigma_1', \rho_1, \mathit{rlocs}(\sigma_1, \varphi, (\varepsilon_2, \mathsf{rd}\, H\, \text{`}\overline{f}))).$$

By definitions this yields $\mathit{Agree}(\sigma_1, \sigma_1', (\varepsilon_2, \mathsf{wr}\, H\, \text{`}\overline{f}, \mathsf{rd}\, H\, \text{`}\overline{f}), \rho_1, \varphi)$. So we can appeal to the Read Effect property of the second premise, to conclude that there is a refperm $\rho \supseteq \rho_1$ such that

$$\mathit{Lagree}(\tau, \tau', \rho, W), \text{ where } W = \mathit{written}(\sigma_1, \tau) \cup \mathit{freshLocs}(\sigma_1, \tau). \tag{23}$$

To complete the proof of (20) observe that

$$\begin{aligned} & \mathit{written}(\sigma, \tau) \cup \mathit{freshLocs}(\sigma, \tau) \\ \subseteq\ & \mathit{written}(\sigma, \sigma_1) \cup \mathit{written}(\sigma_1, \tau) \cup \mathit{freshLocs}(\sigma, \sigma_1) \cup \mathit{freshLocs}(\sigma_1, \tau) \\ =\ & Y \cup W. \end{aligned}$$

From (23), we have $\mathit{Lagree}(\tau, \tau', \rho, W)$, so it remains to show $\mathit{Lagree}(\tau, \tau', \rho, Y)$. By validity of the second premise, we get $(\sigma_1, \sigma_1') \Rightarrow (\tau, \tau') \models_\varphi \varepsilon_2$ and $(\sigma_1', \sigma_1) \Rightarrow (\tau', \tau) \models_\varphi \varepsilon_2$. Since ε_2 has framed reads, using Lemma 6.12, from (21), we get $\mathit{Lagree}(\tau, \tau', \rho, Y)$.

IMPURECALL: Let Φ be $m : (x{:}T)P \rightsquigarrow Q\,[\varepsilon]$ and φ be an arbitrary Φ-interpretation, noting that the partial candidate is empty. To prove $\Phi \models m(z) :\ P^x_z \rightsquigarrow Q^x_z\,[\varepsilon^x_z, \mathsf{rd}\, z]$, suppose $\sigma \models_\varphi P^x_z$ and let μ be a Γ-environment. Let $v = \sigma(z)$. Then we have $\sigma \models_\varphi P^x_v$. The call cannot fault from σ, because that would contradict Definition 5.1(d) of context interpretation. The transitions are $\langle m(z),\, \sigma,\, \_\rangle \overset{\varphi}{\longmapsto} \langle \mathsf{skip},\, \tau,\, \_\rangle$ for all $\tau \in \varphi(m)(\sigma, v)$. By Definition 5.1(e), this yields $\tau \models_\varphi Q^x_v$, and $\sigma{\to}\tau \models_\varphi \varepsilon^x_v$, which gives us $\tau \models_\varphi Q^x_z$ and $\sigma{\to}\tau \models_\varphi \varepsilon^x_z$ since $\sigma(z) = v$.

Finally, to prove Read Effect, for any $\tau, \sigma', \tau', \pi$ suppose that $\sigma' \models^\Gamma_\varphi P^x_{v'}$, $\mathit{Agree}(\sigma, \sigma', (\varepsilon^x_v, \mathsf{rd}\, z), \pi, \varphi)$, $\langle m(z),\, \sigma,\, \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip},\, \tau,\, \_\rangle$ and $\langle m(z),\, \sigma',\, \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip},\, \tau',\, \_\rangle$, where $v' = \sigma'(z)$. From transition semantics $\tau \in \varphi(m)(\sigma, v)$ and $\tau' \in \varphi(m)(\sigma', v')$. Because $\mathit{Agree}(\sigma, \sigma', \mathsf{rd}\, z, \pi, \varphi)$, we have $v \overset{\pi}{\sim} v'$. Thus from Definition 5.1(f), there is $\rho \supseteq \pi$ with $\mathit{Lagree}(\tau, \tau', \rho, \mathit{freshLocs}(\sigma, \tau) \cup \mathit{written}(\sigma, \tau))$.

PURECALL: Recall that a swf specification for a pure method is not allowed to have a write effect. Let φ be any Φ-interpretation. Consider any σ such that $\sigma \models_\varphi P^x_z$. Let $w = \varphi(m)(\sigma, \sigma(z))$. Because φ is a Φ-interpretation, we know that w is not $\lightning$, see Definition 5.1(a). So $\langle y := m(z),\, \sigma,\, \_\rangle \overset{\varphi}{\longmapsto} \langle \mathsf{skip},\, \tau,\, \_\rangle$, where $\tau = [\sigma \,|\, y{:}\, w]$. Thus Safety is immediate. Furthermore, $\sigma \models_\varphi Q^{x,\mathsf{res}}_{z,w}$ by Definition 5.1(b) of context interpretation, hence $\sigma \models_\varphi Q^{x,\mathsf{res}}_{z,y}$. For Write Effect, it is immediate from semantics: $\tau = [\sigma \,|\, y{:}\, w]$ and $\mathsf{wr}\, y$ is in the frame condition. For Post, we must show $\tau \models_\varphi Q^{x,\mathsf{res}}_{z,y}$. Below we show that $\tau \models_\varphi Q^{x,\mathsf{res}}_{z,y}$ iff $\sigma \models_\varphi Q^{x,\mathsf{res}}_{z,w}$, whence we are done.

Because σ and τ possibly differ only on the value of y, and $y \not\equiv z$, we have $\tau(z) = \sigma(z)$. Now note that

$$
\begin{aligned}
& \tau \models_\varphi Q^{x,\mathsf{res}}_{z,y} \\
\Leftrightarrow\ & [\tau + x, \mathsf{res}{:}\, \tau(z), \tau(y)] \models_\varphi Q, \text{ by substitution property} \\
\Leftrightarrow\ & [\tau + x, \mathsf{res}{:}\, \tau(z), w] \models_\varphi Q, \text{ since } \tau(y) = w \\
\Leftrightarrow\ & [\tau + x, \mathsf{res}{:}\, \sigma(z), w] \models_\varphi Q, \text{ since } \tau(z) = \sigma(z) \\
\Leftrightarrow\ & [\sigma + x, \mathsf{res}{:}\, \sigma(z), w] \models_\varphi Q, \text{ since } y \notin FV(Q), y \not\equiv z \\
\Leftrightarrow\ & \sigma \models_\varphi Q^{x,\mathsf{res}}_{z,w}, \text{ by abbreviation (as } w \text{ is a value)}
\end{aligned}
$$

To show Read Effect, for any $\tau, \sigma', \tau', \pi$, suppose that $\mathit{Agree}(\sigma, \sigma', (\mathsf{wr}\, y, \mathsf{rd}\, z, \varepsilon^x_z), \pi, \varphi)$, $\sigma' \models_\varphi P^x_z$, $\langle y := m(z),\, \sigma,\, \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip},\, \tau,\, \_\rangle$ and $\langle y := m(z),\, \sigma',\, \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip},\, \tau',\, \_\rangle$. Let $w' = \varphi(m)(\sigma', \sigma'(z))$. By semantics, we have $\tau = [\sigma \,|\, y{:}\, w]$ and $\tau' = [\sigma' \,|\, y{:}\, w']$. From the agreement assumption we get, $\sigma(z) \overset{\pi}{\sim} \sigma'(z)$. Let $\rho = \pi$. Because φ is a context interpretation, by Definition 5.1(c) we therefore obtain $w \overset{\pi}{\sim} w'$. Hence $\tau(y) \overset{\pi}{\sim} \tau'(y)$ and we are done because τ, τ' differ from σ, σ' only in y.

PURELINK For this, we rely on nomenclature set out in the results of Section 7.4.

Suppose that Θ is $m : (x{:}\, T, \mathsf{res}{:}\, U)R \rightsquigarrow S\,[\eta]$ and $\mathit{dom}\,\theta = \mathit{dom}\,\Theta$. Suppose the side conditions hold:

$$\Phi; \psi \models R \wedge S \Rightarrow \mathsf{res} = m(x) \qquad \theta \models \Phi, \Theta; \psi \qquad m \notin B \tag{24}$$

Suppose the premises are valid, that is

$$\Phi, \Theta; \psi \models^\Gamma C :\ P \rightsquigarrow Q\,[\varepsilon] \tag{25}$$

$$\Phi, \Theta; \psi, \theta \models^{\Gamma, x:T, \mathsf{res}:U} B :\ R \rightsquigarrow S\,[\mathsf{wr}\, \mathsf{res}, \mathsf{rd}\, x, \eta] \tag{26}$$

Valid judgements are closed under renaming, so from (26) we also have, for fresh x', res'

$$\Phi, \Theta; \psi, \theta \models^{\Gamma, x':T, \mathsf{res}':U} B^{x,\mathsf{res}}_{x',\mathsf{res}'} :\ R^x_{x'} \rightsquigarrow S^{x,\mathsf{res}}_{x',\mathsf{res}'}\,[\mathsf{wr}\, \mathsf{res}', \mathsf{rd}\, x', \eta^x_{x'}]$$

Recall that by well-formedness, η is wr-free and does not mention res. The role of side condition $m \notin B$ is that, because B cannot invoke m, we will be able to appeal directly to the premise (26)

for B. (By contrast, if B is recursive an inductive argument is needed to establish its correctness properties, see RLII.)

We are to prove validity of the conclusion of the rule:

$$\Phi; \psi \models^{\Gamma} \mathsf{let}\ m(x{:}T, \mathsf{res}{:}U) = B\ \mathsf{in}\ C :\ P \rightsquigarrow Q\ [\varepsilon]. \tag{27}$$

To that end, let μ be a Γ-environment, let φ be a Φ-interpretation such that $\psi \subseteq \varphi$, and let σ be a Γ-state such that $\sigma \models_{\varphi} P$. The first transition is

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

where $\dot{\mu} = [\mu + m{:}(x : T, \mathsf{res} : U.B)]$. Continuing from there, any trace of $C; \mathsf{elet}(m)$ corresponds step by step with a trace of C containing a trailing $\mathsf{elet}(m)$ in every configuration with exactly the same states, followed by a final step that executes $\mathsf{elet}(m)$. This step just removes m from $\dot{\mu}$, which means it does not fault or change the state. Thus for (27), using lemma 7.9, it is enough to prove the following:

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

(ii) for any τ, if $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \tau,\ \dot{\mu}\rangle$ then $\tau \models_{\varphi} Q$ and $\sigma \to \tau \models_{\varphi} \varepsilon$,

(iii) for all $\tau, \sigma', \tau', \pi$ if $\sigma' \models^{\Gamma}_{\varphi} P$ and $\mathit{Agree}(\sigma, \sigma', \varepsilon, \pi, \varphi)$
and $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \tau,\ \dot{\mu}\rangle$ and $\langle C,\ \sigma',\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \tau',\ \dot{\mu}\rangle$ then there is ρ with $\rho \supseteq \pi$ and $\mathit{Lagree}(\tau, \tau', \rho, \mathit{written}(\sigma, \tau) \cup \mathit{freshLocs}(\sigma, \tau))$.

To use the premises, we need a $\Phi, \Theta; \psi$-interpretation. The side condition $\theta \models \Phi, \Theta; \psi$ directly implies that $\varphi \cup \theta$ is a $\Phi, \Theta; \psi$-interpretation (see Definition 7.1). We prove (i)—(iii) using the following claim involving $\varphi \cup \theta$.

Claim A. For all $C', \sigma', \dot{\mu}'$ and m-truncated trace $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle C',\ \sigma',\ \dot{\mu}'\rangle$ we have $\langle C,\ \sigma,\ \mu\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^{*} \langle C',\ \sigma',\ \mu'\rangle$, where $\mu' = \dot{\mu}' \restriction m$. Also, if $C' = (y := m(z); D)$ for some y, z, D then $\sigma' \models_{\varphi\cup\theta} R^{x}_{z}$.

Note that $\models_{\varphi\cup\theta} R^{x}_{z}$ is the same as $\models_{\varphi} R^{x}_{z}$ because by well-formedness of context (Φ, Θ), the precondition R of m does not invoke m.

