A Relational Program Logic with Data Abstraction and Dynamic Framing

ANINDYA BANERJEE, IMDEA Software Institute, Spain
RAMANA NAGASAMUDRAM, DAVID A. NAUMANN, and MOHAMMAD NIKOUEI,
Stevens Institute of Technology, USA

Dedicated to Tony Hoare.

In a paper published in 1972, Hoare articulated the fundamental notions of hiding invariants and simulations. Hiding: invariants on encapsulated data representations need not be mentioned in specifications that comprise the API of a module. Simulation: correctness of a new data representation and implementation can be established by proving simulation between the old and new implementations using a coupling relation defined on the encapsulated state. These results were formalized semantically and for a simple model of state, though the paper claimed this could be extended to encompass dynamically allocated objects. In recent years, progress has been made toward formalizing the claim, for simulation, though mainly in semantic developments. In this article, hiding and simulation are combined with the idea in Hoare's 1969 paper: a logic of programs. For an object-based language with dynamic allocation, we introduce a relational Hoare logic with stateful frame conditions that formalizes encapsulation, hiding of invariants, and couplings that relate two implementations. Relations and other assertions are expressed in first-order logic. Specifications can express a wide range of relational properties such as conditional equivalence and noninterference with declassification. The proof rules facilitate relational reasoning by means of convenient alignments and are shown sound with respect to a conventional operational semantics. A derived proof rule for equivalence of linked programs directly embodies representation independence. Applicability to representative examples is demonstrated using an SMT-based implementation.

CCS Concepts: • **Theory of computation → Programming logic**; **Hoare logic**; **Semantics and reasoning;**

Additional Key Words and Phrases: Relational properties, relational verification, logics of programs, data abstraction, representation independence, product programs, automated verification

25

Nagasamudram and Nikouei were partially supported by National Science Foundation (NSF) Award No. 1718713. Naumann was partially supported by NSF Award No. 1718713 and Office of Naval Research (ONR) Award No. N00014-17-1-2787. Banerjee's research was based on work supported by the NSF, while working at the Foundation; in particular, he gratefully acknowledges NSF's support of "Long-term Professional Development" for FY 2020. Any opinions, findings, and conclusions or recommendations expressed in this article are those of the authors and do not necessarily reflect the views of the NSF and other funding agencies.

Authors' addresses: A. Banerjee, IMDEA Software Institute, Pozuelo de Alarcon, Madrid, Spain; email: anindya.banerjee@imdea.org; R. Nagasamudram, D. A. Naumann, and M. Nikouei, Stevens Institute of Technology, Hoboken, New Jersey, USA; emails: rnagasam@stevens.edu, naumann@cs.stevens.edu, snikouei@stevens.edu.

0164-0925/2022/12-ART25 $15.00
https://doi.org/10.1145/3551497

ACM Reference format:
Anindya Banerjee, Ramana Nagasamudram, David A. Naumann, and Mohammad Nikouei. 2022. A Relational Program Logic with Data Abstraction and Dynamic Framing. *ACM Trans. Program. Lang. Syst.* 44, 4, Article 25 (December 2022), 136 pages.
https://doi.org/10.1145/3551497

1 INTRODUCTION

Data abstraction has been a cornerstone of software development methodology since the 1970s. Yet it is surprisingly difficult to achieve in a reliable manner in modern programming languages that permit manipulation of the global heap via dynamic allocation, shared mutable objects, and callbacks. Aliasing can violate conventional syntactic means of encapsulation (modules, classes, packages, access modifiers) and therefore can undercut the fundamental guarantee of abstraction: equivalence of client behavior under change of a module's data structure representations.

The theory of data abstraction is well-known since Hoare's seminal paper [52]. Its main ingredients are the encapsulation of effects, hidden invariants (that is, private invariants that do not appear in a method's interface specifications, so that clients are exempt from having to establish them for calls to the method), and relational reasoning: coupling relations and simulations. Hoare's paper provides a semantic formalization of these ideas using a simple model of state and it claims that the ideas can be extended to encompass dynamically allocated objects.

The justification of Hoare's claim is a primary focus of this article, which is in the context of two strands of recent work. One strand has made progress on automating proofs of conditional equivalence and relational properties in general, based on automated theorem proving (e.g., SMT) and techniques to decompose relational reasoning by expressing alignment of executions in terms of "product programs." The other strand has made progress toward formalizing Hoare's claim in semantic theories of representation independence (simulation and logical relations). This article brings the strands together using the idea in Hoare's 1969 paper [51]: a logic of programs. In this way, we address three goals:

Modular reasoning about relational properties of object-based programs. Such properties include not just equivalence but many others such as noninterference. Conditional equivalence, for example, is needed to justify bug fixes and refactorings (regression verification), taking into account preconditions that capture usage context. Conditional noninterference expresses information flow security policies with declassification; similar dependency properties express context conditions for compiler optimizations. Modular reasoning requires *procedural abstraction,* i.e., reasoning about code under hypotheses in the form of method contracts. It requires *local reasoning*, based on frame conditions. And it requires *data abstraction*, based on program modules and encapsulated data representations.

Automated reasoning. We aim to facilitate verification using what have been called auto-active verification tools [63] like Why3 and Dafny. Users may be expected to provide source level annotations (contracts and data invariants) and alignment hints (to decompose relational reasoning) but are not expected to guide proof tactics or provide full functional specifications. The latter is a key point. It is difficult for developers to formulate full functional specs of applications and libraries, and such specs would often need mathematical types not amenable to automated provers. Experience shows the value of weak specs of input validity and data structure consistency. Frame conditions are particularly useful for the developer and for the reasoning system [49].

Foundational justification. We aim for tools that yield strong evidence of correctness based on accurate program semantics. In this article, we consider sequential programs at the source

level, with idealizations—unbounded integers, heap, stack—that often are used to simplify specs and facilitate automated theorem proving. We carefully model dynamic allocation at the level of abstraction of garbage-collected languages such as Java and ML. The ultimate goal is tools for languages used in practice, for which semantics should be machine-checked and based on the compiler and machine model.

Summary of the state of the art with respect to these goals. To position our work, we give a quick summary; thorough discussion with citations can be found in Section 10.

There are several mature automated verifiers for *unary* (non-relational) verification, including local reasoning by separation logic and by stateful frame conditions ("dynamic frames"), based on SMT solvers and other techniques for proof automation including inference of annotations and decentralized invariants [14, 41] to lessen the need for induction. While abstract data types are commonly supported in specifications, encapsulation of heap structures remains a difficult challenge. For relational reasoning, there has been good progress in automation; this has made clear the need for both lockstep alignment of subcomputations using relational formulas and "asynchronous" alignments using unary reasoning. Automated verifiers have varying degrees of foundational justification, but a standard technique is well established: verification conditions are based on a Hoare logic, which in turn is proved sound.

The semantic theory of data abstraction is well understood for a wide range of languages, mostly focused on syntactic means of encapsulation including type polymorphism but also considering state-based notions like ownership using specialized types or program annotations. These theories account for heap encapsulation and simulation but have not been well connected with general program reasoning: in brief, they say why simulation implies program equivalence but do not say how to prove simulation. Some of this theory has been incorporated in interactive verification tools, for example based on the Coq proof assistant. In such a setting, the powerful ambient logic makes it possible to express all the theory, and recent work includes relational program logics that feature local reasoning and hiding. These works focus on concurrency and higher order programs, and have many complications needed to address those challenges—far from the simplicity of first-order specs supported by automated provers and accessible to ordinary developers.

Our contribution, in a nutshell. This article presents a full-featured, general *relational program logic* that supports *modular reasoning* about both unary and relational properties of object-based programs. The logic formalizes state-based encapsulation and the hiding of invariants and coupling relations, including a proof rule for equivalence of linked programs, which directly embodies the theory of representation independence. The logic uses a form of product program,[1] called "biprogram," to designate alignments of subprograms to facilitate use of simple relational assertions that are amenable to automated proof. The verification conditions are all first-order, without need for inductive predicates, and *amenable to SMT-based automation.* A *foundational justification* is provided: detailed soundness proofs with respect to standard operational semantics.

Outline and reader's guide. Section 2 summarizes the problem, the approach taken, and the contributions of this article. Section 3 presents most of the syntactic ingredients of the unary logic, including effect expressions, unary specs, and correctness judgments. Novel syntactic elements are explained informally via examples and an extended example illustrates encapsulation and modular linking.

[1]Some authors restrict the term "product" to mean a representation that is itself a program. Our usage is looser, encompassing representations like pairs of programs [43] and our custom syntax.

```
module MCell
  class Cell
  meth Cell(c: Cell) /* constructor */
  meth cget (c: Cell) : int /* pure */
  meth cset (c: Cell, v: int)
    requires { c ≠ null }
    ensures { cget(c) = v }
```

Fig. 1. Example interface.

Section 4 first presents the syntactic ingredients of the relational logic—biprograms, relation formulas, relational specs and correctness judgments—and then presents a series of examples to illustrate alignment, relations on heap structures, and relational modular linking.

After Sections 2–4, readers who are not interested in semantic details may wish to skip to Section 6, which presents the rules of the unary logic, and then skip again to Section 8, which presents the rules of the relational logic, including the modular linking rule and its derivation from simpler rules.

Section 5 defines the semantics of programs and unary correctness judgments; it is based on standard small-step semantics, but we need a number of notions concerning agreement and dependency, leading to the novel and subtle semantics of encapsulation. Section 7 gives the semantics of biprograms and relational correctness. Section 9 sketches the use of a prototype tool to evaluate viability of the logic's proof obligations for SMT-based verification. Section 10 surveys related work and Section 11 concludes.

A lengthy Appendix provides proofs and additional details, none of which should be needed to understand the contents of the article. Nonetheless, cross-references to the Appendix are included. There is also a list of metavariables in Table 1 and a glossary of symbols in Table 2 in Appendix E. The article is self-contained but includes some remarks to cater for readers who are familiar with prior work on region logic on which we build.

2 SYNOPSIS

2.1 Modular Reasoning about Relational Properties

To introduce the problem addressed in this article, we begin by sketching Hoare's story about proofs of correctness of data representations. Often a software component is revised with the intent to improve some characteristic such as performance while preserving its functional behavior. As a minimal example, consider this program in an idealized object-based language, with integer global variables x, y:

```
var c: Cell in c := new Cell; x := x+1; cset(c,x); y := cget(c)
```

It is a client of the interface in Figure 1. An obvious implementation of the module[2] is for class Cell to declare an integer field val that stores the value. Suppose we change the implementation: store the negated value, in a field named f, and let cget return its negation. Client programs like the one above should not be affected by this change, at the usual level of abstraction (e.g., ignoring timing). To be specific, we have equivalence of the two programs obtained by linking the client with one or the other implementation of the module. (Equivalence means equal inputs lead to equal outputs.) This has nothing to do with the specific client. The point of data abstraction is to free the client programmer from dependence on internal representations, and to free the library programmer from needing to reason about specific clients.

[2]Classes are instantiable. For our purposes, modules are static [9, 77], like packages in Java and other languages.

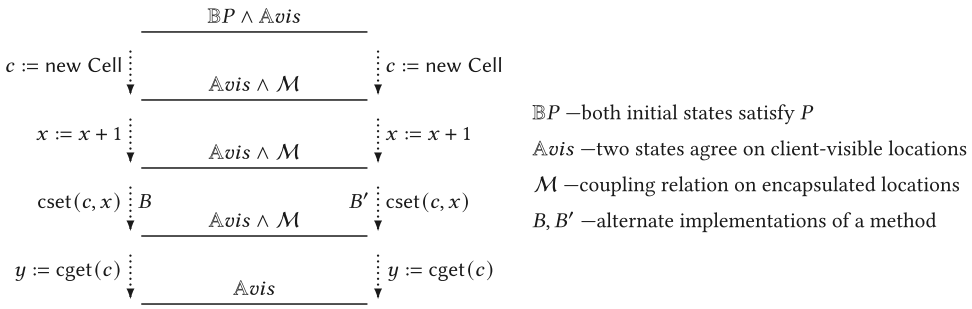


Fig. 2. Two executions, with relations between aligned points.

The (relational) reasoning here is familiar in practice and in theories of ***representation independence***. There is a coupling relation that connects the two data representations; in this case, for corresponding object references o, o' of type Cell,

$$\text{the value of field } o'.\text{f is the negation of } o.\text{val}. \tag{1}$$

This relation is maintained, by paired execution of the two implementations, for each method of the module and for all instances of the class. The fields are encapsulated within the module, so a client can neither falsify the relation nor behave differently from related states, since the visible part of the relation is the identity.

Figure 2 depicts steps of two executions of the example client, linked with alternate implementations of the methods it calls. The top line indicates a relation between the initial states of the left and right executions. The client's precondition P holds in both ($\mathbb{B}$), and the initial states agree ($\mathbb{A}$) on the part of the state that is client-visible. Unknown to the client, the module coupling relation $\mathcal{M}$ is established by the constructors and can be assumed in reasoning about the calls, provided the method's implementations preserve the relation. A client step, like x:=x+1 here, should preserve $\mathcal{M}$ for reasons of encapsulation. The bottom line indicates agreement on the final result. Each method has alternate implementations; the ones for cset are labelled (as B, B') for expository purposes.

In this work, we introduce a logic in which one can specify relational properties such as the preservation of a coupling relation by the two implementations B, B', as well as equivalence of the two linked programs for a client C. Moreover, the equivalence can be inferred directly from the preservation property. Equivalence is expressed in local terms, referring just to the part of the state that C acts on: In the example client program, the pre-relation is agreement on the value of x and the post-relation is agreement on y. If C is part of a larger context, then a relational frame rule can be applied to infer that relations on separate parts of the state are also maintained by C as discussed later.

Encapsulation. The above reasoning depends crucially on encapsulation, and many programming languages have features intended to provide encapsulation. In unary verification, encapsulation serves to protect invariants on internal data structures. It is well known, and often experienced in practice, that references and mutable state can break encapsulation in conventional languages like Java and ML. There has been considerable research on methodologies using type annotations and assertions to enforce disciplines including ownership for the sake of encapsulation and local reasoning. This work focuses on heap encapsulation, without commitment to any specific discipline, but provides a framework in which such disciplines can be used.

In this article, encapsulation is at the granularity of a module, not a class or object. Thus, the implementation of a method cswap(c, d: Cell) that swaps the values of two cells can exploit that

the cells have the same internal representation. However, it is often useful for each instance of an abstraction, say a cell or a stack, to "own" some locations that are separate from those of other instances, so we can do framing at the granularity of an instance. This is manifest in frame conditions, as we will see for cset, and it is also manifest in invariants. For example, a module for stacks implemented using linked lists has the invariant that distinct stacks use disjoint list nodes.

Let us sketch how encapsulation and module invariants can be formalized in a unary logic. The linking of a client C with a method implementation B can be represented by a simple construct, let $m = B$ in C that binds B to method name m. (For clarity, we ignore parameters and consider a single method rather than simultaneous linkage of several methods.) The *modular linking rule* looks as follows, where we use notation $C : P \rightsquigarrow Q$ instead of the usual Hoare triple $\{P\}C\{Q\}$ (for partial correctness)[3]:

$$\frac{m : R \rightsquigarrow S \vdash C : P \rightsquigarrow Q \qquad m : (R \wedge I) \rightsquigarrow (S \wedge I) \vdash B : (R \wedge I) \rightsquigarrow (S \wedge I)}{\vdash \text{let } m = B \text{ in } C : P \rightsquigarrow Q}. \tag{2}$$

The first premise says C is correct under the hypothesis that m satisfies the spec $R \rightsquigarrow S$. (The general form allows other hypotheses, which are retained in the conclusion.) The second premise says the body B of m satisfies a different spec, $R \wedge I \rightsquigarrow S \wedge I$ (and assumes the same, as needed in case of recursive calls to m in B). The spec $R \rightsquigarrow S$ should be understood as the interface on which C relies—indeed, C is *modularly correct* in the sense that it satisfies its spec when linked with any correct implementation of m, so C never calls m outside its specified precondition R. In the verification of B, the internal invariant I can be assumed initially and must be reestablished. The invariant is *hidden* from clients of the module.

As displayed, rule (2) is obviously unsound, because C might write a location on which I depends and then call m in a state where I does not hold. The idea is to prevent that by encapsulation, for which we are required to

(E1) delimit the module's "internal locations,"
(E2) ensure that the module's private invariant I depends only on those locations,
(E3) frame the effects of C and ensure its writes are separate from the internal locations, and
(E4) arrange that I is established initially (e.g., by module initialization and object constructors).

Relational modular linking. Encapsulation licenses more than just the hiding of invariants. Once the requirements (E1)–(E4) are met in a way that makes Equation (2) sound, we can contemplate the adaptation of Equation (2) to relational reasoning and in particular proving equivalence of two linkages, let $m = B$ in C and let $m = B'$ in C. The labels (E1)–(E4) are used to also refer to the requirements as adapted to relational reasoning.

The two linkages cannot be expected to behave identically: B and B' typically have different internal state on which they act differently. What can be expected is that from initial states that are equivalent in terms of client-visible locations, the two linkages yield final states that are equivalent on visible locations, as indicated by the deliberately vague "*vis*" in Figure 2. We say equivalent states, because B and B' may do different allocations; so, the resulting heap structure should be isomorphic but need not be identical. (For many purposes one wants to reason at the source language level of abstraction, ignoring differences due to timing, code size, and absolute addresses; that is our focus.) Given that we have framing (E3), it suffices to establish "local equivalence" in the sense that initial agreement on locations readable by C leads to final agreement on locations writable by C—and on freshly allocated locations. Agreement on other visible locations should then follow.

[3]Following O'Hearn et al. [9, 77], we use the term modular for information hiding, not just procedural abstraction.

We write $(B|B') : \mathcal{R} \approx\!> \mathcal{S}$, for relations $\mathcal{R}$ and $\mathcal{S}$ on states, to say that pairs of terminated executions of programs B and B', from states related by $\mathcal{R}$, end in states related by $\mathcal{S}$. For example, $(C|C) : \mathbb{A}x \approx\!> \mathbb{A}y$ says two runs of C from states that agree on the value of x end in states that agree on the value of y. The relational generalization of Equation (2) is a *relational modular linking rule* of this form:

$$\frac{m : R \rightsquigarrow S \vdash C : P \rightsquigarrow Q \qquad m : \ldots \vdash (B|B') : \mathbb{B}R \wedge \mathbb{A}in \wedge \mathcal{M} \approx\!> \mathbb{B}S \wedge \mathbb{A}out \wedge \mathcal{M}}{\vdash (\mathsf{let}\ m = B\ \mathsf{in}\ C \mid \mathsf{let}\ m = B'\ \mathsf{in}\ C) : \mathbb{B}P \wedge \mathbb{A}vis \approx\!> \mathbb{B}Q \wedge \mathbb{A}vis}. \quad (3)$$

The first premise is unary correctness of C assuming the interface spec of m as in rule (2). The conclusion of Equation (3) expresses local equivalence of the two linkages, under precondition P. The second premise relates the two implementations B and B' and is meant to say that if the client-visible "input" locations are in agreement then the resulting visible outputs are in agreement. In addition, a relation $\mathcal{M}$ is conjoined to the pre- and postcondition. A coupling relation $\mathcal{M}$ usually has three conjuncts: it says the left state satisfies some invariant I on the internal state used by B, the right state satisfies invariant I' on the internal state used by B', and there is some connection between the internal states. (We often use "left" and "right" in connection with two programs, states, or executions to be related.) The hypothesis for m in the second premise is the same spec as proved for $(B|B')$, following the pattern in Equation (2). We elide that hypothesis for readability: relational reasoning involves two of everything and the notations quickly become cluttered! As with the modular linking rule (2), the relational modular linking rule (3) is unsound unless we satisfy requirements (E1)–(E4). For relational reasoning, (E2) and (E4) are adapted to relations, and (E3) is strengthened to ensure separation for reads, as one would expect to avoid dependence on internal representations.

Alignment. One technique for proving some relation on final states is to leverage functional specs: a strong constraint on the output values, such as $out = f(in)$ for some mathematical function f, entails that initial agreement on in leads to final agreement on out. But the need to find and prove functional specs can often be avoided through judicious *alignment* of intermediate points in execution. This technique is used to prove soundness of Equation (3). To illustrate, consider an instantiation of the general rule in which the three methods in Figure 1 are bound simultaneously (cset, cget, and the Cell constructor). We show that two executions of the example client can be aligned as in Figure 2, with the indicated relations holding at the aligned points. After the two constructor calls, the resulting states should agree on visible locations and be related by the coupling, according to the premise proved for the constructor. From any pair of states related by $\mathbb{A}x \wedge \mathbb{A}c \wedge \mathcal{M}$, two executions of x:=x+1 maintain agreement on visible variables including x, and according to (E3) this step in the client code is not touching internal locations on which $\mathcal{M}$ depends, so $\mathcal{M}$ continues to hold. From any pair of states related by $\mathbb{A}vis \wedge \mathcal{M}$, a pair of calls to cset results in states related, by the premise for cset. Similarly for cget. In fact, $\mathcal{M}$ relates the final states in Figure 2, but we omit it there, to emphasize that it is an ingredient of proof rather than the property of ultimate interest.

In a good alignment, most of the intermediate relations are agreements ($\mathbb{A}$) that amount to simple equalities connecting values in locations of the two states. Finding and exploiting good alignments is essential to leverage automatic theorem provers. For cset(c,v) in Figure 1, the first implementation is c.val:= v; return c.val and the second is c.f:= −v; return −c.f. If we align their executions at the semicolons, then we can assert the coupling relation Equation (1) at that point, by unary reasoning about the effect of the two field updates. Again by unary reasoning about the return expressions we get that the same values are returned, as needed for the final agreement on visible variable

y. Alignment does not eliminate the need for unary/functional reasoning, but rather reduces it to small program fragments for which precise semantics can be computed by a theorem prover.

Alignment can be expressed by means of a product program, that is, a program, or some kind of automaton, whose executions correspond to paired executions of the given programs. We call this well known technique the *product principle*: to prove a correctness judgment $(C|C') : \mathcal{R} \approx\!> \mathcal{S}$ relating programs C and C', it suffices to prove the spec for some product program whose executions cover the executions of C and C'.

To emphasize the role of alignment, we consider another example, not about representation independence but about secure information flow. The following program acts on a linked list of integer values, where each node has a boolean field, pub, meant to indicate that this value is public:

$$sumpub: \ \texttt{s:=0; p:=head; while p} \neq \texttt{null do if p.pub then s:=s+p.val fi; p:=p.nxt od.} \tag{4}$$

We want to specify and prove that this does not reveal any information about non-public values. Suppose we can define $listpub(p)$ to be the mathematical list of public values reached from p. To express that the final value of s depends only on public elements of the list we use the spec $\mathbb{A}listpub(p) \approx\!> \mathbb{A}s$. The program satisfies the unary spec $true \rightsquigarrow s = sum(listpub(head))$, and any program that satisfies this must also satisfy $\mathbb{A}listpub(head) \approx\!> \mathbb{A}s$. But, we can prove the relational spec without recourse to the unary spec. At points in execution where two runs have passed the same number of public nodes, the relation $\mathbb{A}s \wedge \mathbb{A}listpub(p)$ holds; this suggests an alignment where it suffices to use relational invariant $\mathbb{A}s \wedge \mathbb{A}listpub(p)$. Adding the same value to s on both sides maintains $\mathbb{A}s$ and there is no need to reason that s is the sum of previously traversed public values. The same relational invariant should suffice if sum is replaced by a more complicated function. The alignment can be described as follows: consider an iteration just on the left (respectively, right), if the next left (respectively, right) node is not public; and simultaneous execution of the body on both sides, if both next nodes are public.

We cannot in fact define *listpub* as a function of p, owing to the possibility of cycles in the heap. Instead, we use an inductive relation when we work out the details of this example Section 4.5.

Summary of ingredients needed. To achieve the three goals in Section 1, we need:

- A unary logic of functional correctness under hypotheses (for procedure-modularity), that supports framing (for local reasoning) and encapsulation (for hiding and abstraction). To support a wide range of programming patterns, the logic should support reasoning in terms of encapsulation at the granularity of an object that "owns" some internal state, say representing an instance of an ADT. It should also support reasoning at the granularity of a module, where many instances of multiple classes may share the internal representation. It should encompass flexible patterns of sharing in data structures and between clients and components.
- A relational logic with framing and encapsulation, in which the relation formulas in specs and intermediate assertions are sufficiently expressive to describe data structures with dynamically allocated objects. Agreement "modulo renaming" is needed to reason at the level of abstraction of Java/ML, which provide reference equality and preclude arithmetic comparisons and operations on pointers, to express local equivalence and other relations. The logic must provide means to reason with alignments that admit simple intermediate relations. Examples like the *sumpub* program in Equation (4) show the need to use state-dependent alignments in addition to alignments of control structure.

These ingredients need to be provided in ways that facilitate verification tools that leverage automated provers especially SMT solvers. Reasoning under hypotheses is straightforward to implement, but effective expression of specs and alignment is less obvious.

2.2 An Approach Based on Region Logic

Our relational logic is based on prior work in which ghost state is used in frame conditions to describe sets of heap locations. This approach, dubbed *dynamic frames* [54], has been shown to be amenable to SMT-based automated reasoning in verification tools [62, 81, 87, 91], and shown to be effective in expressing relations on dynamically allocated data structures [3, 11]. In particular, we build on a series of articles on ***region logic*** **(RL)**; it provides a methodologically neutral basis for heap encapsulation with sufficient generality for sequential first-order object-based programs featuring callbacks between modules. We refer to key articles as RLI [14], RLII [9], and RLIII [12], and summarize key ideas in the following.

Framing. In current tools, the most common form of frame condition is a "modifies clause" that lists some expressions, meant to designate the writable locations. A reads clause is similar. In the formalization of RL, specifications are written in the compact form $pre \rightsquigarrow post\ [frame]$ where the effect expressions in the frame condition are tagged by keywords wr and rd to designate writables and readables. We use rw to abbreviate the possibility to both read and write. In this work, a ***region*** is a set of object references. For example, a possible spec of cset(c,v) is $c \neq \mathsf{null} \rightsquigarrow cget(c) = v\ [\mathsf{rw}\ \{c\}`\mathsf{any}]$, where the postcondition refers to the mathematical interpretation of the pure method cget (as in RLIII). The singleton region $\{c\}$ is used in the frame condition. In the *image expression* $\{c\}`\mathsf{any}$, the token any is a data group [64] that abstracts from field names. Concrete field names can also be used in image expressions, e.g., $\{c\}`val$. This example designates a single location, which may as well be written $c.val$. But the image notation can be used for larger sets of heap locations. For variable $r$ of type region, $r`val$ designates the set of val fields of all Cell objects in r. So $\mathsf{rd}\ r`val$ in a frame condition allows any of these fields to be read.

Following separation logic, RL features local reasoning in the form of a *frame rule*, but achieves this with ordinary first-order assertions. For an example, strengthening the precondition of cset(c,v) gives $c \neq \mathsf{null} \wedge d \neq c \rightsquigarrow cget(c) = v\ [\mathsf{rw}\ \{c\}`\mathsf{any}]$. The frame rule lets us add $d.val = z$ to the pre- and postcondition. Why? Because the condition $d.val = z$ cannot be falsified: the writes allowed by the frame condition are separate from what is read[4] by the formula $d.val = z$. In case of the variables d and z, this is a matter of checking that d and z are not writable. Distinctness of field names can be used similarly. But here, $\mathsf{rw}\ \{c\}`\mathsf{any}$ allows that $c.val$ can be written and val also occurs in the formula $d.val = z$. Separation holds, because the regions $\{c\}$ and $\{d\}$ are disjoint, written $\{c\}\ \#\ \{d\}$, which follows from precondition $d \neq c$. As in the frame rule of separation logic [76], this reasoning is inherently state dependent; separation would not hold if variables d and c held the same reference. Our frame rule has this form:

$$\begin{array}{l}\text{from}\quad C : P \rightsquigarrow Q\ [\varepsilon]\quad\text{infer}\quad C : P \wedge R \rightsquigarrow Q \wedge R\ [\varepsilon],\\ \text{provided that locations read by } R \text{ are separate from locations writable according to } \varepsilon.\end{array} \tag{5}$$

In the frame rule of RL, separation is expressed by a conjunction of set disjointness formulas derived syntactically from the frame condition ε and the read effects of R. In this example, the relevant effects are $\mathsf{wr}\ c.val$ and $\mathsf{rd}\ d.val$ and there is a single disjointness formula: $\{c\}\ \#\ \{d\}$. This formula is obtained by applying the separator function $\cdot/.$ introduced later, in Figure 11.

Encapsulation. RLII features *dynamic boundaries*, in which the idea of dynamic frame is adapted to encapsulation for module interfaces. The dynamic boundary of a module is simply an effect

[4]For a formula's meaning to depend on a location is different from a program reading the location during execution. However, these two notions have closely related extensional semantics based on agreement between states. So, following the RL articles, we use the terminology and notation of read effects for both.

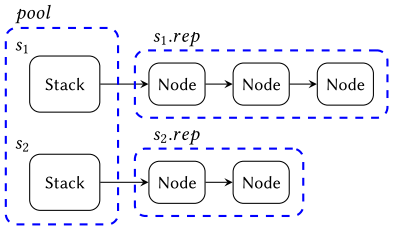


Fig. 3. The pool and rep idiom.

expression that designates the locations meant to be internal to the module. Technically, it is a read effect, in keeping with its role to cover the footprint of the module invariant. In addition to the usual meaning of a partial correctness judgment, there is an additional obligation: the program must not *write* locations within the boundary of any module other than its own module.

For the example module MCell, the dynamic boundary (omitted from Figure 1) is formulated in terms of a ghost variable, *pool*, of type region. The postcondition of the Cell constructor says the new cell is added to *pool*. The boundary is rd *pool*, rd *pool*‘any, so clients must not write the variable *pool* or any field of an object in *pool*. One could as well achieve this effect using module-scoped field names, so let us briefly consider a less degenerate example: a module for stacks.

In addition to ghost variable *pool* containing all instances of the stack class, that class would have a ghost field *rep* of type region. In an implementation using linked lists, each stack's list nodes would be in its *rep*, and the module invariant would specify some "object invariant" for each stack together with its nodes. This is depicted in Figure 3. In an implementation using arrays, *rep* would contain the stack's array, and the module invariant would express some condition that holds for each stack object and its array. Of course there is a single interface for the module. Method frame conditions will refer to *pool* and *rep*, and not expose implementation details. To facilitate per-instance framing, an invariant like $s \neq t \Rightarrow s.rep\#t.rep$ is used, which says the representations for distinct stacks are disjoint. A suitable dynamic boundary is rd *pool*, rd *pool*‘any, rd *pool*‘*rep*‘any. It designates fields of the stack objects in *pool* and also fields of all their rep objects. (Array slots can be viewed as fields.) The mentioned invariant enables use of the frame rule to consider updates of a single instance, and it is suitable to be included in the module interface for use by clients. (Either as explicit conjunct in method pre- and postconditions, or declared as a *public invariant* for syntactic sugar.) For example, s.push(n) writes *s.rep*‘any; in states where $s \neq t$ this preserves the value of t.top(), which reads *t.rep*‘any—and preservation holds in virtue of frame conditions, without recourse to postconditions that specify functional behavior.

In summary, a module interface comprises a collection of method specs, and a dynamic boundary. A module implementation maintains an internal invariant I, the footprint of which should be framed by the boundary. The invariant I should be such that it follows from the initial conditions of the main program. For example, universal quantification over elements of *pool* holds when *pool* is empty. An alternate approach is to require clients to call a module initializer.

Modular linking. Following the lead of O'Hearn et al. [77], the logic in RLII derives a modular linking rule like Equation (2) from two simpler rules: An obviously-sound rule for the linking construct (let $m = B$ in C) and a ***second-order frame (SOF)*** *rule* that accounts for hiding of invariants on encapsulated state. A minimalistic formalization of modules is used, to keep the focus on the main ideas. The unary correctness judgment takes the form $\Phi \vdash_M C : P \rightsquigarrow Q\,[\varepsilon]$ with M the name of the module in which C is to be used. It says that, under hypotheses Φ and precondition P, command C stays within the effects ε and establishes Q if it terminates—and in addition, C *respects* the

boundaries of any modules in Φ other than its own module M. This formalizes requirement (E3). In RLII, "respect of dynamic boundaries" means not writing locations inside them. In the present article, we must strengthen respect to prohibit reading, to ensure that C has no dependency—neither reads nor writes—on the internal representation of modules other than its own.

2.3 Relational Region Logic

Our relational specs have the form $\mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon']$ where $\mathcal{P}$ (respectively, $\mathcal{Q}$) is the relational pre-(respectively, post-)condition. There is a separate frame condition ε for the left execution and ε' for the right. Often those are the same, in which case we abbreviate as $\mathcal{P} \approx\!> \mathcal{Q}\,[\varepsilon]$. The meaning of frame conditions and encapsulation is the same as in the unary logic. Leaving effects aside, there are several ways one could interpret a spec $(C|C') : \mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon']$ in regards to termination. All ways consider a pair of initial states, say σ, σ', that satisfy $\mathcal{P}$. The "$\forall\exists$ interpretation" says that for every execution of C from σ, terminating in a state τ, there is an execution of C' from σ' that terminates in a state related to τ by $\mathcal{Q}$. The $\forall\exists$ interpretation asserts relative termination and caters for nondeterminacy. The "$\forall\forall$ interpretation" was already mentioned just before (3): every pair of terminating runs of C and C' from $\mathcal{P}$-related states end in $\mathcal{Q}$-related states. The $\forall\forall$ form is fine for deterministic programs, which is what we consider, and it is simpler, so we use it.

For relation formulas, we build directly on image expressions. Agreements are interpreted in terms of a partial bijection between the dynamically allocated references of the left and right states, as commonly used to account for bijective renaming of references at the Java/ML level of abstraction [7, 8, 23, 27]; we call these *refperms*. For region expression G, the relation $\mathbb{A}G\text{`}f$ asserts agreement on f-fields for objects in G that correspond according to the refperm. We do not require every allocated reference to be in the refperm: this is important, to specify relational properties that allow differences in allocation behavior. Examples of such differences include internal data structures and reasoning about secure information flow (under low branch condition, allocated locations can be added to the refperm, but not under high branch condition).

We formulate the logic in terms of an explicit representation for product programs that designate alignments. The biprogram form $(C|C')$ indicates no alignment except for the initial and final states. Other biprogram forms express, for example, that iterations of a loop are to be aligned in lockstep, or conditionally as needed for the *sumpub* example (4). For the implementations of cset, the alignment described earlier is expressed as (c.val:= v | c.f:= −v); (return c.val | return −c.f).

A judgment for $(C|C')$ directly entails the expected relation between unary executions of commands C and C' (as confirmed by our adequacy theorem). The choice to use a different alignment of C with C' is formalized by an explicit proof rule. The rule is formulated in terms of a *weaving relation* that connects a biprogram with a more tightly aligned version, typically chosen, because it admits use of simpler relational assertions. The rule says that properties of the woven program hold also for $(C|C')$.

Given that we confine attention to sequential code, it seems natural to expect that programs are deterministic, but we also aim for reasoning at the source code level abstraction—for which determinacy is unrealistic owing to dynamic allocation! The behavior of an allocator typically depends on things that are not visible at the source level. There is no need to make unrealistic assumptions. Our program semantics allows that the allocator may be nondeterministic (while not assuming that it is "maximally nondeterministic" as often done in the literature). Our program semantics is *quasi-deterministic* in the sense that outcomes are unique up to bijective renaming of references. Our relation formulas do not allow pointer arithmetic or comparisons other than equality, so they are invariant under renaming. These design decisions entail some complications in the technical development, but ensure that interesting programs do provably satisfy expected $\forall\forall$ properties.

As already mentioned, the unary modular linking rule (2) is derived (in RLII) from two simpler rules: a basic linking rule, where assumed and proved specs match exactly, together with a second-order frame rule. Our novel relational modular linking rule (3) is derived from a relational linking rule, a relational second-order frame rule, and a third rule. The third rule lifts a unary correctness judgment to a relational judgment that says a program is locally equivalent to itself. For this to be proved, it is stated in a stronger form: a program can be aligned with itself in lockstep such that local equivalence holds at each intermediate step.

As for the goal of foundational justification, our approach is to work directly with a conventional operational semantics for unary correctness, for which we formulate a semantics of encapsulation. The biprogram semantics is based directly on that, so that soundness for rules in the relational logic has a direct connection—adequacy theorem—to unary semantics. One benefit from carrying out the development in terms of this elementary semantics is that one can see that most of the soundness proofs can be adapted easily to total correctness (both runs always terminate) and to relative termination (right run terminates whenever left does).

2.4 Contributions

We highlight the following contributions.

A unary logic for modular reasoning about sequential object-based programs using first-order assertions. The key contribution and most difficult definition to get right is the extensional semantics of encapsulation, which is part of the meaning of correctness judgments. Small-step operational semantics is used, so we can define what it means for a given step to be outside the boundaries of all modules but its own. We build on the semantics in RLII but completely revamp it to handle encapsulation of reads in addition to writes. Dynamic boundaries are taken from RLII; most of the proof rules of RLII need little or no revision, but they must all be re-proved for the new semantics. Owing to the need for quasi-determinacy (for $\forall\forall$ extensional semantics of read effects), the new semantics of hypothetical judgments quantifies over possible denotations (called *context interpretations*) rather than a single "least refined" denotation as in RLII and in O'Hearn et al. [77]. We present detailed soundness proofs of the key rules (Theorem 6.1).

A relational logic. The logic relies on unary judgments for reasoning about atomic commands and for enforcing encapsulation. Relational assertions are first-order formulas. Our presentation focuses on data abstraction, because this is the first relational logic to embody representation independence as a proof rule using only first-order means. But the logic is general, with a full range of rules that facilitate reasoning with convenient alignments.

We present detailed soundness proofs of the key rules (Theorem 8.1). Formally, judgments of the relational logic give properties of biprograms; the adequacy Theorem 7.11 connects those properties with the expected properties in terms of paired unary executions in standard semantics (the product principle).

Demonstration of suitability for automation via case studies in a prototype relational verifier. The prototype translates biprograms and verification conditions specific to our logic, which are all first-order, into Why3 code and lemmas, proved using SMT solvers (why3.lri.fr). The modular linking rules (unary and relational) are implemented by generating suitable Why3 specs for the programs involved. The case studies include noninterference, program transformations, and representation independence.

2.5 About the Proofs

The most difficult technical result is the *lockstep alignment lemma* (Lemma 8.9). It brings together the semantics of encapsulation in the unary logic, which involves a single context interpretation,

with the semantics of relational correctness—which involves three context interpretations, to account for un-aligned calls as well as aligned calls and relational specs.

The direct use of small-step semantics makes for lengthy soundness proofs that require, in some cases, intricate inductive hypotheses. But transition semantics is a critical ingredient for a first-order definition of heap encapsulation. It was quite difficult to arrive at rules for relational linking and second-order framing that are provably sound. Several variations on the semantics of encapsulation turned out to be sound for the unary linking and second-order frame rules but failed to validate a sufficiently strong lockstep alignment property on which relational linking can be based.

Aside from lockstep alignment, the soundness proofs for linking rely on denotational semantics, which in turn relies on quasi-determinacy. This property is also used to establish embedding/projection results on which the adequacy theorem is based.

The semantics of correctness judgments is extensional in the sense that it refers only to behavior in a standard transition semantics—no instrumentation artifacts. Like in RLII, it does rely on use of transition semantics to express that control is currently within a specific module and outside the boundaries of other modules in scope. This affects which program transformations are correctness-preserving; more on this in Section 8.6.

Once the right definitions, lemmas, and induction hypotheses have been determined, the soundness proofs go by induction on traces, with many details to check. We relegate them to appendices.

2.6 Current Limitations

The formal development omits some features that were handled in the prior works on which we build: parameters, private methods, constructor methods, pure methods for abstraction in specs. These are all compatible with the formal development; all are implemented in the prototype and used in exposition. The theory is compatible with standard forms of encapsulation based on scoping mechanisms (e.g., module scoped variables), which for practical purposes should be leveraged as much as possible; for simplicity, we refrain from formalizing such mechanisms.[5] The prototype also supports public invariants; as noted in connection with the stack example, these are important for client reasoning about boundaries using patterns like ownership. Public invariants need not be formalized in the theory, as they can be explicitly included in method specs.

The simplicity of our semantic framework (e.g., standard semantics of formulas and programs) may facilitate foundational justification of a verifier, but we have not formally proved the correctness of our prototype.

There are two technical limitations. First, the semantics of encapsulation and the proved rules handle collections of modules with both import hierarchy and callbacks. But the key rules for relational linking and **relational second-order framing (RSOF)** only handle simultaneous linking of a collection of modules. This is enough to model linking as implemented in a verifier. However, one may hope for a theory that accounts for distinct inference steps that successively link different layers of hierarchy, as in our unary logic. To achieve this, the lockstep alignment lemma needs to be strengthened to ensure agreements for already-linked methods. This requires to further complicate an already intricate theory. In this article, we just sketch the issue (Section 8.5).

Second, the current formulation has a technical condition (boundary monotonicity) that prevents release of encapsulated locations, in the sense of reasoning with specs that describe outward ownership transfer. (Inward transfer is fine.) Modules can create new objects for clients, as in the shared handle objects for priority queues, one of our running examples. But a location that has

[5]Specs involving explicit footprints are more verbose than those based on separation logic, and our minimalist formalization of modules increases verbosity. This article does not propose concrete syntax for practical use, but the issue is addressed in some related work (Section 10).

```
class Pnode { val: int; key: int; sibling: Pnode; child: Pnode; prev: Pnode; }

class Pqueue { head: Pnode; size: int; ghost rep: rgn; }

meth Pqueue (self:Pqueue) =
  self.rep := {null}; pool := pool ∪ {self};

meth insert (self:Pqueue, val:int, key:int): Pnode =
   result := new Pnode(val, key);
   self.rep := self.rep ∪ {result};
   if self.head = null then self.head := result;
   else self.head := link(self, self.head, result) fi;
```

Fig. 4. Excerpts of priority queue (PQ) implementation (in the syntax of our prototype).

been within the boundary must stay there. Overcoming this restriction, or finding idiomatic specification patterns that dodge it, is left to future work. Both inward and outward transfer are possible in RLII (an example is in Section 2.2 of that article).

Addressing the limitations is the subject of ongoing and future work.

3 PROGRAMS: THEIR SYNTAX AND SPECIFICATIONS

This section defines the syntax of programs and their unary specifications and correctness judgments. Sections 3.1–3.4 collect together almost all the syntactic forms and definitions concerning syntax, using a few examples to explain unusual things. Section 3.5 gives more holistic examples to illustrate how the syntax is used and why we need various syntactic elements, focusing on how requirements (E1)–(E4) for encapsulation in Section 2.1 are expressed and checked.

3.1 Programs and Typing

A running example is introduced in Figure 4. We consider the priority queue module PQ, which exposes a class whose instances represent priority queues that store integer values and priorities, referred to as "keys" (smaller key means higher priority) [98]. Our implementations (based on Reference [98]) use pairing heaps, where each queue contains a *head* field that points to a Pnode object and each Pnode contains *sibling*, *prev*, and *child* fields that point to other Pnodes. The *rep* field of a queue is used to hold references to the objects notionally owned by the queue.

The syntax of programs in our formal development is in Figure 5. The grammar includes biprograms, to which we return in Section 4. Field read and write commands are written with dereferencing implicit, as in Java (though using the symbol :=), and are desugared to have a single heap access that simplifies proof rules. The let construct, featured in the modular linking rule (2), represents scoped method declarations.[6] Some examples, like Figure 4, use the syntax of our prototype, in which keyword meth corresponds to the let construct. Examples use some syntax sugars implemented in our prototype, e.g., invocation of method link in an update of field self.head (Figure 4). A method named after a class (e.g., Pqueue) is meant to be used as a constructor, i.e., invoked on a newly allocated object, the fields of which are initialized with default values (null for classes, ∅ for regions).

To lessen the need for uninteresting transitions in program semantics, we equate certain syntactic forms. For example, there is no transition from (skip; C) to C, because we consider them to be the same syntactic object, see Figure 6. Working with syntax trees up to (i.e., quotiented by)

[6]We use the short term "method" for what should properly be called procedure. The term "method" usually implies dynamic dispatch, which is beyond the scope of this article.

$m \in MethName \qquad x, y, r \in VarName \qquad f, g \in FieldName \qquad K \in DeclaredClassName$

(Classes)		::= class K $\{\overline{f{:}T}\}$ (overline indicates finite lists)
(Types)	T	::= int | bool | rgn | K (and math types, in specs and ghost code)
(Prog. expr.)	E	::= x | n | null | $E \otimes E$ where n is in $\mathbb{Z}$ and $\otimes$ is in $\{=, +, -, *, \geq, \wedge, \ldots\}$
(Region expr.)	G	::= x | $\varnothing$ | $\{E\}$ | G`f | G/K | $G \otimes G$ where $\otimes$ is in $\{\cup, \cap, \setminus\}$
(Expressions)	F	::= E | G
(Atomic com.)	A	::= skip | $m()$ | $x := F$ | $x :=$ new K | $x := x.f$ | $x.f := x$
(Commands)	C	::= A | let $m() = C$ in C | if E then C else C | while E do C | $C\,;C$ | var $x{:}T$ in C
(Biprograms)	CC	::= $(C\|C)$ | $\lfloor A \rfloor$ | let $m() = (C\|C)$ in CC | var $x{:}T\|x{:}T$ in CC | $CC\,;CC$ | if $E\|E$ then CC else CC | while $E\|E \cdot \mathcal{P}\|\mathcal{P}$ do CC

Syntax sugar: while $E|E'$ do CC abbreviates while $E|E' \cdot \text{false}|\text{false}$ do CC.
Identifiers: B, C, D for commands, BB, CC, DD for biprograms.

Fig. 5. Programs and biprograms. For relation formulas $\mathcal{P}$ see Figure 14.

$$(\text{skip}; C) \equiv C \quad (C; \text{skip}) \equiv C \quad (C_0; C_1); C_2 \equiv C_0; (C_1; C_2)$$
$$(\text{skip}|\text{skip}) \equiv \lfloor \text{skip} \rfloor \quad \lfloor \text{skip} \rfloor; CC \equiv CC \quad CC; \lfloor \text{skip} \rfloor \equiv CC \quad (CC_0; CC_1); CC_2 \equiv CC_0; (CC_1; CC_2)$$

Fig. 6. Syntactic equivalence $\equiv$ of programs and biprogams.

syntactic equivalence is done in the previous RL articles and elsewhere.[7] We sometimes use the symbol $\equiv$ for equality of other syntactic forms, like variables, just to emphasize that they are syntactic.

Programs and specs are typed in a conventional way. A ***typing context*** Γ maps variable names to data types and method names to the token meth, written as usual as lists, e.g., $x{:}T, y{:}T, m{:}\text{meth}$. (In the formalization, we omit method parameters and results.) Various definitions refer to a typing context typically meant to be the global variables, including ghost variables, which may be of type rgn (region). We do not formalize ghost variables as such [14, 42].

The idea of ghost code is to instrument a program with extra state for the sake of reasoning, in such a way that the termination and behavior of the original program is not affected. This can be formalized in terms of a rule for elimination of ghost state [14, 42, 78]. We refrain from doing so in this article; the additions would not be illuminating.

A class is just a named record type. In the formal development we assume an ambient ***class table*** that declares some class types and the types of their fields. For simplicity this has global scope. We assume that field names in different class declarations are distinct, so any declared field f determines a unique class, $DeclClass(f)$, that declares it, and also a type, which we write $f : T$.

Section 2.2 introduced the region expressions used in frame conditions. In addition to (mutable) variables of type region, there are set operations like union, singleton, subtraction ($\setminus$), and image expressions. The expression $\{x\}$ denotes the singleton set containing the value of x. For G a region expression, the image expression G`$f$ is the empty region if $f$ : int. If $f$ is of some class type, then $G$`f is the set of current values of f-fields of objects (i.e., object references) in G. For f of type rgn the image is the union of the field values. For example, in the idiom using global variable $pool$: rgn containing some objects with field rep : rgn, the image $pool$`rep is the union of their rep fields. The type restriction expression G/K denotes the elements of G of type K (which excludes null).

As usual in program logics, field access and update is limited to the primitive forms $x := y.f$ and $x.f := y$. In specs and ghost code, a dereference chain like $x.f.g.h$ (for reference type fields) can be expressed by the region expression $\{x\}$`$f$`g`h; if x is null the value is the empty set.

[7] See, e.g., Reference [6]. We use the symbol $\equiv$ because it is used for structural congruences in process algebra, which have the same purpose of streamlining the transition system.

$$\frac{\Gamma \vdash E : K}{\Gamma \vdash \{E\} : \mathsf{rgn}} \qquad \frac{\Gamma \vdash G : \mathsf{rgn}}{\Gamma \vdash G\text{`}f : \mathsf{rgn}}$$

Fig. 7. Region expression typing (selected).

Owing to the simple model of classes, the notation $G\text{`any}$ can be defined as shorthand for $G\text{`}\overline{f}$ where $\overline{f}$ is the list of all field names. An implementation can support user-defined data groups, which can be used to abstract from specific sets of fields [64].

The typing rules for expressions and commands are straightforward and omitted, with the exception of those in Figure 7. We highlight those, because we allow f in an image expression $G\text{`}f$ to have any type; as noted above, its value is empty unless f has region or class type.[8]

Program variables are partitioned into two sets, ordinary variables and ***spec-only variables***.[9] The distinguished variable alloc : rgn is an ordinary variable, but it is treated specially: It is present in all states, and is automatically updated in the transition semantics by the transition for new, so in every state its value is exactly the set of allocated references. Spec-only variables are used in specs to "snapshot" initial values for reference in the postcondition. Spec-only variables do not occur in code, even ghost code, or in effects.[10] In our prototype, "old" expressions are used to abbreviate the use of snapshot variables [60].

Commands are typed in a context Γ. We omit the straightforward rules for typing of commands, except to note that a call $\Gamma \vdash m()$ is well formed only if m : meth is in Γ. To streamline the formal development, we omit parameters for methods; by-value parameters can be handled straightforwardly as in RLII and RLIII.[11]

Program expressions E are heap independent. For expressions of reference type, the only constant is null and the only operation is equality test, written =. Region expressions can depend on the heap but are always defined. Null dereference faults only occur in the primitive load and store commands $x := y.f$ and $x.f := y$. By contrast, if x is null then $\{x\}\text{`}f$ is defined to be empty.

3.2 Modules

Assume given a set *ModName* of module names, and map $mdl : MethName \to ModName$ that associates each method with its module. Usually, we use letters M, N, L for module names, but there is a distinguished module name, $\bullet$, that serves both as main program and as default module in the proof rules for atomic commands. Assume given a preorder $\preceq$ (read "imports") on *ModName*, which models the reflexive transitive closure of the import relation of a complete program. We write $\prec$ for the irreflexive part. Cycles are allowed, as needed for interdependent modules that respect each other's encapsulation boundaries. A module interface includes a spec for each method. The function *bnd* from *ModName* to effect expressions associates each module with its dynamic boundary, which is thus part of its interface along with its method specs. This lightweight formalization of modules is adapted from RLII (its Section 6.1).

[8] Typing in RLI,RLII is slightly more restrictive.

[9] As in RLII, we rely on a partition of ordinary variables into ***locals***, which are bound by var (and in RLII also method parameters), and ***globals***; but we ignore the distinction where possible. Also, typing rules impose the ***hygiene property*** that variable and method names are not re-declared; this facilitates modeling of states and environments as maps.

[10] Spec-only variables are also used in RLII. But here, we also disallow the use of alloc in ghost code, which was not necessary in RLII, so we have additional need to snapshot alloc.

[11] As in those works, we also disallow let-commands inside let-bound commands and biprograms: in let $m = B$ in C there must be no let in B. (By modeling only top-level method declarations, we simplify the semantics.) We also disallow free occurrences of local variables in B; thus in var x:T in let $m = B$ in C the module code B can't refer to x. In practice, let is only used outermost.

```
module PQ =
  public pool: rgn
  boundary { pool, pool`any, pool`rep`any }

  meth Pqueue (self: Pqueue)              /* constructor */
  meth isEmpty (self: Pqueue) : bool
  meth findMin (self: Pqueue) : Pnode

  meth insert (self: Pqueue, val: int, key: int) : Pnode
    requires { self ≠ null ∧ self ∈ pool }
    ensures  { ¬ (isEmpty(self)) ∧ result ∈ self.rep ∧ result.val = val ∧ result.key = key }
    writes { {self}`any, self.rep`any, alloc } reads { {self}`any, self.rep`any, alloc }

  meth deleteMin (self: Pqueue)
  meth decreaseKey (self: Pqueue, handle: Pnode, key: int)
end
```

Fig. 8. Priority queue interface PQ, eliding private methods and most specs.

For the PQ interface in Figure 8, $mdl(\text{insert}) = \text{PQ}$. In one of our case studies, the main program implements Dijkstra's **single-source shortest-paths (SSSP)** algorithm, as a client of PQ and another module Graph. The import relations are then $\bullet \prec \text{PQ}$ and $\bullet \prec \text{Graph}$.

A module M specifies a dynamic boundary $bnd(M)$. The boundary can be expressed using regions and data groups for abstraction, to cater for implementations that have differing internals. This is why there is a single type, rgn, for sets of references of any type. Well-formedness conditions for boundaries are defined in Section 3.3.

A proper module system would include module-scoped variables and fields that need not be part of the interface and need not be the same in different implementations of a module N. Our simplified formulation streamlines the formal development, because we do not need syntax, typing contexts, and so on, for a full-fledged module calculus, nor correctness judgments for modules. But this comes at a price: some well-formedness conditions on correctness judgments (in the following subsections) and side conditions (in proof rules) merely serve to express lexical scoping that could be handled more neatly using a proper module system.

3.3 Unary Specifications

We assume a first-order signature providing primitive type, function, and predicate symbols for use in specs and in ghost code. Predicate formulas are in Figure 9. The ***points-to*** relation $x.f = E$ says that x is non-null and the value of field f equals the value of E. For examples, see the postcondition of insert in Figure 8. The predicate $\text{type}(G, \overline{K})$ says that every non-null reference in G has one of the class types in the list $\overline{K}$.

Typing of unary predicate formulas P is straightforward. For example, the points-to formula $x.f = E$ is well formed (***wf***) in Γ provided $\Gamma(x)$ is some type K that declares $f : T$ and E has type T. An expression E counts as an atomic formula if it has type bool; this includes equality tests. The signature may include equality at other math types, with standard interpretation.

Quantifiers at a class type K range over allocated references of type K. The logic does not require quantification at type rgn, but we include it to simplify the grammar. It is often useful to bound the range of quantification at reference type to a specific region, in the form $\forall x : K.\ x \in G \Rightarrow P$, to facilitate framing. (This is explored in RLI.) In sugared form: $\forall x : K \in G.\ P$.

Effect expressions. A ***spec*** $P \rightsquigarrow Q\ [\varepsilon]$ comprises precondition P, postcondition Q, and frame condition ε. Frame conditions are ***effect expressions*** ε, defined by

$$\begin{array}{lrcl} \text{(Left-expression)} & LE & ::= & x \mid G`f, \\ \text{(Effect expression)} & \varepsilon & ::= & \text{rd}\, LE \mid \text{wr}\, LE \mid \varepsilon, \varepsilon \mid \bullet. \end{array} \tag{6}$$

P ::= $E \mid x.f = E \mid G \subseteq G \mid \mathsf{type}(G, \overline{K}) \mid R(\overline{F})$ (atomic formulas, where R is in the signature)
$\mid P \wedge P \mid P \Rightarrow P \mid (\forall x : T.\ P)$
Syntax sugar: $G \# H \triangleq G \cap H \subseteq \{\mathsf{null}\}$ and $x \in G \triangleq \{x\} \subseteq G$ and standard defs of $\neg$, $\vee$, and $(\exists x : T.\ P)$.
Precedence: $\wedge$ binds more tightly than $\Rightarrow$ and less tightly than relations like $=, \subseteq$.
Associativity: $P \Rightarrow Q \Rightarrow R$ means $P \Rightarrow (Q \Rightarrow R)$.

Fig. 9. State predicates. For expression forms E, F and G see Figure 5.

Left-expressions, *LE*, are a subset of expressions (category F in Figure 5). They have l-values, as discussed below, and are used in effects and in agreement formulas.[12] An effect ε is wf in Γ provided each of its left-expressions is.

Notation: Besides ε, we often use identifiers η and δ for effect expressions. We use the short term ***effect*** for effect expressions, including compound ones like $\mathsf{rd}\ x$, $\mathsf{wr}\ x$, $\mathsf{wr}\ \{x\}\text{`}f$. The singleton image $\mathsf{wr}\ \{x\}\text{`}f$ can be abbreviated as $\mathsf{wr}\ x.f$. We use the abbreviation rw to mean rd and wr. The empty effect is given explicit notation $\bullet$ for clarity in certain parts of the development, but we omit it when confusion seems unlikely. We often treat compound effects as sets of atomic reads and writes. We also omit repeated tags, e.g., $\mathsf{rd}\ x, y$ abbreviates $\mathsf{rd}\ x, \mathsf{rd}\ y$; and then reads are separated from writes by semicolon, e.g., $\mathsf{rd}\ x, y; \mathsf{wr}\ z, w$.

l-value and r-value. In common usage, the term r-value refers to the meaning of an expression in contexts like the right side of an assignment. For those expressions allowed on the left of an assignment, the l-value is the location to be assigned and the r-value is the current contents of that location [95]. In our language there are two forms of mutable location: variables and heap locations. A heap location is a pair (o, f) where o is an object reference and f a field name; we write the pair as $o.f$.

