A Logical Analysis of Framing for Specifications with Pure Method Calls

ANINDYA BANERJEE, IMDEA Software Institute, Spain
DAVID A. NAUMANN, Stevens Institute of Technology, USA
MOHAMMAD NIKOUEI, Stevens Institute of Technology, 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 provides modular reasoning principles for end-to-end correctness. Modular reasoning is embodied in a proof rule for linking a module's method implementations with a client that relies on the method contracts. Soundness is proved with respect to a conventional operational semantics and uses an extensional (that is, relational) interpretation of read effects. Applicability to tools based on SMT-solvers is demonstrated through machine-checked verification of examples. The developments in this article can guide the implementations of linking as used in modular verifiers and serve as basis for studying observationally pure methods and encapsulation.

CCS Concepts: • **Theory of computation → Programming logic**; **Hoare logic**; **Program reasoning**; *Automated reasoning*;

ACM Reference Format:
Anindya Banerjee, David A. Naumann, and Mohammad Nikouei. 2018. A Logical Analysis of Framing for Specifications with Pure Method Calls. *ACM Trans. Program. Lang. Syst.* 00, 00, Article 00 (March 2018), 91 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

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, in a Java-like programming notation.

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

We would like to prove the asserted postcondition, by reasoning that the state read by c.get() is disjoint from the state written by d.set(6). One problem is how to make sense of specifications that invoke methods, like get in the assertion and in the postcondition of set. Another problem is how

Authors' addresses: Anindya Banerjee, IMDEA Software Institute, Pozuelo de Alarcon, Madrid, Spain, anindya.banerjee@imdea.org; David A. Naumann, Stevens Institute of Technology, Hoboken, New Jersey, USA, naumann@cs.stevens.edu; Mohammad Nikouei, Stevens Institute of Technology, Hoboken, New Jersey, USA, snikouei@stevens.edu.

0164-0925/2018/3-ART00 $15.00
https://doi.org/0000001.0000001

to specify frame conditions, for sound 'local' reasoning about disjointness of effects, even in the presence of method calls in specifications. This article aims to solve these problems in a way that can be used in verification tools based on automated theorem provers for first-order logic (***FOL***) and proved sound with respect to standard operational semantics of 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 a predicate, say c.get()=5, can be asserted preceding a method call, say d.set(6), then it still holds following that call—provided that the locations on which the predicate depends are disjoint from those that may be written according to the method'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). The challenges include how to precisely describe locations a method may write, and to describe the locations on which a predicate depends (its read effect or 'footprint'), without violating abstraction boundaries. The challenge illustrated by the example is that the predicate may itself involve a method call.

There are practical benefits to using programmed methods in postconditions, like get in the example, as well as in preconditions and in frame conditions. This seems sensible provided that they are ***pure*** in the sense of having no observable effects other than reading, and are terminating so there is a definite value. The idea is that such a method is computing a function and can be used as such in reasoning, with well known benefits of functional and data abstraction. One may call it a pun, akin to the fundamental pun of Hoare logic: treating program variables as logical variables. The puns make it possible for specifications to be expressed in notation close to the programming language, making it more accessible to engineers [22].

Prior work on pure methods has addressed termination of pure methods. Versions of the Java Modeling Language (JML) [35] allow a pure method to be called in its own postcondition, but decreasing a measure, the same requirement as for recursive calls in the body of a method being proved to terminate [25]. Pure method calls are also useful in frame conditions, again leading to apparent recursion for which it is challenging to determine sound reasoning principles.

Pure methods can lead to unsatisfiable specifications. For example, naïve use of the pure method specification

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

could lead to inconsistencies like the formula f(x) = f(x)+1. This example is ruled out by the requirement to decrease a measure. But the formula **result** * **result** = x does not call f at all; yet, as a postcondition it too is unsatisfiable for some values of x. For purposes of runtime assertion checking, it is clear that care must be taken with recursive calls in postconditions. But for purposes of static verification, the potential for unsatisfiable postconditions does not itself necessitate that a pure method call in its own postcondition must decrease a measure. Indeed, the postcondition **result** = f(x) is benign.

Prior work on pure methods focused on verification-condition generation (***VC-gen***), usually taking axiomatic semantics for granted rather than defining and proving soundness with respect to operationally grounded program semantics (see [22, 25, 54, 59] and others in Section 12). The prior work focused on methodological considerations and on encodings that work effectively with SMT solvers. In these works, assumed specifications are encoded as axioms. The linking together of verifications for individual methods is embodied procedurally, in the implementation of the verification tool. This can lead to misunderstandings about what is assumed and what is proved. The intricacies of dealing with heap structure, framing, and purity have led to soundness bugs in

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

Define I $\mathrel{\widehat{=}} \forall x, y : \mathit{Cell} \cdot \; x \in x.\mathit{foot} \;\wedge\; (x = y \;\vee\; x.\mathit{foot} \mathbin{\#} y.\mathit{foot})$. It plays the role of a public invariant.

Fig. 1. Example: Cell.

implemented verification systems (as reported in Heule et al. [31]) as well as intricate restrictions on VCs without clear semantic justification.

This article provides a foundational account, by way of a conventional program logic that caters for SMT solvers 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 *proof rules for linking* the implementation of an interface (that is, collection of method specifications) with a client that relies on that interface. A notion akin to Kassios' 'self-framing' frame conditions [33], used for reasoning about preservation of disjointness, emerges as important for reasoning about read effects. Our account shows that some restrictions in prior work are unnecessary. The restriction to decrease a measure in recursive calls is disentangled and justified directly in terms of a linking rule.

1.1 Approach: first steps

Suppose the internal representation of Cell objects consists of an integer field value. The frame condition for set could say it writes **self**.value. We use the term 'frame condition' to include read effects, which are important for framing assertions that call pure methods; for example, get reads **self**.value. With respect to the client code (at the beginning of Section 1), the frame condition for the call d.set(6) would allow the postcondition of c.set(5), that is, the predicate c.get() = 5, to be framed over the call d.set(6), yielding the desired assertion. But such specifications expose the internal representation. They would preclude, for example, an alternative implementation that uses, instead of integer field value, a pointer to a list of integer arrays (to represent big numbers).

Better specifications appear in Figure 1, using ***ghost state*** to describe the notional 'footprint' of each instance of Cell, and postconditions from which the client can deduce disjointness of the representations of c and d. Ghost state is mutable instrumentation added for reasoning but not affecting concrete program state. Use of ghost state for footprints is a key part of the 'dynamic frames' approach [33] used in some prior work on framing and pure methods [38, 59].

Our specification is based on a type **rgn**, short region. A ***region*** is a set of object references. The value of field **self**.foot is thus a set of references and **self**.foot`value denotes the locations of the value fields of those objects. (The notation `value forms an ***image expression***.) For example, in a state where the value of **self**.foot is the set $\{p, q\}$, the set of locations denoted by **self**.foot`value is the set $\{(p, \text{value}), (q, \text{value})\}$. In the sequel we write such pairs as p.value and q.value. 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. Some other works based on dynamic frames use location sets directly [10, 33, 59], but we follow [8] in describing location sets in terms of fields (or data groups) and regions, because locations are not first-class values in Java-like languages (by contrast with, say, C).

The specification of get is an example of a pure method in its own postcondition. For an example of one in its own frame condition, we could replace the ghost field foot by region-valued method footpm. The read effect of footpm might be footpm()‘**any**, making it ‘self-framing’ [33]. Figure 2 has a more interesting example with pure method calls in a precondition as well as in a frame condition.

The specifications 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**.value+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**.value) is correct only if we interpret its specifications the right way.

By contrast with this simple example, practical applications of pure methods pose the challenge of reasoning about observational purity, that is, benign side effects on encapsulated data representations, an old problem [32]. There are many examples, including memoization, lazy initialization, and path compression in Union-Find structures, which involve allocation of fresh objects and mutation of existing ones. Prior work has addressed an aspect of that challenge, namely ‘weakly pure’ methods that allocate and even return fresh objects, though not modifying pre-existing locations.

1.2 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. The proof rules include a frame rule and rules for linking of pure and impure methods with their clients. The implementations of pure methods are commands which may be recursive and may write local variables.
- We provide semantics for the judgment of correctness under hypotheses, where specifications (pre, post, and frame) can refer to pure methods that can also be called in code. The key notion of partial context interpretation enables us to explain and disentangle restrictions in prior work.
- A relational semantics is used for read effects of commands, directly capturing the extensional meaning of dependency. 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.
- Soundness is proved in detail, directly in terms of a standard small step semantics.
- We explain the attractive idea of weak purity which has been explored in prior works. Our analysis sheds light on why weak purity has turned out to be difficult to get right, which may explain why it has fallen out of favor.
- Case studies are presented, to show how the logic relates to verification tools based on VC-gen.

This article may serve as a foundation to guide the investigation of programming methodologies for early detection of unsatisfiable specifications or for other concerns—but this is not a methodological investigation. The examples are crafted to explain technical points, not to argue for or against uses of pure methods, dynamic frames, or anything else. Although the logic may serve as a foundation for soundness of a VC-generator, there are many engineering considerations and comprehensive discussion of VC-gen is beyond the scope of this article.

A preliminary version [6] of this article appeared in a conference but major changes have been made to the semantics and core definitions. The examples have now been fully verified using SMT solvers, weak purity is considered, and detailed proofs are provided.

Contextual remarks. The approach we take is motivated by the challenge of observational purity, although that is not the focus of this article. The term ‘benevolent side effect’ was introduced by Hoare in seminal work [32] on data abstraction and the hiding of invariants on encapsulated data representations. Hiding is important for modular reasoning, but difficult to achieve in the presence of shared mutable objects. Separation logic provides elegant and effective reasoning about framing and hiding [47], but at the cost of going beyond FOL for assertions. In a semantic account of observational purity for mutable objects, Naumann [46] confirmed the close connection with encapsulation, but it remains an open problem to develop a program logic supporting observational purity for object-based programs.

In prior work, we developed ***region logic*** (***RL***) [8], a Hoare logic for sequential object-based programs, using standard FOL for assertions: the logic supports reasoning via explicit footprints captured in frame conditions, as in Figure 1. 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 module invariants are not falsified by client interaction. This idea is captured in a second-order frame rule for linking method implementations with clients, hiding invariants [5], inspired by a similar rule for separation logic by O’Hearn et al. [47].

Read effects of commands are a relational property [17], so one approach to framing of pure methods would be to use a general relational logic [15, 45, 61]. In ongoing work we adapt RL to a relational version [7], with the eventual aim to formalize observational purity in terms of the hiding of effects. In the present article we study an ordinary (‘unary’) logic, extending RL with read effects and pure method calls in specifications. For brevity we refer to the key papers on RL as ***RLI*** [8] and ***RLII*** [5].

1.3 Approach: linking and partial interpretations

The primary judgment expresses correctness of a program under hypotheses about methods it may call. The judgment is written in the following form:

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

Please ignore ψ for now; it 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 the frame condition, ε. This conclusion is under hypothesis Φ, a list of method specifications called the ***method context***. What’s new in this article, beyond RL, are read effects in ε and Φ, and pure methods used in Φ, P, C, Q, ε and specified in Φ.[1]

[1]Read effects have several applications including compiler optimizations [17]. In this article we do not need read effects for the bodies of impure methods. Nonetheless, we use the single judgment form (1) for all commands. This loses no generality, because there is a maximally permissive read effect, and it is convenient because our syntax allows arbitrary commands for the bodies of pure methods.

One approach to formalizing the semantics of (1) goes by quantifying over all correct implementations of the methods specified by Φ, that is, considering behavior of C when linked with any correct implementation. 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 this approach could be realized by considering all environments that also provide bodies for methods in the hypothesis context. In this approach, the proof of soundness for the linking rule is almost immediate, and semantics only needs to be defined for complete programs. In this paper, we have found it convenient to take a slightly different approach that streamlines much of the technical development. We quantify not over implementations but over possible ***interpretations***, that is, possible denotations of the implementations. For pure m, an interpretation $\varphi(m)$ is a function: it applies to a state and an argument value, and returns a value. For impure m, $\varphi(m)$ applies to a state and an argument value, and can return a set of states; a call to m takes a single step, nondeterministically choosing any of those states. The semantics of (1) quantifies over all φ such that $\varphi(m)$ conforms to the specification $\Phi(m)$ for each m in $dom(\Phi)$.

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**.value+7. By contrast, the expected implementations of get and set are correct only with respect to the interpretation that returns **self**.value.

An interpretation might be given directly, as a mathematical definition provided by the programmer. Or it might be derived from the code as it is in work on VC-gen for pure methods, where pure methods have been restricted to a simple form in order to ensure that the derived interpretation is not inconsistent (see Section 12). We treat interpretations semantically, in order to focus on their use rather than how to obtain them (except in Section 9). We impose no restrictions on the code of pure methods, beyond purity of effect. We do restrict specifications in a method context, to preclude cyclic dependencies between pure methods used in each others' preconditions.

The role of ψ in the judgment form (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 not over all interpretations φ that satisfy Φ, but only those which in addition agree with ψ where it is defined. That is, $\psi(m) = \varphi(m)$ if ψ is defined on m. In the rule of consequence and other rules involving assertions, reasoning can assume both the specifications in Φ and the partial interpretation ψ of pure methods. So ψ is a key part of the linking rule, which we sketch here in simplified form.[2] For this discussion we elide effects. We consider a single method m, specified as $\Theta \mathrel{\widehat{=}} m : (x{:}T, \mathsf{res}{:}U)R \leadsto S$, and ψ with $dom(\psi) = \{m\}$, in the rule:

$$\frac{\Phi, \Theta;\ \vdash C : P \leadsto Q \qquad \Phi, \Theta; \psi \vdash B : R \leadsto \mathsf{res} = m(x) \qquad \psi \models \Phi, \Theta}{\Phi; \vdash \mathsf{let}\ m(x{:}T){:}U = B\ \mathsf{in}\ C\ :\ P \leadsto Q} \tag{2}$$

A client C is linked with the implementation B of a pure method m. The verification of C is under the hypothesis of some specifications Φ, Θ which include the specification Θ of m 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 methods in Φ, Θ. The verification condition for B also has hypothesis Φ, Θ for methods, including m, that may be called in B or used in its specification, and B must be correct with respect to any interpretation of the methods in Φ, but only the fixed interpretation ψ of m. The linking rule discharges the hypothesis Θ about m by providing an implementation B for it.

[2]Rules with similar structure are called recursion rules in the textbook of Apt et al. [3].

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. Indeed, this issue is already present with impure methods. If it is impossible to establish postcondition S then the only implementations B are those that diverge. For the example with postcondition $\mathsf{res} = f(x) + 1$, one can use the rule to prove correctness of B that simply calls $f(x)$ recursively. Of course it diverges.

There are several reasons a specification may be unsatisfiable. For example, the precondition can preclude the postcondition (e.g., $x \geq 0 \rightsquigarrow x < 0$), the frame condition may conflict with the rest (e.g., $\mathsf{true} \rightsquigarrow y = 0$ with empty frame condition), or the postcondition may be unsatisfiable for deep mathematical reasons. In practice it may be helpful to deploy heuristic checks to detect unsatisfiable specifications, but complete checks are not feasible.

Clearly we do not want divergent expressions in formulas. The interpretation $\psi(m)$ must be a total function (at least on inputs satisfying the precondition of m). It is not necessary for its definition to be derived from B; indeed, a useful implementation B may use loops and mutable local variables, whereas $\psi(m)$ could be expressed in convenient mathematical notation. What is required by the rule is that terminating executions of B yield the result defined by $\psi(m)$; that is postcondition $\mathsf{res} = m(x)$. In addition, the condition $\psi \models \Phi, \Theta$ in (2) requires 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.) Notice that the judgment for B does not explicitly require it to establish the postcondition S; to whatever extent the definition of ψ is derived from the code B, one will in fact reason about B—including its termination—to establish $\psi \models \Phi, \Theta$. Because prior work focused on using B both as executable code and as interpretation, and on VC-gen in which linking is not made explicit declaratively, potential divergence and unsatisfiability were studied in terms of consistency of axioms (see Section 9 and Section 12).

Rule (2) addresses uses of pure methods for abstraction, wherein the actual interpretation ψ is not made visible to the client. This is appropriate for complicated or encapsulated data representations and application-specific functionality. By contrast, some prior work addresses situations where the interpretation is defined by an expression that is meaningful (and in scope) in the context of the client. We can account for that as the variation of (2) in which the judgment for C is replaced by

$$\Phi, \Theta; \psi \vdash C : P \rightsquigarrow Q \tag{3}$$

This makes the definition of m 'transparent' by allowing the use of ψ in reasoning about C. Note that if $\psi(m)$ is expressible in terms of the ambient logical theory as some function f, then it can also be made visible with a postcondition S that implies $\mathsf{res} = f(x)$. So we consider rule (2) to have primary importance.

Finally, there is a linking rule that accounts for how an interpretation may be derived from an implementation. The idea is to replace in (2) the premise for B by something like $\Phi, \Theta; \vdash B : R \rightsquigarrow S$, without ψ, and to ensure somehow that B terminates without error from states that satisfy R. This is achieved by augmenting the specification in the premise for B—but not in the hypothesis for client C—with conditions that ensure recursive calls decrease a measure.

1.4 Outline

Section 2 formalizes the programming language, specifications, and definedness formulas derived from specifications.

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: we interpret judgment (1) by quantifying over correct implementations, but correctness is defined with respect to the meaning of specifications—and methods occur in the specifications. Aiming for a foundation for verification tools for first-order programs using SMT solvers, we want to break the circularity and thus avoid fixpoint constructions for the semantics of specifications and correctness judgments. To avoid circularity, Section 3 defines the semantics of expressions and formulas in terms of an arbitrary 'candidate interpretation'; it is not required to satisfy any specifications, and even allows fault ($\lightning$) as an outcome, so the semantics of formulas is three-valued. 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 certain locations 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.

Nondeterminacy is allowed for impure methods, to cater for allocation. Conceptually, an allocator depends on hidden state that is not visible at the level of source code; for a faithful model, we allow it to be nondeterministic. This does not really complicate the technical development. The semantics of read effects involves relating pairs of executions, for which purpose we need to deal with differences in allocation behavior. We do this using bijective renamings ('refperms' in the sequel) in a standard way [4]. The semantics of read effects ensures that correct interpretations are ***quasi-deterministic*** in the sense that the only nondeterminacy is due to allocation.

Section 5 completes the semantics of the correctness judgment (1). For C verified under hypothesis Φ that specifies pure method m called in C or used in the specification of C, linking discharges the hypothesis as explained in Section 1.3. 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 (in Section 2) as in prior work on VC-gen.

The semantics of (1) embodies what is sometimes called 'modular correctness' [36]. It requires that C never calls a pure or impure method of Φ outside its precondition. This becomes explicit in soundness proofs for linking rules.

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 solver. However, we also give proof rules for deriving the subsidiary judgments.

Section 6.3 explicates a notion we call framed reads, which is similar to the notion of self-framing that Kassios introduces for reasoning about separation for freshly allocated objects (Section 12). It turns out that this property is important for reasoning about read effects of commands. Although the extensional semantics of a read effect involves two executions, the readable locations are designated by an effect expression that is interpreted in one of the initial states. That asymmetry can be problematic for composing the effects of commands in sequence, but the asymmetry goes away if the locations on which the effect expression depends are themselves deemed readable. The requisite definitions and results are delicate and were perhaps the most difficult part of our investigation.

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

  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 x.parent = self
    writes self.chrn, x.parent, self.anc()‘size

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

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

Fig. 2. Composite example (adapted from RLI).

Section 7.1 gives the proof rules for the program correctness judgment. Section 7.2 gives examples showing the need for framed reads. Surprisingly, it is untenable to require framed reads in all program judgments, as explained in Section 7.1. Section 7.3 is a worked example highlighting features of the proof system.

Section 8 proves the main theorem: soundness of the rules. The soundness proofs are intricate, especially for the linking rules, because they are proved directly in terms of small-step operational semantics.[3] Soundness for read effects is especially challenging because it involves reasoning about the interpretation of effect expressions in two executions.

Section 9 discusses a variant linking rule that caters for deriving an interpretation from the implementation of a pure method.

Section 10 demonstrates the suitability of our approach for use in SMT-based tools, and explains informally how the logic in this article relates to VC-gen. 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 11 considers weak purity, which allows allocation but not mutation of existing locations.

Section 12 discusses related work and Section 13 concludes. Appendix A provides some additional proofs, and develops the theory of quasi-determinacy as needed to prove soundness of the linking rule for impure methods.

2 PROGRAMS, SPECIFICATIONS, AND DEFINEDNESS FORMULAS

Figure 2 illustrates features of our programming and specification notations, by way of the Composite pattern, a well-known verification challenge problem [1, 18, 53]. A Comp is a node of a tree,

[3]Small-step semantics is essential for the FOL-based form of dynamic frames and encapsulation used in RLII and in planned future work on observational purity.

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

other nodes of which may be accessible to clients. The methods are deliberately under-specified, for expository purposes. To verify the implementations, some invariants are needed, as discussed later (Section 10).

Here is an example client:

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

Aside from the primitive data types int and rgn, the language features class types whose values are object references. Dereferencing is implicit, as in languages like Java. The command b.add(c) faults if the value stored in b is null; otherwise it invokes method add on the referenced object, passing the argument c—itself a reference—by value.

To prove the postcondition asserted in the example client, we want to frame the formula i = d.getSize() over the call b.add(c). The frame condition of add(x) 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 [35] 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 [42] 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 mark **self**.chrn as **specpublic**. See RLI for more discussion of this facet of information hiding.[5]

In order to reason using the frame rule, 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 $\cdot/.$ generates the separator formula and is defined by recursion on syntax. Please note that $\cdot/.$ is not syntax in the logic; it's a function in the metalanguage that is used to obtain formulas from effects. In the example, we compute $\varepsilon\ \cdot/.\ (\mathsf{rd}\ i, d, d.size)$, 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, $d \notin$ b.anc(). The disjointness needs to hold following the elided part of the example client above.

In general, $\eta\ \cdot/.\ \varepsilon$ 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 [53].

2.1 Programs

Figure 3 gives the grammar of programs. It is taken from RLII, with three additions providing for pure methods: a command for linking a result-returning method with its client, and method calls as expressions. These have a single parameter and a single method, to streamline the technical

[5]Owing to the postcondition specified for getSize, the assignment i:=d.getSize() establishes both i=d.getSize() and i=d.size. One can frame i=d.size over the call b.add(c) to establish assertion i=d.size, which is a reasonable specification given that size is specpublic. But the point is to have a simple example of a pure method in a specification.

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

$$\begin{array}{lrl}
\text{(Types)} & T & ::= \mathsf{int} \mid \mathsf{rgn} \mid K \\
\text{(Program Expressions)} & E & ::= x \mid c \mid \mathsf{null} \mid E \oplus E \mid \boxed{m(E)} \text{ where } c \text{ is in } \mathbb{Z}, \oplus \text{ is in } \{=, +, \ldots\} \\
\text{(Region Expressions)} & G & ::= \varnothing \mid x \mid \{E\} \mid G\text{‘}f \mid G \otimes G \mid \boxed{m(F)} \quad \text{where } \otimes \text{ is in } \{\cup, \cap, \backslash\} \\
\text{(Expressions)} & F & ::= E \mid G \\
\text{(Commands)} & C & ::= \mathsf{skip} \mid x := F \mid x := \mathsf{new}\ K \mid x := x.f \mid x.f := x \\
 & & \mid \mathsf{if}\ E\ \mathsf{then}\ C\ \mathsf{else}\ C \mid \mathsf{while}\ E\ \mathsf{do}\ C \mid C\, ;C \mid \mathsf{var}\ x{:}T\ \mathsf{in}\ C \\
 & & \mid m(x) \mid \mathsf{let}\ m(x{:}T) = C\ \mathsf{in}\ C \mid \boxed{\mathsf{let}\ m(x{:}T){:}T = C\ \mathsf{in}\ C}
\end{array}$$

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

development, but the generalization to multiple parameters and methods is straightforward and used in examples. We use over-line notation to indicate multiple elements, e.g., $\overline{T}$ for a list of types.

Assume given a fixed collection of classes. A class has a name and some typed fields. We do not formalize dynamic dispatch or even associate methods with classes; so the term ***method*** is just short for procedure, and a class amounts to a named record type. Distinct classes have distinct field names.[6] The letters T, U, V are used for types and B, C, D for commands. The letters E, F, G are only used for their respective categories in Figure 3.

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. 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. Dereference occurs in the load and store commands $x := y.f$ and $x.f := y$, which we call field access and field update. It can also occur due to image expressions, in the form $x := F$ where x : rgn. For example $x := \{y\}\text{‘}f$ reads $y.f$ in states where $y \neq \mathsf{null}$; it sets x to $\varnothing$ when $y = \mathsf{null}$. We make the semantics precise in Section 3.2.

Please note that ‘$x.f$’ is not an expression; rather, it is part of the syntax of the primitive field access/update commands. It is also part of the syntax of the points-to predicate ‘$x.f = E$’ introduced later. Null dereference is not a cause for faults in the semantics of formulas.

The linking construct, let $m(x{:}T){:}U = C$ in C', designates that m returns a result, of type U. Calls of m are expressions. We refer to result-returning methods as ***pure***, that being their intended use in this paper. However, neither the typing rules nor the operational semantics restricts their effects. Purity is imposed, later, in terms of specifications and proof rules. The body C is executed in a state with both x and the distinguished variable res, the latter initialized to the default value for type U. The final value of res is the value of the call expression. The other linking construct, let $m(x{:}T) = C$ in C', designates that m may be impure; such methods are called in the command form $m(x)$. Both constructs bind m in C' and bind x and m in C.

Typing contexts, ranged over by Γ, are finite maps, written in conventional form, except for a slightly unusual notation for pure methods. 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

[6]Owing to the simple model of classes, the notation G‘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 [42].

[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.

$$\frac{\Gamma, x : T \vdash C}{\Gamma \vdash \mathsf{var}\ x{:}T\ \mathsf{in}\ C} \qquad \frac{\Gamma(m) = (x{:}T)}{\Gamma, z{:}T \vdash m(z)} \qquad \frac{\Gamma(m) = (x{:}T, \mathsf{res}{:}U)}{\Gamma, y{:}U, z{:}T \vdash y := m(z)}$$

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

$$\frac{(f{:}T) \in \mathit{Fields}(\Gamma x)}{\Gamma, y{:}T \vdash x.f := y}$$

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

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

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

Fig. 4. Typing rules for commands.

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:

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

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

If $\Gamma \vdash G : \mathsf{rgn}$ then $\Gamma \vdash G\text{`}f : \mathsf{rgn}$ for any field name $f$ of region or reference type. In case $f : K$, the value of $G\text{`}f$ is the set of f-values of objects in G. In case $f : \mathsf{rgn}$, the value of $G\text{`}f$ is the union of the f-values.

For commands, the 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, simplifies the transition semantics.[8] A typing rule is given for a single pure method, and another rule for a single impure one, for readability. For a let that binds several methods simultaneously, the typing rule checks the body of each method in the context of all the method signatures, to allow mutual recursion.

[8]Its consequence is that we are not fully modeling a module system, because any library in scope for a method is also in scope for its clients. The same restriction is imposed in RLII.

For simplicity, the program syntax does not include designation of ghost code as such. In the examples, anything of type rgn can be considered ghost state, and we sometimes use the keyword **ghost** for emphasis. A modern analysis of ghost code is provided by Filliatre et al. [26].

2.2 Specifications

The syntax of formulas is standard.

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

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

The ***points-to*** predicate $x.f = E$, adapted from separation logic, says x is non-null and the f field of the referenced object is equal to the value of E. (The only change from RLI Section 4.2 is that now E can have pure method calls.) 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 $\{x\}`f \subseteq G \wedge G \subseteq \{x\}`f$.

Effect expressions are given by

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

For brevity we use the term ***effect*** for effect expressions throughout the article, though we also use 'effect' informally to refer to actual computational effects.

We abbreviate a compound effect $\mathsf{wr}\ x, \mathsf{rd}\ x$ as $\mathsf{rw}\ x$ and we often treat compound effects as sets rather than lists. We use identifiers $\varepsilon, \eta, \delta$ for effects, and P, Q, R, S for formulas. Note that, by contrast, E, F, G are used for different kinds of expressions (as in Fig. 3). Finally, we write $\mathsf{wr}\ x.f$ to abbreviate $\mathsf{wr}\ \{x\}`f$.

Effects must be swf for the context Γ in which they occur: $\mathsf{rd}\ x$ and $\mathsf{wr}\ x$ are swf if $x \in dom(\Gamma)$; $\mathsf{rd}\ G`f$, and $\mathsf{wr}\ G`f$ are swf if $\Gamma \vdash G : \mathsf{rgn}$. By contrast with the typing rule for $G`f$ as an expression, which requires the type of $f$ to be a reference type or rgn, we need no restriction on the type of $f$ in the context of an effect. That is because in an effect, $G`f$ refers to the expression's l-value, that is, the locations it designates.

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

Specifications for impure methods take the form $(x\!:\!T)R \leadsto S\ [\eta]$. For pure methods they take the form $(x\!:\!T, \mathsf{res}\!:\!U)R \leadsto S\ [\eta]$. Here R is the precondition, S the postcondition, and η the effects. It is in the following that we restrict pure to have no side effects except possibly divergence.

Definition 2.1 (swf specification). 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 $\Gamma, x\!:\!T$, for the impure form, or $\Gamma, x\!:\!T, \mathsf{res}\!:\!U$, for the pure form. Finally, for a 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 effects that refer to x, for example, $\mathsf{rd}\ \{x\}`f$. 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 formalize 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.

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: rd *vars*(Γ), rd alloc‘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, *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 2.2 (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 Γ, *sigs*(Φ).

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

An example swf context is given by the specifications of get and set from Fig. 1, and another is given by the specifications of add, $\mathit{getSize}$, and anc from Fig. 2. (To be precise, the specifications need to include self as an explicit parameter, and in both cases a constructor method can be added.)

Although we do not formalize modules per se, the linking constructs model linkage of a client to a set of methods that implement an interface. Definition 2.2 has an interesting consequence for linking. Pure method specifications can make mutually recursive reference to each other, and specifications of pure methods can refer to pure methods, but specifications of pure methods cannot refer to impure ones. In addition, for a pure method implementation to satisfy its specification, it cannot invoke any impure method m (unless the m's specification has no writes, in which case calling m is useless). So any verifiable linkage can be written in the form

$$\begin{array}{c}\mathsf{let}\ p_0(x_0{:}T_0){:}U_0 = B_0\,;\ \ldots\ ;\ p_k(x_k{:}T_k){:}U_k = B_k\ \mathsf{in} \\ \mathsf{let}\ m_0(y_0{:}V_0) = C_0\,;\ \ldots\ ;\ m_n(y_n{:}V_n) = C_n\ \mathsf{in}\ D\end{array} \tag{4}$$

where the impure bodies C_i can call both pure and impure methods. This enables us to formulate separate proof rules for linking of pure and impure methods.

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. To this end, we use ***definedness formulas*** [24], 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_F for capture-avoiding syntactic substitution of F for x.

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

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

Although the clause for $df(m(F),\Phi)$ 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 $dom(\Phi)$ 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, $\Gamma \vdash df(P,\Phi)$.

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

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

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

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

[9]Note that any method is allowed to read its parameter(s) and write res, but these effects are not supposed to be included in the specification so the effect for $div6by$ is empty.

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.

3.1 Preliminaries

Assume given an infinite set *Ref* of reference values including a distinguished 'improper reference' *null*. We use o and occasionally p to range over non-null references. 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 *vars*(Γ)-state, where *vars* drops methods and retains only variables. Letters σ, τ, υ range over states.

There is a distinguished variable, alloc : rgn, updates of which are built into the program semantics. Code cannot assign alloc; this restriction is imposed by the typing rules (Figure 4). The semantics of new updates alloc so that in any state it holds the set of all allocated references and does not contain *null*. We are generally concerned with contexts Γ that include alloc.

Treating alloc as a program variable, albeit with special semantics and restricted use, helps streamline the semantic developments and it makes for an expressive assertion language. The main use of alloc in program proofs is for reasoning about freshness, and for practical purposes it may be advisable to use special notation for freshness.

We write $\sigma(x)$ for the value of variable x in state σ, and *Vars*(σ) for the variables of σ (this is called $Dom(\sigma)$ in RLI/II). We write $o.f$ for a non-null reference o paired with field name f (that is, a heap location), and $\sigma(o.f)$ to look up field f of the object referenced by o in the heap. We write $[\sigma \mid x\colon v]$ to update variable x to value v, write $[\sigma + x\colon v]$ to extend σ with additional variable x, and write $[\![\Gamma]\!]$ for the set of Γ-states. For o a non-null reference that is allocated in σ we write *Type*(o, σ) for the class of the object it references, and otherwise *Type*(o, σ) is undefined. In contexts where $\sigma(x)$ cannot be null, we abbreviate $\sigma(\sigma(x).f)$ as $\sigma(x.f)$. Write $[\![T]\!]\sigma$ for the set of values of type T in state σ. Thus $[\![\text{int}]\!]\sigma = \mathbb{Z}$ and $[\![K]\!]\sigma = \{null\} \cup \{o \mid o \in \sigma(\text{alloc}) \wedge Type(o, \sigma) = K\}$. 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 $\Gamma(m) = (x : T, \text{res} : U)$ 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 $\lightning$. This notion reflects that value-returning methods are intended to be pure. If $\Gamma(m) = (x : T)$ then $\theta(m)$ is a 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.[10]

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

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 $\Gamma(m) = (x : T, \text{res} : U)$ then $\theta(m)$ is a function of type
$$(\sigma \in [\![\Gamma]\!]) \times [\![T]\!]\sigma \to ([\![U]\!]\sigma \cup \{\lightning\})$$

[10]We say 'function', and use symbol $\to$, for total functions. We use the notation $(\sigma \in [\![\Gamma]\!]) \times [\![T]\!]\sigma$ for dependently type pairs, that is, pairs (σ, v) such that σ is in $[\![\Gamma]\!]$ and v is in $[\![T]\!]\sigma$.

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

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 $\{=, +, \dots\}$ and $\otimes$ is in $\{\cup, \cap, \setminus\}$. Also, *DeclClass*$(f)$ is the class in which f is declared.

(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 $(\Gamma, \textit{sigs}(\Phi))$-interpretation.