(i) Suppose $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle C',\ \sigma',\ \dot{\mu}'\rangle \overset{\varphi}{\longmapsto} \lightning$. If the part of this trace before faulting is m-truncated then we have $\langle C,\ \sigma,\ \mu\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^{*} \langle C',\ \sigma',\ \mu'\rangle$ by Claim A. In this case, from $\langle C',\ \sigma',\ \dot{\mu}'\rangle \overset{\varphi}{\longmapsto} \lightning$ we have by semantics $\mathit{Active}(C')$ is a field access/update. Thus by the special correspondence Lemma 7.5 we get $\langle C',\ \sigma',\ \mu'\rangle \overset{\varphi\cup\theta}{\longmapsto} \lightning$. But this contradicts the premise (25) for C. Now consider the case that the trace $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle C',\ \sigma',\ \dot{\mu}'\rangle$ is not m-truncated. By semantics it has a prefix of the form

$$\begin{aligned}
&\langle C,\ \sigma,\ \dot{\mu}\rangle \\
&\overset{\varphi}{\longmapsto}{}^{*} \langle y := m(z); D,\ \tau,\ \dot{\nu}\rangle \\
&\overset{\varphi}{\longmapsto} \langle B^{x,\mathsf{res}}_{x',\mathsf{res}'}; y := \mathsf{res}'; \mathsf{ecall}(x', \mathsf{res}'); D,\ \upsilon,\ \dot{\nu}\rangle \\
&\overset{\varphi}{\longmapsto}{}^{*} \langle A; \mathsf{ecall}(x', \mathsf{res}'); D,\ \sigma',\ \dot{\mu}'\rangle
\end{aligned}$$

where x' is a fresh variable and υ is $[\tau + x', \mathsf{res}'{:}\tau(z), \mathit{default}(U)]$. Moreover $\langle A,\ \sigma',\ \dot{\mu}'\rangle \overset{\varphi}{\longmapsto} \lightning$, because this was not a completed call. Notice that $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle m(z); D,\ \tau,\ \dot{\nu}\rangle$ is an m-truncated trace. So by Claim A, we have $\tau \models_{\varphi\cup\theta} R^{x}_{z}$ and thus $\upsilon \models_{\varphi} R^{x}_{x'}$. Therefore, we get $\langle B^{x,\mathsf{res}}_{x',\mathsf{res}'},\ \upsilon,\ \dot{\nu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \lightning$. But this contradicts correctness of $B^{x,\mathsf{res}}_{x',\mathsf{res}'}$, premise (26), using that valid judgments are closed under variable renaming.

(ii) Suppose $\langle C,\, \sigma,\, \dot{\mu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{skip},\, \tau,\, \dot{\mu}\rangle$. This is m-truncated, so by Claim A, we get $\langle C,\, \sigma,\, \mu\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^{*} \langle \mathsf{skip},\, \tau,\, \mu\rangle$. We assumed at the outset that $\sigma \models_{\varphi} P$, so $\sigma \models_{\varphi\cup\theta} P$ because P cannot mention m. Hence by premise (25) for C we have $\tau \models_{\varphi\cup\theta} Q$ and $\sigma{\rightarrow}\tau \models_{\varphi\cup\theta} \varepsilon$. Owing to well-formedness of the conclusion (27), all of P, Q, ε are well-formed in Φ. Thus $\models_{\varphi\cup\theta}$ is $\models_{\varphi}$ and $\mathit{wlocs}(\sigma,\varphi,\varepsilon) = \mathit{wlocs}(\sigma,\varphi\cup\theta,\varepsilon)$. Hence we have $\tau \models_{\varphi} Q$ and $\sigma{\rightarrow}\tau \models_{\varphi} \varepsilon$.

(iii) Suppose $\langle C,\, \sigma,\, \dot{\mu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{skip},\, \tau,\, \dot{\mu}\rangle$ and $\langle C,\, \sigma',\, \dot{\mu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{skip},\, \tau',\, \dot{\mu}\rangle$ Also suppose that there is a refperm π such that $\mathit{Agree}(\sigma,\sigma',\varepsilon,\pi,\varphi)$ and $\sigma' \models^{\Gamma}_{\varphi} P$. The traces are m-truncated. By Claim A, we have traces $\langle C,\, \sigma,\, \dot{\mu}\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^{*} \langle \mathsf{skip},\, \tau,\, \dot{\mu}\rangle$ and $\langle C,\, \sigma',\, \dot{\mu}\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^{*} \langle \mathsf{skip},\, \tau',\, \dot{\mu}\rangle$. Since $\mathit{rlocs}(\sigma,\varphi,\varepsilon) = \mathit{rlocs}(\sigma,\varphi\cup\theta,\varepsilon)$, we have $\mathit{Agree}(\sigma,\sigma',\varepsilon,\pi,\varphi\cup\theta)$. By the Read Effect part of premise (25) for C, there is refperm ρ such that $\rho \supseteq \pi$, $\mathit{Lagree}(\tau,\tau',\rho,\mathit{written}(\sigma,\tau)\cup\mathit{freshLocs}(\sigma,\tau))$.

To prove Claim A, we make the following somewhat intricate claim.

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

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

if the trace $\langle C_0,\, \sigma_0,\, \dot{\mu}_0\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle C',\, \sigma',\, \dot{\mu}'\rangle$ has exactly n completed topmost calls of m, and if we have a trace $\langle C,\, \sigma,\, \mu\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^{*} \langle C_0,\, \sigma_0,\, \mu_0\rangle$, then there is an m-truncated trace

$$\langle C,\, \sigma,\, \mu\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^{*} \langle C_0,\, \sigma_0,\, \mu_0\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^{*} \langle C',\, \sigma',\, \mu'\rangle,$$

where $\mu_0 = \dot{\mu}_0 \restriction m$ and $\mu' = \dot{\mu}' \restriction m$.

To prove Claim A, consider $\langle C_0,\, \sigma_0,\, \dot{\mu}_0\rangle$ to be $\langle C,\, \sigma,\, \dot{\mu}\rangle$. Using Claim B, from trace $\langle C,\, \sigma,\, \mathit{miu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle C',\, \sigma',\, \dot{\mu}'\rangle$, we get the requisite trace $\langle C,\, \sigma,\, \mu\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^{*} \langle C',\, \sigma',\, \mu'\rangle$. For the second part of Claim A, suppose $C' = (y := m(z); D)$ for some y, z, D. If, contrary to the claim, we have $\sigma' \not\models_{\varphi\cup\theta} R^{x}_{z}$ then by semantics of $y := m(z)$ and θ being a context interpretation (that is, condition $\theta \models \Phi, \Theta; \psi$ in (24)), we would have $\langle C',\, \sigma',\, \mu'\rangle \overset{\varphi\cup\theta}{\longmapsto} \lightning$ and hence $\langle C,\, \sigma,\, \mu\rangle \overset{\varphi\cup\theta}{\longmapsto} \lightning$. But this contradicts the premise (25) for C, since $\sigma \models_{\varphi\cup\theta} P$. So Claim A is proved.

To prove Claim B, we build the trace from $\langle C_0,\, \sigma_0,\, \dot{\mu}_0\rangle$ via $\overset{\varphi\cup\theta}{\longmapsto}$. Suppose the given m-truncated trace

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

has exactly n completed topmost calls of m in the second part. Using Lemma 7.7, we obtain intermediate states $\tau_i, \upsilon, \dot\sigma, \ddot\sigma, \sigma_i$ and environments $\dot\mu_i$ such that the following holds.[16]

$$
\begin{array}{ll}
\langle C_0,\, \sigma_0,\, \dot\mu_0\rangle & \\
\overset{\varphi}{\longmapsto}{}^*\langle y_1 := m(z_1); C_1,\, \tau_1,\, \dot\mu_1\rangle & \text{with no invocations of } m \\
\overset{\varphi}{\longmapsto} \langle B^{x,\mathsf{res}}_{x_1,\mathsf{res}_1}; y_1 := \mathsf{res}_1; \mathsf{ecall}(x_1,\mathsf{res}_1); C_1,\, \upsilon_1,\, \dot\mu_1\rangle & \text{where } \upsilon_1 = [\tau_1 + x_1, \mathsf{res}_1 \colon \tau_1(z_1), d] \\
 & \text{and } x_1, \mathsf{res}_1 \text{ are fresh and } d = \mathit{default}(U) \\
\overset{\varphi}{\longmapsto}{}^*\langle y_1 := \mathsf{res}_1; \mathsf{ecall}(x_1,\mathsf{res}_1); C_1,\, \dot\sigma_1,\, \dot\mu_1\rangle & \text{where } \langle B^{x,\mathsf{res}}_{x_1,\mathsf{res}_1},\, \upsilon_1,\, \dot\mu_1\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip},\, \dot\sigma_1,\, \dot\mu_1\rangle \\
\overset{\varphi}{\longmapsto} \langle \mathsf{ecall}(x_1,\mathsf{res}_1); C_1,\, \ddot\sigma_1,\, \dot\mu_1\rangle & \text{where } \ddot\sigma_1 = [\dot\sigma_1 \mid y_1 \colon \dot\sigma_1(\mathsf{res}_1)] \\
\overset{\varphi}{\longmapsto} \langle C_1,\, \sigma_1,\, \dot\mu_1\rangle & \text{where } \sigma_1 = \ddot\sigma_1 \restriction x_1 \restriction \mathsf{res}_1 \\
\vdots & \text{containing } n-1 \text{ topmost invocations of } m \\
\overset{\varphi}{\longmapsto} \langle C_n,\, \sigma_n,\, \dot\mu_n\rangle & \\
\overset{\varphi}{\longmapsto}{}^*\langle C',\, \sigma',\, \dot\mu'\rangle. & \text{with no completed topmost invocations of } m
\end{array}
$$

For any configurations $\langle A,\, \tau,\, \dot\mu\rangle$ and $\langle A',\, \tau',\, \mu\rangle$, we call them matching configurations iff $A = A'$, $\tau = \tau'$ and $\dot\mu = [\mu + m \colon (x : T, \mathsf{res} : U.B)]$ and hence $\mu = \dot\mu \restriction m$.

Below, using Lemma 7.8 we will construct a trace via $\overset{\varphi\cup\theta}{\longmapsto}$ that looks as follows:

$$
\begin{array}{lll}
\langle C_0,\, \sigma_0,\, \mu_0\rangle & & \\
\overset{\varphi\cup\theta}{\longmapsto}{}^*\langle y := m(z_1); C_1,\, \tau_1,\, \mu_1\rangle & \text{matching the configurations above, so } \mu_1 = \dot\mu_1 \restriction m & \\
\overset{\varphi\cup\theta}{\longmapsto} \langle C_1,\, \sigma_1,\, \mu_1\rangle & \text{a single step by Lemma 7.8 (2)} & (*) \\
\vdots & \text{containing } n-1 \text{ additional invocations of } m & \\
\overset{\varphi\cup\theta}{\longmapsto} \langle C_n,\, \sigma_n,\, \mu_n\rangle & & \\
\overset{\varphi\cup\theta}{\longmapsto}{}^*\langle C',\, \sigma',\, \mu'\rangle & \text{again matching configurations} &
\end{array}
$$

By induction on n, we prove that $\langle C_i,\, \sigma_i,\, \dot\mu_i\rangle$ and $\langle C_i,\, \sigma_i,\, \mu_i\rangle$ are matching configurations for $i = 1, 2, \ldots, n$ in two traces. In the base case of the induction, $n = 0$, all but one line of the given decomposed trace is empty. That is, we have $\langle C_0,\, \sigma_0,\, \dot\mu_0\rangle \overset{\varphi}{\longmapsto}{}^* \langle C',\, \sigma',\, \dot\mu'\rangle$ without any intermediate calls of m (but possibly a call in the last configuration). Using special correspondence Lemma 7.5 we can simply drop m from each environment to get a step by step matching trace $\langle C_0,\, \sigma_0,\, \mu_0\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^* \langle C',\, \sigma',\, \mu'\rangle$.