We identify a subset of expressions, called left-expressions (6), which have an l-value—in addition to the r-values described in Section 3.1 (and formalized in Figure 21). In general, the l-value of a left-expression designates a set of locations. In frame conditions, left-expressions are interpreted for their l-values as is common in spec languages. (Note that our left-expression form $G\text{`}f$ is not an assignment target.)

In the write effect $\mathsf{wr}\ x$, the l-value of expression x is a single location, the variable x itself, independent of the current state. For the left-expression $\{x\}\text{`}f$, the l-value is again a single location, namely, $o.f$, where o is the r-value of x in the current state—unless that value is null, in which case the l-value is the empty set.

Consider a variable r : rgn. The l-value of $r\text{`}f$ is the set of $o.f$ where o is a non-null reference that is an element of the current value of r. (We may say "object in r" to be casual.)

What about the l-value of $r\text{`}f\text{`}g$? It is the set of $o.g$ where o is a non-null reference in the region $r\text{`}f$—that is, $o$ is an element of the r-value of $r\text{`}f$. In case f has type int, that region is empty. In case f has some class type K, the region $r\text{`}f$ is the set of contents of $f$ fields of objects in $r$. So, for $o.g$ to be in the l-value of $r\text{`}f\text{`}g$ means o is the value in $p.f$ for some non-null reference p in r.

Suppose instead that f has type rgn. Then the r-value of $r\text{`}f$ is defined to be the union of the values of the $f$-fields of objects in $r$. (We use the union to avoid sets of sets.) So, for $o.g$ to be in the l-value of $r\text{`}f\text{`}g$ means o is an element of the set $p.f$ for some non-null p in r.

In general, the l-value of a left-expression is dependent on the state, for the values of variables and for the values of fields of allocated objects. For example, consider the private method, link, used internally by insert (Figure 4). The ascribed effect of method link is $\mathsf{rw}\ \{\mathsf{self}\}\text{`}rep\text{`}child$,

[12] For readers familiar with prior RL articles: Effect expressions are exactly the same as in previous articles; we have changed the grammar for clarity.

$$
\begin{array}{lcll}
\mathit{ftpt}(x) & \hat{=} & \mathsf{rd}\, x & \\
\mathit{ftpt}(\varnothing) & \hat{=} & \bullet & \\
\mathit{ftpt}(\{E\}) & \hat{=} & \mathit{ftpt}(E) & \\
\mathit{ftpt}(G/K) & \hat{=} & \mathit{ftpt}(G) & \\
\mathit{ftpt}(G\text{`}f) & \hat{=} & \mathsf{rd}\, G\text{`}f, \mathit{ftpt}(G) & \\
\mathit{ftpt}(F_1 \odot F_2) & \hat{=} & \mathit{ftpt}(F_1), \mathit{ftpt}(F_2) & \text{for } \odot \text{ in } \{\cup, \cap, \setminus, +, -\} \\
\mathit{ftpt}(G_0 \subseteq G_1) & \hat{=} & \mathit{ftpt}(G_0), \mathit{ftpt}(G_1) & \\
\mathit{ftpt}(x.f = F) & \hat{=} & \mathsf{rd}\, x, \mathsf{rd}\, \{x\}\text{`}f, \mathit{ftpt}(F) & \\
\mathit{ftpt}(E = E') & \hat{=} & \mathit{ftpt}(E), \mathit{ftpt}(E') &
\end{array}
$$

Fig. 10. Footprints of expressions and atomic formulas.

$\{\mathsf{self}\}\text{`}\mathit{rep}\text{`}\mathit{sibling}$, $\{\mathsf{self}\}\text{`}\mathit{rep}\text{`}\mathit{prev}$. Here, $\{\mathsf{self}\}\text{`}\mathit{rep}$ is used for its r-value, which is a set of objects in the *rep* field (the same as self.*rep*), and the left-expression $\{\mathsf{self}\}\text{`}\mathit{rep}\text{`}\mathit{child}$ is used in the effect to refer to the locations of the child fields of all the Pnodes in $\mathsf{self}\text{`}\mathit{rep}$.

Dynamic boundary and operations on effects. For expressions and atomic formulas, read effects can be computed syntactically by the ***footprint function***, *ftpt*, defined in Figure 10. For example, the private invariant for the PQ module (Figure 8) includes $q.\mathit{rep}\text{`}\mathit{prev} \subseteq q.\mathit{rep}$. Its footprint, computed by *ftpt*, is $\mathsf{rd}\, q, \mathsf{rd}\, \{q\}\text{`}\mathit{rep}, \mathsf{rd}\, \{q\}\text{`}\mathit{rep}\text{`}\mathit{prev}$, which can be abbreviated as $\mathsf{rd}\, q, \{q\}\text{`}\mathit{rep}, q.\mathit{rep}\text{`}\mathit{prev}$. It has a closure property, framed reads, that will play a role in reasoning about encapsulation.

Definition 3.1 (Framed Reads; Candidate Dynamic Boundary). An effect ε has ***framed reads*** provided that for every $\mathsf{rd}\, G\text{`}f$ in ε, its footprint $\mathit{ftpt}(G)$ is in ε. A ***candidate dynamic boundary*** is an effect that has framed reads, has no write effects, and has no spec-only or local variables.

In addition to the well-formedness assumption that the module import relation, $\preceq$, is a preorder, we also assume that every declared boundary, $\mathit{bnd}(M)$, is a candidate dynamic boundary. The distinguished ***default module*** name $\bullet$ has empty boundary: $\mathit{bnd}(\bullet) = \bullet$. For a finite set $X \subseteq \mathit{ModName}$, we use the abbreviation $(+N \in X.\ \mathit{bnd}(N))$ for the catenation (union) of the boundaries. Note that such combined boundaries are themselves candidate dynamic boundaries. For PQ, the dynamic boundary, $\mathit{bnd}(\mathsf{PQ})$, is $\mathsf{rd}\, \mathit{pool}, \mathit{pool}\text{`}\mathsf{any}, \mathit{pool}\text{`}\mathit{rep}\text{`}\mathsf{any}$.

The syntactic operation of ***effect subtraction***, $\varepsilon \backslash \eta$, is used to formulate local equivalence specs; in particular, we subtract a dynamic boundary from a method's frame condition. Subtraction is defined as follows. First, put ε and η into the following normal form[13]: No field occurs outermost in more than one field read or more than one field write. This can be achieved by merging $\mathsf{rd}\, G\text{`}f, \mathsf{rd}\, H\text{`}f$ into $\mathsf{rd}\, (G \cup H)\text{`}f$ and likewise for write. (Occurrences of field images within G and H, not being outermost, are untouched.) Assuming ε, η are in normal form, define $\varepsilon \backslash \eta$ to be $(\delta_0, \delta_1, \delta_2, \delta_3)$ where

$$
\begin{array}{l}
\delta_0 = \{\mathsf{rd}\, x \mid \mathsf{rd}\, x \in \varepsilon \text{ and } \mathsf{rd}\, x \notin \eta\} \\
\delta_1 = \{\mathsf{rd}\, G\text{`}f \mid \mathsf{rd}\, G\text{`}f \in \varepsilon \text{ and } \eta \text{ has no } f \text{ read}\} \cup \{\mathsf{rd}\, (G \backslash H)\text{`}f \mid \mathsf{rd}\, G\text{`}f \in \varepsilon \text{ and } \mathsf{rd}\, H\text{`}f \in \eta\}
\end{array}
\tag{7}
$$

and δ_2, δ_3 are defined the same way for writes. For example, let r and s be region variables. Then $(\mathsf{rd}\, r, \mathsf{rd}\, s, \mathsf{rd}\, (r \cup s)\text{`}\mathit{nxt}, \mathsf{rd}\, r\text{`}\mathit{val}) \backslash (\mathsf{rd}\, r, \mathsf{rd}\, \{x\}\text{`}\mathit{nxt})$ is $\mathsf{rd}\, s, \mathsf{rd}\, ((r \cup s) \backslash \{x\})\text{`}\mathit{nxt}, \mathsf{rd}\, r\text{`}\mathit{val}$.

[13] After replacing the data group any with the fields it stands for.

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

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

The separator function $\cdot/.$, mentioned in connection with the frame rule (5) is defined by structural recursion on effects (Figure 11).[14] Given effects ε, η it generates a formula $\varepsilon \cdot/. \eta$ that implies the read effects in ε are disjoint locations from the writes in η. Please note that $\cdot/.$ is not syntax in the logic; it's a function in the metalanguage that is used to obtain formulas, dubbed ***separator formulas***, from effects. For example, $\mathsf{rd}\, r\text{`}nxt \cdot/. \mathsf{wr}\, r\text{`}val$ is the formula $true$ and $\mathsf{rd}\, r\text{`}nxt \cdot/. \mathsf{wr}\, s\text{`}nxt$ is the disjointness formula[15] $r \# s$. Note that $\varepsilon \cdot/. \eta$ is identical to $rds(\varepsilon) \cdot/. wrs(\eta)$ where rds keeps just the read effects and wrs the writes. The separator function can be used to obtain disjointness conditions for two read effects, say ε and η, by using the function we call $r2w$, which discards write effects and changes reads to writes, as in $\varepsilon \cdot/. r2w(\eta)$. Function $w2r$ does the opposite. The upcoming Example 3.5 shows a use of $\cdot/.$ and the frame rule.

3.4 Unary Correctness Judgments

On the way to formalizing correctness judgments, we first consider specs. Spec-only variables are implicitly scoped over the spec but not explicitly declared.

Definition 3.2 (WF Spec). A spec $P \rightsquigarrow Q\ [\varepsilon]$ is well formed (***wf***) in context Γ if

- Γ has no spec-only variables, and ε is wf in Γ.
- P and Q are wf in $\Gamma, \hat{\Gamma}$, for some $\hat{\Gamma}$ that declares only spec-only variables.[16]
- In P, every occurrence of a spec-only variable s is in an equation $s = F$ that is a top-level conjunct of P, where F has no spec-only variables; and every spec-only variable in Q occurs in P.

The last item says spec-only variables are used as "snapshot" variables.[17] In this article, the $'$ symbol is often used for identifiers on the right side of a pair, so we avoid it for other decorative purposes, instead using $\hat{hats}$ and $\dot{dots}$.

A ***hypothesis context*** Φ (context, for short) maps some procedure names to specs and is written as a comma-separated list of entries $m : P \rightsquigarrow Q\ [\varepsilon]$.

A ***correctness judgment*** has the form $\Phi \vdash^{\Gamma}_{M} C : P \rightsquigarrow Q\ [\varepsilon]$ where Φ is a hypothesis context and M is a module name. The judgment is for code of the ***current module*** M. We distinguish two kinds of method calls in C: ***environment calls*** are those where a called method is bound by let

[14] This is unchanged from prior work (RLI,RLII). The data group "any" can be expanded to all the field names. Computing $\mathsf{rd}\, G\text{`}f \cdot/. \mathsf{wr}\, H\text{`}\text{any}$ yields the formula $G \# H$.

[15] Note that $r \# s$ allows r and/or s to contain null; this is okay, because there are no heap locations based on null.

[16] Here is what is needed to formalize method parameters. They can be referenced in the pre- and postcondition. The frame must not allow write of a parameter, for the usual reason in Hoare logic that the postcondition should refer to the initial value. The frame should not allow read of a parameter: The call rule reflects that what is read is the argument expression in the call. The linking rule allows the body of a method to read its parameters (see RLIII).

[17] In Definition 3.2, $\hat{\Gamma}$ is uniquely determined from the other conditions. This is why we can leave types of spec-only variables implicit. Their scope is also not explicit, but in the semantics they are scoped over the pre- and poststates. We can refer to "the spec-only variables of P" as a succinct way to refer to those used in the spec.

within C; the others, ***context calls***, are those where a called method is specified in Φ. Informally, the correctness judgment says executions of C from P-states read and write only as allowed by ε, and Q holds in the final state if execution terminates. A context call to m in Φ may involve reading and writing encapsulated state for the module, $mdl(m)$, of m, and these effects must be allowed by ε. Commands are given small step semantics, with bodies of let-bound methods kept in an environment. The judgment also says that, aside from context calls, steps of C must neither read nor write locations encapsulated by any module in Φ except its own module M. These conditions must hold for any correct implementation of Φ, so the judgment expresses "modular correctness" [61].

Typically, in a judgment $\Phi \vdash_M C : \ldots$ we will have $M \preceq N$ for each N in Φ (i.e., each N for which some m in Φ has $mdl(m) = N$). However, we do not want to say Φ must contain every N with $M \preceq N$, because we use "small axioms" [76] to specify atomic commands, which are stated in terms of the minimum relevant context. Additional hypotheses can be added using "context introduction" rules with side conditions that enforce encapsulation, as discussed in Sections 3.5 and 6.3. At the point in a proof where a client C is linked with implementations of its context Φ, the judgment for C will include all methods of the modules in Φ, and all transitive imports.

Because we are not formalizing a separate calculus of modules and module judgments, some module-related scoping and typing conditions are associated with correctness judgments for commands. The lack of an explicit binder for the spec-only variables of a spec also requires some care.

Definition 3.3 (WF Correctness Judgment). A correctness judgment $\Phi \vdash^{\Gamma}_{M} C : P \rightsquigarrow Q\ [\varepsilon]$ is wf if

- Φ is wf, i.e., each spec in Φ is wf in Γ and they have disjoint spec-only variables.[18]
- No spec-only variables, nor alloc, occur in C.
- No methods occur in Γ, and C is wf[19] in the typing context that extends Γ to declare the methods in Φ.
- for all N with $N \in \Phi$ or $N = M$, the candidate dynamic boundary $bnd(N)$ is wf in Γ.
- $P \rightsquigarrow Q\ [\varepsilon]$ is wf in Γ, and its spec-only variables are distinct from those in Φ.

For example,

$$m : \mathsf{true} \rightsquigarrow x > 0\ [\mathsf{rw}\, x] \vdash_{\bullet}^{x:\mathsf{int},\, y:\mathsf{int}} x := 0; m() : x \leq 0 \rightsquigarrow x > 0\ [\mathsf{rw}\, x]$$

is a wf judgment; in particular, we have the typing x:int, y:int, m:meth $\vdash x := 0; m()$.

Example 3.4. This example illustrates boundaries and specs. To specify the priority queue ADT (Figure 8), we use an ownership idiom mentioned earlier (Section 2.2). A ghost variable *pool* : rgn is used to keep track of queue instances and each queue's *rep* field contains objects it notionally owns. For a particular implementation, the private invariant includes conditions that imply all allocated queues have valid representations.

In one of our case studies, we verify two implementations of the PQ module using pairing heaps [98], both using objects of class Pnode. The private invariant of both versions includes the condition that for each $q \in pool$, $q.rep`sibling \cup q.rep`prev \cup q.rep`child \subseteq q.rep$. This says the *rep* of q is closed under these field images. An interesting feature of this example is that clients manipulate Pnode references, as "handles" returned by insert, but must respect encapsulation by not reading or writing the fields.

[18] The latter condition loses no generality, since spec-only variables have scope over a single spec, and distinctness helps streamline notation in some soundness proofs.

[19] Strictly speaking, we assume that for any subprogram of the form if E then C else D, we have $C \not\equiv D$. This loses no generality: it can be enforced using labels, or through the addition of dummy assignments. This is needed to express, in the definitions for encapsulation (Definition 5.10), that two executions follow exactly the same control path.

The leaves of the pairing heap are represented using null for the child in one implementation and using references to a sentinel Pnode in the other. One benefit of using sentinels is that certain checks for null can be avoided; our motivation is simply to exemplify two different but similar data structures.

As per Figure 8 the dynamic boundary, $bnd(\mathsf{PQ})$, is $\mathsf{rd}\, pool, pool\text{`}\mathsf{any}, pool\text{`}rep\text{`}\mathsf{any}$. To reason that operations on one priority queue have no effect on others, the public invariant expresses disjointness following the idiom mentioned in Section 2.2:

$$\forall p, q \in pool.\ p \neq q \Rightarrow p.rep \,\#\, q.rep \wedge p \notin q.rep. \tag{8}$$

While it is convenient for a module to declare a public invariant, there is no subtle semantics: A public invariant simply abbreviates a predicate that is conjoined to the pre- and postconditions of the module's method specs. That invariant is typically framed by the boundary, in which case clients easily maintain the invariant (and use it in their loop invariants).

As an example spec, consider the one for PQ's insert (Figure 8). Abbreviating the parameters as q, v, k, a call $\mathsf{insert}(q, v, k)$ adds to a given queue q, a Pnode with value v and key k. Its spec is

$$q \neq \mathsf{null} \wedge q \in pool \quad \rightsquigarrow \quad \neg\mathsf{isEmpty}(q) \wedge res \in q.rep \wedge res.val = v \wedge res.key = k \\ [\mathsf{rw}\ \{q\}\text{`}\mathsf{any}, q.rep\text{`}\mathsf{any}, \mathsf{alloc}],$$

where res is the return value, which references the inserted Pnode. This pointer to an internal object serves as handle for a client to increase the priority, for which purpose it calls $\mathsf{decreaseKey}(q, n, k)$ with spec

$$q \neq \mathsf{null} \wedge q \in pool \wedge \neg\mathsf{isEmpty}(q) \wedge n \neq \mathsf{null} \wedge k \leq n.key \wedge n \in q.rep \\ \rightsquigarrow\ n.key = k\ [\mathsf{rw}\ \{q\}\text{`}\mathsf{any}, q.rep\text{`}\mathsf{any}].$$

Clients see these pre- and postconditions conjoined with the public invariant.

Example 3.5. The separator function $(\cdot/.)$ is used in the frame rule (5) (formalized in Figure 23). To illustrate, consider a program with variables p : Pqueue and q : Pqueue. In accord with Example 3.4, the proof rule for method call gives a judgment like this (eliding hypothesis context):

$$n := \mathsf{insert}(q, v, k) : R \rightsquigarrow S\ [\mathsf{rd}\, q, v, k; \mathsf{wr}\, n; \mathsf{rw}\ \{q\}\text{`}\mathsf{any}, q.rep\text{`}\mathsf{any}, \mathsf{alloc}],$$

where R, S are the pre- and postcondition of *insert*'s spec. Note that the call reads the arguments, and writes the result, in addition to the effects of the method spec (Figure 8).

Consider the formula $p \neq q$. It depends only on p and q, which are not written by the displayed call to *insert*; so the frame rule lets us infer

$$n := \mathsf{insert}(q, v, k) : R \wedge p \neq q \rightsquigarrow S \wedge p \neq q\ [\mathsf{rd}\, q, v, k; \mathsf{wr}\, n; \mathsf{rw}\ \{q\}\text{`}\mathsf{any}, q.rep\text{`}\mathsf{any}, \mathsf{alloc}].$$

To be precise, the rule requires a *framing judgment* confirming that $\mathsf{rd}\, p, q$ covers the footprint of formula $p \neq q$. (This is formalized in Section 6.1 and used in rule Frame, which appears in Figure 23.) That is, $p \neq q$ is "framed by $\mathsf{rd}\, p, q$." The rule also requires to compute a separator for the reads of the formula $(\mathsf{rd}\, p, q)$ and the writes of the command, namely, $\mathsf{rd}\, p, q \ /.\ \mathsf{wr}\ \{q\}\text{`}\mathsf{any}, q.rep\text{`}\mathsf{any}, \mathsf{alloc}$ (see Figure 11) and show it follows from the precondition. In this case the separator formula is simply true; the only locations read are the variables p and q, and the only variable written is alloc.

Now consider the formula $\mathsf{isEmpty}(p)$. The spec of isEmpty has frame condition $\mathsf{rd}\ \{self\}\text{`}size$, so the formula $\mathsf{isEmpty}(p)$ is framed by $\mathsf{rd}\, p, p.size$, which abbreviates $\mathsf{rd}\, p, \mathsf{rd}\ \{p\}\text{`}size$. The Frame rule lets us add the formula before and after the call $n := \mathsf{insert}(q, v, k)$:

$$R \wedge p \neq q \wedge \mathsf{isEmpty}(p) \rightsquigarrow S \wedge p \neq q \wedge \mathsf{isEmpty}(p)\ [\mathsf{rd}\, q, v, k, \mathsf{rw}\ \{q\}\text{`}\mathsf{any}, q.rep\text{`}\mathsf{any}, \mathsf{alloc}].$$

```
module UnionFind

  class Ufind {id: IntArray; part: partition; rep: rgn;}

  public pool : rgn
  boundary { pool, pool‘any, pool‘rep‘any }

  meth Ufind(self:Ufind, k:int) : unit
  meth find(self:Ufind, k:int) : int
  meth union(self:Ufind, x:int, y:int) : unit
end.
```

Fig. 12. Excerpts of union-find interface, eliding private methods and specs.

Here the separator is $\mathsf{rd}\, p, \mathsf{rd}\, \{p\}‘\mathit{size} \cdot/\cdot \mathsf{wr}\, \{q\}‘\mathsf{any}, q.\mathit{rep}‘\mathsf{any}, \mathsf{alloc}$. Unfolding the definition of $\cdot/\cdot$, and using that the data group, any, covers every field including *size*, we get the formula $\{p\} \# \{q\} \wedge \{p\} \# \{q\}‘\mathit{rep}$. Rule Frame requires that the separator follows from the precondition. The first conjunct, $\{p\} \# \{q\}$, follows from precondition $p \neq q$. The second conjunct follows using Equation (8), which implies both $p \notin q.\mathit{rep}$ and $q \notin p.\mathit{rep}$.

Summary. So far, we introduced the syntax of commands, unary specs and unary correctness judgments. The symbol $\equiv$ is sometimes used for equality of syntactic objects like variable names, and especially in the case of commands and biprograms, which we identify up to the equivalences in Figure 6.

There are also a number of meta-operators on syntax that are used pervasively and should not be confused with the syntax: effect subtraction ($\varepsilon \backslash \eta$), separator ($\varepsilon \cdot/\cdot \eta$), footprint ($\mathit{ftpt}(\eta)$), converting write effects to reads ($\mathit{w2r}$), and so on. There is no concrete syntax for modules; instead there are meta-operators for the boundary $\mathit{bnd}(M)$ of the module named M, the import relation $\preceq$ on module names, and the module name $\mathit{mdl}(m)$ associated with method m.

Appendix E has a table of notations and a table of metavariables.

3.5 Encapsulation in Unary Reasoning about Modules and Clients

In this subsection, we consider how the requirements (E1)–(E4) for encapsulation in Section 2.1, are met in the unary logic. Figure 12 shows the interface of a module that provides a class whose instances are union-find structures. The first requirement for encapsulation, (E1), is to delimit some locations internal to the module. That is the purpose of the dynamic boundary, which in the logic would be written $\mathsf{rd}\, \mathit{pool}, \mathsf{rd}\, \mathit{pool}‘\mathsf{any}, \mathsf{rd}\, \mathit{pool}‘\mathit{rep}‘\mathsf{any}$ (in accord with Definition 3.1) and abbreviated as $\mathsf{rd}\, \mathit{pool}, \mathit{pool}‘\mathsf{any}, \mathit{pool}‘\mathit{rep}‘\mathsf{any}$. An equivalent formulation of the boundary is $\mathsf{rd}\, \mathit{pool}, (\mathit{pool} \cup \mathit{pool}‘\mathit{rep})‘\mathsf{any}$.

In this example, we follow the idiom, and even the naming convention, sketched in Section 2.2 for a module providing stacks. Aside from *rep*, the boundary does not mention specific fields but rather uses the data group any for the sake of abstraction.

Because $\mathsf{rd}\, \mathit{pool}$ is in the boundary of UnionFind, client programs may neither read nor write this variable. It serves in specs to designate references to, at least, the Ufind instances managed by the module; so the constructor method Ufind, which should be invoked on newly allocated Ufind objects, adds the new object to *pool*. The boundary includes $\mathsf{rd}\, \mathit{pool}‘\mathsf{any}$, which says fields of these objects may neither be read nor written by client programs. In specs and reasoning about clients, the *rep* field of a Ufind is important: it is used to delimit the locations modified by method calls on that instance, and a public invariant of the module says distinct Ufind instances have disjoint *rep*. This enables reasoning that performing an operation on one Ufind does not affect the state of another Ufind—which is locality, not encapsulation. Fields of objects in *rep* are encapsulated by the

module, as expressed by rd *pool*‘*rep*‘any. Here *pool*‘*rep* is the union of the *rep* fields of all allocated Ufinds.

We consider an implementation based on the quick-find data structure [88]. Math type partition represents a partition on a set of numbers $0 \ldots n-1$. It is used in ghost code and specs, in particular, the private invariant, which says each queue p satisfies a predicate defined on its internal representation, which is an array referenced by field *id*.

```
predicate ufInv (p: Ufind) =
  p.id ≠ null ∧
  let n = p.id.len in
    size(p.part) = n ∧  p.rep = { p.id } ∧
    (∀ x:int. 0 ≤ x < n ⇒ 0 ≤ p.id[x] < n ∧ p.id[p.id[x]] = p.id[x]) ∧
    (∀ x:int, y:int. 0 ≤ x < n ∧ 0 ≤ y < n ⇒ ( y ∈ pfind(x,p.part) ⇔ p.id[x] = p.id[y]) )

private invariant I_qf = ∀ p: Ufind ∈ pool. ufInv(p)
```

The union-find implementation uses a representative element for each block of the partition, with $id[x]$ being the representative of x, for each x in $0 \ldots n-1$. If x is a representative, then $id[x] = x$. The private invariant says that for any x, $id[x]$ is a representative: $p.id[p.id[x]] = p.id[x]$. The last conjunct says x and y have the same representative in $p.id$ just if they are in the same block of the abstract partition. The ghost field *rep* has nothing to do with representatives; as in our usual idiom it holds references to the internal representation objects, in this case just the *id*.

Requirement (E2) for encapsulation is that a private invariant depends only on locations within the boundary. This is formalized in the logic by a *framing judgment*, which in our example is written $\models$ (rd *pool*, rd (*pool* ∪ *pool*‘*rep*)‘any) frm I_{qf}. As formalized later, its meaning is that if I_{qf} holds in some state, then it holds in any other state that agrees on the values in the locations designated by the read effect. Looking at its definition, I_{qf} depends on only one variable, *pool*. The heap locations on which it depends are in expressions $p.id$ and index expressions $p.id[x]$. As we have $p.id \in p.rep$, by the invariant, and the slots of the array are effectively fields of *id*, these heap locations are indeed covered by rd (*pool* ∪ *pool*‘*rep*)‘any. The meaning of the framing judgment can be encoded as a universally quantified formula; this and other framing judgments in our case studies are easily validated by SMT solvers.

Here, we consider the quick-find implementation, which for the find method is

```
meth find (self: Ufind, k: int) : int
= result := self.id[k]
```

A key postcondition of the spec of find is that $result \in pfind(k, self.part)$, where *pfind* is the function that returns the block of the abstract partition that contains k. The postcondition holds in virtue of conditions in the private invariant, including that $id[k]$ is a representative, for any k, and the connection between $self.part$ and $self.id$.

Encapsulation of a client. As a case study, we have verified Kruskal's minimum spanning tree algorithm as client, but for present purposes we consider a very simple client:

```
uf:=new Ufind(100); x:=new Thing; x.f:=y; z := find(uf,1)
```

To verify the client code, its hypothesis context needs to include the module specs, in particular for find. So UnionFind is in scope and its boundary must be respected by the client. The logic enforces encapsulation of clients, i.e., requirement (E3), using separation checks similar to those for frame-based reasoning as in Example 3.5.

To explain the checks, let us write δ_{uf} for the boundary of UnionFind. The command $x :=$ new Thing has frame wr x, rw alloc. Respect of δ_{uf} by this command is formulated in terms of

the separator function, in this case δ_{uf} ·/. wr x, alloc. Unfolding the definition (Figure 11) yields the formula true ∧ true. The only variable designated by δ_{uf} is *pool*, and this is distinct from x and from alloc. The proof obligation here also rules out client code that assigns or reads *pool*. In general, it is untenable to include rd alloc in a boundary, or even an image expression mentioning alloc, because clients typically do allocation.

The command $x.f := y$ has frame condition rd x, rd y, wr $\{x\}$‘f. For the write to be outside the boundary, the obligation can be written δ_{uf} ·/. wr $\{x\}$‘f. Unfolding by definition of the separator function, and expanding the abbreviation any to be all field names in scope, we get a conjunction of *true*s (because the read and written variables are distinct) and two nontrivial conjuncts: $pool \# \{x\}$ and $pool$‘$rep \# \{x\}$. That is, the assigned object must be in neither *pool* nor any *rep* fields of objects in *pool*. One way this obligation can be proved is via freshness: neither *pool* nor *rep* have been updated since x was assigned a fresh object. A related idiom used in some method specs is a postcondition that says all fresh objects are in $self.rep$, which a client can use to reason that its own regions remain disjoint. In a postcondition, the fresh references are denoted by alloc\old(alloc). In the formal logic state predicates only refer to a single state, so a postcondition must be expressed in the same way that tools desugar “old” expressions. That is, a fresh spec-only variable, say r, is used to snapshot the initial value: the precondition includes r = alloc and the idiomatic postcondition is now alloc$\setminus r \subseteq self.rep$.

We are not finished with $x.f := y$. In addition to its writes, its reads must be outside the boundary, specifically, x and y must be outside δ_{uf}. This can be written δ_{uf} ·/. wr x, wr y. Why wr ? Just so we can use the separator function ·/. unchanged from prior work, though it is defined to separate read effects from writes. (The proof rule for field update uses another metafunction, *r2w*, to convert the reads to writes.)

As an example of how encapsulation checks can fail, consider a bad client of the PQ interface (Figure 8) that calls insert and assigns the returned Pnode to variable *nd*, and then writes the *key* field of *nd*—potentially invalidating a private invariant. The boundary of PQ is similar to the one for UnionFind, so the separator formula is $pool \# \{nd\} \wedge pool$‘$rep \# \{nd\}$. This is not valid, since the value of *nd* is in $pool$‘rep.

So far, we saw how the frame conditions of atomic commands give rise to proof obligations that ensure the client reads and writes are to locations disjoint from the locations designated by the boundary. Please note that the interpretation of the boundary is at the point in execution where the atomic command has its effects. This does not make a difference for variables, in the sense that a separator rd x ·/. wr y is just true or false depending on whether the variable names are distinct. It does make a difference for heap locations, designated by expressions like $pool$‘any and $\{x\}$‘f; in this case the obligation $pool \# \{x\}$ discussed above must hold in the pre-state of the assignment command $x.f := y$.

Loops and conditionals also incur an encapsulation obligation that their test expressions read outside the boundary. In our desugared syntax (Figure 5) these expressions are heap independent. In the example the check is simply that variable *pool* does not occur in a test expression, since the other locations in the boundary are heap locations. Here is an example where a test crosses the boundary of PQ:

q := new Pqueue(); nd := insert(q,0,0); if nd.prev ≠ null then q := null fi; nd := insert(q,1,1)

This client works fine with the first implementation of PQ, since $nd.prev$ will be null. But for the implementation with sentinels, the second call to insert will fault due to null dereference. The client is not representation independent and the read of $nd.prev$ will fail the encapsulation check.

In our prototype, WhyRel, encapsulation checks like this are straightforward. At points where the encapsulation check is state dependent, like $x.f := y$, WhyRel generates an assert statement that encodes the disjointness obligation (Section 9). In the logic, encapsulation checks are disentangled from other reasoning considerations by the context introduction proof rules. The modules whose boundary must be respected are those of the methods in the hypothesis context, given using the *mdl* function defined in Section 3.2. The technical details are not conceptually important, and are explained in Section 6.3.

In summary, encapsulation requirement (E3) is achieved by checking separation from the relevant boundaries, for each part of the client command. Separation is checked the same way as it is for the ordinary Frame rule, using formulas generated from the effects using the separator function $(\cdot/\cdot)$. For effects on variables it is true or false depending on whether the requisite variables are distinct, but for effects on heap locations (load and store commmands, method calls) the separation checks are region disjointness formulas that must hold at the relevant points in control flow.

Modular linking. Suppose we verify the client, using the public specs, and discharge the proof obligations, just discussed, for encapsulation. We verify the implementation of find, union, etc using the private invariant I_{qf}, i.e., assuming it as precondition and establishing it as post, in accord with the modular linking rule sketched as Equation (2) in Section 2.1. Having verified the client and the implementations of module methods, we would like to conclude that the linked program is correct, i.e., satisfies the client spec as per rule (2). The private invariant is hidden from the client, in the sense that the method bodies are verified for specs that include it, but it is omitted from the hypotheses used to verify the client. There is one more requirement for this to be sound, namely, (E4): the client precondition implies the private invariant of the module. An appropriate such precondition is $pool = \emptyset$, the default value for regions, which implies I_{qf} owing to its quantification over *pool*.

The intuition that justifies Equation (2) is that, given the client's respect for the boundary, any judgment $D : P \rightsquigarrow Q[\varepsilon]$ about a client subprogram D yields $D : P \wedge I \rightsquigarrow Q \wedge I[\varepsilon]$ by an application of the frame rule (because the encapsulation obligation ensured the footprint of the private invariant I is disjoint from the effects in ε). In particular, at a point where the client has established public precondition R of a method that has been verified using precondition $R \wedge I$, we do in fact have $R \wedge I$. For example, having proved the judgment $\mathsf{find} : R \rightsquigarrow S \vdash C : P \rightsquigarrow Q$ (omitting frame condition) together with the encapsulation obligations for client C, we have

$$\mathsf{find} : R \wedge I_{qf} \rightsquigarrow S \wedge I_{qf} \vdash C : P \wedge I_{qf} \rightsquigarrow Q \wedge I_{qf}\,.$$

This is formalized as the ***second-order frame rule***, SOF in Figure 23. The modular linking rule (2) is a consequence of SOF together with the obvious linking rule that requires the method bodies to satisfy exactly the specs assumed by the client. Please note that all formulas involved in the specs are first-order; the SOF rule is called second order only in the sense that the framed formula is conjoined to specs in the hypothesis context as well as to the consequent of the judgment.

On dynamic boundaries. In this article, we repeatedly use the idiom with *pool* and *rep*, but this is merely one convenient way to write specs that support module-based encapsulation and per-instance local reasoning. Ghost variables and fields can just as well be used to express hierarchical ownership or cooperating clusters of objects as in design patterns like subject-observer. Such examples can be found in RLI–III.

A key point is that the dynamic boundary is part of a module interface, and should be expressed in such a way that different module implementations can have different internal data structures. Thus, the same dynamic boundary may denote different locations for different implementations.

$$
\begin{array}{lcl}
\overleftarrow{(C|C')} & \triangleq & C \\
\overleftarrow{\lfloor A \rfloor} & \triangleq & A \\
\overleftarrow{\text{if } E|E' \text{ then } BB \text{ else } CC} & \triangleq & \text{if } E \text{ then } \overleftarrow{BB} \text{ else } \overleftarrow{CC} \\
\overleftarrow{\text{while } E|E' \cdot \mathcal{P}|\mathcal{P}' \text{ do } CC} & \triangleq & \text{while } E \text{ do } \overleftarrow{CC} \\
\overleftarrow{BB\,;CC} & \triangleq & \overleftarrow{BB}\,;\,\overleftarrow{CC} \\
\overleftarrow{\text{var } x{:}T|x'{:}T' \text{ in } CC} & \triangleq & \text{var } x{:}T \text{ in } \overleftarrow{CC} \\
\overleftarrow{\text{let } m = (C|C') \text{ in } CC} & \triangleq & \text{let } m = C \text{ in } \overleftarrow{CC}
\end{array}
$$

Symmetrically, $\overrightarrow{(C|C')} \triangleq C'$, $\overrightarrow{\lfloor A \rfloor} \triangleq A$, etc.

Fig. 13. Syntactic projections $\overleftarrow{\ }$ and $\overrightarrow{\ }$ of biprograms.

This can be achieved using ghost state, data groups, and pure methods. In this article, we only formalize a single data group, any, and we omit pure methods (see Section 2.6).

To prove the disjointnesses needed for client code to be outside a boundary, one can rely on invariants that constrain the relevant ghost state. For this purpose it is convenient for a module interface to include public invariants such as Equation (8) in Example 3.4.

4 BIPROGRAMS: SYNTAX AND RELATIONAL REASONING

This section formalizes biprograms (Section 4.1), relation formulas (Section 4.2), relational specs and correctness judgments (Section 4.3). Section 4.4 uses an example to illustrate how regions are used in relation formulas and how biprograms express convenient alignments. Section 4.5 defines the weaving relation and explains its use to account for helpful alignments. Section 4.6 sketches example of relational modular linking.

In this section, as in Section 3, we use the syntax of our prototype for program code, together with the math notations of the formal logic. We use syntax sugar and also some features that are not formalized in the logic, namely, parameters and return values (see Section 2.6), for the sake of readable examples. More about the prototype can be found in Section 9.

4.1 Biprograms

Figure 5 gives the grammar of biprograms. A biprogram CC represents a pair of commands, which are given by syntactic projections defined in Figure 13. For example, the left projection $\overleftarrow{(\text{skip}|x := 0);(y := 0|z := 1)}$ is $y := 0$, taking into account that we identify skip; $y := 0$ with $y := 0$ (see Figure 6). The symbol | is used throughout the article, in program and spec syntax and also as alternate notation for pairing in the metalanguage, when the pair represents a pair of states or similar.[20]

Biprograms are given small-step semantics. The ***bi-com*** form $(C|D)$ represents executions of commands C and D, which are meant to be aligned on their initial state and, if they terminate, final state. Their execution steps are interleaved (i.e., dovetailed, in the terminology of automata theory), to ensure that the traces of $(C|D)$ cover all traces of C and D by making progress on both sides even if one diverges. The parentheses of bi-coms are obligatory and the operator binds less tightly than others: $(A; B|C; D)$ is the same as $((A; B)|(C; D))$. In Section 4.5 we consider how the other biprogram forms are introduced for a verification problem specified using a bi-com. For now, we briefly explain the other forms.

The ***sync*** form $\lfloor A \rfloor$ represents two executions of the atomic command A, aligned as a single step. This is mainly of interest for allocations and method calls. For a call, $\lfloor m() \rfloor$ indicates that a relational

[20] A small version of the symbol is used, interchangeably, for clarity in some contexts such as grammar rules.

$$
\begin{array}{lll}
FF ::= & \langle\!|F|\!\rangle \mid |\!\rangle F|\!\rangle & \text{Value in left (resp. right) state} \\
\mathcal{P} ::= & R(\overline{FF}) & \text{Primitive } R \text{ in signature} \\
& \mid F \doteqdot F & \text{Equal expressions, mod refperm} \\
& \mid \mathbb{A}\, LE & \text{Agreement mod refperm} \\
& \mid \Diamond \mathcal{P} & \text{Possibly (in some extended refperm)} \\
& \mid \langle\!|P|\!\rangle \mid |\!\rangle P|\!\rangle & \text{In the left (resp. right) state} \\
& \mid \mathcal{P} \wedge \mathcal{P} \mid \mathcal{P} \Rightarrow \mathcal{P} \mid \forall x{:}T|x{:}T.\, \mathcal{P} &
\end{array}
$$

Syntax sugar: $\mathbb{B}P \triangleq \langle\!|P|\!\rangle \wedge |\!\rangle P|\!\rangle$

$\Box\mathcal{P} \triangleq \neg\Diamond\neg\mathcal{P}$

$\mathsf{false} \triangleq \mathbb{B}\mathsf{false} \qquad \mathsf{true} \triangleq \mathbb{B}\mathsf{true}$

$\mathbb{A}x.f \triangleq \mathbb{A}\{x\}\text{`}f$

$\mathbb{A}(\mathsf{rd}\, LE) \triangleq \mathbb{A}\, LE \qquad \mathbb{A}(\mathsf{wr}\ \ldots) \triangleq \mathsf{true} \qquad \mathbb{A}(\varepsilon, \eta) \triangleq \mathbb{A}(\varepsilon) \wedge \mathbb{A}(\eta)$

Precedence: (tightest) $\mathbb{A}$, $\Diamond$, $\doteqdot$, $\wedge$, $\Rightarrow$ (loosest).

Fig. 14. Relation formulas. See Figure 9 for unary formulas P and Equation (6) for left-expressions LE.

spec should be used to reason about the two calls. For an allocation, the form $\lfloor x := \mathsf{new}\ K \rfloor$ has a proof rule in which the two new references are considered in agreement, i.e., "added to the refperm." In the grammar (Figure 5), the bi-var form allows different names and types but one also wants to allow multiple variables on each side; this is implemented in our prototype. The bi-if form, if $E|E'$ then CC else DD, asserts that the two initial states agree on the value of the test expressions E and E'. The bi-while form while $E|E' \cdot \mathcal{P}|\mathcal{P}'$ do CC incorporates relation formulas $\mathcal{P}$ and $\mathcal{P}'$, which serve as ***alignment guards***. These serve as directives to indicate how to align iterations of the loop, catering for situations like the *sumpub* program in Equation (4). This is explained in more detail in Section 4.5; see the aligned *sumpub* in Equation (15).

Typing of biprograms can be defined in terms of syntactic projection, roughly as $\Gamma|\Gamma' \vdash CC$ iff $\Gamma \vdash \overleftarrow{CC}$ and $\Gamma' \vdash \overrightarrow{CC}$. But the alignment guard formulas in a bi-while should also be typechecked in $\Gamma|\Gamma'$, and are required to be free of agreement formulas, i.e., those of the form $\mathbb{A}G\text{`}f$ and $F \doteqdot F'$; this ensures that the formula is refperm-independent as explained later. Although the two sides of a biprogram may have different typing contexts, for simplicity a single class table is assumed. It is straightforward to generalize this to allow different field declarations for a given class (and it is implemented in our prototype).

4.2 Relation Formulas

Relation formulas are interpreted over a pair of states, meant to be at aligned points in two executions. What is important is to express not only conditions relating integers and other mathematical values but also conditions relating structures between the two heaps. There are many ways to formalize such formulas; it is only in the treatment of heap relations that the design choices made here have significant impact on the later development.

The relation formulas are defined in Figure 14. Quantifiers range over allocated references; the relational form binds a variable on each side. The form $\langle\!|P|\!\rangle$ (respectively, $|\!\rangle P|\!\rangle$) says unary predicate P holds in the left state (respectively, right). Left and right embedded expressions are written $\langle\!|F|\!\rangle$ and $|\!\rangle F|\!\rangle$ and have nothing to do with left-expressions LE. They may be used as arguments to atomic predicates in the ambient mathematical theories: $\langle\!|F|\!\rangle$ (respectively, $|\!\rangle F|\!\rangle$) evaluates F in the left (respectively, right) state.[21]

[21] Written $\langle 1\rangle F$ and $\langle 2\rangle F$ in works following Benton [25]. Our notations $\langle\!|F|\!\rangle$ and $\langle\!|P|\!\rangle$ are meant to point leftward.

$$
\begin{array}{lcl}
\overleftarrow{\langle\!|P|\!\rangle} & \triangleq & P \\
\overleftarrow{|\!|P|\!|} & \triangleq & \mathsf{true} \\
\overleftarrow{\Diamond\mathcal{P}} & \triangleq & \Diamond\overleftarrow{\mathcal{P}} \\
\overleftarrow{F \doteq F'} & \triangleq & (F = F) \\
\overleftarrow{\mathbb{A}LE} & \triangleq & (LE = LE) \\
\overleftarrow{\forall x{:}T|x'{:}T'.\,\mathcal{P}} & \triangleq & \forall x : T.\, \overleftarrow{\mathcal{P}} \\
\overleftarrow{R(\overrightarrow{FF})} & \triangleq & \mathsf{true} \\
\\
\overleftarrow{\mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon']} & \triangleq & \overleftarrow{\mathcal{P}} \rightsquigarrow \overleftarrow{\mathcal{Q}}\ [\varepsilon]
\end{array}
$$

Fig. 15. Syntactic projection $\overleftarrow{\ }$ of relation formulas and specs; right projection $\overrightarrow{\ }$ is symmetric.

The forms $\mathbb{A}\,LE$ and $F \doteq F'$ are called ***agreement formulas***. For E and E' of some reference type K, the form $E \doteq E'$ (pronounced "E bi-equals E'") says the value of E in the left is the same as E' on the right, modulo refperm in the case of reference values. Similarly with $G \doteq G'$ for regions. The form $\mathbb{A}G\text{`}f$ says for each reference $o \in G$, with corresponding value $o'$ in the other state, the value of $o.f$ is the same as the value of $o'.f$, modulo refperm if the value is of reference type. For example, $\mathbb{A}r\text{`}rep\text{`}val$ means the val fields agree, for all objects in the rep field of all objects in r.

The form $\mathbb{A}x$ is equivalent to $x \doteq x$. But the form $\mathbb{A}G\text{`}f$ is not equivalent to $G\text{`}f \doteq G\text{`}f$. The former means pointwise field agreement (modulo refperm) and the latter means equal values (modulo refperm), the two values being reference sets.

The modal form $\Diamond\mathcal{P}$, read ***possibly*** $\mathcal{P}$ (for lack of a better word), says $\mathcal{P}$ holds in a refperm possibly extended from the current one. More on these points later.

Relation formulas and relational correctness judgments are typed in a context of the form $\Gamma|\Gamma'$ comprises contexts Γ and Γ' for the left and right sides.[22] Leaving aside left/right embedded expressions, typing can be reduced to typing of unary formulas: $\Gamma|\Gamma' \vdash \mathcal{P}$ iff $\Gamma \vdash \overleftarrow{\mathcal{P}}$ and $\Gamma' \vdash \overrightarrow{\mathcal{P}}$. This refers to syntactic projections defined in Figure 15. This does not work for left/right embedded expressions; we gloss over those for clarity, in the following sections as well, but handle them in our prototype.

In accord with the definition of projections, we have the formula typing $\Gamma|\Gamma' \vdash \mathbb{A}x$ just if $x \in dom(\Gamma) \cap dom(\Gamma')$. We have $\Gamma|\Gamma' \vdash \mathbb{A}G\text{`}f$ just if $\Gamma \vdash G : \mathrm{rgn}$ and $\Gamma' \vdash G : \mathrm{rgn}$, with f of any type. Similarly, $\Gamma|\Gamma' \vdash F \doteq F'$ provided $\Gamma \vdash F : T$ and $\Gamma' \vdash F' : T$. Also $\Gamma|\Gamma' \vdash \langle\!|P|\!\rangle$ if $\Gamma \vdash P$ and $\Gamma|\Gamma' \vdash |\!|P|\!|$ if $\Gamma' \vdash P$.

4.3 Relational Specifications and Correctness Judgment

A ***relational spec*** $\mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon']$ has relational pre- and postconditions and a pair of frame conditions. We write $\mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon]$ to abbreviate the frame condition $[\varepsilon|\varepsilon]$. A spec $\mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon']$ is wf in $\Gamma|\Gamma'$ provided $\overleftarrow{\mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon']}$ is wf in Γ (respectively, $\overrightarrow{\mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon']}$ in Γ'), as per Definition 3.2. See Figure 15 for syntactic projections. The precondition $\mathcal{P}$ of a wf relational spec has spec-only variables only as snapshot equations in top level conjuncts of $\mathcal{P}$ (inside the left and right embedding operators $\{-\langle\!|, |\!|-\}$). Any spec-only variables in postcondition $\mathcal{Q}$ must occur in $\mathcal{P}$.

[22] This enables reasoning about two versions of a program acting on the same variables, by contrast with other works where related programs are assumed to have been renamed to have no identifiers in common. Logics should account for renaming.

Recall from Section 2.1 that one important relational property is local equivalence. Later, we define a general construction, *locEq*, that applies to a unary spec $P \rightsquigarrow Q\,[\varepsilon]$ and yields a relational spec (Example 4.3 and Section 8.1). The general form takes into account that encapsulated locations are not expected to be in agreement; that is formalized by means of effect subtraction.

For local equivalence and other purposes, we often want postconditions that assert agreements on fresh locations. These agreements are modulo refperm, so a relational correctness judgment should say there is some refperm for which the final states are related. This can be expressed using the $\diamond$ modality. Many specs of interest have the form $\mathcal{P} \approx\!> \diamond\mathcal{Q}\,[\eta|\eta']$ where $\mathcal{P}, \mathcal{Q}$ are $\diamond$-free. Such specs are said to be in ***standard form***. We gloss over this in some examples. In our prototype, the encoding maintains a "current refperm" in ghost state to interpret agreement formulas, and does not use the $\diamond$ modality explicitly in specs. The dual, $\Box$, is used in a couple of proof rules.

A ***relational hypothesis context*** for $\Gamma|\Gamma'$ is a triple $\Phi = (\Phi_0, \Phi_1, \Phi_2)$ comprising unary hypothesis contexts Φ_0 for Γ and Φ_1 for Γ', together with a mapping Φ_2 of method names to relational specs that are wf.

Definition 4.1 (WF Relational Hypothesis Context). A relational hypothesis context for $\Gamma|\Gamma'$ is wf in $\Gamma|\Gamma'$ provided that Φ_0, Φ_1, Φ_2 specify the same methods,[23] Φ_0 and $\overleftarrow{\Phi_2}$ are wf in Γ, Φ_1 and $\overrightarrow{\Phi_2}$ are wf in Γ', the specs in Φ_2 are wf in $\Gamma|\Gamma'$, and the distinct methods have distinct spec-only variables in Φ_2 (just as in Φ_0 and Φ_1). Moreover, for every m, the formula

$$pre(\Phi_2(m)) \Rightarrow \{\!|pre(\Phi_0(m))|\!\} \wedge [\![pre(\Phi_1(m))]\!]$$

is valid (where metafunction *pre* extracts the precondition), and the effects of $\Phi_2(m)$ project to those of $\Phi_0(m)$ and $\Phi_1(m)$.[24]

The constraint on preconditions ensures a compatibility condition needed to connect relational with unary context models, see Definition 7.9. Definition 4.1 allows left and right to have different global variables. It also allows that some spec-only variables on the left may also occur on the right. However, well formedness is in the context of a single module structure (module names and their association with methods and dynamic boundaries; import relation).

Definition 4.2. A ***relational correctness judgment*** has the form $\Phi \vdash_M^{\Gamma|\Gamma'} CC : \mathcal{P} \approx\!> \mathcal{Q}\,[\varepsilon|\varepsilon']$. It is wf provided

- Φ is wf in $\Gamma|\Gamma'$ (see above).
- No spec-only variables, nor alloc, occur in CC. Moreover, alignment guard assertions in bi-whiles contain no agreement formulas.
- No methods occur in $\Gamma|\Gamma'$, and CC is wf in the typing context that extends $\Gamma|\Gamma'$ to declare the methods in Φ.
- $bnd(N)$ is wf in Γ and wf in Γ', for all N with $N \in \Phi$ or $N = M$.
- $\mathcal{P} \approx\!> \mathcal{Q}\,[\varepsilon|\varepsilon']$ is wf in $\Gamma|\Gamma'$, and its spec-only variables are distinct from those in Φ.

Example 4.3 (Coupling and Local Equivalence for PQ). The coupling relation expresses that for any two corresponding queues in the left and right states' *pool*, all the Pnodes in their *reps* are in the refperm. The sentinel is in *pool*, not in a *rep*, and each pair of corresponding Pnodes have the same value and priority. Moreover, null appears in the left state where the sentinel appears in the

[23] One can allow different methods in context, provided that left (respectively, right; respectively, sync'd) context calls have left (respectively, right; respectively, relational) spec's, and this is implemented in our prototype.

[24] In detail: Suppose $\Phi_2(m)$ is $\mathcal{R} \approx\!> \mathcal{S}\,[\eta|\eta']$, and the unary specs $\Phi_0(m)$ and $\Phi_1(m)$ are $R_0 \rightsquigarrow S_0\,[\eta_0]$ and $R_1 \rightsquigarrow S_1\,[\eta_1]$, respectively. Then, $\eta = \eta_0$ and $\eta' = \eta_1$.

right. As a relation formula:

$$
\begin{array}{l}
\forall q : \mathsf{Pqueue} \in \mathit{pool} \mid q : \mathsf{Pqueue} \in \mathit{pool} \\
\quad \mathbb{A}q \quad \Rightarrow \quad (\mathbb{A}(q.\mathit{head}) \vee (\{\!|q.\mathit{head} = \mathsf{null}|\!\} \wedge [\![q.\mathit{head} = q.\mathit{sntnl}]\!])) \\
\qquad\qquad \wedge\ q.\mathit{rep}/\mathsf{Pnode} \doteqdot q.\mathit{rep}/\mathsf{Pnode} \\
\qquad\qquad \wedge\ \forall\ n{:}\mathsf{Pnode} \in q.\mathit{rep} \mid n{:}\mathsf{Pnode} \in q.\mathit{rep}\ . \\
\qquad\qquad\quad \mathbb{A}n \Rightarrow\ \ \mathbb{A}(n.\mathit{val}) \wedge \mathbb{A}(n.\mathit{key}) \\
\qquad\qquad\qquad\qquad \wedge(\mathbb{A}(n.\mathit{sibling}) \vee (\{\!|n.\mathit{sibling} = \mathsf{null}|\!\} \wedge [\![n.\mathit{sibling} = q.\mathit{sntnl}]\!])) \\
\qquad\qquad\qquad\qquad \wedge(\mathbb{A}(n.\mathit{child}) \vee (\{\!|n.\mathit{child} = \mathsf{null}|\!\} \wedge [\![n.\mathit{child} = q.\mathit{sntnl}]\!])) \\
\qquad\qquad\qquad\qquad \wedge(\mathbb{A}(n.\mathit{prev}) \vee (\{\!|n.\mathit{prev} = \mathsf{null}|\!\} \wedge [\![n.\mathit{prev} = q.\mathit{sntnl}]\!]))
\end{array}
$$

Here, we use syntax sugar $\mathbb{A}n.\mathit{val}$ for $\mathbb{A}\{n\}\text{`}\mathit{val}$. Also, the pattern $\forall q{:}K \in r \mid q{:}K \in r \ldots$ is sugar for $\forall q{:}K \mid q{:}K.\{\!|q \in r|\!\} \wedge [\![q \in r]\!] \Rightarrow \ldots$. Note the type restriction expressions in the agreement $q.\mathit{rep}/\mathsf{Pnode} \doteqdot q.\mathit{rep}/\mathsf{Pnode}$. Let $\mathcal{M}_{PQ}$ be the above formula, conjoined with $\{\!|I|\!\} \wedge [\![I']\!]$ where I, I' are the private invariants.

The relational spec for insert obtained by applying *locEq* looks like this:

$$\mathbb{A}q \wedge \mathbb{A}k \wedge \mathbb{B}P \approx\!\!> \Diamond(\mathbb{A}(\mathit{res}.\mathit{val}) \wedge \mathbb{A}(\mathit{res}.\mathit{key}) \wedge \ldots \wedge \mathbb{B}Q)\ [\mathsf{rw}\ \{q\}\text{`}\mathsf{any}, q.\mathit{rep}\text{`}\mathsf{any}, \mathsf{alloc}], \qquad (9)$$

where P and Q are the unary pre- and postconditions for insert, including the public invariant of PQ. We elide some postconditions like $\mathbb{A}((\mathit{pool} \backslash (\mathit{pool} \cup \mathit{pool}\text{`}\mathit{rep}))\text{`}\mathit{head})$, which arise by subtracting the boundary from writes in the spec (and expanding any to all field names). This one can obviously be simplified to $\mathbb{A}\varnothing\text{`}\mathit{head}$, which is equivalent to true. The meta-function *locEq* need not perform such simplifications, as the reasoning can safely be left to the SMT solver or to the logic's relational consequence rule.

To verify the two implementations of insert, we conjoin $\mathcal{M}_{PQ}$ to both the pre- and postcondition of the relational spec above. The resulting precondition is $\mathbb{A}q \wedge \mathbb{A}k \wedge \mathbb{B}P \wedge \mathcal{M}_{PQ}$ and the postcondition is $\Diamond(\mathbb{A}(\mathit{res}.\mathit{val}) \wedge \mathbb{A}(\mathit{res}.\mathit{key}) \wedge \ldots \wedge \mathbb{B}Q \wedge \mathcal{M}_{PQ})$. Later, we introduce a notation $\circledcirc\mathcal{M}_{PQ}$ for this.

4.4 Relational Verification with Biprograms

We consider an example of relational verification, which is modular in the sense of using relational method specs, but no information hiding. We highlight how regions are used in relational specs, and how biprograms are used to represent convenient alignments.

List tabulation: illustrating procedure-modular reasoning. Consider the two programs in Figure 16, which both tabulate a linked list of the values of some method mf that computes a function, applied to the numbers n down to 1. Objects of class List have two fields: *head* : Node references the head of a linked list and *nds* : rgn is ghost state, to which we return soon. The goal in this example is to prove the programs are equivalent. We reason about executions of the two programs in close alignment, to exploit their similarities and make use of a relational spec for mf. The example also serves to show the use of regions to describe heap structure and in particular to express the equivalence of the lists returned. The example illustrates two aspects of modular reasoning: procedural abstraction and local reasoning; the third aspect, data abstraction, is considered in Section 4.6.