A ***partial candidate* Φ*-interpretation***, or just ***partial candidate***, is a θ defined on some but not necessarily all of the pure methods in Φ, satisfying condition **(pure)** for each m on which θ is defined.

Note that a candidate Φ-interpretation is defined on pure 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 $\textit{dom}(\Gamma)$ which acts on Γ-states, which are $\textit{vars}(\Gamma)$-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 $\Phi; \psi \vdash^\Gamma C : P \leadsto Q\ [\varepsilon]$ 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 $\Gamma, \textit{sigs}(\Phi)$. We often elide Γ.[11]

3.2 Semantics of expressions, formulas, and commands

The denotation of an expression in context $\Gamma \vdash F : T$ in candidate Γ-interpretation θ and state σ is written $[\![\Gamma \vdash F : T]\!]_\theta\sigma$ and defined straightforwardly. See Figure 6, where we write $[\![F]\!]$ for short. Note that

$$[\![\Gamma \vdash E : T]\!]_\theta \in (\sigma \in [\![\Gamma]\!]) \to [\![T]\!]\sigma \cup \{\lightning\}$$

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).

The semantics is designed to cater for convenient reasoning with regions used as ghost code. In particular, the semantics of image expression $G\text{`}f$ never faults unless due to method calls in $G$. For example, in a state where $x$ is null, the value of $\{x\}\text{`}f$ is simply the empty set. (See RLI for more discussion.)

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

Fig. 7. Formulas: three-valued semantics, $[\![\Gamma \vdash P]\!]_\theta\sigma \in \{\mathit{true}, \mathit{false}, \lightning\}$ where σ ranges over Γ-states. Typing context is elided in most cases. As in Figure 6, the $\lightning$-strict let-binder is used.

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

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

Using the semantics for expressions, the 3-valued semantics of formulas is defined in Figure 7. The satisfaction relation $\models_\theta^\Gamma$ is defined by

$$\sigma \models_\theta^\Gamma P \quad \text{iff} \quad [\![P]\!]_\theta\sigma = \mathsf{true} \tag{5}$$

Later we show that when the definedness formulas hold, the usual 2-valued clauses hold for $\models_\theta^\Gamma$ (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 $\Gamma, x{:}U$-state σ we have $[\![\Gamma, x : U \vdash E : T]\!]_\theta\sigma = [\![\Gamma \vdash E : T]\!]_\theta(\sigma\!\upharpoonright\! x)$.

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

$$\sigma \models_\varphi P \text{ iff } \sigma \models_\theta P \quad \text{for all } \sigma, P \tag{6}$$

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 $\theta(m)(\sigma, v) = \theta(m)(\sigma{\upharpoonright}xs, v)$. Here $\sigma{\upharpoonright}xs$ has the same heap as σ but the store is defined only on *dom* (Γ). 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 method 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 control state in a configuration can be an ***extended command***, that is, possibly containing end-markers. The end-marker elet(m) causes m to be removed from the method environment; ecall$(x, \ldots)$ and evar$(x, \ldots)$ cause 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. Apropos the rules for sequence, we identify skip; C with C.

The call of a let-bound method m executes the body $\mu(m)$ with variables renamed to avoid clashes with the calling context. We call this an ***environment call***. 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 similarly named variables like res′.

We use the term ***context call*** for calls to methods that are in the interpretation θ rather than in the environment μ. For such a pure method call $y := m(z)$, the transition semantics 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 skip or a call to an impure method. This can be checked by inspection of the transition rules. In case of an impure method call $m(z)$, it is possible that, even if σ satisfies the precondition, the set $\theta(m)(\sigma, \sigma(z))$ is empty.

For a pure method call $y := m(z)$ that is a context call, the only effect is to assign y. If it is an environment call, there may well be other modifications of state because the transition semantics executes the body like any other command.

4 SEMANTICS OF EFFECTS

This section lays groundwork for defining, in Section 5, correct interpretations and the semantics of correctness judgments. The key notion, called 'allowed dependence', provides a sequentially composable formulation of dependency for read effects in frame conditions. It is defined using a notion of 'agreement' that also plays a role in the framing of formulas.

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

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

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

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

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

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

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

$$\frac{x' \notin \mathit{Vars}(\sigma) \qquad C' = C^{x}_{x'}}{\langle \mathsf{var}\ x : T\ \mathsf{in}\ C,\ \sigma,\ \mu\rangle \overset{\theta}{\longmapsto} \langle C'; \mathsf{evar}(x'),\ [\sigma + x' \colon \mathit{default}(T)],\ \mu\rangle}$$

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

$$\langle \mathsf{ecall}(x),\ \sigma,\ \mu\rangle \overset{\theta}{\longmapsto} \langle \mathsf{skip},\ \sigma \upharpoonright x,\ \mu\rangle \qquad \frac{\langle C,\ \sigma,\ \mu\rangle \overset{\theta}{\longmapsto} \langle C',\ \sigma',\ \mu\rangle}{\langle C\,; D,\ \sigma,\ \mu\rangle \overset{\theta}{\longmapsto} \langle C'\,; D,\ \sigma',\ \mu\rangle} \qquad \frac{\langle C,\ \sigma,\ \mu\rangle \overset{\theta}{\longmapsto} \lightning}{\langle C\,; D,\ \sigma,\ \mu\rangle \overset{\theta}{\longmapsto} \lightning}$$

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

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

Fig. 9. Transition rules, parameterized by a 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 $o \notin \sigma(\mathsf{alloc})$.

A ***location*** is either a variable name x or a heap location[12] 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

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

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

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

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, $\sigma{\rightarrow}\tau \models_\theta \varepsilon$). 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 $\sigma{\rightarrow}\tau \models_\theta \varepsilon$, iff $\sigma \hookrightarrow \tau$ and

(a) for every y in $\mathit{dom}(\Gamma)$, 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 the written, pre-existing locations by

$$\mathit{written}(\sigma,\tau) = \{x \mid \sigma(x) \neq \tau(x)\} \cup \{o.f \mid o.f \in \mathit{locations}(\sigma) \wedge \sigma(o.f) \neq \tau(o.f)\}$$

Then $\sigma{\rightarrow}\tau \models_\theta \varepsilon$ 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 the outcome of a computation can depend on. Dependency is expressed by considering two initial states that agree on the set of locations deemed readable, though they may differ arbitrarily on other locations. Agreement between a pair of states needs to take into account variation in allocation, as the relevant pointer structure in the two states may be isomorphic but involve 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

- $\mathit{dom}(\pi) \subseteq \sigma(\mathsf{alloc})$ 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 π restricts to 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

$$x \in \pi(W) \quad \text{iff} \quad x \in W \qquad\qquad o.f \in \pi(W) \quad \text{iff} \quad (\pi^{-1}(o)).f \in W \tag{7}$$

In words: variables map to themselves, and a heap location $p.f$ is transformed by applying π to the reference p.

Definition 4.2 (agreement on a location set, Lagree). For a set W of locations, and π a refperm from σ to σ', define

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

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

As noted earlier, an important example of a location set is that denoted by a read effect. We often instantiate W by *rlocs*$(\ldots)$, as in the following key definition of what it means for two states to agree on the locations denoted by a read effect.

Definition 4.3 (agreement on read effects, Agree). 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 σ and σ' ***agree on*** ε ***modulo*** π, written $Agree(\sigma, \sigma', \varepsilon, \pi, \theta)$, iff $Lagree(\sigma, \sigma', \pi, rlocs(\sigma, \theta, \varepsilon))$.

As an abbreviation, define $Agree(\sigma, \sigma', \varepsilon, \theta) = Agree(\sigma, \sigma', \varepsilon, \pi, \theta)$ where π is the identity on $\sigma(\mathsf{alloc}) \cap \sigma'(\mathsf{alloc})$.

Agreement on some rd $G`f$ (modulo $\pi$) implies that $\sigma(G) \subseteq dom(\pi)$. However, it is important to note that agreement on rd $G`f$ 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 $\pi(o) \neq o'$.

Agreement on location sets has a kind of symmetry:

$$Lagree(\sigma, \sigma', \pi, W) \text{ implies } Lagree(\sigma', \sigma, \pi^{-1}, \pi(W)) \text{ for all } \sigma, \sigma', \pi, W \tag{8}$$

By contrast, Definition 4.3 of agreement on read effects is left-skewed, in the sense that it refers to the locations denoted by effects interpreted in the left state. So agreement on read effects does not in general exhibit a symmetry property like (8). For example, consider these states, written in suggestive notation:

$$\sigma = [\mathsf{alloc}{:}\{o, p\},\ r{:}\{o\},\ o.f{:}3,\ p.f{:}4] \quad \sigma' = [\mathsf{alloc}{:}\{o, p\},\ r{:}\{o, p\},\ o.f{:}3,\ p.f{:}5] \tag{9}$$

We have $Agree(\sigma, \sigma', id, \text{rd } r`f)$, with $id$ the identity relation on $\{o, p\}$, but unfortunately we do not have $Agree(\sigma', \sigma, id, \text{rd } r`f)$. The asymmetry makes working with agreement somewhat delicate.

At a higher level, there will be symmetry, for two reasons. One has to do with the notion of framed reads, to which we return in Section 6.3. Roughly, it means that if rd $r`f$ is in the effects then so is rd r. The other reason is that the semantics of correctness judgments, defined in the following Section 5, imposes a condition on all pairs of executions. The condition is a property parameterized by states $\sigma, \sigma', \tau, \tau'$. Intuitively, the condition says that if σ and σ' agree on given read effects ε, then τ and τ' agree on any preexisting locations that were written as well as on any fresh locations. In uses of the condition, σ, σ' are initial states and τ, τ' are the corresponding final states resulting from executions of a command or applications of a candidate interpretation. First, define

$$\begin{aligned} &freshRefs(\sigma, \tau) = \tau(\mathsf{alloc}) \setminus \sigma(\mathsf{alloc}) \\ &freshLocs(\sigma, \tau) = \{p.f \mid p \in freshRefs(\sigma, \tau) \land f \in Fields(Type(p, \tau))\} \end{aligned}$$

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

Note that the first conjunct is equivalent to $\rho(freshLocs(\sigma, \tau)) \subseteq freshLocs(\sigma', \tau')$ and this is the form we often use in proofs.

Like Definition 4.3, this definition is left-skewed, both because ε is interpreted in the left state σ and because the fresh and written locations are determined by the left transition σ to τ. The asymmetry is tamed, in later sections, through the use of framed reads and application of the condition to all pairs of executions.

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 notation for values, even references—although strictly speaking the syntax does not include reference literals.[13] 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 $\sigma{\rightarrow}\tau \models_\theta \varepsilon^x_v$ abbreviates $[\sigma + x{:}\, v]{\rightarrow}[\tau + x{:}\, v] \models_\theta \varepsilon$. Finally, $\mathit{wlocs}(\sigma, \theta, \varepsilon^x_v)$ abbreviates $\mathit{wlocs}([\sigma + x{:}\, v], \theta, \varepsilon)$.

A correct candidate interpretation, called context 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 $(x{:}T, \mathsf{res}{:}\,U)P \leadsto Q\ [\varepsilon]$, 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 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', letting $w = \varphi(m)(\sigma, v)$ and $w' = \varphi(m)(\sigma', v')$: if $\mathit{Agree}([\sigma + x : v], [\sigma' + x : v'], (\varepsilon, \mathsf{rd}\,x), \pi, \varphi)$ then $w \overset{\pi}{\sim} w'$
- If m has specification $(x{:}T)P \leadsto Q\ [\varepsilon]$ then for any $\sigma \in [\![\Gamma]\!]$ and $v \in [\![T]\!]\sigma$,
  (d) $\lightning \in \varphi(m)(\sigma, v)$ iff $\sigma \not\models_\varphi P^x_v$, and also $\lightning \in \varphi(m)(\sigma, v)$ implies $\varphi(m)(\sigma, v) = \{\lightning\}$.
  (e) For all $\tau \in \varphi(m)(\sigma, v)$, if $\sigma \models_\varphi P^x_v$ then $\tau \models_\varphi Q^x_v$ and $\sigma{\rightarrow}\tau \models_\varphi \varepsilon^x_v$
  (f) For all τ, σ', τ', v', if
    – $\sigma \models_\varphi P^x_v$,
    – $\sigma' \models_\varphi P^x_{v'}$,
    – $\tau \in \varphi(m)(\sigma, v)$, and
    – $\tau' \in \varphi(m)(\sigma', v')$
  then $[\sigma + x{:}\, v], [\sigma' + x{:}\, v'] {\Rightarrow} \tau, \tau' \models_\varphi (\varepsilon, \mathsf{rd}\,x)$.

We refer to the second part of (d), that is, $\lightning \in \varphi(m)(\sigma, v)$ implies $\varphi(m)(\sigma, v) = \{\lightning\}$, as ***fault determinacy*** because it says faulting is mutually exclusive with non-fault outcomes.

We use the notation for allowed dependence in the read effect condition (f). However, since the read effect of a pure method refers only to final values, not states, we cannot use that notation in (c). Apropos (a) and (d), the negated satisfaction is equivalent to saying $[\![P]\!]_\varphi[\sigma + x{:}\, v]$ is $\lightning$ or *false*, as per (5). 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.[14] 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.

Recall from Section 1 that a correctness judgment includes a partial candidate ψ that interprets zero or more of the pure methods in the method context Φ of the judgment. The judgment makes a claim about program executions (using transition semantics, Figure 9), which rely on a candidate interpretation φ defined on all methods, pure and impure, of Φ. Validity of a judgment is in terms

[13]We do not want literals in formulas, as otherwise we would lose the agreement lemmas or else need to include literal values in read effects. For clarity, we refrain from using reference literals (other than null) anywhere.

[14]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.

of those φs that are Φ-interpretations (Definition 5.1). To define validity, the last ingredient is to connect the partial candidate ψ in the judgment with the context interpretations φ over which the judgment quantifies. 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 $\Phi; \psi \vdash^{\Gamma} C : P \leadsto Q\ [\varepsilon]$ is ***valid*** iff the following conditions hold for all Φ-interpretations φ such that φ extends ψ, and all states σ such that $\sigma \models_{\varphi}^{\Gamma, sigs(\Phi)} P$.

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

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

Because the judgment is swf, Γ is method-free hence the only relevant method environment is the empty one, written _ . In (Post) and (Read) we omit Γ, *sigs*(Φ) from $\models_{\varphi}$. In (Read), 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. As anticipated in the discussion following (8), note that if $\langle C,\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \tau,\ \_\rangle$ and $\langle C,\ \sigma',\ \_\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \tau',\ \_\rangle$ as in the antecedent of (Read), and the initial states satisfy the precondition P, then (Read) can also be instantiated with the states swapped, to obtain $\sigma', \sigma \Rightarrow \tau', \tau \models_{\varphi} \varepsilon$.

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. As remarked in Section 1.3, 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 $\sigma \models_{\varphi} df(P, \Phi)$ then $[\![P]\!]_{\varphi}\sigma$ is not $\lightning$. (b) For any σ and any Φ-interpretation φ, if $\sigma \models_{\varphi} df(P, \Phi)$ 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 $\sigma \models_{\varphi} df(F, \Phi)$ then $[\![F]\!]_{\varphi}\sigma$ is not $\lightning$.

The proof of lemma A goes by (structural) 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 $\lightning$).

$\mathit{df}(m(F),\Phi) = \mathit{df}(F,\Phi) \wedge P^x_F$, where $\Phi(m) = (x : T, \mathsf{res} : U)P \rightsquigarrow Q\ [\varepsilon]$. Thus $\sigma \models_\varphi \mathit{df}(F,\Phi)$, hence by induction hypothesis, $[\![F]\!]_\varphi\sigma$ is not $\lightning$. Let $v = [\![F]\!]_\varphi\sigma$, so $[\![m(F)]\!]_\varphi\sigma = \varphi(m)(\sigma, v)$. We have $\sigma \models_\varphi P^x_F$, so by Definition 5.1 (a), $\varphi(m)(\sigma, v)$ 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), that is, $\sigma \models_\varphi \mathit{df}(P,\Phi)$ implies $[\![P]\!]_\varphi\sigma \neq \lightning$. The proof goes by induction on P, using the definitions in Figures 5 and 7.

For base case $x.f = E$, the points-to relation, we have $\mathit{df}(x.f = E,\Phi) = (x \neq \mathsf{null} \Rightarrow \mathit{df}(E,\Phi))$. Thus $\sigma(x) \neq \mathsf{null} \Rightarrow \sigma \models_\varphi \mathit{df}(E,\Phi)$. By Lemma A, if $\sigma(x) \neq \mathsf{null}$, then $[\![E]\!]_\varphi\sigma$ is not $\lightning$. On the other hand, by semantics, 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 $\Gamma \vdash \forall x : K \in G \cdot P$. We have $\mathit{df}(\forall x : K \in G \cdot P,\Phi) = \mathit{df}(G,\Phi) \wedge (\forall x : K \in G \cdot \mathit{df}(P,\Phi))$. Thus $\sigma \models_\varphi \mathit{df}(G,\Phi)$. By Lemma A, $[\![G]\!]_\varphi\sigma$ is not $\lightning$. Also, we have $[\sigma + x\!: o] \models_\varphi \mathit{df}(P,\Phi)$, for all $o \in ([\![G]\!]_\varphi\sigma) \setminus \{\mathsf{null}\}$ with $\mathit{Type}(o,\sigma) = K$. By induction hypothesis, $[\![P]\!]_\varphi[\sigma + x\!: o]$ is not $\lightning$, for all $o \in ([\![G]\!]_\varphi\sigma) \setminus \{\mathsf{null}\}$ with $\mathit{Type}(o,\sigma) = K$. So $[\![\Gamma \vdash \forall x : K \in G \cdot P]\!]_\varphi\sigma$ is not $\lightning$.

Case $P_1 \wedge P_2$. We have $\mathit{df}(P_1 \wedge P_2,\Phi) = \mathit{df}(P_1,\Phi) \wedge (P_1 \Rightarrow \mathit{df}(P_2,\Phi))$. Thus $\sigma \models_\varphi \mathit{df}(P_1,\Phi)$ and $\sigma \models_\varphi (P_1 \Rightarrow \mathit{df}(P_2,\Phi))$. 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 $\sigma \models_\varphi \mathit{df}(P_2,\Phi)$ and by the induction hypothesis on P_2, $[\![P_2]\!]_\varphi\sigma \neq \lightning$. Thus $[\![P_1 \wedge P_2]\!]_\varphi\sigma = [\![P_2]\!]_\varphi\sigma$, 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 Γ, $\mathit{sigs}(\Phi)$. 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.* (6).

The term '$\Phi;\psi$-valid' elides dependency on Γ, but the relevant typing context should always be clear. 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 VC-gen 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 $\mathit{df}(P,\Phi)$ is $\Phi;\psi$-valid. A swf impure method specification $(x\!:\!T)P \rightsquigarrow Q\ [\eta]$ is ***healthy*** (with respect to Γ,Φ,ψ) iff the three formulas $\mathit{df}(P,\Phi)$, $\mathit{df}(Q,\Phi)$, and $P \Rightarrow \mathit{df}(\eta,\Phi)$ are $\Phi;\psi$-valid. A swf pure method specification $(x\!:\!T, \mathsf{res}\!:\!U)P \rightsquigarrow Q\ [\eta]$ is ***healthy*** if $\mathit{df}(P,\Phi)$, $P \Rightarrow \mathit{df}(Q,\Phi)$, and $P \Rightarrow \mathit{df}(\eta,\Phi)$ 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 formulas $\mathit{df}(P,\Phi)$ and $P \Rightarrow \mathit{df}(\eta,\Phi)$ are $\Phi;\psi$-valid, and
- either $\mathit{df}(Q,\Phi)$ is $\Phi;\psi$-valid or η has no writes other than $\mathsf{wr}\ \mathsf{res}$ and $P \Rightarrow \mathit{df}(Q,\Phi)$ is $\Phi;\psi$-valid.

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

Note that for impure methods, and for correctness judgments, the definition requires the validity of $\mathit{df}(Q,\Phi)$ rather than the weaker condition $P \Rightarrow \mathit{df}(Q,\Phi)$. In the case of a pure method, the pre and post conditions are applied to the same state, usually one that satisfies the precondition P. But for impure methods and for correctness judgments, the post-state is typically not the same as the pre-state. In reasoning about Q in the post state, we cannot rely on P holding, so the weaker condition $P \Rightarrow df(Q, \Phi)$ would be inadequate. The last part of the definition is to cater to judgments for pure methods, making the condition on judgments consistent with the condition on specifications in accord with Definition 2.1.

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 a candidate interpretation of the 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.

Remark on valid formulas. Definition 5.4 says that $\Phi; \psi \models P$ if $\sigma \models_\varphi P$ for all states σ (and all Φ-interpretations φ that extend ψ). Here and throughout the paper, states are required to be type correct and self-contained in the sense that there are no dangling references; moreover the value of alloc is exactly the set of allocated references in the heap. So, in addition to first-order tautologies and consequences of Φ, there are some valid formulas concerning alloc, such as $x \in \mathsf{alloc} \cup \{\mathsf{null}\}$ for any reference type variable x in scope. Also $\forall x : K \in G \cdot x.f \in \mathsf{alloc} \cup \{\mathsf{null}\} \land x.g \subseteq \mathsf{alloc} \cup \{\mathsf{null}\}$ for reference (resp. region) typed field f (resp. g).

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 new notion which plays a role similar to immunity but for read effects. Technically, framed reads restores symmetry to allowed dependence.

Concerning the subeffect and framing judgments, we start with their semantics, which is amenable to direct checking using an SMT solver. Then proof rules are given for deriving the judgments syntactically (following RLI). 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 candidate ψ for Φ, 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 $rlocs(\sigma, \varphi, \varepsilon) \subseteq rlocs(\sigma, \varphi, \eta)$ and $wlocs(\sigma, \varphi, \varepsilon) \subseteq wlocs(\sigma, \varphi, \eta)$.

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

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

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

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

Fig. 10. Selected rules for well-formed subeffect judgments. We write $\lesseqgtr$ to abbreviate two inclusion rules.

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

The first two parts are immediate from definitions. To show the third part, (allowed dependency), suppose $\sigma, \sigma' \Rightarrow \tau, \tau' \models_\varphi \varepsilon$ and consider any $\sigma, \sigma', \tau, \tau', \pi$ 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 there is $\rho \supseteq \pi$ such that $\rho(\mathit{freshRefs}(\sigma, \tau)) \subseteq \mathit{freshRefs}(\sigma', \tau')$ and $\mathit{Lagree}(\tau, \tau', \rho, \mathit{freshLocs}(\sigma, \tau) \cup \mathit{written}(\sigma, \tau))$.
Figure 10 provide rules for subeffecting, to be applied to well-formed subeffect judgments.

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.
Recall that for clarity we often abuse notation and treat compound effects as sets. As a consequence, it is immediate to derive judgments of the forms $P; \Phi; \psi \vdash \varepsilon, \varepsilon \leq \varepsilon$ and $P; \Phi; \psi \vdash \varepsilon, \eta \leq \eta, \varepsilon$. We get $\mathit{true}; \Phi; \psi \vdash \varepsilon \leq \varepsilon$ by instantiating $\mathit{true}; \Phi; \psi \vdash \varepsilon \leq \varepsilon, \eta$ with η as the empty effect.

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 $\mathit{df}(P, \Phi)$, $P \Rightarrow \mathit{df}(\eta, \Phi)$, and $P \Rightarrow \mathit{df}(Q, \Phi)$ 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 $\mathit{Agree}(\sigma, \tau, \eta, \pi, \varphi)$, and $\sigma \models^\Gamma_\varphi P \land Q$, then $\tau \models^\Gamma_\varphi Q$.

In this article, it would suffice to define frame validity in terms of the identity refperm on the initial references $\sigma(\mathsf{alloc})$, as that suffices for its use in the frame rule. The extra generality has

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

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

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

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

little cost and is convenient because we need the general form of agreement in order to formulate quasi-determinacy (Section A.1). It is also useful in relational logic [7].

A verifier can check framing judgments in terms of the validity property (see Sec. 10 and [53]), but our logic includes rules to derive framing judgments. A basic rule allows to infer, for atomic formula P, the judgment $\mathit{true}; \Phi; \psi \vdash \mathit{ftpt}(P, \Phi)$ 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 $\mathit{df}(F, \Phi)$ is valid and $\mathit{Agree}(\sigma, \sigma', \mathit{ftpt}(F, \Phi), \pi, \varphi)$. Then $[\![F]\!]_\varphi \sigma \overset{\pi}{\sim} [\![F]\!]_\varphi \sigma'$.*

Separator formulas and immunity. The point of establishing $P; \Phi; \psi \vdash \eta$ 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 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 $\eta \cdot\!/\!. \varepsilon$ is defined in Figure 11 using function $\cdot\!/\!.$ which recurses on effects. The most interesting case is the first line: $\mathsf{rd}\ G\text{`}f \ \cdot\!/\!. \ \mathsf{wr}\ H\text{`}f$ is the disjointness formula $G \mathbin{\#} H$. (Throughout the paper, $\equiv$ means syntactic identity.) We use the data group any to abstract from all field names, so $\mathsf{rd}\ G\text{`}\mathsf{any} \ \cdot\!/\!. \ \mathsf{wr}\ H\text{`}f$ is $G \mathbin{\#} H$ for any f. Writes on the left and reads on the right are ignored, so $\eta \cdot\!/\!. \varepsilon$ is the same as $\mathit{reads}(\eta) \cdot\!/\!. \mathit{writes}(\varepsilon)$. Note that $G \mathbin{\#} H$ 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:

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

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 $\sigma \models_\psi \eta \ \cdot\!/\!. \ \varepsilon$. Then $\mathit{Agree}(\sigma, \tau, \eta, \mathit{id}, \psi)$, where id is the identity on $\sigma(\mathsf{alloc})$.*

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

$$\textsc{FrmDisj}\ \frac{P \vdash \varepsilon \text{ frm } Q_1 \qquad P \vdash \varepsilon \text{ frm } Q_2}{P \vdash \varepsilon \text{ frm } Q_1 \vee Q_2} \qquad \textsc{FrmConj}\ \frac{P \vdash \varepsilon \text{ frm } Q_1 \qquad P \wedge Q_1 \vdash \varepsilon \text{ frm } Q_2}{P \vdash \varepsilon \text{ frm } Q_1 \wedge Q_2}$$

$$\textsc{Frm}\forall_{\mathsf{int}}\ \frac{P \vdash^{\Gamma, x:\mathsf{int}} \varepsilon, \mathsf{rd}\ x \text{ frm } Q}{P \vdash^{\Gamma} \varepsilon \text{ frm } \forall x : \mathsf{int} \cdot Q}$$

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

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

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

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.

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:

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

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

For example, $\mathsf{wr}\ x$ and $\mathsf{rd}\ x$ are $\mathit{true};\Phi;\psi/\mathsf{wr}\ x$-immune vacuously. On the other hand, $\mathsf{wr}\ \{x\}`f$ is not $\mathit{true};\Phi;\psi/\mathsf{wr}\ x$-immune because the region expression $\{x\}$ in $\mathsf{wr}\ \{x\}`f$ is not: $\mathit{ftpt}(\{x\}, \Phi) \cdot/. \ \mathsf{wr}\ x = \mathsf{rd}\ x \cdot/. \ \mathsf{wr}\ x = \mathit{false}$. In contrast, $\mathsf{wr}\ \{x\}`f$ 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 $\mathit{wlocs}(\sigma, \varphi, \eta) = \mathit{wlocs}(\sigma', \varphi, \eta)$.

Framing rules. Figure 13 specifies (mostly) syntax-directed rules for the framing judgment $P;\Phi;\psi \vdash \varepsilon \text{ frm } Q$. 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 $P;\Phi;\psi \vdash \varepsilon \text{ frm } Q_1 \wedge Q_2$

from $P;\Phi;\psi \vdash \varepsilon$ frm Q_1 and $P \wedge Q_1;\Phi;\psi \vdash \varepsilon$ frm Q_2. There are also subsidiary rules for subsumption of effects and for logical manipulation of P. The latter means that $P;\Phi;\psi \vdash \varepsilon$ frm Q may be inferred from $R;\Phi;\psi \vdash \varepsilon$ 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 derivations, using soundness of the rules. Soundness of the rules is proved using using Lemmas 6.3 and 6.5.

6.3 Framed reads

An effect ε is said to have ***framed reads***, in method context Φ, provided that for every rd G‘f in ε, its footprint *ftpt*(G,Φ) is in ε. For example, with r:rgn the effect rd r‘f does not have framed reads, but it is a subeffect of rd r‘f, rd r which does.

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, which hinges critically on symmetry of allowed dependence. It seems advisable for most judgments and specifications to have framed reads, but not all, as discussed later in connection with the proof rule for sequence. We prove key properties of effects with framed reads and then give examples showing the necessity of framed reads for these properties.

If the frame conditions of pure methods in Φ have framed reads then *ftpt*(F,Φ) has framed reads, for any expression F. This carries over to framing judgments for formulas derivable by the rules in Figure 13, with the exception of some uses of the subeffect rule.

For ε that has framed reads, if *Agree*$(\sigma,\sigma',\varepsilon,\pi,\theta)$ then $[\![G]\!]_\theta\sigma \overset{\pi}{\sim} [\![G]\!]_\theta\sigma'$ for any rd G‘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 $df(\varepsilon,\Phi)$ is $\Phi;\varphi$-valid. Consider any states σ,σ' and any refperm π such that *Agree*$(\sigma,\sigma',\varepsilon,\pi,\varphi)$. Then

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

Proof. (a) For variables the equality follows immediately by definition of *rlocs* and definition (7). For heap locations the argument is by mutual inclusion. To show *rlocs*$(\sigma',\varphi,\varepsilon) \subseteq \pi($*rlocs*$(\sigma,\varphi,\varepsilon))$, let $o.f \in$ *rlocs*$(\sigma',\varphi,\varepsilon)$. By definition of *rlocs*, there exists region G such that ε contains rd G‘f and $o \in [\![G]\!]_\varphi\sigma'$. Since ε has framed reads, ε contains *ftpt*(G,Φ), hence from *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 $o.f \in \pi($*rlocs*$(\sigma,\varphi,\varepsilon))$. Proof of the reverse inclusion is similar.

(b) For variables this is straightforward. For heap locations, consider any $o.f \in$ *rlocs*$(\sigma',\varphi,\varepsilon)$. From (a), we have $\pi^{-1}(o).f \in$ *rlocs*$(\sigma,\varphi,\varepsilon)$. From *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 example following (9) in Section 4 shows that the lack of framed reads leads to asymmetry of agreement.

The second critical, but non-obvious, property is that for a pair of states σ, σ' that are in 'symmetric' agreement and transition to a pair τ, τ' forming an allowed dependence, the transitions preserve 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 $\sigma', \sigma \Rightarrow \tau', \tau \models_\varphi \varepsilon$. Let $Agree(\sigma, \sigma', \varepsilon, \pi, \varphi)$ and $Agree(\sigma', \sigma, \varepsilon, \pi^{-1}, \varphi)$. Let ρ be any refperm $\rho \supseteq \pi$ for which

$$Lagree(\tau, \tau', \rho, freshLocs(\sigma, \tau) \cup written(\sigma, \tau)) \tag{10}$$

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

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

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

$$Lagree(\tau', \tau, \rho', freshLocs(\sigma', \tau') \cup written(\sigma', \tau')) \tag{11}$$

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

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

- If $x \in written(\sigma, \tau)$ then from from (10), we have $\tau(x) \overset{\rho}{\sim} \tau'(x)$.
- If $\sigma(x) = \tau(x)$, we claim that $\sigma'(x) = \tau'(x)$. Then from $Lagree(\sigma, \sigma', \pi, W)$ 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 $x \in written(\sigma', \tau')$. By (11) 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 $o.f \in written(\sigma, \tau)$ or $\sigma(o.f) = \tau(o.f)$.

- If $o.f \in written(\sigma, \tau)$ then from (10), we have $\tau(o.f) \overset{\rho}{\sim} \tau'(\rho(o).f)$.
- If $\sigma(o.f) = \tau(o.f)$, we claim that $\sigma'(\pi(o).f) = \tau'(\pi(o).f)$. Then from $Lagree(\sigma, \sigma', \pi, W)$ 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 $\pi(o).f \in written(\sigma', \tau')$. By (11) 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 $Lagree(\tau, \tau', \pi, W)$ for heap locations. □

Lemma 6.13 (freshness symmetry). 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 $Agree(\sigma, \sigma', \varepsilon, \pi, \varphi)$ and $Agree(\sigma', \sigma, \varepsilon, \pi^{-1}, \varphi)$. Let ρ be any refperm $\rho \supseteq \pi$ for which

$$\begin{array}{l} \rho(freshLocs(\sigma, \tau)) \subseteq freshLocs(\sigma', \tau') \text{ and} \\ Lagree(\tau, \tau', \rho, freshLocs(\sigma, \tau) \cup written(\sigma, \tau)) \end{array} \tag{12}$$

Then we have $Lagree(\tau', \tau, \rho^{-1}, freshLocs(\sigma', \tau'))$.

Proof. From $Agree(\sigma', \sigma, \varepsilon, \pi^{-1}, \varphi)$ and $\sigma', \sigma \Rightarrow \tau', \tau \models_\varphi \varepsilon$, there is a refperm $\rho' \supseteq \pi^{-1}$ for which $\rho'(freshLocs(\sigma', \tau')) \subseteq freshLocs(\sigma, \tau)$ and $Lagree(\tau', \tau, \rho', freshLocs(\sigma', \tau') \cup written(\sigma', \tau'))$. From (10), we have $\rho(freshLocs(\sigma, \tau)) \subseteq freshLocs(\sigma', \tau')$. From Definition 4.2, we know that ρ and ρ' are total on $freshLocs(\sigma, \tau)$ and $freshLocs(\sigma', \tau')$ respectively. Since ρ and ρ' are bijections, we have $|freshLocs(\sigma, \tau)| = |freshLocs(\sigma', \tau')|$. So we get $\rho(freshLocs(\sigma, \tau)) = freshLocs(\sigma', \tau')$. Now from (12) using (8) we get $Lagree(\tau', \tau, \rho^{-1}, \rho(freshLocs(\sigma, \tau)))$. Hence $Lagree(\tau', \tau, \rho^{-1}, freshLocs(\sigma', \tau'))$. □

Example 6.14. 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

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

and consider type K with $\mathit{Fields}(K) = \{f : \mathsf{int}\}$. In this example we suppose that that method context and its candidate interpretation are empty and we omit them. Now consider the 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 $\mathit{written}(\sigma', \tau') = \{o.f\}$ and $\tau(o.f) \neq \tau'(o.f)$. Thus we do not have

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

This shows that allowed dependence is not symmetric: $\sigma, \sigma' \Rrightarrow \tau, \tau' \models \mathsf{rd}\, r\text{‘}f$ but not $\sigma', \sigma \Rrightarrow \tau', \tau \models \mathsf{rd}\, r\text{‘}f$. 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 the states in this example do not arise in any method interpretation, because an interpretation is required to satisfy the allowed dependency both ways around (that is, the bound variables σ, σ' in Def. 5.1 can be instantiated by both σ, σ' and σ', σ above). The next example is another where agreement fails to be preserved. 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 conditions in the proof rules for sequence and iteration.