For the inductive case, $n > 0$, the initial steps $\langle C_0,\, \sigma_0,\, \dot\mu_0\rangle \overset{\varphi}{\longmapsto}{}^* \langle y := m(z_1); C_1,\, \tau_1,\, \dot\mu_1\rangle$ are matched as in the base case, up to the first invocation of m, in some state τ_1, environment $\dot\mu_1$, and with continuation C_1. At that point we have $\tau_1 \models_\varphi R^x_v$, where we let $v = \tau_1(z_1)$ —equivalently, $\tau_1 \models_{\varphi\cup\theta} R^x_v$— as otherwise we have a contradiction: We just established $\langle C_0,\, \sigma_0,\, \mu_0\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^* \langle y := m(z_1); C_1,\, \tau_1,\, \mu_1\rangle$, and if $\tau_1 \not\models_{\varphi\cup\theta} R^x_v$, then we get $\langle y := m(z_1); C_1,\, \tau_1,\, \mu_1\rangle \overset{\varphi\cup\theta}{\longmapsto} \lightning$. Furthermore, by hypothesis of Claim B we have $\langle C,\, \sigma,\, \mu\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^* \langle C_0,\, \sigma_0,\, \mu_0\rangle$. Putting these together we would obtain a faulting trace from $\langle C,\, \sigma,\, \mu\rangle$ via $\overset{\varphi\cup\theta}{\longmapsto}$. This contradicts premise (25) for C since $\sigma \models_\varphi P$ which gives us $\sigma \models_{\varphi\cup\theta} P$.

Having established $\tau_1 \models_\varphi R^x_v$, we get $\upsilon_1 \models_\varphi R^x_{x_1}$ by definition of υ_1. (In the trace for $\overset{\varphi}{\longmapsto}$ displayed above, υ_1 extends τ_1 with $x_1 : v$, that is, $x_1 : \tau_1(z_1)$.) By premise (26) for $B^{x,\mathsf{res}}_{x_1,\mathsf{res}_1}$, we have

[16] The names $\dot\mu_i$ indicate that each binds m to $(x : T, \mathsf{res} : U.B)$), but $\dot\sigma_i$ and $\ddot\sigma_i$ are fresh names with no significance beyond what is stated.

$\dot\sigma_1 \models_\varphi S^{x,\mathsf{res}}_{x_1,\mathsf{res}_1}$ (where $\dot\sigma_1$ is defined in trace for $\overset{\varphi}{\longmapsto}$ displayed above). We also have by premise (26) that $\upsilon_1{\to}\dot\sigma_1 \models \mathsf{wr}\,\mathsf{res}_1, \mathsf{rd}\,x_1, \eta^x_{x_1}$ and since η is wr-free this implies υ_1 and $\dot\sigma_1$ are identical except for res_1. Also σ_1 and τ_1 are identical except for y_1. Since res is not free in R we get $\dot\sigma_1 \models_\varphi R^x_{x_1}$ and hence $\dot\sigma_1 \models_\varphi (R \wedge S)^{x,\mathsf{res}}_{x_1,\mathsf{res}_1}$. From the side condition $\Phi;\psi \models R \wedge S \Rightarrow \mathsf{res} = m(x)$ (see (24)), we have $\dot\sigma_1(\mathsf{res}_1) = \theta(m)(\dot\sigma_1, \dot\sigma_1(x_1))$. and hence $\sigma_1(y_1) = \theta(m)(\tau_1, \tau_1(z_1))$. On the other hand, for $\overset{\varphi\cup\theta}{\longmapsto}$ the pure call rule in Figure 9 gives the step $\langle y := m(z); C_1,\ \tau_1,\ \mu_1\rangle \overset{\varphi\cup\theta}{\longmapsto} \langle C_1, [\tau_1 \mid y_1 : \theta(m)(\tau_1, \tau_1(z_1))],\ \mu_1\rangle$. The state $[\tau_1 \mid y_1 : \theta(m)(\tau_1, \tau_1(z_1))]$ is identical to σ_1 as defined above for the trace via $\overset{\varphi}{\longmapsto}$, justifying our use of σ_1 in the line marked (*) in the trace via $\overset{\varphi\cup\theta}{\longmapsto}$.

In conclusion, after the first call to m the traces reach matching configurations, namely $\langle C_1,\ \sigma_1,\ \dot\mu_1\rangle$ and $\langle C_1,\ \sigma_1,\ \mu_1\rangle$. What remains from $\langle C_1,\ \sigma_1,\ \dot\mu_1\rangle$ onward is a trace via $\overset{\varphi}{\longmapsto}$ with $n-1$ completed invocations of m, from a configuration reachable from $\langle C,\ \sigma,\ \dot\mu\rangle$ via $\overset{\varphi}{\longmapsto}$. Moreover, we have $\langle C,\ \sigma,\ \mu\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^* \langle C_1,\ \sigma_1,\ \mu_1\rangle$. So we can apply the inductive hypothesis to the trace $\langle C_1,\ \sigma_1,\ \dot\mu_1\rangle \overset{\varphi}{\longmapsto}{}^* \langle C',\ \sigma',\ \dot\mu'\rangle$, to conclude the proof of Claim B.

8. CASE STUDIES

To evaluate suitability of our approach for use in SMT-based tools we have used the Why3 verification system to fully verify examples including versions of the Cell and Composite examples. Why3 implements procedure-modular reasoning for a first-order intermediate language in which the heap can be explicitly encoded. These case studies are done by writing Why3 code and specifications that model the proof structure described in Section 7.3. Specifications in Why3 feature coarse-grained frame conditions. The finer-grained frame conditions of our logic are encoded semantically, that is, in postconditions and frame axioms. We also use assume statements, as described later. Why3 generates VCs for a range of theorem provers but in our work we rely only on SMT provers (alt-ergo, CVC4, and occasionally z3). The Why3 files are available online.[17]

The heap is represented in a standard way. References are an uninterpreted type. Each field is a map on pointers. The heap is a record with a mutable field for each of these maps, together with a map that encodes which references are allocated, and their types. Invariants are used for well-formedness conditions on the heap that would be ensured by type safety of a Java-like source language: fields have type correct values and there are no dangling pointers.

We use the 'ghost' marker of Why3 where possible, but in Why3 it means what we call 'specification-only', that is, ghost state cannot be explicitly assigned in Why3 code. Why3 features an 'invariant' declaration to be associated with a data type; these invariants are enforced on procedure call/return boundaries.

It is straightforward to encode a pair of heaps connected by a refperm (Section 4), using for the refperm a pair of maps subject to universally quantified axioms that one is inverse of the other. However, for the experiments in this section we only need identity refperms, because we are not checking read effects of impure methods (see remark following Definition 6.4). Whereas Definition 4.3 formalizes agreement and read effects in terms of the entire program state, our encoding focuses on the heap.

Why3 is carefully designed to make a clear distinction between mathematical definitions and programs. For each pure method p we therefore make three declarations:

— $pCode$ is the implementation of p, defined using Why3 **let** construct.
— $pUnInt$ is an uninterpreted function for which the specification is provided using axioms. Generally there is one axiom for the pre/post behavior and one axiom for the read effects in the frame condition. Uses of p in the specification of $pCode$ (and the specifications of other methods) is written using $pUnInt$.

[17] `http://www.cs.stevens.edu/~naumann/pub/readRLWhy3.tar`

— $pInt$ is a defined mathematical function. It serves as the "partial pure interpretations" with which one verifies that $pCode$ satisfies its specification (see Definition 5.1 and rule PURELINK in Figure 14). For this purpose, the implementation of $pCode$ begins with an **assume** statement that says $pUnInt$ is the same as $pInt$. On the other hand, client code is verified without access to the interpretation $pInt$.

Although the logic in this article does not formalize hiding of invariants, we do take some care with information hiding in the case studies. To do so, we take advantage of the Why3 module system, although that does not provide direct support for hiding of invariants.

8.1. Cell in Why3

The Cell example (Figures 1 and 15) is implemented in Why3 using one theory and three modules. A theory only contains logical definitions.

```
theory Pointer

  type pointer
  constant null: pointer
  type rType = Unalloc | Cell

end
```

All of our experiments in Why3 have such a theory, which describes the pointer structure of the heap. References are an uninterpreted type, called pointer to avoid confusion with native references in Why3. The type rType provides names for the types of allocated objects. In this example there is a single class, Cell, but in general there may be many.

Module Heap describes the heap structure.

```
type heap = {
        ghost mutable alloct : map pointer rType;
        mutable val_ : map pointer int;
        mutable foot : map pointer (set pointer);
}
```

A heap is a record. The first field, alloct, records the allocated pointers and their types. In a given heap, the set of p for which alloct[p] is not Unalloc corresponds to the value of variable alloc in our logic. Field val_ corresponds to the val field in Figure 1; the name avoids conflict with the Why3 keyword val.

Module Heap also declares, as a Why3 invariant, the condition I used in the specifications of the Cell methods in Figure 15.

```
invariant { forall p q: pointer. p <> q ->
                    (is_empty (inter self.foot[p] self.foot[q]) /\ not mem p self.foot[q]) }
```

Set intersection is defined in the Why3 library and named inter. In Why3, the keyword **self** refers to an instance of the type with which an invariant is associated, in this case Heap. As a Why3 invariant, the condition is included in the generated precondition of code using Heap, and as a postcondition of any code that may write the heap.

Module Heap also defines agreement, specialized to the fields of Cell to interpret effects that refer to the data group "any".

```
predicate agreeOnAny (h h': heap) (ftp: set pointer)
    = forall p: pointer . mem p ftp -> h.val_[p] = h'.val_[p] /\ h.foot[p] == h'.foot[p]
```

Module Cell gives the specifications for Cell class methods as in Figure 15. For pure method get the uninterpreted and interpreted functions are as follows.

```
function getUnInt (h: heap) (p: pointer): int

function getInt (h: heap) (p: pointer): int =
                  if h.alloct[p] = Cell then h.val_[p] else 0

axiom frameGet: forall h h': heap . forall p: pointer .
    agreeOnAny h h' h'.foot[p] -> getUnInt h p = getUnInt h' p

predicate getInterpreted =
    forall h: heap. forall p: pointer. h.alloct[p] = Cell -> getUnInt h p = getInt h p };

lemma frameGetInt: forall h h': heap. forall p: pointer. h.alloct[p] = Cell ->
    mem p h.foot[p] -> h'.alloct[p] = Cell -> agreeOnAny h h' h'.foot[p] ->
    getInt h p = getInt h' p
```

The axiom frameGet expresses the read effect rd self.$foot$`any. The predicate getInterpreted is used to connect the interpreted and uninterpreted functions. Lemma frameGetInt expresses the read effect of get method for getInt. Since the implementation of getCode(p) just returns h.val_[p], similar to the definition of getInt, this lemma expresses the read effect for the implementation. In Why3 there is no direct way of proving the read effect for the actual implementation.

For get, there is no axiom for pre-post behavior: it would be trivial since the postcondition (according to Figure 15) is just result = get(s).

Aside from the read effect, the rest of the specifications appear as Why3 contracts on the code.

```
let getCode (h: heap) (s: pointer) : int
        requires { h.alloct[s] = Cell }
        ensures { getUnInt h s = result }
=
        assume { getInterpreted };
        h.val_[s]

let setCode (h: heap) (s: pointer) (v: int) : unit
        requires { h.alloct[s] = Cell }
        ensures { getUnInt h s = v }
        writes { h.val_ }
        ensures { forall p: pointer . mem p h.foot[s] \/ agreeOnAny h (old h) (singleton p)
            }
=
        assume { getInterpreted };
        h.val_ <- set h.val_ s v
```

The precondition h.alloct[s] = Cell amounts to non-nullity of s plus a property ensured by typing of the source language. The Why3 frame condition for set says that any cell's value may be written. The postcondition following the **writes** clause expresses the semantics of our finer frame condition wr self.$foot$`any. The **assume** statements connect the uninterpreted function getUnInt, used in contracts, with its interpretation.

In addition to the constructor, we specify the behavior of the construct new $Cell$.

```
val newCell (h: heap) : pointer
        ensures { (old h).alloct[result] = Unalloc }
        ensures { h.alloct = Map.set (old h).alloct result Cell }
        ensures { forall p: pointer. (old h).alloct[p] = Cell -> result <> p }
        writes { h.alloct }
```

Finally, here is the constructor.

```
let init (h: heap) (s: pointer): unit
        requires { h.alloct[s] = Cell }
        ensures { mem s h.foot[s] }
        ensures { forall p: pointer. p <> s -> h.alloct[p] = Cell
                  -> not (mem p h.foot[s]) } // freshness of s.foot
        writes { h.foot}
        ensures { forall p: pointer. p <> s -> h.alloct[p] = Cell
                  -> h.foot[p] = (old h).foot[p] } // writes self.foot
=
        h.foot <- set h.foot s (singleton s)
```

The last module provides the client for Cell.

```
let main (h: heap) : unit
=
        let c = newCell h in init h c;
        let d = newCell h in init h d;
        setCode h c 5;
        assert { getUnInt h c = 5 }
        setCode h d 4;
        assert { getUnInt h c = 5 }
```

Note the absence of an assumption connecting getUnInt with the interpretation getInt. The splitting tactic provided by Why3 generates 14 goals that are automatically verified using z3 in 10.85 seconds. It relies on framing, and fails to verify if axiom frameGet is removed.