Both versions of the program use field *nds* to hold references to the nodes reached from *head*. It is initially empty (the default value), and in each iteration the newly allocated node is added to the list's *nds*. An invariant of the loop, in both programs, is $t.\mathit{nds}\text{`}\mathit{next} \subseteq t.\mathit{nds}$. Here $t.\mathit{nds}$ is set of references. The image expression $t.\mathit{nds}\text{`}\mathit{next}$ denotes the set of values in the next fields of objects in $t.\mathit{nds}$ (a direct image, thinking of the field as a relation). The containment $t.\mathit{nds}\text{`}\mathit{next} \subseteq t.\mathit{nds}$ says for any object reference in $t.\mathit{nds}$, the value of the object's *next* field is in $t.\mathit{nds}$. There are no

```
meth tabulate (n:int) : List =
  var t: List, i: int, p: Node;
  t := new List;
  i := 0;
  while i < n do
    i := i + 1;
    p := new Node;
    p.val := mf(i);
    p.nxt := t.head;
    t.head := p;
    t.nds := t.nds ∪ {p};
  od;
  result := t;   /* return value */
```

(a) Left version, *tabu*

```
meth tabulate (n:int) : List =
  var t: List, i: int, p: Node;
  t := new List;
  i := 1;
  while i ≤ n do
    p := new Node;
    p.val := mf(i);
    p.nxt := t.head;
    t.head := p;
    t.nds := t.nds ∪ {p};
    i := i + 1;
  od;
  result := t;
```

(b) Right version, *tabu′*

```
/* Agr n */
⌊ t := new List ⌋; connect t;
(i := 0 | i := 1);
while (i < n) | (i ≤ n) do
  (i := i + 1 | skip);  /* i ≅ i */
  ⌊ p := new Node ⌋; connect p;
  ⌊ p.val := mf(i) ⌋;  /* Agr p.val */
  ⌊ p.nxt := t.head ⌋;
  ⌊ t.nds := t.nds ∪ {p} ⌋;
  (skip | i := i + 1);
od;
⌊ result := t ⌋;
```

(c) Biprogram CC_{tabu}

Fig. 16. Two implementations of tabulate, and a biprogram weaving them together.

recursive definitions involved. The containment, together with invariant $t.head \in t.nds$, implies that everything reachable from $t.head$ is in $t.nds$. It does not say that $t.nds$ is exactly the reachable set, though it will be; we do not need that stronger fact.

Method mf has an integer parameter x and returns an integer result. Its unary spec is $true \rightsquigarrow true$ [•], which says very little but the empty frame condition says it has no effect on the heap or global variables. In particular, it does no allocation, since otherwise its frame condition would have to include rw alloc. Implicitly it is allowed to read its parameter x and write its *result*, as we saw in Example 3.5. As relational spec, we use $\mathbb{A}x \approx\!> \mathbb{A}result$ [•], which expresses determinacy as self-equivalence in a way that is local: it refers only to locations that may be read or written. It is this relational spec, and nothing more, that we wish to use for mf in relational reasoning about tabulate.

For tabulate, the frame condition is [rw alloc]. It allocates, which implicitly updates the special variable alloc by adding the newly allocated reference; the new value of alloc depends on its old value, so the frame condition says alloc may be both read and written. Like method mf, method tabulate reads its parameter and writes its result, but neither reads nor writes any other preexisting locations.

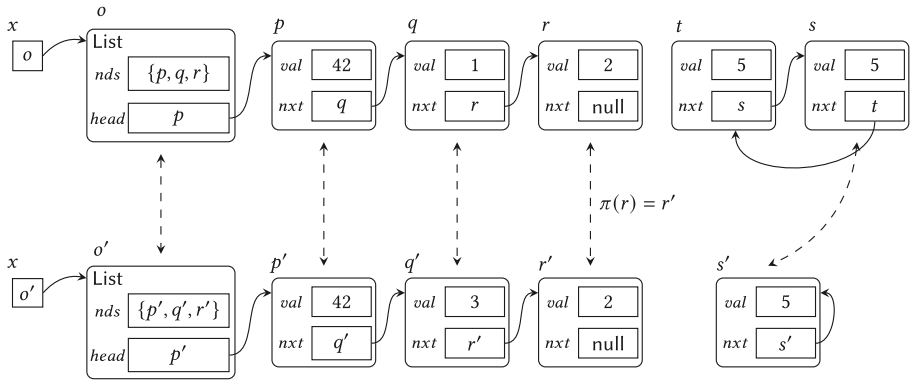
Although we aim to prove equivalence of the two versions of tabulate without recourse to a precise functional spec, we do include a postcondition that constrains *nds*, as this plays a role in specifying equivalence. The postcondition says *nds* contains *head* and is closed under *next*; formally: $result.nds`next \subseteq result.nds$ and $result.head \in result.nds$.

To express equivalence of the two versions, the (relational) precondition is agreement on what is readable, namely, the parameter n. The agreement formula $\mathbb{A}n$, or equivalently $n \cong n$, simply means the two initial states have the same value for n. We do not assume agreement on alloc; we want the equivalence to encompass initial states without constraint on allocated but irrelevant objects.

For the postcondition, we want agreement on what is writable (aside from alloc), thus $\mathbb{A}result$. We also specify that the unary postcondition holds in both final states:

$$\mathbb{B}(result.nds`next \subseteq result.nds \wedge result.head \in result.nds). \tag{10}$$

But *result* is just a reference to newly allocated list structure. To express that the two result lists have the same content, we need more than $\mathbb{A}result$. A first guess is the agreement formula $\mathbb{A}result.nds`val$. The formula uses syntax sugar, to abbreviate $\mathbb{A}\{result\}`nds`val$. Agreement formulas, as mentioned in Section 2.3, are interpreted with respect to a refperm, that is, a type-respecting partial bijection on references of the two states. Whereas $\mathbb{A}n$ means identical values for



left-expression	l-value in σ	r-value in σ
x	$\{x\}$	o
$\{x\}$‘nds	$\{o.nds\}$	$\{p, q, r\}$
$\{x\}$‘nds‘val	$\{p.val, q.val, r.val\}$	$\emptyset$
$\{x\}$‘nds‘nxt	$\{p.nxt, q.nxt, r.nxt\}$	$\{q, r, null\}$

$\sigma(\text{alloc}) = \{o, p, q, r, s, t\}$
$\sigma'(\text{alloc}) = \{o', p', q', r', s'\}$
$\pi = \{(o, o'), (p, p'), (q, q'), (r, r'), (s, s')\}$

$\sigma|\sigma' \models_\pi \mathbb{A}x$ is true because $o \overset{\pi}{\sim} o'$
$\sigma|\sigma' \models_\pi \mathbb{A}\{x\}$‘$nds$ is true because $o \overset{\pi}{\sim} o'$ and $\{p, q, r\} \overset{\pi}{\sim} \{p', q', r'\}$
$\sigma|\sigma' \models_\pi \mathbb{A}\{x\}$‘$nds$‘$nxt$ is true; note $p.nxt \overset{\pi}{\sim} p'.nxt$, $q.nxt \overset{\pi}{\sim} q'.nxt$, and $r.nxt \overset{\pi}{\sim} r'.nxt$
$\sigma|\sigma' \models_\pi \mathbb{A}\{x\}$‘$nds$‘$val$ is false because $\sigma(q.val) = 1 \neq 3 = \sigma'(q'.val)$
$\sigma|\sigma' \models_\pi \{x\}$‘$nds \doteq \{x\}$‘$nds$ is true because $\{p, q, r\} \overset{\pi}{\sim} \{p', q', r'\}$, regardless of whether (o, o') is in π

Fig. 17. Refperm π and relations between two states, σ, σ' with variable x (see Example 4.4).

integer n, the formula $\mathbb{A}result$ means equivalent reference values, i.e., connected via the bijection. The formula $\mathbb{A}result.nds$‘val says that for pairs o, o' of references connected by the bijection, with $o \in result.nds$, the fields $o.val$ and $o'.val$ have equal contents (equal because the type is integer).

To fully constrain the lists to have the same structure, we use this postcondition:

$$\Diamond(\mathbb{A}result \wedge \mathbb{A}result.nds \wedge \mathbb{A}result.nds\text{‘}next \wedge \mathbb{A}result.nds\text{‘}val). \tag{11}$$

Here $\Diamond$ says there exists some refperm. The formula $\mathbb{A}result.nds$ abbreviates $\mathbb{A}\{result\}$‘nds and says the refperm cuts down to a (total) bijection between the regions $result.nds$ in the two states. The condition $\mathbb{A}result.nds$‘$next$ says that bijection is compatible with the linked list structure.

The semantics of relation formulas is formalized in Section 7.1. It is a little subtle: $\{x\}$‘$f \doteq \{x\}$‘f is different from $\mathbb{A}\{x\}$‘f, unless guarded by $\mathbb{A}x$ (as a conjunct or antecedent). We invariably use such guarded formulas, e.g., conjuncts in Equation (11) and antecedents in the coupling of Example 4.3.

Example 4.4. To illustrate the meaning of agreement formulas like those in Equation (11), Figure 17 shows an example of two states with a single variable x : List, and using $\{x\}$‘nds rather than its sugared form $x.nds$. The semantic notations are defined in Section 7.1 but the picture is meant to be understandable now. The values of some left-expressions are given; we consider the l-value of any left-expression to be a set of locations, such as the single location x (a variable name) and $p.val$ (a heap location).

Taken together, Equations (10) and (11) say the results from tabulate are lists for which the nodes can be put in bijective correspondence that is compatible with the nxt pointers and for which

corresponding elements have the same value. They serve as postcondition, with precondition $\mathbb{A}n$, to specify equivalence for tabulate. What else would we mean by equivalence of the programs? We do not want to say they have literally identical values, because we want equivalence to be local: It should not involve what else may have been allocated, so we do not assume agreement on alloc. Hence, the resulting lists may not have identical reference values. What matters is that the heap data produced by the two implementations has the same structure.

On the modality $\Diamond$. The modal operator $\Diamond$ is needed for the relational postcondition (11) and in any spec where allocation is possible. We gloss over it in some examples, but specs of interest usually have this standard form: $\mathcal{R} \approx\!> \Diamond\mathcal{S}\,[\varepsilon]$ where $\Diamond$ does not occur in $\mathcal{R}$ or $\mathcal{S}$. The tabulate spec can be put in standard form, because Equation (10) expresses unary conditions, with no dependence on refperm, so that formula can be put inside the $\Diamond$ in Equation (11).

While SMT solvers typically provide some heuristic support for quantifiers, existential quantifiers are problematic, and we cannot expect a solver to find witnesses for the existential expressed by $\Diamond$. In the WhyRel prototype, specs do not include $\Diamond$ explicitly. Instead, a refperm is maintained in ghost state, thus witnessing the existential. A ghost instruction, connect – with –, can be used to designate which references the user wants to be considered as corresponding. For example, the biprogram Figure 16(c) uses connect p, which abbreviates connect p with p, to add newly allocated Node references to the refperm, thereby establishing $p \doteqdot p$. The general form of connect caters for programs using different variables.

Alignment for tabulate. Recall that Equations (10) and (11) are meant to comprise the postcondition of a spec to relate the bodies, *tabu* and *tabu′*, of the two implementations of tabulate in Figure 16(a) and (b). To say that they satisfy the relational spec, we use a judgment like this:

$$\Phi \vdash (tabu|tabu') : \mathbb{A}n \approx\!> \mathcal{R}\ [\mathsf{rw}\,\mathsf{alloc}], \quad \text{where } \mathcal{R} \text{ is Equations (10)}\wedge\text{(11)}.$$

The hypothesis context specifies mf; Φ is a triple, with $\Phi_2(\mathsf{mf})$ being the relational spec $\mathbb{A}x \approx\!> \mathbb{A}result$. The unary specs $\Phi_0(\mathsf{mf})$ and $\Phi_1(\mathsf{mf})$ are not relevant to this example.

We derive the judgment for $(tabu|tabu')$ from a judgment with the same spec for the more conveniently aligned biprogram CC_{tabu} in Figure 16(c), in a way that will be justified in Section 4.5. Several features of CC_{tabu} are important. First, its left and right syntactic projections are the two commands, *tabu* and *tabu′*, to be related; semantically it represents pairs of their executions, aligned in a particular way. Second, the calls to mf are in the sync'd form, which signals that reasoning is to be done using the relational spec of mf. A comment in the biprogram indicates that we get agreement on $p.val$ following the calls to $\mathsf{mf}(i)$, in virtue of that spec. Similarly, the two allocations are also in the sync'd form and followed by the connect ghost operation, achieving agreement on the allocated references. In the proof system, there is a rule for sync'd allocations, with postcondition that yields for example $\Diamond\mathbb{A}p$ for the Node allocation. Using this rule (or the connect ghost operation) is a good choice in the present example, but in general it is not necessary to connect allocations, even if they happen to be aligned; this is important when relating programs that are not building the same heap structure, or when proving noninterference and reasoning about branches with tests that depend on secrets. Finally, the bi-while in CC_{tabu} signals that we reason in terms of lockstep alignment of the loop iterations. This enables us to reason that the two executions are building isomorphic pointer structures, using a relational invariant similar to the postcondition of the relational spec (11), conjoined with a simple relation between the counter variables:

$$i - 1 \doteqdot i \wedge \mathbb{A}n \wedge \mathbb{A}t \wedge \mathbb{A}t.nds \wedge \mathbb{A}t.nds\text{`}nxt \wedge \mathbb{A}t.nds\text{`}val.$$

The biprogram provides a convenient alignment but incurs an additional proof obligation: the invariant must imply that the loop tests agree, as otherwise it would be unsound to assume the

iterations can be considered to be aligned in lockstep. Indeed, the implication is valid: $\mathbb{A}n$ and $i - 1 \doteq i$ implies $i < n \doteq i \leq n$.

In summary, this example shows biprograms express alignment of the programs under consideration to facilitate procedure-modular reasoning using relational specs and to facilitate the use of simpler relational invariants for loops. In passing, we introduced ways to express relations on pointer structures, abstracting from specific addresses (as appropriate for Java- and ML-like languages) and making it possible to specify relations where some parts of the heap are meant to have isomorphic structure while other parts may be entirely different. There are at least two important use cases for such differences: encapsulated data structures, when relating implementations of a module interface, and structure manipulated by "secret" computations, when proving information flow properties.

The example happens to work well with close alignment of the program structure and agreement on all the data involved. The logic must handle aligned allocation in a loop, as in this example. It must also handle differing allocations, for example to relate programs using different encapsulated data representations. Differing allocations also arise when proving noninterference, in cases where allocation occurs under high branch conditions.

The proof rules used to derive a relational modular linking rule like Equation (3) make use of a general form of local equivalence specification, derived from the frame condition of a unary spec (and defined in Section 8.1). But it is also possible to express local equivalence notions suited to specific situations, as in the example, and it is possible to work with differing program structures as illustrated in some case studies (e.g., Figure 19 and Section 4.6).

4.5 Defining and Using Biprogram Weaving for Alignment

In this subsection, we define the weaving relation on biprograms. The purpose of the weaving relation is to connect a bi-com $(C|C')$, that expresses a relational verification problem, with a more tightly aligned version that facilitates reasoning. If $(C|C')$ weaves to DD, written $(C|C') \rightsquigarrow DD$, then the syntactic projections of DD are C and C', so DD models executions of the two commands. The weaving relation $\rightsquigarrow$ is used in a proof rule that realizes the product principle: any judgment that holds for DD also holds for $(C|C')$, given $(C|C') \rightsquigarrow DD$. In general, weaving brings together similarly structured subprograms, introducing additional alignment points while preserving syntactic projections. In addition to defining the relation $\rightsquigarrow$, the rest of this section gives examples of its use, and sketches the semantic considerations that justify the proof rule and explain the orientation of the relation.

The ***weaving relation*** $\rightsquigarrow$ is defined inductively by axioms and congruence rules in Figure 18. The axioms replace a bi-com by another biprogram form including those that can assert agreements (bi-if and bi-while). The congruence rules, displayed as one rule with multiple conclusions, allow weaving in all contexts except the procedure bodies in bi-let. Apropos congruence for bi-let, note that bi-let does not bind general biprograms but only pairs of commands despite the appearance of the concrete syntax (see Figure 5).

The weaving that introduces bi-while allows the introduction of so-called alignment guards. The biprogram CC_{tabu} omits them (Figure 16(c)), which is syntax sugar taking them to be false. As an example of their use, later in this subsection, we follow up on the example program (4) discussed in Section 2.1, sketching the three-premise relational loop rule that enables verification of the example using a simple invariant.

Example 4.5. The sequence weaving axiom (second line of Figure 18) can be used for an example mentioned in Section 2.3, namely, (c.val:= v | c.f:= −v); (return c.val | return −c.f). For the bi-com $(a; b; c \mid d; e; f)$ (temporarily using lower case letters for atomic commands), there are four different

$(A|A) \rightsquigarrow \lfloor A \rfloor$

$(C; D \mid C'; D') \rightsquigarrow (C|C'); (D|D')$

$(\text{if } E \text{ then } C \text{ else } D \mid \text{if } E' \text{ then } C' \text{ else } D') \rightsquigarrow \text{if } E|E' \text{ then } (C|C') \text{ else } (D|D')$

$(\text{while } E \text{ do } C \mid \text{while } E' \text{ do } C') \rightsquigarrow \text{while } E|E' \cdot \mathcal{P}|\mathcal{P}' \text{ do } (C|C')$

$(\text{let } m = B \text{ in } C \mid \text{let } m = B' \text{ in } C') \rightsquigarrow \text{let } m = (B|B') \text{ in } (C|C')$

$(\text{var } x{:}T \text{ in } C \mid \text{var } x'{:}T' \text{ in } C') \rightsquigarrow \text{var } x{:}T|x'{:}T' \text{ in } (C|C')$

$$\frac{BB \rightsquigarrow CC}{\begin{array}{c} BB; DD \rightsquigarrow CC; DD \qquad DD; BB \rightsquigarrow DD; CC \qquad \text{if } E|E' \text{ then } BB \text{ else } DD \rightsquigarrow \text{if } E|E' \text{ then } CC \text{ else } DD \\ \text{if } E|E' \text{ then } DD \text{ else } BB \rightsquigarrow \text{if } E|E' \text{ then } DD \text{ else } CC \\ \text{while } E|E' \cdot \mathcal{P}|\mathcal{P}' \text{ do } BB \rightsquigarrow \text{while } E|E' \cdot \mathcal{P}|\mathcal{P}' \text{ do } CC \qquad \text{let } m = (B|B') \text{ in } BB \rightsquigarrow \text{let } m = (B|B') \text{ in } CC \\ \text{var } x{:}T|x'{:}T' \text{ in } BB \rightsquigarrow \text{var } x{:}T|x'{:}T' \text{ in } CC \end{array}}$$

Fig. 18. Axioms and congruence rules that define the weaving relation $\rightsquigarrow$. Recall A ranges over atomic commands (Figure 5).

alignments that can be obtained by a single application of sequence weaving[25]:

$$\begin{array}{l} (a; b; c|d; e; f) \rightsquigarrow (a; b|d); (c|e; f), \\ (a; b; c|d; e; f) \rightsquigarrow (a|d; e); (b; c|f), \\ (a; b; c|d; e; f) \rightsquigarrow (a; b; c|\text{skip}); (\text{skip}|d; e; f), \\ (a; b; c|d; e; f) \rightsquigarrow (\text{skip}|d; e; f); (a; b; c|\text{skip}). \end{array} \tag{12}$$

These weavings introduce a semicolon at the biprogram level, which makes it possible to assert a relation at that point. Different weavings of the same biprogram serve to align different intermediate points.

Using the sequence axiom and congruence, we have $(a; b; c|d; e; f) \rightsquigarrow (a|d); (b; c|e; f) \rightsquigarrow (a|d); (b|e); (c|f)$, which illustrates how fine-grained alignment can be achieved when desired. We also have $(tabu|tabu') \rightsquigarrow^* CC_{tabu}$, which connects $tabu, tabu'$ to the particular alignment we choose for reasoning about them.

As noted earlier, the bi-if and bi-while forms are meant to designate reasoning in which it will be shown that the test conditions are in agreement. Technically, we define small step semantics for biprograms, in which these forms can have a fault—dubbed ***alignment fault***—if the tests are not in agreement. This can be seen as a kind of assertion failure. As an example, recall the implementation of insert in the PQ module in Figure 4. Part of the alternate implementation using sentinels (mentioned in Example 3.4) is shown in Figure 19. We weave the two conditionals using a bi-if, which introduces the possibility of alignment fault. We can use this weaving, because our coupling relation will ensure that $self.head = \text{null}$ in the left state just when $self.head = self.sntnl$ on the right.

Use of bi-if or bi-while incurs additional proof obligations that ensure the absence of alignment fault, which in turn implies that the designated alignment covers all pairs of executions of the underlying programs. The weaving transformations can introduce the bi-if and bi-while forms but not eliminate them; nor can they eliminate any other faults. For example, $(\text{if } x > 0 \text{ then } y.f := x \text{ else skip} \mid \text{if } x > 0 \text{ then } y.f := x \text{ else skip})$ weaves to $\text{if } x > 0|x > 0 \text{ then } (y.f := x \mid y.f := x) \text{ else } \lfloor \text{skip} \rfloor$, noting that $(\text{skip}|\text{skip}) \equiv \lfloor \text{skip} \rfloor$. Both biprograms can fault due to null dereference, but the second also faults in a pair of states where $x > 0$ on one side but $x \leq 0$ on the other.

[25]Keep in mind the syntactic equivalences in Figure 6, which enable these different weavings.

```
result := new Pnode(val, key);
result.sibling := self.sntnl;
result.child := self.sntnl;
result.prev := self.sntnl;
self.rep := self.rep ∪ {result};
if (self.head = self.sntnl) then
   self.head := result;
else
   self.head := link(self,self.head,result);
fi;
self.size := self.size + 1;
```

```
⌊ result := new Pnode(val, key) ⌋;
( skip
| result.sibling := self.sntnl;
  result.child := self.sntnl;
  result.prev := self.sntnl );
⌊ self.rep := self.rep ∪ {result} ⌋;
if (self.head = null | self.head = self.sntnl) then
   ⌊ self.head := result ⌋;
else
   ⌊ self.head := link(self,self.head,result) ⌋;
fi;
⌊ self.size := self.size + 1 ⌋;
```

Fig. 19. Body of alternative implementation of PQ's insert (left) and woven biprogram (right).

$$
\begin{array}{lcll}
\|A\| & \triangleq & \lfloor A \rfloor & \text{(atomic commands)} \\
\|C; D\| & \triangleq & \|C\|; \|D\| & \\
\|\text{if } E \text{ then } C \text{ else } D\| & \triangleq & \text{if } E|E \text{ then } \|C\| \text{ else } \|D\| & \\
\|\text{while } E \text{ do } C\| & \triangleq & \text{while } E|E \cdot \text{false}|\text{false do } \|C\| & \\
\|\text{let } m = B \text{ in } C\| & \triangleq & \text{let } m = (B|B) \text{ in } \|C\| & \\
\|\text{var } x{:}T \text{ in } C\| & \triangleq & \text{var } x{:}T|x{:}T \text{ in } \|C\| &
\end{array}
$$

Fig. 20. Full alignment.

Suppose DD can be obtained from CC by a sequence of weavings, i.e., $CC \hookrightarrow^* DD$. The relation $\hookrightarrow$ can introduce the possibility of additional alignment faults, but it cannot eliminate such possibility. In this sense, $\hookrightarrow$ is oriented (and not symmetric). A consequence is the following: if, under some precondition, DD has no faults, then under that precondition the executions of DD cover all those of CC. This is the gist of the argument for soundness of the following proof rule:

$$\text{from} \quad BB : \mathcal{R} \approx\!> \mathcal{S}\,[\varepsilon] \quad \text{infer} \quad (C|C') : \mathcal{R} \approx\!> \mathcal{S}\,[\varepsilon] \quad \text{provided} \quad (C|C') \hookrightarrow^* BB. \tag{13}$$

(See rule rWeave in Figure 30.) It is this rule that yields a relational judgment for $(tabu|tabu')$ from the same judgment for CC_{tabu} (Figure 16).

In general, a biprogram may admit several possible weavings. For the form $(C|C)$ relating C to itself there is a biprogram that is maximal in the sense that it allows us to reason about two executions aligned in lockstep. We write $\|C\|$ for the ***full alignment*** defined in Figure 20. Apropos linking, we have (let $m = B$ in C | let $m = B'$ in C) $\hookrightarrow^*$ let $m = (B|B')$ in $\|C\|$. Full alignment plays a key role in deriving the relational modular linking rule that was sketched as Equation (3) and is formalized in Figure 31.

Lemma 4.6. $(\overleftarrow{CC}|\overrightarrow{CC}) \hookrightarrow^* CC$ *for any* CC.

As a corollary, we have $(C|C) \hookrightarrow^* \|C\|$ for any C, because $\overleftarrow{\|C\|} \equiv \overrightarrow{\|C\|} \equiv C$.

Sumpub: illustrating conditionally aligned loops. For the tabulate example it is effective to reason by aligning all iterations of the two loops in lockstep. This is not the case for program (4) in Section 2.1, recalled here:

sumpub : s:=0; p:=head; while p ≠ null do if p.pub then s:=s+p.val fi; p:=p.nxt od

It sums the elements of a list that are flagged public. It has an information flow property: the output, in variable s, depends only on the public elements of the input list. (This can be viewed as a declassification or as a value-dependent classification [4].) Typically such properties are expressed using a precondition of agreement on some expression, which in this case should denote "the public elements of the input list."

As a pointer structure, the list can have cycles, so care needs to be taken in defining predicates and functions. In the tabulate example, we choose specs that do not involve inductively defined predicates or relations. Here, we inductively define a predicate $listpub(p, ls)$ that says ls is the list of values of the public elements in a null-terminated list from p:

$$\begin{array}{lcl} p = null & \Rightarrow & listpub(p, []), \\ p \neq null \wedge \neg p.pub \wedge listpub(p.nxt, ls) & \Rightarrow & listpub(p, ls), \\ p \neq null \wedge p.pub \wedge p.val = h \wedge listpub(p.nxt, ls) & \Rightarrow & listpub(p, h :: ls). \end{array}$$

We consider the following relational spec, eliding the frame condition for clarity. The bound variables, ls, ls' are of the math type int list:

$$\exists ls : \text{int list} \mid ls' : \text{int list}.\ \{\!|listpub(head, ls)|\!\} \wedge \{\!|listpub(head, ls')|\!\} \wedge ls \doteq ls' \approx\!> \mathbb{A}s.$$

The syntax of quantifiers in relation formulas explicitly designates left- and right-side variables, which is important in case of reference or region type (since the values must be allocated in the respective states). There is no need to use distinct names here, so we can use a more succinct precondition for the spec: $\exists ls|ls.\ \mathbb{B}(listpub(head, ls)) \wedge \mathbb{A}ls$.

We want to prove that $(sumpub|sumpub)$ satisfies the relational spec. One way is to first prove unary judgment $sumpub : listpub(p, ls) \rightsquigarrow s = sum(ls)$, again treating ls as spec-only, and thus universally quantified over the spec. A simple embedding rule (rEmb in Figure 30) lifts this to $(sumpub|sumpub) : \mathbb{B}(listpub(p, ls)) \approx\!> \mathbb{B}(s = sum(ls))$. The relational frame rule lets us conjoin agreement on ls, to get

$$(sumpub|sumpub) : \mathbb{B}(listpub(p, ls)) \wedge \mathbb{A}ls \approx\!> \mathbb{B}(s = sum(ls)) \wedge \mathbb{A}ls.$$

The postcondition implies $\mathbb{A}s$, so we complete the proof using the relational consequence rule.

Lifting unary judgments is an important pattern of reasoning and is satisfactory for reasoning about assignment commands including those in the tabulate example. But *sumpub* has a loop, so this argument comes at the cost of proving functional correctness, i.e., the judgment $sumpub : listpub(p, ls) \rightsquigarrow s = sum(ls)$. Finding a loop invariant is not difficult in this case, but it would be if sum is replaced by a sufficiently complex computation.

There is an alternative proof of the relational spec that avoids functional correctness, using for the loops a simple relational invariant:

$$\exists xs|xs.\ \mathbb{B}(listpub(p, xs)) \wedge \mathbb{A}xs \wedge \mathbb{A}s. \tag{14}$$

We verified the example using WhyRel, and instead of asking the solvers to handle the existential, we used the standard technique: xs on each side is a ghost variable, initialized based on the precondition and explicitly updated as appropriate.

The point of this example is that this simple invariant only suffices if we align the iterations judiciously. In case $p.pub$ holds on both left and right, we take a lockstep iteration, i.e., both sides execute the loop body, and it is straightforward to show the invariant holds afterwards using the last clause in the definition of *listpub* and the fact that $\mathbb{A}xs$, i.e., equality of the mathematical lists, implies agreement on their tails. If *pub* is true on one side but not the other, then lockstep iteration does not preserve Equation (14). However, if $p.pub$ is false on the left, then $listpub(p, xs)$ implies $listpub(p.nxt, xs)$, and executing the body just on the left maintains the relation Equation (14). Notice Equation (14) does not include agreement on p; indeed the precondition requires no agreement on references. *Mutatis mutandis on the right side.* To express this reasoning, we

weave (*sumpub*|*sumpub*) to this biprogram:

$$\begin{array}{l}(s := 0; p := \mathit{head} \mid s := 0; p := \mathit{head});\\ \textsf{while } p \neq \textsf{null} \mid p \neq \textsf{null} \;.\; \{\!|\neg p.\mathit{pub}|\!\} \mid |\!\{\neg p.\mathit{pub}\}\!| \textsf{ do}\\ \quad (\textsf{ if } p.\mathit{pub} \textsf{ then } s := s + p.\mathit{val} \textsf{ fi}; p := p.\mathit{nxt}\\ \quad \mid \textsf{if } p.\mathit{pub} \textsf{ then } s := s + p.\mathit{val} \textsf{ fi}; p := p.\mathit{nxt} \;) \textsf{ od}\end{array} \tag{15}$$

Although the program is being related to itself, we do not bother to fully align the initialization or loop body: these do not involve allocation or method calls, so reasoning about those parts of the code is straightforward. For this reason, some uses of sync in Figure 16(c) could as well be bi-coms. What is important is to use a bi-while. For loop alignment guards, we choose the relation formulas $\{\!|\neg p.\mathit{pub}|\!\}$ and $|\!\{\neg p.\mathit{pub}\}\!|$. The alignment guards are used in the proof rule for bi-while, which has the following form:

$$\frac{\begin{array}{ll}\vdash CC : Q \wedge \neg\mathcal{P} \wedge \neg\mathcal{P}' \wedge \{\!|E|\!\} \wedge |\!\{E'\}\!| \approx\!> Q & \vdash (\overleftarrow{CC}|\textsf{skip}) : Q \wedge \mathcal{P} \wedge \{\!|E|\!\} \approx\!> Q\\ \vdash (\textsf{skip}|\overrightarrow{CC}) : Q \wedge \mathcal{P}' \wedge |\!\{E'\}\!| \approx\!> Q & Q \Rightarrow E \doteq E' \vee (\mathcal{P} \wedge \{\!|E|\!\}) \vee (\mathcal{P}' \wedge |\!\{E'\}\!|)\end{array}}{\vdash \textsf{while } E|E' \cdot \mathcal{P}|\mathcal{P}' \textsf{ do } CC : Q \approx\!> Q \wedge \{\!|\neg E|\!\} \wedge |\!\{\neg E'\}\!|} \tag{16}$$

This rule has omissions! For clarity, we omit details not relevant to the current discussion: frame conditions, hypothesis context, and side conditions that enforce encapsulation and immunity. The encapsulation condition is discussed later and is lifted from the unary logic, as is *immunity*, a technical condition needed for stateful frame conditions (adapted unchanged from RLI).

In the rule, Q is the relational loop invariant, like Equation (14) in the example. The three premises cover a lockstep iteration, a left-side iteration, and a right-side iteration. The one-sided iterations are expressed using the syntactic projection metafunctions (Figure 13) to obtain unary commands. In the example the two projections of the loop body are the same, namely, if p.pub then s := s+p.val; fi; p := p.nxt. In each premise the invariant must be preserved, but each has a strengthened precondition based on the alignment guards. For the example, the first premise applies when both sides are at a public element. The second (respectively, third) premise applies when the element on the left (respectively, right) is not public. Besides alignment guards, the premises include the loop tests in the usual way, as does the conclusion of the rule.

The side condition, $Q \Rightarrow E \doteq E' \vee (\mathcal{P} \wedge \{\!|E|\!\}) \vee (\mathcal{P}' \wedge |\!\{E'\}\!|)$, ensures that for any initial states satisfying Q, at least one of the three premises is applicable. The reader can confirm that the side condition holds in the example, and thus the rule can be used to carry out the proof as described.

As another example, for tabulate in Figure 16(c), we use false alignment guards, so the one-sided premises hold trivially and the side condition simplifies to the implication mentioned earlier: the invariant implies agreement on loop tests. That is, $i - 1 \doteq i \wedge \mathbb{A}n \Rightarrow i < n \doteq i \leq n$.

The biprogram syntax allows $\mathcal{P}$ and $\mathcal{P}'$ to be relation formulas, but it happens that in the example $\{\!|\neg p.\mathit{pub}|\!\}$ only constrains the left state and the other alignment guard constrains the right state. As stated in Section 3.1, $\mathcal{P}$ and $\mathcal{P}'$ are not allowed to have agreement formulas; it is not evident what refperm would be used to interpret agreements in such a context.

4.6 Relational Reasoning with Hiding and Encapsulation

Having illustrated general relational reasoning (Sections 4.4 and 4.5) and the use of dynamic framing for encapsulation in unary reasoning (Section 3.5), we now illustrate encapsulation in relational reasoning. In doing so, we sketch how requirements (E1)–(E4) adapt to the relational setting.

In Section 3.5, we considered the verification of a client linked with a quick-find implementation of UnionFind, hiding the private invariant. Here, we consider two implementations of that interface and consider a more interesting client: an implementation, *MST*, of Kruskal's minimum

spanning tree algorithm. For a second implementation of UnionFind, we consider the quick-union data structure [88].

The goal is to prove a relational property: equivalence of the two programs made by linking *MST* with the two module implementations. To do so, we use relational modular linking, as sketched in the rule (3), hiding a coupling relation between the two implementations, which includes their private invariants. To use the rule, we do the following:

(i) Prove a unary judgement for *MST*, with the UnionFind specs in context. As explained in Section 3.5, this ensures that *MST* respects the boundary of UnionFind, as per requirement (E3).
(ii) Define a coupling relation $\mathcal{M}_{uf}$ to connect the encapsulated data structures of the two implementations of UnionFind. Show that it is framed by the dynamic boundary, as per requirement (E2), and follows from the *MST* precondition, as per (E4).
(iii) For the two bodies B, B' that provide alternate implementations of find, prove a relational judgment for $(B|B')$ (and likewise for the implementations of union). The specification should express local equivalence, but with $\mathcal{M}_{uf}$ conjoined to the pre- and postcondition.

It then follows that the two linkages satisfy a local equivalence property, specifically a relational spec that is derived by a general construction from the unary spec of *MST*. Similar to the relational spec of tabulate in Section 4.4, it requires agreement on inputs and ensures agreement on outputs. But encapsulation must be taken into account: the two linkages will be equivalent in terms of client-visible inputs and outputs, but the encapsulated data structures are different. More on this later.

For item (i), we choose *MST* for the sake of a nontrivial example, but we do not use a functional correctness spec, i.e., we do not specify that it produces a minimum spanning tree. All we need is a precondition under which *MST* does not fault, and a frame condition. The global variables of *MST* are g of type Graph and es of type List. For simplicity, g is an abstract mathematical graph; es references a list like that used in Section 4.4. The graph interface provides an enumeration of edges and *MST* produces, in es, a list of edge numbers for edges in the spanning tree:

$$\mathsf{numVerts}(g) > 0 \wedge pool = \varnothing \rightsquigarrow \mathsf{true}\ [\mathsf{rd}\, g; \mathsf{rw}\, es, \mathsf{alloc}, pool, (pool \cup pool\text{`}rep\text{`}\mathsf{any}]. \tag{17}$$

Note that the effects here include effects produced by call to UnionFind methods. We verify the judgment $\Phi_{uf} \vdash_{\bullet} MST : spec$ where *spec* is Equation (17) and Φ_{uf} has the public specs of find and union, i.e., without the private invariants. The current module is $\bullet$, the default module with empty boundary.

The local equivalence spec for the two linked programs is derived, by a general construction called *locEq*, based on the frame condition of a unary spec, and the dynamic boundaries of the modules in scope. In the example there is just one module with a nontrivial boundary, UnionFind; math modules like Graph have empty boundaries. Agreements in the precondition are derived directly from the read effects and boundary, using the effect subtraction operator that excludes from agreement the encapsulated locations. In this example, the relational precondition is

$$\mathbb{B}(\mathsf{numVerts}(g) > 0 \wedge pool = \varnothing) \wedge \mathbb{B}(s_{\mathsf{alloc}} = \mathsf{alloc}) \wedge \mathbb{A}es.$$

The conjunct $\mathbb{B}(s_{\mathsf{alloc}} = \mathsf{alloc})$ introduces snapshot variable s_{alloc} to be used in the postcondition to express freshness. The agreement $\mathbb{A}es$ is in simplified form. The general construction takes the read effect, $\mathsf{rd}\, es, \mathsf{alloc}, pool, (pool \cup pool\text{`}rep\text{`}\mathsf{any}$ and subtracts the boundary $\mathsf{rd}\, pool, (pool \cup pool\text{`}rep\text{`}\mathsf{any}$ and alloc, which results in the effect $\mathsf{rd}\, es, ((pool \cup pool\text{`}rep\text{`}\mathsf{any}) \setminus (pool \cup pool\text{`}rep\text{`}\mathsf{any}))$, which trivially simplifies to $\mathsf{rd}\, es, \varnothing\text{`}\mathsf{any}$ and then to $\mathsf{rd}\, es$.

What about agreements for a postcondition? In general, a command may write preexisting locations and allocate new ones. In this case the only preexisting locations that are writable are the variables *es* and alloc, so the postcondition includes $\mathbb{A}es$. (In general, to handle writable heap locations the general definition of *locEq* uses snapshots of the relevant expressions in write effects; for details see Section 8.1.) To handle fresh locations, *locEq* uses the snapshot s_{alloc} in the way described in Section 3.5: the fresh references are $\text{alloc}\backslash s_{\text{alloc}}$ so the fresh locations are $(\text{alloc}\backslash s_{\text{alloc}})\text{`any}$. Again, effect subtraction is used to exclude alloc and the boundary. The resulting agreement is $\mathbb{A}((\text{alloc}\backslash s_{\text{alloc}})\backslash(pool \cup pool\text{`}rep))\text{`any}$.

In summary, the local equivalence spec that we get from Equation (17) for *MST* is

$$\begin{aligned}&\mathbb{B}(\text{numVerts}(g) > 0 \wedge pool = \varnothing) \wedge \mathbb{B}(s_{\text{alloc}} = \text{alloc}) \wedge \mathbb{A}es\\ &\approx\!\!> \Diamond(\mathbb{B}(\text{true}) \wedge \mathbb{A}es \wedge \mathbb{A}((\text{alloc}\backslash s_{\text{alloc}})\backslash(pool \cup pool\text{`}rep))\text{`any})\ [\ldots].\end{aligned} \tag{18}$$

If one simply wants to know that the new and old versions of the program are the same, aside from encapsulated state, then this is enough. By construction, the *locEq* spec requires agreement on what the program can read and ensures agreement on its results.

In this particular case, to obtain a more explicit postcondition that refers to the list constructed, we can do as follows. First, strengthen the unary postcondition from true to something like $es.head \in es.nds \wedge es.nds\text{`}next \subseteq es.nds \wedge (\{es\} \cup es.nds) \subseteq (\text{alloc}\backslash s_{\text{alloc}})$, which expresses the closure of *nds* and the freshness of the list (see Section 4.4). The relational spec Equation (18) then changes to have these conditions in place of true. Then using the rule of consequence and reasoning about sets, we get $\mathbb{A}es.nds\text{`}next$ and $\mathbb{A}es.nds\text{`}val$ much like in the tabulate example.

For item (ii), as expected since Hoare'72, the coupling relation $\mathcal{M}_{uf}$ conjoins a relational formula that connects the two implementations, together with the two private invariants. In particular, $\mathcal{M}_{uf}$ is $\{\!|I_{qf}|\!\} \wedge [\![I_{qu}]\!] \wedge \ldots$, where I_{qf} is the invariant discussed in Section 3.5, and I_{qu} is the private invariant of the quick-union implementation. The two implementations have similar internal data structure, in the sense that both use an array to represent an up-pointing tree, but quick-find and quick-union manipulate the tree quite differently. To specify the connection between the two data structures, the third conjunct of $\mathcal{M}_{uf}$ is this formula:

$$\mathbb{A}pool \wedge \forall u : \text{Ufind} \in pool | u : \text{Ufind} \in pool.\ \mathbb{A}u \Rightarrow eqPartition(\{\!|u.part|\!\}, [\![u.part]\!]). \tag{19}$$

This says the two pools are in agreement, and for corresponding elements u in the pool, the abstract partition $u.part$ on the left side is an equivalent partition to the one on the right. This means they have the same blocks. This coupling uses a common idiom. The coupling relation is defined using a mathematical abstraction: the two data structures are related if they have the same abstraction. This idiom is especially suitable if the two data structures are very different. By contrast, in our two implementations of PQ, we consider two similar pointer structures, and for their coupling, we use agreement formulas to describe fine-grained correspondence between the two pointer structures; see Example 4.3.

To show that $\mathcal{M}_{uf}$ is framed by the boundary, the technique is essentially the same as for unary framing of an invariant (Section 3.5). The difference is that here we consider a pair of states that satisfy $\mathcal{M}_{uf}$, and a second pair where the two left (respectively, right) states agree on locations within the boundary, to show the second pair satisfies $\mathcal{M}_{uf}$. Given a suitable representation of states, as in our prototype, the implication is easily checked by SMT solvers.

The last part of item (ii) is that $\mathcal{M}_{uf}$ is implied by the precondition of the client spec, in this case (17). To be precise, it is an implication at the level of relations: $\mathbb{B}(numVertices(g) > 0 \wedge pool = \varnothing) \Rightarrow \mathcal{M}_{uf}$. It holds owing to $pool = \varnothing$.

For item (iii), for each method, we verify the local equivalence spec derived from the method's unary spec, with $\mathcal{M}_{uf}$ conjoined to pre- and postcondition. For example, the frame condition of

union is $[\mathsf{rw}(\{self\} \cup self.rep)\text{`any}]$, and its parameters are $self, x, y$. Based on this, *locEq* uses a precondition based on the agreement $\mathbb{A}self \wedge \mathbb{A}x \wedge \mathbb{A}y \wedge \mathbb{A}(\{self\} \cup self.rep)\text{`any}$. A snapshot variable s is used in precondition $\mathbb{B}s = \{self\} \cup self.rep$ so the postcondition can express agreement on writables by $\mathbb{A}s\text{`any}$, in addition to agreement on fresh locations as described for *MST*. Recall that *locEq* then subtracts locations within the boundary; it is not agreement that we want for those locations, but rather the connection expressed by $\mathcal{M}_{uf}$.

The implementations of union and find are fairly different. For quick-find, the union operation eagerly updates "parents" so find takes constant time. For quick-union, find has to traverse multiple parents to reach the representative element. To prove the relational judgments for the method bodies, we use biprograms that are not tightly woven. The corresponding implementations are not very similar and are not making external calls or doing allocation, so there is little motivation for close alignment the way there is for the tabulate example.

More details about the *MST* verification can be found in Section 9. For now, we review why relational modular linking—shown in Equation (3) and formalized in rule rMLink in Figure 31—is sound. In other words, why do (i)–(iii) suffice to prove equivalence of the linkages? Intuitively, the coupling is preserved by client steps owing to encapsulation, just like private invariants in the unary case. This is formalized by a relational version of the SOF rule, called rSOF. For that rule to be sound, the client needs to be aligned so that context calls can be sync'd (like the call to mf in the tabulate example) so a relational spec can be used—namely, a local equivalence spec conjoined with the coupling relation. So rule rSOF applies to the full alignment of some command, and its premise is that this fully aligned biprogram satisfies a local equivalence spec. This we obtain from the unary judgment of (i), by a rule that lifts a unary judgment to a relational one for the local equivalence derived from the unary spec (rule rLocEq in Figure 30). It relates the command to itself, expressing the dependency property of its read effect as a relational judgment.

Notations to conjoin couplings. To conclude this section, we define a metafunction that conjoins a relation to a relational spec; this is used to formulate rSOF and the modular linking rule. It is based on a similar metafunction, $\mathbin{⍉}$, which applies to a unary spec and a unary invariant I:

$$(R \rightsquigarrow S\,[\eta]) \mathbin{⍉} I \;\;\hat{=}\;\; R \wedge I \rightsquigarrow S \wedge I\,[\eta]. \tag{20}$$

This lifts to an operation on unary contexts, written $\Phi \mathbin{⍉} I$, by mapping $⍉I$ over the specs in Φ.

For relation formula $\mathcal{M}$, the operation $⍉\mathcal{M}$ conjoins $\mathcal{M}$ to a relational spec. The operation only applies to relational specs in the ***standard form***, meaning that $\diamond$ occurs only outermost on the postcondition, or not at all.

Definition 4.7 (Conjoin Coupling $⍉\mathcal{M}$). If $\mathcal{R}$ and $\mathcal{S}$ are $\diamond$-free, then

$$(\mathcal{R} \mathrel{\approx\!\!>} \diamond\mathcal{S}\,[\eta]) \mathbin{⍉} \mathcal{M} \;\;\hat{=}\;\; \mathcal{R} \wedge \mathcal{M} \mathrel{\approx\!\!>} \diamond(\mathcal{S} \wedge \mathcal{M})\,[\eta],$$
$$(\mathcal{R} \mathrel{\approx\!\!>} \mathcal{S}\,[\eta]) \mathbin{⍉} \mathcal{M} \;\;\hat{=}\;\; \mathcal{R} \wedge \mathcal{M} \mathrel{\approx\!\!>} \mathcal{S} \wedge \mathcal{M}\,[\eta].$$

For context Φ, let $\Phi \mathbin{⍉} \mathcal{M}$ conjoin $\mathcal{M}$ to the specs in Φ_2 and for the unary specs give $\Phi_0 \mathbin{⍉} \overleftarrow{\mathcal{M}}$ and $\Phi_1 \mathbin{⍉} \overrightarrow{\mathcal{M}}$. In other words, $(\Phi_0, \Phi_1, \Phi_2) \mathbin{⍉} \mathcal{M}$ is $(\Phi_0 \mathbin{⍉} \overleftarrow{\mathcal{M}},\ \Phi_1 \mathbin{⍉} \overrightarrow{\mathcal{M}},\ \Phi_2 \mathbin{⍉} \mathcal{M})$.

Note that $\Phi \mathbin{⍉} \mathcal{M}$ is only defined if the specs in Φ_2 are in standard form, and then so is the result.

5 SEMANTICS OF PROGRAMS AND UNARY CORRECTNESS

For a correctness judgment $\Phi \vdash^{\Gamma}_{M} C : P \rightsquigarrow Q\,[\varepsilon]$, an informal sketch of the semantics is given preceding Definition 3.3. To make it precise, we use transition semantics, so we can formulate the semantics of encapsulation in terms of the module in which a given step is taken, initially module M. To express modular correctness with respect to assumed specs, a context call makes a single

step to the result of the call, given by a ***context model*** φ, which provides denotations that satisfy the specifications of the hypothesis context Φ. Transitions go to ***fault***, $\lightning$, in case of runtime failure (null dereference). Fault is also used to represent precondition violation in context calls.[26]

A ***pre-model*** provides method denotations that do not necessarily satisfy specs; the transition relation $\overset{\varphi}{\longmapsto}$ is defined for any pre-model φ.

For readers familiar with O'Hearn et al. [77] or RLII, we note that unlike those works here, we cannot use a single "most nondeterministic" denotation. We need context models to be quasi-deterministic, in accord with the $\forall\forall$-interpretation of relational correctness for deterministic programs.

This section spells out the details, which are somewhat intricate. The most important and novel part is the semantics of encapsulation, a condition called Encap in the semantics of correctness judgments (Definition 5.10). Some readers may wish to skip to Section 6, after skimming Sections 5.1 and 5.2.

5.1 States, Expressions, Method Environments, and Configurations

Assume given an infinite set *Ref* of references, disjoint from the integers, with distinguished element *null*. A Γ-***state*** comprises a finite heap and a type-respecting assignment of values to the variables in Γ. We confine attention to contexts Γ that include the special variable alloc. We write $\sigma(x)$ to look up the value of x in state σ. In particular, $\sigma(\mathsf{alloc})$ is the finite set of allocated references. Any reference $o \in \sigma(\mathsf{alloc})$ has a class K, which we write as $\mathit{Type}(o, \sigma)$.

A ***location*** is either a variable x or a ***heap location*** $o.f$, where we write $o.f$ for the pair (o, f) of a non-null reference o and field name f. For any state σ, define the set of its locations by

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

The heap provides a type-respecting assignment of values to heap locations. We write $\sigma(o.f)$ for the value of field f of allocated reference o. Type-respecting means that if $\mathit{Type}(o, \sigma)$ is K and $f : T$ is in $\mathit{Fields}(K)$ then $\sigma(o.f)$ is in $[\![T]\!]\sigma$. We write $[\![T]\!]\sigma$ for the values of type T in state σ. In the case of a reference type K, define $[\![K]\!]\sigma$ by

$$[\![K]\!]\sigma \;\hat{=}\; \{\mathit{null}\} \cup \{o \in \sigma(\mathsf{alloc}) \mid \mathit{Type}(o, \sigma) = K\}.$$

Define $[\![\mathsf{rgn}]\!]\sigma$ to be $\mathbb{P}(\sigma(\mathsf{alloc}) \cup \{\mathit{null}\})$. We write $[\![\Gamma]\!]$ for the set of Γ-states.

The transition semantics of a command typed in Γ may introduce additional variables for local blocks, so it is convenient to define $\mathit{Vars}(\sigma)$ to be the variables of the state. We write $[\sigma{+}x{:}\, v]$ to extend the state with additional variable x with value v, and $[\sigma \mid x{:}\, v]$ to override the value of x that is already in $\mathit{Vars}(\sigma)$. We write $\sigma \upharpoonright x$ to remove x from the domain of σ.

We write $\sigma(F)$ for the value of expression F. The semantics of program expressions E and region expressions G is in Figure 21. (To be very precise, the semantics of expressions is defined on a typing $\Gamma \vdash F : T$, such that $\sigma(F)$ is in $[\![T]\!]\sigma$.) The syntax is designed to avoid undefinedness. We are not formalizing arithmetic operators that can fail, there are no dangling pointers, and program expressions E do not depend on the heap. Region expressions can depend on the heap, in the case of images $G`f$, and they are defined in any state. If $f{:}K$ for some $K$, then $\sigma(G`f)$ is the set of values of the f fields of objects in $\sigma(G)$. If $f{:}\mathsf{int}$, then $\sigma(G`f)$ is empty. Finally, for $f{:}\mathsf{rgn}$, $\sigma(G`f)$ is the union of the regions $\sigma(o.f)$ for o in $\sigma(G)$.

Transitions relate configurations of the form $\langle C, \sigma, \mu\rangle$. The ***environment*** μ maps method names to commands. The empty environment is written _. In a configuration, the command C may include

[26] One could distinguish between these two kinds of faults using different tokens, as done in RLII. Here, we would need a third kind, for alignment fault. But the correctness judgments disallow all three kinds, so for simplicity, we conflate them.

$$
\begin{array}{lcl}
\sigma(E_1 \otimes E_2) & \hat{=} & \sigma(E_1) \otimes \sigma(E_2) \quad \text{where } \otimes \text{ is in } \{=, \leq, +, \dots\} \\
\sigma(\{E\}) & \hat{=} & \{\sigma(E)\} \\
\sigma(\varnothing) & \hat{=} & \varnothing \\
\sigma(G_1 \otimes G_2) & \hat{=} & \sigma(G_1) \otimes \sigma(G_2) \quad \text{where } \otimes \text{ is in } \{\cup, \cap, \setminus\} \\
\sigma(G/K) & \hat{=} & \{o \mid o \in \sigma(G) \wedge o \neq \mathit{null} \wedge \mathit{Type}(o, \sigma) = K\} \\
\sigma(G\text{`}f) & \hat{=} & \varnothing \quad \text{if } f\text{:int (or any primitive type)} \\
 & \hat{=} & \{\sigma(o.f) \mid o \in \sigma(G) \wedge o \neq \mathit{null} \wedge \mathit{Type}(o, \sigma) = \mathit{DeclClass}(f)\} \quad \text{if } f\text{:}K \text{ for some } K \\
 & \hat{=} & \bigcup\{\sigma(o.f) \mid o \in \sigma(G) \wedge o \neq \mathit{null} \wedge \mathit{Type}(o, \sigma) = \mathit{DeclClass}(f)\} \quad \text{if } f\text{:rgn}
\end{array}
$$

Fig. 21. Semantics $\sigma(F)$ of selected program and region expressions (r-values), for state σ.

the pseudo-commands: ecall(m) ends the code of a call to method m, evar(x) ends the scope of a local variable, and elet($\overline{m}$) ends the scope of some methods $\overline{m}$ (arising from simultaneous binding let $\overline{m}$=$\overline{B}$ in C). The pseudo-commands do not occur in source programs. The code of a configuration thus takes a form that represents the execution stack for environment calls:

$$C_n; \mathsf{ecall}(m_n); \dots; C_1; \mathsf{ecall}(m_1); C_0 \quad \text{where } n \geq 0 \text{ and each } C_i \text{ is ecall-free.}$$

So the leftmost command C_n is on top of the stack and m_n is the leftmost environment call. We write $\mathit{Active}(C)$ for the ***active command*** (which one might call the redex), i.e., the unique sub-command that gets rewritten by the applicable transition rule.[27] For example, $\mathit{Active}(x := 0; y := 1)$ is $x := 0$.

To formalize the semantics of encapsulation, we need to refer to the module of the active command: it must stay outside the boundary of every module except its own. So, we define the ***top module*** $\mathit{topm}(C, M)$ to be N where $N = \mathit{mdl}(m_n)$ and m_n is the leftmost environment call (see above), or M if C has no ecall (i.e., $n = 0$). This is used in Definition 5.10, where the argument M is from the judgment under consideration. In Definition 5.10, we also write $N \in (\Phi, \mu)$, for hypothesis context Φ and method environment μ, to mean there is $m \in \mathit{dom}(\Phi) \cup \mathit{dom}(\mu)$ with $\mathit{mdl}(m) = N$.

For an empty method context, the transition relation is standard (Figure 34). For non-empty contexts the transition relation depends on a pre-model, which is defined in terms of the semantics of specs, to which we proceed.

5.2 Semantics of State Predicate Formulas and Effects

Satisfaction of formula P in state σ is written $\sigma \models P$. The semantics of formulas is standard and two-valued. The points-to relation $x.f = E$ is defined by $\sigma \models x.f = E$ iff $\sigma(x) \neq \mathit{null}$ and $\sigma(\sigma(x).f) = \sigma(E)$. The type predicate is defined by $\sigma \models \mathsf{type}(G, \overline{K})$ iff $\mathit{Type}(o, \sigma) \in \overline{K}$ for all $o \in \sigma(G)$. Quantifiers for reference types range over allocated (thus non-null) references: $\sigma \models \forall x : K.\, P$ iff $[\sigma{+}x{:}\, o] \models P$ for all $o \in \sigma(\mathsf{alloc})$ with $\mathit{Type}(o, \sigma) = K$.

Lemma 5.1 (Unique Snapshots). *If $P, \Gamma, \hat{\Gamma}$ satisfy the condition for precondition P in Definition 3.2, then for all Γ-states σ there is at most one $(\Gamma, \hat{\Gamma})$-state $\hat{\sigma}$ that extends σ such that $\hat{\sigma} \models P$.*

In contexts where we consider a precondition P and suitable state σ, we adopt the ***hat convention*** of writing $\hat{\sigma}$ for the extension of σ uniquely determined by σ and P as in Lemma 5.1.

For an effect ε in a given state σ, its read effects designate a set $\mathit{rlocs}(\sigma, \varepsilon)$ of locations. Specifically, it is the set of l-values of the left-expressions in its read effects:

$$
\begin{array}{lcl}
\mathit{rlocs}(\sigma, \varepsilon) & \hat{=} & \{x \mid \varepsilon \text{ contains } \mathsf{rd}\, x\} \cup \\
 & & \{o.f \mid \varepsilon \text{ contains some } \mathsf{rd}\, G\text{`}f \text{ with } o \in \sigma(G),\, o \neq \mathit{null},\, f \in \mathit{Fields}(\mathit{Type}(o, \sigma))\}.
\end{array}
$$

[27] We identify sequentially composed commands up to associativity (Figure 6) so $\mathit{Active}(C)$ can be defined as the leftmost non-sequence command of a sequence.

Define $wlocs(\sigma, \varepsilon)$ the same way but for the l-values in write effects. Note that for an effect of the form rd $G\text{`}f$ the definition of *rlocs* uses the r-value $\sigma(G)$ (Figure 21) where G may itself involve images. These functions are used in the key lemma about effect subtraction (see Equation (7)).