Example 6.15. This example builds on (9) 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

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

for impure parameterless m. Let the precondition be 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 does not contain any pure methods, we use the empty interpretation, and omit it. Consider distinct references o, p and the following states:

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

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 $\sigma(p.f) \neq 5 = \sigma'(p.f)$, 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 $\sigma, \sigma' \Rrightarrow \tau, \tau' \models \mathsf{rw}\, r\text{‘}f$. We also have the symmetric instance of allowed dependence, that is, $\sigma', \sigma \Rrightarrow \tau', \tau \models \mathsf{rw}\, r\text{‘}f$, because the antecedent in its definition, $\mathit{Agree}(\sigma', \sigma, \mathsf{rw}\, r\text{‘}f, \pi)$, is false for all π. 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 6.16. Although Example 6.15 seems contrived, the states $\sigma, \sigma', \tau, \tau'$ can in fact arise in programs. Here we give an interpretation φ, of Φ in Example 6.15, such that $\varphi(m)(\sigma) = \{\tau\}$ and $\varphi(m)(\sigma') = \{\tau'\}$. Consider the map defined for all states v by

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

For brevity in this example we omit the argument value; strictly, we should define $\varphi(m)(v)(v)$ as above, independent of value v. 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 $v \in [\![\Gamma]\!]$.

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

7 PROOF SYSTEM FOR PROGRAM CORRECTNESS

This section gives the proof system and works out an example. Soundness is proved in Section 8.

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 notion which was introduced in Section1.3 in connection with (2).

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.[15]

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 $\cup$.

7.1 The proof system

Figures 14 and 15 present the 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

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

$$\textsc{FieldUpd}\quad \Phi;\vdash x.f := y :\ x \neq \mathsf{null} \leadsto x.f = y\ [\mathsf{wr}\, x.f, \mathsf{rd}\, x, \mathsf{rd}\, y]$$

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

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

$$\textsc{Alloc}\ \frac{\mathit{Fields}(K) = \overline{f} : \overline{T}}{\Phi;\vdash x := \mathsf{new}\ K :\ r = \mathsf{alloc} \leadsto x \notin r \land \mathsf{alloc} = r \cup \{x\} \land x.\overline{f} = \mathit{default}(\overline{T})\ [\mathsf{wr}\, x, \mathsf{rw}\,\mathsf{alloc}]}$$

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

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

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

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

TranspPureLink — same as PureLink except $\Phi,\Theta;\psi,\theta \vdash^{\Gamma} C :\ P \leadsto Q\ [\varepsilon]$

$$\textsc{Var}\ \frac{\Phi;\psi \vdash^{\Gamma,x:T} C :\ P \land x = \mathit{default}(T) \leadsto P'\ [\mathsf{rw}\, x, \varepsilon]}{\Phi;\psi \vdash^{\Gamma} \mathsf{var}\ x : T\ \mathsf{in}\ C :\ P \leadsto P'\ [\varepsilon]}$$

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

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

$$\textsc{If}\ \frac{\Phi;\psi \vdash C_1 :\ P \land E \neq 0 \leadsto P'\ [\varepsilon] \qquad \Phi;\psi \vdash C_2 :\ P \land E = 0 \leadsto P'\ [\varepsilon]}{\Phi;\psi \vdash \mathsf{if}\ E\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 :\ P \leadsto P'\ [\varepsilon, \mathit{ftpt}(E)]}$$

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

$$\textsc{Frame}\ \frac{\Phi;\psi \vdash C : P \leadsto Q\ [\varepsilon] \qquad P;\Phi;\psi \models \eta \ \mathsf{frm}\ R \qquad \Phi;\psi \models P \wedge R \Rightarrow \eta \cdot\!/\!. \ \varepsilon}{\Phi;\psi \vdash C : P \wedge R \leadsto Q \wedge R\ [\varepsilon]}$$

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

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

$$\textsc{Conj}\ \frac{\Phi;\psi \vdash C : P_1 \leadsto Q_1\ [\varepsilon] \qquad \Phi;\psi \vdash C : P_2 \leadsto Q_2\ [\varepsilon]}{\Phi;\psi \vdash C : P_1 \wedge P_2 \leadsto Q_1 \wedge Q_2\ [\varepsilon]}$$

$$\textsc{Exist}\ \frac{\Phi;\psi \vdash^{\Gamma, x:K} C : x \in G \wedge P \leadsto Q\ [\varepsilon]}{\Phi;\psi \vdash^{\Gamma} C : (\exists x : K \in G \cdot P) \leadsto Q\ [\varepsilon]}$$

$$\textsc{ExistRegion}\ \frac{\Phi;\psi \vdash^{\Gamma, x:\mathsf{rgn}} C : x = F \wedge P \leadsto Q\ [\varepsilon]}{\Phi;\psi \vdash^{\Gamma} C : P^x_F \leadsto Q\ [\varepsilon]}$$

Fig. 15. Structural proof rules

with well-formed premises and conclusion, the conclusion is valid if the premises are valid and the side conditions hold. In the soundness proof we make pervasive use of the healthiness of judgments. It ensures that definedness formulas hold where they are needed, so we can use two-valued reasoning (in light of Lemma 5.3).

Theorem 7.2. *The rules in Figures 14 and 15 are sound.*

An immediate corollary is that every derivable correctness judgment is valid. The proof is given in Sections 8, with some additional cases in appendices.

Here are a few comments on the rules, in order of their appearance in the figures. The first rule, for field update, is a 'local axiom' that precisely describes the effect. Note that the reference variable x is read, in order to write the field $x.f$. Readers familiar with RLI (Section 7.1) and RLII (Section 7.1) may notice that the addition of read effects for this and most subsequent rules is straightforward.

Rule FieldAcc, is another local axiom. It uses an extra variable to refer to the initial value of y, for soundness in the case that $y \equiv x$. In case $y \not\equiv x$, one can derive the convenient axiom $\Phi; \vdash x := y.f : y \neq \mathsf{null} \leadsto x = y.f\ [\mathsf{wr}\ x, \mathsf{rd}\ y, \mathsf{rd}\ y.f]$ as noted in RLI.

Rule Assign is formulated using the *ftpt* function to compute the read effect (Figure 12).

Rule Alloc 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.[16] Note that the command reads alloc.

Rule Alloc has some important consequences. First, we can derive that the allocated object is disequal from existing ones in variables. For y of reference type (with $y \not\equiv x$), the formula

[16]In RLI the postcondition also includes an assertion type$(K, \{x\})$ that x has type K but in this article we refrain from including that among the primitive formulas. In RLI, the allocation rule is formulated using a freshness effect. It is shown there that a rule like Alloc is derivable.

$y \in \mathsf{alloc} \cup \{\mathsf{null}\}$ is valid (see remark at the end of Section 5). So the formulas $r = \mathsf{alloc}$ and $r = \mathsf{alloc} \land y \in r \cup \{\mathsf{null}\}$ are logically equivalent. Hence we can use Conseq to add $y \in r \cup \{\mathsf{null}\}$ to the precondition (and to drop irrelevant postconditions):

$$\vdash x := \mathsf{new}\ K : r = \mathsf{alloc} \land y \in r \cup \{\mathsf{null}\} \leadsto x \notin r \cup \{\mathsf{null}\}\ [\mathsf{wr}\ x, \mathsf{rw}\ \mathsf{alloc}]$$

Then since y and r are not written we can use Frame to get

$$\vdash x := \mathsf{new}\ K : r = \mathsf{alloc} \land y \in r \cup \{\mathsf{null}\} \leadsto x \notin r \cup \{\mathsf{null}\} \land y \in r \cup \{\mathsf{null}\}\ [\mathsf{wr}\ x, \mathsf{rw}\ \mathsf{alloc}]$$

Now use Conseq to simplify the precondition and weaken the postcondition, to get

$$\vdash x := \mathsf{new}\ K : r = \mathsf{alloc} \leadsto x \neq y\ [\mathsf{wr}\ x, \mathsf{rw}\ \mathsf{alloc}]$$

which says y is distinct from the fresh reference. Furthermore, since r does not occur in the frame or postcondition, we can use rule ExistRegion together with Conseq to eliminate the precondition $r = \mathsf{alloc}$. That yields a judgment in which alloc only occurs in the frame condition. Using similar derivations, one can show $x \notin s$ for region variable s, and $x \neq y.g$ or $x \notin y.g$ for a field. These postconditions can be combined with that of Alloc through use of rule Conj.

A formula may refer to fields of a variable bound by a quantifier, and again we can derive that a fresh reference is distinct from existing ones. For example, this formula is valid: $\forall z : K' \in \mathsf{alloc} \cdot z.f \in \mathsf{alloc} \cup \{\mathsf{null}\} \land z.g \subseteq \mathsf{alloc} \cup \{\mathsf{null}\}$ (for f of reference type and g of region type). So the precondition $r = \mathsf{alloc}$ of Alloc is logically equivalent to $r = \mathsf{alloc} \land P$ where

$$P \ \widehat{=}\ \forall z : K' \in \mathsf{alloc} \cdot z.f \in r \cup \{\mathsf{null}\} \land z.g \subseteq r \cup \{\mathsf{null}\}$$

By Conseq and Frame we get

$$\vdash x := \mathsf{new}\ K : r = \mathsf{alloc} \land P \leadsto x \notin r \cup \{\mathsf{null}\} \land P\ [\mathsf{wr}\ x, \mathsf{rw}\ \mathsf{alloc}]$$

Now use Conseq to simplify the precondition and weaken the postcondition, to get

$$\vdash x := \mathsf{new}\ K : r = \mathsf{alloc} \leadsto \forall z : K' \in \mathsf{alloc} \cdot x \neq z.f \land x \notin z.g\ [\mathsf{wr}\ x, \mathsf{rw}\ \mathsf{alloc}]$$

Because for any G the formula $G \subseteq \mathsf{alloc}$ is valid, we can now use Conseq and Exist Region to obtain $\vdash x := \mathsf{new}\ K : \mathit{true} \leadsto \forall z : K' \in G \cdot x \neq z.f \land x \notin z.g\ [\mathsf{wr}\ x, \mathsf{rw}\ \mathsf{alloc}]$ where again alloc appears only in the frame condition.

For future reference, we summarize these considerations by the following derivable rule.

$$\textsc{Alloc1}\ \frac{y \not\equiv x \qquad \mathit{Fields}(K) = \overline{f} : \overline{T} \qquad f : K'' \text{ is in } \mathit{Fields}(K') \qquad g : \mathsf{rgn} \text{ is in } \mathit{Fields}(K')}{\Phi; \vdash x := \mathsf{new}\ K : r = \mathsf{alloc} \leadsto \begin{array}{l} x \neq y \land x \notin r \land \mathsf{alloc} = r \cup \{x\} \land x.\overline{f} = \mathit{default}(\overline{T}) \\ \land (\forall z : K' \in \mathsf{alloc} \cdot x \neq z.f \land x \notin z.g) \end{array} [\mathsf{wr}\ x, \mathsf{rw}\ \mathsf{alloc}]}$$

Although our small axioms for assignment are as succinct as those of separation logic, the axiom Alloc is not quite as beautiful since its use in context involves more than mere use of the frame rule.

Apropos rule PureCall, the command $y := m(z)$ is an assignment and as such establishes postcondition $y = m(z)$. Calling it out as a special case enables us to also assert the postcondition Q. The variable condition $y \notin FV(Q)$, that is, y not free in Q, is included for clarity; It actually follows from specifications being swf, using the distinction between local and global variables mentioned in Footnote 7.

Rules PureLink and ImpureLink are for linking a client with a single method implementation, either pure or impure. As discussed in Section 1.3, 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. This addresses the use of pure methods for abstraction, e.g., in situations where the pure method should be ‘opaque’ because its definition acts on state that is not in scope or reachable for the client. As noted in connection with equation (3), there are also situations in which it is appropriate for the interpretation to be visible to clients, addressed by this 'transparent' linking rule rule TRANSPPURELINK. It is not a derived rule, because the weaker judgment $\Phi, \Theta; \psi, \theta \vdash^{\Gamma} C : P \leadsto Q\ [\varepsilon]$ does not entail the stronger one that requires C to be correct for all interpretations of m.

In general, it is necessary to simultaneously link several pure and impure methods—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). Simultaneous linking is needed for multiple pure methods that share a data representation and need to have compatible interpretations (*cf.* Definition 7.1). However, owing to the observation in (4) following Definition 2.2, it is sufficient to simultaneously link a set of impure methods, after which a set of pure methods can be linked. So it suffices to have one rule like PURELINK but linking multiple pure methods, and one like IMPURELINK but linking multiple impure ones. In these general rules, each method body must satisfy its specification, in the context of all of the specifications. This can be seen in the examples in Section 7.3. Soundness of these rules can be proved by a straightforward generalization of the proofs in this article.

Rule VAR for local variable blocks is adapted straightforwardly from RLI. The premise allows reading and writing the local variable; these effects are dropped in the conclusion. This makes it possible for a pure method to traverse heap structure using loops—indeed, quite complicated algorithms can be pure methods if the language is extended to allow local variables of mathematical types like sequences. For example, one can implement the anc function of Fig. 2 using a loop that traverses parent pointers using an accumulator variable (or res itself) and a variable that points at the current node.

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 read effects, but in addition we need effects to have framed reads. 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 conditions. Note that framed reads are not required for the effects involving the expression H to reason about writes to freshly allocated objects. This is important, because it allows H to itself be expressed in terms of freshly allocated objects. For an example, see the use of expression H_3 in the proof of the client of Cell in Section 7.3.

About WHILE, note that $\mathsf{rd}\,x$ may be in ε but need not be if the loop body does not read x. In useful code, ε will contain $\mathsf{rw}\,x$. The grammar allows a program expression E for the guard condition. For proving soundness it is convenient, and loses no generality, to restrict to the case of a variable x.

The remaining rules, in Fig. 15, are for general reasoning about judgments. The adaptation of rule FRAME from RLI/II is straightforward: adding partial candidate ψ and adding context $\Phi; \psi$ for the side conditions. Similarly for the other structural rules.

Apropos FRAME, observe that $x := \mathsf{new}\ \mathit{Cell}$ satisfies $\mathsf{true} \leadsto \mathsf{true}\ [\mathsf{rw\,alloc}, \mathsf{wr}\,x]$ but it does not satisfy

$$\mathsf{true} \wedge (\forall x : \mathit{Cell} \in \mathsf{alloc}\ \cdot\ \mathit{false}) \leadsto \mathsf{true} \wedge (\forall x : \mathit{Cell} \in \mathsf{alloc}\ \cdot\ \mathit{false})\ [\mathsf{rw\,alloc}, \mathsf{wr}\,x] \qquad (13)$$

It also does not satisfy the equivalent judgment $\mathsf{alloc} = \varnothing \leadsto \mathsf{alloc} = \varnothing\ [\mathsf{rw\,alloc}, \mathsf{wr}\,x]$. This is not surprising: the FRAME rule is inapplicable in these cases, because any frame judgment for either formula includes $\mathsf{rd\,alloc}$, and the command writes alloc. We return to this point in connection with weak purity (Section 11).

In this article the assertion language does not include quantification at type rgn. So to achieve the effect of rule Exist for region variables we need rule ExistRegion (just as in RLI/II).

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

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

Dropping 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 6.15:

$$\Phi \ \widehat{=}\ m() : P \leadsto \mathsf{true}\,[\mathsf{rw}\,r`f]$$
$$P \ \widehat{=}\ 1 \le |r| \le 2 \land (\forall a, b : K \cdot a.f = 3 \land b.f = 5 \Rightarrow r = \{a, b\})$$

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

$$\Phi; \varnothing \vdash m() : P \land x \ne \mathsf{null} \leadsto \mathsf{x} \ne \mathsf{null}\,[\mathsf{rw}\,r`f]. \tag{14}$$

For the second command, define

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

Consider the judgment

$$\Phi; \varnothing \vdash C_1 : x \ne \mathsf{null} \leadsto \mathsf{true}\,[\mathsf{rd}\,x.f, \mathsf{rd}\,x, \mathsf{rw}\,j]. \tag{15}$$

This can be derived using the proof rules; note in particular that the effect has framed reads. Using rule Seq on (14) and (15)—but ignoring the side condition that the first judgment has framed reads—would give us the judgment

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

We show that this judgment is invalid because the read effect property fails. Consider the interpretation φ from Example 6.16, and these states from Example 6.15:

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

We have $Agree(\sigma, \sigma', (\mathsf{rw}\,r`f, \mathsf{rd}\,x.f, \mathsf{rd}\,x, \mathsf{wr}\,j))$. 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 $j \in written(\sigma, \kappa)$, $\kappa(j) = -1$, and $\kappa'(j) = 1$. This contradicts $Lagree(\tau, \tau', \pi, freshLocs(\sigma, \tau) \cup written(\sigma, \tau))$.

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. This example shows the necessity for the second command in a sequence to have framed reads, except for locations freshly allocated by the first command in the sequence. We begin by considering this judgment with P and Φ as above:

$$\Phi; \varnothing \vdash x.f := 3 : P \land x \ne \mathsf{null} \leadsto P\,[\mathsf{rd}\,x, \mathsf{wr}\,x.f]. \tag{16}$$

This can be derived using FieldUpd, Frame, and Conseq, so it is valid. Using rule Seq on (14) and (16)—but ignoring the side condition that the effect rw r‘f of (14) has framed reads—yields

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

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 $Agree(\upsilon, \upsilon', id, \mathsf{rw}\ r\text{‘}f, \mathsf{rd}\ x)$. We also have transitions

$$\langle x.f := 3; m(),\ \upsilon,\ \_\rangle \xmapsto{\varphi} \langle m(),\ \sigma,\ \_\rangle \xmapsto{\varphi} \langle \mathsf{skip},\ \tau,\ \_\rangle$$

and $\langle x.f := 3; m(),\ \upsilon',\ \_\rangle \xmapsto{\varphi} \langle m(),\ \sigma',\ \_\rangle \xmapsto{\varphi} \langle \mathsf{skip},\ \tau',\ \_\rangle$. Here $\sigma, \sigma', \tau, \tau'$ are as in Example 7.3. Notice that $written(\upsilon, \tau) = \{o.f\}$ and $freshLocs(\upsilon, \tau) = \varnothing$. But since $\tau(o.f) = 3 \neq 6 = \tau'(o.f)$, we do not have $Lagree(\tau, \tau', id, freshLocs(\upsilon, \tau) \cup written(\upsilon, \tau))$.

In conclusion, without the requirement of framed reads for the second command in a sequence, we could derive invalid conclusions from valid premises. For an example showing why it is untenable to require all reads to be framed, see the use of H_3 in the verification of the client in Section 7.3.

7.3 Example proof: the Cell methods and a linked client

To illustrate features of some of the rules, we sketch proofs of the implementations of the Cell methods from Figure 16 and this linked client.

$$Cli \ \mathrel{\widehat{=}}\ d := \mathsf{new}\, Cell; init(d); set(c, 5); set(d, 4) \ : \ I \land c \neq \mathsf{null} \leadsto I \land get(c) = 5\ [\eta]$$

where $\eta \mathrel{\widehat{=}} \mathsf{rw}\ d, \mathsf{rw}\,\mathsf{alloc}, \mathsf{rw}\ c.foot\text{‘}\mathsf{any}, \mathsf{rd}\ c.foot, \mathsf{rd}\ cd$. More precisely, we aim to use the PureLink and ImpureLink rules to prove that the linked command (cf. (4))

$$\begin{array}{l} \mathsf{let}\ get(\mathsf{self} : Cell) : \mathsf{int} = \mathsf{res} := \mathsf{self}.value\ \mathsf{in} \\ \mathsf{let}\ set(\mathsf{self} : Cell, v : \mathsf{int}) = \mathsf{self}.value := v;\ \ init(\mathsf{self} : Cell) = \mathsf{self}.foot := \{\mathsf{self}\}\ \mathsf{in} \\ Cli \end{array} \tag{17}$$

has specification $I \ \land \ c \neq \mathsf{null} \leadsto I \ \land \ get(c) = 5[\eta]$. Recall that rw in Figure 16 abbreviates a read and write, *e.g.*, rw $c.foot$‘any abbreviates the effects rd $c.foot$‘any, wr $c.foot$‘any.

Method init serves as constructor for Cell. The implementation of init is simply self.$foot$:= {self}, which suffices for our implementation of set. But init's specification follows good practice, which is to allow allocation in constructors. For simplicity we treat constructors as ordinary methods, so the specification needs to explicitly require that self is fresh with respect to preexisting state of interest; in this example we need self $\notin x.foot$ for preexisting x.

In a richer specification language, I in the figure would be declared as a class invariant—a public one, because it is useful to clients and does not expose the internal representation.

As an abbreviation, define this variation on I:

$$J(s) \ \mathrel{\widehat{=}}\ \forall x, y : Cell \in s \ \cdot \ x \in x.foot \land (x = y \lor x.foot \ \#\ y.foot)$$

The parameter s is just a convenience so we can write $J(r)$ for the instantiation J^s_r, and similarly for other instantiations of s. Note that I and $J(r)$ respectively are framed by rd alloc, rd alloc‘$foot$, and rd r, rd r‘$foot$.

The judgment for (17) is considered in context

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

The ghost variable r, declared in Γ, is used as 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.

```
pure method get(self: Cell): int
  requires self ≠ null ∧ 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.foot=∅ ∧ ∀x:Cell ∈ alloc\{self} · self ∉ x.foot
  requires ∀x,y:Cell ∈ alloc\{self} · x ∈ x.foot ∧ (x=y ∨ x.foot # y.foot)
  ensures I
  reads alloc, self.foot
  writes alloc, self.foot

I ≙ ∀x,y:Cell ∈ alloc · x ∈ x.foot ∧ (x=y ∨ x.foot # y.foot)
```

Fig. 16. Specifications for methods of Cell, adapted from Figure 1, using syntax close to the formalization but allowing multiple parameters (and making **self** explicit). The name **self** has no special semantics.

To use PURELINK note that the hypothesis context for (17) is empty. But the hypothesis context used for the judgment of get's implementation comprises get's specification; this hypothesis context is also the one for the inner let. The use of IMPURELINK requires that context be augmented by the specifications of set and init to set up the judgments for the implementations of set and init. Let Φ comprise the specifications, given in Figure 16, of the three methods. Then the client's judgment uses hypothesis context Φ. The partial candidate φ for pure method get is defined as $\varphi(get)(\sigma, o) = \sigma(o.value)$ for all states σ and non-null references o in $[\![Cell]\!]\sigma$ such that $\sigma \models I$. (Otherwise, $\varphi(get)(\sigma, o) = \lightning$, in accord with Def. 5.1.) To save space we elide Γ, Φ, φ in the development below, although we will point to the use of φ when necessary.

Proof of the Cell method implementations. We prove the correctness of the method implementations with respect to the specifications of get, set and init given in Figure 16.

- get: With $B \mathrel{\widehat{=}} \mathsf{res} := \mathsf{self}.value$, we must prove

$$B : \mathsf{self} \neq \mathsf{null} \ \wedge\ I \leadsto \mathsf{res} = get(\mathsf{self})\ [\mathsf{wr\,res}, \mathsf{rd\,self}, \mathsf{rd\,self}.foot\text{‘}\mathsf{any}] \tag{18}$$

Using FIELDACC we can obtain $B : \mathsf{self} \neq \mathsf{null} \leadsto \mathsf{res} = \mathsf{self}.value\ [\mathsf{wr\,res}, \mathsf{rd\,self}, \mathsf{rd\,self}.value]$. Note that CONSEQ can be used to strengthen the precondition above to $\mathsf{self} \neq \mathsf{null} \ \wedge\ I$. The rules for subeffect judgments allow us to conclude $\{\mathsf{self}\} \subseteq \mathsf{self}.foot \models \mathsf{rd\,self}.value \leq \mathsf{rd\,self}.foot\text{‘}\mathsf{any}$. Because $\mathsf{self} \neq \mathsf{null} \ \wedge\ I \Rightarrow \{\mathsf{self}\} \subseteq \mathsf{self}.foot$, we get the subeffect judgment $\mathsf{self} \neq \mathsf{null} \ \wedge\ I \models \mathsf{rd\,self}.value \leq \mathsf{rd\,self}.foot\text{‘}\mathsf{any}$. Furthermore the formula $\mathsf{res} = \mathsf{self}.value \Rightarrow get(\mathsf{self}) = v$ is valid by definition of φ. Now CONSEQ yields (18).

- set: With $C \mathrel{\widehat{=}} \mathsf{self}.value := v$, we must prove

$$C : \mathsf{self} \neq \mathsf{null} \ \wedge\ I \leadsto get(\mathsf{self}) = v \ \wedge\ I\ [\mathsf{rw\,self}.foot\text{‘}\mathsf{any}, \mathsf{rd\,self}, \mathsf{rd}\ v] \tag{19}$$

Rule FieldUpd yields $C : \mathsf{self} \neq \mathsf{null} \leadsto \mathsf{self}.value = v$ [wr self.$value$, rd self, rd v]. Aiming to use Frame with I, since I is framed by rd alloc, rd alloc‘$foot$, we compute the separator formula

$$\begin{aligned}\mathsf{rd\ alloc}, \mathsf{rd\ alloc}`foot \cdot\!/.\ \mathsf{wr\ self}.value &= (\mathsf{rd\ alloc} \cdot\!/.\ \mathsf{wr\ self}.value) \\ &\quad \wedge (\mathsf{rd\ alloc}`foot \cdot\!/.\ \mathsf{wr\ self}.value) \\ &= \mathsf{true} \wedge \mathsf{true}\end{aligned}$$

Thus by Frame we obtain $C : \mathsf{self} \neq \mathsf{null} \wedge I \leadsto \mathsf{self}.value = v \wedge I$ [wr self.$value$, rd self, rd v]. The rules for subeffect judgments allow us to conclude $\{\mathsf{self}\} \subseteq \mathsf{self}.foot \models \mathsf{wr\ self}.value \leq \mathsf{wr\ self}.foot`\mathsf{any}, \mathsf{rd\ self}.foot`\mathsf{any}$. Because $\mathsf{self} \neq \mathsf{null} \wedge I \Rightarrow \{\mathsf{self}\} \subseteq \mathsf{self}.foot$, we get the subeffect judgment $\mathsf{self} \neq \mathsf{null} \wedge I \models \mathsf{wr\ self}.value \leq \mathsf{rw\ self}.foot`\mathsf{any}$. Furthermore the formula $\mathsf{self}.value = v \Rightarrow get(\mathsf{self}) = v$ is valid by definition of φ. Thus using Conseq we get (19).

- init: With $D \mathrel{\widehat{=}} \mathsf{self}.foot := \{\mathsf{self}\}$, we must prove

$$D : \begin{array}{l}\mathsf{self}.foot = \varnothing \\ \wedge\, \forall x : Cell \in \mathsf{alloc} \setminus \{\mathsf{self}\} \cdot \mathsf{self} \notin x.foot \\ \wedge\, J(\mathsf{alloc} \setminus \{\mathsf{self}\})\end{array} \leadsto I\ [\mathsf{rw\ alloc}, \mathsf{rw\ self}.foot, \mathsf{rd\ self}] \qquad (20)$$

FieldUpd yields $D : \mathsf{self} \neq \mathsf{null} \leadsto \mathsf{self}.foot = \{\mathsf{self}\}$ [wr self.$foot$, rd self]. Aiming to use Frame with $J(\mathsf{alloc} \setminus \{\mathsf{self}\})$, since the formula is framed by rd alloc, rd (alloc $\setminus$ {self})‘$foot$, rd self, we compute the separator formula

$$\begin{aligned}&\mathsf{rd\ alloc}, \mathsf{rd}\ (\mathsf{alloc} \setminus \{\mathsf{self}\})`foot, \mathsf{rd\ self} \cdot\!/.\ \mathsf{wr\ self}.foot \\ =\ &\quad \mathsf{rd\ alloc} \cdot\!/.\ \mathsf{wr\ self}.foot \\ &\wedge (\mathsf{rd}\ (\mathsf{alloc} \setminus \{\mathsf{self}\})`foot \cdot\!/.\ \mathsf{wr\ self}.foot) \\ &\wedge (\mathsf{rd\ self} \cdot\!/.\ \mathsf{wr\ self}.foot) \\ =\ &\ \mathsf{true} \wedge (\mathsf{alloc} \setminus \{\mathsf{self}\} \# \{\mathsf{self}\}) \wedge \mathsf{true}\end{aligned}$$

which is a conjunction of trues. So, by Frame we get

$$D : \mathsf{self} \neq \mathsf{null} \wedge J(\mathsf{alloc} \setminus \{\mathsf{self}\}) \leadsto \mathsf{self}.foot = \{\mathsf{self}\} \wedge J(\mathsf{alloc} \setminus \{\mathsf{self}\})\ [\mathsf{wr\ self}.foot, \mathsf{rd\ self}]$$

Now $\mathsf{self}.foot = \{\mathsf{self}\} \wedge J(\mathsf{alloc} \setminus \{\mathsf{self}\}) \Rightarrow I$, and we have the subeffect judgment $\mathsf{self} \neq \mathsf{null} \wedge J(\mathsf{alloc} \setminus \{\mathsf{self}\}) \models \mathsf{wr\ self}.foot, \mathsf{rd\ self} \leq \mathsf{rw\ alloc}, \mathsf{rw\ self}.foot, \mathsf{rd\ self}$. By Conseq we get

$$D : \mathsf{self} \neq \mathsf{null} \wedge J(\mathsf{alloc} \setminus \{\mathsf{self}\}) \leadsto I\ [\mathsf{rw\ alloc}, \mathsf{rw\ self}.foot, \mathsf{rd\ self}]$$

Now (20) follows by Conseq: the desired precondition can be obtained by strengthening the precondition $\mathsf{self} \neq \mathsf{null}$ in the above judgment, noting that $\mathsf{self}.foot = \varnothing \Rightarrow \mathsf{self} \neq \mathsf{null}$.