8.2. Composite in Why3

Next we consider the example in Figure 2. Where possible, our formalization uses annotations similar to those used by Rosenberg et al. [2010], to facilitate comparison.

The pointer structure for Composite is similar to the one for Cell. The only difference is that the type of an allocated pointer is Comp instead of Cell. Other than pointer theory, the Why3 file for Composite contains six modules. The first of these modules is Heap.

```
type heap = {
        ghost mutable alloct: map pointer rType;
        mutable chrn : map pointer (list pointer);
        mutable size : map pointer int;
        mutable parent: map pointer pointer;
}
```

As in the Cell experiment, alloct is a ghost field that keeps track of allocated pointers and their types. The map chrn represents list of children for a Composite node. The main difference between our version and that of Rosenberg et al. [2010] is that we don't need to consider the ghost field desc for keeping track of descendants of a Composite. Instead we define a pure method anc which computes the ancestors of a given Composite and use it directly to reason about the ancestors of each Comp node.

The second module called CompInvs contains a set of predicates working as invariants for Composite class. As is standard, our encoding defines a predicate (called okHeap and not shown here) that expresses invariants on heap structure that are ensured by type safety in Java-like languages. First, there are no dangling pointers. Second, the values in a field have their correct type. For integer field size the Why3 type suffices. But for parent and chrn the Why3 type merely constrains the value to be a pointer and a list of pointers respectively; our language in the case of parent ensures the additional condition that the pointer is either **null** or allocated. If Comp declared a field f of

reference type Thing, the Why3 field would be declared as a map from pointer to pointer and the invariant would say h.f[p]=**null** $\vee$ h.alloct[h.f[p]]=Thing. These heap invariants can be assumed in method bodies and need not appear in pre- and post-conditions. Readers familiar with BoogiePL will recognize the notion of "free requires" to express such language-based assumptions [Leino and Rümmer 2010] made explicit in an intermediate verification language.

Another bit of typing is for the list returned by ancestors: every element should be an allocated reference to a Comp; see AncUnInt_alloc below.

There are three other invariants in this module. In contrast to the Cell example, here we do not use the Why3 invariant declaration but make invariants explicit in contracts. This is a convenience: it is easier to keep track of the invariants by name in the Why3 IDE. Another reason for explicit invariants is hiding: we can choose which part of Why3 specification use which invariant. Using **invariant** clause would add the invariants to specifications of all methods in Composite module. This means that these internal invariants can be used to prove the client. The first invariant explains the relationship between a Composite and its parent and children. In the notation of Section 2 it is defined as follows.

$$\forall p : Comp \cdot p \neq \mathsf{null} \Rightarrow \begin{array}{l} (\forall q : Comp \cdot q \in p.chrn \Rightarrow q.parent = p) \\ \wedge (\forall s : Comp \cdot s = p.parent \Rightarrow p \in s.chrn) \\ \wedge p \notin p.chrn \wedge nodup\ p.chrn \end{array}$$

where $nodup$ checks that $p.chrn$ does not have any duplicates. The second invariant specifies the $size$ field.

$$\forall p : Comp \cdot p \neq \mathsf{null} \Rightarrow (p.size > 0 \wedge p.size = 1 + (\Sigma_{q \in p.chrn} \cdot q.size))$$

To be able to define the summation above, we need two extra functions in Why3, which are simple recursive functions on lists and we omit them here. In Why3 code these two invariants are called ptcdInv and sizeInv respectively.

The last part of CompInvs module is a mathematical definition of the list of ancestors. Here we first give the mathematical definition and then show its relation to the actual implementation in Why3. For any $p : Comp$ where $p \neq \mathsf{null}$ define

$$f(p) = \begin{cases} Cons\ p\ Nil & p.parent = \mathsf{null} \\ Cons\ p\ f(p.parent) & p.parent \neq \mathsf{null} \end{cases}.$$

Note that with this definition p is an ancestor of itself. This recursive definition cannot be implemented in Why3, since it may not terminate if there are cycles via parent. Instead one can define it as an inductive predicate is_loa that is well founded in the list argument [Reynolds 2002]. The predicate is_loa h p q l indicates that l is the list of ancestors of p until q in heap h.

```
inductive is_loa (heap) (pointer) (pointer) (list pointer) =
  |Nil_l: forall h: heap, p: pointer. ptcdInv h -> h.alloct[p] = Comp ->
      h.parent[p] = null -> is_loa h p h.parent[p] (Cons p Nil)
  |Tree: forall h: heap, p q: pointer, l: list pointer. ptcdInv h -> h.alloct[p] = Comp ->
      h.alloct[h.parent[p]] = Comp -> okPtr h q -> h.parent[p] <> q ->
      is_loa h h.parent[p] q l -> is_loa h p q (Cons p l)
```

The predicate okPtr h p above indicates that p is a value of type Comp, that is, p is either null or allocated in heap h. Now consider the function anc defined as follows.

```
function anc (h: heap) (p: pointer): (list pointer)

axiom anc_def: forall h: heap, p: pointer. okPtr h p -> is_loa h p null (anc h p)
```

The axiom anc_def defines the ancestors function indirectly. This indirect definition is useful for inductive proofs in the CompInduct module. To make sure that anc is well-defined, we have a lemma that shows the uniqueness of a list which satisfies is_loa predicate for given h, p and q.

Verification of the client does not rely on module CompInvs, so it does not depend on any internal invariants of Composite class. This is how we implement information hiding.

The next two modules called SizeInvs and CompInduct are collections of lemmas needed for reasoning about the size field and ancestors function. Most of them are written using the **let rec lemma** statement which is a way of expressing induction proofs in Why3.

The fifth module is Composite. This is the main module of Composite class. For size the code is as follows.

```
axiom sizeUIN_rd: forall h h': heap, s: pointer. h.alloct[s] = Comp ->
  h'.alloct[s] = Comp -> h.size[s] = h'.size[s] -> sizeUnInt h s = sizeUnInt h' s

function sizeInt (h: heap) (s: pointer): int =
            if h.alloct[s] = Comp then h.size[s] else 0

lemma sizeInt_rd: forall h h': heap, s: pointer. h.alloct[s] = Comp ->
  h'.alloct[s] = Comp -> h.size[s] = h'.size[s] -> sizeInt h s = sizeInt h' s

predicate sizeInterpreted =
  forall h: heap, p: pointer. h.alloct[p] = Comp -> sizeUnInt h p = sizeInt h p

let sizeCode (h: heap) (s: pointer): int
    requires{ h.alloct[s] = Comp }
    ensures{ result = sizeUnInt h s }
=
    assume { okHeap h };
    assume { ptcdInv h };
    assume { sizeInv h };
    assume { sizeInterpreted };
    h.size[s]
```

The two functions sizeUnInt and sizeInt are the uninterpreted and interpreted $size$ function, respectively. The axiom sizeUIN_rd is the read effect for $size$ in terms of sizeUnInt. Using Region Logic effect syntax, this is rd self.$size$. Lemma sizeInt_rd shows that read effect is correct for the given interpretation. Since the implementation (sizeCode) only returns the size as well, this shows the read effect for the implementation. The assumed predicate sizeInterpreted connects uninterpreted size function with its interpretation for use in the body of methods which use sizeUnInt in their specification. The method for $size$ which is an implementation of sizeInt is sizeCode. It only returns the value $p.size$.

In the same order the method anc is presented.

```
axiom ancUnInt_alloc: forall h: heap, p q: pointer. h.alloct[p] = Comp ->
  mem q (ancUnInt h p) -> h.alloct[q] = Comp

axiom ancUnInt_rd: forall h h': heap, p: pointer. h.alloct[p] = Comp ->
  h'.alloct[p] = Comp -> (forall q: pointer. mem q (ancUnInt h p) ->
  h.parent[q] = h'.parent[q]) -> ancUnInt h p = ancUnInt h' p

function ancInt (h: heap) (p: pointer): list pointer
          = if h.alloct[p] = Comp then anc h p else Nil

let rec lemma ancInt_rd (h h': heap) (p: pointer)
```

```
        requires { h.alloct[p] = Comp /\ h'.alloct[p] = Comp }
        requires { forall q: pointer. mem q (ancInt h p) -> h.parent[q] = h'.parent[q] }
        requires { okHeap h /\ okHeap h' }
        requires { ptcdInv h /\ ptcdInv h' }
        ensures { ancInt h p = ancInt h' p }
        variant { length (ancInt h p) }
    =
        if h.parent[p] <> null then
            (assert { ancInt h p = Cons p (ancInt h h.parent[p]) };
             ancInt_rd h h' h.parent[p])


    predicate ancInterpreted = forall h: heap, p: pointer. okPtr h p ->
        ancUnInt h p = ancInt h p

    let rec ancCode (h: heap) (s: pointer): list pointer
        requires{ h.alloct[s] = Comp }
        ensures{ result = ancUnInt h s }
        variant { ancUnInt h s }
    =
        assume { okHeap h /\ ptcdInv h /\ sizeInv h /\ ancInterpreted };
        if h.parent[s] = null then
            Cons s Nil
        else
            Cons s (ancCode h h.parent[s])
```

As in the part for $size$ method, the two functions ancUnInt and ancInt are the uninterpreted and interpreted anc function, respectively. The axiom ancUnInt_rd is the read effect for ancUnInt. This read effect is rd$(self.anc())$‘$parent$ in RL syntax. To use the read effect, we need to know that any reference in ancUnInt list is already allocated. But since this function is not defined, we give this property as an axiom before its read effect, called ancUnInt_alloc. Lemma ancInt_rd is an inductive proof that shows read effect is correct for ancInt. Since the implementation of ancestor method, namely ancCode, is very close to the definition of ancInt, this shows that the read effect is correct for the implementation. As is the case of read effect for $size$ method and get method in Cell experiment, in Why3 there is no way to prove the read effect property for the actual implementation. The predicate ancInterpreted connects the interpretation of anc to ancUnInt function, similar to sizeInterpreted. The implementation of anc is given as a recursive function. We verify that it returns the same list of ancestors as ancUnInt function.

A private method addtosizeCode is defined after that (not shown). For a given heap h, pointer s and integer v this method just adds v to the size of ancestors of s. This method breaks sizeInv, but clients cannot call it directly. It is only used by addCode so a full functional spec is appropriate, which mentions private fields.

The last method in Composite module is addCode.

```
let addCode (h: heap) (s x: pointer) : unit
        requires { h.alloct[x] = Comp /\ h.alloct[s] = Comp }
        requires { h.parent[x] = null /\ not (mem x (ancUnInt h s)) }
        ensures { h.parent[x] = s }
        writes { h.parent, h.size, h.chrn }
        ensures { forall p: pointer. h.alloct[p] = Comp ->
            not (mem p (ancUnInt (old h) s)) -> h.size[p] = (old h).size[p] }
        ensures { forall p: pointer. h.alloct[p] = Comp -> x <> p ->
            h.parent[p] = (old h).parent[p] }
```

```
=
assume { okHeap h /\ ptcdInv h /\ sizeInv h /\ sizeInt_UnInt /\ ancInt_UnInt };
        let l = ancCode h s in
                h.chrn <- set h.chrn s (Cons x h.chrn[s]);
                assert{ l = ancInt h s };
                h.parent <- set h.parent x s;
                assert{ l = ancInt h s };
                addtosizeCode h s h.size[x];
                assert{ l = ancInt h s /\ sizeInv h /\ ptcdInv h }
```

For a heap h and two pointers x and s, this method adds x as a child of s, provided that parent of x be null and x is not in the ancestors of s. The effects of this method using the syntax of RL are wr $self$`any, wr $self.anc()$`$size$ and wr $x.parent$. Note that this method writes $self.chrn$, to hide the internal structure of Composite from the client, we use wr $self$`any in RL. Why3 does not have the abstraction, any, so we are forced to have $h.chrn$ in the writes of this method. In the body of addCode the variable l is used to guide the provers to the fact that ancestors of s are not changing. The invariants of Composite class are assumed at the beginning of body and asserted at the end.