Lemma 5.2 (Subtraction). *$rlocs(\sigma, \varepsilon \backslash \eta) = rlocs(\sigma, \varepsilon) \backslash rlocs(\sigma, \eta)$ and the same for wlocs.*

For use in the semantics of write effects, define the locations of σ that have been changed in τ as

$$wrttn(\sigma, \tau) \triangleq \{x \mid x \in Vars(\sigma) \cap Vars(\tau) \wedge \sigma(x) \neq \tau(x)\} \cup \{o.f \mid o.f \in locations(\sigma) \wedge \sigma(o.f) \neq \tau(o.f)\}$$

This captures the variables still in scope that have been changed, together with changed heap locations.[28] Say τ ***can succeed*** σ, written $\sigma \hookrightarrow \tau$, provided $\sigma(\text{alloc}) \subseteq \tau(\text{alloc})$ and $Type(o, \sigma) = Type(o, \tau)$ for all $o \in \sigma(\text{alloc})$. Say ε ***allows change*** from σ *to* τ, in symbols $\sigma \rightarrow \tau \models \varepsilon$, iff $\sigma \hookrightarrow \tau$ and $wrttn(\sigma, \tau) \subseteq wlocs(\sigma, \varepsilon)$. The locations of τ not present in σ are designated by $freshL(\sigma, \tau)$. Define $freshRefs(\sigma, \tau) \triangleq \tau(\text{alloc}) \backslash \sigma(\text{alloc})$ and

$$freshL(\sigma, \tau) \triangleq \{p.f \mid p \in freshRefs(\sigma, \tau) \wedge f \in Fields(Type(p, \tau))\} \cup Vars(\tau) \backslash Vars(\sigma).$$

Read effects and refperms. Read effects constrain the locations on which the outcome of a computation can depend. Dependency is expressed by considering two initial states that agree on the values in the locations deemed readable, though the states may differ on the values in 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. Such variation must also be taken into account in relation formulas, as in Example 4.3. For use with both read effects and relation formulas, agreements are formalized using refperms, as mentioned in Section 2.3.

Let π range over ***partial bijections*** on $Ref \backslash \{null\}$, i.e., injective partial functions. Write $\pi(p) = p'$ to express that π is defined on p and has value p'. A ***refperm from*** σ ***to*** σ' is a partial bijection π such that $dom(\pi) \subseteq \sigma(\text{alloc})$, $rng(\pi) \subseteq \sigma'(\text{alloc})$, and $\pi(p) = p'$ implies $Type(p, \sigma) = Type(p', \sigma')$. Define $p \overset{\pi}{\sim} p'$ to mean $\pi(p) = p'$ or $p = null = p'$. Extend $\overset{\pi}{\sim}$ to a relation on integers by $i \overset{\pi}{\sim} j$ iff $i = j$. For reference sets X, Y, define $X \overset{\pi}{\sim} Y$ to mean that $\pi \cup \{(null, null)\}$ restricts to a total bijection between X and Y. The image of π on location set W is written $\pi(W)$ and defined for variables and heap locations by two conditions: $x \in \pi(W)$ iff $x \in W$, and $o.f \in \pi(W)$ iff $(\pi^{-1}(o)).f \in W$. In other words: variables map to themselves, and a heap location $p.f$ is transformed by applying π to the reference p.

Next, we define notations for agreement between states. Agreement is formalized in terms of a condition that applies to two states together with a refperm and a subset W of the locations of σ. The location agreement $Lagree(\sigma, \sigma', \pi, W)$ holds just if W is a set of locations of σ and for each of these locations, the contents in σ is the same as the contents of the location that corresponds according to π. Of course, "same as" is modulo π, for reference values.

Definition 5.3 (Agreement on a Location Set, Lagree). For W a set of locations in σ, and π a refperm from σ to σ', define

$$Lagree(\sigma, \sigma', \pi, W) \text{ iff } \forall x \in W.\ \sigma(x) \overset{\pi}{\sim} \sigma'(x) \ \wedge \forall (o.f) \in W.\ o \in dom(\pi) \wedge \sigma(o.f) \overset{\pi}{\sim} \sigma'(\pi(o).f).$$

This is defined for any $W \subseteq locations(\sigma)$. Agreement is monotonic in the refperm, in the sense that

$$Lagree(\sigma, \sigma', \pi, W) \text{ and } \pi \subseteq \rho \text{ implies } Lagree(\sigma, \sigma', \rho, W). \tag{21}$$

[28] The definitions are formulated to be applicable to intermediate states in the scope of local blocks, which introduce variables not present in the typing context of the initial command.

Definition 5.4 (Agreement on Read Effects, Agree). Let ε be an effect that is wf in Γ. Consider Γ-states σ, σ'. Let π be a refperm. Say that σ and σ' ***agree on*** ε ***modulo*** π, written $Agree(\sigma, \sigma', \pi, \varepsilon)$, iff $Lagree(\sigma, \sigma', \pi, rlocs(\sigma, \varepsilon))$. Let $Agree(\sigma, \sigma', \varepsilon) \triangleq Agree(\sigma, \sigma', \pi, \varepsilon)$ where π is the identity on $\sigma(\text{alloc}) \cap \sigma'(\text{alloc})$.

Often we use $Agree(\sigma, \tau, \varepsilon)$ where $\sigma \hookrightarrow \tau$, in which case $\sigma(\text{alloc}) \cap \tau(\text{alloc}) = \sigma(\text{alloc})$.
Agreement on location sets enjoys a kind of symmetry:

$$Lagree(\sigma, \sigma', \pi, W) \text{ implies } Lagree(\sigma', \sigma, \pi^{-1}, \pi(W)) \text{ for all } \sigma, \sigma', \pi, W. \tag{22}$$

By contrast, Definition 5.4 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. The asymmetry makes working with agreement somewhat delicate. For example, agreement on $\mathrm{rd}\, G\text{`}f$ (modulo π) implies that $\sigma(G) \subseteq dom(\pi)$ (by Definition 5.3), but it does not imply $\sigma(G) \overset{\pi}{\sim} \sigma'(G)$. At a higher level there will be symmetry, for reasons explained in due course.

5.3 Pre-models and Program Semantics

The transition relation depends on a pre-model φ, defined below, and is written $\overset{\varphi}{\longmapsto}$. The pre-model provides semantics for context calls and represents denotations of method bodies. Transitions act on configurations where the environment μ has procedures distinct[29] from those of φ.

Definition 5.5 (State Isomorphism $\overset{\pi}{\approx}$, Outcome Equivalence $\cong_\pi$). For Γ-states σ, σ', define $\sigma \overset{\pi}{\approx} \sigma'$ (read: ***isomorphic mod*** π) to mean that refperm π is a total bijection from $\sigma(\text{alloc})$ to $\sigma'(\text{alloc})$ and the states agree mod π on all variables and all fields of all objects. That is, $Lagree(\sigma, \sigma', \pi, locations(\sigma))$.[30] For $S, S' \in \mathbb{P}(\llbracket \Gamma \rrbracket \cup \{\lightning\})$, define $S \cong_\pi S'$ (read ***equivalent mod*** 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) $S = \emptyset$ iff $S' = \emptyset$.

Note that item (ii) involves extensions of π, whereas the relations $\overset{\pi}{\sim}$ and $\overset{\pi}{\approx}$ involve only π itself.

Lemma 5.6. *Suppose $\sigma \overset{\pi}{\approx} \sigma'$. Then $\sigma(F) \overset{\pi}{\sim} \sigma'(F)$, and $\sigma \models P$ iff $\sigma' \models P$.*

Definition 5.7. A ***pre-model*** for Γ is a mapping from some set of method names, such that for $m \in dom(\varphi)$, $\varphi(m)$ is a function of type $\llbracket \Gamma \rrbracket \to \mathbb{P}(\llbracket \Gamma \rrbracket \cup \{\lightning\})$ such that $\sigma \hookrightarrow \tau$ for all σ, τ with $\tau \in \varphi(m)(\sigma)$, and

(***fault determinacy***) $\lightning \in \varphi(m)(\sigma)$ implies $\varphi(m)(\sigma) = \{\lightning\}$,
(***state determinacy***) $\sigma \overset{\pi}{\approx} \sigma'$ implies $\varphi(m)(\sigma) \cong_\pi \varphi(m)(\sigma')$.

For Φ wf in Γ, a pre-model of Φ is a pre-model for Γ and $dom(\Phi)$.

We say pre-models are ***quasi-deterministic***, because from a given initial state, these three outcomes are mutually exclusive: fault, non-empty set of states, empty set. Moreover, instantiating $\sigma' := \sigma$ and setting π to the identity on $\sigma(\text{alloc})$ in the condition (state determinacy) yields that all results from a given initial state are isomorphic.[31]

[29] This representation takes advantage of the hygiene condition that variable and method names are never re-used in nested declarations.

[30] Which is equivalent to $Lagree(\sigma', \sigma, \pi^{-1}, locations(\sigma'))$, in this context where $\sigma(\text{alloc}) \overset{\pi}{\sim} \sigma'(\text{alloc})$.

[31] In light of these definitions and the results to follow, we could as well replace the codomain of a pre-model, i.e., $\mathbb{P}(\llbracket \Gamma \rrbracket \cup \{\lightning\})$, by the disjoint sum of $\mathbb{P}(\llbracket \Gamma \rrbracket)$ and $\{\lightning\}$. The chosen formulation helps streamline a few things later.

$$\textsc{uCall}\ \frac{\tau \in \varphi(m)(\sigma)}{\langle m(),\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \langle \mathsf{skip},\ \tau,\ \mu\rangle} \qquad \textsc{uCallX}\ \frac{\lightning \in \varphi(m)(\sigma)}{\langle m(),\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \lightning} \qquad \textsc{uCall0}\ \frac{\varphi(m)(\sigma) = \varnothing}{\langle m(),\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \langle m(),\ \sigma,\ \mu\rangle}$$

$$\textsc{uCallE}\ \frac{\mu(m) = C}{\langle m(),\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \langle C; \mathsf{ecall}(m),\ \sigma,\ \mu\rangle} \qquad \textsc{uECall}\ \langle \mathsf{ecall}(m),\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \langle \mathsf{skip},\ \sigma,\ \mu\rangle$$

$$\textsc{uLet}\ \langle \mathsf{let}\ \overline{m} = \overline{B}\ \mathsf{in}\ C,\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \langle C; \mathsf{elet}(\overline{m}),\ \sigma,\ [\mu{+}\overline{m}{:}\,\overline{B}]\rangle \qquad \textsc{uELet}\ \langle \mathsf{elet}(\overline{m}),\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \langle \mathsf{skip},\ \sigma,\ \mu \upharpoonright \overline{m}\rangle$$

Fig. 22. Selected transition rules, for pre-model φ. The others are in Appendix Figure 34.

The transition relation is defined in Figure 22. A ***trace*** via pre-model φ is a non-empty finite sequence of configurations that are consecutive for the transition relation $\overset{\varphi}{\longmapsto}$. For example, this sequence is a trace (for any φ):

$$\langle x := 1; y := 2,\ [x{:}0, y{:}0],\ \_\rangle\langle y := 2,\ [x{:}1, y{:}0],\ \_\rangle\langle \mathsf{skip},\ [x{:}1, y{:}2],\ \_\rangle.$$

Recall that we identify $(\mathsf{skip}; C)$ with C (Figure 6). By definition, a trace does not contain $\lightning$.

5.4 Context Models and Program Correctness

For syntactic substitution, we use the notation P^x_F. Substitution notations are mainly used with spec-only variables. In addition, for clarity, we also use substitution notation for values, even references—although the syntax does not include reference literals.

Definition 5.8 (Substitution Notation). If $\Gamma, x{:}T \vdash P$ and $\sigma \in [\![\,\Gamma\,]\!]$ and v is a value in $[\![\,T\,]\!]\sigma$, then we write $\sigma \models^{\Gamma} P^x_v$ to abbreviate $[\sigma{+}x{:}\,v] \models^{\Gamma, x:T} P$.

A context model, or Φ-model when we refer to a specific context Φ, is a pre-model that satisfies its specs.

Definition 5.9 (Context Model). Let Φ be wf in Γ and let φ be a pre-model. Say φ is a Φ-***model*** iff $dom(\varphi) = dom(\Phi)$ and for each m in $dom(\Phi)$ with $\Phi(m) = R \rightsquigarrow S\,[\eta]$ and for any σ and σ' in $[\![\,\Gamma\,]\!]$,

(a) $\lightning \in \varphi(m)(\sigma)$ iff there are no values $\overline{v}$ with $\sigma \models R^{\overline{s}}_{\overline{v}}$ where $\overline{s}$ are the spec-only variables.
(b) For all $\tau \in \varphi(m)(\sigma)$, and all $\overline{v}$, if $\sigma \models R^{\overline{s}}_{\overline{v}}$ then $\tau \models S^{\overline{s}}_{\overline{v}}$ and $\sigma{\rightarrow}\tau \models \eta$.
(c) For all $\tau \in \varphi(m)(\sigma)$ and all N with $mdl(m) \preceq N$, $rlocs(\sigma, bnd(N)) \subseteq rlocs(\tau, bnd(N))$.
(d) For all π, if $Lagree(\sigma, \sigma', \pi, rlocs(\sigma, \eta)\backslash\{\mathsf{alloc}\})$, then
  (i) $\varphi(m)(\sigma) = \varnothing$ iff $\varphi(m)(\sigma') = \varnothing$, and
  (ii) if $\tau \in \varphi(m)(\sigma)$ and $\tau' \in \varphi(m)(\sigma')$, then there is $\rho \supseteq \pi$ with $\rho(freshL(\sigma, \tau)) \subseteq freshL(\sigma', \tau')$ and $Lagree(\tau, \tau', \rho, (freshL(\sigma, \tau) \cup wrttn(\sigma, \tau))\backslash\{\mathsf{alloc}\})$.

Condition (a) says $\varphi(m)$ faults just on states outside the precondition of m, (b) says the postcondition holds and write effect is respected, (c) is a technical condition we call boundary monotonicity, and (d) is the dependency condition of the read effect.

The snapshot values $\overline{v}$ in (a) and (b) are uniquely determined by σ (Lemma 5.1). So (a) can be rephrased: $\lightning \in \varphi(m)(\sigma)$ iff $\sigma \not\models R^{\overline{s}}_{\overline{v}}$ where $\overline{v}$ are the values uniquely determined by R in σ. Similarly for (b), which treats spec-only variables as being quantified over the pre- and postcondition.

Finally, we can give the semantics of correctness judgments, which embodies encapsulation for dynamic boundaries. In the definition to follow, we write $\delta^{\oplus}$ to abbreviate $\delta, \mathsf{rd\ alloc}$. Apropos Definition 5.9(d), note that $\{\mathsf{alloc}\} = rlocs(\sigma, \mathsf{rd\ alloc}) = rlocs(\sigma, \bullet^{\oplus})$.

The conditions for a valid correctness judgment include that there are no faulting executions, terminated executions satisfy the postcondition and write effect, and boundary monotonicity. These conditions are like (a)–(c) above for context model. The absence of fault means more than no null dereference; it means there are no method calls outside the method's precondition—because otherwise the call would fault, by condition (a) for context models. An additional condition for correctness is that the read effects of the judgment should subsume the read effects in the specs of methods in context calls; this is called r-safety. Finally, the Encap condition says that each step reads and writes outside the boundaries of any module the step is not within. The Encap condition is formulated using the read effects of the judgments and implies the expected end-to-end read effect as will be explained later. Reading is meant in the extensional sense of a two-run dependency property, similar to condition (d) for context model.

The Encap condition applies to every reachable step, and refers to the initial state, so we use the following schema to designate identifiers for the elements of a step reached from command C and state σ:

$$\langle C,\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto}{}^{*}\ \langle B,\ \tau,\ \mu\rangle \overset{\varphi}{\longmapsto}\ \langle D,\ \upsilon,\ \nu\rangle.$$

The step is taken by the active command of B, from state τ to state υ. For such a step, we need to refer to the locations encapsulated by all modules except the current module, M, of the correctness judgment. To this end, the ***collective boundary*** is an effect δ defined by cases:

$$\begin{array}{rcll} \delta & \hat{=} & (+N \in (\Phi,\mu), N \neq \mathit{topm}(B,M).\ \mathit{bnd}(N)), & \text{if } \mathit{Active}(B) \text{ is not a context call,} \\ & \hat{=} & (+N \in (\Phi,\mu), \mathit{mdl}(m) \not\preceq N.\ \mathit{bnd}(N)), & \text{if } \mathit{Active}(B) \text{ is a context call of } m. \end{array} \tag{23}$$

Definition 5.10 (Valid Judgment). A wf judgment $\Phi \vdash^{\Gamma}_{M} C :\ P \rightsquigarrow Q\ [\varepsilon]$ is ***valid*** iff the following hold for all Φ-models φ, all values $\overline{v}$ for the spec-only variables $\overline{s}$ in P, and all states σ such that $\sigma \models^{\Gamma} P^{\overline{s}}_{\overline{v}}$.

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

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

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

(***R-safe***) Every reachable configuration $\langle C,\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto}{}^{*}\ \langle B,\ \tau,\ \mu\rangle$ satisfies the ***r-safe condition for*** $(\Phi, \varepsilon, \sigma)$: If $\mathit{Active}(B)$ is a context call to m with $\Phi(m) \equiv m : R \rightsquigarrow S\ [\eta]$, then $\mathit{rlocs}(\tau,\eta) \subseteq \mathit{freshL}(\sigma,\tau) \cup \mathit{rlocs}(\sigma,\varepsilon)$.

(***Encap***) Every reachable step $\langle C,\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto}{}^{*}\ \langle B,\ \tau,\ \mu\rangle \overset{\varphi}{\longmapsto}\ \langle D,\ \upsilon,\ \nu\rangle$ ***respects*** $(\Phi, M, \varphi, \varepsilon, \sigma)$, i.e.,

- For every N with $N \in (\Phi,\mu)$ and $N \neq \mathit{topm}(B,M)$, the step ***w-respects*** N, which means: either $\mathit{Active}(B)$ is a call to some m with $\mathit{mdl}(m) \preceq N$ or $\mathit{Agree}(\tau,\upsilon,\mathit{bnd}(N))$.
- For δ the collective boundary given by Equation (23) for B,τ,μ, the step ***r-respects*** δ ***for*** $(\varphi,\varepsilon,\sigma)$, which means: for any[32] π,τ',υ',D'

$$\begin{array}{l} \text{if } \langle B,\ \tau',\ \mu\rangle \overset{\varphi}{\longmapsto} \langle D',\ \upsilon',\ \nu\rangle \text{ and } \mathit{Agree}(\tau',\upsilon',\delta) \text{ and} \\ \mathit{Lagree}(\tau,\tau',\pi,(\mathit{freshL}(\sigma,\tau) \cup \mathit{rlocs}(\sigma,\varepsilon))\backslash \mathit{rlocs}(\tau,\delta^{\oplus})) \end{array} \tag{24}$$

  then $D' \equiv D$ and there is ρ with $\rho \supseteq \pi$ such that

$$\begin{array}{l} \mathit{Lagree}(\upsilon,\upsilon',\rho,(\mathit{freshL}(\tau,\upsilon) \cup \mathit{wrttn}(\tau,\upsilon))\backslash \mathit{rlocs}(\upsilon,\delta^{\oplus})) \text{ and} \\ \rho(\mathit{freshL}(\tau,\upsilon)\backslash \mathit{rlocs}(\upsilon,\delta)) \subseteq \mathit{freshL}(\tau',\upsilon')\backslash \mathit{rlocs}(\upsilon',\delta) \end{array} \tag{25}$$

- For every N with $N \in \Phi$ or $N = M$, the step satisfies ***boundary monotonicity***:

  $\mathit{rlocs}(\tau,\mathit{bnd}(N)) \subseteq \mathit{rlocs}(\upsilon,\mathit{bnd}(N))$.

[32] To be precise: such that τ' has the same variables as τ—there may be local variables in addition to those declared by Γ.

In addition to the terms introduced above to refer to parts of the definition, we also use the following derived notions: A trace from $\langle C,\ \sigma,\ \_\rangle$ ***respects*** $(\Phi, M, \varphi, \varepsilon, \sigma)$ just if each step of the trace does, and it is ***r-safe for*** $(\Phi, \varepsilon, \sigma)$ just if each configuration is. A step is called ***r-safe*** if its starting configuration is r-safe.

While w-respect can be defined one module at a time, this is not the case for r-respect, because dependency properties do not compose in a simple way.[33] The absence of dependency needs to be expressed in terms of the collective boundary δ with which a given step must not interfere. As with w-respect, this depends on whether the step is a context call. If not, then the current module's boundary is exempt (see condition $N \neq topm(B, M)$ in Equation (23)). If so, then the step is exempt from the boundary of the callee's module together with modules into which its implemenation may call (second condition in Equation (23)). Dependency is expressed as usual by an implication from initial agreement Equation (24) on reads to final agreement Equation (25) on writes—subtracting the encapsulated locations. The read effects in ε are interpreted in the pre-state σ, as are the write effects (which cover the written locations according to the condition labelled Write). The collective boundary δ is interpreted at intermediate states.

In case the module boundaries are all empty, in Definition 5.10, two parts of the Encap condition become vacuous, namely, w-respect and boundary monotonicity. And r-respect reduces to the property that the dependency of each step is within the readable locations of the given frame condition. This implies an end-to-end read effect condition given in the following lemma.[34] The lemma is used to prove soundness of the linking rule; in that proof we derive a pre-model from the denotation of the method body, and the lemma is used to show it is a context model.

Lemma 5.11 (Read Effect). *Suppose* $\Phi \models^{\Gamma}_{M} C : P \rightsquigarrow Q[\varepsilon]$ *and* φ *is a* Φ*-model. Suppose* $\sigma \models P$ *and* $\sigma' \models P$. *Suppose* $Lagree(\sigma, \sigma', \pi, rlocs(\sigma, \varepsilon)\backslash\{\mathsf{alloc}\})$. *Then* $\langle C,\ \sigma,\ \_\rangle$ *diverges iff* $\langle C,\ \sigma',\ \_\rangle$ *diverges. And for any* τ, τ', *if* $\langle C,\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \tau,\ \_\rangle$ *and* $\langle C,\ \sigma',\ \_\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \tau',\ \_\rangle$ *then*

$$\exists \rho \supseteq \pi.\quad Lagree(\tau, \tau', \rho, (freshL(\sigma, \tau) \cup wrttn(\sigma, \tau))\backslash\{\mathsf{alloc}\}) \quad \textit{and} \\ \rho(freshL(\sigma, \tau)) \subseteq freshL(\sigma', \tau').$$

6 UNARY LOGIC

Correctness judgments of the unary logic play a crucial role in the relational logic. They are premises in relational rules such as local equivalence. Framing and encapsulation are handled at the unary level, separate from the concerns of alignment and relation formulas.

The unary proof rules use two subsidiary judgments, for subeffects and framing of formulas. These can be presented by inference rules (as shown in RLI). In this article, we present them semantically, in Section 6.1, as the semantics is amenable to direct checking by SMT solver. Informal descriptions are given, but for the detailed definitions in Section 6.1 the reader needs to be familiar with the definitions in Sections 5.1 and 5.2. Aside from that, Section 6 can be read without being familiar with Section 5.

6.1 Framing and Subeffects

The ***subeffect judgment***, written $P \models \varepsilon \leq \eta$, says that in states satisfying P, the readable or writable locations designated by ε are contained in those designated by η. It is defined as follows:

$$P \models \varepsilon \leq \eta \text{ iff } rlocs(\sigma, \varepsilon) \subseteq rlocs(\sigma, \eta) \text{ and } wlocs(\sigma, \varepsilon) \subseteq wlocs(\sigma, \eta) \text{ for all } \sigma \text{ with } \sigma \models P. \quad (26)$$

[33] For readers familiar with RLII, the w-respect condition is the same except that, here, to support r-respect we add w-respect of modules in the environment (in addition to those in context).

[34] The condition is much like the semantics of effects in RLIII, with a small difference concerning the treatment of variable alloc. (See Definition 5.2 in RLIII.)

The ***framing judgment*** for formulas, written $P \models \eta \text{ frm } Q$, can loosely be understood to say the read effects in η cover the footprint of Q. It is used in the frame rule and also second-order frame rule, where we need framing of the module invariant by the dynamic boundary. To be precise, the judgment says of states σ and τ that if σ satisfies $P \wedge Q$ and τ agrees with σ on the contents of locations designated by the read effects of η, then τ satisfies Q. Here η is interpreted in state σ, which only matters if its effect expressions mention mutable variables. The judgment is defined as follows:

$$P \models \eta \text{ frm } Q \text{ iff for all } \sigma, \tau, \text{ if } Agree(\sigma, \tau, \eta) \text{ and } \sigma \models P \wedge Q \text{ then } \tau \models Q. \tag{27}$$

For example, we have $x \in r \models \mathsf{rd}\, x, \mathsf{rd}\, r\text{`}f \text{ frm } x.f = 0$. The *ftpt* function, defined in Figure 10, provides framing for atomic formulas. The basic lemmas about *ftpt* are that $\models ftpt(P) \text{ frm } P$, for atomic P, and

$$Agree(\sigma, \sigma', \pi, ftpt(F)) \text{ implies } \sigma(F) \overset{\pi}{\sim} \sigma'(F). \tag{28}$$

The framing judgment is used, in the Frame rule, in combination with a separator formula (Figure 11). A key property of separators is that a formula obtained as $\eta \cdot/. \varepsilon$ holds in σ iff $rlocs(\sigma, \eta) \cap wlocs(\sigma, \varepsilon) = \emptyset$. From this it follows that

$$\sigma \rightarrow \tau \models \varepsilon \text{ and } \sigma \models \eta \cdot/. \varepsilon \text{ implies } Agree(\sigma, \tau, \eta). \tag{29}$$

Separator formulas are also used in the notion of immunity, which amounts to framing for frame conditions. Immunity is only needed for the sequence and loop rules, which we relegate to the Appendix as there is no interesting change from RLI. Framing and immunity are about preserving the value of an expression or formula from one control point to a later one. For preservation of agreements, framed reads (Definition 3.1) are crucial; e.g., in proving the lockstep alignment Lemma 8.9.

6.2 Proof Rules

Selected proof rules are in Figure 23. They are to be instantiated only with wf premises and conclusions. In the rest of the section, we comment briefly about some rules and derive the modular linking rule. Then Section 6.3 discusses how the rules work together to enforce encapsulation.

The proof rules for assignment, like FieldUpd and Alloc, are "small axioms" [76] that have empty context, are in the default module, and have precise frame conditions. The Conseq rule can be used to subsume a frame condition like $\mathsf{wr}\, \{x\}\text{`}f$ by a more general one like $\mathsf{wr}\, r\text{`}f$, given precondition $x \in r$ and using subeffect judgment $x \in r \models \mathsf{wr}\, \{x\}\text{`}f \leq \mathsf{wr}\, r\text{`}f$. Rule Alloc can be used with the Frame rule to express freshness in several ways.[35] These and the method call rule have the minimum needed hypothesis context. Extending the context is done by rules discussed in Section 6.3.

The gist of the second-order frame rule, SOF, is to conjoin a formula not only to the spec in the conclusion, like rule Frame, but also conjoin it to the specs in the hypothesis context. The rule distils a property of program semantics; its practical role is to derive the modular linking rule.

In rule SOF, the conditions $N \in \Theta$ and $N \neq M$ ensure that the command C respects the encapsulation of $bnd(N)$, in accord with the semantic condition Encap of Definition 5.10. Together with the framing judgment $\models bnd(N) \text{ frm } I$, this ensures that C does not falsify I. The condition C ***binds no*** N***-method*** means C contains no let-binding of a method m with $mdl(m) = N$. This and the condition $\forall m \in \Phi.\ mdl(m) \not\leq N$ ensure that all of N's method specs are in Θ and have the invariant added simultaneously. Such conditions are the price we pay for not cluttering the logic with explicit syntax and judgments for a module calculus. Rule Link has analogous conditions.

[35] Shown in detail in RLIII (Section 7.1).

$$\textsc{Conseq}\ \frac{\Phi \vdash_M C : P \rightsquigarrow Q\ [\varepsilon] \qquad P_1 \Rightarrow P \qquad Q \Rightarrow Q_1 \qquad P_1 \models \varepsilon \leq \varepsilon_1}{\Phi \vdash_M C : P_1 \rightsquigarrow Q_1\ [\varepsilon_1]}$$

$$\textsc{Frame}\ \frac{\Phi \vdash_M C : P \rightsquigarrow Q\ [\varepsilon] \qquad P \models \eta\ \mathsf{frm}\ R \qquad P \wedge R \Rightarrow \eta \cdot/. \varepsilon}{\Phi \vdash_M C : P \wedge R \rightsquigarrow Q \wedge R\ [\varepsilon]}$$

$$\textsc{SOF}\ \frac{\Phi, \Theta \vdash_M C : P \rightsquigarrow Q\ [\varepsilon] \qquad \models bnd(N)\ \mathsf{frm}\ I \qquad N \in \Theta \qquad N \neq M \qquad \forall m \in \Phi.\ mdl(m) \not\leq N \qquad C \text{ binds no } N\text{-method}}{\Phi, (\Theta \between I) \vdash_M C : P \wedge I \rightsquigarrow Q \wedge I\ [\varepsilon]}$$

$$\textsc{CtxIntroIn1}\ \frac{\Phi \vdash_M C : P \rightsquigarrow Q\ [\varepsilon] \qquad mdl(m) \in \Phi}{\Phi, m{:}R \rightsquigarrow S\ [\eta] \vdash_M C : P \rightsquigarrow Q\ [\varepsilon]}$$

$$\textsc{CtxIntro}\ \frac{\Phi \vdash_M A : P \rightsquigarrow Q\ [\varepsilon] \qquad P \Rightarrow bnd(mdl(m)) \cdot/. \varepsilon \qquad P \Rightarrow bnd(mdl(m)) \cdot/. r2w(\varepsilon)}{\Phi,\ m : R \rightsquigarrow S\ [\eta] \vdash_M A : P \rightsquigarrow Q\ [\varepsilon]}$$

$$\textsc{Call}\quad m{:}P \rightsquigarrow Q\ [\varepsilon] \vdash_\bullet m() : P \rightsquigarrow Q\ [\varepsilon] \qquad \textsc{FieldUpd} \vdash_\bullet x.f := y : x \neq \mathsf{null} \rightsquigarrow x.f = y\ [\mathsf{wr}\, x.f, \mathsf{rd}\, x, \mathsf{rd}\, y]$$

$$\textsc{Link}\ \frac{\Phi, \Theta \vdash_\bullet C : P \rightsquigarrow Q\ [\varepsilon] \qquad dom(\Theta) = \overline{m} \qquad \Phi, \Theta \vdash_{mdl(m_i)} B_i : \Theta(m_i) \qquad \forall N \in \Phi, L \in \Theta.\ N \not\leq L \qquad \forall N, L.\ N \in \Theta \wedge N \prec L \Rightarrow L \in (\Phi, \Theta)}{\Phi \vdash_\bullet \mathsf{let}\ \overline{m} = \overline{B}\ \mathsf{in}\ C : P \rightsquigarrow Q\ [\varepsilon]}$$

$$\textsc{Alloc}\ \frac{Fields(K) = \overline{f} : \overline{T} \qquad \text{spec-only}(r)}{\vdash_\bullet x := \mathsf{new}\ K :\ r = \mathsf{alloc} \rightsquigarrow x \notin r \wedge \mathsf{alloc} = r \cup \{x\} \wedge x.\overline{f} = default(\overline{T})\ [\mathsf{wr}\, x, \mathsf{rw}\, \mathsf{alloc}]}$$

$$\textsc{If}\ \frac{\Phi \vdash_M C_1 : P \wedge E \rightsquigarrow Q\ [\varepsilon] \qquad \Phi \vdash_M C_2 : P \wedge \neg E \rightsquigarrow Q\ [\varepsilon] \qquad (+N \in \Phi, N \neq M.\ bnd(N)) \cdot/. r2w(ftpt(E))}{\Phi \vdash_M \mathsf{if}\ E\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 : P \rightsquigarrow P'\ [\varepsilon, ftpt(E)]}$$

Fig. 23. Selected unary proof rules. For others see Appendix Figures 35 and 36.

In rule Link, let $\overline{m} = \overline{B}$ in C means the simultaneous linking of m_i with B_i for i in some range. This version of Link supports simultaneous linking of multiple methods that may be defined in different modules. Note that Θ is in the hypotheses for B_i, because some methods in Θ may call others in Θ, and for recursion. Condition $\forall N \in \Phi, L \in \Theta.\ N \not\leq L$ precludes dependency of the ambient modules on the ones being linked. Condition $\forall N, L.\ N \in \Theta \wedge N \prec L \Rightarrow L \in (\Phi, \Theta)$ expresses import closure, which is needed to ensure that all relevant boundaries are considered in the Encap condition of the premises.

Recall the modular linking rule (2) sketched in Section 2.1. It can now be made precise as follows:

$$\textsc{MLink}\ \frac{\Phi \vdash_\bullet C : P \rightsquigarrow Q\ [\varepsilon] \qquad \Phi \between I \vdash_M B : \Phi(m) \between I \qquad mdl(m) = M \qquad \models bnd(M)\ \mathsf{frm}\ I \qquad P \Rightarrow I}{\vdash_\bullet \mathsf{let}\ m = B\ \mathsf{in}\ C : P \rightsquigarrow Q\ [\varepsilon]}\ .$$

In Section 2.1, we mention requirements for soundness of Equation (2), in vague terms that can now be made precise. Requirement (E1) is to delimit some internal locations, which is expressed as a dynamic boundary $bnd(M)$. Requirement (E2) is that the module invariant I depends only on encapsulated locations, which we express by a framing judgment $\models bnd(M)\ \mathsf{frm}\ I$. Requirement (E3) says the client stays outside boundaries, a part of the meaning of the correctness judgment for C; more on this in Section 6.3. Finally, (E4) requires that the invariant holds initially; we simply require that I follows from the main program's precondition ($P \Rightarrow I$). Rule MLink is derived in

$$\dfrac{\dfrac{\dfrac{\Phi \vdash_\bullet C : P \rightsquigarrow Q\ [\varepsilon]}{\Phi \circledcirc I \vdash_\bullet C : (P \rightsquigarrow Q\ [\varepsilon]) \circledcirc I}\ \text{SOF} \quad \Phi \circledcirc I \vdash_M B : \Phi(m) \circledcirc I}{\vdash_\bullet \text{let } m = B \text{ in } C : (P \rightsquigarrow Q\ [\varepsilon]) \circledcirc I}\ \textsc{Link}}{\vdash_\bullet \text{let } m = B \text{ in } C : P \rightsquigarrow Q\ [\varepsilon]}\ \textsc{Conseq}$$

Fig. 24. Derivation of MLink, with side conditions $mdl(m) = M$, $\models bnd(M)$ frm I, and $P \Rightarrow I$.

Figure 24. The side conditions $\models bnd(M)$ frm I, and $P \Rightarrow I$ are the responsibility of the module developer. The idea is that precondition P expresses initial conditions for the linked program, e.g., that globals have default values (null for class types, $\varnothing$ for rgn). In our examples, the invariant quantifies over elements of the global variable *pool* and holds when *pool* is empty. For a more sophisticated language, we would have module initialization code to establish the module invariant.

Theorem 6.1 (Soundness of Unary Logic). *All the unary proof rules are sound (Figure 23 and Appendix Figures 35 and 36).*

6.3 How the Proof Rules Ensure Encapsulation

The proof rules for commands must enforce requirement (E3), i.e., a command respects the boundaries of modules in context other than the current module. In part, this is done by what we call ***context introduction*** rules. One may expect a weakening rule that allows additional specs to be added to the context, and indeed there is such a rule (CtxIntroIn1) for the case that the method's module is already in context. If the method's module is not already in context, then adding its spec actually strengthens the property expressed by the judgment, namely, respect of the added module's boundary. For this, we have a rule CtxIntro that extends the context by adding a spec for method m and has side conditions (using separator formulas generated by $\cdot/\cdot$) that ensure both the read and write effects of atomic command A are separate from the boundary of m's module. Two other variations are needed to handle method calls and adding a spec for the current module; these are relegated to the Appendix. (A more elegant treatment may be possible using an explicit calculus of modules and their correctness, but that would have its own intricacies.)

As an example, consider this code that acts on variables s: Stack and c,d: Cell.

```
d.val:=0; push(s,d); d:=new Cell; d.val:=1; push(s,d)
```

Using variable r : rgn and idiomatic precondition $d \in r \wedge r \,\#\, (pool \cup pool`rep)$, this code has frame condition rw $d, r,$ alloc, $r`val$. (Here, we use the spec idiom depicted in Figure 3.) The small axiom for the store command $d.val := 0$ says it reads d and writes $d.val$. To add the Stack module to this command's context, rule CtxIntro requires the precondition to imply a separator, which when simplified is $\{d\} \,\#\, pool \wedge \{d\} \,\#\, pool`rep$. This says d is neither in *pool* nor in any *rep* unless d is null.

There is also a rule to change the current module from the default module used in, e.g., rules Call, FieldUpd, and Alloc. In a proof these and the context introduction rules are used at the "leaves" of the proof, i.e., for atomic commands, to introduce the intended modules. This organization is the same as used previously in RLII. However, here the notion of encapsulation is stronger. To enforce that reads do not transgress boundaries (r-respect in Definition 5.10), the proof rules for If and While also have side conditions to ensure the conditional expressions are separate from boundaries. For test expression E, the condition is $(+N \in \Phi, N \neq M.\ bnd(N)) \cdot/\cdot\ r2w(ftpt(E))$. This separator formula simplifies to true or false depending on whether any variable in E occurs in any of the boundaries of modules N in scope other than the current module M. Although the details are different from RLII, the general idea is the same, so we relegate most of these rules to the Appendix (see Figure 35 and Remark 8). Relevant examples can be found in Section 8 of RLII.

$$
\begin{array}{ll}
\sigma|\sigma' \models_\pi (\!|P|\!) & \text{iff } \sigma \models P \\
\sigma|\sigma' \models_\pi F \doteq F' & \text{iff } \sigma(F) \overset{\pi}{\sim} \sigma'(F') \\
\sigma|\sigma' \models_\pi \mathbb{A}G\text{`}f & \text{iff } \mathit{Agree}(\sigma, \sigma', \pi, \mathrm{rd}\, G\text{`}f) \text{ and } \mathit{Agree}(\sigma', \sigma, \pi^{-1}, \mathrm{rd}\, G\text{`}f) \\
\sigma|\sigma' \models_\pi \mathbb{A}x & \text{iff } \sigma(x) \overset{\pi}{\sim} \sigma'(x) \\
\sigma|\sigma' \models_\pi \Diamond \mathcal{P} & \text{iff } \sigma|\sigma' \models_\rho \mathcal{P} \text{ for some } \rho \supseteq \pi \\
\sigma|\sigma' \models_\pi \mathcal{P} \Rightarrow \mathcal{Q} & \text{iff } \sigma|\sigma' \models_\pi \mathcal{P} \text{ implies } \sigma|\sigma' \models_\pi \mathcal{Q} \\
\sigma|\sigma' \models \mathcal{P} & \text{iff } \sigma|\sigma' \models_\pi \mathcal{P} \text{ for all } \pi \\
\models \mathcal{P} & \text{iff } \sigma|\sigma' \models \mathcal{P} \text{ for all } \sigma, \sigma'
\end{array}
$$

Fig. 25. Relation formula semantics $\sigma|\sigma' \models_\pi^{\Gamma|\Gamma'} \mathcal{P}$ (selected). See Appendix Figure 37 for other cases.

7 BIPROGRAMS: SEMANTICS AND CORRECTNESS

This section defines (in Section 7.2) the relational analog of the pre-models used in unary program semantics of Section 5.3. This is used (in Section 7.3) to define the transition semantics of biprograms. Some details are intricate, as needed to ensure quasi-determinacy and to ensure that a biprogram execution faithfully represents a pair of unary executions. On this basis, the semantics of relational judgments is defined and shown to entail the expected relational property of unary executions (Section 7.4). The first step is to define the semantics of relation formulas (Section 7.1).

7.1 Relation Formulas

Refperms and agreement, the basis for semantics of read effects, are also used for semantics of agreement formulas. For relation formulas, satisfaction $\sigma|\sigma' \models_\pi \mathcal{P}$ says state σ relates to σ' according to $\mathcal{P}$ and refperm π (see Figure 25). The propositional connectives have classical semantics. Formula $\mathcal{P}$ is called ***valid*** if $\models \mathcal{P}$.

Recall that semantic agreement (*Lagree*, *Agree*) is skewed in the sense that region expressions are evaluated in the left state, as noted following Equation (22). The semantics of $\mathbb{A}G\text{`}f$ uses agreement via refperm $\pi$ and agreement via $\pi^{-1}$ for the swapped pair of states. As a result, $\sigma|\sigma' \models_\pi \mathbb{A}G\text{`}f$ implies not only $\sigma(G) \subseteq \mathit{dom}(\pi)$ but also $\sigma'(G) \subseteq \mathit{rng}(\pi)$. However, $\mathbb{A}G\text{`}f$ does not imply $G \doteq G$ in general. So the form $G \doteq G \wedge \mathbb{A}G\text{`}f$ is often used, e.g., formula (11); in particular, it appears in the agreements from a read framed effect.

The formulas $\mathbb{A}G\text{`}f$ and $G\text{`}f \doteq G\text{`}f$ have different meaning and in general are incomparable. In case $f$ : int, the region $G\text{`}f$ is empty in which case $\mathbb{A}G\text{`}f$ implies $G\text{`}f \doteq G\text{`}f$ trivially. Using a diagram like in Figure 17, Figure 26 shows two states and a refperm such that $\mathbb{A}\{x\}\text{`}f$ holds (noting that $(q, q') \in \pi$ and $(r, r') \in \pi$). But $\{x\}\text{`}f \doteq \{x\}\text{`}f$ does not; we have $\sigma(\{x\}\text{`}f) = \{q\}$ and $\sigma'(\{x\}\text{`}f) = \{r'\}$ but $(q, r') \notin \pi$. Also $\{x\} \doteq \{x\}$ is false, because $(o, p') \notin \pi$.

Here are some valid schemas: $\mathcal{P} \Rightarrow \Diamond\mathcal{P}$, $\Diamond\Diamond\mathcal{P} \Rightarrow \Diamond\mathcal{P}$, and $\Diamond(\mathcal{P} \wedge \mathcal{Q}) \Rightarrow \Diamond\mathcal{P} \wedge \Diamond\mathcal{Q}$. Another validity is (alloc $\doteq$ alloc) $\wedge \Diamond\mathcal{P} \Rightarrow \mathcal{P}$, in which alloc $\doteq$ alloc says the refperm is a total bijection on allocated references. The strong condition alloc $\doteq$ alloc is not local, and is not a useful requirement for most purposes.

Validity of $\mathcal{P} \Rightarrow \Box\mathcal{P}$ is equivalent to $\mathcal{P}$ being ***refperm monotonic***, i.e., not falsified by extension of the refperm. Agreement formulas are refperm monotonic, as a consequence of Equation (21). A key fact is

$$\text{If } \mathcal{Q} \Rightarrow \Box\mathcal{Q} \text{ is valid, then so is } \Diamond\mathcal{P} \wedge \mathcal{Q} \Rightarrow \Diamond(\mathcal{P} \wedge \mathcal{Q}). \tag{30}$$

Validity of $\Diamond\mathcal{P} \Rightarrow \mathcal{P}$ expresses that $\mathcal{P}$ is ***refperm-independent***, i.e., $\sigma|\sigma' \models_\pi \mathcal{P}$ iff $\sigma|\sigma' \models_\rho \mathcal{P}$, for all $\sigma, \sigma', \pi, \rho$. If $\mathcal{P}$ contains no agreement formula, then it is refperm-independent (even if $\Diamond$

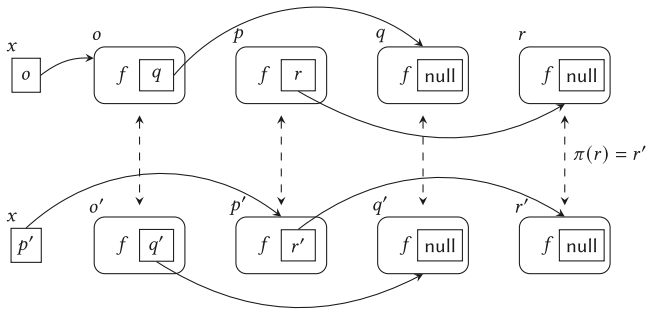


Fig. 26. Refperm π and states σ, σ' that satisfy $\mathbb{A}\{x\}\text{`}f$ but neither $\{x\} \doteq \{x\}$ nor $\{x\}\text{`}f \doteq \{x\}\text{`}f$.

occurs in $\mathcal{P}$). For such formulas the condition in Equation (30) can be strengthened:

$$\text{If } \Diamond Q \Rightarrow Q \text{ is valid, then so is } \Diamond \mathcal{P} \wedge Q \iff \Diamond(\mathcal{P} \wedge Q). \tag{31}$$

Syntactic projection is weakening: $\mathcal{P} \Rightarrow \{\!|P|\!\} \wedge [\![P']\!]$ where P is $\overleftarrow{\mathcal{P}}$ and P' is $\overrightarrow{\mathcal{P}}$. The implication is strict, in general, because projection discards agreements (Figure 15). Syntactic projection is not $\Rightarrow$-monotonic: for boolean variable x, the formula $x \doteq x \wedge [\![x > 0]\!] \Rightarrow \{\!|x > 0|\!\}$ is valid, but $\overleftarrow{x \doteq x \wedge [\![x > 0]\!]} \equiv \mathit{true} \wedge \mathit{true}$ and $\overleftarrow{\{\!|x > 0|\!\}} \equiv x > 0$. The example also shows that agreements can have unary consequences. As another example, this is valid: $\Diamond(x \doteq x' \wedge x \doteq y') \Rightarrow [\![x' = y']\!]$. The antecedent holds if the refperm relates the value of x to both the values of x' and y', or can be extended to do so. Neither is possible if the value of x' is different from the value of y'.

The framing judgment generalizes the unary version (27).

Definition 7.1 (Framing Judgment). Let $\mathcal{P} \models \eta|\eta' \text{ frm } Q$ iff for all $\pi, \sigma, \sigma', \tau, \tau'$, if $\mathit{Agree}(\sigma, \tau, \eta)$, $\mathit{Agree}(\sigma', \tau', \eta')$, and $\sigma|\sigma' \models_\pi \mathcal{P} \wedge Q$ then $\tau|\tau' \models_\pi Q$.

For example, $G \doteq G \models \eta|\eta \text{ frm } \mathbb{A}G\text{`}f$ where $\eta$ is $\mathit{ftpt}(G), \mathrm{rd}\, G\text{`}f$ (Lemma C.2). Apropos relations of the form $\mathcal{R} \mathrel{\hat{=}} G \doteq G \wedge \mathbb{A}G\text{`}f$, we have $\models \delta|\delta \text{ frm } \mathcal{R}$ where $\delta$ is $\mathit{ftpt}(G), \mathrm{rd}\, G\text{`}f$. If $P \models \eta \text{ frm } Q$, then $\{\!|P|\!\} \models \eta|\bullet \text{ frm } \{\!|Q|\!\}$ (and same on the right). Also, $\models \mathit{ftpt}(F)|\mathit{ftpt}(F') \text{ frm } F \doteq F'$, which can be shown using the footprint agreement lemma (28).

The subeffect judgment $\mathcal{P} \models (\varepsilon|\varepsilon') \leq (\eta|\eta')$ is also a direct generalization of the unary version: the inclusions of Equation (26) hold on both sides, for σ, σ', π with $\sigma|\sigma' \models_\pi \mathcal{P}$.

Definition 7.2 (Substitution Notation). If $\Gamma, x{:}T|\Gamma', x'{:}T' \vdash \mathcal{P}$, $\sigma \in [\![\Gamma]\!]$, $v \in [\![T]\!]\sigma$, $\sigma' \in [\![\Gamma']\!]$, and $v' \in [\![T']\!]\sigma'$, then we write $\sigma|\sigma' \models^{\Gamma|\Gamma'} \mathcal{P}^{x|x'}_{v|v'}$ to abbreviate $[\sigma{+}x{:}\,v]|[\sigma'{+}x'{:}\,v'] \models^{\Gamma, x:T|\Gamma', x':T'} \mathcal{P}$.

7.2 Relational Pre-models

A relational pre-model involves two unary pre-models (Definition 5.7) together with a function on state pairs as appropriate for the denotation of a biprogram. This function is subject to similar conditions as for unary pre-models, and must also be compatible with its two unary pre-models.

Definition 7.3 (State Pair ISO $\overset{\pi|\pi'}{\approx}$, $\cong_{\pi|\pi'}$). Building on Definition 5.5, we define isomorphism of state pairs modulo refperms: $(\sigma|\sigma') \overset{\pi|\pi'}{\approx} (\tau|\tau')$ iff $\sigma \overset{\pi}{\approx} \tau$ and $\sigma' \overset{\pi'}{\approx} \tau'$. For relational outcome sets S and S', *i.e.*, S and S' are in $\mathbb{P}((\![\Gamma]\!] \times [\![\Gamma']\!]) \cup \{\lightning\})$, define $S \cong_{\pi|\pi'} S'$ (read ***equivalence mod*** π, π') to mean that (i) $\lightning \in S$ iff $\lightning \in S'$; (ii) for all state pairs $(\sigma|\sigma') \in S$ and $(\tau|\tau') \in S'$ there are ρ, ρ' with $\rho \supseteq \pi$ and $\rho' \supseteq \pi'$, such that $(\sigma|\sigma') \overset{\rho|\rho'}{\approx} (\tau|\tau')$; and (iii) $S \backslash \{\lightning\} = \varnothing$ iff $S' \backslash \{\lightning\} = \varnothing$.

Definition 7.4. A ***relational pre-model*** for $\Gamma|\Gamma'$ is a triple $\varphi = (\varphi_0, \varphi_1, \varphi_2)$ with $dom(\varphi_0) = dom(\varphi_1) = dom(\varphi_2)$, such that φ_0 (respectively, φ_1) is a unary pre-model for Γ (respectively, Γ') (Definition 5.7), and for each m, the ***bi-model*** $\varphi_2(m)$ is a function $\varphi_2(m) : [\![\Gamma]\!] \times [\![\Gamma']\!] \to \mathbb{P}([\![\Gamma]\!] \times [\![\Gamma']\!] \cup \{\lightning\})$ such that

(***fault determinacy***) $\lightning \in \varphi_2(m)(\sigma|\sigma')$ implies $\varphi_2(m)(\sigma|\sigma') = \{\lightning\}$,

(***state determinacy***) $(\sigma|\sigma') \overset{\pi|\pi'}{\approx} (\tau|\tau')$ implies $\varphi_2(m)(\sigma|\sigma') \cong_{\pi|\pi'} \varphi_2(m)(\tau|\tau')$,

(***divergence determinacy***) $(\sigma|\sigma') \overset{\pi|\pi'}{\approx} (\tau|\tau')$ implies that $\varphi_2(m)(\sigma|\sigma') = \varnothing$ iff $\varphi_2(m)(\tau|\tau') = \varnothing$.

Moreover, $\varphi_0, \varphi_1, \varphi_2$ must be compatible in the following sense:

(***unary compatibility***) $\tau|\tau' \in \varphi_2(m)(\sigma|\sigma') \Rightarrow \tau \in \varphi_0(m)(\sigma) \wedge \tau' \in \varphi_1(m)(\sigma')$,
(***relational compatibility***) $\tau \in \varphi_0(m)(\sigma) \wedge \tau' \in \varphi_1(m)(\sigma') \Rightarrow \tau|\tau' \in \varphi_2(m)(\sigma|\sigma') \vee \lightning \in \varphi_2(m)(\sigma|\sigma')$,
(***fault compatibility***) $\lightning \in \varphi_0(m)(\sigma) \vee \lightning \in \varphi_1(m)(\sigma') \Rightarrow \lightning \in \varphi_2(m)(\sigma|\sigma')$.

We do not require $\lightning \in \varphi_2(m)(\sigma|\sigma')$ to imply $\lightning \in \varphi_0(m)(\sigma)$ or $\lightning \in \varphi_1(m)(\sigma')$. The bi-model denoted by a biprogram may fault due to relational precondition, or alignment conditions, even though the underlying commands do not fault.

Lemma 7.5 (Empty Outcome Sets). *For any relational pre-model φ, $\varphi_2(m)(\sigma|\sigma') = \varnothing$ implies that $\varphi_0(m)(\sigma) = \varnothing$ or $\varphi_1(m)(\sigma') = \varnothing$.*

Proof. If either $\varphi_0(m)(\sigma)$ or $\varphi_1(m)(\sigma')$ contains fault, then so does $\varphi_2(m)(\sigma|\sigma')$, by fault compatibility; and if both $\varphi_0(m)(\sigma)$ and $\varphi_1(m)(\sigma')$ contain states, say $\tau \in \varphi_0(m)(\sigma)$ and $\tau' \in \varphi_1(m)(\sigma')$, then by relational compatibility $\varphi_2(m)(\sigma|\sigma')$ contains either $(\tau|\tau')$ or $\lightning$. □

In a relational pre-model, the bi-model outcome sets are convex in this sense:

$$\tau|\tau' \in \varphi_2(m)(\sigma|\sigma') \text{ and } \upsilon|\upsilon' \in \varphi_2(m)(\sigma|\sigma') \text{ imply } \tau|\upsilon' \in \varphi_2(m)(\sigma|\sigma') \text{ and } \upsilon|\tau' \in \varphi_2(m)(\sigma|\sigma').$$

This is a consequence of unary compatibility, relational compatibility, and fault determinacy. But it is not a consequence of the three conditions imposed on bi-models alone.

7.3 Biprogram Transition Relation

Biprograms are given transition semantics by relation $\overset{\varphi}{\Longmapsto}$ on configurations, defined in Figures 27 and 28 for any (relational) pre-model φ. Configurations have the form $\langle CC, \sigma|\sigma', \mu|\mu'\rangle$, which represents an aligned pair of unary configurations. These have projections $\overleftarrow{\langle CC, \sigma|\sigma', \mu|\mu'\rangle} \triangleq \langle \overleftarrow{CC}, \sigma, \mu\rangle$ and $\overrightarrow{\langle CC, \sigma|\sigma', \mu|\mu'\rangle} \triangleq \langle \overrightarrow{CC}, \sigma', \mu'\rangle$. Environments are unchanged from unary semantics: μ and μ' map procedure names to commands, not biprograms.[36] The rules are designed to ensure quasi-determinacy (see Lemma C.8).