Proof of the Cell client. 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. We rely on the derivable consequences of rule Alloc, which are summarized as rule Alloc1 in Section 7.1. Using that the default value for $foot : \mathsf{rgn}$ is $\varnothing$, we get

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

Aiming to use Frame with $J(r)$, since $J(r)$ is framed by rd r, rd r‘$foot$, we use the definition of $\cdot\!/.$ to compute the separator formula

$$\begin{aligned}(\mathsf{rd}\ r, \mathsf{rd}\ r`foot) \cdot\!/.\ \mathsf{wr}\ d, \mathsf{wr\ alloc} &= (\mathsf{rd}\ r \cdot\!/.\ \mathsf{wr}\ d) \wedge (\mathsf{rd}\ r \cdot\!/.\ \mathsf{wr\ alloc}) \\ &\quad \wedge (\mathsf{rd}\ r`foot \cdot\!/.\ \mathsf{wr}\ d) \wedge (\mathsf{rd}\ r`foot \cdot\!/.\ \mathsf{wr\ alloc}) \\ &= \mathsf{true} \wedge \mathsf{true} \wedge \mathsf{true} \wedge \mathsf{true}\end{aligned}$$

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

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

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

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

Hence by FRAME we get

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

By CONSEQ, using the validity of the formulas

$$\begin{array}{l} I \Rightarrow J(r) \\ \mathsf{alloc} = r \cup \{d\} \Rightarrow d \neq \mathsf{null} \text{ (because null} \notin \text{alloc in all states)} \\ \mathsf{alloc} = r \cup \{d\} \wedge J(r) \Rightarrow J(\mathsf{alloc} \setminus \{d\}) \\ \mathsf{alloc} = r \cup \{d\} \wedge \forall b : Cell \in r \cdot d \notin b.foot \Rightarrow \forall b : Cell \in \mathsf{alloc} \setminus \{d\} \cdot d \notin b.foot \end{array}$$

we get the following, where $\eta_1 \mathrel{\widehat{=}} \mathsf{wr}\, d, \mathsf{rw}\, \mathsf{alloc}$.

$$d := \mathsf{new}\,Cell : \ I \wedge r = \mathsf{alloc} \wedge c \neq \mathsf{null} \ \rightsquigarrow \begin{array}{l} d \notin r \wedge d \neq \mathsf{null} \\ \wedge\, d \neq c \wedge c \neq \mathsf{null} \\ \wedge\, d.foot = \varnothing \\ \wedge\, \forall b : Cell \in \mathsf{alloc} \setminus \{d\} \cdot d \notin b.foot \\ \wedge\, J(\mathsf{alloc} \setminus \{d\}) \end{array} [\eta_1] \tag{21}$$

Now by IMPURECALL

$$\begin{array}{ll} init(d) : & d.foot = \varnothing \wedge \forall x : Cell \in \mathsf{alloc} \setminus \{d\} \cdot d \notin x.foot \wedge J(\mathsf{alloc} \setminus \{d\}) \\ & \rightsquigarrow I\, [\mathsf{rw}\, \mathsf{alloc}, \mathsf{rw}\, d.foot, \mathsf{rd}\, d] \end{array}$$

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

$$init(d) : \begin{array}{l} d.foot = \varnothing \\ \wedge\forall x : Cell \in \mathsf{alloc} \setminus \{d\} \cdot d \notin x.foot \\ \wedge J(\mathsf{alloc} \setminus \{d\}) \wedge d \notin r \\ \wedge d \neq \mathsf{null} \wedge d \neq c \wedge c \neq \mathsf{null} \end{array} \rightsquigarrow \begin{array}{l} I \wedge d \notin r \\ \wedge d \neq \mathsf{null} \\ \wedge d \neq c \wedge c \neq \mathsf{null} \end{array} [\mathsf{rw}\, \mathsf{alloc}, \mathsf{rw}\, d.foot, \mathsf{rd}\, d]$$

Using CONSEQ, we can rewrite this judgment as

$$init(d) : \begin{array}{l} d \notin r \wedge d \neq \mathsf{null} \\ \wedge d \neq c \wedge c \neq \mathsf{null} \\ \wedge d.foot = \varnothing \\ \wedge\forall x : Cell \in \mathsf{alloc} \setminus \{d\} \cdot d \notin x.foot \\ \wedge J(\mathsf{alloc} \setminus \{d\}) \end{array} \rightsquigarrow \begin{array}{l} I \wedge d \notin r \\ \wedge d \neq \mathsf{null} \\ \wedge d \neq c \wedge c \neq \mathsf{null} \end{array} [\mathsf{rw}\, \mathsf{alloc}, \mathsf{rw}\, d.foot, \mathsf{rd}\, d] \tag{22}$$

Next we use SEQ to combine (21) and (22) as follows. Observe that 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 (22) 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 $G`f$ 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

$$d := \mathsf{new}\, Cell; init(d) : I \wedge r = \mathsf{alloc} \wedge c \neq \mathsf{null} \leadsto \begin{array}{l} I \wedge d \notin r \wedge d \neq \mathsf{null} \\ \wedge\, d \neq c \wedge c \neq \mathsf{null} \end{array} [\mathsf{rw}\, d, \mathsf{rw}\, \mathsf{alloc}] \tag{23}$$

By ImpureCall for set, we get

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

Let $\eta_3 \mathrel{\widehat{=}} \mathsf{rw}\, d, \mathsf{rw}\, \mathsf{alloc}$ and let $\eta_4 \mathrel{\widehat{=}} \mathsf{rw}\, c.foot`\mathsf{any}, \mathsf{rd}\, c.foot, \mathsf{rd}\, c$. We have $I \wedge c \neq \mathsf{null} \models \mathsf{rw}\, c.foot`\mathsf{any}, \mathsf{rd}\, c \leq \eta_4$. Aiming to frame $d \notin r \wedge d \neq \mathsf{null} \wedge d \neq c \wedge c \neq \mathsf{null}$, we compute $(\mathsf{rd}\, c, \mathsf{rd}\, d, \mathsf{rd}\, r) \cdot\!/\!. \ \eta_4$; this is a conjunction of trues. Thus by Frame and Conseq, we get

$$set(c,5) : \begin{array}{l} I \wedge c \neq \mathsf{null} \wedge d \notin r \wedge d \neq \mathsf{null} \\ \wedge\, d \neq c \end{array} \leadsto \begin{array}{l} I \wedge get(c) = 5 \wedge d \notin r \wedge d \neq \mathsf{null} \\ \wedge\, d \neq c \end{array} [\eta_3] \tag{24}$$

Again we aim to use Seq to compose (23) and (24). Notice that both η_3 and η_4 have framed reads and $\mathsf{wr}\, r \notin \eta_3$. Let $H_2 \mathrel{\widehat{=}} \varnothing$, $P_2 \mathrel{\widehat{=}} I \wedge d \notin r \wedge d \neq \mathsf{null} \wedge d \neq c \wedge c \neq \mathsf{null}$, 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_4 is immune from η_3 under P. The region expressions in η_4 are $c.foot$ and $\{c\}$. Thus we need to show the validity of $P \Rightarrow ftpt(\{c\}`foot, \Phi) \cdot\!/\!. \ \eta_3$ and $P \Rightarrow ftpt(\{c\}, \Phi) \cdot\!/\!. \ \eta_3$. It suffices to show the validity of the first implication. The validity of the second follows because $ftpt(\{c\}`foot, \Phi) = \mathsf{rd}\,\{c\}`foot, ftpt(\{c\}, \Phi) = \mathsf{rd}\,\{c\}`foot, \mathsf{rd}\, c$. The validity of the first implication follows because its consequent reduces to a conjunction of trues. To wit, by definition of separator, we have

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

Observe that c, d are distinct variables in Γ, which is why $\mathsf{rd}\, c \cdot\!/\!. \ \mathsf{wr}\, d = \mathsf{true}$. Thus by Seq we get

$$d := \mathsf{new}\, Cell; init(d); set(c,5) : I \wedge r = \mathsf{alloc} \wedge c \neq \mathsf{null} \leadsto \begin{array}{l} I \wedge d \notin r \wedge d \neq \mathsf{null} \\ \wedge\, d \neq c \wedge get(c) = 5 \end{array} [\eta_3, \eta_4] \tag{25}$$

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

By ImpureCall for set, we get

$$set(d,4) : I \wedge d \neq \mathsf{null} \leadsto I \wedge get(d) = 4 [\eta_5] \quad \text{where} \quad \eta_5 \mathrel{\widehat{=}} \mathsf{rw}\, d.foot`\mathsf{any}, \mathsf{rd}\, d$$

Aiming to frame $d \notin r \wedge d \neq c \wedge get(c) = 5$, note that $(\mathsf{rd}\, c, \mathsf{rd}\, d, \mathsf{rd}\, r, \mathsf{rd}\, c.foot, \mathsf{rd}\, c.foot`\mathsf{any}) \cdot\!/\!. \eta_5$ we can use Frame to get

$$set(d,4) : \begin{array}{l} I \wedge d \notin r \wedge d \neq \mathsf{null} \\ \wedge\, d \neq c \wedge get(c) = 5 \end{array} \leadsto \begin{array}{l} I \wedge d \notin r \wedge get(d) = 4 \\ \wedge\, d \neq c \wedge get(c) = 5 \end{array} [\eta_5] \tag{26}$$

Now we check the side conditions of Seq to compose (25) and (26). Let $H_3 = d.foot$ and $P_3 \mathrel{\widehat{=}} I \wedge d \notin r \wedge d \neq \mathsf{null} \wedge d \neq c \wedge get(c) = 5$. Then $P_3 \Rightarrow H_3 \# r$ is valid. Also, η has framed reads and $\mathsf{wr}\, r \notin \eta$. Since $\mathsf{rd}\, d$ is P/η-immune, by Seq we get

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

Notice that the footprint of H_3 contains $\mathsf{rd}\,\{d\}`foot$ and it does not make sense for this to appear in the effect of the sequence, because it refers to a field $d.foot$ of the freshly allocated object assigned to d. This shows why rule Seq does not require the entire frame condition to be read framed. By contrast, the H_1 earlier has its footprint in the effect.

Now, from Conseq, using $I \land d \notin r \land get(d) = 4 \land d \neq c \land get(c) = 5 \Rightarrow I \land get(c) = 5$, we get

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

The variable r only serves to refer to the initial value of alloc. By rule ExistRegion the above judgment yields

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

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

$$d := \mathsf{new}\,Cell; init(d); set(c, 5); set(d, 4) : I \land c \neq \mathsf{null} \rightsquigarrow I \land get(c) = 5[\eta] \qquad (27)$$

To finish the example we need to link the client to the implementations. Using judgments (18), (19), (20), and (27), rules PureLink and ImpureLink yield that (17) has the specification $I \land c \neq \mathsf{null} \rightsquigarrow I \land get(c) = 5[\eta]$.

8 SOUNDNESS OF THE PROOF SYSTEM

8.1 Proof of soundness for rules other than linking

We prove soundness of each rule in turn, making reference to the named conditions (Safety, Post, Write, and Read) in Definition 5.2. The proof of While is in the Appendix, and soundness of linking is the topic of Sections 8.2 and 8.3.

FieldAcc and Assign: These are straightforward an similar to FieldUpd which we prove in detail. For Assign, the argument for the Read condition uses Lemma 6.5.

FieldUpd: Consider any Φ-interpretation φ, and any state σ satisfying the precondition, that is, such that $\sigma \models_\varphi x \neq \mathsf{null}$. By semantics the configuration $\langle x.f := y,\ \sigma,\ \_\rangle$ does not fault (which proves Safety), and we have $\langle x.f := y,\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto} \langle \mathsf{skip},\ \tau,\ \_\rangle$ where $\tau \mathrel{\widehat{=}} [\sigma \mid x.f\colon \sigma(y)]$. To prove 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 $\tau(x.f) = \tau(y)$.

To prove Write, let $\varepsilon \mathrel{\widehat{=}} \mathsf{wr}\, x.f, \mathsf{rd}\, x, \mathsf{rd}\, y$. Notice that $wlocs(\sigma, \varphi, \varepsilon) = \{\sigma(x).f\}$. Since τ is only different from σ in value of $x.f$, we get Write in accord with Definition 4.1.

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

Alloc: Consider any Φ-interpretation φ and any state σ with $\sigma \models_\varphi r = \mathsf{alloc}$. By semantics it is not the case that $\langle x := \mathsf{new}\ K,\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto} \lightning$. Instead we have $\langle x := \mathsf{new}\ K,\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto} \langle \mathsf{skip},\ [\sigma_1 \mid x\colon o],\ \_\rangle$, where $o \notin \sigma(\mathsf{alloc})$, $\sigma_1 = New(\sigma, o, K, default(\overline{T}))$, and $Fields(K) = \overline{f} : \overline{T}$. Let $\tau = [\sigma_1 \mid x\colon o]$. (Recall that $New(\ldots)$ is defined in the caption of Figure 9.) We have that τ satisfies the postconditions $x \notin r$, $\mathsf{alloc} = r \cup \{x\}$, and $x.\overline{f} = default(\overline{T})$, by definitions.

To prove Write, that is, $\sigma{\to}\tau \models \mathsf{wr}\, x, \mathsf{rw}\, \mathsf{alloc}$, observe that $wlocs(\sigma, \varphi, (\mathsf{wr}\, x, \mathsf{rw}\, \mathsf{alloc})) = \{x, \mathsf{alloc}\}$ and $written(\sigma, \tau) = \{x, \mathsf{alloc}\}$ by definitions. To prove Read, consider additional states σ', τ' and refperm π such that $Agree(\sigma, \sigma', (\mathsf{wr}\, x, \mathsf{rw}\, \mathsf{alloc}), \pi, \varphi)$ and $\langle x := \mathsf{new}\ K,\ \sigma',\ \_\rangle \overset{\varphi}{\longmapsto}$

$\langle \mathsf{skip},\ \tau',\ \_\rangle$. 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 $\rho = \pi \cup \{(\tau(x),\tau'(x))\}$. We must show

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

Note that $\mathit{freshLocs}(\sigma,\tau) = \{o.f_i \mid f_i \in \overline{f}\}$ and $\mathit{freshLocs}(\sigma',\tau') = \{o'.f_i \mid f_i \in \overline{f}\}$, where $o = \tau(x)$ and $o' = \tau'(x)$. So, we have $\rho(o) = o'$. From this we get $\rho(\mathit{freshLocs}(\sigma,\tau)) \subseteq \mathit{freshLocs}(\sigma',\tau')$. Now we show $\mathit{Lagree}(\tau,\tau',\rho,\mathit{written}(\sigma,\tau)\cup\mathit{freshLocs}(\sigma,\tau'))$. By definition of ρ we have $\tau(x) \overset{\rho}{\sim} \tau'(x)$. We also have $\tau(\mathsf{alloc}) \overset{\rho}{\sim} \tau'(\mathsf{alloc})$ because $\tau(\mathsf{alloc}) = \tau(r)\cup\{\tau(x)\} = \sigma(r)\cup\{\tau(x)\} \overset{\rho}{\sim} \sigma'(r)\cup\{\tau'(x)\} = \tau'(\mathsf{alloc})$. Recall $\mathit{freshLocs}(\sigma,\tau) = \{o.f_i \mid f_i \in \overline{f}\}$, and the fields are initialized to default values, so we have $\tau(o.f_i) \overset{\rho}{\sim} \tau'(\rho(o).f_i)$ for each f_i.

<u>ImpureCall</u>: 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 $\sigma{\rightarrow}\tau \models_\varphi \varepsilon^x_v$, which gives us $\tau \models_\varphi Q^x_z$ and $\sigma{\rightarrow}\tau \models_\varphi \varepsilon^x_z$ since $\sigma(z) = v$.

Finally, to prove Read, for any τ,σ',τ',π 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 $\rho(\mathit{freshLocs}(\sigma,\tau)) \subseteq \mathit{freshLocs}(\sigma',\tau')$ and $\mathit{Lagree}(\tau,\tau',\rho,\mathit{freshLocs}(\sigma,\tau)\cup\mathit{written}(\sigma,\tau))$.

<u>PureCall</u>: 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 $\tau = [\sigma \mid y{:}\ w]$. Thus Safety is immediate. Furthermore, $\sigma \models_\varphi Q^{x,\mathsf{res}}_{z,w}$ by Definition 5.1(b) of context interpretation, hence $\sigma \models_\varphi Q^{x,\mathsf{res}}_{z,y}$. We get the postcondition $y = m(z)$ by semantics: According to the typing rule for $y := m(z)$, y is not in scope for the specification of m (see also Footnote 7); thus $[\![m(z)]\!]_\varphi\tau = [\![m(z)]\!]_\varphi\sigma$ and hence $\tau \models_\varphi y = m(z)$.

For Write, it is immediate from semantics: $\tau = [\sigma \mid y{:}\ w]$ 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

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

To show Read, for any τ,σ',τ',π, 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 $\tau = [\sigma \mid y{:}\ w]$ and $\tau' = [\sigma' \mid y{:}\ w']$. 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.

<u>Seq</u>: We only show Read, 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

$$\sigma \models_\varphi P \land r = \mathsf{alloc}, \qquad \sigma' \models_\varphi P \land r = \mathsf{alloc}, \qquad \mathit{Agree}(\sigma, \sigma', (\varepsilon_1, \varepsilon_2), \pi, \varphi), \tag{28}$$

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

$$\rho(\mathit{freshLocs}(\sigma, \tau)) \subseteq \mathit{freshLocs}(\sigma', \tau') \tag{29}$$

$$\mathit{Lagree}(\tau, \tau', \rho, \mathit{written}(\sigma, \tau) \cup \mathit{freshLocs}(\sigma, \tau)) \tag{30}$$

To show the agreement (29), observe 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 (28) we have $\mathit{Agree}(\sigma, \sigma', \varepsilon_1, \pi, \varphi)$, so using the Read property of the first premise we get some refperm $\rho_1 \supseteq \pi$ such that

$$\begin{array}{l} \rho_1(\mathit{freshLocs}(\sigma, \sigma_1)) \subseteq \mathit{freshLocs}(\sigma', \sigma_1') \text{ and } \mathit{Lagree}(\sigma_1, \sigma_1', \rho_1, Y) \\ \text{where } Y = \mathit{written}(\sigma, \sigma_1) \cup \mathit{freshLocs}(\sigma, \sigma_1). \end{array} \tag{31}$$

From the first premise, we also know that $\sigma{\rightarrow}\sigma_1 \models_\varphi \varepsilon_1$. 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 (28) using Definition 4.2, we can derive $\mathit{Lagree}(\sigma, \sigma', \pi, \mathit{rlocs}(\sigma_1, \varphi, \varepsilon_2))$.

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 Lemmas 6.11 and 6.12, we obtain $\rho_1 \subseteq \pi$ such that

$$\mathit{Lagree}(\sigma_1, \sigma_1', \rho_1, \mathit{rlocs}(\sigma_1, \varphi, \varepsilon_2)). \tag{32}$$

From side conditions $\Phi; \psi \models P_1 \Rightarrow H \# r$ and $\mathsf{wr}\ r \notin \varepsilon_1$ of SEQ, using also $\sigma \models_\varphi r = \mathsf{alloc}$, we have $[\![H]\!]_\varphi \sigma_1 \subseteq \mathit{freshRefs}(\sigma, \sigma_1)$. Thus

$$\mathit{rlocs}(\sigma_1, \varphi, \mathsf{rd}\ H\text{`}\overline{f}) \subseteq \mathit{freshLocs}(\sigma, \sigma_1) \subseteq Y \tag{33}$$

With a similar argument we get

$$\mathit{rlocs}(\sigma_1', \varphi, \mathsf{rd}\ H\text{`}\overline{f}) \subseteq \mathit{freshLocs}(\sigma', \sigma_1') \tag{34}$$

From (31) and (33), we get $\mathit{Lagree}(\sigma_1, \sigma_1', \rho_1, \mathit{rlocs}(\sigma_1, \varphi, \mathsf{rd}\ H\text{`}\overline{f}))$. Combined with (32) and from Definition 4.2, we derive $\mathit{Lagree}(\sigma_1, \sigma_1', \rho_1, \mathit{rlocs}(\sigma_1, \varphi, (\varepsilon_2, \mathsf{rd}\ H\text{`}\overline{f})))$. By Definition 4.3 this yields

$$\mathit{Agree}(\sigma_1, \sigma_1', (\varepsilon_2, \mathsf{wr}\ H\text{`}\overline{f}, \mathsf{rd}\ H\text{`}\overline{f}), \rho_1, \varphi) \tag{35}$$

Recall that from the validity of the first premise we have $\sigma', \sigma {\Rightarrow} \sigma_1', \sigma_1 \models_\varphi \varepsilon_1$. Since ε_1 has frame reads, using Lemma 6.11 and (28), we have $\mathit{Agree}(\sigma', \sigma, \varepsilon_1, \pi^{-1}, \varphi)$. From Lemma 6.13, we get $\mathit{Lagree}(\sigma_1', \sigma_1, \rho_1^{-1}, \mathit{freshLocs}(\sigma', \sigma_1'))$. From (34), we get $\mathit{Lagree}(\sigma_1', \sigma_1, \rho_1^{-1}, \mathit{rlocs}(\sigma_1', \varphi, \mathsf{rd}\ H\text{`}\overline{f}))$. Furthermore ε_2 has framed reads. So, using Lemma 6.11 and (35), we get $\mathit{Lagree}(\sigma_1', \sigma_1, \rho_1^{-1}, \mathit{rlocs}(\sigma_1', \varphi, \varepsilon_2))$. Thus we have $\mathit{Lagree}(\sigma_1', \sigma_1, \rho_1^{-1}, \mathit{rlocs}(\sigma_1', \varphi, (\varepsilon_2, \mathsf{rd}\ H\text{`}\overline{f})))$. So we get

$$\mathit{Agree}(\sigma_1', \sigma_1, (\varepsilon_2, \mathsf{wr}\ H\text{`}\overline{f}, \mathsf{rd}\ H\text{`}\overline{f}), \rho_1^{-1}, \varphi) \tag{36}$$

Using (35) we appeal to the Read property of the second premise, that is, $\sigma_1, \sigma_1' \Rightarrow \tau, \tau' \models_\varphi \varepsilon_2, \mathsf{rd}\ H`\overline{f}$ which yields that there is a refperm $\rho \supseteq \rho_1$ such that

$$\begin{array}{c} \rho(\mathit{freshLocs}(\sigma_1, \tau)) \subseteq \mathit{freshLocs}(\sigma_1', \tau) \\ \mathit{Lagree}(\tau, \tau', \rho, W), \text{ where } W = \mathit{written}(\sigma_1, \tau) \cup \mathit{freshLocs}(\sigma_1, \tau). \end{array} \tag{37}$$

Observe that

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

From (37), 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 $\sigma_1, \sigma_1' \Rightarrow \tau, \tau' \models_\varphi \varepsilon_2, \mathsf{wr}\ H`\overline{f}, \mathsf{rd}\ H`\overline{f}$ and $\sigma_1', \sigma_1 \Rightarrow \tau', \tau \models_\varphi \varepsilon_2, \mathsf{wr}\ H`\overline{f}, \mathsf{rd}\ H`\overline{f}$. Using Lemma 6.12, from (35), (36) and (31), we get $\mathit{Lagree}(\tau, \tau', \rho, Y)$. This completes the proof of (30).

To complete the proof of (29), note that $\mathit{freshLocs}(\sigma, \tau) = \mathit{freshLocs}(\sigma, \sigma_1) \cup \mathit{freshLocs}(\sigma_1, \tau)$, since $\sigma(\mathsf{alloc}) \subseteq \sigma_1(\mathsf{alloc}) \subseteq \tau(\mathsf{alloc})$. Thus we have

$$\begin{array}{rcll} \rho(\mathit{freshLocs}(\sigma, \tau)) & = & \rho(\mathit{freshLocs}(\sigma, \sigma_1)) \cup \rho(\mathit{freshLocs}(\sigma_1, \tau)) & \\ & \subseteq & \mathit{freshLocs}(\sigma', \sigma_1') \cup \mathit{freshLocs}(\sigma_1', \tau') & \text{from (31) and (37)} \\ & = & \mathit{freshLocs}(\sigma', \tau') & \end{array}$$

<u>Frame</u>: We must show $\Phi; \psi \models C : P \land R \rightsquigarrow Q \land R\,[\varepsilon]$, assuming validity of premises. Consider any Φ-interpretation φ such that $\psi \subseteq \varphi$. Suppose $\sigma \models_\varphi P \land R$. 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 $\Phi; \psi \models P \land R \Rightarrow \eta \cdot/. \varepsilon$ yields $\sigma \models_\varphi \eta \cdot/. \varepsilon$. Instantiating the premise for C with φ gives $\sigma \rightarrow \tau \models_\varphi \varepsilon$ (Write), so by Lemma 6.6 we have $\mathit{Agree}(\sigma, \tau, \eta, \mathit{id}, \varphi)$ 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 $\mathit{Agree}(\sigma, \tau, \eta, \mathit{id}, \varphi)$ and $\sigma \models_\varphi P \land R$ we obtain $\tau \models_\varphi R$.

<u>Conseq</u>: For all Φ-interpretation φ that extends ψ and all state σ such that $\sigma \models_\varphi P_1$, from $\Phi; \psi \models P_1 \Rightarrow P$, we have $\sigma \models_\varphi P$. Thus by validity of the premise, we conclude that transition from $\langle C,\ \sigma,\ \_\rangle$ via $\xmapsto{\varphi}$ cannot fault. To prove post and safety, consider state τ with $\langle C,\ \sigma,\ \_\rangle \xmapsto{\varphi}{}^* \langle \mathsf{skip},\ \tau,\ \_\rangle$. 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 $P_1\ \Phi; \psi \models \varepsilon \leq \varepsilon_1$, by Lemma 6.2 (allowed change) we have $\sigma \rightarrow \tau \models_\varphi \varepsilon_1$. The read condition is the result of use of Lemma 6.2 (allowed dependency) on Read condition of the premise.

<u>InterpIntro</u>: 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 interpretation that extends $\psi \cup \psi'$ also extends ψ, so the conclusion follows directly from the premise by semantics of correctness judgment.

8.2 Preliminaries for soundness of linking rules

To lay groundwork, we give a high level sketch the soundness arguments for the linking rules, which motivates some definitions and technical results adapted from RLII.

The conclusion of both linking rules is a judgment about the potentially recursive let-binding of method name m to method body B in client C. The let-command executes by taking one step in which B gets bound to m in the environment, followed by execution of C in that environment.

The two premises of the rule are judgments for B and for C, both with m bound to its specification. To prove that the let-command's execution satisfies its specification, we show that any trace of the let-command, in which calls to m are environment calls, gives rise to a trace of C in which calls to m are context calls. This lets us appeal to the premise judgment for C, since its specification is the same as that of the let-command.

To make this argument precise, we need to decompose traces with m in the environment into segments corresponding to 'topmost calls' of m, between which the code of C is executed. Moreover, we need to connect executions of B, in which recursive calls of m are environment calls, with executions of B in which recursive calls are context calls, so that we can appeal to the premise judgment for B. These arguments go by induction on the number of topmost calls to m in a trace, and by induction on recursion depth, for which the rest of this subsection provides notation and nomenclature.

Technical background. 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 (Definition 4.3 there), 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 8.1 (independence). Suppose Φ is swf in Γ and φ is a Φ-interpretation. Consider any $m, C, \sigma, \dot{\mu}, \varphi$ such that $m \in \mathit{dom}(\dot{\mu})$ 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 $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto} ↯$ iff $\langle C,\ \sigma,\ \mu\rangle \overset{\varphi\cup\theta}{\longmapsto} ↯$.

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

Definition 8.2 (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.

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.

Lemma 8.3 (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 $\upsilon_i = [\tau_i + x_i, \mathsf{res}_i\colon \tau_i(z_i), \mathit{default}(U)]$ (note that x_i and res_i are fresh parameter names.)

(3) $\langle B^{x,\mathsf{res}}_{x_i,\mathsf{res}_i},\ v_i,\ \mu_i\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \dot\sigma_i,\ \mu_i\rangle$ and hence by semantics $\langle B^{x,\mathsf{res}}_{x_i,\mathsf{res}_i}; y := \mathsf{res}_i; \mathsf{ecall}(x_i,\mathsf{res}_i); C_i,\ v_i,\ \mu_i\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{ecall}(x_i,\mathsf{res}_i); C_i,\ \ddot\sigma_i,\ \mu_i\rangle$, 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 $\sigma_i = \ddot\sigma_i \upharpoonright x_i \upharpoonright \mathsf{res}_i$

(5) $\langle C_n,\sigma_n,\mu_n\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle D,\tau,v\rangle$ without any completed invocations of m—but allowing a topmost call that is incomplete.

Lemma 8.4 (decomposition for pure interpreted methods). Suppose that μ is a 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 $\langle C_0,\ \sigma_0,\ \mu_0\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle D,\ \tau,\ v\rangle$. 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) $\langle C_n,\sigma_n,\mu_n\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle D,\tau,v\rangle$ without any completed invocations of m—but allowing a topmost call that is incomplete.

Lemma 8.5 (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.

A ***bounded configuration*** has the form $\langle C,\ \sigma,\ \mu\rangle^k$ where k is a natural number. One can think of k as the size of the available stack space. A computation will get stuck (and does not fault) if it attempts to make an environment call on a method when k is 0.

Definition 8.6 (depth-bounded semantics). The transition relation on depth-bounded configurations is written $\overset{\varphi}{\longmapsto}$ just like for standard configurations. It is defined so that the bound is decreased in the invocation step and increased when the end-marker of the method body is reached:

$$\frac{k > 0 \qquad \mu(m) = (x{:}T, \mathsf{res}{:}U.C) \qquad x' \notin \mathit{Vars}(\sigma) \qquad \mathsf{res}' \notin \mathit{Vars}(\sigma) \qquad C' = C^{x,\mathsf{res}}_{x',\mathsf{res}'}}{\langle y := m(z),\ \sigma,\ \mu\rangle^k \overset{\theta}{\longmapsto} \langle C'; y := \mathsf{res}'; \mathsf{ecall}(x',\mathsf{res}'),\ [[\sigma + x' {:} \sigma(z)] + \mathsf{res}' {:} \mathit{default}(U)],\ \mu\rangle^{k-1}}$$

$$\langle \mathsf{ecall}(x),\ \sigma,\ \mu\rangle^k \overset{\theta}{\longmapsto} \langle \mathsf{skip},\ \sigma \upharpoonright x,\ \mu\rangle^{k+1}$$

The bound needs to be propagated in one of the transitions for sequence:

$$\frac{\langle C,\ \sigma,\ \mu\rangle^k \overset{\theta}{\longmapsto} \langle C',\ \sigma',\ \mu'\rangle^{k'}}{\langle C\,; D,\ \sigma,\ \mu\rangle^k \overset{\theta}{\longmapsto} \langle C'\,; D,\ \sigma',\ \mu'\rangle^{k'}}$$

In all other cases, the transition rule is the same as for non depth-bounded configurations except that a single bound k is added uniformly to every configuration in the rule.

Lemma 8.7 (depth-bounded semantics). For any θ and $\langle C,\ \sigma,\ \mu\rangle$ we have

(1) $\langle C,\ \sigma,\ \mu\rangle \overset{\theta}{\longmapsto}{}^{*} \langle C',\ \sigma',\ \mu'\rangle$ iff there are k, j such that $\langle C,\ \sigma,\ \mu\rangle^k \overset{\theta}{\longmapsto}{}^{*} \langle C',\ \sigma',\ \mu'\rangle^j$

(2) $\langle C,\ \sigma,\ \mu\rangle \overset{\theta}{\longmapsto}{}^{*} \lightning$ iff there is some $k \geq 0$ such that $\langle C,\ \sigma,\ \mu\rangle^k \overset{\theta}{\longmapsto}{}^{*} \lightning$.

Lemma 8.8 (judgment renaming). If $\Phi;\varphi \models^{\Gamma, x:T} C : P \leadsto P'\ [\varepsilon]$ and $\Gamma, y : T$ is well formed, then $\Phi;\varphi \models^{\Gamma, y:T} C^x_y : P^x_y \leadsto P'^x_y\ [\varepsilon^x_y]$.

8.3 Soundness of linking rules

Soundness of ImpureLink is proved in the appendix. In addition to features common to the soundness proofs for the other linking rules, ImpureLink relies on a theory of quasi-determinacy which is developed in the appendix. Soundness of TranspPureLink can be proved by an argument very much like the one for PureLink, and we omit it. The rest of this section proves PureLink in detail. For this, we rely on nomenclature set out in the results of Section 8.2.

Suppose that Φ and Φ, Θ are well formed in Γ. Suppose Θ is $m : (x{:}T, \mathsf{res}{:}U)R \rightsquigarrow S\ [\eta]$ and $\mathit{dom}(\theta) = \mathit{dom}(\Theta)$. Suppose the side condition on correctness of the interpretation holds:

$$\theta \models \Phi, \Theta; \psi \tag{38}$$

Suppose the premises are valid, that is

$$\Phi, \Theta; \psi \models^{\Gamma} C : P \rightsquigarrow Q\ [\varepsilon] \tag{39}$$

$$\Phi, \Theta; \psi, \theta \models^{\Gamma, x:T, \mathsf{res}:U} B : R \rightsquigarrow \mathsf{res} = m(x)\ [\mathsf{wr}\ \mathsf{res}, \mathsf{rd}\ x, \eta] \tag{40}$$

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

$$\Phi; \psi \models^{\Gamma} \mathsf{let}\ m(x{:}T){:}U = B\ \mathsf{in}\ C : P \rightsquigarrow Q\ [\varepsilon]. \tag{41}$$

Whereas premise (40) is a judgment about B with any recursive calls of m in B treated as context calls, the following Lemma spells out the correctness property of the body B when executed with recursive calls to m as environment calls, by connecting it with the interpretation $\theta(m)$.

Lemma 8.9 (recursive correctness). The following is a consequence of (38) and (40). Let x', res' not be in $\mathit{dom}(\Gamma) \cup \{x, \mathsf{res}\}$. Let Γ' be $\Gamma, x' : T, \mathsf{res}' : U$. Let σ be any Γ'-state such that $\sigma \models_{\psi} R^{x}_{x'}$. Let $\dot{\mu}$ be any Γ'-environment such that $\dot{\mu}(m) = (x : T, \mathsf{res} : U.B)$. Let φ be a Φ-interpretation such that $\psi \subseteq \varphi$. Then the computation from $\langle B^{x,\mathsf{res}}_{x',\mathsf{res}'},\ \sigma,\ \dot{\mu}\rangle$ via $\overset{\varphi\cup\theta}{\longmapsto}$

(a) does not fault, and
(b) if it reaches $\langle \mathsf{skip},\ \tau,\ \dot{\mu}\rangle$, then $\tau = [\sigma \mid \mathsf{res}'{:}\,\theta(m)(\sigma, \sigma(x'))]$.

We defer the proof of this Lemma and proceed to prove (41).

To that end, let μ be the empty 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

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

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 (41), 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 τ, σ', τ', if $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \tau,\ \dot{\mu}\rangle$ and $\langle C,\ \sigma',\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \tau',\ \dot{\mu}\rangle$ and $\sigma' \models^{\Gamma}_{\varphi} P$ then $(\sigma, \sigma') {\Rightarrow} (\tau', \tau) \models_{\varphi} \varepsilon$.

As sketched at the beginning of Section 8.2, we will connect the traces in (i)–(iii), in which m is in the environment, with traces in which m is in the context and thus has an interpretation. Indeed, its interpretation is given: $\theta(m)$.

To use the premises, we need a $\Phi, \Theta; \psi$-interpretation. The side condition $\theta \models \Phi, \Theta; \psi$ and the assumption that φ is a Φ-interpretation directly imply 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 $\mu' = \dot{\mu}' \upharpoonright m$.
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.

Before proving Claim A we use it to prove (i)–(iii).

(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 *Active*(C') is a field access/update. Thus by Lemma 8.1 we get $\langle C', \sigma', \mu'\rangle \overset{\varphi\cup\theta}{\longmapsto} \lightning$. But this contradicts the premise (39) 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 Lemma 8.3 it can be decomposed as

$$\begin{array}{l}
\langle C,\ \sigma,\ \dot{\mu}\rangle \\
\overset{\varphi}{\longmapsto}{}^{*} \langle y := m(z); D,\ \tau,\ \dot{\nu}\rangle \\
\overset{\varphi}{\longmapsto} \langle B^{x,\mathsf{res}}_{x',\mathsf{res}'}; y := \mathsf{res}'; \mathsf{ecall}(x', \mathsf{res}'); D,\ \upsilon,\ \dot{\nu}\rangle \\
\qquad \text{where } x', \mathit{res}' \text{ are fresh variables and } \upsilon \text{ is } [\tau + x', \mathit{res}' \colon \tau(z), \mathit{default}(U)] \\
\overset{\varphi}{\longmapsto}{}^{*} \langle A; \mathsf{ecall}(x', \mathsf{res}'); D,\ \sigma',\ \dot{\mu}'\rangle \qquad \text{where } C' \text{ is } A; \mathsf{ecall}(x', \mathit{res}'); D \text{ for some } A, D \\
\overset{\varphi}{\longmapsto} \lightning
\end{array}$$

So we have $\langle B^{x,res}_{x',res'},\ \upsilon,\ \dot{\nu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \lightning$. On the other hand, $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle y := m(z); D,\ \tau,\ \dot{\nu}\rangle$ is an m-truncated trace. So by Claim A, we have $\tau \models_{\varphi\cup\theta} R^x_z$ and thus $\upsilon \models_\varphi R^x_{x'}$ (using (6)), whence by Lemma 8.9 $\langle B^x_{x'},\ \upsilon,\ \dot{\nu}\rangle$ does not fault—a contradiction.[17] 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 $\langle C,\ \sigma,\ \mu\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \tau,\ \mu\rangle$. We assumed at the outset that $\sigma \models_\varphi P$, so $\sigma \models_{\varphi\cup\theta} P$ because P cannot mention m (*cf.* (6)). Hence by premise (39) for C we have $\tau \models_{\varphi\cup\theta} Q$ and $\sigma{\to}\tau \models_{\varphi\cup\theta} \varepsilon$. Owing to well-formedness of the conclusion (41), all of P, Q, ε are well-formed in Φ. Thus $\models_{\varphi\cup\theta}$ is $\models_\varphi$ and $\mathit{wlocs}(\sigma, \varphi, \varepsilon) = \mathit{wlocs}(\sigma, \varphi\cup\theta, \varepsilon)$ (again using (6) and the analogous property for *wlocs*). Hence we have $\tau \models_\varphi Q$ and $\sigma{\to}\tau \models_\varphi \varepsilon$

(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 $\langle C,\ \sigma,\ \mu\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \tau,\ \mu\rangle$ and $\langle C,\ \sigma',\ \mu\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \tau',\ \mu\rangle$. 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 part of premise (39) for C, there is refperm ρ such that $\rho \supseteq \pi$, $\rho(\mathit{freshLocs}(\tau, \tau')) \subseteq \mathit{freshLocs}(\tau', \tau)$ and $\mathit{Lagree}(\tau, \tau', \rho, \mathit{written}(\sigma, \tau) \cup \mathit{freshLocs}(\sigma, \tau))$.