The last module is ClientOfComposite containing the client. The client code is as follows.

```
let main (h: heap) (b c d: pointer) : unit
        requires { h.alloct[b] = Comp /\ h.alloct[c] = Comp /\ h.alloct[d] = Comp }
        requires { not mem d (ancUnInt h b) }
        requires { not mem c (ancUnInt h d) }
        requires { h.parent[c] = null /\ not (mem c (ancUnInt h b)) }
        ensures { sizeUnInt h d = sizeUnInt (old h) d }
        ensures { ancUnInt h d = ancUnInt (old h) d }
    =
        addCode h b c;
```

The client just ensures that adding c as a child of b does not affect the size and ancestors of d, provided that all three references are allocated, c is not an ancestor of both d and b, and d is not an ancestor of b. And also c can be added as a child of b. Using the modules system of Why3, we make sure that this module only has access to Heap and Composite. This means that the internal structure and the invariants of Composite class are hidden from the client as in Figure 2. For the pure methods, the interpretations are syntactically visible in the client module, but there is no assumption connecting them with the uninterpreted methods; So the client verification relies only on the uninterpreted versions and their contracts. This verifies automatically in 184.23 seconds using the 'split' tactic and the provers Alt-ergo, CVC4 and z3.

8.3. Remarks on deriving axioms from specifications

In the formalization of this paper, we treat $(\Phi;\psi)$-validity semantically, but it is possible to derive suitable formulas to interpret the specifications. Indeed, this is done in some tools that allow pure methods in specifications. Consider a well-formed context Φ with $\Phi(p) = (x{:}T, \mathsf{res}{:}U)P \rightsquigarrow Q\,[\eta]$. Because no write effects are allowed, the postcondition is effectively a constraint on res. Both pre- and post-condition will be applied to the same state, but for the addition of res. Define $phyp((x{:}T, \mathsf{res}{:}U)P \rightsquigarrow Q\,[\eta])$ to be the formula $(\forall x{:}T, \mathsf{res} : U \;\cdot\; P \Rightarrow Q)$. Define $phyp(\Phi)$ to be the conjunction of $phyp(\Phi(p))$ for pure p in $dom(\Phi)$. Observe that $phyp(\Phi)$ is Φ-valid.

For P to be Φ-valid amounts to its being entailed by $phyp(\Phi)$. The other information about pure methods provided by Φ is their read effects.

9. RELATED WORK

We take the Cell example from the most closely related work, Smans et al. [2010], 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. However their implementation (p.453 of the paper) 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, for example Darvas and Müller [2006], 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.

Our use of explicit ghost state in read and write effects is directly inspired by the state-dependent frame conditions of Kassios [Kassios 2011] who terms them dynamic frames. Kassios's frames predicate f **frames** v (where f is an expression that denotes a location set) says that the current value of v depends only on the locations in f. Thus if there are no writes to the locations denoted by f, the meaning of v is preserved. In Kassios's terminology, this leads to a notion of "disjointness of frames" akin to what is expressed by our FRAME rule: if f is the set of locations on which x depends, g is the set of locations on which y depends, f, g are disjoint and we know that only f is modified, then the value of y is preserved. Kassios introduces self-framing frames to reason about the preservation of disjointness in connection with allocation. Suppose a dynamic frame g frames itself, i.e., g **frames** g, In a state where f and g denote disjoint sets of locations, if the state is modified only by writing locations in f, in such a way that the value of f does not gain any previously allocated locations, then the disjointness of f and g is preserved.

In our work, Kassios's frames predicate and its properties are embodied in the frames judgment as well as the notion of immunity (Lemma 6.9). But in contrast to Kassios, Smans et al., and other related work we also investigate read effects of commands and impure procedures. In RL, reasoning about effects for sequentially composed and iterated commands relies on what amounts to a frame rule for frame conditions, which in RLI appears as an immunity condition for write effects in the proof rules SEQ and WHILE. With the addition of read effects for commands, we found that in addition to immunity we need read effects to be self-framed. As our formulation is somewhat different from what is found in the literature, we adopted a different term, 'read framed'.

Framing in separation logic encompasses read and write effects, implicitly in syntax but explicitly in the semantics; e.g., safety monotonicity, frame property in O'Hearn et al. [2009] where the second order frame rule is introduced. Permission-based separation logic allows distinguishing read effects from write effects [Bornat et al. 2005]. Parkinson and Summers [2012] explores the connection between this logic and one based on implicit dynamic frames (IDF) [Smans et al. 2012]. The IDF logic is a first-order logic extended with accessibility predicates that mediate access to heap locations. The logic is the foundation of Chalice [Leino and Müller 2009], an SMT-based tool for verification of multi-threaded programs. A critical concept used to forge the connection is that of a self-framing assertion. Such an assertion provides a thread with adequate permissions to validate the assertion: interference —that is, potential modifications of heap locations in the assertion by other threads—is impossible. All separation logic assertions are proven to be self-framing. Chalice uses a syntactic definition of self-framing. These works do not address read effects of commands. Our work does not address concurrency.

Relational semantics for effects is explored by [Benton et al. 2007]. Their interpretation of both read and write effects quantifies over different classes of relations that are preserved. By contrast, other work including this article treats dependency in terms of preservation of specific relations.

Benton et al. [2014] consider a denotational semantics of a region-based type and effect system that supports observational purity [Naumann 2007]. The semantics uses a novel variant of logical relations (setoids) that allows clients of a module to validate a number of effect-based program equivalences. The encoding of the semantics for practical use in an SMT-based verifier is not evident.

The abstract predicates approach [Parkinson and Bierman 2005] 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 [Heule et al. 2013]. Another gives a detailed semantics and soundness proof for VCs that provide effective reasoning about recursively defined abstract predicates and abstraction functions [Summers and Drossopoulou 2013]. The latter works have extensive pointers to related work.

In RLI/II an additional "effect" is included, to specify that some references are freshly allocated. Freshness is needed in the proof rules for sequence and loops, to account for writes to freshly allocated objects. But freshness can be expressed in postconditions, so in this paper we do not include freshness effects.

Bao et al. [2015] develop a variation on region logic in which a 'region' is a set of locations, and in which conditional expressions can be used in frame conditions. With the addition of pure methods in the present article, we get some ability to express conditional effects, e.g., the effect $\mathsf{wr}\, p()\text{`}f$ where pure method $p$ returns a region and has postcondition that describes the region conditioned on the pre-state. In RL including the present paper, sets of heap locations are expressed in terms of reference sets (denoted by region expressions $G$) and image expressions like $G\text{`}f$ which are in some sense rectangular. For a finite non-rectangular collection of locations we can just use singletons, say $\{x_0\}\text{`}f_0, \ldots, \{x_n\}\text{`}f_n$, but this falls short of an unbounded collection of references with non-uniform fields. (Perhaps more likely in practice would be an unbounded collection of array references paired with differing indexes.) Use of location sets provides a way to abstract from field names. This is featured in Smans et al. [2010]. Explicit use of fields in RL provides simple syntactic means to establish many disjointnesses. Arguably, data groups are a sufficient means of absracting from field names. The significance of these expressive differences is perhaps best evaluated in connection with abstraction and information hiding.

10. CONCLUSION

We have formalized and proved sound a logic for object-based programs with dynamic allocation, with two unusual features in correctness judgments: pure methods in formulas and read effects in frame conditions for commands. Building on RL, effects are expressed flexibly by means of state dependent expressions typically involving ghost state. A key feature is the frame rule, which says a predicate is preserved by a command if the predicate's read effect is separated from the command's write effect. Additional features —immunity and framed reads— provide what amounts to framing of frame conditions in sequential execution (sequences and loops). Correctness judgments include hypotheses, to formalize assumed method specifications. The semantics is given in terms of conventional operational semantics, together with partial interpetations that model axioms used in prior work on verification conditions for pure methods. The linking rules discharge hypotheses, fully grounding correctness proofs in the operational semantics.

One intended use of the logic is as a stepping stone towards specification and verification using observationally pure methods. The other intended use of the logic is to guide the design (and potentially, the verification of) semi-automated verification tools based on SMT provers and verification condition generators. Modular verification tools implement, in effect, linking rules, though this is not usually explicit in the literature on VC generation. By contrast with linking rules, other proof rules are not directly implemented, but rather guide the design of VC generation, as well as the design of annotation features (loop invariants, frame conditions, and the like). For example, our results suggest that for reasoning about read effects of commands, frame conditions should be read framed. (Instead of imposing that as a restriction on specifications it can be left implicit, as it is a syntactic

closure that can be applied automatically.) In a VC setting, the role of the FRAME rule is played by a combination of framing axioms and heuristic deployment of assertions to trigger use of those axioms to achieve automated modular verification.

Read effects are a dependency property, for which the appropriate extensional semantics is expressed in terms of two runs of the program. A state dependent read effect denotes a set of locations in one of the two initial states. This seeming asymmetry works in part because a correcness judgment quantifies over all pairs of runs, and in part owing to restriction to 'framed reads' which restores symmetry where it is needed. Relational Hoare logics have been developed to reason about dependency properties [Benton 2004; Yang 2007; Amtoft et al. 2006]. In ongoing work we are developing a relational version of the logic. Read effects of commands play a crucial role in that logic, which is one reason they are included in this article. For future work, the next steps towards observational purity will be (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.

The linking rules in this article are proved sound only in the absence of recursive procedures. We see no difficulty proving soundness allowing recursion, in the same way as in RLII, but have not checked the details. The intricacy of the details could make it an interesting exercise for machine checking.

Another future work is to add pure methods, and read effects for commands, to a verifier. One candidate is our prototype SMT-based verifier for RL [Rosenberg et al. 2010; Rosenberg et al. 2012], which already provides limited support for pure function definitions with framing, based on a version of Leino's Dafny.

REFERENCES

AMTOFT, T., BANDHAKAVI, S., AND BANERJEE, A. 2006. A logic for information flow in object-oriented programs. In *ACM Symposium on Principles of Programming Languages*. 91–102.

BANERJEE, A. AND NAUMANN, D. A. 2013. Local reasoning for global invariants, part II: Dynamic boundaries. *Journal of the ACM 60,* 3, 19:1–19:73.

BANERJEE, A. AND NAUMANN, D. A. 2014. A logical analysis of framing for specifications with pure method calls. In *Verified Software: Theories, Tools and Experiments – 6th International Conference, Revised Selected Papers*. LNCS, vol. 8471. 3–20.

BANERJEE, A., NAUMANN, D. A., AND ROSENBERG, S. 2013. Local reasoning for global invariants, part I: Region logic. *Journal of the ACM 60,* 3, 18:1–18:56.

BAO, Y., LEAVENS, G. T., AND ERNST, G. 2015. Conditional effects in fine-grained region logic. In *Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs (FTfJP)*. 5:1–5:6.

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

BENTON, N., HOFMANN, M., AND NIGAM, V. 2014. Abstract effects and proof-relevant logical relations. In *ACM Symposium on Principles of Programming Languages*. 619–632.

BENTON, N., KENNEDY, A., BERINGER, L., AND HOFMANN, M. 2007. Relational semantics for effect-based program transformations with dynamic allocation. In *International Symposium on Principles and Practice of Declarative Programming*. 87–96.

BOBOT, F. AND FILLIÂTRE, J.-C. 2012. Separation predicates: A taste of separation logic in first-order logic. In *International Conference on Formal Engineering Methods*. LNCS, vol. 7635. 167–181.

BORNAT, R., CALCAGNO, C., O'HEARN, P. W., AND PARKINSON, M. J. 2005. Permission accounting in separation logic. In *ACM Symposium on Principles of Programming Languages*. 259–270.

DARVAS, A. AND MÜLLER, P. 2006. Reasoning about method calls in interface specifications. *Journal of Object Technology (JOT)*, 59–85.

HEULE, S., KASSIOS, I. T., MÜLLER, P., AND SUMMERS, A. J. 2013. Verification condition generation for permission logics with abstract predicates and abstraction functions. In *European Conference on Object-Oriented Programming*. LNCS, vol. 7920. 451–476.

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

KASSIOS, I. T. 2011. The dynamic frames theory. *Formal Aspects of Computing 23,* 3, 267–288.

KRISHNASWAMI, N. R., ALDRICH, J., AND BIRKEDAL, L. 2010. Verifying event-driven programs using ramified frame properties. In *ACM workshop on Types In Languages Design And Implementation*. 63–76.