The bi-com $(C|C')$ represents a pair of programs for which the only alignment of interest is the initial states and the final states (if any). Its steps are dovetailed, unless one side has terminated, so that divergence on one side cannot prevent progress on the other side. It make direct use of the unary transition relation. The exact order of dovetailing does not matter; what matters is that one-sided divergence is not possible. Here are the details of the specific formulation we have chosen. The bi-com $(C|C')$ takes a step on the left (rule bComL in Figure 27), leaving the right side unchanged. It transitions to the ***r-bi-com*** form $(C|^{\triangleright}C')$, which does not occur in source programs, and which takes a right step (bComR). In configurations, identifier CC ranges over biprograms

[36] This simplification streamlines the development but is revisited in Section 8.5.

$$\textsc{bSync}\ \frac{A \text{ not a method call} \qquad \langle A,\ \sigma,\ \mu\rangle \overset{\varphi_0}{\longmapsto} \langle \mathsf{skip},\ \tau,\ \nu\rangle \qquad \langle A,\ \sigma',\ \mu'\rangle \overset{\varphi_1}{\longmapsto} \langle \mathsf{skip},\ \tau',\ \nu'\rangle}{\langle \lfloor A\rfloor,\ \sigma|\sigma',\ \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \langle \lfloor \mathsf{skip}\rfloor,\ \tau|\tau',\ \nu|\nu'\rangle}$$

$$\textsc{bSyncX}\ \frac{A \text{ not a method call} \qquad \langle A,\ \sigma,\ \mu\rangle \overset{\varphi_0}{\longmapsto} \lightning \quad \text{or} \quad \langle A,\ \sigma',\ \mu'\rangle \overset{\varphi_1}{\longmapsto} \lightning}{\langle \lfloor A\rfloor,\ \sigma|\sigma',\ \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \lightning}$$

$$\textsc{bCallS}\ \frac{(\tau|\tau') \in \varphi_2(m)(\sigma|\sigma')}{\langle \lfloor m()\rfloor,\ \sigma|\sigma',\ \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \langle \lfloor \mathsf{skip}\rfloor,\ \tau|\tau',\ \mu|\mu'\rangle} \qquad \textsc{bCallX}\ \frac{\lightning \in \varphi_2(m)(\sigma|\sigma')}{\langle \lfloor m()\rfloor,\ \sigma|\sigma',\ \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \lightning}$$

$$\textsc{bCall0}\ \frac{\varphi_2(m)(\sigma|\sigma') = \varnothing}{\langle \lfloor m()\rfloor,\ \sigma|\sigma',\ \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \langle \lfloor m()\rfloor,\ \sigma|\sigma',\ \mu|\mu'\rangle}$$

$$\textsc{bCallE}\ \frac{\mu(m) = B \qquad \mu'(m) = B'}{\langle \lfloor m()\rfloor,\ \sigma|\sigma',\ \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \langle (B|B');\lfloor \mathsf{ecall}(m)\rfloor,\ \sigma|\sigma',\ \mu|\mu'\rangle}$$

$$\textsc{bComL}\ \frac{\langle C,\ \sigma,\ \mu\rangle \overset{\varphi_0}{\longmapsto} \langle D,\ \tau,\ \nu\rangle \qquad DD = ((D \fatsemi C') \text{ if } (C' \not\equiv \mathsf{skip}) \text{ else } (D|\mathsf{skip}))}{\langle (C|C'),\ \sigma|\sigma',\ \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \langle DD,\ \tau|\sigma',\ \nu|\mu'\rangle}$$

$$\textsc{bComR}\ \frac{\langle C',\ \sigma',\ \mu'\rangle \overset{\varphi_1}{\longmapsto} \langle D',\ \tau',\ \nu'\rangle}{\langle (C \fatsemi C'),\ \sigma|\sigma',\ \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \langle (C|D'),\ \sigma|\tau',\ \mu|\nu'\rangle}$$

$$\textsc{bComR0}\ \frac{\langle C',\ \sigma',\ \mu'\rangle \overset{\varphi_1}{\longmapsto} \langle D',\ \tau',\ \nu'\rangle}{\langle (\mathsf{skip}|C'),\ \sigma|\sigma',\ \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \langle (\mathsf{skip}|D'),\ \sigma|\tau',\ \mu|\nu'\rangle}$$

$$\textsc{bComLX}\ \frac{\langle C,\ \sigma,\ \mu\rangle \overset{\varphi_0}{\longmapsto} \lightning}{\langle (C|C'),\ \sigma|\sigma',\ \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \lightning} \qquad \textsc{bComRX}\ \frac{\langle C',\ \sigma',\ \mu'\rangle \overset{\varphi_1}{\longmapsto} \lightning \qquad BB \text{ is } (C \fatsemi C') \text{ or } (\mathsf{skip}|C')}{\langle BB,\ \sigma|\sigma',\ \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \lightning}$$

$$\textsc{bLet}\ \frac{\nu = [\mu{+}m{:}\,C] \qquad \nu' = [\mu'{+}m{:}\,C']}{\langle \mathsf{let}\ m = (C|C')\ \mathsf{in}\ DD,\ \sigma|\sigma',\ \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \langle DD;\lfloor \mathsf{elet}(m)\rfloor,\ \sigma|\sigma',\ \nu|\nu'\rangle}$$

$$\textsc{bIfTT}\ \frac{\sigma(E) = \mathsf{true} = \sigma'(E')}{\langle \mathsf{if}\ E|E'\ \mathsf{then}\ CC\ \mathsf{else}\ DD,\ \sigma|\sigma',\ \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \langle CC,\ \sigma|\sigma',\ \mu|\mu'\rangle}$$

$$\textsc{bIfFF}\ \frac{\sigma(E) = \mathsf{false} = \sigma'(E')}{\langle \mathsf{if}\ E|E'\ \mathsf{then}\ CC\ \mathsf{else}\ DD,\ \sigma|\sigma',\ \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \langle DD,\ \sigma|\sigma',\ \mu|\mu'\rangle}$$

$$\textsc{bIfX}\ \frac{\sigma(E) \neq \sigma'(E')}{\langle \mathsf{if}\ E|E'\ \mathsf{then}\ CC\ \mathsf{else}\ DD,\ \sigma|\sigma',\ \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \lightning}$$

$$\textsc{bVar}\ \frac{\begin{array}{c} w = \mathit{FreshVar}(\sigma) \qquad w' = \mathit{FreshVar}(\sigma') \qquad \tau = [\sigma{+}w{:}\,\mathit{default}(T)] \\ \tau' = [\sigma'{+}w'{:}\,\mathit{default}(T')] \qquad DD = (\lfloor \mathsf{evar}(w)\rfloor \text{ if } w \equiv w' \text{ else } (\mathsf{evar}(w)|\mathsf{evar}(w'))) \end{array}}{\langle \mathsf{var}\ x{:}T|x'{:}T'\ \mathsf{in}\ CC,\ \sigma|\sigma',\ \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \langle CC^{x,x'}_{w,w'};DD,\ \tau|\tau',\ \mu|\mu'\rangle}$$

$$\textsc{bSeq}\ \frac{\langle BB,\ \sigma|\sigma',\ \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \langle CC,\ \tau|\tau',\ \nu|\nu'\rangle}{\langle BB;DD,\ \sigma|\sigma',\ \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \langle CC;DD,\ \tau|\tau',\ \nu|\nu'\rangle} \qquad \textsc{bSeqX}\ \frac{\langle BB,\ \sigma|\sigma',\ \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \lightning}{\langle BB;DD,\ \sigma|\sigma',\ \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \lightning}$$

Fig. 27. Transition rules for biprograms, except bi-while (for which see Figure 28).

$$\textsc{bWhL}\ \frac{\sigma(E) = \text{true} \qquad \sigma|\sigma' \models \mathcal{P}}{\langle CC,\ \sigma|\sigma',\ \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \langle (\overleftarrow{BB}|\text{skip}); CC,\ \sigma|\sigma',\ \mu|\mu'\rangle}$$

$$\textsc{bWhR}\ \frac{\sigma'(E') = \text{true} \qquad \sigma|\sigma' \models \mathcal{P}' \qquad (\sigma(E) = \text{false or } \sigma|\sigma' \not\models \mathcal{P})}{\langle CC,\ \sigma|\sigma',\ \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \langle (\text{skip}|\overrightarrow{BB}); CC,\ \sigma|\sigma',\ \mu|\mu'\rangle}$$

$$\textsc{bWhTT}\ \frac{\sigma|\sigma' \not\models \mathcal{P} \qquad \sigma|\sigma' \not\models \mathcal{P}' \qquad \sigma(E) = \text{true} = \sigma'(E')}{\langle CC,\ \sigma|\sigma',\ \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \langle BB; CC,\ \sigma|\sigma',\ \mu|\mu'\rangle}$$

$$\textsc{bWhFF}\ \frac{\sigma(E) = \text{false} = \sigma'(E')}{\langle CC,\ \sigma|\sigma',\ \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \langle \lfloor \text{skip} \rfloor,\ \sigma|\sigma',\ \mu|\mu'\rangle}$$

$$\textsc{bWhX}\ \frac{\begin{array}{c}(\sigma(E) = \text{true and } \sigma'(E') = \text{false and } \sigma|\sigma' \not\models \mathcal{P}) \\ \text{or } (\sigma(E) = \text{false and } \sigma'(E') = \text{true and } \sigma|\sigma' \not\models \mathcal{P}')\end{array}}{\langle CC,\ \sigma|\sigma',\ \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \lightning}$$

Fig. 28. Transition rules for bi-while, in which we abbreviate $CC \equiv \text{while } E|E' \cdot \mathcal{P}|\mathcal{P}' \text{ do } BB$.

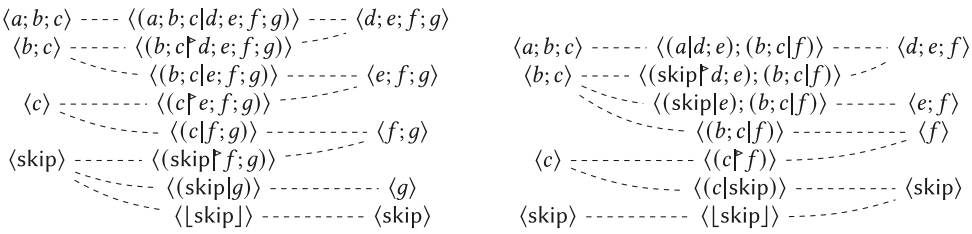


Fig. 29. Two example biprogram traces, with alignments, omitting states and environments.

that may include endmarkers from the unary semantics and also the r-bi-com.[37] Rule bComR0 is needed to handle biprograms of the form (skip|D). The rules ensure that (skip⋔D) never occurs for $D \not\equiv$ skip, and we identify (skip⋔skip) ≡ ⌊skip⌋.

Rules bSeq and bSeqX simply close the transitions under command sequencing. Recall that we identify some biprograms, e.g., (skip|skip) ≡ ⌊skip⌋, to avoid the need for bureaucratic transitions (see Figure 6). A ***trace*** T via φ is a finite sequence of configurations that is consecutive under $\overset{\varphi}{\Longmapsto}$. The projection lemma (Lemma 7.8) confirms that T gives rise to unary trace U on the left via $\overset{\varphi_0}{\longmapsto}$ and V on the right via $\overset{\varphi_1}{\longmapsto}$.

Example 7.6. To illustrate the dovetailed execution of bi-coms, we show a trace for the bi-com $(a; b; c|d; e; f; g)$ of some atomic commands, omitting states and environments from the configurations. The trace is displayed vertically on the left side of Figure 29, between the two corresponding unary traces. Thus, $(a; b; c|d; e; f; g)$ executes the commands in the order a, d, b, e, c, f, g. Dashed lines in the figure show the correspondence between unary and biprogram configurations. In this example, the right side takes additional steps after the left has terminated. The opposite can also happen, as in $\langle (a; b; c|d)\rangle\langle (b; c⋔ d)\rangle\langle (b; c|\text{skip})\rangle\langle (c|\text{skip})\rangle\langle \lfloor \text{skip} \rfloor\rangle$, which executes a, d, b, c.

The right side of Figure 29 shows a trace for the second of the weavings in Equation (12).

[37] The left and right projections of $(-⋔ -)$ are as with $(-|-)$.

The sync atomic command $\lfloor A \rfloor$ steps A by unary transition on both sides, unless A is a context call in which case the context bi-model is used. Endmarkers are considered to be atomic commands, e.g., $\lfloor \mathsf{elet}(m) \rfloor$ transitions via rule bSync and removes m from the environment on both sides.

A bi-if, if $E|E'$ then CC else DD, faults from initial states that do not agree on the tests E, E', which we call an ***alignment fault*** (rule biIfX). A bi-while, while $E|E' \cdot \mathcal{P}|\mathcal{P}'$ do CC, executes the left part of the body, $\overleftarrow{CC}$, if E and the left alignment guard $\mathcal{P}$ both hold, and *mutatis mutandis* for the right. If neither alignment guard holds, then the loop faults unless the tests E, E' agree (bWhX).

The transition relation $\stackrel{\varphi}{\Longmapsto}$ uses the unary models φ_0 and φ_1 for method calls in the bi-com form, e.g., $(m()|\mathsf{skip})$ goes via φ_0 according to bComL. A sync'd call $\lfloor m() \rfloor$ in the body of a loop that has non-false left or right alignment guards may give rise to steps where the active biprogram has the form $(m(); C|D)$ or $(\mathsf{skip}|m(); C)$ (rules bWhL, bWhR). The ***active biprogram***, like the active command in a unary configuration, is the unique sub-biprogram that gets rewritten by the applicable transition rule. As with unary programs, we define $\mathit{Active}(CC)$ to be the unique BB such that $CC \equiv BB; DD$ for some DD and BB is not a sequence; it is what gets rewritten by the applicable transition rule.

Projecting from a biprogram trace does not simply mean mapping the syntactic projections over the trace, because that would result in stuttering steps that do not arise in the unary semantics (where stuttering only happens for context calls and only if the model returns an empty set). In the preceding diagrams, some unary configurations correspond with more than one biprogram configuration; one may say the unary program is idling while a step is taken on the other side.

The alignment of biprogram traces with unary ones is formalized as follows. Here, we treat a trace T as a map defined on an initial segment of the naturals, so $\mathit{dom}(T)$ is the set $\{0, \ldots, \mathit{len}(T) - 1\}$.

Definition 7.7 (Schedule, Alignment, align(l, r, T, U, V)). Let T be a biprogram trace and U, V unary traces. A ***schedule of*** U, V ***for*** T is a pair l, r with $l : (\mathit{dom}(T)) \to (\mathit{dom}(U))$ and $r : (\mathit{dom}(T)) \to (\mathit{dom}(V))$, each surjective and monotonic. A schedule l, r is an ***alignment*** of U, V for T, written $\mathit{align}(l, r, T, U, V)$, iff $U_{l(i)} = \overleftarrow{T_i}$ and $V_{r(i)} = \overrightarrow{T_i}$ for all i in $\mathit{dom}(T)$.

The dashed lines in Figure 29 represent the l and r index mappings of a schedule. For Example 7.6, left side of the figure, the mapping is $r(0) = 0$, $r(1) = 0$, $r(2) = 1$, and so on.

The following result makes precise that every biprogram trace represents a pair of unary traces. It is phrased carefully to take into account the possibility of stuttering transitions at the unary level.

Lemma 7.8 (Trace Projection). *Suppose φ is a pre-model. Then the following hold. (a) For any step $\langle BB,\ \sigma|\sigma',\ \mu|\mu' \rangle \stackrel{\varphi}{\Longmapsto} \langle CC,\ \tau|\tau',\ \nu|\nu' \rangle$, either*

- $\langle \overleftarrow{BB},\ \sigma,\ \mu \rangle \stackrel{\varphi_0}{\longmapsto} \langle \overleftarrow{CC},\ \tau,\ \nu \rangle$ *and* $\langle \overrightarrow{BB},\ \sigma',\ \mu' \rangle \stackrel{\varphi_1}{\longmapsto} \langle \overrightarrow{CC},\ \tau',\ \nu' \rangle$, *or*
- $\langle \overleftarrow{BB},\ \sigma,\ \mu \rangle = \langle \overleftarrow{CC},\ \tau,\ \nu \rangle$ *and* $\langle \overrightarrow{BB},\ \sigma',\ \mu' \rangle \stackrel{\varphi_1}{\longmapsto} \langle \overrightarrow{CC},\ \tau',\ \nu' \rangle$, *or*
- $\langle \overleftarrow{BB},\ \sigma,\ \mu \rangle \stackrel{\varphi_0}{\longmapsto} \langle \overleftarrow{CC},\ \tau,\ \nu \rangle$ *and* $\langle \overrightarrow{BB},\ \sigma',\ \mu' \rangle = \langle \overrightarrow{CC},\ \tau',\ \nu' \rangle$.

(b) For any trace T via $\stackrel{\varphi}{\Longmapsto}$, there are unique traces U via $\stackrel{\varphi_0}{\longmapsto}$ and V via $\stackrel{\varphi_1}{\longmapsto}$, and schedule l, r, such that $\mathit{align}(l, r, T, U, V)$.
(c) If $\mathit{Active}(BB) \equiv \lfloor\!\lfloor B \rfloor\!\rfloor$ for some B, then $\langle \overleftarrow{BB}, \sigma, \mu \rangle \stackrel{\varphi_0}{\longmapsto} \langle \overleftarrow{CC}, \tau, \nu \rangle$ and $\langle \overrightarrow{BB}, \sigma', \mu' \rangle \stackrel{\varphi_1}{\longmapsto} \langle \overrightarrow{CC}, \tau', \nu' \rangle$.

7.4 Relational Context Models, Biprogram Correctness, and Adequacy

Owing to careful design of Definitions 5.9, 5.10, and 7.4, the following notions are mostly about relational aspects. Relational context models are pre-models that satisfy some specs. They play

the same role in the semantics of relational judgments as unary context models play in unary correctness.

Definition 7.9 (Context Model of Relational Spec, Φ-model). A pre-model φ is a Φ-***model*** provided that φ_0, φ_1 are Φ_0, Φ_1-models, and for each m, with $\Phi_2(m) = \mathcal{R} \approx\!> \mathcal{S}\ [\eta|\eta']$, the bi-model $\varphi_2(m)$ satisfies the following, for all σ, σ':

(a) $\lightning \in \varphi_2(m)(\sigma, \sigma')$ iff there are no $\pi, \overline{v}, \overline{v}'$ such that $\sigma|\sigma' \models_\pi \mathcal{R}^{\overline{s},\overline{s}'}_{\overline{v},\overline{v}'}$, where $\overline{s}, \overline{s}'$ are the spec-only variables on left and right.

(b) for all (τ, τ') in $\varphi_2(m)(\sigma, \sigma')$, and all $\pi, \overline{v}, \overline{v}'$ such that $\sigma|\sigma' \models_\pi \mathcal{R}^{\overline{s},\overline{s}'}_{\overline{v},\overline{v}'}$, we have $\tau|\tau' \models_\pi \mathcal{S}^{\overline{s},\overline{s}'}_{\overline{v},\overline{v}'}$ and $\sigma{\rightarrow}\tau \models \eta$ and $\sigma'{\rightarrow}\tau' \models \eta'$

A direct consequence of Definition 7.9, together with unary compatibility of pre-models and condition (c) of Definition 5.9, is that for all N with $mdl(m) \preceq N$, letting $\delta \triangleq bnd(N)$, we have

$$(\tau|\tau') \in \varphi_2(m)(\sigma|\sigma') \text{ implies } rlocs(\sigma, \delta) \subseteq rlocs(\tau, \delta) \text{ and } rlocs(\sigma', \delta) \subseteq rlocs(\tau', \delta),$$

and there is also a direct consequence of condition (d) of Definition 5.9.

The projections of Lemma 7.8 are used in the following definition of relational correctness.

Definition 7.10 (Valid Relational Judgment $\Phi \models_M CC : \mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon']$ *).* The judgment is ***valid*** iff the following conditions hold for all states σ and σ', Φ-models φ, refperms π, and values $\overline{v}, \overline{v}'$ such that $\sigma|\sigma' \models_\pi \mathcal{P}^{\overline{s},\overline{s}'}_{\overline{v},\overline{v}'}$ (where $\overline{s}, \overline{s}'$ are the spec-only variables):

(***Safety***) It is not the case that $\langle CC,\ \sigma|\sigma',\ \_|\_\rangle \overset{\varphi}{\Longmapsto}{}^* \lightning$,

(***Post***) $\tau|\tau' \models_\pi \mathcal{Q}^{\overline{s},\overline{s}'}_{\overline{v},\overline{v}'}$ for every τ, τ' with $\langle CC,\ \sigma|\sigma',\ \_|\_\rangle \overset{\varphi}{\Longmapsto}{}^* \langle \lfloor \mathsf{skip} \rfloor,\ \tau|\tau',\ \_|\_\rangle$,

(***Write***) $\sigma{\rightarrow}\tau \models \varepsilon$ and $\sigma'{\rightarrow}\tau' \models \varepsilon'$ for every τ, τ' with $\langle CC,\ \sigma|\sigma',\ \_|\_\rangle \overset{\varphi}{\Longmapsto}{}^* \langle \lfloor \mathsf{skip} \rfloor,\ \tau|\tau',\ \_|\_\rangle$,

(***R-safe***) For every trace T from $\langle CC,\ \sigma|\sigma',\ \_|\_\rangle$, let U, V be the projections of T; then every configuration of U (respectively, V) satisfies r-safe for $(\Phi_0, \varepsilon, \sigma)$ (respectively, $(\Phi_1, \varepsilon', \sigma')$),

(***Encap***) For every trace T from $\langle CC,\ \sigma|\sigma',\ \_|\_\rangle$, let U, V be the projections of T; then every step of U (respectively, V) satisfies respect for $(\Phi_0, M, \varphi_0, \varepsilon, \sigma)$ (respectively, $(\Phi_1, M, \varphi_1, \varepsilon', \sigma')$).

The values of spec-only variables are uniquely determined by the pre-states, just like in unary specs. In virtue of the universal quantification over refperms π, for a spec in standard form $\mathcal{P} \approx\!> \Diamond\mathcal{Q}$, the judgment says for any π that supports the agreements in $\mathcal{P}$ there exists an extension $\rho \supseteq \pi$ that supports the agreements in $\mathcal{Q}$.

The following result confirms that the relational judgment is about unary executions. In particular, a judgment about a bi-com $(C|C')$ implies the expected property relating executions of C and C'. The proof uses the embedding Lemma C.9, which says a biprogram's traces cover all the executions of its unary projections, unless it faults.

Theorem 7.11 (Adequacy). *Consider a valid judgment* $\Phi \models_M CC : \mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon']$. *Consider any* Φ*-model* φ *and any* σ, σ', π *with* $\sigma|\sigma' \models_\pi \mathcal{P}$. *If* $\langle \overleftarrow{CC},\ \sigma,\ \_\rangle \overset{\varphi_0}{\longmapsto}{}^* \langle \mathsf{skip},\ \tau,\ \_\rangle$ *and* $\langle \overrightarrow{CC},\ \sigma',\ \_\rangle \overset{\varphi_1}{\longmapsto}{}^* \langle \mathsf{skip},\ \tau',\ \_\rangle$, *then* $\tau|\tau' \models_\pi \mathcal{Q}$. *Moreover, all executions from* $\langle \overleftarrow{CC},\ \sigma,\ \_\rangle$ *and from* $\langle \overrightarrow{CC},\ \sigma',\ \_\rangle$ *satisfy Safety, Write, R-safe, and Encap in Definition 5.10.*

Remark 1. It is not straightforward to formalize a converse to this result. The judgment about CC says not only that the underlying unary executions are related as in the conclusion of the theorem, but in addition certain intermediate states are in agreement according to the alignment designated by the bi-ifs and bi-whiles in CC.

8 RELATIONAL LOGIC

This section presents the rules for proving relational correctness judgments. Section 8.1 defines how local equivalence specs are derived from unary specs. Section 8.2 gives the proof rules and discusses them, including the derivation of the modular linking rule RMLINK, sketched as Equation (3) in Section 2.1. Section 8.3 considers derived rules involving framing and the $\diamond$ modality. Section 8.4 states and explains the lockstep alignment lemma, which is the key to proving soundness of rules RLOCEQ, RSOF, and RLINK from which RMLINK is derived. Section 8.5 considers nested linking and Section 8.6 addressess unconditional equivalences. For Section 8.4 readers need to be familiar with the semantic definitions in Section 7.

THEOREM 8.1 (SOUNDNESS OF RELATIONAL LOGIC). *All the relational proof rules are sound (Figure 30 and Appendix Figure 38).*

8.1 Local Equivalence

In Section 2.1, we introduced the notion of local equivalence. There is a relational proof rule, RLOCEQ, which lifts a unary judgment to a relational one. The unary read effect, which has an extensional semantics that is relational (Definition 5.10) gets lifted to an explicit relational property, a local equivalence relating a command to itself. As basis for the proof rule, we now formalize a construction, *locEq*, that applies to a unary spec and makes a relational spec—like the spec (9) in Example 4.3, and others in Section 4.6—that expresses equivalence in terms of the given frame condition and takes into account encapsulation boundaries.

Both unary and relational proof rules have conditions to enforce encapsulation with respect to the boundaries of modules in scope. For unary this is discussed in Section 6.3. The semantic condition Encap, in Definition 5.10, refers to a collective boundary. This is an effect formed as a union of the relevant boundaries, for example in the expression $(+N \in \Phi, N \neq M.bnd(N))$ where M is the current module and Φ is the hypothesis context. For brevity, several relational proof rules are expressed using δ to name the collective boundary; in particular, rule RLOCEQ, which introduces the *locEq* spec we now define.

Given a boundary δ and unary spec $P \rightsquigarrow Q\ [\varepsilon]$, the desired pre-relation expresses agreement on the readable locations. Absent a boundary, this can be written $\mathbb{A}\varepsilon$, taking advantage of our abbreviations, which say that $\mathbb{A}\varepsilon$ abbreviates $\mathbb{A}rds(\varepsilon)$, which in turn abbreviates a conjunction of agreement formulas (Figure 14). But, we should avoid requiring agreement on variable alloc, as we want to allow entirely different data structures within boundaries. The requisite agreement can be expressed, using effect subtraction, as $\mathbb{A}(\varepsilon \backslash \delta^{\oplus})$, where δ is the collective boundary of the modules to be respected. Note that $\delta^{\oplus}$ abbreviates δ, rd alloc (as in Definition 5.9).

A first guess for the post-relation would use agreement on the writable locations, but that cannot be written as $\mathbb{A}w2r(\varepsilon)$, because any state-dependent region expressions in write effects of ε should be interpreted in the pre-state. This is why the concluding agreements in the definition of r-respect are expressed in terms of the fresh and written locations. So this is what we need to express in a spec. The solution is to use snapshot variables. If we use fresh variable s_{alloc} in precondition $s_{\text{alloc}} = \text{alloc}$, then the fresh references can be described in post-states as $\text{alloc} \backslash s_{\text{alloc}}$ and agreement on fresh locations can be expressed as $\mathbb{A}(\text{alloc} \backslash s_{\text{alloc}})$`any. For written (pre-existing) locations, we can obtain the requisite agreements in terms of initial snapshots of the locations deemed writable by ε. For an example, see Equation (18) in Section 4.6.

For each $\text{wr}\, G`f$ in $\varepsilon$, we add a snapshot equation $s_{G,f} = G$ to the precondition, or rather $\mathbb{B}(s_{G,f} = G)$. The desired post-relation is then $\mathbb{A}s_{G,f}`f$. Please note that $s_{G,f}$ is just a fresh identifier, written in a way to keep track of its use in connection with $G`f$. The snapshots and agreements are given by functions *snap* and *Asnap* defined next. The following definitions make use of

effects like $\mathsf{rd}\, s_{G,f}\text{`}f$, in which spec-only variables occur. These are used to define agreement formulas used in postconditions—they are not used in frame conditions, where spec-only variables are disallowed.

Definition 8.2 (Write Snapshots). For any effect ε, we define functions *snap* from effects to unary formulas and *Asnap* from effects to read effects:

$snap(\varepsilon, \eta)$	$\hat{=}$	$snap(\varepsilon) \wedge snap(\eta)$	$Asnap(\varepsilon, \eta)$	$\hat{=}$	$Asnap(\varepsilon),\ Asnap(\eta)$
$snap(\mathsf{wr}\, x)$	$\hat{=}$	true	$Asnap(\mathsf{wr}\, x)$	$\hat{=}$	$\mathsf{rd}\, x$ if $x \not\equiv \mathsf{alloc}$ else $\bullet$
$snap(\mathsf{wr}\, G\text{`}f)$	$\hat{=}$	$s_{G,f} = G$	$Asnap(\mathsf{wr}\, G\text{`}f)$	$\hat{=}$	$\mathsf{rd}\, s_{G,f}\text{`}f$
$snap(\mathsf{wr}\, G\text{`any})$	$\hat{=}$	$s_{G,\mathsf{any}} = G$	$Asnap(\mathsf{wr}\, G\text{`any})$	$\hat{=}$	$\mathsf{rd}\, s_{G,\mathsf{any}}\text{`}f, \mathsf{rd}\, s_{G,\mathsf{any}}\text{`}g, \ldots$
$snap(\ldots)$	$\hat{=}$	true	$Asnap(\ldots)$	$\hat{=}$	$\bullet$

Notice that *Asnap* omits alloc and uses the snapshot variables introduced by *snap*.[38] Notice also that in the case $Asnap(\mathsf{wr}\, G\text{`any})$ a single snapshot variable $s_{G,\mathsf{any}}$ is used, but the image expression in $G\text{`any}$ gets expanded to the constituent fields $(f, g, \ldots)$.

The following result confirms that *Asnap* serves the purpose of designating the writable locations from the perspective of the post-state. It uses semantic notions from Sections 5.1 and 5.2.

Lemma 8.3. *If* $\tau \models snap(\varepsilon)$ *and* $\tau \rightarrow v \models \varepsilon$, *then* $wlocs(\tau, \varepsilon) \backslash rlocs(v, \delta^{\oplus}) = rlocs(v, Asnap(\varepsilon) \backslash \delta)$.

The following definition of *locEq* uses effect subtraction to avoid asserting agreement inside the given boundary, in both pre and post. For example, if ε includes $\mathsf{wr}\, x, \mathsf{wr}\, G\text{`}f$, then we convert to read effects and use the snapshot variable: $\mathsf{rd}\, x, \mathsf{rd}\, s_{G,f}\text{`}f$. Then $(\mathsf{rd}\, x, \mathsf{rd}\, s_{G,f}\text{`}f) \backslash \delta$ will remove $x$ if $\mathsf{rd}\, x$ is in $\delta$, and result in $\mathsf{rd}\, (s_{G,f} \backslash H)\text{`}f$ if $\mathsf{rd}\, H\text{`}f$ is in δ.

Definition 8.4 (Local Equivalence). For spec $P \rightsquigarrow Q\ [\varepsilon]$ and boundary δ, define relational spec

$$locEq_{\delta}(P \rightsquigarrow Q\ [\varepsilon]) \ \hat{=}\ \mathbb{B}P \wedge \mathbb{A}\varepsilon_{\delta}^{\leftarrow} \wedge \mathbb{B}(s_{\mathsf{alloc}} = \mathsf{alloc} \wedge snap(\varepsilon)) \twoheadrightarrow \Diamond(\mathbb{B}Q \wedge \mathbb{A}\varepsilon_{\delta}^{\rightarrow})\ [\varepsilon],$$

where $\varepsilon_{\delta}^{\leftarrow} \ \hat{=}\ rds(\varepsilon) \backslash \delta^{\oplus}$ and $\varepsilon_{\delta}^{\rightarrow} \ \hat{=}\ (\mathsf{rd}\, (\mathsf{alloc} \backslash s_{\mathsf{alloc}})\text{`any}, Asnap(\varepsilon)) \backslash \delta$.

For unary context Φ, define $LocEq_{\delta}(\Phi) \ \hat{=}\ (\Phi, \Phi, \Phi_2)$ where $\Phi_2(m)$ is $locEq_{\delta}(\Phi(m))$ for each $m \in \Phi$.

If $P \rightsquigarrow Q\ [\varepsilon]$ and δ are wf in Γ, then $locEq_{\delta}(P \rightsquigarrow Q\ [\varepsilon])$ is wf in $\Gamma|\Gamma$ and has the same spec-only variables on both sides.

Recall from Section 6.3 the Stack client with precondition $P \ \hat{=}\ c \in r \wedge r \,\#\, (pool \cup pool\text{`}rep)$ and frame $\varepsilon \ \hat{=}\ \mathsf{rw}\, c, r, \mathsf{alloc}, r\text{`}val$, where the boundary δ is $\mathsf{rd}\, pool, pool\text{`any}, pool\text{`}rep\text{`any}$. For the precondition, the reads are $\mathsf{rd}\, c, \mathsf{rd}\, r, \mathsf{rd}\, \mathsf{alloc}, \mathsf{rd}\, r\text{`}val$. Subtracting $\delta^{\oplus}$ leaves the variables c, r and is more interesting for $r\text{`}val$. Expanding abbreviation any and discarding empty regions, we are left with $\mathsf{rd}\, (r \backslash (pool \cup pool\text{`}rep))\text{`}val$. So the precondition $\mathbb{A}\varepsilon_{\delta}^{\leftarrow}$ is $\mathbb{A}c \wedge \mathbb{A}r \wedge \mathbb{A}(r \backslash (pool \cup pool\text{`}rep))\text{`}val$. (In conjunction with $\mathbb{B}P$, the formula $\mathbb{A}(r \backslash (pool \cup pool\text{`}rep))\text{`}val$ is equivalent to $\mathbb{A}r\text{`}val$.) There is a snapshot variable in precondition $s_{r,val} = r$, due to $\mathsf{wr}\, r\text{`}val$. It is used in this conjunct of the *Asnap* part of the postcondition: $\mathbb{A}(s_{r,val} \backslash (pool \cup pool\text{`}rep))\text{`}val$.

[38] The snapshot variables used should be distinct from each other, distinct from the ones used in the original spec, and also globally unique so that the local equivalence specs of different methods use different variables. In the definition of *LocEq*, where multiple method specs are considered, we adopt the convention of naming snapshots for method m as $s_{G,f}^{m}$ (and $snap^{m}$, $Asnap^{m}$ for short), to distinguish them from each other and from the snapshots used in the conclusion of a judgment.

$$\textsc{rLink}\ \frac{\begin{array}{c}\Phi,\Theta \vdash_\bullet \lfloor\!\lfloor C \rfloor\!\rfloor : \mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon] \\ \Phi,\Theta \vdash_{mdl(m)} (B|B') : \Theta_2(m) \qquad \Phi_0,\Theta_0 \vdash_{mdl(m)} B : \Theta_0(m) \qquad \Phi_1,\Theta_1 \vdash_{mdl(m)} B' : \Theta_1(m) \\ \delta = (+L \in (\Phi,\Theta).\ bnd(L)) \qquad (\Phi,\Theta) \Rrightarrow LocEq_\delta(\dot\Phi,\dot\Theta) \qquad \mathcal{P} \Rightarrow pre(locEq_\delta(P \rightsquigarrow Q\ [\varepsilon])) \\ \forall N \in \Phi, L \in \Theta.\ N \not\preceq L \qquad \forall N, L.\ N \in \Theta \wedge N \prec L \Rightarrow L \in (\Phi,\Theta) \qquad C \text{ is let-free}\end{array}}{\Phi \vdash_\bullet \text{let } m = (B|B') \text{ in } \lfloor\!\lfloor C \rfloor\!\rfloor : \mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon]}$$

$$\textsc{rWeave}\ \frac{\Phi \vdash DD :\ \mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon'] \qquad CC \leftrightsquigarrow^* DD}{\Phi \vdash CC :\ \mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon']} \qquad \textsc{rCall}\ \frac{\Phi_0 \vdash m() : \Phi_0(m) \qquad \Phi_1 \vdash m() : \Phi_1(m)}{\Phi \vdash \lfloor m() \rfloor : \Phi_2(m)}$$

$$\textsc{rAlloc}\ \frac{(+L \in (\Phi), L \neq M.\ bnd(L)) \cdot\!/.\ \text{wr}\, x, \text{wr alloc}}{\Phi \vdash_M \lfloor x := \text{new } K \rfloor : \text{true} \approx\!> \Diamond(x \doteq x)\ [\text{wr}\, x, \text{rw alloc}]} \qquad \textsc{rEmpPre}\ \Phi \vdash CC :\ \text{false} \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon']$$

$$\textsc{rLocEq}\ \frac{\Phi \vdash_M C :\ P \rightsquigarrow Q\ [\varepsilon] \qquad P \models w2r(\varepsilon) \leq rds(\varepsilon) \qquad \delta = (+N \in \Phi, N \neq M.\ bnd(N)) \qquad C \text{ is let-free}}{LocEq_\delta(\Phi) \vdash_M \lfloor\!\lfloor C \rfloor\!\rfloor : locEq_\delta(P \rightsquigarrow Q\ [\varepsilon])}$$

$$\textsc{rEmb}\ \frac{\Phi_0 \vdash C :\ P \rightsquigarrow Q\ [\varepsilon] \qquad \Phi_1 \vdash C' :\ P' \rightsquigarrow Q'\ [\varepsilon']}{\Phi \vdash (C|C') :\ \langle\!|P|\!\rangle \wedge \langle\!|P'|\!\rangle \approx\!> \langle\!|Q|\!\rangle \wedge \langle\!|Q'|\!\rangle\ [\varepsilon|\varepsilon']} \qquad \textsc{rPoss}\ \frac{\Phi \vdash CC :\ \mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon']}{\Phi \vdash CC :\ \Diamond\mathcal{P} \approx\!> \Diamond\mathcal{Q}\ [\varepsilon|\varepsilon']}$$

$$\textsc{rSOF}\ \frac{\begin{array}{c}LocEq_\delta(\Phi,\Theta) \vdash_M \lfloor\!\lfloor C \rfloor\!\rfloor : locEq_\delta(P \rightsquigarrow Q\ [\varepsilon]) \qquad \models bnd(N)|bnd(N) \text{ frm } \mathcal{N} \qquad \mathcal{N} \Rightarrow \Box\mathcal{N} \\ N \neq M \qquad N \in \Theta \qquad \forall m \in \Phi.\ mdl(m) \not\preceq N \qquad \delta = (+L \in (\Phi,\Theta), L \neq M.\ bnd(L)) \qquad C \text{ is let-free}\end{array}}{LocEq_\delta(\Phi),\ LocEq_\delta(\Theta)\ \circledcirc\ \mathcal{N} \vdash_M \lfloor\!\lfloor C \rfloor\!\rfloor :\ locEq_\delta(P \rightsquigarrow Q\ [\varepsilon])\ \circledcirc\ \mathcal{N}}$$

$$\textsc{rFrame}\ \frac{\Phi \vdash CC :\ \mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon'] \qquad \mathcal{P} \models \eta|\eta' \text{ frm } \mathcal{R} \qquad \mathcal{P} \wedge \mathcal{R} \Rightarrow \langle\!|\eta \cdot\!/.\ \varepsilon|\!\rangle \wedge \langle\!|\eta' \cdot\!/.\ \varepsilon'|\!\rangle}{\Phi \vdash CC :\ \mathcal{P} \wedge \mathcal{R} \approx\!> \mathcal{Q} \wedge \mathcal{R}\ [\varepsilon|\varepsilon']}$$

$$\textsc{rConseq}\ \frac{\Phi \vdash CC :\ \mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon'] \qquad \mathcal{R} \Rightarrow \mathcal{P} \qquad \mathcal{Q} \Rightarrow \mathcal{S} \qquad \mathcal{P} \models (\varepsilon|\varepsilon') \leq (\eta|\eta')}{\Phi \vdash CC :\ \mathcal{R} \approx\!> \mathcal{S}\ [\eta|\eta']}$$

$$\textsc{rDisj}\ \frac{\Phi \vdash CC :\ \mathcal{P}_0 \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon'] \qquad \Phi \vdash CC :\ \mathcal{P}_1 \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon']}{\Phi \vdash CC :\ \mathcal{P}_0 \vee \mathcal{P}_1 \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon']}$$

$$\textsc{rConj}\ \frac{\Phi \vdash CC :\ \mathcal{P} \approx\!> \mathcal{Q}_0\ [\varepsilon|\varepsilon'] \qquad \Phi \vdash CC :\ \mathcal{P} \approx\!> \mathcal{Q}_1\ [\varepsilon|\varepsilon']}{\Phi \vdash CC :\ \mathcal{P} \approx\!> \mathcal{Q}_0 \wedge \mathcal{Q}_1\ [\varepsilon|\varepsilon']}$$

Fig. 30. Selected relational proof rules (for others see Appendix Figure 38). The typing context $\Gamma|\Gamma'$ is unchanged thoughout, so omitted. The current module is omitted in rules where it is the same in all the judgments and unconstrained.

8.2 Relational Proof Rules and Derivation of rMLink

Selected proof rules are in Figure 30. For relational judgments, the validity conditions (Definition 7.10) have been carefully formulated to leverage the unary ones (Definition 5.10). This obviates the need for rules like CtxIntro at the relational level. Rule rCall, for aligned calls using a relational spec, relies on unary premises to enforce the requisite encapsulation conditions. The relational rules for bi-if and bi-while have separator conditions to enforce encapsulation, taken straight from their unary rules (e.g., If in Figure 23). The relational rules for bi-while and sequence include an immunity condition for framing of their effects, again taken straight from the unary rules.

The linking rule, rLink, relates a client command C to itself using relations that imply its executions can be aligned lockstep. It can be instantiated with local equivalence specs but also with more general specs that include hidden invariants and coupling on encapsulated state. To allow this generality in a sound way, rule rLink uses the following notion.

Definition 8.5 (Covariant Spec Implication $\Rrightarrow$). Define $(\mathcal{R}_0 \approx\!> \mathcal{S}_0\ [\varepsilon_0|\varepsilon_0']) \Rrightarrow (\mathcal{R}_1 \approx\!> \mathcal{S}_1\ [\varepsilon_1|\varepsilon_1'])$ iff $\mathcal{R}_0 \Rightarrow \mathcal{R}_1$ and $\mathcal{S}_0 \Rightarrow \mathcal{S}_1$ are valid and the effects are the same: $\varepsilon_0 = \varepsilon_1$ and $\varepsilon_0' = \varepsilon_1'$. For contexts Φ and Ψ, define $\Phi \Rrightarrow \Psi$ to mean they have the same methods and $\Rrightarrow$ holds for the relational spec of each method.

For example, we have $locEq_\delta(spec) \mathbin{\&} \mathcal{M} \Rrightarrow locEq_\delta(spec)$ for any $\delta, spec, \mathcal{M}$.

In rLink, side conditions constrain module imports, exactly as in unary Link, as part of the enforcement of encapsulation. As with Link, some of the conditions merely express module structure. The soundness proof for rLink goes by induction on biprogram traces, similar to the soundness proof for unary Link; the relational hypothesis can be used, because the relevant context calls are aligned (see Appendices B.10 and D.10).

Rule rEmb lifts unary judgments to a relational one. It applies to arbitrary commands. For example, it can be applied to the *sumpub* program of Equation (4), to prove the judgment about (*sumpub*|*sumpub*) by lifting a unary spec as described in Section 4.5. It is also needed to obtain relational judgments about assignments, and it enables the use of unary specs in one-sided method calls.

For allocation, there needs to be a way to indicate when a pair of allocations are meant to be aligned; this is the purpose of rAlloc. Using rConj, rEmb, the unary rule Alloc, and the frame rules, one can add postconditions like $\mathbb{A}\{x\}\text{`}f$ and freshness of x. (Detailed derivations for freshness can be found in RLIII (Section 7.1)). Like rCall, rule rAlloc does not have the minimal hypothesis context but rather allows an arbitrary one; this is needed, because we do not have context introduction rules at the relational level. To enforce encapsulation, rAlloc has a side condition that simply says neither x nor alloc occur in the boundaries of any models other than the current one.

Rule rLocEq has a side condition about the unary judgment's frame condition: the writes must be subsumed by the reads (subeffect judgment $P \models w2r(\varepsilon) \leq rds(\varepsilon)$). This ensures that the precondition of the relational conclusion has agreement for writable locations. The requirement that C is let-free is needed in accord with Lemma 8.9.

Example 8.6 (How Framing is Used with rLocEq). Just as the unary axioms for assignments are "small" in the sense that they only describe the locations relevant to the command's behavior, we are interested in program equivalence described in terms of the relevant locations. As an example, without methods, consider this valid judgment (omitting the module, which is irrelevant):

$$\vdash (x := y.f; z := w) : y \neq 0 \rightsquigarrow true[\varepsilon],$$

where $\varepsilon \mathrel{\hat{=}} \mathsf{wr}\, x, z, \mathsf{rd}\, w, y, y.f$. It should entail this relational one:

$$\vdash [\![x := y.f; z := w]\!] : \mathbb{B}(y \neq 0) \wedge \mathbb{A}(y, w, \{y\}\text{`}f) \approx\!> \mathbb{B}\mathsf{true} \wedge \mathbb{A}(x, z)[\varepsilon].$$

Desugared, the precondition agreement is $\mathbb{A}y \wedge \mathbb{A}w \wedge \mathbb{A}\{y\}\text{`}f$. The precondition only requires agreement on locations that are read. The postcondition tells about the variables that are written. In fact w and y are unchanged, and we can strengthen the postcondition to

$$\vdash [\![x := y.f; z := w]\!] : \mathbb{B}(y \neq 0) \wedge \mathbb{A}(y, w, \{y\}\text{`}f) \approx\!> \mathbb{B}\mathsf{true} \wedge \mathbb{A}(x, z, y, w)[\varepsilon],$$

using the rFrame rule, because $\mathbb{A}(y, w)$ is separate from the writes. Rule rConseq allows us to strengthen the precondition by adding the agreements $\mathbb{A}(u, \{y\}\text{`}g)$:

$$\vdash [\![x := y.f; z := w]\!] : \mathbb{B}(y \neq 0) \wedge \mathbb{A}(y, w, \{y\}\text{`}f, u, \{y\}\text{`}g) \approx\!> \mathbb{B}\mathsf{true} \wedge \mathbb{A}(x, z, y, w)[\varepsilon].$$

Now rule rFrame allows us to carry these agreements over the command, because the locations u and $y.g$ are separate from the write effects:

$$\vdash [\![x := y.f; z := w]\!] : \mathbb{B}(y \neq 0) \wedge \mathbb{A}(y, w, \{y\}\text{`}f, u, \{y\}\text{`}g) \approx\!> \mathbb{B}\mathsf{true} \wedge \mathbb{A}(x, z, y, w, u, \{y\}\text{`}g)[\varepsilon].$$

$$\textsc{rMLink}\ \frac{\begin{array}{c}\Phi \vdash_\bullet C : P \rightsquigarrow Q\ [\varepsilon] \qquad \Phi \mathbin{\triangle} \mathcal{M} \vdash_M (B|B') : locEq_\delta(\Phi)(m) \mathbin{\triangle} \mathcal{M} \qquad \delta = bnd(M) \\ \Phi \mathbin{\triangle} \overleftarrow{\mathcal{M}} \vdash_M B : \Phi(m) \mathbin{\triangle} \overleftarrow{\mathcal{M}} \qquad \Phi \mathbin{\triangle} \overrightarrow{\mathcal{M}} \vdash_M B' : \Phi(m) \mathbin{\triangle} \overrightarrow{\mathcal{M}} \qquad M = mdl(m) \qquad P \models w2r(\varepsilon) \le rds(\varepsilon) \\ \models \delta|\delta\ \mathsf{frm}\ \mathcal{M} \qquad \mathcal{M} \Rightarrow \Box\mathcal{M} \qquad C \text{ is let-free} \qquad pre(locEq_\delta(P \rightsquigarrow Q\ [\varepsilon])) \Rightarrow \mathcal{M}\end{array}}{\vdash_\bullet (\mathsf{let}\ m = B\ \mathsf{in}\ C \mid \mathsf{let}\ m = B'\ \mathsf{in}\ C) : locEq_\delta(P \rightsquigarrow Q\ [\varepsilon])}$$

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\Phi \vdash_\bullet C : P \rightsquigarrow Q\ [\varepsilon]}{LocEq_\delta(\Phi) \vdash_\bullet [\![C]\!] : locEq_\delta(P \rightsquigarrow Q\ [\varepsilon])}\ \textsc{rLocEq} \qquad \Psi \vdash_M (B|B') : locEq_\delta(\Phi(m)) \mathbin{\triangle} \mathcal{M}}{\Psi \vdash_\bullet [\![C]\!] : locEq_\delta(P \rightsquigarrow Q\ [\varepsilon]) \mathbin{\triangle} \mathcal{M}}\ \textsc{rSOF} \quad \vdots}{\vdash_\bullet \mathsf{let}\ m = (B|B')\ \mathsf{in}\ [\![C]\!] : locEq_\delta(P \rightsquigarrow Q\ [\varepsilon]) \mathbin{\triangle} \mathcal{M}}\ \textsc{rLink}}{\vdash_\bullet \mathsf{let}\ m = (B|B')\ \mathsf{in}\ [\![C]\!] : locEq_\delta(P \rightsquigarrow Q\ [\varepsilon])}\ \textsc{rConseq}}{\vdash_\bullet (\mathsf{let}\ m = B\ \mathsf{in}\ C \mid \mathsf{let}\ m = B'\ \mathsf{in}\ C) : locEq_\delta(P \rightsquigarrow Q\ [\varepsilon])}\ \textsc{rWeave}$$

Fig. 31. rMLink and its derivation, where Ψ abbreviates $LocEq_\delta(\Phi) \mathbin{\triangle} \mathcal{M}$, Φ specifies m, $\delta = bnd(M)$, and $M = mdl(m)$. See text for details.

In summary, the local equivalence spec expresses a program relation in terms of only the locations readable and writable by the command. Such equivalence can be extended to arbitrary other locations not touched by the command.

Rule rSOF follows the pattern of the unary SOF in its use of $\mathbin{\triangle}\mathcal{M}$ from Definition 4.7. It can only be instantiated with specs in standard form, so that $\mathbin{\triangle}\mathcal{M}$ is defined. It requires refperm monotonicity of the coupling, i.e., $\mathcal{N} \Rightarrow \Box\mathcal{N}$; more on this in Section 8.3.

Figure 31 presents the relational modular linking rule, rMLink, and its derivation. (Here specialized to a single method, i.e., $dom(\Phi) = \{m\}$, for clarity). The side conditions are $P \models w2r(\varepsilon) \le rds(\varepsilon)$ (for rLocEq); $\models \delta|\delta$ frm $\mathcal{M}$ and $\mathcal{M} \Rightarrow \Box\mathcal{M}$ (for rSOF); $dom(\Phi) = \{m\}$ (for rLink); and $pre(locEq_\delta(P \rightsquigarrow Q\ [\varepsilon])) \Rightarrow \mathcal{M}$ (for rConseq, to drop $\wedge\mathcal{M}$ from the precondition; of course $\wedge\mathcal{M}$ is also dropped from postcondition). For rWeave, we use the fact that $(\mathsf{let}\ m = B\ \mathsf{in}\ C \mid \mathsf{let}\ m = B'\ \mathsf{in}\ C) \mathrel{\hookrightarrow^*} \mathsf{let}\ m = (B|B')\ \mathsf{in}\ [\![C]\!]$. Vertical elipses in the derivation indicate that, in addition to the expected relational premise for B and B', unary premises are required: $\Phi \mathbin{\triangle} \overleftarrow{\mathcal{M}} \vdash_M B : \Phi(m) \mathbin{\triangle} \overleftarrow{\mathcal{M}}$ and $\Phi \mathbin{\triangle} \overrightarrow{\mathcal{M}} \vdash_M B' : \Phi(m) \mathbin{\triangle} \overrightarrow{\mathcal{M}}$. These are required by rLink, for technical reasons explained in its proof (Section D.10).

The implication $pre(locEq_\delta(P \rightsquigarrow Q\ [\varepsilon])) \Rightarrow \mathcal{M}$ refers to the precondition of local equivalence. Typically, the implication is valid, because P includes initial conditions that imply $\mathcal{M}$ just as in the case of unary modular linking and module invariant. This is the responsibility of the module developer, who defines $\mathcal{M}$, shows its framing by the boundary, and shows refperm monotonicity of $\mathcal{M}$.

Example 8.7 (Illustrating rMLink with SSSP). We instantiate M in the rule with PQ (Section 3) and Φ with the specs of PQ's public methods. Let δ be PQ's dynamic boundary rd *pool*, *pool*‘any, *pool*‘*rep*‘any. We instantiate client C with C_{sssp}, an implementation of Dijkstra's single-source shortest-paths algorithm acting on global variables *gph*, *src*, and *wts*. For simplicity, *gph* is a variable of type "mathematical graph," for which we use an API supporting usual operations. We assume the vertex set $V(gph)$ is an initial segment of naturals so the source vertex variable *src* has type int. Edges have positive integer weights. The integer array *wts*, of length $|V(gph)|$ and allocated by the client, is for the output: for every vertex $v \in V(gph)$, C_{sssp} computes in $wts[v]$ the weight of the shortest path from *src* to v.

The unary spec for C_{sssp} is $P \rightsquigarrow Q\ [\varepsilon]$ where $P \mathrel{\hat{=}} src \in V(gph) \wedge pool = \emptyset$; $Q \mathrel{\hat{=}}$ true; and $\varepsilon \mathrel{\hat{=}}$ rd *gph*, *src*, rw *wts*, *pool*, *pool*‘any, *pool*‘*rep*‘any, alloc. The trivial postcondition does not

specify functional behavior but the spec is still useful. The local equivalence spec $locEq_\delta(P \rightsquigarrow Q[\varepsilon])$ is $\mathcal{R} \approx\!> \Diamond\mathcal{S}\ [\varepsilon]$ where $\mathcal{R} \triangleq \mathbb{B}(src \in V(gph) \wedge pool = \varnothing \wedge s_{\text{alloc}} = \text{alloc}) \wedge \mathbb{A}(wts, gph, src)$; and $\mathcal{S} \triangleq \mathbb{A}(wts, (\text{alloc}\backslash(s_{\text{alloc}} \cup pool \cup pool\text{`}rep))\text{`any})$, eliding details about spec-only variables apart from s_{alloc}. Here s_{alloc} snapshots alloc so fresh locations are those in $\text{alloc}\backslash s_{\text{alloc}}$. This spec ensures agreement on fresh locations that are not in PQ's dynamic boundary.

The coupling $\mathcal{M}_{PQ}$ is $\forall q{:}\text{Pqueue} \in pool | q{:}\text{Pqueue} \in pool.\ \mathbb{A}q \Rightarrow \forall n \in q.rep | n \in q.rep.\ \mathbb{A}n \Rightarrow \ldots$, conjoined with the private invariants I and I' (eliding parts shown in Example 4.3). One side condition of rMLink is $pre(locEq_\delta(P \rightsquigarrow Q\ [\varepsilon])) \Rightarrow \mathcal{M}_{PQ}$, which is easy to show: expanding definitions, the antecedent includes $\mathbb{B}(pool = \varnothing)$, which implies the private invariants and the coupling relation. The subeffect $P \models w2r(\varepsilon) \leq rds(\varepsilon)$ is immediate from the definition of ε. The framing judgment, $\models \delta | \delta$ frm $\mathcal{M}_{PQ}$, is easily proved by SMT, as is refperm monotonicity of $\mathcal{M}_{PQ}$.

8.3 Refperm Monotonicity, Standard form, and Agreement Compatibility

For modular linking and most other purposes, we are concerned with specs in the standard form, i.e., either $\mathcal{R} \approx\!> \Diamond\mathcal{S}\ [\eta]$ or $\mathcal{R} \approx\!> \mathcal{S}\ [\eta]$ where $\mathcal{R}$ and $\mathcal{S}$ are $\Diamond$-free. In this section, we consider the rules that give rise to other forms, and related notions concerning formulas with $\Diamond$. It is possible to reformulate the logic to consider only standard form specs. We choose the present formulation, because some proof rules can be simpler and more orthogonal.

For reasoning about sequential composition one wants to combine judgments for specs $\mathcal{P} \approx\!> \Diamond\mathcal{Q}$ and $\mathcal{Q} \approx\!> \Diamond\mathcal{R}$ into a judgment for $\mathcal{P} \approx\!> \Diamond\mathcal{R}$ (omitting frame for clarity). It is easy to derive a rule for specs of this form, from the more basic rule for sequence together rules rPoss and rConseq. From $\mathcal{Q} \approx\!> \Diamond\mathcal{R}$, we get $\Diamond\mathcal{Q} \approx\!> \Diamond\Diamond\mathcal{R}$ by rPoss. Then, we get $\Diamond\mathcal{Q} \approx\!> \Diamond\mathcal{R}$ by rConseq, because $\Diamond\Diamond\mathcal{R} \iff \Diamond\mathcal{R}$ is valid. From $\mathcal{P} \approx\!> \Diamond\mathcal{Q}$ and $\Diamond\mathcal{Q} \approx\!> \Diamond\mathcal{R}$ we get $\mathcal{P} \approx\!> \Diamond\mathcal{R}$ by the sequence rule.

Similarly, one can derive a relational rule for loops, with premises in standard form and relational invariant $\mathcal{Q}$ that is $\Diamond$-free. In accord with the loop rule sketched as Equation (16), we elide frame conditions, context, and side conditions for immunity and encapsulation. The derived rule looks like this:

$$\frac{\begin{array}{c}\vdash CC : \mathcal{Q} \wedge \neg\mathcal{P} \wedge \neg\mathcal{P}' \wedge \{\!|E|\!\} \wedge [\![E']\!\} \approx\!> \Diamond\mathcal{Q} \qquad \vdash (\overleftarrow{CC}|\text{skip}) : \mathcal{Q} \wedge \mathcal{P} \wedge \{\!|E|\!\} \approx\!> \Diamond\mathcal{Q} \\ \vdash (\text{skip}|\overrightarrow{CC}) : \mathcal{Q} \wedge \mathcal{P}' \wedge [\![E']\!\} \approx\!> \Diamond\mathcal{Q} \qquad \mathcal{Q} \Rightarrow E \doteq E' \vee (\mathcal{P} \wedge \{\!|E|\!\}) \vee (\mathcal{P}' \wedge [\![E']\!\})\end{array}}{\vdash \text{while } E|E' \cdot \mathcal{P}|\mathcal{P}' \text{ do } CC : \mathcal{Q} \approx\!> \Diamond(\mathcal{Q} \wedge \{\!|\neg E|\!\} \wedge [\![\neg E']\!\})}. \tag{32}$$

Given the premises, three applications of rPoss yields $CC : \Diamond(\mathcal{Q} \wedge \neg\mathcal{P} \wedge \neg\mathcal{P}' \wedge \{\!|E|\!\} \wedge [\![E']\!\}) \approx\!> \Diamond\Diamond\mathcal{Q}$, $(\overleftarrow{CC}|\text{skip}) : \Diamond(\mathcal{Q} \wedge \mathcal{P} \wedge \{\!|E|\!\}) \approx\!> \Diamond\Diamond\mathcal{Q}$, and $(\text{skip}|\overrightarrow{CC}) : \Diamond(\mathcal{Q} \wedge \mathcal{P}' \wedge [\![E']\!\}) \approx\!> \Diamond\Diamond\mathcal{Q}$. But $\Diamond\Diamond\mathcal{Q}$ is equivalent to $\Diamond\mathcal{Q}$. Furthermore, $\{\!|E|\!\}$ and $[\![E']\!\}$ are agreement-free and thus refperm independent. Also $\mathcal{P}, \mathcal{P}'$ are refperm independent, because they are agreement free by the wellformedness condition mentioned at the end of Section 3.1. So, using property (31), the precondition of the second judgment, $\Diamond(\mathcal{Q} \wedge \mathcal{P} \wedge \{\!|E|\!\})$ is equivalent to one where $\Diamond$ is applied only to $\mathcal{Q}$, i.e., $\Diamond\mathcal{Q} \wedge \mathcal{P} \wedge \{\!|E|\!\}$. Similarly for the other two preconditions. So by rConseq, we get

- $CC : \Diamond\mathcal{Q} \wedge \neg\mathcal{P} \wedge \neg\mathcal{P}' \wedge \{\!|E|\!\} \wedge [\![E']\!\} \approx\!> \Diamond\mathcal{Q}$,
- $(\overleftarrow{CC}|\text{skip}) : \Diamond\mathcal{Q} \wedge \mathcal{P} \wedge \{\!|E|\!\} \approx\!> \Diamond\mathcal{Q}$,
- $(\text{skip}|\overrightarrow{CC}) : \Diamond\mathcal{Q} \wedge \mathcal{P}' \wedge [\![E']\!\} \approx\!> \Diamond\mathcal{Q}$.