It remains to prove Claim A and Lemma 8.9.

Proof of Claim A. 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

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

[17]Strictly speaking, Lemma 8.9 pertains to executions of B in Γ'-states whereas υ may have additional variables; but these have no influence on execution of B and can be deleted to make an exact connection with the Lemma.

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 there is a trace $\langle C,\ \sigma,\ \mu\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^* \langle C_0,\ \sigma_0,\ \mu_0\rangle$, then there is a trace

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

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 $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}{}^*$ $\langle C',\ \sigma',\ \dot{\mu}'\rangle$, 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 (38)), we would have $\langle C',\ \sigma',\ \mu'\rangle \overset{\varphi\cup\theta}{\longmapsto} ↯$ and hence $\langle C,\ \sigma,\ \mu\rangle \overset{\varphi\cup\theta}{\longmapsto} ↯$. But this contradicts the premise (39) for C, since $\sigma \models_{\varphi\cup\theta} P$. So Claim A is proved.

It remains to prove Claim B, for which we rely on premise (40) via Lemma 8.9. To build the needed trace via $\overset{\varphi\cup\theta}{\longmapsto}$, we go by induction on the number n of completed topmost calls of m in the trace via $\overset{\varphi}{\longmapsto}$. Accordingly, consider an m-truncated trace

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

Using Lemma 8.3, we obtain intermediate states $\tau_i, \upsilon, \dot{\sigma}, \ddot{\sigma}, \sigma_i$ and environments $\dot{\mu}_i$ such that the following holds.[18]

$$\begin{array}{ll}
\langle C_0,\ \sigma_0,\ \dot{\mu}_0\rangle & \\
\overset{\varphi}{\longmapsto}{}^* \langle y_1 := m(z_1);\ C_1,\ \tau_1,\ \dot{\mu}_1\rangle & \text{with no invocations of } m \\
\overset{\varphi}{\longmapsto} \langle B^{x,\mathsf{res}}_{x_1,\mathsf{res}_1};\ y_1 := \mathsf{res}_1; \mathsf{ecall}(x_1, \mathsf{res}_1);\ C_1,\ \upsilon_1,\ \dot{\mu}_1\rangle & \text{where } \upsilon_1 = [\tau_1 + x_1, \mathsf{res}_1{:}\ \tau_1(z_1), d] \\
 & \text{and } x_1, \mathsf{res}_1 \text{ are fresh and } d = \mathit{default}(U) \\
\overset{\varphi}{\longmapsto}{}^* \langle y_1 := \mathsf{res}_1; \mathsf{ecall}(x_1, \mathsf{res}_1);\ C_1,\ \dot{\sigma}_1,\ \dot{\mu}_1\rangle & \text{where } \langle B^{x,\mathsf{res}}_{x_1,\mathsf{res}_1},\ \upsilon_1,\ \dot{\mu}_1\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip},\ \dot{\sigma}_1,\ \dot{\mu}_1\rangle \\
\overset{\varphi}{\longmapsto} \langle \mathsf{ecall}(x_1, \mathsf{res}_1);\ C_1,\ \ddot{\sigma}_1,\ \dot{\mu}_1\rangle & \text{where } \ddot{\sigma}_1 = [\dot{\sigma}_1 \mid y_1{:}\ \dot{\sigma}_1(\mathsf{res}_1)] \\
\overset{\varphi}{\longmapsto} \langle C_1,\ \sigma_1,\ \dot{\mu}_1\rangle & \text{where } \sigma_1 = \ddot{\sigma}_1 \restriction x_1 \restriction \mathsf{res}_1 \\
\vdots & \text{containing } n-1 \text{ topmost invocations of } m \\
\overset{\varphi}{\longmapsto} \langle C_n,\ \sigma_n,\ \dot{\mu}_n\rangle & \\
\overset{\varphi}{\longmapsto}{}^* \langle C',\ \sigma',\ \dot{\mu}'\rangle. & \text{with no completed topmost invocations of } m
\end{array}$$

Say that any two configurations $\langle A,\ \tau,\ \dot{\mu}\rangle$ and $\langle A',\ \tau',\ \mu\rangle$ are ***matching configurations*** iff $A = A'$, $\tau = \tau'$ and $\dot{\mu} = [\mu + m{:}\ (x : T, \mathsf{res} : U.B)]$ and hence $\mu = \dot{\mu} \restriction m$.

Below, using Lemma 8.4 we will construct a trace via $\overset{\varphi\cup\theta}{\longmapsto}$ that looks as follows:

$$\begin{array}{lll}
\langle C_0,\ \sigma_0,\ \mu_0\rangle & & \\
\overset{\varphi\cup\theta}{\longmapsto}{}^* \langle y := m(z_1);\ C_1,\ \tau_1,\ \mu_1\rangle & \text{matching the configurations above, so } \mu_1 = \dot{\mu}_1 \restriction m & \\
\overset{\varphi\cup\theta}{\longmapsto} \langle C_1,\ \sigma_1,\ \mu_1\rangle & \text{a single step by Lemma 8.4 (2)} & (*) \\
\vdots & \text{containing } n-1 \text{ additional invocations of } m & \\
\overset{\varphi\cup\theta}{\longmapsto} \langle C_n,\ \sigma_n,\ \mu_n\rangle & & \\
\overset{\varphi\cup\theta}{\longmapsto}{}^* \langle C',\ \sigma',\ \mu'\rangle & \text{again matching configurations} &
\end{array}$$

[18]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.

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 Lemma 8.1 we can 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 (39) 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 $v_1 \models_\varphi R^x_{x_1}$ by definition of v_1. (In the trace for $\overset{\varphi}{\longmapsto}$ displayed above, v_1 extends τ_1 with $x_1 : v$, that is, $x_1 : \tau_1(z_1)$.) By Lemma 8.9 we have $\dot{\sigma} = [v_1 \,|\, \mathsf{res}_1\colon \theta(m)(v_1, v_1(x_1))]$ 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 $\langle y := m(z); C_1,\ \tau_1,\ \mu_1\rangle \overset{\varphi\cup\theta}{\longmapsto} \langle C_1,\ [\tau_1 \,|\, y_1\colon \theta(m)(\tau_1, \tau_1(z_1))],\ \mu_1\rangle$. The state $[\tau_1 \,|\, y_1\colon \theta(m)(\tau_1, \tau_1(z_1))]$ 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 $\overset{\varphi}{\longmapsto}$. 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 obtain the needed trace via $\overset{\varphi\cup\theta}{\longmapsto}$ and conclude the proof of Claim B.

Proof sketch for Lemma 8.9. The proof is similar to the argument for (i), (ii), (iii) in the proof, above, of PureLink, though about the body B rather than the client C. So it uses the premise (40) rather than the premise (39). Indeed, we need substitution instances of (40), which are available owing to Lemma 8.8. Thus, for fresh x', res′, we have

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

(Recall that by well-formedness, η is wr -free and does not mention res.)

To deal with recursion, we prove the conditions (a) and (b) of Lemma 8.9 for depth-bounded semantics (Def. 8.6) and arbitrary recursion depth k. This implies Lemma 8.9 for normal semantics, using Lemma 8.7. To prove (a) and (b) for arbitrary k we go by induction on k. For a given k, the argument of (a), not faulting, is similar to argument of (i). The proof of (b), that the final state of execution on the call is given by interpretation θ, is similar to proof of (ii) and (iii) combined. For both parts of condition (b), for a given k we follow the lines of the argument for (i)–(iii), except that where the argument (i)–(iii) appeals to Lemma 8.9, we instead appeal to the inductive hypothesis for depth $k-1$. The gist of those arguments is that we obtain an execution of B via $\overset{\varphi\cup\theta}{\longmapsto}$, to which the premise (40) for B applies. From that premise we get that nothing is written except the result variable, and we get the postcondition $\mathsf{res} = m(x)$, or rather $\mathsf{res}' = m(x')$. So the final state τ is unchanged from the initial state σ except for res' whose value is given by $\theta(m)(\sigma, \sigma(x'))$.

9 DERIVING INTERPRETATIONS FROM IMPLEMENTATIONS

The preceding sections develop a logic featuring a linking rule for pure methods that relies on a provided interpretation that gives meaning to the pure method invocations in pre-, post-, and frame conditions. It relies as well on the syntactic restriction of specification to prevent cyclic references between preconditions, in support of definedness conditions. Aside from purity of effect, the bodies of pure methods are unrestricted commands that may use loops and make mutually recursive calls. In this section we consider the question of how to obtain interpretations.

The Why3 tool[19] provides one answer. Why3 provides a very expressive mathematical language, and enables the use of interactive proof assistants when needed for proofs that are not amenable to full automation. In rough terms, Why3 provides the full power of mathematics, which can be used both for defining candidate interpretations and for semantic reasoning to prove they are context interpretations. However, this answer does not help us to provide a foundation for automated verification tools based on SMT-solvers.

Another answer is suggested by prior works in which interpretations are derived directly from code. In these works, the body of a pure method is restricted to be a single expression, like $\mathsf{res} := E$ in our notation. By Hoare's pun, E is essentially taken to be the mathematical definition of what we call a candidate interpretation, and VCs are defined to check that it is a context interpretation. We proceed to sketch how this idea can be embodied in our program logic. In so doing we give a foundational account for restrictions on specifications proposed in the prior work, and show that the restriction of method bodies is unnecessary.

Consider a pure method specification $\Theta \equiv m : (x{:}T, \mathsf{res}{:}U)R \rightsquigarrow S\ [\eta]$ for which the intended implementation is command B. For the moment, assume there are no calls to p in S or in B. The idea can be sketched using the following variation on rule PureLink.

$$\textsc{PLinkDeriv0}\ \frac{\Phi, \Theta; \psi \vdash_{tot} B : R \rightsquigarrow S\ [\mathsf{wr}\,\mathsf{res}, \mathsf{rd}\,x, \eta] \qquad \Phi, \Theta; \psi \vdash C : P \rightsquigarrow Q\ [\varepsilon]}{\Phi; \psi \vdash \mathsf{let}\ m(x{:}T){:}U = B\ \mathsf{in}\ C : P \rightsquigarrow Q\ [\varepsilon]}$$

Here the notation $\vdash_{tot}$ indicates that the judgment is meant in the sense of total correctness, that is, B must terminate from any initial state satisfying R. As is standard, proof rules for total correctness require that loops are provided with variants (that is, measures that decrease), and similarly for recursion, to which we return later.

The point of the rule is that no candidate interpretation is provided for m (though ψ may provide interpretations for some other pure methods). In establishing the judgment for B, one has no recourse to a specific interpretation of m. Nor would an interpretation be of use, given that for the moment we assume there are no calls to m in B or in the specification. We claim the rule is sound, essentially because it ensures that an interpretation can be *derived* from B.

In light of the pure method specification which allows no effects, we can define by standard techniques a denotational semantics of B as a function from states to values (the final value of res) and $\lightning$ (if B faults or diverges). We refrain from spelling out the denotational semantics; it is similar to the one defined in Appendix Section A.1 for other purposes. So the denotational semantics provides a candidate interpretation, say θ, for m.

By the total correctness judgment, we get a value (not $\lightning$) for every initial state that satisfies precondition R. That is, the conditions required for θ to be a context interpretation, that is, $\Phi, \Theta; \psi \models$

[19]See why3.lri.fr and our Section 10.

θ, are immediate consequences of the judgment for B. Moreover, this is valid: $\Phi, \Theta; \psi, \theta \vdash B : R \leadsto S\ [\mathsf{wr}\ \mathsf{res}, \mathsf{rd}\ x, \eta]$. Now conclusion of PLinkDeriv0 follows by exactly the proof of PureLink.

Now let us consider the general case, where there may be calls to m in its body and in its specification. Indeed, let us consider multiple methods, among which there may be mutually recursive calls in the bodies and in the specifications. Let us also assume there are no loops, so nontermination can only be due to recursion. A standard example of mutual recursion are the even and odd functions, here specified in a form taken from Leino and Müller [39].

$$\begin{array}{l} isEven : (n : int, \mathsf{res} : bool)\, 0 \le n \leadsto \mathsf{res} = (\mathit{if}\ n = 0\ \mathit{then}\ true\ \mathit{else}\ isOdd(n-1))\ [\,] \\ isOdd : (m : int, \mathsf{res} : bool)\, 0 \le m \leadsto \mathsf{res} = \neg isEven(m)\ [\,] \end{array}$$

Obvious implementations are the commands if $n = 0$ then res := $true$ else res := $isOdd(n - 1)$ and res := $\neg isEven(m)$.

It is well known that for total correctness the linking rule needs to include a measure that is decreased in recursive calls. The details can be intricate in case of mutual recursion—and even more so in the presence of dynamic dispatch, which is outside the scope of this article. We follow the succinct and general formulation of Apt et al. [3]. In our notation, for a single method, the idea can be sketched as follows:

$$\text{PLinkDeriv}\ \frac{\begin{array}{c}\Phi, m : (x{:}T, \mathsf{res}{:}U) R \wedge t < z \leadsto S\ [\eta]; \psi \vdash B : R \wedge t = z \leadsto S\ [\mathsf{wr}\ \mathsf{res}, \mathsf{rd}\ x, \eta] \\ \Phi, \Theta; \psi \vdash C : P \leadsto Q\ [\varepsilon] \qquad R \Rightarrow t \ge 0\end{array}}{\Phi; \psi \vdash \mathsf{let}\ m(x{:}T){:}U = B\ \mathsf{in}\ C : P \leadsto Q\ [\varepsilon]}$$

Here t is an integer expression and z is a fresh variable, that is, z does not occur in any of the code or specifications. Typically t is an expression in the method parameters. The judgment for B is changed in two ways, so that t can serve as measure for the size of inputs to recursive calls. The equation $t = z$ is conjoined with the precondition for B, to snapshot the initial value of t, and $t < z$ is conjoined with the precondition in the hypothesis for m.[20] Rule PLinkDeriv ensures that the measure is bounded from below and decreases in recursive calls. For linking mutually recursive procedures, the judgment for each procedure body is modified as above, using the single expression t for all.[21]

We would like to argue for soundness of PLinkDeriv as follows. As in the proof of recursion lemma 8.9, we argue by induction on depth-bounded semantics that the premise for B is valid, and moreover B is terminating from all initial states that satisfy precondition R. So a candidate interpretation θ could be derived from the semantics of B, and as a consequence of the judgment for B it would be a context interpretation.

The hope is that B is correct, with respect to the interpretation given by its own denotation. But what if the specification is unsatisfiable (that is, there are some states that satisfy R for which no result exists satisfying S)? Then the premise for B is valid, for any B whatsoever, since it quantifies over all context interpretations. This renders the rule PLinkDeriv0 unsound because calls to m in the client C need not behave as specified. Rule PureLink precludes this: the required context interpretation witnesses satisfiability of the specification. What about PLinkDeriv?

In prior work it was proposed to consider calls to m in its own postcondition, or more generally calls in postconditions of several mutually recursive methods, as recursive calls that should decrease a measure. Rule PLinkDeriv imposes decrease of the measure for calls in B, but it is not obvious how to express the same restriction on calls in postcondition S or in the frame condition η. One

[20]The rule needs to be used together with a rule for instantiation (rule Subst in RLI), to adapt the assumed specification to calls of m. See the discussion in Apt et al. [3] of their Rule 13 Recursion IV. Substitution for z is disallowed.
[21]This may involve some encoding. For the $isEven/isOdd$ example, a ghost variable can be used to encode that the argument number is decreased when calling $isOdd$ and not increased when calling $isEven$.

possibility is to make use of a variation of the definedness conditions (Fig. 5) that imposes the extra precondition $t < z$. We do not formalize the details here. Assuming the restriction is imposed, let us proceed to argue why such restrictions suffice to show soundness of PLinkDeriv.

The idea is to generalize context interpretations (Def. 5.1) to ***approximate context interpretations*** that are allowed to $\lightning$ on some initial states satisfying the precondition. The definition of valid judgment (Def. 5.2) is changed to allow approximate context interpretations. As remarked following that definition, it is well defined even in case methods are called (in formulas and effects) outside their preconditions. Soundness of PLinkDeriv hinges on a recursion lemma (akin to Lemma 8.9) that provides a semantic result that amounts to validity of the premise for B but with context calls replaced by environment calls. The lemma is proved using the depth-bounded semantics (Def. 8.6) and goes by induction on depth k. Whereas the proof of Lemma 8.9 relies on the given interpretation for semantics of formulas and effects, to prove PLinkDeriv we define a chain of approximate interpretations θ_k using the denotational semantics of B. (A similar construction is carried out in detail, for the proof of ImpureLink, in the Appendix Section A.1.) The measure restrictions ($t < z$) ensure that these interpretations are only applied to inputs for which they deliver a non-$\lightning$ result, so we retain two-valued reasoning about formulas.

In summary, this approach is a more nuanced take on the role of definedness conditions and healthiness conditions. Notice that the measure restrictions are imposed in the judgment for method body B, but are not required for the specifications used to verify the client. Moreover, the user of the logic does not explicitly define an interpretation—indeed it may not be expressible in assertion language (or only expressible via encodings with no practical use). Nor can they rely on an interpretation for reasoning about formulas. So in proving the judgment for body B, calls to m in formulas and effects become opaque, just as they are for the client: one can reason from the spec of m but not its definition.

10 CASE STUDIES

To explain how pure methods have been encoded in verification tools based on SMT solvers, and to demonstrate how our logic accounts for such encodings, this section presents verifications of the running examples using the Why3 tool.[22] Why3 implements procedure-modular reasoning for a first-order intermediate language. This tool is used because it provides VC-gen for a convenient intermediate language with pre-post contracts, while enforcing a strict separation between program code and mathematical formulas. This allows us to focus on VC-gen for pure methods, without the need to delve into the routine aspects of translating pre-post contracts and code annotated with assertions into FOL formulas. Why3 generates VCs for a range of provers but we use only SMT solvers (Alt-Ergo, CVC3, CVC4, and occasionally Z3).

The full developments are provided as supplementary material for this article.[23] Here we present highlights, omitting for example the import clauses in theories and modules. In principle our encodings could be automatically generated from the source programs and specifications, but that is not our purpose here.

Preliminaries. Why3 is carefully designed to make a clear distinction between mathematical definitions and programs. A mathematical function is indicated by keyword **function**; it may have a mathematical definition (in Why3 syntax) or be left uninterpreted. Whereas a Why3 'theory' contains only math, a Why3 'module' can contain program functions with mutable state, ghost code, contracts, and annotations including assertions and loop invariants. A program function is

[22] why3.lri.fr

[23] The Why3 files are available at http://www.cs.stevens.edu/~naumann/pub/readRLWhy3.tar

indicated by keyword **let** and is written in WhyML syntax; it has a contract and body. Keyword **val** declares a program function without a body.

Roughly speaking, the VC for a program C with specification $P \leadsto Q\,[\ldots]$ is a first-order formula $P \Rightarrow wp(C, Q)$, validity of which is equivalent to the judgment $C : P \leadsto Q\,[\ldots]$. The formula $wp(C, Q)$ is defined by structural recursion on C to be an approximate weakest (liberal) precondition, taking into acount loop invariants, other annotations, and contracts for functions called in C. Source-language notations in P and Q are desugared in various ways, and additional antecedents may be used to axiomatize the programming language semantics as well as problem-specific mathematical definitions and facts. Just as our logic requires judgments to be healthy, VCs include definedness checks for specifications.

When targeting SMT solvers, Why3 has to translate a recursively defined mathematical function into first-order logic, which it does by means of an uninterpreted function and ***definitional axioms*** that express the defining clauses. Any reasoning system that allows recursive definitions relies on proof obligations or syntactic checks that ensure the recursion is well founded. Why3 relies on syntactic checks based on structural recursion. More on basic VC-gen can be found in [20, chapt. 5] and [37]. The latter includes the encoding of pure methods, along the lines that we discuss in the following.

The Why3 language (including WhyML) allows mutable objects but sharing is very limited by design [27]. So we use mutable records and maps to explicitly model the heap in a standard way. We treat references as an uninterpreted type, called reference as in other sections of this article. Note that Why3 provides a library module for ML-style references, subject to restrictions that disallow aliasing, but we are not using that module. In our model, each field is a map on references. The heap is a record with a mutable field for each of these maps, together with a map that designates which references are allocated and what are their types. Why3 features an 'invariant' declaration to be associated with a data type; these invariants are enforced on procedure call/return boundaries. We use data type invariants for well-formedness conditions on the heap that are ensured by type safety of Java-like source languages: fields have type correct values and there are no dangling pointers.

Specifications in Why3 feature coarse-grained frame conditions—reads and writes clauses for mutable fields of records—which are checked by simple syntactic analysis. We encode the finer-grained frame conditions of our logic using postconditions and frame axioms. Why3 provides ghost annotations and checks that ghost code does not interfere with the underlying program. We use this feature to mark the allocation map, which is part of our heap model. We also use it to mark ghost state in the examples, even though we do not formalize ghost annotations in the logic.

It is straightforward to encode a pair of heaps connected by a refperm (Section 4), representing the refperm as a pair of maps subject to universally quantified axioms that one is inverse of the other. However, for the developments in this section we only need identity refperms in agreements, because we are not checking read effects of impure methods (see remark following Definition 6.4).

Encoding of pure methods. We verify client code, named main, with respect to a method context Φ and linked with implementations of the methods in Φ. In accord with the proof rules PureLink and ImpureLink, main should be verified in a context that provides contracts for the pure and impure methods in scope. For an impure method m we make a single declaration, $mCode$, with contract from $\Phi(m)$. Why3 generates VCs to verify that its body satisfies its contract, and generates VCs for main that include the contract of $mCode$.

For a pure method p with specification $\Phi(p) = (x{:}T, \mathsf{res}{:}U)R \leadsto S\,[\eta]$ we make three declarations in Why3:

- $pCode$ is a program function, including the implementation code of p, defined using the Why3 **let** construct.

- $pUnInt$ is an uninterpreted function for use in specifications (where Why3 disallows the use of $pCode$).
- $pInt$ is a defined mathematical function, the chosen interpretation of p.

These are used in accord with rule PureLink. The definition of $pInt$ is subject to Why3's restrictions that ensure it is well defined as a total function. Invocations of p in the code of main are invocations of $pCode$ and so the VCs for main include the contract of $pCode$. We augment the contract $\Phi(p)$ with postcondition $\mathsf{res} = pUnInt(x)$ in the contract for $pCode$.

For reasoning about p in specifications, there are two axioms about $pUnInt$. The ***frame axiom*** says that $pUnInt$ satisfies the read effect: if σ and σ' agree on η and on the value of argument expression E then they agree on the value of $pUnInt(E)$. The ***pre-post axiom*** says that $pUnInt$ satisfies its postcondition: $(\forall x{:}T \cdot R \Rightarrow S^{\mathsf{res}}_{p(x)})$. These axioms express the contract $\Phi(p)$. They are named as $pFrame$ and $pPrePost$ respectively, and are made available for verifying the client command. In rule PureLink the contract is an explicit hypothesis for the client command, and for reasoning about the implementation of p one can use the contract because it is available indirectly through the side condition $\theta \models \Phi, \Theta; \psi$ and postcondition $\mathsf{res} = p(x)$. As expected, one can prove partial correctness of the non-terminating implementation of p that simply calls itself with the given arguments.

For verification of the body of $pCode$, the interpretation $pInt$ should be available, so the implementation of $pCode$ begins with an **assume** statement that says $pUnInt$ is the same as $pInt$. On the other hand, no such assumption is provided for main, so its verification is done without an interpretation of p. Of course it is also possible to make $pInt$ available for main, providing for the form of reasoning in the proof rule TranspPureLink, but we do not do that in the case studies. As noted in Section 12, several prior works feature that form of reasoning and thus definitional axioms for pure methods.

To justify the frame axiom for $pUnInt$, it is formulated and proved as a lemma about $pInt$. To justify the pre-post axiom for $pUnInt$, we do not directly prove that $pInt$ satisfies the postcondition; instead, the contract is verified for $pCode$. The additional postcondition $\mathsf{res} = pUnInt(x)$, together with the assumption connecting $pUnInt$ with $pInt$, entails that $pInt$ satisfies the postcondition. This deviates slightly from the formulation of PureLink, but is needed because Why3 provides contracts for code (our $pCode$) but not for defined functions (our $pInt$).

Generation and validation of VCs is different from deductive proofs using inference rules for correctness judgments, but it may be helpful to note some parallels. Some proof rules in our logic require $(\Phi; \psi)$-validity of certain formulas P for some relevant method context Φ and partial interpretation ψ (written $\Phi; \psi \models P$ according to Def. 5.4). In Why3, the $pInt$ functions play the role of ψ and the pre-post and frame axioms for the $pUnInt$ functions play the role of Φ in proving formulas.

Hiding data invariants. Although the logic in this article does not formalize the hiding of invariants, we do take some care with information hiding in the case studies. To do so, we make use of the Why3 module system, but that does not provide direct support for hiding of invariants. Internal invariants like the ones in the Composite example do not appear in the contracts (which are public) but are encoded using assume and assert statements within the relevant code. Our treatment of hiding is intended to have tutorial value, not to be a comprehensive treatment.

10.1 Cell in Why3

The Cell example (Figures 1 and 16) is implemented in Why3 using one theory and three modules. A theory only contains logical definitions.

```
theory Reference
   type reference
   constant null: reference
   type rtype = Null | Unalloc | Cell
end
```

All of our examples in Why3 have such a theory, which is the basis for formalizing a Java-like data model in which allocated references have an immutable class type. 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, using maps and sets from the Why3 library.

```
type heap = {
   ghost mutable alloct: map reference rtype;
   mutable value: map reference int;
   ghost mutable foot: map reference (set reference);
}
```

A heap is a record. The first field, alloct, records the allocated references and their types. In a given heap, the set of p for which alloct[p] is not Unalloc or Null corresponds to the value of variable alloc in our logic. Module Heap also declares, as a Why3 data type invariant, the heap typing invariants that are ensured by typing rules for our programming language but are not expressed as Why3 types. Whereas the value field is given type int in the Why3 declaration of heap, we need to use an invariant to express that the region-valued field foot has no dangling references, that is, the field has a value in accord with its type in our source language. (In Section 10.2 we use an axiom to express the same typing constraint on the result of the anc function.)

```
invariant { self.alloct[null] = Null ∧
     (∀ p: reference. self.alloct[p] = Null ⇒ p = null) ∧
     (∀ p q: reference. self.alloct[p] = Cell ⇒ mem q self.foot[p] ⇒ self.alloct[q] ≠ Unalloc)
}
```

In Why3, the keyword self refers to an instance of the type with which an invariant is associated, in this case heap. Function mem is from the Why3 library module Set, from which we also use intersection (named inter) and other functions. We typeset implication as ⇒ and let it associate to the right.

In addition to the typing of field foot, the invariant also says that the one and only reference mapped to Null is null. This is a minor detail that is convenient because null is a value of every reference type. 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.

The formula I, used as a public invariant in the specifications of the Cell methods in Figure 16, is defined in module Heap as well, but named iInv.

```
predicate iInv1 (h: heap) = ∀ p: reference. h.alloct[p] = Cell ⇒ mem p h.foot[p]
predicate iInv2 (h: heap) = ∀ p q: reference. h.alloct[p] = Cell ⇒ h.alloct[q] = Cell ⇒
                                   p ≠ q ⇒ is_empty (inter h.foot[p] h.foot[q])
predicate iInv (h: heap) = iInv1 h ∧ iInv2 h
```

Finally, module Heap defines agreement on a reference set, specialized to the fields of Cell to interpret effects that refer to the data group ‘any’.

```
predicate agreeOnAny (h h′: heap) (ftp: set reference)
  = ∀ p: reference. mem p ftp ⇒ h.value[p] = h′.value[p] ∧ h.foot[p] = h′.foot[p]
```

Module Cell gives the specifications and implementations for the methods of class Cell (see Figure 16). For pure method get the uninterpreted and interpreted functions are as follows.

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

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

axiom getFrame: ∀ h h′: heap. ∀ p: reference.
  agreeOnAny h h′ h′.foot[p] ⇒ getUnInt h p = getUnInt h′ p
```

The axiom getFrame expresses the read effect rd self.$foot$‘any. We omit the pre-post axiom for get because it is trivial. The read effect is verified for getInt in the following form:

```
lemma framegetInt: ∀ h h′: heap. ∀ p: reference. h.alloct[p] = Cell ⇒ h′.alloct[p] = Cell ⇒
  mem p h.foot[p] ⇒ agreeOnAny h h′ h.foot[p] ⇒ getInt h p = getInt h′ p
```

Note that this amounts to proving an interpretation $\theta(get)$ satisfies the frame condition in a specification $\Theta(get)$, which is part of the context interpretation side condition written $\theta \models \Phi, \Theta; \psi$ in rule PureLink. Aside from the read effect, we encode the pre-post condition as a contract on getCode, though in this case there is no non-trivial postcondition. So the only thing to prove about getCode is that its result equals that of getInt, in accord with the postcondition res = $get(self)$ in the premise for get in PureLink. This is expressed as a postcondition in the contract for getCode.

```
predicate getInterpreted =
  ∀ h: heap. ∀ p: reference. h.alloct[p] = Cell ⇒ getUnInt h p = getInt h p

let getCode (h: heap) (s: reference): int
    requires { iInv h ∧ h.alloct[s] = Cell }
    ensures { iInv h ∧ getUnInt h s = result }
=
    assume { getInterpreted };
    h.value[s]

let setCode (h: heap) (s: reference) (v: int): unit
    requires { iInv h ∧ h.alloct[s] = Cell }
    ensures { iInv h ∧ getUnInt h s = v }
    writes { h.value }
    ensures { ∀ p: reference. h.alloct[p] = Cell ⇒
                    not mem p h.foot[s] ⇒ agreeOnAny h (old h) (h.foot[p]) }
=
    assume { getInterpreted };
    h.value ← set h.value s v
```

Owing to the assumption getInterpreted, the verification of getCode and setCode can rely on the definition of getInt as well as its frame condition, for reasoning about getUnInt which appears in the specifications. (Indeed, the frame condition is available both as an axiom about getUnInt and as a lemma about getInt.) 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 setCode 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.

One last feature of our Why3 model of the heap is the semantics of allocation. It is specialized to the relevant classes, here just Cell. A previously unallocated reference is chosen nondeterministically and its fields initialized to default values.

```
val newCell (h: heap): reference
    ensures { (old h).alloct[result] = Unalloc }
    ensures { h.alloct = Map.set (old h).alloct result Cell }
    ensures { h.foot[result] = empty ∧ h.value[result] = 0 }
    writes { h.alloct }
```

The last part of module Cell is the constructor as specified in Figure 16 and adapted to the Why3 specification for newCell. As usual, the postcondition following the **writes** clause is our fine-grained frame condition.

```
let init (h: heap) (s: reference): unit
    requires { h.alloct[s] = Cell ∧ h.foot[s] = empty }
    requires { ∀ p: reference. h.alloct[p] = Cell ⇒ p ≠ s ⇒
        (not mem s h.foot[p] ∧ mem p h.foot[p]) }
    requires { ∀ p q: reference. h.alloct[p] = Cell ⇒ h.alloct[q] = Cell ⇒ p ≠ s ⇒ q ≠ s
         ⇒ (p = q  ∨  is_empty (inter h.foot[p] h.foot[q])) }
    ensures { iInv h }
    writes { h.foot}
    ensures { ∀ p: reference. h.alloct[p] = Cell ⇒ p ≠ s ⇒ h.alloct[p] = Cell ⇒
        h.foot[p] = (old h).foot[p] }
=
    h.foot ← set h.foot s (singleton s); ()
```

(The trailing '()' is needed so Why3 can determine, by syntactic analysis, that the result does not depend on ghost state.)

The last module provides a client, the framing example from the beginning of Section 1.

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

Note the absence of an assumption connecting getUnInt with the interpretation getInt.

To verify the client and all the preceding code, the splitting tactic provided by Why3 generates 23 goals that are automatically verified using CVC4 in 3 seconds. Verification of main relies on framing; it fails to verify if axiom getFrame is removed.

10.2 Composite in Why3

Next we consider the example in Figure 2. To show that our approach caters for weak specifications, we retain the under-specified contracts as in the Figure, though invariants are added in order to verify the implementations. Where possible, our formalization uses annotations similar to those used by Rosenberg et al. [53], to facilitate comparison.

The Reference theory for Composite is just like the one for Cell, but with type Comp instead of Cell. Aside from the Reference theory, the Why3 file for Composite contains six modules. The first of these modules is Heap.