LEAVENS, G. T., BAKER, A. L., AND RUBY, C. 2006. Preliminary design of JML: a behavioral interface specification language for java. *ACM SIGSOFT Software Engineering Notes 31,* 3, 1–38.

LEAVENS, G. T. AND NAUMANN, D. A. 2015. Behavioral subtyping, specification inheritance, and modular reasoning. *ACM Trans. Program. Lang. Syst. 37,* 4, 13.

LEINO, K. R. M. AND MÜLLER, P. 2009. A basis for verifying multi-threaded programs. In *Programming Languages and Systems, European Symposium on Programming*. 378–393.

LEINO, K. R. M., POETZSCH-HEFFTER, A., AND ZHOU, Y. 2002. Using data groups to specify and check side effects. In *ACM Conf. on Program. Lang. Design and Implementation*. 246–257.

LEINO, K. R. M. AND RÜMMER, P. 2010. A polymorphic intermediate verification language: Design and logical encoding. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. LNCS, vol. 6015. 312–327.

ROBBY ET AL. 2008. Seventh International Workshop on Specification and Verification of Component Systems (SAVCBS). Tech. Rep. CS-TR-08-07, School of Electrical Engineering and Computer Science, University of Central Florida.

NANEVSKI, A., AHMED, A., MORRISETT, G., AND BIRKEDAL, L. 2007. Abstract predicates and mutable adts in hoare type theory. In *Programming Languages and Systems, European Symposium on Programming*. LNCS, vol. 4421. 189–204.

NANEVSKI, A., BANERJEE, A., AND GARG, D. 2013. Dependent type theory for verification of information flow and access control policies. *ACM Trans. Program. Lang. Syst. 35,* 2, 6.

NAUMANN, D. A. 2007. Observational purity and encapsulation. *Theor. Comput. Sci. 376,* 3, 205–224.

O'HEARN, P. W., YANG, H., AND REYNOLDS, J. C. 2009. Separation and information hiding. *ACM Trans. Program. Lang. Syst. 31,* 3, 1–50.

PARKINSON, M. J. AND BIERMAN, G. M. 2005. Separation logic and abstraction. In *ACM Symposium on Principles of Programming Languages*. 247–258.

PARKINSON, M. J. AND SUMMERS, A. J. 2012. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science 8,* 3.

REYNOLDS, J. C. 2002. Separation logic: A logic for shared mutable data structures. In *IEEE Symp. on Logic in Computer Science*. 55–74.

ROSENBERG, S., BANERJEE, A., AND NAUMANN, D. A. 2010. Local reasoning and dynamic framing for the composite pattern and its clients. In *Verified Software: Theories, Tools, Experiments*. LNCS, vol. 6217. 183–198. Software distribution at `http://www.cs.stevens.edu/~naumann/pub/VERL/`.

ROSENBERG, S., BANERJEE, A., AND NAUMANN, D. A. 2012. Decision procedures for region logic. In *Int'l Conf. on Verification, Model Checking, and Abstract Interpretation*. LNCS, vol. 7148. 379–395.

SMANS, J., JACOBS, B., AND PIESSENS, F. 2012. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst. 34,* 1, 2.

SMANS, J., JACOBS, B., PIESSENS, F., AND SCHULTE, W. 2010. Automatic verification of Java programs with dynamic frames. *Formal Aspects of Computing 22,* 3-4, 423–457.

SUMMERS, A. J. AND DROSSOPOULOU, S. 2013. A formal semantics for isorecursive and equirecursive state abstractions. In *European Conference on Object-Oriented Programming*. LNCS, vol. 7920. 129–153.

YANG, H. 2007. Relational separation logic. *Theor. Comput. Sci. 375,* 1-3, 308–334.

A. ADDITIONAL SOUNDNESS PROOFS

A.1. Technical results needed to prove ImpureLink

Were it not for our aim to address the small step encapsulation property of RLII, we would define correctness in terms of a big-step semantics derived from the transition semantics. We do in fact need that definition, for use in proving IMPURELINK.

Definition A.1 (*denotation of command*).
Suppose C is swf in Γ and θ is a candidate interpretation for Γ. Define $[\![\Gamma \vdash C]\!]_\theta$ to be the function of type $[\![\Gamma]\!] \to \mathbb{P}([\![\Gamma]\!] \cup \{\lightning\})$ defined by

$$[\![\Gamma \vdash C]\!]_\theta(\sigma) = \{\tau \mid \langle C, \sigma, \_\rangle \overset{\theta}{\longmapsto}{}^* \langle \mathsf{skip}, \tau, \_\rangle\} \cup (\{\lightning\} \text{ if } \langle C, \sigma, \_\rangle \overset{\theta}{\longmapsto}{}^* \lightning \text{ else } \varnothing)$$

where $\_$ is the empty method environment.

The next two lemmas are adapted from [Banerjee and Naumann 2013] Lemma 5.3 and Lemma 5.5. They describe how a trace can be decomposed into convenient segments.

LEMMA A.2 (DECOMPOSITION FOR IMPURE ENVIRONMENT METHODS). Let φ be a Φ-interpretation. Suppose $\mu_0(m) = (x : T.B)$ and $\langle C_0, \sigma_0, \mu_0\rangle$ is compatible with $\Phi;\varphi$, where

$m \notin \mathit{dom}\,\varphi$. Suppose $\langle C_0, \sigma_0, \mu_0\rangle \overset{\varphi}{\longmapsto}{}^* \langle D, \tau, \nu\rangle$. Then there is $n \geq 0$ and there are configurations $\langle C_i, \sigma_i, \mu_i\rangle$, variables z_i and x_i, states τ_i, υ_i and $\dot\sigma_i$ such that for all i $(0 < i \leq n)$

(1) $\langle C_{i-1}, \sigma_{i-1}, \mu_{i-1}\rangle \overset{\varphi}{\longmapsto}{}^* \langle m(z_i); C_i, \tau_i, \mu_i\rangle$ without any intermediate configurations in which m is the active command
(2) $\langle m(z_i); C_i, \tau_i, \mu_i\rangle \overset{\varphi}{\longmapsto} \langle B^x_{x_i}; \mathsf{ecall}(x_i); C_i, \upsilon_i, \mu_i\rangle$
and $\upsilon_i = [\tau_i + x_i\colon \tau_i(z_i)]$ (note that x_i is fresh parameter names)
(3) $\langle B^x_{x_i}, \upsilon_i, \mu_i\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip}, \dot\sigma_i, \mu_i\rangle$ and hence by semantics
$\langle B^x_{x_i}; \mathsf{ecall}(x_i); C_i, \upsilon_i, \mu_i\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{ecall}(x_i); C_i, \dot\sigma_i, \mu_i\rangle$
(4) $\langle \mathsf{ecall}(x_i); C_i, \dot\sigma_i, \mu_i\rangle \overset{\varphi}{\longmapsto} \langle C_i, \sigma_i, \mu_i\rangle$ and $\sigma_i = \dot\sigma_i \restriction x_i$
(5) $\langle C_n, \sigma_n, \mu_n\rangle \overset{\varphi}{\longmapsto}{}^* \langle D, \tau, \nu\rangle$ without any completed invocations of m —but allowing a topmost call that is incomplete.

LEMMA A.3 (DECOMPOSITION FOR IMPURE INTERPRETED METHODS). Suppose that μ is method environment such that $m \notin \mathit{dom}\,\mu$ and $\langle C_0, \sigma_0, \mu_0\rangle$ is compatible with $\Phi;\varphi$, where $m \in \mathit{dom}\,\varphi$. Also, suppose $\langle C_0, \sigma_0, \mu_0\rangle \overset{\varphi}{\longmapsto}{}^* \langle D, \tau, \nu\rangle$. Then there is $n \geq 0$ and there are configurations $\langle C_i, \sigma_i, \mu_i\rangle$, variables z_i and states τ_i such that for all i $(0 < i \leq n)$

(1) $\langle C_{i-1}, \sigma_{i-1}, \mu_{i-1}\rangle \overset{\varphi}{\longmapsto}{}^* \langle m(z_i); C_i, \tau_i, \mu_i\rangle$ without any intermediate configurations in which m is the active command
(2) $\langle m(z_i); C_i, \tau_i, \mu_i\rangle \overset{\varphi}{\longmapsto} \langle C_i, \sigma_i, \mu_i\rangle$ and $\sigma_i \in \varphi(m)(\tau_i, \tau_i(z_i))$
(3) $\langle C_n, \sigma_n, \mu_n\rangle \overset{\varphi}{\longmapsto}{}^* \langle D, \tau, \nu\rangle$ without any completed invocations of m —but allowing a topmost call that is incomplete.

LEMMA A.4 (JUDGMENT RENAMING). If $\Phi;\varphi \models^{\Gamma, x:T} C : P \rightsquigarrow P'\,[\varepsilon]$ and $\Gamma, y : T$ is well formed, then $\Phi;\varphi \models^{\Gamma, y:T} C^x_y : P^x_y \rightsquigarrow P'^x_y\,[\varepsilon^x_y]$.

THEOREM A.5 (SOUNDNESS OF IMPURE LINK). *Suppose that* Θ *is* $m : (x\colon T)R \rightsquigarrow S\,[\eta]$, $m \notin B$, *and*

$$\Phi;\psi \models^{\Gamma, x:T} B : R \rightsquigarrow S\,[\mathsf{rd}\,x, \eta] \text{ and } \Phi,\Theta;\psi \models^{\Gamma} C : P \rightsquigarrow Q\,[\varepsilon]$$

then

$$\Phi;\psi \models^{\Gamma} \mathsf{let}\ m(x\colon T) = B\ \mathsf{in}\ C : P \rightsquigarrow Q\,[\varepsilon].$$

PROOF. Let μ be a Γ-environment, φ be a Φ-interpretation such that $\psi \subseteq \varphi$ and σ be a state such that $\sigma \models^{\Gamma, \mathit{sigs}(\Phi)}_{\varphi} P$. For $\mathsf{let}\ m(x\colon T) = B\ \mathsf{in}\ C$ via $\overset{\varphi}{\longmapsto}$, we must show Safety, Post, Write Effect and Read Effect. We need to reason about executions via $\overset{\varphi}{\longmapsto}$ but the premises of pertain to execution via Φ,Θ-interpretations that extend ψ. To bridge this gap we extend φ by defining a candidate Θ-interpretation θ as follows. Let $\theta(m)$ be the map of type $(\upsilon \in [\![\Gamma]\!]) \times [\![T]\!]\upsilon \to \mathbb{P}([\![\Gamma]\!] \cup \{\lightning\})$ defined, for all $\tau \in [\![\Gamma]\!]$ and $v \in [\![T]\!]\tau$, as follows.

— If $\tau \not\models_\varphi R^x_v$ then $\theta(m)(\tau, v) = \{\lightning\}$
— If $\tau \models_\varphi R^x_v$ then $\theta(m)(\tau, v) = [\![\mathit{sigs}(\Phi,\Theta), \Gamma \vdash B]\!]_\varphi([\tau + x\colon v])$.

The second clause uses Def. A.1.[18] It is immediate that $\varphi \cup \theta$ extends ψ. It is straightforward to check that $\psi \cup \theta$ is a Φ,Θ-interpretation. For methods other than m, it is direct from ψ being a Φ-interpretation. For m, it follows from the premise of IMPURELINK for B.

[18] Strictly speaking we should drop x from the final state, so that $\theta(m)(\tau, v)$ is a Γ-state. We omit that since, in any event, we rely on implicit coercion (Sec. 3) to apply interpretations to states with additional invariants.

In the rest of the proof, appeals to the premises use $\varphi \cup \theta$.

By transition semantic, there is a single transition from the initial configuration as follows.

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

where $\dot{\mu} = [\mu + m{:}(x : T.B)]$. Any trace of $C; \mathsf{elet}(m)$ corresponds step by step with a trace of C containing a trailing $\mathsf{elet}(m)$ in every configuration with exactly the same states, followed by a final step that executes $\mathsf{elet}(m)$. This step just removes m from $\dot{\mu}$, which means it does not fault nor changes the state. Thus to finish, using lemma 7.9, the proof it is enough to prove the following:

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

(ii) for any τ, if $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \tau,\ \dot{\mu}\rangle$ then $\tau \models_{\varphi} Q$ and $\sigma{\to}\tau \models_{\varphi} \varepsilon$,

(iii) for all $\tau, \sigma', \tau', \pi$ if $\sigma' \models^{\Gamma}_{\varphi} P$ and $\mathit{Agree}(\sigma, \sigma', \varepsilon, \pi, \varphi)$
and $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \tau,\ \dot{\mu}\rangle$ and $\langle C,\ \sigma',\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \tau',\ \dot{\mu}\rangle$ then there is ρ with $\rho \supseteq \pi$ and $\mathit{Lagree}(\tau, \tau', \rho, \mathit{written}(\sigma, \tau) \cup \mathit{freshLocs}(\sigma, \tau))$.