With these, we instantiate the rule (16) with $\Diamond\mathcal{Q}$ for $\mathcal{Q}$, which yields $\text{while } E|E' \cdot \mathcal{P}|\mathcal{P}' \text{ do } CC : \Diamond\mathcal{Q} \approx\!> \Diamond\mathcal{Q} \wedge \{\!|\neg E|\!\} \wedge [\![\neg E']\!\}$. Finally, the implication $\mathcal{Q} \Rightarrow \Diamond\mathcal{Q}$ is valid, and we can distribute refperm independent formulas under $\Diamond$; so using rConseq, we obtain the conclusion of Equation (32).

For a bi-while with false alignment guards, there is a derived rule with a single premise $\vdash CC : Q \wedge (\!|E|\!) \wedge (\!|E'|\!) \approx\!> \Diamond Q$. It can be derived, using rule rEmpPre.

Refperm monotonicity. Given a judgment $\Phi \vdash CC : \mathcal{P} \approx\!> \Diamond Q\ [\varepsilon|\varepsilon']$, rule rFrame yields $\Phi \vdash CC : \mathcal{P} \wedge \mathcal{R} \approx\!> \Diamond Q \wedge \mathcal{R}\ [\varepsilon|\varepsilon']$, which is not in the standard form. But suppose $\mathcal{R}$ is refperm monotonic, i.e., $\mathcal{R} \Rightarrow \Box\mathcal{R}$ is valid. Then by Equation (30), we have $\Diamond Q \wedge \mathcal{R} \Rightarrow \Diamond(Q \wedge \mathcal{R})$. So using rConseq we get this derived frame rule:

$$\frac{\Phi \vdash CC : \mathcal{P} \approx\!> \Diamond Q\ [\varepsilon|\varepsilon'] \qquad \mathcal{P} \models \eta|\eta' \text{ frm } \mathcal{R} \qquad \mathcal{P} \wedge \mathcal{R} \Rightarrow (\!|\eta \cdot/. \varepsilon|\!) \wedge (\!|\eta' \cdot/. \varepsilon'|\!) \qquad \mathcal{R} \Rightarrow \Box\mathcal{R}}{\Phi \vdash CC : \mathcal{P} \wedge \mathcal{R} \approx\!> \Diamond(Q \wedge \mathcal{R})\ [\varepsilon|\varepsilon']}.$$

Refperm monotonicity is also a side condition for the coupling relation in rule rSOF. In that rule, moving the coupling relation under $\Diamond$ is done by the $\textcircled{\Diamond}$ operation (Definition 4.7).

Agreement formulas are refperm monotonic, as are refperm independent formulas. But negation does not preserve refperm monotonicity, and in particular a formula of the form $\mathbb{A}x \Rightarrow \mathcal{R}$ is not refperm monotonic even if $\mathcal{R}$ is. Such implications are used in our example couplings. In particular, implication is used in the following idiomatic pattern:

$$G \doteq G' \wedge (\forall x{:}K|x{:}K.\ (\!|x \in G|\!) \wedge (\!|x \in G'|\!) \wedge \mathbb{A}x \Rightarrow \mathcal{R}). \tag{33}$$

The second conjunct can be written in sugared form as $\forall x{:}K \in G|x{:}K \in G'.\ \mathbb{A}x \Rightarrow \mathcal{R}$.

Lemma 8.8 (Refperm Monotonicity). *(i) Any agreement formula is refperm monotonic and so is any refperm independent formula. (ii) Refperm monotonicity is preserved by conjunction, disjunction, and quantification. (iii) Any formula of the form (33), with $\mathcal{R}$ refperm monotonic, is refperm monotonic.*

The coupling $\mathcal{M}_{uf}$ in Section 4.6 is refperm monotonic. The embedded invariants $(\!|I_{qf}|\!)$ and $(\!|I_{qu}|\!)$ are refperm monotonic, by (i) in the lemma, as is the consequent $eqPartition((\!|u.part|\!), (\!|u.part|\!))$ in the relation (19). So refperm monotonicity of $\mathcal{M}_{uf}$ follows using (ii) and (iii).

The coupling $\mathcal{M}_{PQ}$ in Example 4.3 is refperm monotonic. To see why, first note that Equation (33) is equivalent to $G/K \doteq G'/K \wedge (\forall x{:}K \in G|x{:}K \in G'.\ \mathbb{A}x \Rightarrow \mathcal{R})$, because a quantified variable of type K ranges over allocated (non-null) references of type K. So inside the quantification, $x \in G$ is equivalent to $x \in G/K$. The relevant subformula of $\mathcal{M}_{PQ}$ is $q.rep/\text{Pnode} \doteq q.rep/\text{Pnode}$. Now, we distill the following pattern from $\mathcal{M}_{PQ}$, in which we assume f : rgn and assume both Q and $\mathcal{R}$ are refperm monotonic:

$$G \doteq G \wedge (\forall x{:}K \in G|x{:}K \in G.\ \mathbb{A}x \Rightarrow Q \wedge \{x\}`f \doteq \{x\}`f \wedge (\forall y{:}L \in \{x\}`f|y{:}L \in \{x\}`f.\ \mathbb{A}y \Rightarrow \mathcal{R})).$$

By (iii) in the lemma, the subformula $\{x\}`f \doteq \{x\}`f \wedge (\forall y{:}L \in x.f|y{:}L \in x.f.\ \mathbb{A}y \Rightarrow \mathcal{R})$ is refperm monotonic. Then by (ii), we extend that to the conjunction with Q. Then by (iii), the displayed formula is refperm monotonic. Note that this relies on agreement of the region values, $\{x\}`f \doteq \{x\}`f$, not pairwise agreement $\mathbb{A}\{x\}`f$ on field values.

This discussion provides guidelines for writing specs, but checking refperm monotonicity can be automated. Validity of $\mathcal{R} \Rightarrow \Box\mathcal{R}$ only involves universal quantification. Unfolding semantic definitions, it says: for all $\pi, \rho, \sigma, \sigma'$, if $\sigma|\sigma' \models_\pi \mathcal{R}$ and $\rho \supseteq \pi$ then $\sigma|\sigma' \models_\rho \mathcal{R}$. A straightforward encoding of this in our prototype suffices to show refperm monotonicity of the example couplings.

Agreement compatibility. The last rule for which $\Diamond$ is an issue is rConj. With premises of the form $\mathcal{P} \approx\!> \Diamond Q_0$ and $\mathcal{P} \approx\!> \Diamond Q_1$ it yields $\mathcal{P} \approx\!> \Diamond Q_0 \wedge \Diamond Q_1$. To obtain the standard form $\mathcal{P} \approx\!> \Diamond(Q_0 \wedge Q_1)$ one can use rConseq but only if Q_0 and Q_1 are ***agreement compatible***, which means this implication is valid:

$$\Diamond Q_0 \wedge \Diamond Q_1 \Rightarrow \Diamond(Q_0 \wedge Q_1). \tag{34}$$

An easy case is where Q_0 or Q_1 is refperm independent, in which case agreement compatibility holds by Equation (31). Formulas that depend on the refperm involve agreements, and for these, we do not have an easy characterization of agreement compatibility.

In the prototype, $\diamond$ is not explicit in specs. A current refperm is witnessed in ghost state, so even when using conjunctive splitting, we effectively get $\diamond(Q_0 \wedge Q_1)$ as desired. So agreement compatibility is not an issue in the tool. Morever our case studies show that agreement compatibility is achievable in practical examples where it is needed. Please note that nontrivial formulas of the form (34) are not amenable to validity checking by SMT, owing to the existential quantifier that underlies $\diamond$ in the consequent.[39]

We end this section with some examples regarding agreement compatibility. But it is not needed later so it is safe to skip now to Section 8.4.

As a first example, consider the agreements $\mathbb{A}(G/\mathsf{List})\text{`}head$ and $\mathbb{A}(G/\mathsf{Cell})\text{`}val$, where class List has field $head$: Node and class Cell has field val : int. The truth value of $\mathbb{A}(G/\mathsf{List})\text{`}head$ depends only on references of type List and Node. The truth value of $\mathbb{A}(G/\mathsf{Cell})\text{`}val$ depends only on references of type Cell. Refperms respect types, so extensions of a refperm to witness $\diamond\mathbb{A}(G/\mathsf{List})\text{`}head$ and $\diamond\mathbb{A}(G/\mathsf{Cell})\text{`}val$ can be combined to witness $\diamond(\mathbb{A}(G/\mathsf{List})\text{`}head \wedge \mathbb{A}(G/\mathsf{Cell})\text{`}val)$. Such considerations also apply in a case like $\mathbb{B}\mathsf{type}(G, \mathsf{List}) \wedge \mathbb{A}G\text{`}head$ and $\mathbb{B}\mathsf{type}(H, \mathsf{Cell}) \wedge \mathbb{A}H\text{`}val$.

Agreement compatibility of Q_0 and Q_1 may fail even if both formulas are Q and $\mathcal{R}$ are refperm monotonic. For example, the formula $\diamond(x \doteq y) \wedge \diamond(x \doteq z \wedge \llparenthesis z \neq y \rrparenthesis)$ is satisfiable but $\diamond(x \doteq y \wedge x \doteq z \wedge \llparenthesis z \neq y \rrparenthesis)$ is not. This example may give the impression that disequalities are the culprit but they are not. Consider these two formulas: $\diamond(x \doteq x' \wedge y \doteq y')$ and $\diamond(x \doteq y' \wedge y \doteq x')$ (for distinct variables x, x', y, y'). Both are satisfiable. In fact their combination, $\diamond(x \doteq x' \wedge y \doteq y' \wedge x \doteq y' \wedge y \doteq x')$, is also satisfiable: it can hold when $\{\!|x = y|\!\} \wedge \llparenthesis x' = y' \rrparenthesis$. But the agreement-compatibility implication is not valid. Consider σ, σ', π where x, y, x', y' have four distinct values, none of which are in the domain or range of π. Then both $\diamond(x \doteq x' \wedge y \doteq y')$ and $\diamond(x \doteq y' \wedge y \doteq x')$ are true but $\diamond(x \doteq x' \wedge y \doteq y' \wedge x \doteq y' \wedge y \doteq x')$ is false.

One might guess $\mathbb{A}G\text{`}f$ is agreement compatible with $\mathbb{A}H\text{`}g$ where f, g are distinct field names. But consider $\mathbb{A}\{x\}\text{`}f$ and $\mathbb{A}\{x\}\text{`}g$ for distinct fields f, g of some reference type. Suppose $\sigma|\sigma' \models_\pi x \doteq x$, so $\pi(\sigma(x)) = \sigma'(x)$. Suppose $\sigma(x.f)$ and $\sigma(x.g)$ are non-null values not in $dom(\pi)$, and likewise $\sigma'(x.f)$ and $\sigma'(x.g)$ are non-null values not in $rng(\pi)$. Then, we have $\sigma|\sigma' \models_\pi \diamond\mathbb{A}\{x\}\text{`}f \wedge \diamond\mathbb{A}\{x\}\text{`}g$, because π can be extended to link $\sigma(x.f)$ with $\sigma'(x.f)$ and *mut. mut.* for g. However, if $\sigma(x.f) = \sigma(x.g)$ and $\sigma'(x.f) \neq \sigma'(x.g)$ then there is no single extension of π that satisfies $\mathbb{A}\{x\}\text{`}f \wedge \mathbb{A}\{x\}\text{`}g$.

Region disjointness $G\#H$ does not entail agreement compatiblity of $\mathbb{A}G\text{`}f$ with $\mathbb{A}H\text{`}f$. Consider $\mathbb{A}\{x\}\text{`}f$ and $\mathbb{A}\{y\}\text{`}g$. Suppose $\sigma|\sigma' \models_\pi x \doteq x \wedge y \doteq y \wedge \mathbb{B}(x \neq y)$. Similar to the preceding example, if $\sigma(x.f) = \sigma(y.g)$ and $\sigma'(x.f) \neq \sigma'(y.g)$ and none of the field values are in π, then we have $\sigma|\sigma' \models_\pi \diamond\mathbb{A}\{x\}\text{`}f \wedge \diamond\mathbb{A}\{y\}\text{`}g$ but again there is no extension of π that satisfies $\mathbb{A}\{x\}\text{`}f \wedge \mathbb{A}\{y\}\text{`}g$.

8.4 Lockstep Alignment Lemma

The lockstep alignment lemma brings together the semantics of encapsulation in the unary logic (Definition 5.10), in which dependency is expressed in terms of two runs under a single unary context model, with the biprogram semantics, which involves two possibly different unary context models as needed for linking with two module implementations. The lemma says that, from states that agree on what may be read, a fully-aligned biprogram remains fully aligned through its

[39]For the record, earlier versions of this article had a slightly different RSOF, with agreement compatibility as a side condition for the coupling rather than refperm monotonicity (arXiv:1910.14560v3).

execution, and maintains agreements sufficient to establish the postcondition of local equivalence—for any of its traces that satisfy the r-safe and respect conditions of Definition 5.10. In light of trace projection (Lemma 7.8), it says a pair of unary executions can be aligned lockstep, with strong agreements asserted at each aligned pair of configurations. The result does not rely on validity of a judgment—rather, we use this result to prove soundness of rules rLocEq, rSOF, and rLink.

A number of subtleties in the unary semantics of encapsulation, in the biprogram semantics, and in the definition of *locEq* are all motivated by difficulties in obtaining a result that is sufficiently strong to support the soundness proofs for the three rules from which the modular relational linking rule is derived (rLocEq, rSOF, and rLink).

Lemma 8.9 (Lockstep Alignment). *Suppose*

(i) $\Phi \Rrightarrow LocEq_\delta(\Psi)$ and φ is a Φ-model, where $\delta = (+N \in \Psi, N \neq M.\ bnd(N))$,
(ii) $\sigma|\sigma' \models_\pi pre(locEq_\delta(P \rightsquigarrow Q\ [\varepsilon]))$,
(iii) T is a trace $\langle \|C\|,\ \sigma|\sigma',\ \_|\_\rangle \overset{\varphi}{\Longrightarrow}^* \langle BB,\ \tau|\tau',\ \mu|\mu'\rangle$ and C is let-free,
(iv) Let U, V be the projections of T. Then U (respectively, V) is r-safe for $(\Phi_0, \varepsilon, \sigma)$ (respectively, for $(\Phi_1, \varepsilon, \sigma')$) and respects $(\Phi_0, M, \varphi_0, \varepsilon, \sigma)$ (respectively, $(\Phi_1, M, \varphi_1, \varepsilon, \sigma')$).

Then there are B, ρ, with

(v) $BB \equiv \|B\|$, $\rho \supseteq \pi$, and $\mu = \mu'$,
(vi) $Lagree(\tau, \tau', \rho, (freshL(\sigma, \tau) \cup rlocs(\sigma, \varepsilon) \cup wrttn(\sigma, \tau))\backslash rlocs(\tau, \delta^{\oplus}))$, and
(vii) $Lagree(\tau', \tau, \rho^{-1}, (freshL(\sigma', \tau') \cup rlocs(\sigma', \varepsilon) \cup wrttn(\sigma', \tau'))\backslash rlocs(\tau', \delta^{\oplus}))$.

In other words, the Lemma says that if we have fully aligned code, unary encapsulation (iv), initial agreement (ii), and relational specs that imply the local equivalence spec (but may be strengthened to include hidden invariants and coupling) (i), then the code remains fully aligned at every step, and agreements outside encapsulated state are preserved. Condition (v) can be strengthened to say μ and μ' are empty, which holds owing to the assumption that C is let-free. We keep this formulation, because it suffices and shows what we expect for the extensions discussed in Section 8.5.

The lemma is proved by induction on steps, maintaining (v)–(vii), using several technical lemmas for preservation of agreement (in Appendix Section D.2).

Lemma 8.9 resembles Lemma 5.11 but has significant differences. Lemma 8.9 is for client code outside boundaries, in a setting where there are different implementations of methods. Lemma 5.11 is for code potentially inside boundaries, but relating two runs of exactly the same program. In the proofs of both results, r-safety helps ensure that the small-step dependency embodied by r-respect implies an end-to-end dependency condition.

8.5 Nested Linking

The unary and relational linking rules allow simultaneous linking of multiple modules, for example linking *MST* with the PQ and Graph modules. In RLII (Section 9), a modular linking rule is derived for simultaneous linking of two modules with mutually recursive methods, each respecting the other's boundary. That can be done with both the unary and relational rules in this article: the judgments for correctness of the bodies are extended with the other module's invariant or coupling (using SOF or rSOF) and then linked (using Link or rLink). In RLII and the unary logic in this article, it is also possible for linking to be nested (shown by examples in Sections 2.4 and 8.4 of RLII). However, there is a limitation of the relational rules with nested use of bi-let.

To set the stage, we carry out the derivation of modular linking as in Figure 24 but with a second module in context, to which we then apply modular linking. Methods of Φ may be used in both the client C and the implementation B. The implementation of Φ has its own internal state with invariant J:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Phi, \Theta \vdash_\bullet C : P \rightsquigarrow Q\ [\varepsilon]}{\Phi, (\Theta \between I) \vdash_\bullet C : (P \rightsquigarrow Q\ [\varepsilon]) \between I} \qquad \Phi, (\Theta \between I) \vdash_M B : \Theta(m) \between I}{\Phi \vdash_\bullet \mathsf{let}\ m = B\ \mathsf{in}\ C : (P \rightsquigarrow Q\ [\varepsilon]) \between I}}{\Phi \between J \vdash_\bullet \mathsf{let}\ m = B\ \mathsf{in}\ C : (P \rightsquigarrow Q\ [\varepsilon]) \between I \between J} \qquad \Phi \between J \vdash_N D : \Phi(n) \between J}{\vdash_\bullet \mathsf{let}\ n = D\ \mathsf{in}\ \mathsf{let}\ m = B\ \mathsf{in}\ C : (P \rightsquigarrow Q\ [\varepsilon]) \between I \between J}$$

We would like the relational analog of this derivation, so that with coupling $\mathcal{M}$ for module M and coupling $\mathcal{N}$ for N one could obtain the judgment

$$\vdash_\bullet \mathsf{let}\ n = (D|D')\ \mathsf{in}\ \mathsf{let}\ m = (B|B')\ \mathsf{in}\ [\![C]\!] : locEq_\delta(P \rightsquigarrow Q\ [\varepsilon]) \between \mathcal{M} \between \mathcal{N}.$$

Following the pattern of the derivation above, one would like to apply RSOF for $\mathcal{N}$ to the judgment $LocEq_\delta(\Phi) \vdash_\bullet \mathsf{let}\ m = (B|B')\ \mathsf{in}\ [\![C]\!] : locEq_\delta(P \rightsquigarrow Q\ [\varepsilon]) \between \mathcal{M}$, where $\delta = bnd(M), bnd(N)$. However, the current RSOF and RLINK are only for fully aligned client code, and the "client" body $\mathsf{let}\ m = (B|B')\ \mathsf{in}\ [\![C]\!]$ of the outer let is not in that form. Soundness of RSOF hinges on the calls being sync'd—but in the program $\mathsf{let}\ m = (B|B')\ \mathsf{in}\ [\![C]\!]$, calls to n (the method of Φ) from B or B' are not sync'd, because $m()$ steps to $(B|B')$, which has no sync'd calls. The restriction of bi-let to separate unary commands simplifies the technical development considerably. But, we would like to generalize the bi-let form to allow $\mathsf{let}\ m = BB\ \mathsf{in}\ CC$ where BB is sufficiently woven that all its calls are sync'd, and CC is a nest of such bi-lets enclosing a fully aligned client. This requires Lemma 8.9 to be generalized to account for such biprogram computations. The Lemma relies on agreements derived from unary Encap, but this is no longer sufficient to handle computations with sub-computations that are not fully aligned. The premises of RSOF and RLINK entail that such computations can make sync'd calls, but this fact is not retained in the semantics of relational judgments. Details of our solution are beyond the scope of this article.

8.6 Unconditional Equivalence Transformations

An important feature of relational logic that is introduced in Banerjee et al. [11] (long version) is unconditional rewrites. These are correctness-preserving transformations of control structure in commands that enable the use of the bi-if and bi-while forms for programs with differing control structure. An example is the equivalence $\mathsf{while}\ E\ \mathsf{do}\ C \cong \mathsf{while}\ E\ \mathsf{do}\ (C; \mathsf{while}\ E \wedge E0\ \mathsf{do}\ C)$. Banerjee et al. use this and another loop unrolling equivalence to prove correctness of a loop tiling optimization. In that proof the loop iterations are aligned lockstep, i.e., rule RWHILE and a bi-while with false alignment guards.

In the cited work, it suffices to define $\cong$ as a safety-preserving trace equivalence. These sorts of transformations do not alter the series of states reached and which atomic commands are executed. From the same initial state and environment, the computations proceed almost in step-by-step correspondence, the exceptions being different manipulation of the control state in some cases, which leaves the (data) state and method environment unchanged. As a result, correctness is preserved in the sense that if $C \cong D$ then $\Phi \models C : P \rightsquigarrow Q\ [\varepsilon]$ implies $\Phi \models D : P \rightsquigarrow Q\ [\varepsilon]$. Moreover, $\Phi \models (C|C') : \mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon']$ implies $\Phi \models (D|C') : \mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon']$ (and the same on the right side). However, to cater for the stronger conditions of valid unary and relational judgments in the present work (Definitions 5.10 and 7.10), a stronger notion is needed, because those conditions refer to the control.

As an example, suppose we have a valid correctness judgment $\Phi \vdash_M \mathsf{while}\ E\ \mathsf{do}\ C : P \rightsquigarrow Q\ [\varepsilon]$ and consider the form $\mathsf{while}\ E\ \mathsf{do}\ (C; \mathsf{while}\ E \wedge E0\ \mathsf{do}\ C)$. If $E0$ reads some variable that is encapsulated by a module, different from M, in Φ, then it may violate the Encap condition of Definition 5.10 and invalidate the judgment $\Phi \vdash_M \mathsf{while}\ E\ \mathsf{do}\ (C; \mathsf{while}\ E \wedge E0\ \mathsf{do}\ C) : P \rightsquigarrow Q\ [\varepsilon]$. For the equivalences considered here, which involve rearranging control structure, branch conditions turn

out to be the main complication. Details of our formalization of ≅ and its rules are beyond the scope of this article.

9 REMARKS ON CASE STUDIES

WhyRel is a proof-of-principle prototype relational verifier which we developed and used to investigate the applicability of the logic and its amenability to automation. The tool supports general relational verification and includes support for relational modular linking. It has been used to specify and verify a number of examples. This includes examples discussed in earlier sections: Kruskal's *MST* as client of two implementations of union-find; Dijkstra's shortest-path algorithm as client of two implementations of PQ; and the *tabulate* and *sumpub* examples. We have done other examples taken from recent literature on relational verification, including information flow, other relational properties, and equivalence for program transformations. A current version of the prototype and examples are available open source.[40] In addition to the following highlights and the documentation in the software distribution, further information is available in the thesis of Nikouei [75] (but note it describes a previous implementation of WhyRel).

The WhyRel prototype is based on the Why3 platform.[41] Why3 serves as an intermediate verification language to which WhyRel translates specs and programs. Why3 generates verification conditions for pre-post specs and programs in a first-order fragment of ML (WhyML) without shared references, and discharges those conditions by orchestrating calls to automated provers and proof assistants. Like Why3, WhyRel is "auto-active" [63], requiring some user interaction while leveraging automated provers especially SMT solvers. Our translation involves substantial encoding, because Why3 does not support shared mutable objects, dynamic frames, or hiding of invariants. In this section, we describe the encoding, the user interaction needed, and our experience with the case studies.

The language supported by WhyRel extends the language of Figure 5 and Section 3.2 with arrays, parameters/results, and mathematical data types (defined in Why3 theories). Module interfaces are separate from module implementations and class fields can have module scope. The spec language is like that of the article (with usual keywords requires, ensures, etc.), extended with "old" expressions, assertions, loop invariants, assumptions, and explicit ghost declarations. WhyRel effectively works with relational specs in standard form: the possibility modal ($\Diamond$) is not used and instead a ghost refperm is updated by the connect–with ghost operation described in Section 4.4.

WhyRel has three main capabilities: unary verification, relational verification, and relational verification with modular linking. The user provides module interfaces (class declarations, method specs, and boundaries which may be empty) and unary module implementations which can import Why3 theories providing mathematical types (like lists, graphs, and partitions used in our case studies). These theories can include lemmas, which get proved by Why3. The user can also state lemmas in our source language, e.g., useful consequences of public invariants. For relational verification, the user provides a module with biprograms, which we call a bimodule. Each bimodule relates two unary modules. WhyRel checks, for each bimethod in a bimodule, that its unary projections conform to the (unary) programs being related. This ensures the biprogram can be constructed by weaving those unary programs (Lemma 4.6). Thus, verification of the biprogram implies a relation between the unary programs, as per the weaving rule (13).

For relational modular linking of a client program and two versions of a module the client imports, WhyRel can generate the local equivalence specs for the module methods. The user can edit

[40] https://github.com/dnaumann/RelRL.
[41] why3.lri.fr.

the specs to add the chosen coupling relation, and use these in a bimodule for relating the module methods. WhyRel also generates the side conditions of rule rMLink, which include framing of invariants/coupling by the boundary and refperm monotonicity of the coupling.

The user provides specs and also loop invariants and loop frame conditions; for hiding, the user provides boundaries, private invariants, and coupling relations. Once WhyRel has translated the specs and programs/biprograms to WhyML, Why3 generates verification conditions. The user guides Why3 to prove these, by applying tactics (called transformations) like splitting conjunctions. To complete a verification the user typically has to assert intermediate facts and sometimes state and prove lemmas (expressed in our source language). In our case studies, the SMT-solvers Alt-Ergo, Z3, and CVC4 discharge all obligations automatically.

Translation to Why3. We encode methods and specs as Why3 functions that have specs. Why3 is procedure-modular: it verifies each function assuming the specs of the ones it imports, which corresponds to a hypothesis context in our logic. Why3 provides ghost annotations and checks that ghost code terminates and does not interfere with the underlying program. We use this feature to mark the allocation map, which is part of our heap model, and translate source code ghost state to Why3 ghost state. Why3 is sound under idealizations also made in our logic: unbounded integers and unbounded maps (which we used to model unbounded heap).

The Why3 language (including WhyML) does not include shared mutable objects. So, we use mutable records and maps to explicitly model the heap using the standard field-as-array representation, with references as an uninterpreted type and an extra field, alloct, for allocation to model the alloc variable and typing of references. WhyML has ML-style references constrained by static analysis that precludes aliasing; we use those to encode local variables. Invariants of source language semantics, like the absence of dangling pointers, are encoded using Why3's invariant feature for the data type of states. (States have the heap and global variables.) Common elements of translation are included in a WhyRel standard library that includes lemmas about operations on regions, which aids automated proving. Why3 specs include coarse grained reads and writes clauses enforced by simple syntactic analysis, which is not suited to our purposes. To encode the stateful frame conditions of our logic, WhyRel expresses write effects semantically, in universally quantified postconditions using "old" expressions. In accord with Definition 5.10, read effects are checked together with the encapsulation checks, discussed below.

WhyRel translates a biprogram to a WhyML function acting on a pair of states together with the current refperm. Relational pre- and postconditions are translated to WhyML requires/ensures. WhyRel represents a refperm by a pair of maps subject to universally quantified formulas that express bijectivity and are type-respecting. As an example, Figure 32 shows our source code for *sumpub* biprogram (15), together with its translation to WhyML. The WhyML loop body reflects the semantics of loop alignment guards. For readability, some dead code has been removed from the actual translation.

Checking read effects and encapsulation. By contrast with the check of write effects, WhyRel does not directly check the relational semantics of read effects (r-respect in Definition 5.10). Rather, it performs local checks based on the relevant conditions in the proof rules of our logic. When used for relational modular linking of modules with nontrivial boundaries, WhyRel must also enforce encapsulation, that is, the conditions on reads of if, while, bi-if, and bi-while, as well as the conditions of the context introduction rules used for atomic commands. These checks involve computing separator formulas, following a preliminary step that normalizes dynamic boundaries and expands the any datagroup to concrete fields. The tool immediately reports a violation when variables are required to be distinct but are not, or are read but not included in the read effect. For

```
meth sum (self:List | self:List) : (int | int)
  requires { 𝔹 self ≠ null }
  requires { ∃ ls:int list | ls:int list.
               𝔹 listpub(self.head,ls) ∧ ls ≐ ls }
  ensures  { 𝔸 result }
= var ghost xs : int list | ghost xs : int list in
  (* Initial values of math type variables are havoc'd;
     assume they witness the existential
     in the precondition *)
  assume { 𝔹 listpub(self.head,xs) };
  (* Initial value of result:int is 0 *)
  var p : Node | p : Node in
  ⌊ p := self.head ⌋;
  while (p ≠ null) | (p ≠ null) . ⟪ ¬ p.pub ⟨| | |⟩ ¬ p.pub ⟫
    invariant { 𝔹 listpub(p,xs) ∧ 𝔸 xs ∧ 𝔸 result }
    ( if p.pub then
        result := result + p.value; xs := tl(xs);
      fi; p := p.nxt
    | if p.pub then
        result := result + p.value; xs := tl(xs);
      fi; p := p.nxt )
  od;
```

```
let sum (σℓ σr: state) (π: refperm)
        (selfℓ selfr: reference) : (int, int)
  requires { selfℓ ≠ null ∧ σℓ.alloct[selfℓ] = List }
  requires { selfr ≠ null ∧ σr.alloct[selfr] = List }
  requires { ∃ lsℓ, lsr: int list.
               listpub σℓ σℓ.heap.head[selfℓ] lsℓ
             ∧ listpub σr σr.heap.head[selfr] lsr
             ∧ lsℓ = lsr }
  ensures { fst result = snd result }
= let ref resultℓ = 0 in (* default value for int *)
  let ref resultr = 0 in
  (* variables of math type initialized using any *)
  let ghost ref xsℓ = any (int list) in
  let ghost ref xsr = any (int list) in
  assume { listpub σℓ σℓ.heap.head[selfℓ] xsℓ
           ∧ listpub σr σr.heap.head[selfr] xsr }
  let ref pℓ = σℓ.heap.head[selfℓ] in
  let ref pr = σr.heap.head[selfr] in
  while (pℓ ≠ null) || (pr ≠ null) do
    invariant { listpub σℓ pℓ xsℓ ∧ listpub σr pr xsr }
    invariant { xsℓ = xsr ∧ resultℓ = resultr }
    invariant { (* generated using alignment guards *)
                  pℓ ≠ null ∧ ¬ σℓ.heap.pub[pℓ]
                ∨ pr ≠ null ∧ ¬ σr.heap.pub[pr]
                ∨ pℓ ≠ null ∧ pr ≠ null
                ∨ pℓ = null  ∧ pr = null }
    if (pℓ ≠ null && ¬ σℓ.heap.pub[pℓ]) then (* left *)
      pℓ ⟵ σℓ.heap.nxt[pℓ]
    else begin
      if (pr ≠ null && ¬ σr.heap.pub[pr]) then (* right *)
        pr ⟵ σr.heap.nxt[pr]
      else begin (* lockstep *)
        resultℓ ⟵ resultℓ + σℓ.heap.value[pℓ];
        xsℓ ⟵ tl xsℓ;
        pℓ ⟵  σℓ.heap.nxt[pℓ];
        resultr ⟵ resultr + σr.heap.value[pr];
        xsr ⟵ tl xsr;
        pr ⟵ σr.heap.nxt[pr]
      end;
    end;
  done; (resultℓ, resultr)
```

Fig. 32. WhyRel source biprogram for *sumpub* and translated WhyML (eliding frame conditions).

separation of heap locations, it generates disjointness formulas (in accord with Figure 11) in assert statements added to the generated code where the encap checks should be made. For reads of heap locations, it asserts an inclusion based on the reads allowed by the frame condition. A snapshot of the initial state is used so the frame condition can be interpreted where it should be; the asserted inclusion is at the point in the code where the read takes place, which may follow updates to the state.

When true, the disjointness and inclusion assertions for reads and encapsulation are usually proved without any need for user interaction. The user does see the assertions among the proof obligations enumerated by Why3. The user does not compute separators or effect subtractions, those are done by WhyRel.

Modular linking. In terms of the logic, Why3 verifies the premises of the standard linking rule (Link in Figure 23) so the contracts assumed by a procedure's callers are the ones for which the procedure's implementation is verified. WhyRel generates code that expresses hiding, i.e., the premises of our modular linking rules: the implementations get to assume the private invariant (or coupling, in the relational case) and must maintain it. For this to be sound, WhyRel checks encapsulation, as described above, and generates Why3 lemmas to encode the additional proof obligations.

```
lemma boundary_frames_QuickFind_invariant :
 ∀ σ: state, τ: state, π: refperm.
  okRefperm σ τ π ∧ identityRefperm π (domain σ.alloct) (domain τ.alloct) ⇒
  idRgn π σ.pool τ.pool ⇒ (* σ(pool) ≅ τ(pool) *)
  agreeAny σ τ π (union σ.pool (imgRep σ σ.pool)) ⇒
  ufPriv σ ⇒ (* private invariant I_uf *)
  ufPriv τ

lemma boundary_frames_UnionFind_coupling :
 ∀ σ: state, τ: state, σ': state, τ': state, π: refperm, π': refperm, ρ: refperm.
  okRefperm σ τ π ∧ identityRefperm π (domain σ.alloct) (domain τ.alloct) ⇒
  okRefperm σ' τ' π' ∧ identityRefperm π' (domain σ'.alloct) (domain τ'.alloct) ⇒
  okRefperm σ σ' ρ ∧ okRefperm τ τ' ρ ⇒
  idRgn π σ.pool τ.pool ⇒ (* σ(pool) ≅ τ(pool) *)
  agreeAny σ τ π (union σ.pool (imgRep σ σ.pool)) ⇒
  idRgn π' σ'.pool τ'.pool ⇒ (* σ'(pool) ≅ τ'(pool) *)
  agreeAny σ' τ' π' (union σ'.pool (imgRep σ' σ'.pool)) ⇒
  ufCoupling σ σ' ρ ⇒ (* coupling relation M_uf *)
  ufCoupling τ τ' ρ
```

Fig. 33. Framing judgments as lemmas.

For unary hiding, the private invariant should be framed by the module boundary; this obligation is generated in the form of a lemma that expresses the framing semantics (27). At the same time, WhyRel generates the obligation that the client precondition implies the private invariant. For relational hiding, the coupling invariant should be framed, on both left and right, by the boundary (using relational framing semantics Definition 7.1). Example framing lemmas are in Figure 33.

Another obligation generated in the form of a lemma is that the coupling should be refperm monotonic:

```
lemma ufCoupling_is_monotonic :
 ∀ σ: state, τ: state, π: refperm.
  okRefperm σ τ π ⇒ ufCoupling σ τ π ⇒
  ∀ ρ: refperm. okRefperm σ τ ρ ⇒ extends π ρ ⇒ ufCoupling σ τ ρ
```

WhyRel can generate a local equivalence spec, given boundaries and a unary spec; it is generated as source code, which the user can include in a biprogram. Local equivalence specs are defined in Section 8.1 and examples appear in Section 4.

Experience and findings. Despite achieving a high level of automation based on SMT solvers, auto-active tools require user effort and intelligence to devise specs and find loop invariants. Here, there is the additional task of writing a biprogram to express an alignment for which straightforward invariants suffice. (See Section 10 for work on automated inference of alignments.) Use of dynamic frames entails extensive reasoning about set expressions, set disjointness and containment. Aided by some lemmas in the WhyRel standard library, the solvers have little difficulty in this regard; the requisite reasoning about refperms also works fine. In most of our examples, the user needs to do a few clicks in Why3 to invoke the tactic to split conjunctions, and sometimes introduce assertions or lemmas that aid the solvers in finding proofs. Why3's assert tactic is helpful for this. This sort of interaction is typical in ordinary use of Why3.

For *sumpub*, we provide a couple of lemmas about the *listpub* relation, proved using the rule-induction transformation (i.e., a Why3 induction rule, dispatched to SMT). For the SSSP biprogram, we needed a number of asserts in the code (plus assert tactics); but not many for the other examples. Our priority has been to complete illustrative examples and a prototype that can be used by interested researchers; we have not tried to find optimal specs and minimal use of Why3 tactics. We are not proposing the concrete syntax for use in practice, nor does the tool provide sufficient error handling to be usable by software engineers. Moreover, although the prototype implements

some syntax sugar relative to the formal development, the current language has desugared loads and stores, which entails the use of annoyingly many temporary variables (sugared in examples in the article).

Finally, Why3 generates many proof obligations about the state being well formed, which is actually guaranteed by type-checking of source programs. The obligations are simple to prove but it is still one more thing to do. It should be possible to eliminate these through more sophisticated use of Why3's abstraction mechanisms. In BoogiePL these pointless obligations could be avoided using "free requires/ensures," and we could achieve the same effect using Why3 assumptions instead of type invariants; but the latter make it easier to read the generated WhyML.

Why3 records sessions to replay the user's choices of provers and tactics to apply. Replaying the sessions for our big case studies takes on the order of an hour or more of prover time, though clock time is a little faster owing to parallelism. The smaller examples take minutes or less. Less time would be needed if we used assumptions to avoid pointless checks about states being well formed. Significantly more automation could be achieved if Why3 enabled scripting of routine choices of tactics.

In summary, the formal development in preceding sections shows that general relational reasoning with encapsulation, for first-order programs, can be carried out using only first-order assertions and relations. The case studies carried out using WhyRel demonstrate that the verification conditions are well within what can be automated by SMT solvers. User interaction is needed mainly to deal with specs and loop invariants involving mathematical properties of data types and inductively defined predicates and relations. Inductive definitions are often needed for problem-specific properties, but are not required for encapsulation, framing, hiding or any other element of the logic.

10 RELATED WORK

Our main result (Theorem 8.1) brings together modular reasoning techniques, relational properties, representation independence, automated verification, and their semantic foundations.

We make a rough categorization of related work as follows: (Section 10.1) Directly related precursors; (Section 10.2) Algorithmic studies and implementations of automated verification for relational properties, often lacking detailed foundational justification and support for dynamic allocation or data abstraction, but identifying FOL fragments enabling automated inference of relational invariants and alignment; and (Section 10.3) Semantic studies of representation independence, focused on contextual equivalence and challenging language features including dynamic allocation, higher order procedures, and concurrency, leading to the higher order relational separation logic ReLoC implemented in the Coq proof assistant.

Union-find implementations have been verified interactively using Coq [32]. Functional correctness of Kruskal has been verified in a proof assistant [48]. Functional correctness of C implementations of Dijkstra's, Kruskal's, and Prim's algorithms have been verified by Mohan et al. [66] using VST [31]. The point of our case studies is to achieve automated equivalence proof for clients, without recourse to functional correctness. A purely applicative implementation of pairing heaps has been verified in Why3 (http://toccata.lri.fr/gallery/).

10.1 Region Logic and Other Logics with Explicit Footprints

Bao et al. [15] introduce a unified fine-grained region logic with both separating conjunction and explicit read/write effects, subsuming a fragment of separation logic. To enable effective use of SMT solvers, Piskac et al. [80, 81] encode separation logic style specifications using explicit regions. Several works implement implicit dynamic frames [67, 90], which combines the succinctness of

separation logic with the automation of SMT. For recent work on decidable fragments of separation logic, see Echenim et al. [38]. Using an extension of FOL with recursive definitions, the logic of Murali et al. [68] has an expression form for the footprint of a formula, akin to our *ftpt* operator but usable in formulas, avoiding the need for a separate framing judgment; this can encode a fragment of separation logic but effectiveness for automation has not been thoroughly evaluated.

The most closely related works are the RL articles. The image notation, introduced in RLI [14], was inspired by the use of field images to express relations in the information flow logic of Amtoft et al. [3]. In RLI this style of dynamic framing was shown to facilitate local reasoning about global invariants, and this was extended to dynamic boundaries and hiding of invariants in RLII [9].

In RLIII [12], pure methods are formalized with end-to-end read effects. The end-to-end semantics of read effects is also used in the preliminary work [11], from which we take biprograms, weaving, and bi-while alignment guards. But, we change the semantics of bi-com $(C|C')$ to eliminate one-sided divergences and to allow models to diverge (see rules uCall0 in Figure 22 and bCall0 in Figure 27). This validates a better weaving rule (no termination conditions) and a stronger adequacy theorem (Theorem 7.11). We drop their semantics of read effects, which is inadequate for our purposes (and is subsumed by r-respects in Definition 5.10), but use quasi-determinacy and agreement-preservation results from RLIII. Neither RLIII nor [11] addresses information hiding or encapsulation. Our semantics of encapsulation (Definition 5.10) is a major extension of that in RLII, from which we take the minimalist formalization of modules; but we change the semantics to use context models (from RLIII where models are called interpretations) and add r-respects, and so on. We adapt unary rules from RLII but use the term modular linking for what they call mismatch. The case studies in RLIII are implemented using Why3 with an encoding of heaps and frame conditions similar to the one used by WhyRel.

10.2 Relational Verification

Francez [43, 74] articulated the product principle reducing relational verification to the inductive assertion method and introduced a number of proof rules. Benton [25] introduced the term Relational Hoare Logic and brought to light applications including compiler optimizations. Yang [100] introduced relational separation logic, motivated by data abstraction although the logic does not formalize that as such. Beringer [27] extends Benton's logic with heap (still not procedures), and provides proof rules for non-lockstep loops, on which our rWhile is based; a similar rule appears in Barthe et al. [22]. There has been a lot of work on relational logics and verification techniques [24], e.g., applications in security and privacy [21, 70, 83] and merges of software versions [94]. A shallow embedding of relational Hoare logic in $F^\star$ is used to interactively prove refinements between union-find implementations [47]. Aguirre et al. [1] develop a logic based on relational refinement types, for terminating higher order functional programs, and provide an extensive discussion of work on relational logics.

Automated relational verification based on product programs is implemented in several works that address effective alignment of control flow points and the inference of alignment points and relational assertions and procedure summaries [16–18, 34, 40, 55, 99, 101, 102]. One line of work, centered around the SymDiff verifier [50, 56, 57], proves properties of program differences using relational procedure summaries. Godlin and Strichmann [46] prove soundness of proof rules for equivalence checking taking into account similar and differing calls. Eilers et al. [39] implement a novel product construction for procedure-modular verification of k-safety properties of a program, maximizing use of relational specs for procedure calls. (We follow O'Hearn et al. [77] in using "modular" to imply also information hiding.) Girka et al. [45] explore forms of alignment automata. Shemer et al. [89] provide for flexible alignments and infer state-dependent alignment conditions,

as do Unno et al. [97]. The latter works rely on constraint solving techniques, which are not yet applicable to the heap. For the heap the state of the art for finding alignments is syntactic matching heuristics.

For ∀∃ properties, product constructions appear in some recent works [5, 17, 35, 59, 97]. Pioneering work by Rinard and Marinov [85, 86] introduces a logic of ∀∃ simulations for correct compilation, for programs represented as control flow graphs.

Sousa and Dillig's Cartesian Hoare Logic [93] (a generalization of Benton's logic) can be used to reason about k-safety properties such as secure information flow (2-safety) and transitivity (3-safety). They also develop an algorithm, based on an implicit product construction, for automatically proving k-safety properties; The corresponding tool, Descartes, has been used in the verification of several user-defined relational operators in Java programs. For more efficient relational verification, Pick et al. [79] introduce a new algorithm atop Descartes, which automatically detects opportunities for alignment (the synchrony phase) and detects opportunities for pruning subtasks by exploiting symmetries in program structure and relational specs.

None of the above works address hiding, and many do not fully handle the heap [58]. Our work is complementary, providing a foundation for verified toolchains implementing these algorithmic techniques. The use of rWhile with alignment guards, together with the disjunction rule to split cases and unconditional rewriting (Section 8.6), enables our logic to express a wide range of state-dependent alignments.

10.3 Representation Independence

It is difficult to account for encapsulation in semantics of languages with dynamically allocated mutable state and especially with higher order features. Crary's tour de force proves parametricity for a large fragment of ML but excluding reference types [36]. Semantic studies of the problem [2, 7] have been connected with unary [10] and relational logics [37]. The latter relies on intensional atomic propositions about steps in the transition semantics. In this sense it is very different from standard (Hoare-style) program logics.

Birkedal and Yang [30] show client code proved correct using the SOF rule of separation logic is relationally parametric, using a semantics that does not validate the rule of conjunction, which plays a key role in automated verification. That rule is an issue in some other models as well, e.g., Iris (in part owing to its treatment of ghost updates as logical operators).

Thamsborg et al. [96] also lift separation logic to a relational interpretation, but instead of second-order framing, address abstract predicates. Their goal is to give a relational interpretation of proofs. They uncover and solve a surprising problem: due to the nature of entailment in separation logic, not all uses of the rule of consequence lift to relations. Our logic does not directly lift proofs but does lift judgments from unary to relational (the rEmb and rLocEq rules). In general, most works on representation independence, including work on encapsulation of mutable objects, are essentially semantic developments [7, 10]; general categorical models of Reynolds' relational parametricity [84], which validate his abstraction theorem and identity extension lemma have been developed and are under active study by Johann et al. [92].

The state of the art for data abstraction in separation logics is abstract predicates, which are satisfactory in many specs where some abstraction of ADT state is of interest to clients, but less attractive for composing libraries such as runtime resource management with no client-relevant state. Such logics have been implemented in interactive provers [29, 53, 71]. These are unary logics with concurrency; they do not feature second-order framing but they have been used to verify challenging concurrent programs. As shown by the recent extension of VST with Verified Software Units [28], higher order logics with impredicative quantification facilitate expressive interface specifications for modular reasoning about heap-based programs.

ReLoC [44], based on Iris [53], is a relational logic for conditional contextual refinement of higher order concurrent programs. Iris and the works in the preceding paragraph do support hiding in the sense of abstraction: through existential quantification and abstract predicates, and in Iris through the invariant-box modality and the associated "masks." With respect to our context and goals, we find such machinery to be overkill. Like O'Hearn et al. [77], we only need invariants in the sense of conditions that hold when control enters or exits the module—not conditions that hold at every step. There is a considerable gap between this work and the properties/techniques for which automation has been developed; moreover their step-indexed semantics does not support termination reasoning or transitive composition of relations (which needs relative termination [50]); our logic is easily adapted to both.

Maillard et al. [65] provide a general framework for relational program logics that can be instantiated for different computational effects represented by monads. The paper does not address encapsulation, except insofar as the system is based on dependent-type theory.

11 CONCLUSION

We introduced a relational Hoare logic that accounts for strong encapsulation of data representations in object-based programs with dynamic allocation and shared mutable data structures. Consequently, changes to internal data representations of a module can be proved to lead to equivalent observable behaviors of clients that have been proved to respect encapsulation. The technique of simulation, articulated by Hoare [52] and formalized in theories of representation independence, is embodied directly in the logic as a proof rule (rMLink in Figure 31). The logic provides means for specifying state-based encapsulation methodologies such as ownership. It also supports effective relational reasoning about simulation between both similar and disparate control and data structure. Although our exposition focuses on encapsulation and simulation, the logic is general, encompassing a range of relational properties including conditional equivalence (including compiler optimizations), specified differencing (as in regression verification), and secure information flow with downgrading [3, 11, 13, 33]. The rules are proved sound.

The programmer's perspective articulated by Hoare is about a single module and client, distinguishing inside versus outside. The general case, with state-based encapsulation for a hierarchy of modules, requires a precise definition of the boundaries within which a given execution step lies. While we build on prior work on state-based encapsulation, we find that to support change of representation, the semantics of encapsulation needs to be formulated in terms of not only the context (hypotheses/library APIs) but also modular structure of what's already linked, via the dynamic call chain embodied by the runtime stack. This novel formulation of an extensional semantics for encapsulation against dependency is subtle (Definition 5.10), yet it remains amenable to simple enforcement. Our relational assertions and verification conditions for modules and clients are first-order. As proof of concept, we demonstrate that they can be effectively used in an auto-active SMT-based verification prototype.

To a great extent, the three goals in Section 1 have been achieved. Beyond this progress, for foundational justification one might like to machine check the soundness proofs. For automation, one could explore techniques for inferring alignment conditions and relational invariants [89, 97].

Apropos completeness of the logic, the ordinary notion of completeness is that valid relational judgments are provable (relative to validity of entailments). Completeness in this sense is an immediate consequence of completeness of the underlying unary logic together with the presence of a single rule (like rEmb) that lifts unary judgments to relational ones [19, 20, 43]—provided that unary assertions can express relations. That proviso is easy to establish for simple imperative programs, by using renamed variables. For pointer programs, expressing a relation as an assertion can be done using separating conjunction [19], but to do so using only FO assertions requires a

complicated encoding [72]. The recently introduced notion of alignment completeness [69] is better than ordinary completeness as a way to evaluate relational logics. We have not yet investigated completeness for either unary or relational region logic.

12 ENVOI

Hoare's 1972 paper articulates the fundamental notions of hiding and encapsulation with a minimum of extraneous formalization. In seeking to formulate the ideas in a logic for first-order programs using first-order assertions, we hoped to achieve a comparably elementary and transparent account. To handle dynamically allocated mutable state, however, we have been unable to avoid some amount of auxiliary notions.

Having incorporated encapsulation into a unary+relational logic that supports hiding of internal invariants, we are poised to investigate a longstanding problem: the hiding of unobservable effects for object-based programs. This is intimately connected with encapsulation [26, 73, 82] and appears already in Hoare's work under the term benevolent side effects [52].

APPENDICES

A PROGRAM SEMANTICS AND UNARY CORRECTNESS (RE SECTION 5)

A.1 On Effects, Agreement, and Valid Correctness Judgment

Lemma 5.2 (Subtraction). *$rlocs(\sigma, \varepsilon\backslash\eta) = rlocs(\sigma, \varepsilon)\backslash rlocs(\sigma, \eta)$ and the same for wlocs.*

Proof. Assume w.l.o.g. that ε and η are in the normal form described as part of the definition, Equation (7). For a variable x, we get $x \in rlocs(\sigma, \varepsilon\backslash\eta)$ iff $x \in rlocs(\sigma, \varepsilon)\backslash rlocs(\sigma, \eta)$ directly from definitions. For a heap location, $o.f$ is in $rlocs(\sigma, \varepsilon)\backslash rlocs(\sigma, \eta)$ just if there is $\mathrm{rd}\, G\text{`}f$ in $\varepsilon$ with $o \in \sigma(G)$ and there is no $\mathrm{rd}\, H\text{`}f$ in η with $o \in \sigma(H)$ (by definitions). This can happen in two cases: either there is no read for f in η, or there is $\mathrm{rd}\, H\text{`}f$ in $\eta$ but $o \notin \sigma(H)$. In the first case, $\mathrm{rd}\, G\text{`}f$ is in $\varepsilon\backslash\eta$ so $o \in rlocs(\varepsilon\backslash\eta)$. In the second case, $\mathrm{rd}\, (G\backslash H)\text{`}f$ is in $\varepsilon\backslash\eta$, and since $o \in \sigma(G\backslash H)$, we have $o \in rlocs(\varepsilon\backslash\eta)$. □

Lemma 5.6. *Suppose $\sigma \overset{\pi}{\approx} \sigma'$. Then $\sigma(F) \overset{\pi}{\sim} \sigma'(F)$, and $\sigma \models P$ iff $\sigma' \models P$.*

Proof. Straightforward, by induction on F and induction on P. □

Remark 2. For partial correctness, all specs are satisfiable (at least by divergence). This is manifest in Definition 5.9, which allows that $\varphi(m)(\sigma)$ can be $\varnothing$ for any σ that satisfies the precondition. In RLII, a context call faults in states where the precondition does not hold. It gets stuck if the precondition holds but there is no successor state that satisfies the postcondition. Here (and in RLIII, for impure methods), the latter situation can be represented by a model that returns the empty set. Instead of letting the semantics get stuck, we include a stuttering transition, uCall0.

Remark 3. Apropos Definition 5.10, one might expect r-respect to consider steps $\langle B, \tau', \mu\rangle \overset{\varphi}{\longmapsto} \langle D', v', v'\rangle$ with potentially different environment v', and add to the consequent that $v' = v$. But in fact the only transitions that affect the environment are those for let and for the elet command used in the semantics at the end of its scope. The transitions for these are independent of the state, and so B and μ suffice to determine v.

Remark 4. The consequent (25) of r-respect express that the visible (outside boundary) writes and allocations depend only on the visible starting state. One may wonder whether the conditions fully capture dependency, noting that they do not consider faulting. But r-respects is used in conjunction with the (Safety) condition that rules out faults.

Remark 5. In separation logic, preconditions serve two purposes: in addition to the usual role as an assumption about initial states, the precondition also designates the "footprint" of the command. This is usually seen as a frame condition: the command must not read or write any preexisting locations outside the footprint of the precondition. In a logic such as the one in this article, where frame conditions are distinct from preconditions, it is possible for the frame condition to designate a smaller set of locations than the footprint of the precondition. As a simple example, consider the spec $x > 0 \wedge y > 0 \rightsquigarrow \text{true}\ [\text{rw}\, x]$. In our logic, it is possible for two states to agree on the read effect but disagree on the precondition. For example, the states $[x : 1, y : 0]$ and $[x : 1, y : 1]$ agree on x but only the second satisfies $x > 0 \wedge y > 0$. Lemma 5.11 describes the read effect only in terms of states that satisfy the precondition. For a command satisfying the example spec, and the states $[x : 1, y : 1]$ and $[x : 1, y : 2]$, which satisfy the precondition but do not agree on y, that the command must either diverge on both states or converge to states that agree on the value of x.

Lemma A.1 (Agreement Symmetry). *Suppose ε has framed reads. If $Agree(\sigma, \sigma', \pi, \varepsilon)$, then (a) $rlocs(\sigma', \varepsilon) = \pi(rlocs(\sigma, \varepsilon))$ and (b) $Agree(\sigma', \sigma, \pi^{-1}, \varepsilon)$.*

Proof. (a) For variables the equality follows immediately by definition of *rlocs*. For heap locations the argument is by mutual inclusion. To show $rlocs(\sigma', \varepsilon) \subseteq \pi(rlocs(\sigma, \varepsilon))$, let $o.f \in rlocs(\sigma', \varepsilon)$. By definition of *rlocs*, there exists region G such that ε contains $\text{rd}\, G\text{`}f$ and $o \in \sigma'(G)$. Since ε has framed reads, ε contains $ftpt(G)$, hence from $Agree(\sigma, \sigma', \pi, \varepsilon)$ by Equation (28) we get $\sigma(G) \overset{\pi}{\sim} \sigma'(G)$. Thus, $o \in \pi(\sigma(G))$. So, we have $o.f \in \pi(rlocs(\sigma, \varepsilon))$. Proof of the reverse inclusion is similar.

(b) For variables this is straightforward. For heap locations, consider any $o.f \in rlocs(\sigma', \varepsilon)$. From (a), we have $\pi^{-1}(o).f \in rlocs(\sigma, \varepsilon)$. From $Agree(\sigma, \sigma', \pi, \varepsilon)$, 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 definition of r-respect is formulated (in Definition 5.10) in a way to make evident that client steps are independent from locations within the boundary. But r-respect can be simplified, as follows, when used in conjunction with w-respects.

The following notion is used to streamline the statement of some technical results. It is used with states $\sigma, \tau, \tau', \upsilon, \upsilon'$, where σ is an initial state from which τ and then later υ is reached, and in a parallel execution τ' reaches υ'. Moreover, δ is a dynamic boundary. We write $\delta^{\oplus}$ to abbreviate δ, rd alloc.

Definition A.2. Say ε ***allows dependence from*** τ, τ' ***to*** υ, υ' ***for*** σ, δ, π, written $\tau, \tau' \overset{\pi}{\Rightarrow} \upsilon, \upsilon' \models^{\sigma}_{\delta} \varepsilon$ iff the agreement $Lagree(\tau, \tau', \pi, (freshL(\sigma, \tau) \cup rlocs(\sigma, \varepsilon)) \backslash rlocs(\tau, \delta^{\oplus}))$ implies there is $\rho \supseteq \pi$ with $Lagree(\upsilon, \upsilon', \rho, (freshL(\tau, \upsilon) \cup wrttn(\tau, \upsilon)) \backslash rlocs(\upsilon, \delta^{\oplus}))$.

Like Definition 5.4, 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. This is tamed in case ε has framed reads (Lemma A.1).

Allowed dependence gives an alternate way to express part of the Encap condition in Definition 5.10. For a step $\langle B, \tau, \mu \rangle \overset{\varphi}{\longmapsto} \langle D, \upsilon, \nu \rangle$ that r-respects δ for $(\varphi, \varepsilon, \sigma)$ and $Active(B)$ is not a call, and alternate step (24), the condition implies $\tau, \tau' \overset{\pi}{\Rightarrow} \upsilon, \upsilon' \models^{\sigma}_{\delta} \varepsilon$ in the notation of Definition A.2.

A critical but non-obvious consequence of framed reads 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. The formal statement is somewhat intricate; it generalizes RLIII Lemma 6.12.