```
type heap = {
    ghost mutable alloct: map reference rtype;
    mutable chrn: map reference (list reference);
    mutable size: map reference int;
    mutable parent: map reference reference;
    ghost mutable depth: map reference int;
}
```

As in the Cell example, alloct is a ghost field that keeps track of allocated references and their types. The others fields are those of class Comp. The map chrn represents list of children for a Composite node. The main difference between our version and that of [53] is that we do not use a ghost field for keeping track of descendants of a Composite (which required a number of invariants). Instead we define a pure method anc which recursively computes the ancestors of a given Composite and use it directly to reason about the ancestors of each Comp node.

As in the Cell example, our encoding includes an invariant for type safety. For integer field size the Why3 type suffices. But for parent and chrn the Why3 type merely constrains the value to be a reference and a list of references respectively. Typing rules of our source language ensure that the parent is either null or allocated and of type Comp. Similarly for chrn. Unlike the Cell example, the Composite is sufficiently complicated that we need to refer to type safety conditions in lemmas, which are not in the scope of Why3's data type invariants. So we define and use a predicate on heaps, named okHeap, to express the type safety conditions.

The second module, CompInvs, defines some predicates to be used as private invariants for class Composite. The first invariant is a 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 specifies the depth of an element in the composite tree.

$$\forall p : Comp \;\cdot\; p \neq \mathsf{null} \Rightarrow \begin{array}{l} p.depth \geq 0 \\ \wedge(p.parent = \mathsf{null} \Leftrightarrow p.depth = 0) \\ \wedge(p.parent \neq \mathsf{null} \Rightarrow p.depth = 1 + p.parent.depth) \end{array}$$

In Why3 code, the name ptcdInv is used for the conjunction of the first two invariants. Maintaining these two predicates as invariants helps to ensure the acyclicity of the chain of parents for any given composite. The third 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 this invariant is called sizeInv.

The last part of CompInvs module is a mathematical definition of the list of ancestors, which is the basis for the interpretation, ancInt, of method *anc*. Whereas we used a set for the footprint region in Section 10.1, here we encode ancestors as a list. The reason is that we need induction on ancestors, and the Why3 library module for finite sets is less well developed than the one for lists.

Based on the code in Fig. 2, one might like to formulate the definition as follows, for non-null references p of type $Comp$.

$$f(p) \mathrel{\widehat{=}} \mathit{if}\ p.parent = \text{null}\ \mathit{then}\ Cons\ p\ Nil\ \mathit{else}\ Cons\ p\ (f(p.parent))$$

This makes p an ancestor of itself, as intended. However, this is a bogus definition, owing to the possibility of cycles via parent. Indeed, Why3 rejects such a definition. Instead we define a predicate is_loa with a list argument supporting a well founded induction. This technique is often used in separation logic [52]. The predicate is_loa h p l says that l is the list of ancestors of p in heap h.

```
inductive is_loa (heap) (reference) (list reference) =
  | Nil_l: ∀ h: heap, p: reference. okHeap h ⇒ ptcdInv h ⇒ is_loa h null Nil
  | Tree: ∀ h: heap, p: reference, l: list reference. okHeap h ⇒ ptcdInv h ⇒
        h.alloct[p] = Comp ⇒ is_loa h h.parent[p] l ⇒ is_loa h p (Cons p l)
```

The depth condition in ptcdInv rules out cyclic parent structure, so we can prove lemmas, not shown, that the ancestor list exists and is unique. (It is our only use of the ghost field depth.) This justifies an axiom to define function anc:

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

axiom anc_def: ∀ h: heap,  p: reference. okHeap h  ⇒ ptcdInv h ⇒ okReft h p Comp ⇒
      is_loa h p (anc h p)
```

The predicate okReft h p Comp above indicates that p is a value of type Comp, that is, p is either null or allocated in heap h with type Comp.

The next two modules, 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. For getSize the code is as follows.

```
function getSizeUnInt (h: heap) (s: reference): int

axiom getSizeFrame: ∀ h h′: heap, s: reference. h.alloct[s] = Comp ⇒
  h′.alloct[s] = Comp ⇒ h.size[s] = h′.size[s] ⇒ getSizeUnInt h s = getSizeUnInt h′ s

axiom getSizePrePost: ∀ h: heap, s: reference. h.alloct[s] =  Comp ⇒
  getSizeUnInt h s = h.size[s]

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

lemma framegetSizeInt: ∀ h h′: heap, s: reference. h.alloct[s] = Comp ⇒
  h′.alloct[s] = Comp ⇒ h.size[s] = h′.size[s] ⇒ getSizeInt h s = getSizeInt h′ s
```

```
predicate getSizeInterpreted =
  ∀ h: heap, p: reference. h.alloct[p] = Comp ⇒ getSizeUnInt h p = getSizeInt h p

let getSizeCode (h: heap) (s: reference): int
   requires{ h.alloct[s] = Comp }
   ensures{ result = getSizeUnInt h s }
=
   assume { ptcdInv h ∧ sizeInv h ∧ getSizeInterpreted };
   h.size[s]
```

The axiom getSizeFrame is the read effect for *getSize* in terms of getSizeUnInt. Using RL effect syntax, this is rd self.*size*. Lemma framegetSizeInt shows that read effect is correct for the given interpretation. Unlike our other examples, the postcondition of getSize is nontrivial so its pre-post axiom is included, named getSizePrePost.

For method *anc* there is an additional axiom, ancUnInt_type, that says its result is a region, that is, no dangling references. (Compare with the invariant for field foot in Section 10.1). We omit the pre-post axiom because in this case it is trivial.

```
function ancUnInt (h: heap) (s: reference): list reference

axiom ancUnInt_type: ∀ h: heap, p q: reference. h.alloct[p] = Comp ⇒
  mem q (ancUnInt h p) ⇒ h.alloct[q] ≠ Unalloc

axiom ancFrame: ∀ h h′: heap, p: reference. h.alloct[p] = Comp ⇒
  h′.alloct[p] = Comp ⇒ (∀ q: reference.  mem q (ancUnInt h p) ⇒ q ≠ null ⇒
  h.parent[q] = h′.parent[q]) ⇒ ancUnInt h p = ancUnInt h′ p

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

let rec lemma frameancInt (h h′: heap) (p: reference)
   requires { h.alloct[p] = Comp ∧ h′.alloct[p] = Comp }
   requires { ∀ q: reference. mem q (ancInt h p) ⇒ h.parent[q] = h′.parent[q] }
   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 = ∀ h: heap, p: reference. okReft h p Comp ⇒
   ancUnInt h p = ancInt h p

let rec ancCode (h: heap) (s: reference): list reference
```

```
        requires{ h.alloct[s] = Comp }
        ensures{ result = ancUnInt h s }
        variant { ancUnInt h s }
    =
        assume { ptcdInv h ∧ sizeInv h ∧ ancInterpreted };
        if h.parent[s] = null then Cons s Nil else Cons s (ancCode h h.parent[s])
```

The axiom ancFrame is the read effect for ancUnInt; in RL syntax, rd $(self.anc())`parent$. 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 frameancInt has an inductive proof that shows read effect is correct for ancInt. The implementation of anc is given as a recursive function. We verify that it returns the same list of ancestors as ancUnInt.

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

A private ghost method depthUpdate is defined after addtosizeCode. For a given heap h and reference x this method updates the depth of x and its descendents to ensure ptcdInv h.

The last method in the Composite module is addCode.

```
let addCode (h: heap) (s x: reference): 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, h.depth }
      ensures { ∀ p: reference. h.alloct[p] = Comp ⇒
         not (mem p (ancUnInt (old h) s)) ⇒ h.size[p] = (old h).size[p] }
      ensures { ∀ p: reference. h.alloct[p] = Comp ⇒ x ≠ p ⇒
         h.parent[p] = (old h).parent[p] }
=
      assume { ptcdInv h ∧ sizeInv h ∧ getSizeInterpreted ∧ ancInterpreted };
      let l = ancCode h s in
      h.chrn ← set h.chrn s (Cons x h.chrn[s]);
      h.parent ← set h.parent x s;
      depthUpdate h x;
      addtosizeCode h s h.size[x];
      assert{ l = ancInt h s ∧ sizeInv h ∧ ptcdInv h }
```

For a heap h and two references x and s, this method adds x as a child of s, provided that parent of x is 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.$chrn$; however, 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 immutable 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 from the beginning of Section 2. It does not import CompInvs, as it relies on the public specifications of Composite and not on the private invariants or the interpretations ancInt and getSizeInt. The client code is as follows.

```
let main (h: heap) (b c d: reference): unit
    requires { h.alloct[b] = Comp ∧ h.alloct[c] = Comp ∧ h.alloct[d] = Comp }
    requires { not mem d (ancUnInt h b) ∧ h.parent[c] = null }
    requires { not (mem c (ancUnInt h b)) ∧ not mem c (ancUnInt h d) }
  =
    let i = getSizeCode h d  in
    addCode h b c;
    assert { i = getSizeUnInt h d ∧ i = h.size[d] };
```

This uses immutable variable i, but the heap h is mutable and indeed updated by addCode. The main point of the example is that the value of $d.getSize()$, computed by getSizeCode, is unchanged by the call to addCode, and this fact can be established owing to its frame condition. The preconditions of main reflect those of the call to add. In addition to the postcondition i = getSizeUnInt h d from the beginning of Section 2, we prove a second postcondition: i = h.size[d]. This follows by the pre-post axiom, getSizePrePost.

Using the module 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. The first postcondition (i = getSizeUnInt h d) fails to verify if the frame axiom for getSize is omitted, and the second postcondition (i = h.size[d]) fails to verify if the pre-post axiom is omitted.

To verify the client and all the preceding code, Why3 generates 23 goals (coincidentally the same number as for Cell). We use the splitting tactic provided by Why3 only for goals that are not otherwise verified. Using Alt-Ergo, CVC3 and CVC4, the goals all verify, in a total of 83 seconds.

11 WEAK PURITY

In this article, a pure method's code can have no effects except writes to local variables and non-termination; and the associated interpretation must be total on states that satisfy the method's precondition. Much of the prior work on pure methods addresses ***weak purity*** which allows the additional effect of allocation; a weakly pure method may use and even return a reference to freshly allocated objects. This section reviews weak purity with reference to prior works. Section 12 reviews related work in connection with other topics.

The use of pure and weakly pure methods in specifications is motivated by the need for functional abstraction and by the opportunity to leverage, for purposes of specification, pure functions that happen to be defined as part of the program under consideration. To introduce issues raised by weak purity we quote from the survey paper of Hatcliff et al. [29], which identifies three problems with pure methods. One is "verifying that the method is well defined and does not lead to an unsound axiomatization" (they cite [25] and [23] which we discuss further in Section 12). The issue is that most reasoning systems are based on total functions whereas code can diverge. To reconcile potential non-termination of pure methods with their use in specifications it suffices to leverage preconditions and definedness conditions, as shown in the preceding sections of this article.

The second problem identified by Hatcliff et al. is "checking that the method really is pure, that is, free of side effects. This is not as simple as a syntactic check for assignment statements, because

```
pure method plus1(self: Cell): Cell
    requires self ≠ null ∧ I
    ensures get(result) = get(plus1(self)) ∧ I
    reads self.foot‘any, self.foot, alloc
    writes alloc
```

Fig. 17. Specifications of method plus1

one wants to allow the method body to, for example, allocate new objects (perhaps an iterator object or a string builder) and modify their state. Hence, the desired check is that of observational purity, which says that the method may have some side effects, but it appears pure to any observer." (Here they cite [12, 21, 46, 55]). If the side effects are of any use, they are surely observable in the intended context; what is meant here is any observer outside the relevant encapsulation boundary. Observational purity has been identified decades ago [32] but has yet to be formalized in a program logic or general verification system that handles object-based programs. For example, RLII has a proof rule for hiding of invariants on encapsulated state but not for hiding of effects on it.

Weak purity is an attractive special case of observational purity: it applies in many practical situations but appears to avoid the need to bring encapsulation into the picture. Put differently, allocation in Java-like languages is an abstraction with a built-in encapsulation boundary: owing to absence of pointer arithmetic, the state of the allocator is not observable. The most broadly relevant kind of application is where a query method needs to allocate data structures used to compute a result, and possibly to represent that result, in the program itself.[24] Several verification systems allow the use of weakly pure methods in specifications [22, 23, 25, 59] with varying degrees of justification. The current trend, however, seems to be to insist on strong purity, as in Dafny [38] and Why3. Our analysis in this section may help explain the trend.

Allocation is one cause of Hatcliff et al.'s third problem with pure methods: they "are not necessarily deterministic. More precisely, calling a pure method twice may yield two different results, because the side effects supposedly not visible to observers cause the second call to start in a slightly different state. This means that pure methods cannot be represented as mathematical functions of their arguments and of an unchanging heap. One solution is to make the definition of observational purity strict enough to avoid this situation; another is to allow slightly different results and to prevent callers from assuming anything more than some sort of equivalence between the results." (Here citing [40]).

As an example the Java expression new $Cell()$ == new $Cell()$ is always false—apparently violating reflexivity of equality! Our desugared language does not include such expressions; instead, allocation is at the level of commands. Nonetheless the problem is still present in our language, in which the phenomenon can be illustrated this way: the command var $y, z : Cell$ in $y :=$ new $Cell$; $z :=$ new $Cell$ establishes postcondition $y \neq z$.

The idea of the first solution is to restrict programs and specifications sufficiently that it is sound to ignore the effects. To make this precise and to justify it, a first step is to consider ***threaded semantics*** of expressions in formulas [25]. This is just like ordinary program semantics of expressions that can have side effects: to evaluate the comparison new $Cell$ = new $Cell$ in a state σ, first the left argument new $Cell$ is evaluated, yielding both a value and an updated state τ in which the

[24]The other kind of application has been featured in JML, which has been designed to avoid the need for mathematical types beyond those of the programming language itself, so in particular ubiquitous mathematical abstractions like sets and sequences are defined by Java libraries rather than logical theories.

second argument is evaluated. Threaded semantics is necessary to make sense of expressions in which subexpressions can allocate. For example, consider the specification in Fig. 17 of a method that computes the successor of $Cell$, as a new object. To evaluate $get(plus1(y))$, the evaluation of get must be in the new state in which the reference returned by $plus1(y)$ is allocated. In threaded semantics, the expression $plus1(y) = plus1(y)$ is always false. So no implementation could satisfy the postcondition res $= plus1(self)$ stipulated by PureLink in Figure 14. The specification in Figure 17 tries to avoid the problem by expressing a suitable equivalence using equality of integers, as in the second solution in the quote above.

Naumann's semantic analysis [46] considers these issues in connection with reconciling runtime assertion checking with static verification and shows how weakly pure expressions and assertions do not cause problems provided the assertions are insensitive to garbage collection and differences in allocation. As in the present paper, differences in allocation are 'quotiented away' in the semantics of read effects, using refperms. (Refperms also serve as basis for defining notions of agreement for purposes such as program equivalence [7].) However, the semantic analysis only deals with the interface between assertions and code. In this paper we are dealing with a program logic, with elements including definedness conditions which are also needed in verification tools.

In semantics, axiomatic or otherwise, it is not difficult to formalize the threading of state through semantics of expressions since the heap is an explicit parameter (see Section 10). There is a price however: by explicitly modeling the program semantics, such a formalization is partly breaking the pun of Hoare logic, whereby program variables and expressions are directly translated to mathematical variables and expressions.

In keeping with the source language level of abstraction, verification tools like those cited in this article do not model garbage collection. Their specification languages, however, provide means to refer to the currently allocated objects, e.g., for the range of quantifiers. This admits formulas that are not garbage insensitive.

The next considerations are whether effects need to be threaded through the semantics of formulas and the semantics of effects. Let us begin with conjunction. If formulas P and Q have weakly pure subexpressions, should Q be evaluated in the state after P has been evaluated, that is, threaded, or should ***snapback*** semantics be used, in which Q is evaluated in the original state? Consider these formulas:

$$Pa \mathrel{\widehat{=}} \mathsf{alloc} = \varnothing \qquad Pf \mathrel{\widehat{=}} \forall x : Cell \in \mathsf{alloc} \cdot false \qquad Pn \mathrel{\widehat{=}} get(make()) > 0$$

where $make$ is a pure method that returns a new $Cell$. In threaded semantics, $Pa \land Pn$ is satisfiable whereas $Pn \land Pa$ is unsatisfiable, and *mutatis mutandis* for Pf and Pn—breaking the symmetry of conjunction. Idempotence of conjunction is also broken by threaded semantics: $Pa \land Pn$ is satisfiable but $(Pa \land Pn) \land (Pa \land Pn)$ is not. Given that only boolean values flow between operands of the propositional connectives, snapback seems plausible for those. But consider a quantified formula $\forall x : K \in G \cdot Q$ where G is a singleton expression $\{p(F)\}$ with p returning a fresh reference; evaluation of G needs to be threaded to evaluation of Q, if the latter depends on fields of x.

Like for expressions, a verification system can explicitly thread updates through the semantics of the propositional connectives—but this comes at high cost. The specifier loses their ability to interpret specifications in terms of familiar mathematical and logical notions, and the verifier loses the ability to rely on shallow embedding of logical connectives to leverage automated theorem provers.

One reaction to these considerations is to seek restrictions on specifications such that there is no observable difference between threaded and snapback semantics. This is the approach taken by Naumann [46] and Darvas and Müller [25]. In our setting, an obvious restriction is to disallow in formulas explicit references to the ghost variable alloc. Further restrictions are needed to preclude, for example, calls of a pure method that returns the value of alloc. And that cannot be done in terms of the pure method's frame condition: any method that allocates also reads alloc, so rd alloc does not distinguish between a method that returns a new $Cell$ and one that returns the value of alloc.

Let us consider a formalization of weak purity in the present framework. In the semantics of frame conditions, effects are interpreted conjunctively: for example, see the definitions of *rlocs* and *wlocs*, and Def. 4.3. In terms of the function *rlocs*, snapback semantics would define $rlocs(\sigma, \theta, (\mathsf{rd}\ G`f, \mathsf{rd}\ H`g))$ as $rlocs(\sigma, \theta, \mathsf{rd}\ G`f) \cup rlocs(\sigma, \theta, \mathsf{rd}\ H`g)$ which among other things makes composition of effects symmetric and idempotent—which in turn justifies our abuse of notation, confusing lists and sets of effects. By contrast, threaded semantics for this example would be

$$rlocs(\sigma, \theta, (\mathsf{rd}\ G`f, \mathsf{rd}\ H`g)) = rlocs(\sigma, \theta, \mathsf{rd}\ G`f) \cup rlocs(\tau, \theta, \mathsf{rd}\ H`g)$$

where τ is the state after evaluating G in σ. So threaded semantics complicates all notions concerning effects.

Once threaded semantics is formalized, we can prove soundness of rules like Frame. As expected, a formula that calls a weakly pure method cannot be framed over code that allocates, just like formulas that explicitly mention alloc (recall equation (13)).

At the level of a judgment, say $\ldots \vdash C : P \leadsto Q\ [\varepsilon]$, there are several considerations about threading versus snapback. Definition 5.2 of valid judgment quantifies over all states σ that satisfy P. It refers to executions of C from σ, but a threaded version would execute C in states c' resulting from evaluation of P in σ. Moreover, the conditions for write effects and read effects involve evaluating expressions in the frame condition. Should effects in these too be threaded? If so, in which order should reads, writes, and precondition be evaluated? Finally, it is the well-formed judgments that are of real interest, which do not depend on the meaning of pure methods outside their preconditions—recall Definition 5.5 and the definedness formulas of Figure 5. Definedness conditions are also checked by verifiers that allow pure methods (see Section 12). Again there is not a single obviously right way to formulate threaded semantics of these conditions, or to connect their evaluation with the semantics of judgments.

In connection with proof rules, consider threaded interpretation of preconditions in the semantics of judgments. The proof rules Seq and While hinge on assertions to hold at intermediate points in computation. In a threaded interpretation of the premises of Seq, there is a state update between evaluation of the intermediate assertion and the second command. Yet execution of the sequenced commands involves no such update. This is one place where we would need commands and formulas to be insensitive to garbage collection. In verification tools and in [46] the analog is intermediate assert statements.

We investigated variations on threaded semantics of judgments, with the goal to characterize well-behaved specifications for which threaded and snapback semantics are equivalent. Among other things, we used the analysis of quasi-determinacy in Section A.1 (because the interpretation of a weakly pure method returns a set of states paired with values, and must do so quasi-deterministically). However, it is difficult to find a useful syntactic characterization because, as remarked above, a pure method may depend on alloc for differing reasons that are not distinguished by its frame condition. Moreover, stating a formal result requires tedious intricate definitions of threaded semantics that have little inherent interest.

One issue that emerges is the question what 'garbage insensitivity' should mean in the presence of ghost state. Though the syntax in this article does not explicitly designate ghost state, that can be added, and for purposes of the present discussion one can simply consider everything of type rgn to be ghost state. If we consider ghost variables including alloc like ordinary variables, there is no garbage to collect. An alternative is to define garbage collection in terms of what is reachable from non-ghost variables, and then update all rgn-typed locations by removing garbage references.

Perhaps the least obvious finding in our investigation is in connection with linking. Rule PureLink relies on the equality res $= m(x)$ to connect the mathematical interpretation $\psi(m)$ of pure method m with the result computed by its implementation, but that is untenable in case the result can be a fresh reference. Nor is there a way to express, at the level of the assertion language, that the allocations defined by the interpretation $\psi(m)$ coincide with those of the implementation. In the literature, the connection is made by restricting the implementation to simple expressions from which a mathematical denotation can be derived and formulated in terms amenable to SMT solvers. For a linking rule that allows arbitrary code restricted only by frame conditions with no writes other than wr alloc, we defined a denotational semantics of weakly pure methods (along the lines of Definition A.6 in the Appendix). We formalized, and proved sound using snapback semantics except for expressions, a linking rule in which $\psi(m)$ is required to be the denotation of the method body. Such reasoning is unsatisfactory because it violates the abstraction provided by the logic and its assertion language.

The investigation sketched in the preceding paragraphs suggests why it is so complicated to formulate and justify sufficient restrictions on programs and their specifications to achieve operationally sound reasoning, even for the sequential first-order programs considered in this article. No single approach has emerged in prior work as clearly the best, balancing simplicity of specifications, flexibility of programming, and usable proof rules or corresponding VCs. The obvious conclusion is that weak purity is not, after all, a useful special case of observational purity. Leino [38] reaches the same conclusion (using ‘pure’ to mean weakly pure): “pure methods are surprisingly complicated to get right. A major problem is that pure methods do have effects; for example, a pure method may allocate a hashtable that it uses during its computation…These problems make it tricky to provide the programming logic with the desirable illusion that pure methods are functions…I conclude with a slogan: pure methods are hard, functions are easy.”

12 RELATED WORK

Pure and weakly pure methods have been studied in the context of VC-gen for various verification systems such as ESC/Java2, Spec#, and Eiffel. We are not aware of verification tools that support observational purity. The formulation of pure methods in these studies is roughly like ours in Section 10: an uninterpreted function is given a pre-post axiom and a frame axiom. The formulation of VCs in these works does not make linking explicit.

Cok [22] explains the practical importance of pure methods for conciseness and abstraction in specifications, specifically in the Java Modeling Language (JML), and proposes to use uninterpreted functions to encode pure methods for VC-gen and automated theorem proving. Non-termination and exceptional termination are considered, as well as the issue of equality test for pure methods that return freshly allocated objects. The encodings cater for the ESC/Java2 system which does not explicitly model the heap as such: a specialized encoding is used to achieve an effect like threaded semantics.[25]

A number of earlier works point out the importance of read effects for pure methods. Frame axioms are explicitly featured in [40], [37], and [59].

[25]Cok [22] points out that pure methods essentially subsume the notion of model field used in JML and in other specification notations. Some related works address model fields and object invariants in connection with pure methods, but those topics are outside the scope of our formal development and we refrain from discussing them.

Darvas and Müller [25] introduce the term ‘weak purity’ for methods that are allowed to allocate and return fresh objects. They formulate VCs with threaded semantics, using two uninterpreted functions for a pure method: one for the result (constrained by what we call the pre-post axiom) and one for the updated state. They address what are called ‘recursive specifications’ in which the postcondition refers to the method itself, and point out the unsatisfiability of specifications like one that says $f(x)$ ensures res $= f(x) + 1$ (but recursive method bodies are not considered). The VCs effectively require existence of an interpretation. The VCs are shown to be consistent even in such cases and without restriction on the use of reference equality. That is, the assumed axioms are consistent; it is not shown that the VCs are sound with respect to operational semantics. Rudich et al. [54] add recursive method bodies, remove weak purity, and again show consistency of VCs.

Darvas et al. [24] investigate variations on definedness conditions in the presence of mutually recursive pure (but not weakly pure) methods, developing a formulation that avoids exponential blowup in formula size while remaining complete.

Darvas and Leino [23] explore the approach to weak purity in which the use of reference equality is restricted. Leino and Müller [40] develop this further: a specification can designate a ‘similarity relation’ to be used in formulas instead of equality, and VCs are formulated to check a relational property like our allowed dependency but requiring similarity of the result values. These and other works exploit ownership methodologies for framing. Leino and Müller [40] use self composition [13] for the relational property.

Leino and Müller [39] focus on the problem that specifications may be indirectly contradictory, as in $f(x)$ with postcondition res $= g(x)$ and $g(y)$ with postcondition res $= f(x) + 1$. They provide means to detect dependencies like this and to check satisfiability of specifications via witnesses, together with heuristics to find witness expressions that can be used by SMT solvers to prove satisfiabliity of the pre-post axioms in the presence of mutually recursive pure method bodies. They give a consistency theorem that relies on somewhat complicated well founded ordering as measure for method calls, which is less restrictive than the syntactic restrictions in [23, 25, 54].

To cater for deriving an interpretation from the implementation, works like [25] and [59] restrict method bodies to a single return expression. In Section 9, we show why this restriction is not necessary. We also show the role of measures for recursion: a measure is needed to prove definedness and correctness of interpretations derived from pure method bodies. For this purpose, measure constraints can be added to specifications, but need not be added to specifications used by clients.

As noted in Section 1.3, contradictory postconditions of pure methods are just one of several ways in which unsatisfiable specifications arise. Inconsistencies like the interdependent f and g above are more likely than others to be exploited by an SMT solver if used in faulty axioms. Reasoning under the assumption of unsatisfiable specifications is harmless, in that a component’s verification cannot be completed if it relies on methods with unsatisfiable specifications. In our view, checks for unsatisfiable specifications, like ‘smoke tests’ for inconsistent assumptions of other kinds, are important practical tools, perhaps especially important in the presence of inheritance and dynamic dispatch or other features that result in unwieldy call graphs. But such checks should not be considered part of the VCs that serve to establish correctness of programs.

We take the Cell example from the most closely related work [59] where read effects of pure methods are specified using a form of dynamic frames, and methods may be self-framing. They define (and implement) a VC-generator including VCs that encode the agreement semantics of read effects, albeit only for a pair of states in succession. That avoids the need for refperms, and suffices for framing (see remark following Definition 6.4); but not for relational reasoning for data abstraction and encapsulation. Unlike any of the works cited above, which only prove consistency of axioms, 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 not recursive). The expression E is visible to clients, as in our rule TranspPureLink and in several of the cited works. (Their 'implementation axiom' is what we call a definition axiom in Section 10.) Pure methods do not have postconditions in the formal development, but 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.

Our use of explicit ghost state in read and write effects is directly inspired by dynamic frames, the state-dependent frame conditions introduced by Kassios [33, 34]. Kassios' frames predicate f **frames** v (where f is an expression that denotes a location set) says that the current value of expression 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' 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, that is, 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 this article our frames are expressed in terms of reference sets (regions) and effect expressions rather than location sets. More significantly, by contrast with Kassios' work we can use recursively defined and parameterized pure methods, implemented by commands, to express the sets. Kassios' frames predicate and its properties are embodied, in our work, in the frames judgment as well as the notion of immunity (Lemma 6.9). Apropos the connection between preservation of disjointness and effects with our notion of framed reads, note first that in this article there is no explicit notation for freshness. By contrast, Smans et al. use a special predicate to express freshness in postconditions, RLI/II expresses freshness in frame conditions using an effect notation, and Kassios can directly express freshness because his notations explicitly describe relations between initial and final states. In this article we dropped the freshness effect from RL and instead express freshness in specifications and judgments where the precondition 'snapshots' the initial value of alloc and the postcondition asserts disjointness from that snapshot. (See, e.g., the discussion of derived rule Alloc1 and its use in the example proofs.) Consider the following specification, using some region expressions G, H:

$$G = \mathsf{alloc} \land \mathit{ftpt}(G) \;\cdot\!/\!.\; \eta \;\rightsquigarrow\; H \mathbin{\#} G \;\; [\eta]$$

If the specifications of methods called in G have framed reads then $\mathit{ftpt}(G)$ has framed reads. In this case the initial separation $\mathit{ftpt}(G) \;\cdot\!/\!.\; \eta$ ensures that the final value of G is the same as its initial value—which entails that references in H (in the final state) are fresh. Note that $H \mathbin{\#} G$ may imply $\mathit{ftpt}(G) \;\cdot\!/\!.\; \eta$ or some other disjointness of frames, as in Kassios' principle described above. The significance of freshness is that it can be used to establish disjointnesses, so it seems natural to express freshness in terms of disjointness.

The Dafny verifier features dynamic frames in a form much like that of RL, and includes pure (but not weakly pure) methods [38]. It is under active development and has been used in large verification efforts such as [30]. A number of other contract languages also permit the use of pure methods, including Eiffel, JML, Spec# [11], Chalice [41], VeriCool [57], and Viper [43].

In Section 1.3 we mention a couple of approaches to the semantics of the hypothetical correctness judgment form (1). Yet another approach goes by using nondeterminacy to represent a single 'worst implementation' of each method, akin to the 'specification statement' used in axiomatic semantics.

This approach facilitates proof of second-order frame rules, and is used for that purpose by O'Hearn et al. [47] and in RLII [5]. It is also used in the conference version of this article [6]. For read effects it requires use of a possibilistic property in $\forall\exists$ form: For all σ, σ' and all states τ reached by C from σ, there exists τ' reached by C from σ' such that τ agrees with τ'. Unfortunately, this seems incompatible with linking. 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. Another significant flaw in the conference version is in the semantics of read effects. By contrast with Def. 4.4 of allowed dependence, where it requires agreement on locations actually written, the flawed version requires agreements on writable locations. Because not all writables get changed, it turns out to be much easier to work with the current definition. Nevertheless, a reconciliation with the $\forall\exists$ semantics remains an open problem.

Framing in separation logic encompasses read and write effects, implicitly in syntax but explicitly in the semantics, *cf.* the conditions 'safety monotonicity' and 'frame property' in O'Hearn et al. [47] where the second-order frame rule is introduced. Permission-based separation logic allows distinguishing read effects from write effects [19]. Parkinson and Summers [50] explore the connection between this logic and one based on implicit dynamic frames (IDF) [58]. 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, 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, like us, uses a syntactic definition of self-framing. In this article we are not addressing concurrency. The GRASShopper tool [51] does not feature pure methods but it provides IDF-like notation in specifications. The VCs are generated using an encoding with explicit ghost state for procedure footprints as reference sets; application of the frame rule is encoded in terms of these footprints and a global alloc variable. The VCs are decidable, for specifications based on lists or similar predicates.

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. The abstract predicates approach to data abstraction [28, 29, 44, 49] has inspired several works that cater for SMT solvers 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 [31]. Another gives a detailed semantics and soundness proof for VCs that provide effective reasoning about recursively defined abstract predicates and abstraction functions [60].

Parkinson and Bierman [48] formalize abstract predicates in connection with inheritance, using statically dispatched versus dynamic handling of predicates, but do not consider pure methods. Inheritance and dynamic dispatch are not addressed in most of the prior literature on pure methods. Smans et al. [59] does provide detailed rules for the encoding of inheritance, adapted from Parkinson and Bierman [48], but it is not included in their formalization or soundness proof. The encoding addresses dynamic binding but requires all virtual methods to be overriden in every class, effectively requiring re-verification when inheritance is used. For modular verification it is well understood that inheritance should usually conform with behavioral subtyping, which requires the specification of a method at a subtype to refine its specification at a supertype [36]. In our setting that means refinement for specifications that include read effects. This topic is left for future work.

Bao [9] investigates behavioral subtyping with explicit frame conditions, in the setting of a fine-grained version of region logic [10] to which we return later. Bao extends the theory of [36] to account for frame conditions and encapsulation, and shows modular soundness for a logic with supertype abstraction. Bao's thesis [9] also presents a unified fine-grained region logic that features read effects in the style of separation logic, i.e., as permission to access locations. This is used to establish connections between fine-grained region logic and separation logic.

Relational semantics for effects is explored by Benton et al. [17]. 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. [16] consider a denotational semantics of a region-based type and effect system that supports observational purity. 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.

Schmitt et al. [56] define and implement rules for a form of read effect for pure methods. The setting is the KeY tool which is based on dynamic logic [14]. Dynamic frames are expressed in terms of location sets and the assertion language features an explicit representation of the heap.

Bao et al. [10] 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 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 cannot express 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. [59]. Explicit use of fields in RL provides simple syntactic means to establish many disjointnesses. Arguably, data groups [42] are a sufficient means of abstracting from field names. The significance of these expressive differences is perhaps best evaluated in connection with abstraction and information hiding.

13 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. 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 a logic for specification and verification using observationally pure methods. Another intended use of the logic is to guide the design and validation of semi-automated verification tools based on SMT solvers and verification condition generators. Modular verification tools implement, in effect, linking rules, though this is not usually explicit in the literature on VC-gen. By contrast with linking rules, other proof rules are not directly implemented, but rather guide the design of VC-gen, 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 in method specifications should have framed reads. (In the logic, however, we cannot require all judgments to have framed reads, in light of the discussion of rules Seq and While.) Instead of imposing framed reads as a restriction on specifications, for practical purposes it can be left implicit, as it is a syntactic closure that can be applied automatically.

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 correctness judgment quantifies over all pairs of runs, and in part owing to restriction to 'framed reads' which ensures symmetry where it is needed. Relational Hoare logics have been developed to reason about dependency properties [2, 15, 61]. In ongoing work [7] we are developing a relational version of the logic. Read effects of commands play a crucial role in that logic.

ACKNOWLEDGMENTS

The authors thank the anonymous referees for numerous apposite suggestions. Banerjee was partially supported by the US National Science Foundation (NSF). Naumann was partially supported by NSF awards CNS-1228930, CCF-1649894, and CNS-1718713. Nikouei was partially supported by NSF awards CNS-1228930 and CCF-1649894. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

REFERENCES

[1] Jonathan Aldrich, Mike Barnett, Dimitra Giannakopoulou, Gary T. Leavens, Natasha Sharygina, and Robby. 2008. *Seventh International Workshop on Specification and Verification of Component Systems (SAVCBS).* Technical Report CS-TR-08-07. School of Electrical Engineering and Computer Science, University of Central Florida.