We prove (i)—(iii) using the following claim.

> **Claim A.** For all $C', \sigma', \dot{\mu}'$ and m-truncated trace $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle C',\ \sigma',\ \dot{\mu}'\rangle$, there is a trace $\langle C,\ \sigma,\ \mu\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^{*} \langle C',\ \sigma',\ \mu'\rangle$, where $\mu' = \dot{\mu}' \restriction m$. Also, if $C' = m(z); D$ for some z, D then $\sigma' \models_{\varphi\cup\theta} R^{x}_{z}$.

(i) Suppose $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle C',\ \sigma',\ \dot{\mu}'\rangle \overset{\varphi}{\longmapsto} \lightning$. If the part of this trace before faulting is m-truncated then we have $\langle C, \sigma, \mu\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^{*} \langle C',\ \sigma',\ \mu'\rangle$ by Claim A. In this case, from $\langle C',\ \sigma',\ \dot{\mu}'\rangle \overset{\varphi}{\longmapsto} \lightning$ we have by semantics $\mathit{Active}(C')$ is a field access/update or a context call, and hence not a call to m. Thus by the special correspondence Lemma 7.5 we get $\langle C',\ \sigma',\ \mu'\rangle \overset{\varphi\cup\theta}{\longmapsto} \lightning$. But this contradicts validity of the correctness judgment for C.

Now suppose that the trace $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle C',\ \sigma',\ \dot{\mu}'\rangle$ is not m-truncated. From Lemma A.2 it has a prefix of the form

$$\begin{aligned}
&\langle C,\ \sigma,\ \dot{\mu}\rangle \\
&\overset{\varphi}{\longmapsto}{}^{*} \langle m(z); D,\ \tau,\ \dot{\nu}\rangle \\
&\overset{\varphi}{\longmapsto} \langle B^{x}_{x'}\mathsf{ecall}(x'); D,\ \upsilon,\ \dot{\nu}\rangle \\
&\overset{\varphi}{\longmapsto}{}^{*} \langle A; \mathsf{ecall}(x'); D,\ \sigma',\ \dot{\mu}'\rangle
\end{aligned}$$

where x' is a fresh variable and υ is $[\tau + x'{:}\tau(z)]$. Moreover $\langle A,\ \sigma',\ \dot{\mu}'\rangle \overset{\varphi}{\longmapsto} \lightning$, because this was not a completed call. Notice that $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle m(z); D,\ \tau,\ \dot{\nu}\rangle$ is a m-truncated trace. So by Claim A, we have $\tau \models_{\varphi\cup\theta} R^{x}_{z}$ and thus $\upsilon \models_{\varphi} R^{x}_{x'}$ (using Equation (4)). We also have $\langle B^{x}_{x'},\ \upsilon,\ \dot{\nu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \lightning$. But this contradicts correctness of $B^{x}_{x'}$, which follows from the premise for B by Lemma A.4. So (i) is proved.

(ii) Suppose $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \tau,\ \dot{\mu}\rangle$. This is m-truncated, so by Claim A, we get $\langle C,\ \sigma,\ \mu\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \tau,\ \mu\rangle$.

Also by the hypothesis, we have $\sigma \models_{\varphi} P$. From (4), we get $\sigma \models_{\varphi\cup\theta} P$. By the validity of the judgment $\Phi, \Theta; \psi \vdash^{\Gamma} C :\ P \rightsquigarrow Q\ [\varepsilon]$, we have $\tau \models_{\varphi\cup\theta} Q$ and $\sigma{\to}\tau \models_{\varphi\cup\theta} \varepsilon$. Thus $\tau \models_{\varphi} Q$ and $\sigma{\to}\tau \models_{\varphi} \varepsilon$, since $\mathit{wlocs}(\sigma, \varphi, \varepsilon) = \mathit{wlocs}(\sigma, \varphi \cup \theta, \varepsilon)$.

(iii) Suppose $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \tau,\ \dot{\mu}\rangle$, $\langle C,\ \sigma',\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \tau',\ \dot{\mu}\rangle$ and there is a refperm π such that $\mathit{Agree}(\sigma, \sigma', \varepsilon, \pi, \varphi)$ and $\sigma' \models^{\Gamma}_{\varphi} P$. The traces are m-truncated. By Claim A, we have traces $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \tau,\ \dot{\mu}\rangle$ and $\langle C,\ \sigma',\ \dot{\mu}\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \tau',\ \dot{\mu}\rangle$. Since $\mathit{rlocs}(\sigma, \varphi, \varepsilon) =$

rlocs$(\sigma, \varphi \cup \theta, \varepsilon)$, we have *Agree*$(\sigma, \sigma', \varepsilon, \pi, \varphi \cup \theta)$. By the Read Effect property of the premise for C, there is refperm ρ such that $\rho \supseteq \pi$, *Lagree*$(\tau, \tau', \rho,$ *written*$(\sigma, \tau) \cup$ *freshLocs*$(\sigma, \tau))$.

To prove Claim A, we consider the following claim.

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

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

if the trace $\langle C_0,\, \sigma_0,\, \dot{\mu}_0\rangle \overset{\varphi}{\longmapsto}{}^* \langle C',\, \sigma',\, \dot{\mu}'\rangle$ has exactly n completed topmost calls of m, then there is a trace

$$\langle C,\, \sigma,\, \mu\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^* \langle C_0,\, \sigma_0,\, \mu_0\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^* \langle C',\, \sigma',\, \mu'\rangle,$$

where $\mu_0 = \dot{\mu}_0 \restriction m$ and $\mu' = \dot{\mu}' \restriction m$, if we have a trace $\langle C,\, \sigma,\, \mu\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^* \langle C_0,\, \sigma_0,\, \mu_0\rangle$.

To prove Claim A, take $\langle C_0,\, \sigma_0,\, \dot{\mu}_0\rangle$ to be $\langle C,\, \sigma,\, \dot{\mu}\rangle$. Using Claim B, from trace $\langle C,\, \sigma,\, \mu\rangle \overset{\varphi}{\longmapsto}{}^* \langle C',\, \sigma',\, \dot{\mu}'\rangle$, we get trace $\langle C,\, \sigma,\, \mu\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^* \langle C',\, \sigma',\, \mu'\rangle$. For the second part of Claim A, suppose C' is $m(z); D$ for some z, D. If $\sigma' \not\models_{\varphi\cup\theta} R^x_z$ then we would have $\langle C',\, \sigma',\, \mu'\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^* \lightning$ and hence $\langle C,\, \sigma,\, \mu\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^* \lightning$. But this would contradict the premise for C, since we assumed at the outset that $\sigma \models_\varphi P$. This proves Claim A.

To prove Claim B we build the needed trace by induction on n. To build the trace via $\overset{\varphi\cup\theta}{\longmapsto}$, consider the m-truncated trace

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

with exactly n completed topmost calls of m in the second part. Using Lemma A.2, we obtain intermediate states $\tau_i, \upsilon, \dot{\sigma}, \sigma_i$ and environments $\dot{\mu}_i$ (using names $\dot{\mu}_i$ to indicate that each binds m to $(x : T.B)$) such that

$$\begin{array}{ll}
\langle C_0,\, \sigma_0,\, \dot{\mu}_0\rangle & \\
\overset{\varphi}{\longmapsto}{}^* \langle m(z_1); C_1,\, \tau_1,\, \dot{\mu}_1\rangle & \text{with no invocations of } m \\
\overset{\varphi}{\longmapsto} \langle B^x_{x_1}; \mathsf{ecall}(x_1); C_1,\, \upsilon_1,\, \dot{\mu}_1\rangle & \text{where } \upsilon_1 = [\tau_1 + x_1 \colon \tau_1(z_1)] \text{ and } x_1 \text{ is fresh} \\
\overset{\varphi}{\longmapsto}{}^* \langle \mathsf{ecall}(x_1); C_1,\, \dot{\sigma}_1,\, \dot{\mu}_1\rangle & \text{where } \langle B^x_{x_1},\, \upsilon_1,\, \dot{\mu}_1\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip},\, \dot{\sigma}_1,\, \dot{\mu}_1\rangle \\
\overset{\varphi}{\longmapsto} \langle C_1,\, \sigma_1,\, \dot{\mu}_1\rangle & \text{where } \sigma_1 = \dot{\sigma}_1 \restriction x_1 \\
\vdots & \text{containing } n-1 \text{ topmost invocations of } m \\
\overset{\varphi}{\longmapsto} \langle C_n,\, \sigma_n,\, \dot{\mu}_n\rangle & \\
\overset{\varphi}{\longmapsto}{}^* \langle C',\, \sigma',\, \dot{\mu}'\rangle. & \text{with no completed topmost invocations of } m
\end{array} \tag{28}$$

For any two configurations $\langle A,\, \tau,\, \dot{\mu}\rangle$ and $\langle A',\, \sigma',\, \mu\rangle$, we call them matching configurations iff $A = A'$, $\tau = \tau'$ and $\dot{\mu} = [\mu + m \colon (x : T.B)]$ and hence $\mu = \dot{\mu} \restriction m$.

Below, using Lemma A.3 we will construct a trace via $\overset{\varphi\cup\theta}{\longmapsto}$ that looks as follows:

$$\begin{array}{ll}
\langle C_0,\, \sigma_0,\, \mu_0\rangle & \\
\overset{\varphi\cup\theta}{\longmapsto}{}^* \langle m(z_1); C_1,\, \tau_1,\, \mu_1\rangle & \text{matching the configurations above, so } \mu_1 = \dot{\mu}_1 \restriction m \\
\overset{\varphi\cup\theta}{\longmapsto} \langle C_1,\, \sigma_1,\, \mu_1\rangle & \text{a single step by Lemma 7.8 (2)} \qquad (*) \\
\vdots & \text{containing } n-1 \text{ additional invocations of } m \\
\overset{\varphi\cup\theta}{\longmapsto} \langle C_n,\, \sigma_n,\, \mu_n\rangle & \\
\overset{\varphi\cup\theta}{\longmapsto}{}^* \langle C',\, \sigma',\, \mu'\rangle & \text{again matching configurations}
\end{array} \tag{29}$$

By induction on n, we prove that $\langle C_i, \sigma_i, \dot{\mu}_i\rangle$ and $\langle C_i, \sigma_i, \mu_i\rangle$ are matching configurations for $i = 1, 2, \ldots, n$ in two traces. In the base case of the induction, $n = 0$, all but one line of the given decomposed trace is empty. That is, we have $\langle C_0, \sigma_0, \dot{\mu}_0\rangle \overset{\varphi}{\longmapsto}{}^* \langle C', \sigma', \dot{\mu}'\rangle$ without any intermediate calls of m (but possibly a call in the last configuration). Using special correspondence Lemma 7.5 we can simply drop m from each environment to get a step by step matching trace $\langle C_0, \sigma_0, \mu_0\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^* \langle C', \sigma', \mu'\rangle$.

For the inductive case, $n > 0$, the initial steps $\langle C_0, \sigma_0, \dot{\mu}_0\rangle \overset{\varphi}{\longmapsto}{}^* \langle m(z_1); C_1, \tau_1, \dot{\mu}_1\rangle$ are matched as in the base case, up to the first invocation of m, in state τ_1, environment $\dot{\mu}_1$, and with continuation C_1. At that point we have $\tau_1 \models_\varphi Q^x_{z_1}$, otherwise we can derive a contradiction: We just established $\langle C_0, \sigma_0, \mu_0\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^* \langle m(z_1); C_1, \tau_1, \mu_1\rangle$, and if $\tau_1 \not\models_{\varphi\cup\theta} Q^x_{z_1}$, then we get $\langle m(z_1); C_1, \tau_1, \mu_1\rangle \overset{\varphi\cup\theta}{\longmapsto} \lightning$. Furthermore, by hypothesis of the claim we have $\langle C, \sigma, \mu\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^* \langle C_0, \sigma_0, \mu_0\rangle$. Putting these together we would obtain a faulting trace from $\langle C, \sigma, \mu\rangle$ via $\overset{\varphi\cup\theta}{\longmapsto}$. This contradicts the premise for C, since $\sigma \models_\varphi P$ which gives us $\sigma \models_{\varphi\cup\theta} P$. Thus we have $\sigma_1 \in \theta(m)(\tau_1, \tau_1(z))$ by the definition of θ. By transition semantics for impure call in Fig. 9, we get $\langle m(z); C_1, \tau_1, \mu\rangle \overset{\varphi\cup\theta}{\longmapsto} \langle C_1, \sigma_1, \mu\rangle$. Thus $\langle C_1, \sigma_1, \dot{\mu}_1\rangle$ and $\langle C_1, \sigma_1, \mu_1\rangle$ in both traces are matching configurations.