Lemma A.3 (Balanced Symmetry). *Suppose* $\tau, \tau' \overset{\pi}{\Rightarrow} v, v' \models^{\sigma}_{\delta} \varepsilon$ *and* $\tau', \tau \overset{\pi^{-1}}{\Rightarrow} v', v \models^{\sigma'}_{\delta} \varepsilon$. *Suppose*

$$\begin{array}{l} Lagree(\tau, \tau', \pi, (freshL(\sigma, \tau) \cup rlocs(\sigma, \varepsilon)) \backslash rlocs(\tau, \delta^{\oplus})), \\ Lagree(\tau', \tau, \pi^{-1}, (freshL(\sigma', \tau') \cup rlocs(\sigma', \varepsilon)) \backslash rlocs(\tau', \delta^{\oplus})). \end{array}$$

Let ρ, ρ' *be any refperms with* $\rho \supseteq \pi$ *and* $\rho' \supseteq \pi^{-1}$ *that witness the allowed dependencies, i.e.,*

$$\begin{array}{l} Lagree(v, v', \rho, (freshL(\tau, v) \cup wrttn(\tau, v)) \backslash rlocs(v, \delta^{\oplus})), \\ Lagree(v', v, \rho', (freshL(\tau', v') \cup wrttn(\tau', v')) \backslash rlocs(v', \delta^{\oplus})). \end{array} \tag{35}$$

Furthermore, suppose

$$\begin{array}{l} \rho(freshL(\tau, v) \backslash rlocs(v, \delta)) \subseteq freshL(\tau', v') \backslash rlocs(v', \delta), \\ \rho'(freshL(\tau', v') \backslash rlocs(v', \delta)) \subseteq freshL(\tau, v) \backslash rlocs(v, \delta). \end{array} \tag{36}$$

Then, we also have

$$\begin{array}{l} Lagree(v', v, \rho^{-1}, (freshL(\tau', v') \cup wrttn(\tau', v')) \backslash rlocs(v', \delta^{\oplus})), \\ \rho(freshL(\tau, v)) \backslash rlocs(v, \delta) \;=\; freshL(\tau', v') \backslash rlocs(v', \delta). \end{array}$$

Proof. From Definition 5.3 and Equation (35), we know that ρ and ρ' are total on $freshL(\tau, v) \backslash rlocs(v, \delta)$ and $freshL(\tau', v') \backslash rlocs(v', \delta)$, respectively. Since ρ and ρ' are bijections, from Equation (36), we have equal cardinalities: $|freshL(\tau, v) \backslash rlocs(v, \delta)| = |freshL(\tau', v') \backslash rlocs(v', \delta)|$. So, we get $\rho(freshL(\tau, v) \backslash rlocs(v, \delta)) = freshL(\tau', v') \backslash rlocs(v', \delta)$. Now from Equation (35) using the symmetry lemma Equation (22) for *Lagree*, we get

$$Lagree(v', v, \rho^{-1}, \rho(freshL(\tau, v) \backslash rlocs(v, \delta))).$$

So, we have $Lagree(v', v, \rho^{-1}, freshL(\tau', v') \backslash rlocs(v', \delta))$. However, we have $wrttn(\tau', v') \subseteq locations(\tau')$, and we have $\rho'|_{locations(\tau')} = \pi^{-1}|_{locations(\tau')} = \rho^{-1}|_{locations(\tau')}$, using vertical bar for domain restriction. So, from Equation (35), we get

$$Lagree(v', v, \pi^{-1}, wrttn(\tau', v') \backslash rlocs(v', \delta^{\oplus})),$$

which we can write as $Lagree(v', v, \rho^{-1}, wrttn(\tau', v') \backslash rlocs(v', \delta^{\oplus}))$. Thus, we get

$$Lagree(v', v, \rho^{-1}, (freshL(\tau', v') \cup wrttn(\tau', v')) \backslash rlocs(v', \delta^{\oplus})). \qquad \square$$

Lemma A.4 (Preservation of Agreement). *Suppose* $\tau, \tau' \overset{\pi}{\Rightarrow} v, v' \models^{\sigma}_{\delta} \varepsilon$ *and* $\tau', \tau \overset{\pi^{-1}}{\Rightarrow} v', v \models^{\sigma'}_{\delta} \varepsilon$. *Suppose*

$$\begin{array}{l} Lagree(\tau, \tau', \pi, (freshL(\sigma, \tau) \cup rlocs(\sigma, \varepsilon)) \backslash rlocs(\tau, \delta^{\oplus})) \quad \textit{and} \\ Lagree(\tau', \tau, \pi^{-1}, (freshL(\sigma', \tau') \cup rlocs(\sigma', \varepsilon)) \backslash rlocs(\tau', \delta^{\oplus})). \end{array}$$

Then for any $W \subseteq locations(\tau)$, *if* $Lagree(\tau, \tau', \pi, W)$ *then* $Lagree(v, v', \rho, W \backslash rlocs(v, \delta^{\oplus}))$, *for any refperm* ρ *that witnesses* $\tau, \tau' \overset{\pi}{\Rightarrow} v, v' \models^{\sigma}_{\delta} \varepsilon$.

Proof. Suppose $Lagree(\tau, \tau', \pi, (freshL(\sigma, \tau) \cup rlocs(\sigma, \varepsilon)) \backslash rlocs(\tau, \delta^{\oplus}))$ suppose that $\rho \supseteq \pi$ witnesses $\tau, \tau' \overset{\pi}{\Rightarrow} v, v' \models^{\sigma}_{\delta} \varepsilon$, so we get

$$Lagree(v, v', \rho, (freshL(\tau, v) \cup wrttn(\tau, v)) \backslash rlocs(v, \delta^{\oplus}))). \tag{37}$$

Suppose $Lagree(\tau', \tau, \pi^{-1}, (freshL(\sigma', \tau') \cup rlocs(\sigma', \varepsilon)) \backslash rlocs(\tau', \delta^{\oplus}))$ and let $\rho' \supseteq \pi^{-1}$ witness $\tau', \tau \overset{\pi^{-1}}{\Rightarrow} v', v \models^{\sigma'}_{\delta} \varepsilon$, so we get

$$Lagree(v', v, \rho', (freshL(\tau', v') \cup wrttn(\tau', v')) \backslash rlocs(v', \delta^{\oplus})). \tag{38}$$

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

$$Lagree(v, v', \rho, W \backslash rlocs(v, \delta^{\oplus})).$$

For $x \in W \backslash rlocs(v, \delta^{\oplus})$, either $x \in wrttn(\tau, v)$ or $\tau(x) = v(x)$.

- If $x \in wrttn(\tau, v)$, then from Equation (37), we have $v(x) \overset{\rho}{\sim} v'(x)$.
- If $\tau(x) = v(x)$, then we claim that $\tau'(x) = v'(x)$. It follows that from $Lagree(\tau, \tau', \pi, W)$, we have $v(x) = \tau(x) \overset{\pi}{\sim} \tau'(x) = v'(x)$.

  We prove the claim by contradiction. If it does not hold, then $x \in wrttn(\tau', v')$. By Equation (38) this implies $v'(x) \overset{\rho'}{\sim} v(x) = \tau(x) \overset{\pi}{\sim} \tau'(x)$. Then, since $\rho' \supseteq \pi^{-1}$, we would have $\tau'(x) = \pi(\pi^{-1}(v'(x))) = v'(x)$, which is a contradiction.

For $o.f \in W \backslash rlocs(v, \delta^{\oplus})$, either $o.f \in wrttn(\tau, v)$ or $\tau(o.f) = v(o.f)$.

- If $o.f \in wrttn(\tau, v)$, then from Equation (37), we have $v(o.f) \overset{\rho}{\sim} v'(\rho(o).f)$.
- If $\tau(o.f) = v(o.f)$, then we claim that $\tau'(\pi(o).f) = v'(\pi(o).f)$. It follows that from $Lagree(\tau, \tau', \pi, W)$, we have $v(o.f) = \tau(o.f) \overset{\pi}{\sim} \tau'(\pi(o).f) = v'(\pi(o).f)$.

  The claim $\tau'(\pi(o).f) = v'(\pi(o).f)$ is proved by contradiction. If it does not hold, then $\pi(o).f \in wrttn(\tau', v')$. By (38) this implies $v'(\pi(o).f) \overset{\rho'}{\sim} v(\rho'\pi(o).f) = v(o.f) = \tau(o.f) \overset{\pi}{\sim} \tau'(\pi(o).f)$. Then, since $\rho' \supseteq \pi^{-1}$, we would have $\tau'(\pi(o).f) = \pi(\pi^{-1}(v'(\pi(o).f))) = v'(\pi(o).f)$, hence $\tau'(\pi(o).f) = v'(\pi(o).f)$, which is a contradiction.

This completes the proof of $Lagree(v, v', \pi, W \backslash rlocs(v, \delta^{\oplus}))$ for heap locations. □

Lemma A.5 (Subeffect). *If $P \models \varepsilon \leq \eta$, then the following hold for all $\sigma, \sigma', \tau, \tau', v, v', \pi, \delta$ such that $\sigma \models P$ and $\sigma' \models P$: (a) $\sigma \rightarrow \tau \models \varepsilon$ implies $\sigma \rightarrow \tau \models \eta$; (b) $Agree(\sigma, \sigma', \pi, \eta)$ implies $Agree(\sigma, \sigma', \pi, \varepsilon)$; and (c) $\tau, \tau' \overset{\pi}{\Rightarrow} v, v' \models^{\sigma}_{\delta} \varepsilon$ implies $\tau, \tau' \overset{\pi}{\Rightarrow} v, v' \models^{\sigma}_{\delta} \eta$.*

Proof. Straightforward from the definitions. For part (c), we have $rlocs(\sigma, \varepsilon) \subseteq rlocs(\sigma, \eta)$, so η gives a stronger antecedent in Definition A.2 and the consequent is unchanged between ε and η. □

A.2 On the Transition Relation

Figure 34 completes the definition of the transition relation, with respect to a given pre-model φ.[42] The definition is also parameterized by a function, *Fresh*, for which we assume that, for any σ, $Fresh(\sigma)$ a non-empty set of non-null references that are not in σ(alloc).

We take care to model realistic allocators, allowing their behavior to be nondeterministic at the level of states, to model their dependence on unobservable low-level implementation details, yet not requiring the full, unbounded allocator required by some separation logics. However, the language is meant to be deterministic modulo allocation. To make that possible for local variables, we assume given a function $FreshVar : states \rightarrow LocalVar$ such that $FreshVar(\sigma) \notin Vars(\sigma)$. We also assume that *FreshVar* depends only on the domain of the state:

$$Vars(\sigma) \backslash SpecOnlyVars = Vars(\sigma') \backslash SpecOnlyVars \text{ implies } FreshVar(\sigma) = FreshVar(\sigma'). \quad (39)$$

[42] To be very precise, in the transition rules for context calls (Figure 22), we implicitly use a straightforward coercion: the pre-model is applied to states that may have more variables than the ones in scope for the method context Φ for φ. Suppose Φ is wf in Γ. For method m in Φ, $\varphi(m)$ is defined on Γ-states. Suppose σ is a state for Γ plus some additional variables $\overline{x}$ (including but not limited to spec-only variables). Then $\varphi(m)(\sigma)$ is defined by discarding the additional variables and applying σ. If the result is a set of states, then each of these states is extended with the additional variables mapped to their initial values. This coercion is implicitly used in the rules context calls, i.e., rules uCall, uCallX, and uCall0 in Figure 22. The coercion is also used in RLIII where it is formalized in more detail.

$$\textsc{uLoad}\ \frac{\sigma(y) = o \qquad o \neq \mathit{null}}{\langle x := y.f,\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \langle \mathsf{skip},\ [\sigma \mid x{:}\,\sigma(o.f)],\ \mu\rangle} \qquad \textsc{uLoadX}\ \frac{\sigma(y) = \mathit{null}}{\langle x := y.f,\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \lightning}$$

$$\textsc{uStore}\ \frac{\sigma(x) = o \qquad o \neq \mathit{null}}{\langle x.f := y,\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \langle \mathsf{skip},\ [\sigma \mid o.f{:}\,\sigma(y)],\ \mu\rangle} \qquad \textsc{uStoreX}\ \frac{\sigma(x) = \mathit{null}}{\langle x.f := y,\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \lightning}$$

$$\textsc{uAssg}\ \langle x := F,\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \langle \mathsf{skip},\ [\sigma \mid x{:}\,\sigma(F)],\ \mu\rangle \qquad \textsc{uSeq}\ \frac{\langle C,\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \langle D,\ \tau,\ \nu\rangle}{\langle C; B,\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \langle D; B,\ \tau,\ \nu\rangle} \qquad \textsc{uSeqX}\ \frac{\langle C,\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \lightning}{\langle C; B,\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \lightning}$$

$$\textsc{uNew}\ \frac{\begin{array}{c} o \in \mathit{Fresh}(\sigma) \\ \mathit{Fields}(K) = \overline{f} : \overline{T} \qquad \sigma_1 = \text{“}\sigma \text{ with } o \text{ added to heap, with type } K \text{ and default field values”} \end{array}}{\langle x := \mathsf{new}\ K,\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \langle \mathsf{skip},\ [\sigma_1 \mid x{:}\,o],\ \mu\rangle}$$

$$\textsc{uVar}\ \frac{x' = \mathit{FreshVar}(\sigma)}{\langle \mathsf{var}\ x{:}T\ \mathsf{in}\ C,\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \langle C^{x}_{x'}; \mathsf{evar}(x'),\ [\sigma{+}x'{:}\,\mathit{default}(T)],\ \mu\rangle} \qquad \textsc{uEVar}\ \langle \mathsf{evar}(x),\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \langle \mathsf{skip},\ \sigma \restriction x,\ \mu\rangle$$

$$\textsc{uWhT}\ \frac{\sigma(E) = \mathsf{true}}{\langle \mathsf{while}\ E\ \mathsf{do}\ C,\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \langle C; \mathsf{while}\ E\ \mathsf{do}\ C,\ \sigma,\ \mu\rangle} \qquad \textsc{uWhF}\ \frac{\sigma(E) = \mathsf{false}}{\langle \mathsf{while}\ E\ \mathsf{do}\ C,\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \langle \mathsf{skip},\ \sigma,\ \mu\rangle}$$

$$\textsc{uIfT}\ \frac{\sigma(E) = \mathsf{true}}{\langle \mathsf{if}\ E\ \mathsf{then}\ C\ \mathsf{else}\ D,\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \langle C,\ \sigma,\ \mu\rangle} \qquad \textsc{uIfF}\ \frac{\sigma(E) = \mathsf{false}}{\langle \mathsf{if}\ E\ \mathsf{then}\ C\ \mathsf{else}\ D,\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \langle D,\ \sigma,\ \mu\rangle}$$

Fig. 34. Rules for unary transition relation $\overset{\varphi}{\longmapsto}$ omitted from Figure 22.

These technicalities are innocuous and consistent with stack allocation of locals.

A configuration *cfg* ***faults*** if $\mathit{cfg} \overset{\varphi}{\longmapsto}{}^{*} \lightning$. It ***faults next*** if $\mathit{cfg} \overset{\varphi}{\longmapsto} \lightning$. It ***terminates*** if $\mathit{cfg} \overset{\varphi}{\longmapsto}{}^{*} \langle \mathsf{skip},\ \tau,\ \_\rangle$ for some τ— so “terminates” means eventual normal termination. When applied to traces, these terms refer to the last configuration: a trace faults if it can be extended to a trace in which the last configuration faults next. Perhaps it goes without saying that *cfg* ***diverges*** means it begins an infinite sequence of transitions; in other words, it has traces of unbounded length.

For any pre-model φ, the transition relation $\overset{\varphi}{\longmapsto}$ is total in the sense that, for any $\langle C,\ \sigma,\ \mu\rangle$ with $C \not\equiv \mathsf{skip}$, there is an applicable rule and hence a successor—which may be another configuration or $\lightning$. This relies on the starting configuration being well formed in the sense that all free methods are bound either in the model or the environment, all free variables are bound in the state, and the command has no occurrences of evar or elet. Moreover, $\mathsf{evar}(x)$ (respectively, $\mathsf{elet}(m)$) only occurs in a configuration if x is in the state (respectively, m is in the environment).

Well formedness is preserved by the transition rules, and can be formalized straightforwardly (see RLII), but in this article, we gloss over it for the sake of clarity.

The transition relation $\overset{\varphi}{\longmapsto}$ is called ***rule-deterministic*** if for every configuration $\langle C,\ \sigma,\ \mu\rangle$ there is at most one applicable transition rule. Strictly speaking, this is a property of the definition (Figures 22 and 34), not of the relation $\overset{\varphi}{\longmapsto}$.

Lemma A.6 (Quasi-determinacy of Transitions). *For any pre-model φ,*

(a) $\overset{\varphi}{\longmapsto}$ is rule-deterministic.
(b) 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 $D \equiv D'$, $\nu = \nu'$, and $\tau \overset{\rho}{\approx} \tau'$ for some $\rho \supseteq \pi$.
(c) If $\sigma \overset{\pi}{\approx} \sigma'$, then $\langle C,\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \lightning$ iff $\langle C,\ \sigma',\ \mu\rangle \overset{\varphi}{\longmapsto} \lightning$.

Proof. (a) This is straightforward to check by inspection of the transition rules: for each command form, check that the applicable rules are mutually exclusive. One subtlety is in the case of context call. If there is $\tau \in \varphi(m)(\sigma)$, and also $\lightning \in \varphi(m)(\sigma)$, then two transition rules can be used for $\langle m(),\ \sigma,\ \mu\rangle$. This is disallowed by Definition 5.7 (fault determinacy). Also, Definition 5.7 (state determinacy), and condition (iii) in the definition of $\cong_\pi$ (Definition 5.5) distinguishes between the two transition rules for empty and non-empty $\varphi(m)(\sigma)$ (see Figure 22).

(b) Go 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 mod π. For the case of $x := E$, we need that expression evaluation respects isomorphism of states, Lemma 5.6. For allocation, let $\rho = \{(o, o')\} \cup \pi$ where o, o' are the allocated objects. For context call, we get the result by the determinacy conditions of Definition 5.7. The only commands that alter the environment are let and elet, and we get $\nu = \nu'$, because their behavior is independent of the state.

(c) Similar to the proof of (b); using item (i) in the definition of $\cong_\pi$, for context calls. □

A consequence of (a) is that the transition relation is fault deterministic: no configuration has both a fault and non-fault successor (by inspection, no single rule yields both fault and non-fault). We note these other corollaries:

(d) 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 $D \equiv D'$, $\nu = \nu'$, and $\tau \overset{\rho}{\approx} \tau'$ for some $\rho \supseteq \pi$ (by induction on i).

(e) 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 $\rho \supseteq \pi$ (because only skip lacks a successor).

(f) From a given configuration $\langle C,\ \sigma,\ \mu\rangle$, exactly one of these three outcomes is possible: normal termination, faulting termination, divergence.

Lemma 5.11 (Read Effect). *Suppose $\Phi \models^\Gamma_M C : P \rightsquigarrow Q[\varepsilon]$ and φ is a Φ-model. Suppose $\sigma \models P$ and $\sigma' \models P$. Suppose $Lagree(\sigma, \sigma', \pi, rlocs(\sigma, \varepsilon)\backslash\{\text{alloc}\})$. Then $\langle C,\ \sigma,\ \_\rangle$ diverges iff $\langle C,\ \sigma',\ \_\rangle$ diverges. And for any τ, τ', if $\langle C,\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle \text{skip},\ \tau,\ \_\rangle$ and $\langle C,\ \sigma',\ \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle \text{skip},\ \tau',\ \_\rangle$ then*

$$\exists \rho \supseteq \pi.\quad Lagree(\tau, \tau', \rho, (freshL(\sigma, \tau) \cup wrttn(\sigma, \tau))\backslash\{\text{alloc}\})\ \ and\ \ \rho(freshL(\sigma, \tau)) \subseteq freshL(\sigma', \tau').$$

Proof. To prove the lemma, we prove a stronger result. □

Claim. *Under the assumptions of Lemma 5.11, for any $i \geq 0$ and any B, B', μ, μ' with*

$$\langle C,\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto}{}^i \langle B,\ \tau,\ \mu\rangle \text{and} \langle C,\ \sigma',\ \_\rangle \overset{\varphi}{\longmapsto}{}^i \langle B',\ \tau',\ \mu'\rangle,$$

there is some $\rho \supseteq \pi$ such that $B \equiv B'$, $\mu = \mu'$, and

$$\begin{aligned}
&Lagree(\tau, \tau', \rho, (freshL(\sigma, \tau) \cup rlocs(\sigma, \varepsilon) \cup wrttn(\sigma, \tau))\backslash\{\text{alloc}\}),\\
&Lagree(\tau', \tau, \rho^{-1}, (freshL(\sigma', \tau') \cup rlocs(\sigma', \varepsilon) \cup wrttn(\sigma', \tau'))\backslash\{\text{alloc}\}),\\
&\rho(freshL(\sigma, \tau)) \subseteq freshL(\sigma', \tau'),\\
&\rho^{-1}(freshL(\sigma', \tau')) \subseteq freshL(\sigma, \tau).
\end{aligned}$$

This directly implies the conclusion of the Lemma.

The claim is proved by induction on i. The base case holds, because the fresh and written locations are empty, and agreement on $rlocs(\sigma, \varepsilon)$ is an assumption of the Lemma. For the induction step, suppose the above holds and consider the next steps:

$$\langle B,\ \tau,\ \mu\rangle \overset{\varphi}{\longmapsto} \langle D,\ \upsilon,\ \nu\rangle \text{ and } \langle B,\ \tau',\ \mu\rangle \overset{\varphi}{\longmapsto} \langle D',\ \upsilon',\ \nu'\rangle.$$

Go by cases on whether $Active(B)$ is a call.

Case $Active$(B) not a call. By judgment $\Phi \models^{\Gamma}_{M} C : P \rightsquigarrow Q\ [\varepsilon]$, the step from τ to υ respects $(\Phi, M, \varphi, \varepsilon, \sigma)$, as does the step from τ' to υ'. As this is not a call, the collective boundary is

$$\delta = (+N \in (\Phi, \mu), N \neq mod(B, M).\ bnd(N)).$$

So by w-respect for each step, we have $Agree(\tau, \upsilon, \delta)$ and $Agree(\tau', \upsilon', \delta)$.

We begin by proving the left-to-right agreement and inclusion for the induction step, i.e., we will find $\dot\rho$ such that $Lagree(\upsilon, \upsilon', \dot\rho, (freshL(\sigma, \upsilon) \cup rlocs(\sigma, \varepsilon) \cup wrttn(\sigma, \upsilon)) \backslash \{\mathsf{alloc}\})$ and $\dot\rho(freshL(\sigma, \upsilon)) \subseteq freshL(\sigma', \upsilon')$.

We will apply r-respect of the left step, instantiated with $\pi := \rho$ and with the right step. The two antecedents in r-respect are $Agree(\tau', \upsilon', \delta)$, which we have, and

$$Lagree(\tau, \tau', \rho, (freshL(\sigma, \tau) \cup rlocs(\sigma, \varepsilon)) \backslash rlocs(\tau, \delta^{\oplus})),$$

which follows directly from the induction hypothesis. So r-respect yields some $\dot\rho \supseteq \rho$ (and hence $\dot\rho \supseteq \pi$) with $D \equiv D'$, $\nu = \nu'$, and

$$\begin{aligned} &Lagree(\upsilon, \upsilon', \dot\rho, (freshL(\tau, \upsilon) \cup wrttn(\tau, \upsilon)) \backslash rlocs(\upsilon, \delta^{\oplus})),\\ &\dot\rho(freshL(\tau, \upsilon)) \subseteq freshL(\tau', \upsilon'). \end{aligned} \tag{40}$$

To conclude the left-to-right $Lagree$ part of the induction step it remains to show the two conditions

$$\begin{aligned} &Lagree(\upsilon, \upsilon', \dot\rho, rlocs(\sigma, \varepsilon) \backslash \{\mathsf{alloc}\}),\\ &Lagree(\upsilon, \upsilon', \dot\rho, (freshL(\tau, \upsilon) \cup wrttn(\tau, \upsilon)) \cap rlocs(\upsilon, \delta^{\oplus})). \end{aligned}$$

The latter holds because the intersection is empty, owing to $Agree(\tau, \upsilon, \delta)$ and $Agree(\tau', \upsilon', \delta)$ (noting that $rlocs(\upsilon, \delta) = rlocs(\tau, \delta)$ from those agreements and using Equation (28) and the requirement that boundaries have framed reads). For the same reasons, we have

$$Lagree(\upsilon, \upsilon', \dot\rho, rlocs(\sigma, \varepsilon) \cap rlocs(\upsilon, \delta)).$$

So it remains to show $Lagree(\upsilon, \upsilon', \dot\rho, rlocs(\sigma, \varepsilon) \backslash rlocs(\upsilon, \delta^{\oplus}))$. This we get by applying Lemma A.4, instantiated by $\pi, \rho := \rho, \dot\rho$ and $W := rlocs(\sigma, \varepsilon)$ (fortunately, the other identifiers in the Lemma are just what we need here). The antecedents of the Lemma include allowed dependencies and agreements that we have established above, and also the reverse of Equation (40), for $\dot\rho^{-1}$, which we get by symmetric arguments, using the reverse conditions in the induction hypothesis. The Lemma yields exactly what we need: $Lagree(\upsilon, \upsilon', \dot\rho, rlocs(\sigma, \varepsilon) \backslash rlocs(\upsilon, \delta^{\oplus})$.

Finally, we have $\dot\rho(freshL(\sigma, \upsilon)) = \rho(freshL(\sigma, \tau)) \cup \dot\rho(freshL(\tau, \upsilon)) \subseteq freshL(\sigma', \tau') \cup \dot\rho(freshL(\tau, \upsilon)) \subseteq freshL(\sigma', \tau') \cup freshL(\tau', \upsilon') = freshL(\sigma', \upsilon')$ by definitions, (40), and the induction hypothesis.

The reverse agreement and containment in the induction step is proved symmetrically.

Case $Active$(B) is a call. Let the method be m and suppose $\Phi(m) = R \rightsquigarrow S\ [\eta]$. By R-safe from the judgment $\Phi \models^{\Gamma}_{M} C : P \rightsquigarrow Q\ [\varepsilon]$, we have $rlocs(\tau, \eta) \subseteq rlocs(\sigma, \varepsilon) \cup freshL(\sigma, \tau)$. So by induction hypothesis, we have $Lagree(\tau, \tau', \rho, rlocs(\tau, \eta) \backslash \{\mathsf{alloc}\})$. So by $\varphi \models \Phi$ and Definition 5.9(d), there are two possibilities:

- $\varphi(m)(\tau) = \varnothing = \varphi(m)(\tau')$ and the steps both go by uCall0,
- $\varphi(m)(\tau) \neq \varnothing \neq \varphi(m)(\tau')$ and the steps both go by uCall.

In the first case, $D \equiv B \equiv D'$, $\nu = \mu = \nu'$, and the states are unchanged, so the agreements hold and we are done.

In the second case, we have $D \equiv B \equiv D'$, $\nu = \mu = \nu'$, $v \in \varphi(m)(\tau)$ and $v' \in \varphi(m)(\tau')$. Moreover, by Definition 5.9(d) there is some $\dot\rho \supseteq \rho$ such that

$$\begin{array}{l} \mathit{Lagree}(v, v', \dot\rho, (\mathit{freshL}(\tau, v) \cup \mathit{wrttn}(\tau, v))\backslash\{\mathrm{alloc}\}, \\ \dot\rho(\mathit{freshL}(\tau, v)) \subseteq \mathit{freshL}(\tau', v'). \end{array} \tag{41}$$

We also get reverse conditions, for $\dot\rho^{-1}$, by instantiating Definition 5.9(d) with ρ^{-1} and the states reversed. We must show

$$\begin{array}{l} \mathit{Lagree}(v, v', \dot\rho, (\mathit{freshL}(\sigma, v) \cup \mathit{rlocs}(\sigma, \varepsilon) \cup \mathit{wrttn}(\sigma, v))\backslash\{\mathrm{alloc}\}, \\ \dot\rho(\mathit{freshL}(\sigma, v)) \subseteq \mathit{freshL}(\sigma', v') \end{array}$$

(and the reverse, which is by a symmetric argument). We get $\dot\rho(\mathit{freshL}(\sigma, v)) \subseteq \mathit{freshL}(\sigma', v')$ using the induction hypothesis and Equation (41), similar to the proof above for the non-call case. For the *Lagree* condition for v, v', we have it for some locations by Equation (41). It remains to show v, v' agree via $\dot\rho$ on the locations $\mathit{freshL}(\sigma, \tau)$, $\mathit{rlocs}(\sigma, \varepsilon)\backslash\mathit{wrttn}(\tau, v)$, and $\mathit{wrttn}(\sigma, v)\backslash\mathit{wrttn}(\tau, v)$. The latter simplifies to $\mathit{wrttn}(\sigma, \tau)$, because $\mathit{wrttn}(\sigma, v) \subseteq \mathit{wrttn}(\sigma, \tau) \cup \mathit{wrttn}(\tau, v)$. We obtain the agreements by applying Lemma A.4 with $\delta := \bullet$, $\pi := \rho$, and $W := \mathit{freshL}(\sigma, \tau) \cup \mathit{rlocs}(\sigma, \varepsilon)\backslash\mathit{wrttn}(\tau, v) \cup \mathit{wrttn}(\sigma, \tau)$. To that end, observe that the above arguments have established $\tau, \tau' \overset{\rho}{\Rightarrow} v, v' \models^{\sigma}_{\bullet} \varepsilon$, and symmetric arguments establish $\tau', \tau \overset{\rho}{\Rightarrow} v', v \models^{\sigma'}_{\bullet} \varepsilon$. Moreover, we have the antecedent agreements and $\dot\rho$ as witness. So Lemma A.4 yields the requisite agreements and we are done.

Definition A.7 (Denotation of Command, $[\![\Gamma \vdash C]\!]$). Suppose C is wf in Γ and φ is a pre-model that includes all methods called in C and not bound by let in C. Define $[\![\Gamma \vdash C]\!]_\varphi$ to be the function of type $[\![\Gamma]\!] \to \mathbb{P}([\![\Gamma]\!] \cup \{\lightning\})$ given by

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

The denotation of a command can be used as a pre-model (Definition 5.7), owing to this easily-proved property of the transition semantics: if $\langle C,\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto}{}^* \langle D,\ \tau,\ \mu\rangle$ then $\sigma \hookrightarrow \tau$. We define a pre-model suited to be a context model, by taking into account a possible precondition: Given C, φ, formula R, and method name m not in $\mathit{dom}(\varphi)$ and not called in C, one can extend φ to $\dot\varphi$ that models m by

$$\dot\varphi(m)(\sigma) \;\hat{=}\; (\{\lightning\} \text{ if } \sigma \not\models R \text{ else } [\![\Gamma \vdash C]\!]_\varphi(\sigma)). \tag{42}$$

The outcome is empty in case C diverges. The conditions of Definition 5.7 hold owing to Lemma A.6, see corollaries (e) and (f) mentioned following that Lemma. (Note that $\sigma \not\models R$ means there is no extension of σ with values for spec-only variables in R that make it hold.)

Lemma A.8 (Context Model Denoted by Command). *Suppose $\Phi \models^{\Gamma}_{M} C : R \rightsquigarrow S\ [\eta]$ and $M = \mathit{mdl}(m)$. Suppose φ is a Φ-model. Let $\dot\Phi$ be Φ extended with $m : R \rightsquigarrow S[\eta]$, where $m \notin \mathit{dom}(\Phi)$ and m not called in C. Let $\dot\varphi$ be the extension given by Equation (42). If $N \in \Phi$ for all N with $\mathit{mdl}(m) \preceq N$, then $\dot\varphi$ is a $\dot\Phi$-model.*

Proof. To check $\dot\varphi(m)$ with respect to $R \rightsquigarrow S\ [\eta]$, observe that C does not fault (via φ) from states that satisfy R, by $\Phi \models_M C : R \rightsquigarrow S\ [\eta]$ and φ being a Φ-model. So, we get part (a) in Definition 5.9. Part (b) is an immediate consequence of $\Phi \models_M C : R \rightsquigarrow S\ [\eta]$. Part (c) requires boundary monotonicity for every N with $\mathit{mdl}(m) \preceq N$. Encap for the judgment gives monotonicity for every $N \in \Phi$ and also for M itself. We're done owing to hypothesis $N \in \Phi$ for every N with $M \prec N$. That condition is for single steps, but by simple induction on steps it implies $\mathit{rlocs}(\sigma, \delta) \subseteq$

$rlocs(\tau, \delta)$ for any τ such that $\langle C,\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle B,\ \tau,\ \mu\rangle$ for some B, μ. Part (d) is by application of Lemma 5.11. □

B UNARY LOGIC AND ITS SOUNDNESS (RE SECTION 6)

B.1 Additional Definitions and Proof Rules; Soundness Theorem

Figures 35 and 36 present the proof rules omitted from Figure 23. They are to be instantiated only with well-formed premises and conclusions. To emphasize the point, we make the following definitions. A correctness judgment is ***derivable*** iff it can be inferred using the proof rules instantiated with well-formed premises and conclusion. A proof rule is ***sound*** if for any instance with well-formed premises and conclusion, the conclusion is valid if the premises are valid and the side conditions hold.

Expression G is P/ε-***immune*** iff this is valid: $P \Rightarrow ftpt(G) \cdot/. \varepsilon$. Effect η is P/ε-***immune*** iff G is P/ε-immune for every G with $\mathrm{wr}\, G\text{`}f$ or $\mathrm{rd}\, G\text{`}f$ in η (see RLI). The key fact about immunity is that if η is P/ε-immune then

$$\sigma \models P \text{ and } \sigma{\rightarrow}\tau \models \varepsilon \text{ imply } rlocs(\sigma, \eta) = rlocs(\tau, \eta) \text{ and } wlocs(\sigma, \eta) = wlocs(\tau, \eta). \tag{43}$$

Definition B.1 (Boundary Monotonicity Spec). $BndMonSp(P, \varepsilon, M)$ is $P \wedge Bsnap_M \rightsquigarrow Bmon_M\ [\varepsilon]$ where $Bsnap_M$ and $Bmon_M$ are defined as follows. Let δ be $bnd(M)$, normalized so that for each field f for which $\mathrm{rd}\, H\text{`}f$ occurs in $bnd(M)$ for some $H$, there a single region expression $G_f$ with $\mathrm{rd}\, G_f\text{`}f$ in δ. Let $Bsnap_M$ (for "boundary snap") be the conjunction over fields f of formulas $s_f = G_f$ where each s_f is a fresh spec-only variable. Let $Bmon_M$ be the conjunction over fields f of formulas $s_f \subseteq G_f$.

Remark 6. In case boundaries are empty, the postcondition becomes vacuously true. As a result, the second premises in rules ModIntro and CtxIntroCall, for boundary monotonicity, become trivial consequences of the main premises.

Remark 7. The syntax directed rules in Figure 35 are very similar to the unary proof rules in RLIII. Other than addition of modules, one noticeable difference is that in RLIII rules Seq and While require the effects to be read framed. This is not needed with the current definition of valid judgment, which imposes a stronger condition for read effects (Definition 5.10).

Remark 8. Recall that rule CtxIntro (Figure 23) allows the introduction of additional modules, by adding methods to the hypothesis context (see Section 6.3). It has side conditions that ensure encapsulation. For method calls, CtxIntro is useful to add context that is not imported by the method's module. A separate rule, CtxIntroCall, is needed to add context that is imported by the method's module (as it was in RLII). To add a method of the current module to the context, rule CtxIntroIn2 is used if the judgment is for a non-call; otherwise CtxIntroCall is used. To add a method to the context for a module already present in context, rule CtxIntroIn1 is used. The context intro rules are not applicable to control structures, so requisite context should be introduced for their constituents before their proof rules are used.

The axioms for atomic commands (e.g., Alloc in Figure 23) are for the default module $\bullet$ and the empty context, or in the case of Call the context with just the called method. Rule ModIntro changes the current module from $\bullet$ to another one; this is not needed in RLII, because its main significance is to enforce boundary monotonicity (Definition 5.10), which is not needed in RLII. For non-call atomic commands, the rule needs to be used before introducing methods of the current module into the context.

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

$$\textsc{Assign}\ \frac{y \not\equiv x}{\vdash_\bullet x := F :\ x = y \rightsquigarrow x = F^x_y\ [\mathsf{wr}\, x, \mathit{ftpt}(F)]}$$

$$\textsc{Seq}\ \frac{\Phi \vdash_M C_1 :\ P \rightsquigarrow P_1\ [\varepsilon_1] \qquad \Phi \vdash_M C_2 :\ P_1 \rightsquigarrow Q\ [\varepsilon_2, \mathsf{rw}\, H\text{`}\overline{f}\text{`}] \qquad P_1 \Rightarrow H\#r \qquad \varepsilon_2 \text{ is } P/\varepsilon_1\text{-immune} \qquad \mathit{spec\text{-}only}(r)}{\Phi \vdash_M C_1; C_2 :\ P \wedge r = \mathsf{alloc} \rightsquigarrow Q\ [\varepsilon_1, \varepsilon_2]}$$

$$\textsc{While}\ \frac{\Phi \vdash_M C :\ P \wedge E \rightsquigarrow P\ [\varepsilon, \mathsf{rw}\, H\text{`}\overline{f}\text{`}] \qquad \varepsilon \text{ is } P/(\varepsilon, \mathsf{wr}\, H\text{`}\overline{f}\text{`})\text{-immune} \qquad P \Rightarrow H\#r \qquad (+N \in \Phi, N \neq M.\ \mathit{bnd}(N)) \cdot\!/.\ \mathit{r2w}(\mathit{ftpt}(E)) \qquad \mathit{spec\text{-}only}(r)}{\Phi \vdash_M \mathsf{while}\ E\ \mathsf{do}\ C :\ P \wedge r = \mathsf{alloc} \rightsquigarrow P \wedge \neg E\ [\varepsilon, \mathit{ftptE}]}$$

Fig. 35. Syntax-directed proof rules not given in Figure 23.

$$\textsc{ModIntro}\ \frac{\Phi \vdash_\bullet A : P \rightsquigarrow Q\ [\varepsilon] \qquad \Phi \vdash_\bullet A : P \wedge \mathit{Bsnap}_M \rightsquigarrow \mathit{Bmon}_M\ [\varepsilon] \qquad \text{if } M \in \Phi \text{ then } A \text{ is a call}}{\Phi \vdash_M A : P \rightsquigarrow Q\ [\varepsilon]}$$

$$\textsc{CtxIntroIn2}\ \frac{\Phi \vdash_M A : P \rightsquigarrow Q\ [\varepsilon] \qquad \mathit{mdl}(m) = M \qquad A \text{ is not a call}}{\Phi, m{:}R \rightsquigarrow S\ [\eta] \vdash_M A :\ P \rightsquigarrow Q\ [\varepsilon]}$$

$$\textsc{CtxIntroCall}\ \frac{\Phi \vdash_M p() : P \rightsquigarrow Q\ [\varepsilon] \qquad \Phi \vdash_M p() : P \wedge \mathit{Bsnap}_N \rightsquigarrow \mathit{Bmon}_N\ [\varepsilon] \qquad N = \mathit{mdl}(m) \qquad \mathit{mdl}(p) \preceq \mathit{mdl}(m)}{\Phi, m{:}R \rightsquigarrow S\ [\eta] \vdash_M p() :\ P \rightsquigarrow Q\ [\varepsilon]}$$

$$\textsc{Conj}\ \frac{\Phi \vdash_M C :\ P \rightsquigarrow Q_0\ [\varepsilon] \qquad \Phi \vdash_M C :\ P \rightsquigarrow Q_1\ [\varepsilon]}{\Phi \vdash_M C :\ P \rightsquigarrow Q_0 \wedge Q_1\ [\varepsilon]}$$

$$\textsc{Disj}\ \frac{\Phi \vdash_M C :\ P_0 \rightsquigarrow Q\ [\varepsilon] \qquad \Phi \vdash_M C :\ P_1 \rightsquigarrow Q\ [\varepsilon]}{\Phi \vdash_M C :\ P_0 \vee P_1 \rightsquigarrow Q\ [\varepsilon]} \qquad \textsc{Exist}\ \frac{\Phi \vdash_M^{\Gamma, x:T} C :\ P \rightsquigarrow Q\ [\varepsilon]}{\Phi \vdash_M^{\Gamma} C :\ (\exists x : T.\ P) \rightsquigarrow Q\ [\varepsilon]}$$

Fig. 36. Structural proof rules not given in Figure 23.

Some of the rules use a second premise, the boundary monotonicity spec of Definition B.1, to enforce boundary monotonicity.[43] In many cases, this judgment can be derived from the primary judgment of the rule, by a simple use of the Frame rule to get *Bsnap* in the postcondition, and then Conseq to get *Bmon*.

Theorem 6.1 (Soundness of Unary Logic). *All the unary proof rules are sound (Figure 23 and Appendix Figures 35 and 36).*

The proofs comprise Appendices B.2–B.10. We prove the R-safe and Encap conditions for all rules, since Encap differs from the definition in RLII and R-safe is a new addition. Otherwise, the

[43] One can contrive a rule with only one premise, subject to conditions that ensure it refines the second spec, but we prefer this way.

proofs are mostly as in RLII. We give full proofs for the rules that have significantly changed from RLII,RLIII, e.g., CtxIntro and SOF.

B.2 Soundness of Call

To show soundness of the axiom $m : P \rightsquigarrow Q\ [\varepsilon] \vdash_\bullet m() : P \rightsquigarrow Q\ [\varepsilon]$, consider any σ with $\hat{\sigma} \models P$ where $\hat{\sigma} \triangleq [\sigma{+}\overline{s}{:}\,\overline{v}]$ and $\overline{s}$ are the spec-only variables of P. Consider any φ that is an $(m : P \rightsquigarrow Q\ [\varepsilon])$-model. Owing to $\hat{\sigma} \models P$ and Definition 5.9 of context model, there is no faulting transition. So either $\varphi(m)(\sigma)$ is empty and the stuttering transition is taken (transition rule uCall0), or execution terminates in a single step $\langle m(), \sigma, \_\rangle \overset{\varphi}{\longmapsto} \langle \mathsf{skip}, \tau, \_\rangle$ with $\tau \in \varphi(m)(\sigma)$ (transition rule uCall). The stuttering transition repeats indefinitely, and Safety, Post, Write, R-safe, and Encap all hold, because the configuration never changes. In case execution terminates in $\langle \mathsf{skip},\ \tau,\ \_\rangle$, Safety, Post, and Write are immediate from Definition 5.9, which in particular says $\hat{\tau} \models Q$ where $\tau \triangleq [\tau{+}\overline{s}{:}\,\overline{v}]$. For R-safe, there is only one configuration that is a call, the initial one, and it is r-safe, because the frame condition in the judgment is exactly the frame condition of the method's spec.

Encap requires boundary monotonicity for the current module and every module in context. Boundary monotonicity for module $\bullet$ holds, because $bnd(\bullet) = \bullet$. It holds for $mdl(m)$, the one module in context, by Definition 5.9(c), since $\preceq$ is reflexive.

Encap requires w-respect for every N in context different from the current module, which in this case means either $mdl(m)$ or nothing, depending whether $mdl(m) = \bullet$. The step w-respects $mdl(m)$, because it is a call and $mdl(m) \preceq mdl(m)$.

Encap considers σ', π such that $Lagree(\sigma, \sigma', \pi, rlocs(\sigma, \eta) \backslash rlocs(\sigma, \delta^{\oplus}))$ where collective boundary δ is the union of boundaries for N in context and not imported by $mdl(m)$; hence $\delta = \bullet$. By condition (d) in Definition 5.9, we have $\varphi(m)(\sigma) = \emptyset$ iff $\varphi(m)(\sigma') = \emptyset$, so either both transition go via uCall0 to unchanged states, thus satisfying r-respect, or both transition go via uCall to states τ, τ' with $\tau \in \varphi(m)(\sigma)$ and $\tau' \in \varphi(m)(\sigma')$. In the latter case, $rlocs(\sigma, \bullet)^{\oplus}$ is $\{\mathsf{alloc}\}$ by definition of *rlocs*, and the r-respect condition to be proved is exactly the condition (d) in Definition 5.9. In a little more detail, we must show the final states agree on $freshL(\sigma, \tau) \cup wrttn(\sigma, \tau) \backslash rlocs(\tau, \bullet^{\oplus})$, which simplifies to $freshL(\sigma, \tau) \cup wrttn(\sigma, \tau) \backslash \{\mathsf{alloc}\}$. R-respects also requires a condition that simplies to $\rho(freshL(\sigma, \tau)) \subseteq freshL(\sigma', \tau')$, because $rlocs(\tau, \bullet) = \emptyset$.

B.3 Soundness of FieldUpd

This is an axiom: $\vdash_\bullet x.f := y :\ x \neq \mathsf{null} \rightsquigarrow x.f = y\ [\mathsf{wr}\, x.f, \mathsf{rd}\, x, \mathsf{rd}\, y]$. The Safety, Post, and Write conditions are straightforward and proved the same way as in RLI. R-safe holds because there is no method call. For Encap, the only steps to consider are the single terminating steps from states where x is not null. So suppose $\langle x.f := e,\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto} \langle \mathsf{skip},\ v,\ \_\rangle$, where $v = [\sigma \mid \sigma(x).f : \sigma(y)]$. For Encap, boundary monotonicity: the only relevant boundary is $bnd(\bullet)$, which is empty, so monotonicity holds vacuously. For Encap, w-respect is vacuously true for the empty boundary. For r-respect, since the command is not a call the collective boundary is empty. As we are considering the initial step and the boundary is empty, the antecedent of r-respect can be written

$$Lagree(\sigma, \sigma', \pi, rlocs(\sigma, \varepsilon) \backslash \{\mathsf{alloc}\}) \text{ and } \langle x.f := e,\ \sigma',\ \_\rangle \overset{\varphi}{\longmapsto} \langle \mathsf{skip},\ v',\ \_\rangle. \tag{44}$$

Since there is no allocation, extending π is not relevant, and the condition about fresh locations is vacuous, so it remains to show that $Lagree(v, v', \pi, (wrttn(\sigma, v)) \backslash \{\mathsf{alloc}\})$. What is written is the location $\sigma(x).f$, so this simplifies to $Lagree(v, v', \pi, \{\sigma(x).f\})$. Given that $\mathsf{rd}\, x$ is in the frame condition, we have $x \in rlocs(\sigma, \varepsilon)$ so the assumption Equation (44) gives agreement on which location is written. It remains to show agreement on the value written, which is $\sigma(y)$ versus $\sigma'(y)$. From the frame condition, we have $y \in rlocs(\sigma, \varepsilon)$, so by Equation (44), we have initial agreement on it and we are done.

B.4 Soundness of If

Suppose the premises are valid: $\Phi \models_M C_1 : P \wedge E \rightsquigarrow Q\ [\varepsilon]$ and $\Phi \models_M C_2 : P \wedge \neg E \rightsquigarrow Q\ [\varepsilon]$. Suppose the side condition is valid: $(+N \in \Phi, N \neq M.\ bnd(N)) \cdot\!/.\ r2w(ftpt(E))$. To show $\Phi \vdash_M$ if E then C_1 else $C_2 : P \rightsquigarrow Q\ [\varepsilon, ftpt(E)]$, we only consider R-safe and Encap, because the rest is straightforward and similar to previously published proofs. Consider any Φ-model φ, noting that the premises have the same context. Consider and any σ with $\sigma \models P$. Consider the case that $\sigma(E) = true$ (the other case being symmetric). So the first step is $\langle \text{if } E \text{ then } C_1 \text{ else } C_2,\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto} \langle C_1,\ \sigma,\ \_\rangle$. This is not a call, so the step (or rather, its starting configuration) satisfies r-safe. For Encap, the first step does not write, so it satisfies boundary monotonicity and w-respect.

For r-respect, the requisite collective boundary is $\delta = (+N \in (\Phi, N \neq M.\ bnd(N))$, because there is no ecall and the environment is empty. We show r-respect for the first step, i.e., instantiating r-respect with $\tau, \upsilon := \sigma, \sigma$. The requisite condition for this step is that for any σ', if

$$\langle \text{if } E \text{ then } C_1 \text{ else } C_2,\ \sigma',\ \_\rangle \overset{\varphi}{\longmapsto} \langle D',\ \sigma',\ \_\rangle$$

and $Lagree(\sigma, \sigma', \pi, (freshL(\sigma, \sigma) \cup rlocs([\sigma{+}\overline{s}{:}\,\overline{v}], (\varepsilon, ftpt(E)))\backslash rlocs(\sigma, \delta^{\oplus}))$, then $D' \equiv C_1$ and two agreement conditions about fresh and written locations. (We omitted one antecedent, $Agree(\sigma', \sigma', \delta)$, which is vacuous.) There are no fresh or written locations, so those two conditions hold. It remains to prove $D' \equiv C_1$. We can simplify the antecedent to

$$Lagree(\sigma, \sigma', \pi, (rlocs(\sigma, (\varepsilon, ftpt(E)))\backslash rlocs(\sigma, \delta^{\oplus}))).$$

Because the side condition is true, $(+N \in \Phi, N \neq M.\ bnd(N)) \cdot\!/.\ r2w(ftpt(E))$, we have $rlocs(\sigma, ftpt(E))$ disjoint from $rlocs(\sigma, \delta^{\oplus})$. So $Lagree(\sigma, \sigma', \pi, (rlocs(\sigma, (\varepsilon, ftpt(E)))\backslash rlocs(\sigma, \delta^{\oplus})))$ implies $Lagree(\sigma, \sigma', \pi, rlocs(\sigma, ftpt(E)))$. Hence, $\sigma(E) = \sigma'(E)$ by footprint agreement lemma. By semantics, $D' \equiv C_1$ and we are done.

For subsequent steps in the case $\sigma(E) = true$, we can appeal to the premise for C_1, which applies to the trace starting from $\langle C_1,\ \sigma,\ \_\rangle$, since $\sigma \models P \wedge E$. This yields r-safe and respect (as well as the other conditions for validity).

B.5 Soundness of Var

Suppose the premise is valid: $\Phi \models_M^{\Gamma, x:T} C :\ P \wedge x = default(T) \rightsquigarrow P'\ [\text{rw}\, x, \varepsilon]$. To prove the R-safe and Encap conditions for $\Phi \models_M^{\Gamma} \text{var } x{:}T \text{ in } C :\ P \rightsquigarrow P'\ [\varepsilon]$, let φ be a Φ-model and $\hat{\sigma} \models P$ (where $\hat{\sigma}$ extends σ with values for the spec-only variables of P). The first step is $\langle \text{var } x{:}T \text{ in } C,\ \sigma,\ \mu\rangle \overset{\varphi}{\longmapsto} \langle C^x_{x'}; \text{evar}(x'),\ [\sigma{+}x'{:}\,default(T)],\ \mu\rangle$ where $x' = FreshVar(\sigma)$. Let $\delta = (+N \in \Phi, N \neq M.\ bnd(N))$. This step satisfies w-respect, because the variables in δ are already in scope, so are distinct from x'. (Indeed, x' is a local variable and boundaries cannot contain locals.) The first configuration satisfies r-safe, because it is not a call. To show the first step satisfies r-respect, note first that $rlocs(\sigma, \delta) = rlocs([\sigma{+}x'{:}\,default(T)], \delta)$, again, because x' is not in δ. Consider taking the first step from an alternate state σ' satisfying the requisite agreements with σ. Now σ' has the same variables as σ (by definition of r-respect, including footnote 32), and by assumption (39) the choice of x' depends only on the domain of σ, so the alternate step introduces the same local x' and the same command $C^x_{x'}; \text{evar}(x')$. We have $freshL(\sigma, [\sigma{+}x'{:}\,default(T)]) = \{x'\}$ by definition, and the agreements for r-respect follow directly, noting that $default(T)$ is a fixed value dependent only on the type T.

If execution reaches the last step, then that last step satisfies r-safe and respects, because it merely removes x' from the state. For any other step, the result follows straightforwardly from R-safe and Encap for the premise: The state $[\sigma{+}x'{:}\,default(T)])$ satisfies $P \wedge x = default(T)$, and a

trace of $C^{x}_{x'}$; evar(x') gives rise to a trace of C (by dropping evar(x') and renaming), for which the premise yields r-safe, respects, indeed Safety, and so on.

B.6 Soundness of ModIntro

$$\textsc{ModIntro}\ \frac{\Phi \vdash_\bullet A : P \rightsquigarrow Q\ [\varepsilon] \qquad \Phi \vdash_\bullet A : P \wedge Bsnap_M \rightsquigarrow Bmon_M\ [\varepsilon] \qquad \text{if } M \in \Phi \text{ then } A \text{ is a call}}{\Phi \vdash_M A : P \rightsquigarrow Q\ [\varepsilon]}$$

For Encap, as A is an atomic command A, the only reachable step is the single step taken in a terminating execution $\langle A,\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto} \langle \mathsf{skip},\ \tau,\ \_\rangle$ or the stutter step by uCall0, which has the form $\langle A,\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto} \langle A,\ \sigma,\ \_\rangle$. (A stutter step may repeat, but no other state is reached.) In either case, there is no ecall in the configuration, and the environment is empty.

For Encap, boundary monotonicity for $N \in \Phi$ is from the first premise, and boundary monotonicity for $N = M$ is from the second premise.

For Encap, the w-respect condition quantifies over $N \in (\Phi, \_)$ different from the $mod(A, M)$. Since the environment is empty, $N \in (\Phi, \_)$ is the same as $N \in \Phi$. Since A has no ecall, $mod(A, M)$ is M. So the condition quantifies over $N \in \Phi$ with $N \neq M$. By side condition $M \notin \Phi$, this is the same as $N \in \Phi$. So the condition for the conclusion is the same as for the first premise, from which we obtain Encap (a).

For Encap r-respect, go by cases whether A is a method call. If not, then the collective boundary for the premise is $(+N, N \in (\Phi, \_), N \neq mod(A, \bullet).\ bnd(N))$, and for the conclusion it is $(+N, N \in (\Phi, \_), N \neq mod(A, M).\ bnd(N))$. These are the same, owing to side condition $M \notin \Phi$, and simplifying as above. So r-respect is immediate by the first premise.

If A is a call to some method p, then the collective boundary is $(+N, N \in (\Phi, \_), mdl(p) \not\leq N.\ bnd(N))$. This is independent of the current module, so again the conclusion is direct from the first premise.

B.7 Soundness of CtxIntro

$$\textsc{CtxIntro}\ \frac{\Phi \vdash_M A :\ P \rightsquigarrow Q\ [\varepsilon] \qquad P \Rightarrow bnd(mdl(m))\ \cdot\!/.\ \varepsilon \qquad P \Rightarrow bnd(mdl(m))\ \cdot\!/.\ r2w(\varepsilon)}{\Phi,\ m : R \rightsquigarrow S\ [\eta] \vdash_M A :\ P \rightsquigarrow Q\ [\varepsilon]}$$

Proof. Consider any $(\Phi, m{:}R \rightsquigarrow S\ [\eta])$-model φ. By definitions, $\varphi \restriction m$ is a Φ-model, with which we can instantiate the premise. The Safety, Post, Write, and R-safe conditions follow from those for the premise—it is only the Encap condition that has a different meaning for the conclusion than it does for the premise.

For Encap, as A is an atomic command A, the only reachable step is a single step, either the terminating step $\langle A,\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto} \langle \mathsf{skip},\ \tau,\ \_\rangle$ given by uCall or the stuttering step by uCall0, which is $\langle A,\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto} \langle A,\ \tau,\ \_\rangle$ with $\tau = \sigma$.

For Encap, for boundary monotonicity, we need $rlocs(\sigma, bnd(N)) \subseteq rlocs(\tau, bnd(N))$ for all N with $N \in (\Phi, m : R \rightsquigarrow S\,[\eta])$ or $N = M$. This holds for all $N \in \Phi$, and for $N = M$, by the same condition from the premise, so it remains to consider $N = mdl(m)$. From the premise, we have $\sigma{\rightarrow}\tau \models \varepsilon$. By side condition (and $\sigma \models P$), we have $\sigma \models bnd(N)\ \cdot\!/.\ \varepsilon$. So, we have $Agree(\sigma, \tau, bnd(N)$ by separator property (29). Since boundaries are read framed (Definition 3.1), we can apply footprint agreement (28) to get $rlocs(v, bnd(N)) = rlocs(\tau, bnd(N))$.

For Encap, we need w-respect of each N with $N \in (\Phi, m : R \rightsquigarrow S\ [\eta])$ and $N \neq mod(A, M)$. (simplified for the empty environment, as in the proof of ModIntro). Since ecall does not occur

in A, $N \neq mod(A, M)$ simplifies to $N \neq M$. Again, we have this condition from the premise for all N except $N = mdl(m)$. For that, in the case that A is not a call to a method m with $mdl(m) \preceq N$, we must show $Agree(\sigma, \tau, bnd(N))$; and it was shown already in the proof of (c).

For Encap, we show r-respect by cases:

Case: the step is not a call. Then the collective boundary is $\delta = (+N \in (\Phi, m : R \rightsquigarrow S\ [\eta]), N \neq mod(A, M).\ bnd(N))$, and $N \neq mod(A, M)$ is just $N \neq M$.

Let $\dot{\delta}$ be the collective boundary for the premise: $\dot{\delta} = (+N \in \Phi, N \neq M.\ bnd(N))$ (again, simplifying $N \neq mod(A, M)$ to $N \neq M$). So δ is $\dot{\delta}, bnd(N)$. If $N = M$, or $N \in \Phi$, or $bnd(N) = \bullet$, then $\dot{\delta}$ is equivalent to δ, and we get r-respect directly from the premise. Otherwise, suppose $\langle A,\ \sigma',\ \_\rangle \overset{\varphi}{\longmapsto} \langle B,\ \tau',\ \_\rangle$ and $Agree(\sigma', \tau', \delta)$ and

$$Lagree(\sigma, \sigma', \pi, rlocs(\sigma, \varepsilon) \backslash rlocs(\sigma, \delta^{\oplus})). \tag{45}$$

(This is simplified from the general condition of r-respect, which includes fresh locations in the assumed agreement; here, because we consider the first step of computation, there are none.) We must show

$$\begin{array}{l} Lagree(\tau, \tau', \rho, (freshL(\sigma, \tau) \cup wrttn(\sigma, \tau)) \backslash rlocs(\tau, \delta^{\oplus})), \\ \rho(freshL(\sigma, \tau) \backslash rlocs(\tau, \delta)) \subseteq freshL(\sigma', \tau') \backslash rlocs(\tau', \delta). \end{array} \tag{46}$$

The premise gives an implication similar to (45)⇒(46) but for $\dot{\delta}$. Now $\dot{\delta}$ may be a proper subeffect of δ, so we only have $rlocs(\sigma, \dot{\delta}) \subseteq rlocs(\sigma, \delta)$ and thus $rlocs(\sigma, \varepsilon) \backslash rlocs(\sigma, \delta^{\oplus})$ may be a proper subset of $rlocs(\sigma, \varepsilon) \backslash rlocs(\sigma, \dot{\delta}^{\oplus})$. This means Equation (45) does not imply the antecedent in r-respects for the premise, so we cannot simply apply that. Instead, we exploit the fact that the command A is one of the assignment forms: $x := F$, $x :=$ new K, $x := x.f$, $x.f := x$. Each of these has a minimal set of locations on which it depends in the relevant sense.