[2] T. Amtoft, S. Bandhakavi, and A. Banerjee. 2006. A Logic for Information Flow in Object-Oriented Programs. In *ACM Symposium on Principles of Programming Languages*. 91–102. https://doi.org/10.1145/1111037.1111046

[3] Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. 2009. *Verification of Sequential and Concurrent Programs* (3 ed.). Springer. https://doi.org/10.1007/978-1-84882-745-5

[4] Anindya Banerjee and David A. Naumann. 2005. Ownership Confinement Ensures Representation Independence for Object-Oriented Programs. *J. ACM* 52, 6 (2005), 894–960. https://doi.org/10.1145/1101821.1101824

[5] Anindya Banerjee and David A. Naumann. 2013. Local Reasoning for Global Invariants, Part II: Dynamic Boundaries. *J. ACM* 60, 3 (2013), 19:1–19:73. https://doi.org/10.1145/2485981

[6] Anindya Banerjee and David A. Naumann. 2014. A Logical Analysis of Framing for Specifications with Pure Method Calls. In *Verified Software: Theories, Tools, Experiments (Lecture Notes in Computer Science)*, Vol. 8471. 3–20. https://doi.org/10.1007/978-3-319-12154-3_1

[7] Anindya Banerjee, David A. Naumann, and Mohammad Nikouei. 2016. Relational Logic with Framing and Hypotheses. In *36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (LIPIcs)*, Vol. 65. 11:1–11:16. https://doi.org/10.4230/LIPIcs.FSTTCS.2016.11

[8] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. 2013. Local Reasoning for Global Invariants, Part I: Region Logic. *J. ACM* 60, 3 (2013), 18:1–18:56. https://doi.org/10.1145/2485982

[9] Yuyan Bao. 2017. *Reasoning About Frame Properties in Object-Oriented Programs.* Technical Report CS-TR-17-05. University of Central Florida. www.cs.ucf.edu/~leavens/tech-reports/UCF/CS-TR-17-05/TR.pdf

[10] Yuyan Bao, Gary T. Leavens, and Gidon Ernst. 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. https://doi.org/10.1145/2786536.2786537

[11] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. 2011. Specification and verification: the Spec# experience. *Commun. ACM* 54, 6 (2011), 81–91. https://doi.org/10.1145/1953122.1953145

[12] Mike Barnett, David A. Naumann, Wolfram Schulte, and Qi Sun. 2004. 99.44% pure: Useful Abstractions in Specifications. In *ECOOP workshop on Formal Techniques for Java-like Programs (FTfJP).* Technical Report NIII-R0426, University of Nijmegen.

[13] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2004. Secure Information Flow by Self-Composition. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW'04).* 100–114. https://doi.org/10.1109/CSFW.2004.17

[14] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt (Eds.). 2007. *Verification of Object-Oriented Software: The KeY Approach*. Lecture Notes in Computer Science, Vol. 4334. Springer-Verlag. https://doi.org/10.1007/978-3-540-69061-0

[15] N. Benton. 2004. Simple Relational Correctness Proofs for Static Analyses and Program Transformations. In *ACM Symposium on Principles of Programming Languages*. 14–25. https://doi.org/10.1145/964001.964003

[16] Nick Benton, Martin Hofmann, and Vivek Nigam. 2014. Abstract effects and proof-relevant logical relations. In *ACM Symposium on Principles of Programming Languages*. 619–632. https://doi.org/10.1145/2535838.2535869

[17] Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. 2007. Relational semantics for effect-based program transformations with dynamic allocation. In *International Symposium on Principles and Practice of Declarative Programming*. 87–96. https://doi.org/10.1145/1273920.1273932

[18] François Bobot and Jean-Christophe Filliâtre. 2012. Separation Predicates: A Taste of Separation Logic in First-Order Logic. In *International Conference on Formal Engineering Methods (LNCS)*, Vol. 7635. 167–181. https://doi.org/10.1007/978-3-642-34281-3_14

[19] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. 2005. Permission Accounting in Separation Logic. In *ACM Symposium on Principles of Programming Languages*. 259–270. https://doi.org/10.1145/1040305.1040327

[20] Aaron R. Bradley and Zohar Manna. 2007. *The Calculus of Computation – Decision Procedures with Applications to Verification*. Springer. https://doi.org/10.1007/978-3-540-74113-8

[21] David Cok and Gary T. Leavens. 2008. Extensions of the theory of observational purity and a practical design for JML. In *Proceedings of the 7th International Workshop on Specification and Verification of Component-Based Systems*. 43–50. Technical Report CS-TR-08-07, School of EECS, University of Central Florida.

[22] David R. Cok. 2005. Reasoning with Specifications Containing Method Calls and Model Fields. *Journal of Object Technology* 4, 8 (2005), 77–103. https://doi.org/10.5381/jot.2005.4.8.a4

[23] Ádám Darvas and K. Rustan M. Leino. 2007. Practical Reasoning About Invocations and Implementations of Pure Methods. In *Fundamental Approaches to Software Engineering (Lecture Notes in Computer Science)*, Vol. 4422. 336–351. https://doi.org/10.1007/978-3-540-71289-3_26

[24] Ádám Darvas, Farhad Mehta, and Arsenii Rudich. 2008. Efficient Well-Definedness Checking. In *Automated Reasoning, 4th International Joint Conference (Lecture Notes in Computer Science)*, Vol. 5195. 100–115. https://doi.org/10.1007/978-3-540-71070-7_8

[25] Á. Darvas and P. Müller. 2006. Reasoning about method calls in interface specifications. *Journal of Object Technology (JOT)* 5, 5 (June 2006), 59–85. https://doi.org/10.5381/jot.2006.5.5.a3

[26] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. 2016. The Spirit of Ghost Code. *Formal Methods in System Design* 48, 3 (2016), 152–174. https://doi.org/10.1007/s10703-016-0243-x

[27] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. 2016. A Pragmatic Type System for Deductive Verification. (Feb. 2016). https://hal.inria.fr/hal-01256434 working paper or preprint.

[28] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended Static Checking for Java. In *ACM Conf. on Program. Lang. Design and Implementation*. 234–245. https://doi.org/10.1145/512529.512558

[29] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew J. Parkinson. 2012. Behavioral interface specification languages. *ACM Comput. Surv.* 44, 3 (2012), 16:1–16:58. https://doi.org/10.1145/2187671.2187678

[30] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *11th USENIX Symposium on Operating Systems Design and Implementation, (OSDI)*. 165–181.

[31] Stefan Heule, Ioannis T. Kassios, Peter Müller, and Alexander J. Summers. 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. https://doi.org/10.1007/978-3-642-39038-8_19

[32] C. A. R. Hoare. 1972. Proofs of Correctness of Data Representations. *Acta Informatica* 1 (1972), 271–281. https://doi.org/10.1007/BF00289507

[33] Ioannis T. Kassios. 2006. Dynamic Frames: Support for Framing, Dependencies and Sharing without Restrictions. In *Formal Methods (Lecture Notes in Computer Science)*, Vol. 4085. 268–283. https://doi.org/10.1007/11813040_19

[34] Ioannis T. Kassios. 2011. The dynamic frames theory. *Formal Aspects of Computing* 23, 3 (2011), 267–288. https://doi.org/10.1007/s00165-010-0152-5

[35] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. 2006. Preliminary design of JML: a behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes* 31, 3 (2006), 1–38. https://doi.org/10.1145/1127878.1127884

[36] Gary T. Leavens and David A. Naumann. 2015. Behavioral Subtyping, Specification Inheritance, and Modular Reasoning. *ACM Transactions on Programming Languages and Systems* 37, 4 (2015), 13:1–13:88. https://doi.org/10.1145/2766446

[37] K. Rustan M. Leino. 2008. Specification and Verification in Object-oriented Software. (2008). Marktoberdorf lecture notes.

[38] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

[39] K. Rustan M. Leino and Ronald Middelkoop. 2009. Proving Consistency of Pure Methods and Model Fields. In *Fundamental Aspects to Software Engineering*. 231–245. https://doi.org/10.1007/978-3-642-00593-0_16

[40] K. Rustan M. Leino and Peter Müller. 2008. Verification of Equivalent-Results Methods. In *ESOP*. 307–321. https://doi.org/10.1007/978-3-540-78739-6_24

[41] K. Rustan M. Leino and Peter Müller. 2009. A Basis for Verifying Multi-threaded Programs. In *Programming Languages and Systems, European Symposium on Programming*. 378–393. https://doi.org/10.1007/978-3-642-00590-9_27

[42] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. 2002. Using Data Groups to Specify and Check Side Effects. In *ACM Conf. on Program. Lang. Design and Implementation*. 246–257. https://doi.org/10.1145/512529.512559

[43] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation (Lecture Notes in Computer Science)*, Vol. 9583. 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

[44] Aleksandar Nanevski, Amal Ahmed, Greg Morrisett, and Lars Birkedal. 2007. Abstract Predicates and Mutable ADTs in Hoare Type Theory. In *Programming Languages and Systems, European Symposium on Programming (Lecture Notes in Computer Science)*, Vol. 4421. 189–204. https://doi.org/10.1007/978-3-540-71316-6_14

[45] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. 2013. Dependent Type Theory for Verification of Information Flow and Access Control Policies. *ACM Trans. Program. Lang. Syst.* 35, 2 (2013), 6. https://doi.org/10.1145/2491522.2491523

[46] David A. Naumann. 2007. Observational Purity and Encapsulation. *Theoretical Computer Science* 376, 3 (2007), 205–224. https://doi.org/10.1016/j.tcs.2007.02.004

[47] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. 2009. Separation and Information Hiding. *ACM Transactions on Programming Languages and Systems* 31, 3 (2009), 1–50. https://doi.org/10.1145/964001.964024

[48] Matthew Parkinson and Gavin Bierman. 2008. Separation Logic, Abstraction and Inheritance. In *ACM Symposium on Principles of Programming Languages*. 75–86. https://doi.org/10.1145/1328438.1328451

[49] Matthew J. Parkinson and Gavin M. Bierman. 2005. Separation Logic and Abstraction. In *ACM Symposium on Principles of Programming Languages*. 247–258. https://doi.org/10.1145/1040305.1040326

[50] Matthew J. Parkinson and Alexander J. Summers. 2012. The Relationship Between Separation Logic and Implicit Dynamic Frames. *Logical Methods in Computer Science* 8, 3 (2012). https://doi.org/10.2168/LMCS-8(3:1)2012

[51] Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014. Grasshopper. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science)*, Vol. 8413. 124–139.

[52] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE Symp. on Logic in Computer Science*. 55–74. https://doi.org/10.1109/LICS.2002.1029817

[53] Stan Rosenberg, Anindya Banerjee, and David A. Naumann. 2010. Local Reasoning and Dynamic Framing for the Composite Pattern and Its Clients. In *Verified Software: Theories, Tools, Experiments (Lecture Notes in Computer Science)*, Vol. 6217. 183–198. https://doi.org/10.1007/978-3-642-15057-9_13 Software distribution at http://www.cs.stevens.edu/~naumann/pub/VERL/.

[54] Arsenii Rudich, Ádám Darvas, and Peter Müller. 2008. Checking Well-Formedness of Pure-Method Specifications. In *Formal Methods*. 68–83. https://doi.org/10.1007/978-3-540-68237-0_7

[55] Alexandru Salcianu and Martin C. Rinard. 2005. Purity and Side Effect Analysis for Java Programs. In *Verification, Model Checking, and Abstract Interpretation (Lecture Notes in Computer Science)*, Radhia Cousot (Ed.), Vol. 3385. 199–215. https://doi.org/10.1007/978-3-540-30579-8_14

[56] Peter H. Schmitt, Mattias Ulbrich, and Benjamin Weiß. 2011. Dynamic Frames in Java Dynamic Logic. In *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers (Lecture Notes in Computer Science)*, Vol. 6528. 138–152. https://doi.org/10.1007/978-3-642-18070-5_10

[57] Jan Smans, Bart Jacobs, and Frank Piessens. 2008. VeriCool: An Automatic Verifier for a Concurrent Object-Oriented Language. In *Formal Methods for Open Object-Based Distributed Systems (Lecture Notes in Computer Science)*, Vol. 5051. Springer, 220–239. https://doi.org/10.1007/978-3-540-68863-1_14

[58] Jan Smans, Bart Jacobs, and Frank Piessens. 2012. Implicit dynamic frames. *ACM Transactions on Programming Languages and Systems* 34, 1 (2012), 2. https://doi.org/10.1145/2160910.2160911

[59] Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. 2010. Automatic Verification of Java Programs with Dynamic Frames. *Formal Aspects of Computing* 22, 3-4 (2010), 423–457. https://doi.org/10.1007/s00165-010-0148-1

[60] Alexander J. Summers and Sophia Drossopoulou. 2013. A Formal Semantics for Isorecursive and Equirecursive State Abstractions. In *European Conference on Object-Oriented Programming (LNCS)*, Vol. 7920. 129–153. https://doi.org/10.1007/978-3-642-39038-8_6

[61] Hongseok Yang. 2007. Relational Separation Logic. *Theoretical Computer Science* 375, 1-3 (2007), 308–334. https://doi.org/10.1016/j.tcs.2006.12.036

A ADDITIONAL SOUNDNESS PROOFS

A.1 Definitions and results needed to prove soundness of ImpureLink

Were it not for our aim to be compatibly with the small step encapsulation property of RLII, we would define correctness in terms of a denotational semantics derived from the transition semantics. We do in fact need such a semantics for use in proving ImpureLink. The proof follows the lines sketched at the beginning of Section 8.2, and in some ways the proof of PureLink in Section 8.3. But for PureLink we are given a context interpretation for the method to be linked, as needed to appeal to the premises of the rule. For ImpureLink we use the denotational semantics to construct the interpretation needed to appeal to the premises.

Quasi-determinacy. Formalization of the denotational semantics relies on a kind of determinacy mentioned in Sec. 1 but not explicit in the body of the paper.

Definition A.1 ($\overset{\pi}{\approx}$, $\approxeq_\pi$, quasi-determinacy). Fix Γ. For Γ-states σ, σ', define $\sigma \overset{\pi}{\approx} \sigma'$ to mean that π is a total bijection from $\sigma(\mathsf{alloc})$ to $\sigma'(\mathsf{alloc})$ and the states agree modulo π on all variables and all fields of all objects. That is, $\mathit{Lagree}(\sigma, \sigma', \pi, \mathit{locations}(\sigma))$, which under these conditions is equivalent to $\mathit{Lagree}(\sigma', \sigma, \pi^{-1}, \mathit{locations}(\sigma'))$.

For outcome sets S and S', *i.e.*, $S, S' \in \mathbb{P}(\llbracket\Gamma\rrbracket \cup \{\lightning\})$, and any partial bijection π on references, define $S \approxeq_\pi S'$ (read ***equivalence modulo*** π) to mean that

(i) $\lightning \in S$ iff $\lightning \in S'$,
(ii) for all states $\sigma \in S$ and $\sigma' \in S'$ there is $\rho \supseteq \pi$ such that $\sigma \overset{\rho}{\approx} \sigma'$, and
(iii) $S \setminus \{\lightning\} = \varnothing$ iff $S' \setminus \{\lightning\} = \varnothing$.

A candidate interpretation φ is ***quasi-deterministic*** if

- For every pure m, $\varphi(m)$ is quasi-deterministic in the following sense: if $\sigma \overset{\pi}{\approx} \sigma'$ and $v \overset{\pi}{\sim} v'$ then $\varphi(m)(\sigma, v) \overset{\pi}{\sim} \varphi(m)(\sigma', v')$ where we lift $\overset{\pi}{\sim}$ to relate $\lightning$ only to itself.
- For every impure m, $\varphi(m)$ is quasi-deterministic in the following sense:
  - $\lightning \in \varphi(m)(\sigma, v)$ iff $\varphi(m)(\sigma, v) = \{\lightning\}$ (fault determinacy)
  - $\sigma \overset{\pi}{\approx} \sigma'$ and $v \overset{\pi}{\sim} v'$ imply $\varphi(m)(\sigma, v) \approxeq_\pi \varphi(m)(\sigma', v')$ (state determinacy)

We refer to $\overset{\pi}{\approx}$ as ***state isomorphism*** modulo π. Note that item (ii) in the definition of $\approxeq_\pi$ involves extensions of π, whereas the relations $\overset{\pi}{\sim}$ and $\overset{\pi}{\approx}$ involve only π itself. Instantiating $\sigma' := \sigma$, $v' := v$, and π to be identity on $\sigma(\mathsf{alloc})$ in the condition (state determinacy) yields that all results from a given input are isomorphic.[26]

Note that for impure m, $\sigma \overset{\pi}{\approx} \sigma'$ and $v \overset{\pi}{\sim} v'$ do not imply that $\varphi(m)(\sigma, v)$ and $\varphi(m)(\sigma', v')$ have the same cardinality; but item (iii) in the definition of $\approxeq_\pi$ ensures that one contains states iff the other does.

The following result says that expressions and formulas respect isomorphism of states, for candidate interpretations where the relevant methods are quasi-deterministic.

Lemma A.2. Suppose $\sigma \overset{\pi}{\approx} \sigma'$. Then $\llbracket F\rrbracket_\varphi \sigma \overset{\pi}{\sim} \llbracket F\rrbracket_\varphi \sigma'$ for any F such that $\varphi(m)$ is quasi-deterministic for every m that occurs in F. Moreover $\sigma \models_\varphi P$ iff $\sigma' \models_\varphi P$ for any P such that $\varphi(m)$ is quasi-deterministic for every m that occurs in P.

The proof is straightforward, by induction on F and induction on P.

[26]In light of these definitions and the results to follow, we could as well replace the codomain of a impure method interpretation, *i.e.*, $\mathbb{P}(\llbracket\Gamma\rrbracket \cup \{\lightning\})$, by $\mathbb{P}(\llbracket\Gamma\rrbracket) \cup \{\lightning\}$. The chosen formulation helps slightly streamline a few things later.

Lemma A.3 (quasi determinacy of transitions). Let φ be a quasi-deterministic candidate interpretation. Then

(a) The transition relation $\overset{\varphi}{\longmapsto}$ is rule-deterministic in the sense that for every configuration $\langle C,\ \sigma,\ \mu\rangle$ there is at most one applicable transition rule.

(b) No configuration has both $\lightning$ and a non-fault successor. That is, if $\langle C,\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \langle B,\ \tau,\ \nu\rangle$ then it is not the case that $\langle C,\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \lightning$; and if $\langle C,\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \lightning$ then $\langle C,\ \sigma,\ \mu\rangle$ has no other successor.

(c) If $\sigma \overset{\pi}{\approx} \sigma'$ and $\langle C,\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \langle D,\ \tau,\ \nu\rangle$ and $\langle C,\ \sigma',\ \mu\rangle \overset{\varphi}{\longmapsto} \langle D',\ \tau',\ \nu'\rangle$ then $D = D'$, $\nu = \nu'$, and $\tau \overset{\rho}{\approx} \tau'$ for some $\rho \supseteq \pi$.

(d) If $\sigma \overset{\pi}{\approx} \sigma'$ then $\langle C,\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \lightning$ iff $\langle C,\ \sigma',\ \mu\rangle \overset{\varphi}{\longmapsto} \lightning$.

(e) For all i, if $\sigma \overset{\pi}{\approx} \sigma'$ and $\langle C,\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto}{}^{i} \langle D,\ \tau,\ \nu\rangle$ and $\langle C,\ \sigma',\ \mu\rangle \overset{\varphi}{\longmapsto}{}^{i} \langle D',\ \tau',\ \nu'\rangle$ then $D = D'$, $\nu = \nu'$, and $\tau \overset{\rho}{\approx} \tau'$ for some $\rho \supseteq \pi$.

(f) If $\sigma \overset{\pi}{\approx} \sigma'$ then $\langle C,\ \sigma,\ \mu\rangle$ is a stuck context call iff $\langle C,\ \sigma',\ \mu\rangle$ is.

(g) If $\sigma \overset{\pi}{\approx} \sigma'$ and $\langle C,\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \langle D,\ \tau,\ \nu\rangle$ then $\langle C,\ \sigma',\ \mu\rangle \overset{\varphi}{\longmapsto} \langle D,\ \tau',\ \nu\rangle$ and $\tau \overset{\rho}{\approx} \tau'$, for some τ and some $\rho \supseteq \pi$.

Apropos (f), it is only impure methods that can be stuck.

Proof. (a) Consider any $\langle C,\ \sigma,\ \mu\rangle$. There is no transition in case $C \equiv$ skip. There is no transition in case *Active*(C) is a context call for an impure method m with argument value v and $\varphi(m)(\sigma, v) = \emptyset$. Otherwise, by cases on *Active*(C) and inspection of the rules, there is exactly one rule applicable. In the case of context call of an impure method, this relies on Def. A.1(fault determinacy).

(b) Fault and non-fault outcomes are given by different rules, so this follows from (a).

(c) By cases on *Active*(C). For any command other than context call or allocation, take $\rho = \pi$ and inspect the transition rules. For example, $x.f := y$ changes the state by updating a field with values that are in agreement modulo π. For the case of $x := E$ we need that $[\![E]\!]$ respects isomorphism of states, Lemma A.2. For allocation, let $\rho = \{(o, o')\} \cup \pi$ where o, o' are the allocated objects. For context call, both pure and impure, we get the result by quasi-determinacy.

(d) Similar to the proof of (c), using in particular item (i) in the definition of $\cong_\pi$.

(e) By straightforward induction on i, using (c).

(f) By Def. A.1(state determinacy), and using item (iii) in the definition of $\cong_\pi$.

(g) Follows from (d), (e), and (f). $\square$

Lemma A.4. Context interpretations are quasi-deterministic.

Proof. Let Φ be a well formed context and let φ be a Φ-interpretation.

First we show that $\varphi(m)$ is quasi-deterministic for every pure m. The argument goes by induction on the ordering $\prec_\Phi$ used in Def. 2.2. For given m, suppose $\sigma \overset{\pi}{\approx} \sigma'$ and $v \overset{\pi}{\sim} v'$. We must show $\varphi(m)(\sigma, v) \overset{\pi}{\sim} \varphi(m)(\sigma', v')$. Let P be the precondition of m and note that any methods called in P are quasi-deterministic, by induction. So $\sigma \models_\varphi P$ iff $\sigma' \models_\varphi P$ by Lemma A.2. Thus $\varphi(m)(\sigma, v) = \lightning$ iff $\varphi(m)(\sigma', v') = \lightning$ by Def. 5.1(a). In the non-fault case we get $\varphi(m)(\sigma, v) \overset{\pi}{\sim} \varphi(m)(\sigma', v')$ from the read effect, Def. 5.1(c).

For any impure m, we get fault determinacy directly from Def. 5.1(d). To show state determinacy, suppose $\sigma \overset{\pi}{\approx} \sigma'$ and $v \overset{\pi}{\sim} v'$. We must show $\varphi(m)(\sigma, v) \cong_\pi \varphi(m)(\sigma', v')$. To that end, let R be the precondition of m. Let $\dot\sigma \mathrel{\widehat{=}} [\sigma + x\colon v]$ and $\dot\sigma' \mathrel{\widehat{=}} [\sigma' + x\colon v']$. Thus we have $\dot\sigma \overset{\pi}{\approx} \dot\sigma'$. By Lemma A.2 and the quasi-determinacy of pure method interpretations (proved above), we get $\dot\sigma \models_\varphi R$ iff $\dot\sigma' \models_\varphi R$. So $\varphi(m)(\sigma, v) = \{\lightning\}$ iff $\varphi(m)(\sigma', v') = \{\lightning\}$ so in the faulting case we have $\varphi(m)(\sigma, v) \cong_\pi \varphi(m)(\sigma', v')$. For the non-faulting case, suppose $\dot\sigma \models_\varphi R$ and $\dot\sigma' \models_\varphi R$. Consider any $\tau \in \varphi(m)(\sigma, v)$ and $\tau' \in \varphi(m)(\sigma', v')$. We must find $\rho \supseteq \pi$ such that $\tau \overset{\rho}{\approx} \tau'$. By read effect, Def. 5.1(f), we have $\dot\sigma, \dot\sigma' \Rightarrow \tau, \tau' \models_\varphi \varepsilon, \mathsf{rd}\, x$ and $\dot\sigma', \dot\sigma \Rightarrow \tau', \tau \models_\varphi \varepsilon, \mathsf{rd}\, x$. From $\dot\sigma \overset{\pi}{\approx} \dot\sigma'$, we have $Agree(\dot\sigma, \dot\sigma', (\varepsilon, \mathsf{rd}\, x), \pi, \varphi)$ and $Agree(\dot\sigma', \dot\sigma, (\varepsilon, \mathsf{rd}\, x), \pi^{-1}, \varphi)$. By $\dot\sigma, \dot\sigma' \Rightarrow \tau, \tau' \models_\varphi \varepsilon, \mathsf{rd}\, x$, there exists $\rho \supseteq \pi$ such that $Lagree(\tau, \tau', \rho, freshLocs(\dot\sigma, \tau) \cup written(\dot\sigma, \tau))$. Because π is total on the initially allocated locations (according to $\dot\sigma \overset{\pi}{\approx} \dot\sigma'$), we get that ρ is total on all the locations of τ, τ'. To complete the argument for $\tau \overset{\rho}{\approx} \tau'$, it remains to get agreement on W where $W \mathrel{\widehat{=}} locations(\tau) \setminus (freshLocs(\dot\sigma, \tau) \cup written(\dot\sigma, \tau))$. Note that $W \subseteq locations(\dot\sigma)$. So from $\dot\sigma \overset{\pi}{\approx} \dot\sigma'$ we have $Lagree(\dot\sigma, \dot\sigma', \pi, W)$. Now we can use Lemma 6.12 to get $Lagree(\tau, \tau', \rho, W)$. □

Denotational semantics. We use the outcome $\lightning$ to represent runtime faults (null dereference) and also the invocation of a context method outside its specified precondition (*i.e.*, failure of modular correctness). For purposes of approximation, we can use the empty set of outcomes to represent undefinedness.

Definition A.5 (approximation, $\sqsubseteq$). Define the approximation ordering $\sqsubseteq$ on outcome sets, *i.e.*, on $\mathbb{P}(\llbracket\Gamma\rrbracket \cup \{\lightning\})$ by

$$S \sqsubseteq S' \quad \text{iff} \quad S = \varnothing \text{ or } S = S' \tag{42}$$

Functions f, f' of type $\llbracket\Gamma\rrbracket \to \mathbb{P}(\llbracket\Gamma\rrbracket \cup \{\lightning\})$ are ordered pointwise: $f \sqsubseteq f'$ iff $f(\sigma) \sqsubseteq f'(\sigma)$ for all σ. Similarly for method interpretations: for f, f' of type $(\sigma \in \llbracket\Gamma\rrbracket) \times \llbracket T \rrbracket\sigma \to \mathbb{P}(\llbracket\Gamma\rrbracket \cup \{\lightning\})$ let $f \sqsubseteq f'$ iff $f(\sigma, v) \sqsubseteq f'(\sigma, v)$ for all σ, v. This is used to define the ordering on candidate interpretations of a given method context Φ, *i.e.*, $\theta \sqsubseteq \theta'$ iff $\theta(m) \sqsubseteq \theta'(m)$ for all $m \in dom(\Phi)$.

Definition A.6 (denotation of command). Suppose C is swf in Γ and θ is a candidate interpretation for Γ. Define $\llbracket \Gamma \vdash C \rrbracket_\theta$ to be the function of type $\llbracket\Gamma\rrbracket \to \mathbb{P}(\llbracket\Gamma\rrbracket \cup \{\lightning\})$ defined by

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

where _ is the empty method environment.

Lemma A.7 (denotation monotonic). Let θ, φ be candidate interpretations for Γ that are quasi-deterministic. If $\theta \sqsubseteq \varphi$ then $\llbracket \Gamma \vdash C \rrbracket_\theta \sqsubseteq \llbracket \Gamma \vdash C \rrbracket_\varphi$ for any C.

One way to prove this is to first connect the trace-based denotational semantics (Def. A.6) with one defined compositionally on program syntax (*e.g.*, see Lemma 3.3 in [3]). Instead we sketch a different argument, which relies only on the quasi-determinacy properties of the transition relations.

Proof. Consider any σ. We must show that either $\llbracket C \rrbracket_\theta(\sigma)$ is empty or $\llbracket C \rrbracket_\varphi(\sigma) = \llbracket C \rrbracket_\theta(\sigma)$.

First, we show $\llbracket C \rrbracket_\theta(\sigma) \subseteq \llbracket C \rrbracket_\varphi(\sigma)$. Suppose $\langle C, \sigma, \_\rangle \overset{\theta}{\longmapsto}{}^* \langle D, \tau, \mu\rangle$ and $\langle D, \tau, \mu\rangle$ is terminated or transitions to $\lightning$. A simple induction shows that $\langle C,\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle D,\ \tau,\ \mu\rangle$ because each step can be matched: By $\theta \sqsubseteq \varphi$, the only difference between $\overset{\theta}{\longmapsto}$ and $\overset{\varphi}{\longmapsto}$ is on context calls, and only in case where (for some m, v, v) we have $\theta(m)(v, v) = \varnothing$ and $\varphi(m)(v, v) \neq \varnothing$. Any context call in the execution $\langle C,\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle D,\ \tau,\ \mu\rangle$ must have a non-empty outcome, which can thus be matched by φ. Finally, if the last configuration faults via θ, it also faults via φ.

Now, we show that either $\llbracket C \rrbracket_\varphi(\sigma) \subseteq \llbracket C \rrbracket_\theta(\sigma)$, or $\llbracket C \rrbracket_\theta(\sigma) = \varnothing$. Suppose that, via $\overset{\varphi}{\longmapsto}$, a terminated configuration, or $\lightning$, can be reached from $\langle C,\ \sigma,\ \_\rangle$. The steps can be matched via $\overset{\theta}{\longmapsto}$, unless and until a configuration is reached where a context call applies $\theta(m)(v, v)$ but $\theta(m)(v, v) = \varnothing$.

To be more precise, we have two cases. If every terminated configuration or $\lightning$ reached via $\xmapsto{\varphi}$ from $\langle C,\ \sigma,\ \_\rangle$ is also reached via $\xmapsto{\theta}$ then we have $[\![C]\!]_\varphi(\sigma) \subseteq [\![C]\!]_\theta(\sigma)$ and we are done. Otherwise, suppose there is $\langle C,\ \sigma,\ \_\rangle \xmapsto{\varphi}{}^* \langle D,\ \tau,\ \mu\rangle$, that leads to an outcome, but *Active*(D) calls m for some v with $\theta(m)(\tau, v) = \varnothing$, so this particular trace has no outcome via $\xmapsto{\theta}$. We show there are no traces via $\xmapsto{\theta}$ that lead to an outcome, which proves $[\![C]\!]_\theta(\sigma) = \varnothing$. Suppose, for the sake of contradiction, that $\langle C,\ \sigma,\ \_\rangle \xmapsto{\theta}{}^* \langle B,\ v,\ \nu\rangle$ and the latter configuration is terminated or faults next. Let i be the length of the stuck trace $\langle C,\ \sigma,\ \_\rangle \xmapsto{\theta}{}^* \langle D,\ \tau,\ \mu\rangle$ and j be the length of $\langle C,\ \sigma,\ \_\rangle \xmapsto{\theta}{}^* \langle B,\ v,\ \nu\rangle$. We have $i \neq j$ because $\langle D,\ \tau,\ \mu\rangle$ is non-terminated and stuck, unlike $\langle B,\ v,\ \nu\rangle$. If $i < j$ then consider the length-i prefix of the latter; say it ends at $\langle D',\ \tau',\ \mu'\rangle$. By Lemma A.3(f), with $\sigma' := \sigma$, we have $D = D'$, $\mu = \mu'$, and $\tau \overset{\pi}{\approx} \tau'$ for some π. So by Lemma A.3(e), $\langle D',\ \tau',\ \mu'\rangle$ is stuck, contradicting $i < j$. If $i > j$ then consider the length j prefix of the first trace. By Lemma A.3(f), with $\sigma' := \sigma$, and Lemma A.3(e), the first trace should have terminated or faulted rather than getting stuck; again, contradiction. □

Definition A.8 (denotation of impure method body). Suppose Φ is a swf method context in typing context Γ and Θ is a single specification $m : (x{:}T)R \leadsto S\ [\eta]$ such that Φ, Θ is swf in Γ. Suppose B is swf as a body for m, *i.e.*, *sigs*$(\Phi, \Theta), \Gamma, x{:}T \vdash B$. Suppose φ is a candidate Φ-interpretation. We define by induction a sequence of functions θ_i with domain $\{m\}$ such that each $\theta_i(m)$ is in $(\sigma \in [\![\Gamma]\!]) \times [\![T]\!]\sigma \to \mathbb{P}([\![\Gamma]\!] \cup \{\lightning\})$. Define $\theta_0(m)$ as follows, for all $\sigma \in [\![\Gamma]\!]$, $v \in [\![T]\!]\sigma$:

- If $\sigma \not\models_\varphi R^x_v$ then $\theta_0(m)(\sigma, v) = \{\lightning\}$
- If $\sigma \models_\varphi R^x_v$ then $\theta_0(m)(\sigma, v) = \varnothing$

For $i > 0$ define $\theta_i(m)$ as follows, for all $\sigma \in [\![\Gamma]\!]$, $v \in [\![T]\!]\sigma$:

- If $\sigma \not\models_\varphi R^x_v$ then $\theta_i(m)(\sigma, v) = \{\lightning\}$
- If $\sigma \models_\varphi R^x_v$ then $\theta_i(m)(\sigma, v) = drop(x, [\![sigs(\Phi, \Theta), \Gamma, x : T \vdash B]\!]_{\varphi \cup \theta_{i-1}}([\sigma + x{:}\,v]))$.

where *drop*$(x, -)$ maps $-\restriction x$ over outcome sets and is defined for all τ, S by

$$\lightning \in drop(x, S) \text{ iff } \lightning \in S \qquad \text{and} \qquad \tau \in drop(x, S) \text{ iff } \tau = v\restriction x \text{ for some } v \in S$$

Finally, define the denotation θ by

$$\theta(m)(\sigma, v) = lub_i(\theta_i(m)(\sigma, v)) \tag{43}$$

Note that each $\varphi \cup \theta_i$ is a candidate interpretation of Φ, Θ, and so is $\varphi \cup \theta$.