What remains from this configuration onward is a trace with $n-1$ completed invocations of m, from a configuration reachable from $\langle C, \sigma, \dot{\mu}\rangle$. Moreover, we have $\langle C, \sigma, \dot{\mu}\rangle \overset{\varphi}{\longmapsto}{}^* \langle C_1, \sigma_1, \dot{\mu}_1\rangle$. So we can apply the inductive hypothesis to the trace $\langle C_1, \sigma_1, \dot{\mu}_1\rangle \overset{\varphi}{\longmapsto}{}^* \langle C', \sigma', \dot{\mu}'\rangle$. This concludes the proof of Claim B. □

A.2. Read Effect for While Rule

Since the soundness of a similar rule is proved in [Banerjee et al. 2013], we only show the Read Effect property.

$$\textsc{While}\ \frac{\Phi;\psi \vdash C:\ P \wedge (x \neq 0) \rightsquigarrow P\,[\varepsilon, \mathsf{wr}\,H\,\text{`}\overline{f}, \mathsf{rd}\,H\,\text{`}\overline{f}] \qquad \varepsilon \text{ has framed reads} \quad \varepsilon \text{ is } P;\Phi;\psi/(\varepsilon, \mathsf{wr}\,H\,\text{`}\overline{f})\text{-immune} \qquad \Phi;\psi \models P \Rightarrow H\#g \qquad \mathsf{wr}\,g \notin \varepsilon}{\Phi;\psi \vdash \mathsf{while}\ x\ \mathsf{do}\ C:\ P \wedge g = \mathsf{alloc} \rightsquigarrow P \wedge (x = 0)\,[\varepsilon, \mathsf{rd}\,x]}$$

Let $D = \mathsf{while}\ x\ \mathsf{do}\ C$ and $\eta = \varepsilon, \mathsf{rd}\,x$. To prove Read Effect property for this rule, consider any Φ-interpretation φ that extends ψ. Suppose for states $\sigma, \sigma', \tau, \tau'$ and refperm π we have

$$\sigma \models_\varphi P \wedge g = \mathsf{alloc}, \qquad \sigma' \models_\varphi P \wedge g = \mathsf{alloc}, \qquad \mathit{Agree}(\sigma, \sigma', \eta, \pi, \varphi), \tag{30}$$

and

$$\langle D, \sigma, \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip}, \tau, \_\rangle \text{ and } \langle D, \sigma', \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip}, \tau', \_\rangle.$$

We show that there is a refperm ρ such that $\rho \supseteq \pi$ and $\mathit{Lagree}(\tau, \tau', \rho, \mathit{written}(\sigma, \tau) \cup \mathit{freshLocs}(\sigma, \tau))$.

By semantics (as pointed out in Theorem 7.4 in [Banerjee et al. 2013]), the two traces can be decomposed into iterations. That is, there are $m, n \geq 0$ and states $\sigma_0, \ldots, \sigma_m$ and $\sigma'_0, \ldots, \sigma'_n$ such that $\sigma_0 = \sigma$, $\sigma'_0 = \sigma'$, $\sigma_m = \tau$ and $\sigma'_n = \tau'$. And for $0 < i \leq m$ and $0 < j \leq n$, we have traces

$$\langle C, \sigma_{i-1}, \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip}, \sigma_i, \_\rangle \text{ and } \langle C, \sigma'_{j-1}, \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip}, \sigma'_j, \_\rangle.$$

Also, we have $\sigma_i(x) \neq 0$, $\sigma'_j(x) \neq 0$, for $0 \leq i < m$ and $0 \leq j < n$, and $\sigma_m(x) = 0$ and $\sigma'_n(x) = 0$. From these traces, we get

$$\langle D, \sigma_{i-1}, \_\rangle \overset{\varphi}{\longmapsto} \langle C; D, \sigma_{i-1}, \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle D, \sigma_i, \_\rangle$$

and

$$\langle D, \sigma'_{j-1}, \_\rangle \overset{\varphi}{\longmapsto} \langle C; D, \sigma'_{j-1}, \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle D, \sigma'_j, \_\rangle,$$

for $0 < i \leq m$ and $0 < j \leq n$.

To finish the proof, we prove the following Claim.

> **Claim.** For k, $0 \leq k \leq m$, we have $k \leq n$ and there is a refperm $\rho_k \supseteq \pi$ such that
> $\mathit{rlocs}(\sigma_k, \varphi, \eta) = \mathit{rlocs}(\sigma_0, \varphi, \eta)$,
> $\mathit{Agree}(\sigma_k, \sigma'_k, (\eta, \mathsf{rd}\, H\,\text{`}\overline{f}), \rho_k, \varphi)$ and
> $\mathit{Lagree}(\sigma_k, \sigma'_k, \rho_k, \mathit{written}(\sigma_0, \sigma_k) \cup \mathit{freshLocs}(\sigma_0, \sigma_k))$.

To prove the Read Effect condition, first note that from the claim we have $m \leq n$ and $\mathit{Agree}(\sigma_m, \sigma'_m, \eta, \rho_m, \varphi)$. Hence $\sigma'_m(x) = \sigma_m(x) = 0$. This means that the computation starting at σ' stops at state σ'_m. Thus $m = n$. So $\tau' = \sigma'_n = \sigma'_m$. The last statement of the claim for $k = m$ gives us

$$\mathit{Lagree}(\tau, \tau', \rho, \mathit{written}(\sigma, \tau) \cup \mathit{freshLocs}(\sigma, \tau)),$$

where $\rho = \rho_m$. This finishes the soundness proof.

Proof of the claim by is induction on k. For base case of $k = 0$, we take $\rho_0 = \pi$. From (30), we know that $\mathit{Agree}(\sigma, \sigma', \eta, \pi, \varphi)$. Since $\Phi; \psi \models P \Rightarrow H\#g$ and $\sigma \models_\varphi P \wedge g = \mathsf{alloc}$, we have $\sigma_0 \models H \subseteq \{\mathit{null}\}$, so $\mathit{rlocs}(\sigma_0, \varphi, \mathsf{rd}\, H\,\text{`}\overline{f}) = \varnothing$. Thus we have $\mathit{Agree}(\sigma_0, \sigma'_0, (\eta, \mathsf{rd}\, H\,\text{`}\overline{f}), \pi, \varphi)$. We also have $\mathit{Lagree}(\sigma_0, \sigma'_0, \pi, \mathit{written}(\sigma_0, \sigma_0) \cup \mathit{freshLocs}(\sigma_0, \sigma_0))$, because $\mathit{written}(\sigma_0, \sigma_0) = \mathit{freshLocs}(\sigma_0, \sigma_0) = \varnothing$.

To prove induction step for $k \neq 0$, we assume that the claim holds for $k-1$. Since $k \neq 0$ we have $\sigma_{k-1}(x) \neq 0$ (by semantics). From induction hypothesis for $k-1$, we know that

$$\mathit{Agree}(\sigma_{k-1}, \sigma'_{k-1}, (\eta, \mathsf{rd}\, H\,\text{`}\overline{f}), \rho_{k-1}, \varphi) \tag{31}$$

Thus $\sigma'_{k-1}(x) \neq 0$. This means $k \leq n$ or the computation starting from state σ'_0 has at least k iterations. Since η is $\varepsilon, \mathsf{rd}\, x$, the agreement (31) implies

$$\mathit{Agree}(\sigma_{k-1}, \sigma'_{k-1}, (\varepsilon, \mathsf{wr}\, H\,\text{`}\overline{f}, \mathsf{rd}\, H\,\text{`}\overline{f}), \rho_{k-1}, \varphi)$$

So from the Read Effect property of the first premise of the rule, there exists a refperm $\rho_k \supseteq \rho_{k-1}$ such that

$$\mathit{Lagree}(\sigma_k, \sigma'_k, \rho_k, \mathit{written}(\sigma_{k-1}, \sigma_k) \cup \mathit{freshLocs}(\sigma_{k-1}, \sigma_k)) \tag{32}$$

Also, from the Write Effect property of the first premise, we have $\sigma_{k-1} \rightarrow \sigma_k \models_\varphi \varepsilon, \mathsf{wr}\, H\,\text{`}\overline{f}, \mathsf{rd}\, H\,\text{`}\overline{f}$. Since ε is $P, \Phi, \psi/(\varepsilon, \mathsf{wr}\, H\,\text{`}\overline{f})$-immune, we have $\eta$ is $P, \Phi, \psi/(\varepsilon, \mathsf{wr}\, H\,\text{`}\overline{f})$-immune. From Lemma 6.9 and induction hypothesis for $k-1$, we have

$$\mathit{rlocs}(\sigma_k, \varphi, \eta) = \mathit{rlocs}(\sigma_{k-1}, \varphi, \eta) = \mathit{rlocs}(\sigma_0, \varphi, \eta). \tag{33}$$

From induction hypothesis, we have $\mathit{Lagree}(\sigma_{k-1}, \sigma'_{k-1}, \rho_{k-1}, \mathit{written}(\sigma_0, \sigma_{k-1}) \cup \mathit{freshLocs}(\sigma_0, \sigma_{k-1}))$. From the first premise and (31), we have

$$\begin{array}{l} \sigma_{k-1}, \sigma'_{k-1} \Rightarrow \sigma_k, \sigma'_k \models_\varphi \varepsilon, \mathsf{wr}\, H\,\text{`}\overline{f}, \mathsf{rd}\, H\,\text{`}\overline{f}, \\ \sigma'_{k-1}, \sigma_{k-1} \Rightarrow \sigma'_k, \sigma_k \models_\varphi \varepsilon, \mathsf{wr}\, H\,\text{`}\overline{f}, \mathsf{rd}\, H\,\text{`}\overline{f} \text{ and} \\ \mathit{Agree}(\sigma_{k-1}, \sigma'_{k-1}, (\varepsilon, \mathsf{wr}\, H\,\text{`}\overline{f}, \mathsf{rd}\, H\,\text{`}\overline{f}), \rho_{k-1}, \varphi). \end{array} \tag{34}$$

Using Lemma 6.12, we get $\mathit{Lagree}(\sigma_k, \sigma'_k, \rho_k, \mathit{written}(\sigma_0, \sigma_{k-1}) \cup \mathit{freshLocs}(\sigma_0, \sigma_{k-1}))$. Since $\mathit{written}(\sigma_0, \sigma_k) \subseteq \mathit{written}(\sigma_0, \sigma_{k-1}) \cup \mathit{written}(\sigma_{k-1}, \sigma_k)$ and $\mathit{freshLocs}(\sigma_0, \sigma_k) = \mathit{freshLocs}(\sigma_0, \sigma_{k-1}) \cup \mathit{freshLocs}(\sigma_{k-1}, \sigma_k)$, from (32), we have

$$\mathit{Lagree}(\sigma_k, \sigma'_k, \rho_k, \mathit{written}(\sigma_0, \sigma_k) \cup \mathit{freshLocs}(\sigma_0, \sigma_k)).$$

Now (31) is equivalent to

$$\mathit{Lagree}(\sigma_{k-1}, \sigma'_{k-1}, \rho_{k-1}, \mathit{rlocs}(\sigma_{k-1}, \varphi, (\eta, \mathsf{rd}\, H\,\text{`}\overline{f}))).$$

From (34) and using Lemma 6.12, we get $Lagree(\sigma_k, \sigma'_k, \rho_k, rlocs(\sigma_k, \varphi, (\eta, \mathsf{rd}\, H\, \text{`}\overline{f})))$. This means $Agree(\sigma_k, \sigma'_k, (\eta, \mathsf{rd}\, H\, \text{`}\overline{f}), \rho_k, \varphi)$. This proves the Claim.

Index