To justify the definition, we first show that the sequence θ_i is an ascending chain, *i.e.*, $\theta_i \sqsubseteq \theta_{i+1}$ for all i. The proof is by induction on i. The case $\theta_0 \sqsubseteq \theta_1$ is direct from the definitions of $\theta_0, \theta_1, \sqsubseteq$. For the inductive step $\theta_i \sqsubseteq \theta_{i+1}$, the inductive hypothesis is $\theta_{i-1} \sqsubseteq \theta_i$, which amounts to $\theta_{i-1}(m) \sqsubseteq \theta_i(m)$. To prove the step, we have for any (σ, v) that

$$[\![sigs(\Phi, \Theta), \Gamma, x : T \vdash B]\!]_{\varphi \cup \theta_{i-1}}([\sigma + x{:}\,v]) \sqsubseteq [\![sigs(\Phi, \Theta), \Gamma, x : T \vdash B]\!]_{\varphi \cup \theta_i}([\sigma + x{:}\,v])$$

by Lemma A.7. So by definition of θ_i and θ_{i+1} we have $\theta_i(m)(\sigma, v) \sqsubseteq \theta_{i+1}(m)(\sigma, v)$.

Notice that the least upper bound used to define θ in (43) is for an ascending chain of outcome sets. Owing to the flat ordering on outcome sets, defined by (42), any chain consists of all empty sets, or some empty and all the rest equal. As a direct consequence we obtain the following.

Lemma A.9. The least upper bound of the chain θ_i is θ. Moreover, for any (σ, v), there is i such that for all $j \geq i$ we have $\theta(m)(\sigma, v) = \theta_j(m)(\sigma, v)$.

Next, we show that each $\theta_i(m)$ is quasi-deterministic, provided that φ is. The proof is by induction on i. For θ_0 it suffices that formulas respect isomorphism of states, Lemma A.2. For θ_{i+1}, using quasi-determinacy of $\varphi \cup \theta_i$, we get the conditions of Lemma A.3 for $\overset{\varphi\cup\theta_i}{\longmapsto}$. In particular, conditions (d), (f), and (g) can be used with the definitions of $[\![C]\!]_{\varphi\cup\theta_i}$ and $\theta_{i+1}(m)$ to show that $\theta_{i+1}(m)$ has the requisite properties when applied to any state,value pair.

Now we can show that $\varphi \cup \theta$ is quasi-deterministic. Suppose, by Lemma A.9, that $\theta(m)(\sigma, v) = \theta_i(m)(\sigma, v)$. Then the requisite properties of $\theta(m)(\sigma, v)$ hold because θ_i is quasi-deterministic.

The following key result says how, if B is correct and φ is a context interpretation for Φ, the θ defined above makes $\varphi \cup \theta$ be a context interpretation for Φ, Θ. (In that case, quasi-determinacy of $\varphi \cup \theta$ can be obtained via Lemma A.4.)

Lemma A.10 (context interpretation for impure). Suppose Φ, Θ is swf in Γ, where Θ is $m : (x{:}T)R \rightsquigarrow S\ [\eta]$, and suppose $\Phi, \Theta; \psi \models^{\Gamma, x:T} B : R \rightsquigarrow S\ [\mathsf{rd}\ x, \eta]$. For any Φ-interpretation φ that extends ψ, we have that $\varphi \cup \theta_i$ from Definition A.8 is a Φ, Θ-interpretation, for all i. Moreover, the least upper bound θ is a Φ, Θ-interpretation.

Proof. First, by induction on i we show that $\varphi \cup \theta_i$ is a Φ, Θ-interpretation. Since φ is a context interpretation, it is enough to show that $\theta_i(m)$ satisfies the conditions of Definition 5.1(d–f), taking $\varphi := \varphi \cup \theta_i$ in the Definition. For any $\sigma \in [\![\Gamma]\!]$ and $v \in [\![T]\!]\sigma$, we have the following.

<u>Base case $i = 0$</u>. All of (d), (e), and (f) are direct from the definition of $\theta_i(m)$.

<u>Inductive case $i > 0$</u>.

(d) If $\sigma \not\models_{\varphi\cup\theta_i} R^x_v$ then (d) is direct from the definition of $\theta_i(m)$. If $\sigma \models_{\varphi\cup\theta_i} R^x_v$ then $\sigma \models_\varphi R^x_v$ (since m is impure). Observe that $\varphi \cup \theta_{i-1}$ is a Φ, Θ-interpretation, by induction, and it extends ψ. So from $\Phi, \Theta; \psi \models^{\Gamma, x:T} B : R \rightsquigarrow S\ [\mathsf{rd}\ x, \eta]$ we get that execution of B from σ, v via $\varphi \cup \theta_{i-1}$ does not fault; again (d) follows by definition of $\theta_i(m)$.

(e) We only need to consider σ, v such that $\sigma \models_{\varphi\cup\theta_i} R^x_v$. Observe that $\varphi \cup \theta_{i-1}$ is a Φ, Θ-interpretation, by induction, and it extends ψ. So from $\Phi, \Theta; \psi \models^{\Gamma, x:T} B : R \rightsquigarrow S\ [\mathsf{rd}\ x, \eta]$ we get that if execution of B from σ, v via $\varphi \cup \theta_{i-1}$ terminates in a state τ then $\tau \upharpoonright x$ satisfies the postcondition and write effect. Thus, by definition of $\theta_i(m)$, we get (e).

(f) Similar to the proof of (e) but considering two executions.

This finishes the proof that each $\varphi \cup \theta_i$ is a context interpretation.

To show that $\varphi \cup \theta$ is a Φ, Θ-interpretation, it is enough to show that $\theta(m)$ satisfies the conditions of Definition 5.1(d–f), taking $\varphi := \varphi \cup \theta$ in the Definition. For each of (d), (e), and (f) the argument goes by spelling out the condition on $\theta(m)(\sigma, v)$, choosing k such that $\theta(m)(\sigma, v) = \theta_k(m)(\sigma, v)$ (by Lemma A.9), and appealing to the same condition for $\theta_k(m)$ which is proved above. In the case of (f) we consider two inputs, with $\theta(m)(\sigma, v) = \theta_k(m)(\sigma, v)$ and $\theta(m)(\sigma', v') = \theta_{k'}(m)(\sigma, v)$, and choose the max of k, k' to complete the argument. □

Trace decomposition. The remaining lemmas needed to prove soundness of ImpureLink describe how a trace can be decomposed into convenient segments, similar to Lemmas 8.3 and 8.4 for pure methods.

Lemma A.11 (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 $\langle C_0,\ \sigma_0,\ \mu_0 \rangle \overset{\varphi}{\longmapsto}{}^{*} \langle D,\ \tau,\ v \rangle$. 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, v_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 $\sigma_i = \dot\sigma_i \upharpoonright x_i$

(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.12 (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.

A.2 Soundness of the ImpureLink rule

Suppose that Θ is $m : (x{:}T)R \rightsquigarrow S\ [\eta]$. Suppose the premises of the rule are valid:

$$\Phi, \Theta; \psi \models^{\Gamma, x:T} B :\ R \rightsquigarrow S\, [\mathsf{rd}\, x, \eta] \quad \text{and} \quad \Phi, \Theta; \psi \models^{\Gamma} C :\ P \rightsquigarrow Q\, [\varepsilon] \tag{44}$$

We must prove the conclusion is valid:

$$\Phi; \psi \models^{\Gamma} \mathsf{let}\ m(x{:}T) = B\ \mathsf{in}\ C :\ P \rightsquigarrow Q\, [\varepsilon] \tag{45}$$

That judgment is about about executions via $\overset{\varphi}{\longmapsto}$ but the premises pertain to execution via Φ, Θ-interpretations that extend ψ. For any Φ-interpretation φ that extends ψ, we will use $\varphi \cup \theta$ where θ is given by Def. A.8. Lemma A.10 says that $\varphi \cup \theta$ is a Φ, Θ-interpretation. Using this interpretation, we get the following recursion lemma.

Lemma A.13 (recursion for impure). Let x' not be in $\mathit{dom}(\Gamma) \cup \{x\}$. Let Γ' be $\Gamma, x' : T$. Let σ be any Γ'-state such that $\sigma \models_\psi R^x_{x'}$. Let $\dot\mu$ be any Γ'-environment such that $\dot\mu(m) = (x : T.B)$. Then the computation from $\langle B^x_{x'},\, \sigma,\, \dot\mu\rangle$ via $\overset{\varphi}{\longmapsto}$ does not fault and if it reaches $\langle \mathsf{skip},\, \tau,\, \dot\mu\rangle$ then $\tau \upharpoonright x'$ is in $\theta(m)(\sigma \upharpoonright x', \sigma(x'))$.

The proof of the Lemma is deferred. We proceed to prove (45).

To that end, let φ be a Φ-interpretation such that $\psi \subseteq \varphi$ and let σ be a state such that $\sigma \models^{\Gamma, \mathit{sigs}(\Phi)}_\varphi P$. We only need to consider executions from the empty environment, since by well-formedness all method calls in C are to m or to methods let-bound within C or to methods in context Φ. For notational clarity, let us write μ for the empty environment. To prove (45) we consider executions from $\langle \mathsf{let}\ m(x{:}T) = B\ \mathsf{in}\ C,\, \sigma,\, \mu\rangle$. By transition semantics, 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 $\dot\mu = [\mu + m\colon (x : T.B)]$. 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 and having exactly the same states, followed by a final step that executes $\mathsf{elet}(m)$. The final step just removes m from $\dot\mu$, which means it does not fault or change the state. Thus to finish the proof it is enough to prove, owing to Lemma 8.5, the following:

(i) it is not the case that $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}^* ↯$,
(ii) for any τ, if $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}^* \langle \text{skip},\ \tau,\ \dot{\mu}\rangle$ then $\tau \models_\varphi Q$ and $\sigma{\rightarrow}\tau \models_\varphi \varepsilon$,
(iii) for all $\tau, \sigma', \tau', \pi$ if $\sigma' \models^\Gamma_\varphi P$ and $Agree(\sigma, \sigma', \varepsilon, \pi, \varphi)$
and $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}^* \langle \text{skip},\ \tau,\ \dot{\mu}\rangle$ and $\langle C,\ \sigma',\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}^* \langle \text{skip},\ \tau',\ \dot{\mu}\rangle$ then there is ρ with $\rho \supseteq \pi$, $\rho(freshLocs(\sigma,\tau)) \subseteq freshLocs(\sigma',\tau')$ and $Lagree(\tau, \tau', \rho, written(\sigma,\tau) \cup freshLocs(\sigma,\tau))$.

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 $\mu' = \dot{\mu}' {\upharpoonright} m$. Also, if $C' = m(z); D$ for some z, D then $\sigma' \models_{\varphi\cup\theta} R^x_z$.

(i) Suppose $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}^* \langle C',\ \sigma',\ \dot{\mu}'\rangle \overset{\varphi}{\longmapsto} ↯$. 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 $\langle C',\ \sigma',\ \dot{\mu}'\rangle \overset{\varphi}{\longmapsto} ↯$ we have by semantics that $Active(C')$ is a field access/update or a context call, and hence not a call to m. Thus by the independence Lemma 8.1 we get $\langle C',\ \sigma',\ \mu'\rangle \overset{\varphi\cup\theta}{\longmapsto} ↯$. But this contradicts validity of the correctness judgment (44) for C (instantiated by $\varphi \cup \theta$). 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.11 it can be decomposed as

$$\begin{array}{ll} \langle C,\ \sigma,\ \dot{\mu}\rangle & \\ \overset{\varphi}{\longmapsto}^* \langle m(z); D,\ \tau,\ \dot{\nu}\rangle & \\ \overset{\varphi}{\longmapsto} \langle B^x_{x'}; \text{ecall}(x'); D,\ \upsilon,\ \dot{\nu}\rangle & \text{where } x' \text{ is a fresh variable and } \upsilon \text{ is } [\tau + x' \colon \tau(z)] \\ \overset{\varphi}{\longmapsto}^* \langle A; \text{ecall}(x'); D,\ \sigma',\ \dot{\mu}'\rangle & \text{where } C' \text{ is } A; \text{ecall}(x'); D \\ \overset{\varphi}{\longmapsto} ↯ & \end{array}$$

So we have $\langle B^x_{x'},\ \upsilon,\ \dot{\nu}\rangle \overset{\varphi}{\longmapsto}^* ↯$. On the other hand, $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}^* \langle m(z); D,\ \tau,\ \dot{\nu}\rangle$ is an m-truncated trace. So by Claim A, we have $\tau \models_{\varphi\cup\theta} R^x_z$ and thus $\upsilon \models_\varphi R^x_{x'}$ (using (6)), whence by Lemma A.13 $\langle B^x_{x'},\ \upsilon,\ \dot{\nu}\rangle$ does not fault — a contradiction. To be precise, note that υ may have additional variables (locals and parameters) besides those of $\Gamma, x' : T$; but since B does not touch them, they can be projected out to obtain a trace to which the Lemma applies literally. So (i) is proved.

(ii) Suppose $\langle C,\ \sigma,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}^* \langle \text{skip},\ \tau,\ \dot{\mu}\rangle$. This is m-truncated, so by Claim A, we get $\langle C,\ \sigma,\ \mu\rangle \overset{\varphi\cup\theta}{\longmapsto}^* \langle \text{skip},\ \tau,\ \mu\rangle$. Also by the hypothesis, we have $\sigma \models_\varphi P$. Since m is impure, $\models_{\varphi\cup\theta}$ is $\models_\varphi$ and $wlocs(\sigma, \varphi, \varepsilon) = wlocs(\sigma, \varphi \cup \theta, \varepsilon)$. So we have $\sigma \models_{\varphi\cup\theta} P$, and again we can instantiate the premise for C with $\varphi \cup \theta$, obtaining $\tau \models_\varphi Q$ and $\sigma{\rightarrow}\tau \models_\varphi \varepsilon$.

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

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, and there is a trace $\langle C,\ \sigma,\ \mu\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^{*} \langle C_0,\ \sigma_0,\ \mu_0\rangle$ then there is a trace

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

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

To prove Claim A, 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 $\langle C',\ \sigma',\ \mu'\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^{*} ⇯$ and hence $\langle C,\ \sigma,\ \mu\rangle \overset{\varphi\cup\theta}{\longmapsto}{}^{*} ⇯$. 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 via $\overset{\varphi\cup\theta}{\longmapsto}$, by induction on the number n of completed topmost calls of m in the trace via $\overset{\varphi}{\longmapsto}$. Accordingly, consider an m-truncated trace

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

Using Lemma A.11, 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

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

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

In accord with Lemma A.12 we will construct a trace via $\overset{\varphi\cup\theta}{\longmapsto}$ that looks as follows:

$\langle C_0,\ \sigma_0,\ \mu_0\rangle$	
$\overset{\varphi\cup\theta}{\longmapsto}{}^{*}\langle m(z_1);\ C_1,\ \tau_1,\ \mu_1\rangle$	matching the configurations above, so $\mu_1 = \dot{\mu}_1 \restriction m$
$\overset{\varphi\cup\theta}{\longmapsto} \langle C_1,\ \sigma_1,\ \mu_1\rangle$	a single step by Lemma 8.4 (2) $(*)$
$\vdots$	containing $n-1$ additional invocations of m
$\overset{\varphi\cup\theta}{\longmapsto} \langle C_n,\ \sigma_n,\ \mu_n\rangle$	
$\overset{\varphi\cup\theta}{\longmapsto}{}^{*}\langle C',\ \sigma',\ \mu'\rangle$	again matching configurations

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 Lemma 8.1 we can 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 validity of the correctness judgment (44) for C, which we can appeal to since $\sigma \models_\varphi P$ gives us $\sigma \models_{\varphi\cup\theta} P$ by (6). Since $\tau_1 \models_\varphi Q^x_{z_1}$ we get $v_1 \models_\varphi Q^x_{x_1}$. From the given trace and its decomposition, we have $\langle B^x_{x_1},\ v_1,\ \dot{\mu}_1\rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip},\ \dot{\sigma}_1,\ \dot{\mu}_1\rangle$, so we have $\dot{\sigma}_1 \restriction x_1 \in \theta(m)(v_1 \restriction x_1, \tau_1(z_1))$ by Lemma A.13. So by definitions of σ_1 and v_1 we have $\sigma_1 \in \theta(m)(\tau_1, \tau_1(z_1))$. (Strictly speaking, if there are any extra variables in τ_1 they should be dropped from v_1 and the subsequent configurations before applying the Lemma; then added back to the conclusion of the Lemma using the implicit coercion for interpretations, *cf.* Sec. 3.2.) Now we can instantiate the transition semantics for impure call in Fig. 9, to get $\langle m(z_1);\ C_1,\ \tau_1,\ \mu\rangle \overset{\varphi\cup\theta}{\longmapsto} \langle C_1,\ \sigma_1,\ \mu\rangle$. 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 configuration $\langle C_1,\ \sigma_1,\ \dot{\mu}_1\rangle$ onward is a trace with $n-1$ completed invocations of m, from a configuration reachable from $\langle C,\ \sigma,\ \dot{\mu}\rangle$. 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 obtain the needed trace via $\overset{\varphi\cup\theta}{\longmapsto}$ and conclude the proof of Claim B.

Recursion lemma. It remains to prove Lemma A.13. We just give a sketch, as most of the details are similar to the argument for (i–iii). Here we only review the major differences. To deal with recursion, we prove the following, which applies to $\sigma, \dot{\mu}$ etc. satisfying the hypotheses of the Lemma, and uses depth-bounded semantics.

Main Claim For all $k \geq 0$, the computation from $\langle B^x_{x'},\ \sigma,\ \dot{\mu}\rangle^k$ via $\overset{\varphi}{\longmapsto}$, (a) does not fault and (b) if reaches $\langle \mathsf{skip},\ \tau,\ \dot{\mu}\rangle^k$ then $\tau \restriction x'$ is in $\theta_{k+1}(m)(\sigma \restriction x', \sigma(x'))$.

Lemma A.13 follows directly from the Main Claim using Lemmas A.9 and 8.7, and Lemma A.10.

We proceed to prove the Main claim, by induction on k.

For the base case of $k = 0$ the argument of (a) is as follows. Suppose that $\langle B^x_{x'},\ \sigma,\ \dot{\mu}\rangle^0 \overset{\varphi}{\longmapsto}{}^* \langle C',\ \sigma',\ \dot{\mu}\rangle^0 \overset{\varphi}{\longmapsto} \lightning$. Since $k = 0$, there is no call to m in this trace before the last configuration. So, there is a matching trace $\langle B^x_{x'},\ \sigma,\ \dot{\mu}\rangle^0 \overset{\varphi\cup\theta'}{\longmapsto}{}^* \langle C',\ \sigma',\ \dot{\mu}\rangle^0$, for any θ' such that $\varphi \cup \theta'$ is a Φ, Θ-interpretation. On the other hand, *Active*(C') is field access/update or context call. Thus we have $\langle C',\ \sigma',\ \dot{\mu}\rangle^0 \overset{\varphi\cup\theta'}{\longmapsto} \lightning$. Using Lemma 8.7, we get

$$\langle B^x_{x'},\ \sigma,\ \dot{\mu}\rangle \overset{\varphi\cup\theta'}{\longmapsto}{}^* \langle C',\ \sigma',\ \dot{\mu}\rangle \overset{\varphi\cup\theta'}{\longmapsto} \lightning$$

This contradicts the premise (44) for B. Thus the computation form $\langle B^x_{x'},\ \sigma,\ \dot{\mu}\rangle^0$ via $\overset{\varphi}{\longmapsto}$ does not fault. To prove (b) for $k = 0$, suppose we have $\langle B^x_{x'},\ \sigma,\ \dot{\mu}\rangle^0 \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip},\ \tau,\ \dot{\mu}\rangle^0$. For abbreviation, let $\sigma^{x'}_x$

be σ with x' renamed to x, *i.e.*, $\sigma^{x'}_x = [\sigma + x\colon \sigma(x')] \restriction x'$ (and the same for $\tau^{x'}_x$). By renaming each state in the trace we get $\langle B, \sigma^{x'}_x, \dot{\mu}\rangle^0 \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip}, \tau^{x'}_x, \dot{\mu}\rangle^0$. So by definition of $[\![\ldots]\!]$, using Lemma 8.7, we get $\tau^{x'}_x \in [\![sigs(\Phi,\Theta), \Gamma, x : T \vdash B]\!]_\varphi(\sigma^{x'}_x)$ By definition of $\theta_1(m)$ we get $\tau^{x'}_x \restriction x \in \theta_1(m)(\sigma^{x'}_x \restriction x, \sigma^{x'}_x(x))$ which simplifies to $\tau \restriction x' \in \theta_1(m)(\sigma \restriction x', \sigma(x'))$ and we are done with the base case.

To prove the inductive case $k > 0$, we need the following claim.

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

Using Claim A′ and the induction hypothesis, the proof of (a) in the main claim is similar to the argument for (i) earlier, and is omitted. To prove (b), suppose we have $\langle B^x_{x'}, \sigma, \dot{\mu}\rangle^k \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip}, \tau, \dot{\mu}\rangle^k$. By Claim A′, we get $\langle B^x_{x'}, \sigma, \dot{\mu}\rangle^k \overset{\varphi\cup\theta_k}{\longmapsto}{}^* \langle \mathsf{skip}, \tau, \dot{\mu}\rangle^k$. By reasoning similar to the argument above for (b) in the base case, but using the definition of θ_{k+1}, we get $\tau \restriction x' \in \theta_{k+1}(m)(\sigma \restriction x', \sigma(x'))$.

It remains to prove Claim A′. The proof follows the lines of the argument for Claim A, including its supporting Claim B, but in place of appeals to the premise (44) for C and to Lemma A.13 for B, we appeal to the induction hypothesis for $k-1$.

A.3 Read for While Rule

The WHILE rule is similar to the one proved sound in [8]. Here, we only show the Read property.

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

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

$$\sigma \models_\varphi P \land r = \mathsf{alloc}, \qquad \sigma' \models_\varphi P \land r = \mathsf{alloc}, \qquad Agree(\sigma, \sigma', \eta, \pi, \varphi), \tag{46}$$

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 $\rho \supseteq \pi$, $\rho(freshRefs(\sigma,\tau)) \subseteq freshRefs(\sigma',\tau')$, and $Lagree(\tau, \tau', \rho, written(\sigma,\tau) \cup freshLocs\sigma,\tau))$.

By semantics (as pointed out in Theorem 7.4 in [8]), 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 $\sigma'_n(x) = 0$.

To finish the proof, we prove the following Claim.

Claim. For all k, $0 \leq k \leq m$, we have $k \leq n$, $\sigma_k \models_\varphi P$ and there is a refperm $\rho_k \supseteq \pi$ such that $rlocs(\sigma_k, \varphi, \eta) = rlocs(\sigma_0, \varphi, \eta)$ and $rlocs(\sigma'_k, \varphi, \eta) = rlocs(\sigma'_0, \varphi, \eta)$, $Agree(\sigma_k, \sigma'_k, (\eta, \mathsf{rd}\ H`\overline{f}), \rho_k, \varphi)$ and $Agree(\sigma'_k, \sigma_k, (\eta, \mathsf{rd}\ H`\overline{f}), \rho_k^{-1}, \varphi)$, $\rho_k(freshRefs(\sigma_0, \sigma_k)) \subseteq freshRefs(\sigma'_0, \sigma'_k)$, $Lagree(\sigma_k, \sigma'_k, \rho_k, written(\sigma_0, \sigma_k) \cup freshLocs(\sigma_0, \sigma_k))$ and $Lagree(\sigma'_k, \sigma_k, \rho_k^{-1}, freshLocs(\sigma'_0, \sigma'_k))$.

To prove Read, first note that from the claim we have $m \leq n$ and $Agree(\sigma_m, \sigma'_m, \eta, \rho_m, \varphi)$. 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 statements of the claim for $k = m$ give us

$$\rho(freshRefs(\sigma, \tau)) \subseteq freshRefs(\sigma', \tau') \text{ and } Lagree(\tau, \tau', \rho, written(\sigma, \tau) \cup freshLocs(\sigma, \tau)),$$

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

Proof of the claim is by induction on k. For base case of $k = 0$, we take $\rho_0 = \pi$. From (46), we know that $Agree(\sigma, \sigma', \eta, \pi, \varphi)$. Since $\Phi; \psi \models P \Rightarrow H\#r$ and $\sigma \models_\varphi P \wedge r = \mathsf{alloc}$, we have $\sigma_0 \models H \subseteq \{null\}$, so $rlocs(\sigma_0, \varphi, \mathsf{rd}\ H`\overline{f}) = \varnothing$. Thus we have $Agree(\sigma_0, \sigma'_0, (\eta, \mathsf{rd}\ H`\overline{f}), \rho_0, \varphi)$. On the other hand, since ε has framed reads, η has also framed reads. So using Lemma 6.11 we have $Agree(\sigma_0, \sigma'_0, \eta, \rho_0^{-1}, \varphi)$. With a similar argument for σ'_0, we get $rlocs(\sigma'_0, \varphi, \mathsf{rd}\ H`\overline{f}) = \varnothing$. Thus we have $Agree(\sigma'_0, \sigma_0, (\eta, \mathsf{rd}\ H`\overline{f}), \rho_0^{-1}, \varphi)$. We also have $\rho_0(freshRefs(\sigma_0, \sigma_0)) \subseteq freshRefs(\sigma'_0, \sigma'_0)$, $Lagree(\sigma_0, \sigma'_0, \pi, written(\sigma_0, \sigma_0) \cup freshLocs(\sigma_0, \sigma_0))$ and $Lagree(\sigma'_0, \sigma_0, \pi^{-1}, freshLocs(\sigma'_0, \sigma'_0))$, because $written(\sigma_0, \sigma_0) = freshRefs(\sigma_0, \sigma_0) = freshRefs(\sigma'_0, \sigma'_0) = \varnothing$.

To prove the 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

$$Agree(\sigma_{k-1}, \sigma'_{k-1}, (\eta, \mathsf{rd}\ H`\overline{f}), \rho_{k-1}, \varphi) \text{ and } Agree(\sigma'_{k-1}, \sigma_{k-1}, (\eta, \mathsf{rd}\ H`\overline{f}), \rho_{k-1}^{-1}, \varphi) \tag{47}$$

Thus $\sigma'_{k-1}(x) \neq 0$. This means $k \leq n$, i.e., the computation starting from state σ'_0 has at least k iterations. Since η is $\varepsilon, \mathsf{rd}\ x$, the agreement (47) implies

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

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

$$\begin{gathered} \rho_k(freshRefs(\sigma_{k-1}, \sigma_k)) \subseteq freshRefs(\sigma'_{k-1}, \sigma'_k) \\ Lagree(\sigma_k, \sigma'_k, \rho_k, written(\sigma_{k-1}, \sigma_k) \cup freshLocs(\sigma_{k-1}, \sigma_k)) \end{gathered} \tag{48}$$

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

$$rlocs(\sigma_k, \varphi, \eta) = rlocs(\sigma_{k-1}, \varphi, \eta) = rlocs(\sigma_0, \varphi, \eta). \tag{49}$$

With a similar argument we get

$$rlocs(\sigma'_k, \varphi, \eta) = rlocs(\sigma'_{k-1}, \varphi, \eta) = rlocs(\sigma'_0, \varphi, \eta).$$

From Post condition of the premise for C and the induction hypothesis, we have $\sigma_k \models_\varphi P$. From side conditions $\Phi; \psi \models P \Rightarrow H\#r$ and $\mathsf{wr}\ r \notin \varepsilon$ of WHILE, using also $\sigma_0 \models_\varphi r = \mathsf{alloc}$, we have $[\![H]\!]_\varphi \sigma_k \subseteq freshRefs(\sigma_0, \sigma_k)$. Thus

$$rlocs(\sigma_k, \varphi, \mathsf{rd}\ H`\overline{f}) \subseteq freshLocs(\sigma_0, \sigma_k) \tag{50}$$

With a similar argument we get

$$rlocs(\sigma'_k, \varphi, \mathsf{rd}\ H`\overline{f}) \subseteq freshLocs(\sigma'_0, \sigma'_k) \tag{51}$$

From induction hypothesis, we have $Lagree(\sigma_{k-1}, \sigma'_{k-1}, \rho_{k-1}, written(\sigma_0, \sigma_{k-1}) \cup freshLocs(\sigma_0, \sigma_{k-1}))$. From the premise for C and (47), we have

$$\begin{array}{l} \sigma_{k-1}, \sigma'_{k-1} {\Rightarrow} \sigma_k, \sigma'_k \models_\varphi \varepsilon, \mathsf{wr}\ H`\overline{f}, \mathsf{rd}\ H`\overline{f} \quad , \\ \sigma'_{k-1}, \sigma_{k-1} {\Rightarrow} \sigma'_k, \sigma_k \models_\varphi \varepsilon, \mathsf{wr}\ H`\overline{f}, \mathsf{rd}\ H`\overline{f} \quad , \\ Agree(\sigma_{k-1}, \sigma'_{k-1}, (\varepsilon, \mathsf{rw}\ H`\overline{f}), \rho_{k-1}, \varphi) \quad \text{and} \\ Agree(\sigma'_{k-1}, \sigma_{k-1}, (\varepsilon, \mathsf{rw}\ H`\overline{f}), \rho_{k-1}^{-1}, \varphi). \end{array} \tag{52}$$

Lemma 6.12 yields $Lagree(\sigma_k, \sigma'_k, \rho_k, written(\sigma_0, \sigma_{k-1}) \cup freshLocs(\sigma_0, \sigma_{k-1}))$. Since $written(\sigma_0, \sigma_k) \subseteq written(\sigma_0, \sigma_{k-1}) \cup written(\sigma_{k-1}, \sigma_k)$ and $freshLocs(\sigma_0, \sigma_k) = freshLocs(\sigma_0, \sigma_{k-1}) \cup freshLocs(\sigma_{k-1}, \sigma_k)$, from (48), we have

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

and from the induction hypothesis and (48) we have

$$\begin{array}{rcl} \rho_k(freshRefs(\sigma_0, \sigma_k)) & = & \rho_k(freshRefs(\sigma_0, \sigma_{k-1})) \cup \rho_k(freshRefs(\sigma_{k-1}, \sigma_k)) \\ & \subseteq & freshRefs(\sigma'_0, \sigma'_{k-1}) \cup freshRefs(\sigma'_{k-1}, \sigma'_k) \\ & = & freshRefs(\sigma'_0, \sigma'_k) \end{array}$$

Thus we get

$$\begin{array}{c} Lagree(\sigma_k, \sigma'_k, \rho_k, written(\sigma_0, \sigma_k) \cup freshLocs(\sigma_0, \sigma_k)) \\ \rho_k(freshRefs(\sigma_0, \sigma_k)) \subseteq freshRefs(\sigma'_0, \sigma'_k) \end{array} \tag{53}$$

From (52) and (48), using Lemma 6.13, we get

$$Lagree(\sigma'_k, \sigma_k, \rho_k^{-1}, freshLocs(\sigma'_{k-1}, \sigma'_k)) \tag{54}$$

By induction hypothesis we have

$$Lagree(\sigma'_{k-1}, \sigma_{k-1}, \rho_{k-1}^{-1}, freshLocs(\sigma'_0, \sigma'_{k-1}))$$

Using lemma 6.12 from (52), we get

$$Lagree(\sigma'_k, \sigma_k, \rho', freshLocs(\sigma'_0, \sigma'_{k-1}))$$

for some $\rho' \supseteq \rho_{k-1}^{-1}$. Since $freshRefs(\sigma'_0, \sigma'_{k-1}) \subseteq \sigma'_{k-1}(\text{alloc})$, any fresh references in ρ' are not relevant, so we get

$$Lagree(\sigma'_k, \sigma_k, \rho_{k-1}^{-1}, freshLocs(\sigma'_0, \sigma'_{k-1}))$$

Since the restriction of ρ_k^{-1} to $\sigma'_{k-1}(\text{alloc})$ is equal to ρ_{k-1}^{-1}, we get

$$Lagree(\sigma'_k, \sigma_k, \rho_k^{-1}, freshLocs(\sigma'_0, \sigma'_{k-1}))$$

Combining with (54), since $freshRefs(\sigma'_0, \sigma'_k) = freshRefs(\sigma'_0, \sigma'_{k-1}) \cup freshRefs(\sigma'_{k-1}, \sigma'_k)$, we have

$$Lagree(\sigma'_k, \sigma_k, \rho_k^{-1}, freshLocs(\sigma'_0, \sigma'_k)) \tag{55}$$

The remaining part of the claim is the agreements on effects. Note (47) implies

$$Lagree(\sigma_{k-1}, \sigma'_{k-1}, \rho_{k-1}, rlocs(\sigma_{k-1}, \varphi, \eta)).$$

From (52) and (49), using Lemma 6.12, we get $Lagree(\sigma_k, \sigma'_k, \rho_k, rlocs(\sigma_k, \varphi, \eta))$. This means

$$Agree(\sigma_k, \sigma'_k, \eta, \rho_k, \varphi) \tag{56}$$

Using Lemma 6.11, since η has framed reads, we get

$$Agree(\sigma'_k, \sigma_k, \eta, \rho_k^{-1}, \varphi) \tag{57}$$

Now, for $\text{rd } H`\overline{f}$, from (50) and (53), we get

$$Lagree(\sigma_k, \sigma'_k, \rho_k, rlocs(\sigma_k, \varphi, \text{rd } H`\overline{f})) \tag{58}$$

and from (51) and (55)

$$Lagree(\sigma'_k, \sigma_k, \rho_k^{-1}, rlocs(\sigma'_k, \varphi, \text{rd } H`\overline{f})) \tag{59}$$

So using (56) and (58) we get $Agree(\sigma_k, \sigma'_k, (\eta, \text{rd } H`\overline{f}), \rho_k, \varphi)$ and from (57) and (59) we get $Agree(\sigma'_k, \sigma_k, (\eta, \text{rd } H`\overline{f}), \rho_k^{-1}, \varphi)$. This finishes the proof.

INDEX

Received December 2015; revised August 2017; accepted December 2017