Claim. *For each of the atomic, non-call commands, and for each $\sigma, \sigma', \mu, \mu'$, there is a finite number of minimal sets $X \subseteq locations(\sigma)$ such that if $\langle A, \sigma, \mu\rangle \longmapsto \langle \text{skip}, \tau, \mu\rangle$, $\langle A, \sigma', \mu\rangle \longmapsto \langle \text{skip}, \tau', \mu\rangle$, and $Lagree(\sigma, \sigma', \pi, X)$, then there is $\rho \supseteq \pi$ with*

$$Lagree(\tau, \tau', \rho, freshL(\sigma, \tau) \cup wrttn(\sigma, \tau)) \text{ and } \rho(freshL(\sigma, \tau)) \subseteq freshL(\sigma', \tau').$$

(Here, we omit the model for $\longmapsto$, which is not relevant to semantics of non-call atomics.) In fact, the minimal sets are unique in most cases, but we do not need that.[44]

Now, consider the antecedent of r-respect for the premise: $Lagree(\sigma, \sigma', \pi, rlocs(\sigma, \varepsilon) \backslash rlocs(\sigma, \dot{\delta}^{\oplus}))$. We must have $X \subseteq rlocs(\sigma, \varepsilon) \backslash rlocs(\sigma, \dot{\delta}^{\oplus})$, as otherwise, according to the Claim, r-respect would not hold for the premise. By side condition, we have $\hat{\sigma} \models bnd(mdl(m)) \cdot /. r2w(\varepsilon)$, hence $rlocs(\sigma, bnd(N))$ is disjoint from $rlocs(\sigma, \varepsilon)$ by the basic separator property mentioned just before (29). By set theory, from $X \subseteq rlocs(\sigma, \varepsilon) \backslash rlocs(\sigma, \dot{\delta}^{\oplus})$ and $rlocs(\sigma, bnd(N)) \cap rlocs(\sigma, \varepsilon) = \varnothing$ we get $X \subseteq rlocs(\sigma, \varepsilon) \backslash rlocs(\sigma, \delta^{\oplus})$. By monotonicity of *Lagree*, Equation (21), the agreement Equation (45) implies by $X \subseteq rlocs(\sigma, \varepsilon) \backslash rlocs(\sigma, \delta^{\oplus})$ the antecedent agreement in the Claim. Whence by the Claim we get agreement on everything fresh and written, which implies the agreement in Equation (46). As for the second line of Equation (46), what the Claim gives is $\rho(freshL(\sigma, \tau)) \subseteq freshL(\sigma', \tau')$. This implies $\rho(freshL(\sigma, \tau) \backslash rlocs(\tau, \delta)) \subseteq freshL(\sigma', \tau')$. From $Agree(\sigma', \tau', \delta)$, we have $rlocs(\tau', \delta) = rlocs(\sigma', \delta$ so there are no fresh locations in $rlocs(\tau', \delta)$. Hence, $freshL(\sigma', \tau') = freshL(\sigma', \tau') \backslash rlocs(\tau', \delta)$, so we have $\rho(freshL(\sigma, \tau) \backslash rlocs(\tau, \delta)) \subseteq freshL(\sigma', \tau') \backslash rlocs(\tau', \delta)$, and we are done.

[44] It is only assignments $x := F$ for which non-uniqueness is possible, owing to information loss in arithmetic expressions. For example, with the assignment $x := y * z$ and for σ with $\sigma(y) = 0 = \sigma(z)$ then agreement on either y or z is enough to ensure the values written to x agree. The minimal sets are $\{y\}$ and $\{z\}$. This also happens with conditional branches, like "if x or y."

The Claim is a straightforward property of the semantics. For each of the assignment forms, one defines the evident location set (which underlies the small axioms in the proof system) and shows that it suffices for the final agreement. Then by counterexamples one shows that the location set is minimal.

Case: the step is a call. We show r-respect in the case that A is a call to some method p. Note that $p \neq m$, because rules can only be instantiated by wf judgments and m is not in scope in the premise. The primary step has the form $\langle p(),\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto} \langle A_0,\ \tau,\ \_\rangle$, where either $A_0 \equiv \mathsf{skip}$ and $\tau \in \varphi(p)(\sigma)$ or $A_0 \equiv p()$, $\tau = \sigma$, and $\varphi(p)(\sigma) = \emptyset$. It turns out that we do not need to distinguish between these cases. We need r-respect for

$$\delta = (+N \in (\Phi, m{:}R \rightsquigarrow S\ [\eta]),\ mdl(p) \not\preceq N.\ bnd(N))$$

(as the environment is empty). The premise gives r-respect for $\dot\delta = (+N \in \Phi,\ mdl(p) \not\preceq N.\ bnd(N))$. If $mdl(m) \in \Phi$ or $mdl(p) \preceq mdl(m)$, then δ is $\dot\delta$, and we have r-respect from the premise. It remains to consider the case that $mdl(m) \notin \Phi$ and $mdl(p) \not\preceq mdl(m)$, in which case $\delta = \dot\delta, bnd(mdl(m))$. Let us spell out r-respect for the premise and this step. The r-respect from the premise says that

$$Lagree(\sigma, \sigma', \pi, rlocs(\sigma, \varepsilon)\backslash rlocs(\sigma, \dot\delta^{\oplus})) \text{ and } Agree(\sigma', \tau', \delta) \tag{47}$$

implies there is ρ with $\rho \supseteq \pi$, such that $Lagree(\tau, \tau', \rho, (freshL(\sigma, \tau) \cup wrttn(\sigma, \tau))\backslash rlocs(\tau, \dot\delta^{\oplus}))$ and $\rho(freshL(\sigma, \tau)\backslash rlocs(\tau, \dot\delta)) \subseteq freshL(\sigma', \tau')\backslash rlocs(\tau', \dot\delta)$. (The antecedent is simplified from the definition of r-respect, by omitting the set of fresh locations, which is empty in the initial state.)

For the conclusion, the condition is the same except with δ in place of $\dot\delta$. So suppose

$$Lagree(\sigma, \sigma', \pi, rlocs(\sigma, \varepsilon)\backslash rlocs(\sigma, \delta^{\oplus})).$$

This implies Equation (47), because $rlocs(\sigma, \varepsilon)$ is disjoint from $bnd(mdl(m))$ owing to the condition $bnd(mdl(p))\ \cdot\!/.\ \varepsilon$ in the rule. So, we get some ρ as above, and the agreement $Lagree(\tau, \tau', \rho, (freshL(\sigma, \tau) \cup wrttn(\sigma, \tau))\backslash rlocs(\tau, \dot\delta^{\oplus}))$ implies the needed agreement for δ, since $\dot\delta$ is a subeffect of δ, which is being subtracted. Finally, we need to show $\rho(freshL(\sigma, \tau)\backslash rlocs(\tau, \delta)) \subseteq freshL(\sigma', \tau')\backslash rlocs(\tau, \delta)$. By w-respect for the σ-to-τ step and by assumption $Agree(\sigma', \tau', \delta)$, there are no fresh locations in $rlocs(\tau, \delta)$ or $rlocs(\tau', \delta)$, so this simplifies to $\rho(freshL(\sigma, \tau) \subseteq freshL(\sigma', \tau')$, which for the same reasons is equivalent to the inclusion $\rho(freshL(\sigma, \tau)\backslash rlocs(\tau, \dot\delta)) \subseteq freshL(\sigma', \tau')\backslash rlocs(\tau', \dot\delta)$ from the premise. □

B.8 Soundness of other Context Introduction Rules

In RLII the rule "CtxIntroIn" has a disjunctive antecedent. In the present work, we need additional side conditions, so we split the rule into multiple rules:

$$\textsc{CtxIntroIn1}\ \frac{\Phi \vdash_M C :\ P \rightsquigarrow Q\ [\varepsilon] \qquad mdl(m) \in \Phi}{\Phi, m{:}R \rightsquigarrow S\ [\eta] \vdash_M C :\ P \rightsquigarrow Q\ [\varepsilon]}$$

Proof. Given a model φ for the conclusion, $\varphi \restriction m$ is a model for the hypotheses of the premise. Owing to $mdl(m) \in \Phi$, we have $N \in (\Phi, m : spec)$ iff $N \in \Phi$. As a result, all the conditions of Encap (a–c) are have identical meaning for the conclusion as for the premise. The same is true for Safety, Post, Write, and R-safe: □

$$\textsc{CtxIntroIn2}\ \frac{\Phi \vdash_M A : P \rightsquigarrow Q\ [\varepsilon] \qquad mdl(m) = M \qquad A \text{ is not a call}}{\Phi, m{:}R \rightsquigarrow S\ [\eta] \vdash_M A :\ P \rightsquigarrow Q\ [\varepsilon]}$$

Proof. Note that A is an atomic command. Given a model φ for the conclusion, $\varphi\restriction m$ is an model for the hypotheses of the premise. Validity of the premise implies validity of the conclusion, for all conditions except Encap. Boundary monotonicity is immediate, because the premise already requires boundary monotonicity for all $N \in \Phi$ and for $N = M$. For w-respect, note that A is not a call and there is only a single step that has no ecall in the configuration. The condition exempts the current module M and is a direct consequence of Encap (a) of the premise, owing to $mdl(m) = M$. For r-respect, the current module is not included in the collective boundary for non-call commands, so again the addition of m does not change the requirement. □

$$\textsc{CtxIntroCall}\ \frac{\Phi \vdash_M p() : P \rightsquigarrow Q\ [\varepsilon] \qquad \Phi \vdash_M p() : P \wedge Bsnap_N \rightsquigarrow Bmon_N\ [\varepsilon] \qquad N = mdl(m) \qquad mdl(p) \preceq mdl(m)}{\Phi, m{:}R \rightsquigarrow S\ [\eta] \vdash_M p() :\ P \rightsquigarrow Q\ [\varepsilon]}$$

Proof. We get Safety, Post, Write, and R-safe from the first premise. For Encap, we get boundary monotonicity from the first premise, except for N in the case that $N = mdl(m) \neq M$ and $mdl(m) \notin \Phi$. Boundary monotonicity for N is directly checked by the second premise.

We get w-respect, by side condition $mdl(p) \preceq mdl(m)$, as a consequence of the first premise.

Finally, r-respect is also a consequence of the first premise, because the collective boundary for the premise is $(+N \in \Phi, mdl(p) \not\preceq N.\ bnd(N))$ and by side condition $mdl(p) \preceq mdl(m)$ this is the same set as for the conclusion. □

B.9 Soundness of SOF

$$\textsc{SOF}\ \frac{\Phi, \Theta \vdash_M C :\ P \rightsquigarrow Q\ [\varepsilon] \qquad \models bnd(N)\ \mathsf{frm}\ I \qquad N \in \Theta \qquad N \neq M \qquad \forall m \in \Phi.\ mdl(m) \not\preceq N \qquad C\ \text{binds no } N\text{-method}}{\Phi, (\Theta \mathbin{⍉} I) \vdash_M C :\ P \wedge I \rightsquigarrow Q \wedge I\ [\varepsilon]}$$

Observe that, because boundaries have no spec-only variables (Definition 3.1), and $bnd(N)$ frames I, the latter does not depend on any spec-only variables. To prove validity of the conclusion, suppose ψ^+ is a $(\Phi, \Theta \mathbin{⍉} I)$-model. To use the premise, define $\psi^-(m)$ as follows. For m in Φ, let $\psi^-(m) \mathrel{\hat{=}} \psi^+(m)$. For m in Θ with $\Theta(m) = R \rightsquigarrow S\ [\eta]$ define, for any τ

$$\psi^-(m)(\tau) \mathrel{\hat{=}} \begin{cases} \{\lightning\} & \tau \not\models R, \\ \varnothing & \tau \models R \wedge \neg I, \\ \psi^+(m)(\tau) & \tau \models R \wedge I. \end{cases}$$

The precondition R may have spec-only variables, in which case $\tau \models R \wedge I$ abbreviates that there are some values for the spec-only variables so that $R \wedge I$ holds. Because I has no spec-only variables, the clauses are exhaustive and mutually disjoint. It is straightforward to check that ψ^- is a (Φ, Θ)-model according to Definition 5.9.

For the rest of the proof, we consider arbitrary σ with $\hat{\sigma} \models P \wedge I$, where $\hat{\sigma} \mathrel{\hat{=}} [\sigma{+}\overline{s}{:}\overline{v}]$ is the extension of σ uniquely determined by P and σ according to Lemma 5.1.

To finish the proof, we need the following.

> Claim. *If* $\langle C,\ \sigma,\ \_\rangle \overset{\psi^+}{\longmapsto}{}^*\ \langle B,\ \tau,\ \mu\rangle$, *then* $\tau \models I$ *and that sequence of configurations is also a trace* $\langle C,\ \sigma,\ \_\rangle \overset{\psi^-}{\longmapsto}{}^*\ \langle B,\ \tau,\ \mu\rangle$ *via* ψ^-.

We also need the following observations, to prove the Claim and to prove the rule. For any B, τ, μ:

(a) If $Active(B)$ is not a call to method in Θ, then the transitions from $\langle B,\ \tau,\ \mu\rangle$ via $\overset{\psi^+}{\longmapsto}$, to $\lightning$ or to a configuration, are the same as those via ψ^-. Because: the model is only used for calls, and the models differ only on methods of Θ.

(b) If $Active(B)$ is a call to some method m of Θ, and $\tau \models I$, then the transitions from $\langle B,\ \tau,\ \mu\rangle$ via $\overset{\psi^+}{\longmapsto}$ are the same as those via ψ^-. Because: For faults, fault via $\overset{\psi^+}{\longmapsto}$ is when the precondition of the original spec $\Theta(m)$ does not hold, and that is one conjunct of the precondition for ψ^-, the other being I. For non-fault, $\psi^-(m)(\tau)$ is defined to be $\psi^+(m)(\tau)$ when $\tau \models I$.

Before proving the Claim, we use it to prove the conditions for validity of the conclusion of SOF.

Safety. Suppose $\langle C,\ \sigma,\ \_\rangle \overset{\psi^+}{\longmapsto}{}^* \langle B,\ \tau,\ \mu\rangle \overset{\psi^+}{\longmapsto} \lightning$. By the Claim, $\langle C,\ \sigma,\ \_\rangle \overset{\psi^-}{\longmapsto}{}^* \langle B,\ \tau,\ \mu\rangle$ and $\tau \models I$. So by observations (a) and (b), we get a faulting step from $\langle B,\ \tau,\ \mu\rangle$ via ψ^-, whence $\langle C,\ \sigma,\ \_\rangle \overset{\psi^-}{\longmapsto}{}^* \lightning$, which contradicts the premise of SOF.

Post. For all τ such that $\langle C,\ \sigma,\ \_\rangle \overset{\psi^+}{\longmapsto}{}^* \langle \mathsf{skip},\ \tau,\ \_\rangle$, we have $\tau \models I$ and $\langle C,\ \sigma,\ \_\rangle \overset{\psi^-}{\longmapsto}{}^* \langle \mathsf{skip},\ \tau,\ \_\rangle$ by the Claim. By premise of the rule, we have $\tau \models Q^{\overline{s}}_{\overline{v}}$. So, we have $\tau \models (Q \wedge I)^{\overline{s}}_{\overline{v}}$, because I has no spec-only variables.

Write. Direct consequence of the premise and the Claim.

R-safe. For m in Θ, the frame condition of $(\Theta \between I)(m)$ is the same as that of $\Theta(m)$, by definition of $\between$. So this is a direct consequence of the premise and the Claim.

Encap. Boundary monotonicity is a direct consequence of the Claim, using the premise. So too the w-respects condition: the condition for the conclusion is the same as for the premise, because $\Phi, \Theta \between I$ has the same methods, thus the same modules, as Φ, Θ has.

For r-respects, consider any reachable step $\langle C,\ \sigma,\ \_\rangle \overset{\psi^+}{\longmapsto}{}^* \langle B,\ \tau,\ \mu\rangle \overset{\psi^+}{\longmapsto} \langle D,\ v,\ v\rangle$ and an alternate step $\langle B,\ \tau',\ \mu\rangle \overset{\psi^+}{\longmapsto} \langle D',\ v',\ v'\rangle$ where $Agree(\tau', v', \delta)$ and τ' agrees with τ according to the r-respect condition for δ, where the collective boundary δ is determined by $Active(B)$, Φ, Θ, and M, in the same way for the conclusion as for the premise (i.e., δ is the same for both).

If the active command of B is not a call to a method in Θ, then the steps can be taken via ψ^- (see observation (a) above) and so r-respect from the premise can be applied. If the active command of B is a call to some method $m \in \Theta$, then we have $\tau \models I$ and $\tau' \models I$ by definition of $\psi^+(m)$. So the steps can both be taken via ψ^- (see observation (b) above). So, we can appeal to r-respect from the premise, and we are done.

Proof of Claim. By induction on steps.

Base case zero steps: immediate from $\hat{\sigma} \models P \wedge I$.

Induction case: $\langle C,\ \sigma,\ \_\rangle \overset{\psi^+}{\longmapsto}{}^* \langle B,\ \tau,\ \mu\rangle \overset{\psi^+}{\longmapsto} \langle D,\ v,\ v\rangle$. The inductive hypothesis is that $\langle C,\ \sigma,\ \_\rangle \overset{\psi^-}{\longmapsto}{}^* \langle B,\ \tau,\ \mu\rangle$, by the same intermediate configurations, and $\tau \models I$.

Case $Active(B)$ not a call to a method of Θ: by observation (a) above, the step to D can be taken via ψ^-. So, we can use Encap from the premise. In particular, we get $Agree(\tau, v, bnd(N))$ by w-respect, owing to side condition $N \in \Theta$ and $M \neq N$ and also the fact that if the step calls m in Φ then $mdl(m) \not\leq N$ by side condition. Moreover, we use side condition that C binds no N-method, so that in the definition of w-respect, we have that $topm(B, M)$ is not N. So from $\models bnd(N)$ frm I and induction hypothesis $\tau \models I$, by definition (27) of the frames judgment, we get $v \models I$.

Case $Active(B)$ is a call to some $m \in \Theta$. Suppose $\Theta(m) = R \rightsquigarrow S\ [\eta]$. By induction hypothesis $\langle C,\ \sigma,\ \_\rangle \overset{\psi^-}{\longmapsto}{}^* \langle B,\ \tau,\ \mu\rangle$ we have $\tau \models R^{\overline{t}}_{\overline{u}}$ (with $\overline{u}$ the uniquely determined values of R's spec-only variables $\overline{t}$), because otherwise there would be a fault via ψ^- contrary to the premise. Because $\tau \models R^{\overline{t}}_{\overline{u}} \wedge I$, we have $\psi^-(m)(\tau) = \psi^+(m)(\tau)$ by definition of $\psi^-(m)$, so the step can be taken via ψ^- and moreover $v \models I$, because ψ^+ is a $\Phi, (\Theta \between I)$-model.

B.10 Soundness of Link

$$\textsc{Link}\ \frac{dom(\Theta) = \overline{m} \qquad \Phi,\Theta \vdash_{mdl(m_i)} B_i : \Theta(m_i) \qquad \forall N \in \Phi, L \in \Theta.\ N \not\leq L \qquad \Phi,\Theta \vdash_\bullet C : P \rightsquigarrow Q\ [\varepsilon] \qquad \forall N, L.\ N \in \Theta \wedge N \prec L \Rightarrow L \in (\Phi,\Theta)}{\Phi \vdash_\bullet \mathsf{let}\ \overline{m} = \overline{B}\ \mathsf{in}\ C : P \rightsquigarrow Q\ [\varepsilon]}$$

Remark 9. It is sound to generalize the rule to allow any module M for C and for the linkage, provided that $bnd(M) = \bullet$.

For clarity, the proof is specialized to case that Θ has a single method named m. We spell out the proof in considerable detail, as there are a number of subtleties. However, we assume there are no recursive calls in the bodies of the linked method. There is no difficulty with recursion, it just complicates the proof: recursion can be handled using a fixpoint construction for the denotational semantics (as in proof of the linking rule in Section A.1 of RLIII, and using quasi-determinacy) and an extra induction on calling depth (as in the linking proofs in both RLII and RLIII).

We use the following from RLII: For method m in the environment, a trace is called ***m-truncated*** provided that $\mathsf{ecall}(m)$ does not occur in the last configuration. This means that a call to m is not in progress, though it allows that a call may happen next. In a trace that is not m-truncated, an environment call has been made to m, making the transition from a command of the form $m(); C$ to $B; \mathsf{ecall}(m); C$ where B is the method body, and then further steps may have been taken. Note that in an m-truncated trace, it is possible that the active command of the last configuration is $m()$.

To prove soundness of the rule, suppose $\Theta(m)$ is $R \rightsquigarrow S[\eta]$ and let $N \mathrel{\hat{=}} mdl(m)$. Assume validity of the premises for B and C:

$$\Phi,\Theta \models_N B : R \rightsquigarrow S\ [\eta] \quad \text{and} \quad \Phi,\Theta \models_\bullet C : P \rightsquigarrow Q\ [\varepsilon]. \tag{48}$$

To prove validity of the conclusion, i.e.,

$$\Phi \models_\bullet \mathsf{let}\ m = B\ \mathsf{in}\ C : P \rightsquigarrow Q\ [\varepsilon], \tag{49}$$

let φ be any Φ-model. Define θ to be the singleton mapping $[m{:}[\![B]\!]_\varphi]$, using the denotation of B, so that $\varphi \cup \theta$ is a (Φ,Θ)-model, by Lemma A.8. (To handle recursive methods, the generalization of Lemma A.8 is proved by induction as in Lemma A.10 of RLIII.) For brevity, we write φ, θ for $\varphi \cup \theta$ and $\overset{\varphi\theta}{\longmapsto}$ for $\overset{\varphi\cup\theta}{\longmapsto}$.

For any σ, the first step is $\langle \mathsf{let}\ m{=}B\ \mathsf{in}\ C, \sigma, \_\rangle \overset{\varphi}{\longmapsto} \langle C; \mathsf{elet}(m), \sigma, [m{:}B]\rangle$, and if the computation reaches a terminal configuration then the last step is the transition for $\mathsf{elet}(m)$, which removes m from the environment but does not change the state. So to prove Equation (49), we use facts about traces from $\langle C,\ \sigma,\ [m{:}B]\rangle$.

The following result is used not only to prove Equation (49) but also used to prove soundness of the relational linking rule. In its statement, we rely on Lemma 5.1 about spec-only variables in wf preconditions.

Lemma B.2. *Suppose we have valid judgments $\Phi,\Theta \models_N B : \Theta(m)$ and $\Phi,\Theta \models_\bullet C : P \rightsquigarrow Q\ [\varepsilon]$, and also $m \notin B$. Let φ be a Φ-model and $\theta \mathrel{\hat{=}} [m{:}[\![B]\!]_\varphi]$. Let σ be any state such that $\sigma \models P$. Suppose $\langle C,\ \sigma,\ [m{:}B]\rangle \overset{\varphi}{\longmapsto}{}^* \langle D,\ \tau,\ \dot{\mu}\rangle$ is m-truncated (for some $D, \tau, \dot{\mu}$). Then*

- $\langle C,\ \sigma,\ \_\rangle \overset{\varphi\theta}{\longmapsto}{}^* \langle D,\ \tau,\ \mu\rangle$, where $\mu = \dot{\mu} \restriction m$.
- If $D \equiv m(); D_0$ for some D_0, then $\tau \models R$.

(Here, the abbreviations $\sigma \models P$ and $\tau \models R$ mean satisfaction by the states extended with the uniquely determined values for spec-only variables.)

Proof. We refrain from giving a detailed proof; it requires a somewhat intricate induction hypothesis, similar to the one for impure methods in RLIII (Section A.2, Claim B) and the one in RLII (Section 7.6). The main ideas are as follows.

The combination φ, θ is a (Φ, Θ)-model, by Lemma A.8. If $\langle C,\ \sigma,\ [m{:}B]\rangle \overset{\varphi}{\longmapsto}{}^{*}\ \langle D,\ \tau,\ \dot{\mu}\rangle$ is m-truncated, then we can factor it into segments alternating between code of C and code of B during environment calls to m. The steps taken in code of C can be taken via $\overset{\varphi\theta}{\longmapsto}$, because the two transition relations are identical except for calls to m. A completed call to m amounts to a terminated execution of B (with a continuation command and environment left unchanged). A completed call gives rise to a single step via $\overset{\varphi\theta}{\longmapsto}$ with the same outcome, because $\theta(m)$ is the denotation of B, which is defined directly in terms of executions of B.[45] Reasoning by induction on the number of completed calls, we construct a trace via $\overset{\varphi\theta}{\longmapsto}$. At each call of m, we appeal to the premise for C to conclude that the precondition of m holds, as otherwise there would be a faulting trace of C via $\overset{\varphi\theta}{\longmapsto}$. □

Proof of Link. Using Lemma B.2, we prove Equation (49), validity of the conclusion of rule Link, as follows, for any σ such that $\hat{\sigma} \models P$ where $\hat{\sigma}$ is $[\sigma{+}\overline{s}{:}\,\overline{v}]$ for the unique values $\overline{v}$ determined by σ.

Post. An execution of $\langle \mathsf{let}\ m = B\ \mathsf{in}\ C,\ \sigma,\ \_\rangle$ via φ that terminates in state τ gives an execution for $\langle C,\ \sigma,\ [m{:}B]\rangle$ via φ that ends in τ. It is m-truncated, so by Lemma B.2 we have $\langle C,\ \sigma,\ \_\rangle \overset{\varphi\theta}{\longmapsto}{}^{*}$ $\langle \mathsf{skip},\ \tau,\ \_\rangle$. By validity of the premise for C, see Equation (48), we get $\tau \models Q^{\overline{s}}_{\overline{v}}$.

Write. By an argument very similar to the one for Post.

Safety. By semantics of let $m = B$ in C and of $\mathsf{elet}(m)$, a faulting execution has the form

$$\langle \mathsf{let}\ m = B\ \mathsf{in}\ C,\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto} \langle C; \mathsf{elet}(m),\ \sigma,\ [m{:}B]\rangle \overset{\varphi}{\longmapsto}{}^{*}\ \langle D; \mathsf{elet}(m),\ \tau,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto} \lightning$$

for some $D, \tau, \dot{\mu}$, with $D \not\equiv \mathsf{skip}$. This yields a faulting execution:

$$\langle C,\ \sigma,\ [m{:}B]\rangle \overset{\varphi}{\longmapsto}{}^{*}\ \langle D,\ \tau,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto} \lightning\,. \tag{50}$$

We show by two cases that this contradicts the premises (48) of Link.

Case. The trace $\langle C,\ \sigma,\ [m{:}B]\rangle \overset{\varphi}{\longmapsto}{}^{*}\ \langle D,\ \tau,\ \dot{\mu}\rangle$ is m-truncated. Note that $\mathit{Active}(D)$ is not a call to m, because that would be an environment call and would not fault next. By Lemma B.2, we get $\langle C,\ \sigma,\ \_\rangle \overset{\varphi\theta}{\longmapsto}{}^{*}\ \langle D,\ \tau,\ \mu\rangle$ (where $\mu = \dot{\mu} \restriction m$), and the transition from $\langle D,\ \tau,\ \mu\rangle$ to $\lightning$ can be taken via $\overset{\varphi\theta}{\longmapsto}$, because it is the same relation as $\overset{\varphi}{\longmapsto}$ except for calls to m. But a faulting trace via φ, θ contradicts the premise for C.

Case. The trace $\langle C, \sigma, [m{:}B]\rangle \overset{\varphi}{\longmapsto}{}^{*}\ \langle D, \tau, \dot{\mu}\rangle$ is not m-truncated. So Equation (50) can be factored as

$$\langle C,\ \sigma,\ [m{:}B]\rangle \overset{\varphi}{\longmapsto}{}^{*}\ \langle m(); D_0,\ \tau_0,\ \dot{\mu}_0\rangle \overset{\varphi}{\longmapsto} \langle B; D_0,\ \tau_0,\ \dot{\mu}_0\rangle \overset{\varphi}{\longmapsto}{}^{*}\ \langle B_0; D_0,\ \tau,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto} \lightning$$

for some $D_0, B_0, \tau_0, \dot{\mu}_0$, where $D \equiv B_0; D_0$. Applying Lemma B.2 to the m-truncated prefix, we get $\langle C,\ \sigma,\ \_\rangle \overset{\varphi\,\theta}{\longmapsto}{}^{*}\ \langle m(); D_0,\ \tau_0,\ \mu_0\rangle$ (where $\mu_0 = \dot{\mu}_0 \restriction m$) and $\tau_0 \models R^{\overline{t}}_{\overline{u}'}$ for some $\overline{u}'$. We also have a faulting execution of B from τ_0, i.e., $\langle B,\ \tau_0,\ \mu_0\rangle \overset{\varphi}{\longmapsto}{}^{*}\ \langle B_0,\ \tau,\ \mu\rangle \overset{\varphi}{\longmapsto} \lightning$, which (because m is not called in B) yields the same via φ, θ, which contradict the premise for B in Equation (48).

[45] A fine point: Calls of m may occur in the scope of local variable blocks, so the state may have locals in addition to the variables of the context Γ of the judgment; this is handled using the implicit conversion of context models is discussed in Section 5.3, footnote 42.

R-safe. The first step is not a call, nor is the elet step if reached. Consider any other reachable configuration: $\langle C,\ \sigma,\ [m{:}B]\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle D,\ \tau,\ \dot{\mu}\rangle$. If $Active(D)$ is a call to some p where $\Phi(p)$ is $R_p \rightsquigarrow S_p[\eta_p]$, then we must show $rlocs(\tau, \eta_p) \subseteq freshL(\sigma, \tau) \cup rlocs(\sigma, \varepsilon)$. Depending on whether $Active(D)$ is in code of C or B, the conclusion follows from the premise of C or B, similarly to the proof for Safety. In the non-m-truncated case, i.e., steps of B, a called method p is different from m, since we are assuming no recursion. The R-safe condition refers to starting state of B (which is τ_0 in the Safety proof above). The premise yields an inclusion of the p's readable locations in those of m in its starting state τ_0. Because the R-safe condition holds for the call of m (by induction hypothesis), its readable locations are included in $rlocs(\sigma, \varepsilon)$. Moreover, locations that are fresh relative to τ_0 are also fresh relative to σ. So the result follows using transitivity of inclusion. A more detailed argument of this form can be found in the proof of Encap below.

Encap. For boundary monotonicity, we must prove, for every N' with $N' = \bullet$ or $N' \in \Phi$, that every reachable step, say with states τ to υ, has $rlocs(\tau, bnd(N')) \subseteq rlocs(\upsilon, bnd(N'))$. For steps of C this is immediate from boundary monotonicity from the premise for C, where boundary monotonicity is for all $N' \in (\Phi, \Theta)$ and $N' = \bullet$. For steps of B and $N' \in \Phi$ this is immediate from Encap from the premise for B, where boundary monotonicity is for all $N' \in (\Phi, \Theta)$ and $N' = N$. However, the judgment for B does not imply anything about the boundary of $\bullet$ (unless $\bullet$ happens to be in Φ, Θ). But by wf, we have $bnd(\bullet) = \bullet$, which makes boundary monotonicity for $bnd(\bullet)$ vacuous.

For w-respect and r-respect, we need to consider arbitrary reachable steps. The first step of let $m = B$ in C deterministically steps to $C; \mathsf{elet}(m)$, putting $m : B$ into the environment without changing or reading the state, so both w-respect and r-respect hold for that step. Both conditions also hold for the step of $\mathsf{elet}(m)$, which again does not change or read the state. So it remains to consider reachable steps of the following form, in which we abbreviate $A \mathrel{\hat{=}} \mathsf{elet}(m)$:

$$\langle \mathsf{let}\ m = B\ \mathsf{in}\ C,\ \sigma,\ \_\rangle \overset{\varphi}{\longmapsto} \langle C; A,\ \sigma,\ [m{:}B]\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle D; A,\ \tau,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto} \langle D_0; A,\ \upsilon,\ \dot{\nu}\rangle, \tag{51}$$

where $D \not\equiv \mathsf{skip}$. Aside from the first step, such traces correspond to traces of the form

$$\langle C,\ \sigma,\ [m{:}B]\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle D,\ \tau,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto} \langle D_0,\ \upsilon,\ \dot{\nu}\rangle,$$

i.e., exactly the same sequence of configurations, but for lacking the trailing $\mathsf{elet}(m)$.

For w-respect, our obligation is to prove that the step $\langle D,\ \tau,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto} \langle D_0,\ \upsilon,\ \dot{\nu}\rangle$ w-respects L for every $L \in (\Phi, \dot{\mu})$ and $L \neq topm(D, \bullet)$. In the case of an m-truncated trace from C to D, we appeal to Lemma B.2. In the case of a non m-truncated trace from C to D, the above step is one arising from an environment call to m and therefore occurs in the trace from B. So, we use w-respects for B. The result follows, because the condition for w-respects L for B is $L \in (\Phi, \Theta, \mu)$ and $L \neq topm(D, N)$ and this is equivalent to the w-respects condition for the step from D, because both conditions are equivalent to $L \in (\Phi, \mu)$. In the case of an m-truncated trace from C to D, we appeal to Lemma B.2. We can use w-respects for the premise C. In the case where $Active(D)$ is not a context call this condition is $L \in (\Phi, \Theta, \mu)$ and $L \neq topm(D, \bullet)$, which is equivalent to $L \in (\Phi, \dot{\mu})$ and $L \neq topm(D, \bullet)$. In the case where $Active(D)$ is a context call to some $p \in \Phi$, the condition to be proved is $L \in (\Phi, \dot{\mu})$ and $L \neq topm(D, \bullet)$ and $mdl(p) \preceq L$. We obtain this from the w-respects condition for the premise, which is $L \in (\Phi, \Theta, \mu)$ and $L \neq topm(D, \bullet)$ and $mdl(p) \preceq L$.

For r-respect, we must show the step $\langle D,\ \tau,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto} \langle D_0,\ \upsilon,\ \dot{\nu}\rangle$ r-respects δ for $(\varphi, \varepsilon, \sigma)$ where δ is defined by cases on $Active(D)$:

- if $Active(D)$ is not a call, then $\delta \mathrel{\hat{=}} (+L \in (\Phi, \dot{\mu}), L \neq topm(D, \bullet).\ bnd(L))$
- if $Active(D)$ is a call to some m, then $\delta \mathrel{\hat{=}} (+L \in (\Phi, \dot{\mu}), mdl(m) \not\preceq L.\ bnd(L))$

Let us spell out the r-respect conditions for the given trace (51).

(*) For any π, τ', υ', if $Agree(\tau', \upsilon', \delta)$ and $\langle D,\ \tau',\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto} \langle D'_0,\ \upsilon',\ \dot{\nu}\rangle$ and $Lagree(\tau, \tau', \pi, freshL(\sigma, \tau) \cup rlocs(\sigma, \varepsilon) \backslash rlocs(\tau, \delta^{\oplus}))$, then $D'_0 \equiv D_0$ and there is $\rho \supseteq \pi$ such that

$$\begin{array}{l} Lagree(\upsilon, \upsilon', \rho, freshL(\tau, \upsilon) \cup wrttn(\tau, \upsilon) \backslash rlocs(\upsilon, \delta^{\oplus})), \\ \rho(freshL(\tau, \upsilon) \backslash rlocs(\upsilon, \delta)) \subseteq freshL(\tau', \upsilon') \backslash rlocs(\upsilon', \delta). \end{array} \quad (\dagger)$$

To prove (*), we go by cases on whether the trace up to D, τ is m-truncated.

Suppose the antecedent of (*) holds: that is,

$$\begin{array}{l} Agree(\tau', \upsilon', \delta) \text{ and } \langle D,\ \tau',\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto} \langle D'_0,\ \upsilon',\ \dot{\nu}\rangle \text{ and} \\ Lagree(\tau, \tau', \pi, (freshL(\sigma, \tau) \cup rlocs(\sigma, \varepsilon)) \backslash rlocs(\tau, \delta^{\oplus})). \end{array}$$

Case. $\langle C,\ \sigma,\ [m{:}B]\rangle \overset{\varphi}{\longmapsto}{}^{*} \langle D,\ \tau,\ \dot{\mu}\rangle$ is m-truncated.

Then by Lemma B.2, we have $\langle C,\ \sigma,\ \_\rangle \overset{\varphi\theta}{\longmapsto}{}^{*} \langle D,\ \tau,\ \mu\rangle$, where $\mu = \dot{\mu} \restriction m$.
If $Active(D)$ is not a context call, then the r-respect condition to be proved is for

$$\begin{array}{rcl} \delta & = & (+L \in (\Phi, \dot{\mu}), L \neq topm(D, \bullet).\ bnd(L)) \\ & = & (+L \in (\Phi, \mu), L \neq topm(D, \bullet).\ bnd(L)), bnd(N). \end{array}$$

We have the additional step $\langle D,\ \tau,\ \mu\rangle \overset{\varphi\theta}{\longmapsto} \langle D'_0,\ \upsilon,\ \nu\rangle$, because in this case φ and $\varphi\theta$ agree. For the same reason the step $\langle D,\ \tau',\ \dot{\mu}\rangle$ to $\langle D'_0,\ \upsilon',\ \dot{\nu}\rangle$ can also be taken via $\varphi\theta$, so $\langle D,\ \tau',\ \mu\rangle \overset{\varphi\theta}{\longmapsto} \langle D'_0,\ \upsilon',\ \nu\rangle$, where $\nu = \dot{\nu} \restriction m$. The Encap condition for the premise for C says that

$$\langle C,\ \sigma,\ \_\rangle \overset{\varphi\theta}{\longmapsto}{}^{*} \langle D,\ \tau,\ \mu\rangle \overset{\varphi\theta}{\longmapsto} \langle D'_0,\ \upsilon,\ \nu\rangle$$

respects $((\Phi, \Theta), \bullet, (\varphi\theta), \varepsilon, \sigma)$.

Unpacking definitions, from r-respect, we have that the step $\langle D,\ \tau,\ \mu\rangle \overset{\varphi\theta}{\longmapsto} \langle D'_0,\ \upsilon,\ \nu\rangle$ r-respects $\dot{\delta}$ for $(\varphi\theta, \varepsilon, \sigma)$, where $\begin{array}{rcl} \dot{\delta} & = & (+L \in (\Phi, \Theta, \mu), L \neq topm(D, \bullet).\ bnd(L)) \\ & = & (+L \in (\Phi, \mu), L \neq topm(D, \bullet).\ bnd(L)), bnd(N) \\ & = & \delta. \end{array}$
Now to establish (†), we show $Agree(\tau', \upsilon', \dot{\delta})$ and $Lagree(\tau, \tau', \pi, freshL(\sigma, \tau) \cup rlocs(\sigma, \varepsilon) \backslash rlocs(\tau, \dot{\delta}^{\oplus}))$. Because $\dot{\delta} = \delta$, both hold by assumption.

If $Active(D)$ is a context call to $p \in \Phi$, then the r-respect condition to be proved is for

$$\begin{array}{rcl} \delta & = & (+L \in (\Phi, \dot{\mu}), mdl(p) \not\leq L.\ bnd(L)) \\ & = & (+L \in (\Phi, \mu), mdl(p) \not\leq L.\ bnd(L)), bnd(N), \end{array}$$

where the last equality follows, because $mdl(m) = N$ and $mdl(p) \not\leq N$ by side condition of Link, and $bnd(\bullet)$ is empty. For the premise for C, note that there is a step $\langle D,\ \tau,\ \mu\rangle \overset{\varphi\theta}{\longmapsto} \langle D'_0,\ \upsilon,\ \nu\rangle$, because φ and $\varphi\theta$ agree on p. For the same reason, the step $\langle D,\ \tau',\ \dot{\mu}\rangle$ to $\langle D'_0,\ \upsilon',\ \dot{\nu}\rangle$ can also be taken via $\varphi\theta$, so $\langle D,\ \tau',\ \mu\rangle \overset{\varphi\theta}{\longmapsto} \langle D'_0,\ \upsilon',\ \nu\rangle$, where $\nu = \dot{\nu} \restriction m$. The r-respect condition for the premise is for collective boundary $\dot{\delta}$, where $\begin{array}{rcl} \dot{\delta} & = & (+L \in (\Phi, \Theta, \mu), mdl(p) \not\leq L.\ bnd(L)) \\ & = & (+L \in (\Phi, \mu), mdl(p) \not\leq L.\ bnd(L)), bnd(N) \\ & = & \delta, \end{array}$
where the second equality follows because $mdl(p) \not\leq N$ by the side condition of the Link rule. From these, we get an argument similar to above, because $Active(\tau', \upsilon', \delta)$ and $Lagree(\tau, \tau', \pi, freshL(\sigma, \tau) \cup rlocs(\sigma, \varepsilon) \backslash rlocs(\tau, \delta^{\oplus}))$ hold by assumption.

This completes the proof of (*) for m-truncated traces.

Case. $\langle C,\ \sigma,\ [m{:}B]\rangle \overset{\varphi}{\longmapsto}{}^{*}\ \langle D,\ \tau,\ \dot{\mu}\rangle$ is not m-truncated. As in the proof of Safety, we factor out the m-truncated prefix for the last call to m. That is, there are $B_0, D_1, \tau_1, \dot{\mu}_1$ such that

$$\begin{array}{lll}\langle C,\ \sigma,\ [m{:}B]\rangle \overset{\varphi}{\longmapsto}{}^{*}\ \langle m();D_1,\ \tau_1,\ \dot{\mu}_1\rangle & \overset{\varphi}{\longmapsto}\ \langle B;\mathsf{ecall}(m);D_1,\ \tau_1,\ \dot{\mu}_1\rangle, & \text{since } \dot{\mu}_1(m)=B,\\ & \overset{\varphi}{\longmapsto}{}^{*}\ \langle B_0;\mathsf{ecall}(m);D_1,\ \tau,\ \dot{\mu}\rangle, & \text{with } D\equiv B_0;\mathsf{ecall}(m);D_1,\\ & \overset{\varphi}{\longmapsto}\ \langle B_1;\mathsf{ecall}(m);D_1,\ \upsilon,\ \dot{\nu}\rangle, & \text{with } D_0\equiv B_1;\mathsf{ecall}(m);D_1.\end{array}$$

So, for just B, we have

$$\langle B,\ \tau_1,\ \dot{\mu}_1\rangle \overset{\varphi}{\longmapsto}{}^{*}\ \langle B_0,\ \tau,\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}\ \langle B_1,\ \upsilon,\ \dot{\nu}\rangle,$$

and as in the proof of Safety, we have $\hat{\tau_1} \models R$ by Lemma B.2. Note that $\mathit{Active}(D) = \mathit{Active}(B_0)$. Moreover, m does not occur in B, B_0, B_1, because there is no recursion. Hence, φ and $\varphi\theta$ agree so that

$$\langle B,\ \tau_1,\ \mu_1\rangle \overset{\varphi\theta}{\longmapsto}{}^{*}\ \langle B_0,\ \tau,\ \mu\rangle \overset{\varphi\theta}{\longmapsto}\ \langle B_1,\ \upsilon,\ \nu\rangle.$$

By assumption, $\langle D,\ \tau',\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}\ \langle D_0',\ \upsilon',\ \dot{\nu}\rangle$. That is,

$$\langle B_0;\mathsf{ecall}(m);D_1,\ \tau',\ \dot{\mu}\rangle \overset{\varphi}{\longmapsto}\ \langle B_1';\mathsf{ecall}(m);D_1',\ \upsilon',\ \dot{\nu}\rangle,$$

where $D_0' \mathrel{\hat{=}} B_1';\mathsf{ecall}(m);D_1'$. There are no calls to m, so

$$\langle B_0,\ \tau',\ \mu\rangle \overset{\varphi\theta}{\longmapsto}\ \langle B_1',\ \upsilon',\ \nu\rangle.$$

Because τ is reached from σ via τ_1, we have $\mathit{freshL}(\sigma,\tau) = \mathit{freshL}(\sigma,\tau_1) \cup \mathit{freshL}(\tau_1,\tau)$, whence $\mathit{freshL}(\tau_1,\tau) \subseteq \mathit{freshL}(\sigma,\tau)$. Moreover, by the validity of premise for C, we can use its R-safe condition for the call to m to obtain $\mathit{rlocs}(\tau_1,\eta) \subseteq \mathit{rlocs}(\sigma,\varepsilon)$.

If $\mathit{Active}(D)$ is a context call to some $p \in \Phi$, then the r-respect condition to be proved is for collective boundary $\begin{array}{rcl}\delta &=& (+L \in (\Phi,\dot{\mu}),\ \mathit{mdl}(p) \not\leq L.\ \mathit{bnd}(L))\\ &=& (+L \in (\Phi,\mu),\ \mathit{mdl}(p) \not\leq L.\ \mathit{bnd}(L)),\ \mathit{bnd}(N)\end{array}$
(in which we omit $L = \bullet$, because $\mathit{bnd}(\bullet)$ is empty). For the premise for B, the r-respect condition is for collective boundary $\dot{\delta}$ where $\begin{array}{rcl}\dot{\delta} &=& (+L \in (\Phi,\Theta,\mu),\ \mathit{mdl}(p) \not\leq L.\ \mathit{bnd}(L))\\ &=& (+L \in (\Phi,\mu),\ \mathit{mdl}(p) \not\leq L.\ \mathit{bnd}(L)),\ \mathit{bnd}(N)\\ &=& \delta,\end{array}$
where the second equality holds by side condition $\mathit{mdl}(p) \not\leq N$ of the Link rule.

Using the antecedent of (*) and noting $\dot{\delta} = \delta$, we get

$$\mathit{Lagree}(\tau,\tau',\pi,(\mathit{freshL}(\tau_1,\tau) \cup \mathit{rlocs}(\tau_1,\eta)\backslash\mathit{rlocs}(\tau,\delta^{\oplus}))).$$

Now, by the r-respect condition for the premise for B (and because $\mathit{Agree}(\tau',\upsilon',\delta)$ holds by assumption), we obtain $\rho \supseteq \pi$ such that

$$\begin{array}{l}\mathit{Lagree}(\upsilon,\upsilon',\rho,(\mathit{freshL}(\tau,\upsilon) \cup \mathit{wrttn}(\tau,\upsilon))\backslash\mathit{rlocs}(\upsilon,\delta^{\oplus}) \text{ and}\\ \rho(\mathit{freshL}(\tau,\upsilon)\backslash\mathit{rlocs}(\upsilon,\delta)) \subseteq \mathit{freshL}(\tau',\upsilon')\backslash\mathit{rlocs}(\upsilon',\delta).\end{array}$$

Furthermore, $B_1' \equiv B_1$, whence $D_1' \equiv D_1$, because B_1 in the source code has a unique continuation. Thus, $D_0' \equiv D_0$. Thus, (†) is established.

If $\mathit{Active}(D)$ is not a context call, then note that $\mathit{topm}(D,\bullet) = \mathit{topm}(B_0;\mathsf{ecall}(m);D_1,\bullet)$. Hence, the r-respect condition to be proved is for collective boundary

$$\delta \quad = \quad (+L \in (\Phi,\dot{\mu}), L \neq \mathit{topm}(D,\bullet).\ \mathit{bnd}(L)).$$

If B_0 does not contain an ecall, then $\mathit{topm}(D,\bullet) = N$. Then,

$$\begin{array}{rcl}\delta &=& (+L \in (\Phi,\dot{\mu}), L \neq N.\ \mathit{bnd}(L))\\ &=& (+L \in (\Phi,\mu).\ \mathit{bnd}(L)),\end{array}$$

where the second equality follows, because $mdl(m) = N$ and $m \in dom\,\dot{\mu}$.
If B_0 contains an outermost ecall(p), then $p \neq m$ and $topm(D, \bullet) = mdl(p)$. Then

$$\begin{array}{rcl} \delta & = & (+L \in (\Phi, \dot{\mu}), L \neq mdl(p).\ bnd(L)) \\ & = & (+L \in (\Phi, \mu), L \neq \bullet.\ bnd(L)), bnd(mdl(p)), bnd(N) \\ & = & (+L \in (\Phi, \mu).\ bnd(L)), bnd(mdl(p)), bnd(N). \end{array}$$

The premise for B gives r-respect for the collective boundary

$$\dot{\delta} \quad = \quad (+L \in (\Phi, \Theta, \mu), L \neq topm(B_0, N).\ bnd(L)).$$

If B_0 has no ecalls, then $topm(B_0, N) = N$. In this case,

$$\begin{array}{rcl} \dot{\delta} & = & (+L \in (\Phi, \Theta, \mu), L \neq N.\ bnd(L)) \\ & = & (+L \in (\Phi, \mu).\ bnd(L)). \end{array}$$

If B_0 contains an outermost ecall(p) as above, then $p \neq m$ and $topm(B_0, N) = mdl(p)$. Then

$$\begin{array}{rcl} \dot{\delta} & = & (+L \in (\Phi, \Theta, \mu), L \neq mdl(p).\ bnd(L)) \\ & = & (+L \in (\Phi, \mu).\ bnd(L)), bnd(mdl(p)), bnd(N). \end{array}$$

In either case, $\dot{\delta} = \delta$. To obtain (†), we must show $Agree(\tau', v', \dot{\delta})$ and

$$Lagree(\tau, \tau', \pi, (freshL(\tau_1, \tau) \cup rlocs(\tau_1, \eta)) \backslash rlocs(\tau, \dot{\delta}^{\oplus}).$$

Since $\dot{\delta} = \delta$, both of these hold by assumption.

C BIPROGRAM SEMANTICS AND RELATIONAL CORRECTNESS (RE SECTION 7)

C.1 On Relation Formulas

Semantics of relation formulas is given in Figures 25 and 37. Omitted in the figures are the left and right typing contexts for the formula. Semantics for quantifiers is written in a way to make clear there is no built-in connection between the left and right values. In particular, we allow one side to bind a reference type while the other binds a variable of integer type. This is useful when a variable is only needed on one side (whereas using a dummy of reference type would make the formula vacuously true in states with no allocated references on that side). For practical purposes, we find little use for quantification at type rgn ; however, it is convenient to exclude null at reference type.

The form $R(\overline{FF})$, where $\overline{FF}$ is a list of 2-expressions, is restricted for simplicity to heap-independent expressions of mathematical type (including integers but excluding references and regions). So the semantics can be defined in terms of given denotations $[\![R]\!]$ that provide a fixed interpretation for atomic predicates R in the signature, as assumed already for semantics of unary formulas. The semantics of left and right expressions is written using $[\![-]\!]$ and defined as follows: $[\![\langle F \rangle]\!](\sigma|\sigma') = \sigma(F)$ and $[\![\rangle F \langle]\!](\sigma|\sigma') = \sigma'(F)$.

Lemma C.1 (Unique Snapshots). *If $\mathcal{P}$ is the precondition in a wf relational spec with spec-only variables $\overline{s}$ on the left and $\overline{s}'$ on the right, then for all σ, σ', π there is at most one valuation $\overline{v}, \overline{v}'$ such that $\sigma|\sigma' \models_\pi \mathcal{P}^{\overline{s},\overline{s}'}_{\overline{v},\overline{v}'}$. Moreover, they are independent from π, i.e., determined by σ, σ' and $\overleftarrow{\mathcal{P}} \wedge \overrightarrow{\mathcal{P}}$.*

The proof is straightforward.

Lemma C.2 (Framing of Region Agreement). $G \doteq G \models \eta|\eta$ frm $\mathbb{A}G\text{`}f$ *where $\eta$ is* $ftpt(G)$, rd $G\text{`}f$.

$$
\begin{array}{lll}
\sigma|\sigma' \models_\pi \langle\!|P|\!\rangle & \text{iff } \sigma' \models P & \\
\sigma|\sigma' \models_\pi \mathcal{P} \wedge \mathcal{Q} & \text{iff } \sigma|\sigma' \models_\pi \mathcal{P} \text{ and } \sigma|\sigma' \models_\pi \mathcal{Q} & \\
\sigma|\sigma' \models_\pi \mathcal{P} \vee \mathcal{Q} & \text{iff } \sigma|\sigma' \models_\pi \mathcal{P} \text{ or } \sigma|\sigma' \models_\pi \mathcal{Q} & \\
\sigma|\sigma' \models_\pi \forall x{:}K|x'{:}K'.\ \mathcal{P} & \text{iff } [\sigma{+}x{:}v]|[\sigma'{+}x'{:}v'] \models_\pi \mathcal{P} & \text{for all } v \in [\![K]\!]\sigma\backslash\{null\} \text{ and } v' \in [\![K']\!]\sigma'\backslash\{null\} \\
\sigma|\sigma' \models_\pi \forall x{:}\mathrm{rgn}|x'{:}\mathrm{rgn}.\ \mathcal{P} & \text{iff } [\sigma{+}x{:}v]|[\sigma'{+}x'{:}v'] \models_\pi \mathcal{P} & \text{for all } v \in [\![\mathrm{rgn}]\!]\sigma \text{ and } v' \in [\![\mathrm{rgn}]\!]\sigma' \\
\sigma|\sigma' \models_\pi \forall x{:}\mathrm{int}|x'{:}\mathrm{int}.\ \mathcal{P} & \text{iff } [\sigma{+}x{:}v]|[\sigma'{+}x'{:}v'] \models_\pi \mathcal{P} & \text{for all } v \in \mathbb{Z} \text{ and } v' \in \mathbb{Z} \\
\sigma|\sigma' \models_\pi R(FF) & \text{iff } [\![FF]\!](\sigma|\sigma') \in [\![R]\!] \text{ (and similarly for list } \overline{FF}) &
\end{array}
$$

Fig. 37. Relation formula semantics cases omitted from Figure 25. See Figure 14 for syntax.

Proof. Suppose $\sigma|\sigma' \models_\pi G \doteq G \wedge \mathbb{A}G\text{`}f$ and $Agree(\sigma, \tau, \eta)$ and $Agree(\sigma', \tau', \eta)$. By semantics, $\sigma|\sigma' \models_\pi \mathbb{A}G\text{`}f$ iff $Agree(\sigma, \sigma', \pi, \mathrm{rd}\, G\text{`}f)$ and $Agree(\sigma', \sigma, \pi^{-1}, \mathrm{rd}\, G\text{`}f)$, i.e.,

$$Lagree(\sigma, \sigma', \pi, rlocs(\sigma, \mathrm{rd}\, G\text{`}f)) \text{ and } Lagree(\sigma', \sigma, \pi, rlocs(\sigma', \mathrm{rd}\, G\text{`}f)).$$

We must show $Lagree(\tau, \tau', \pi, rlocs(\tau, \mathrm{rd}\, G\text{`}f))$ and $Lagree(\tau', \tau, \pi^{-1}, rlocs(\tau', \mathrm{rd}\, G\text{`}f))$.

From $Agree(\sigma, \tau, \eta)$ we get $\sigma(G) = \tau(G)$, and from $Agree(\sigma', \tau', \eta)$ we get $\sigma'(G) = \tau'(G)$. From $\sigma(G) = \tau(G)$, we get that $rlocs(\sigma, \mathrm{rd}\, G\text{`}f) = rlocs(\tau, \mathrm{rd}\, G\text{`}f)$ and from $\sigma'(G) = \tau'(G)$, we get that $rlocs(\sigma', \mathrm{rd}\, G\text{`}f) = rlocs(\tau', \mathrm{rd}\, G\text{`}f)$. So, it suffices to show

$$Lagree(\tau, \tau', \pi, rlocs(\sigma, \mathrm{rd}\, G\text{`}f)) \text{ and } Lagree(\tau', \tau, \pi^{-1}, rlocs(\sigma', \mathrm{rd}\, G\text{`}f)).$$

First the left conjunct: For any $o.f \in rlocs(\sigma, \mathrm{rd}\, G\text{`}f)$, we have from above that $\tau(o.f) = \sigma(o.f) \overset{\pi}{\sim} \sigma'(\pi(o).f)$ so it remains to show $\sigma'(\pi(o).f) = \tau'(\pi(o).f)$. From $\sigma|\sigma' \models_\pi G \doteq G$, we have $\sigma(G) \overset{\pi}{\sim} \sigma'(G)$, i.e., $\pi(\sigma(G)) = \sigma'(G)$. So $\pi(o) \in \sigma'(G)$, and we get $\sigma'(\pi(o).f) = \tau'(\pi(o).f)$ from $Agree(\sigma', \tau', \mathrm{rd}\, G\text{`}f)$.

Now the right conjunct: For any $o.f \in rlocs(\sigma', \mathrm{rd}\, G\text{`}f)$, $\sigma(\pi^{-1}(o).f) \overset{\pi}{\sim} \sigma'(o.f) = \tau'(o.f)$ so it remains to show $\tau(\pi^{-1}(o).f) = \sigma(\pi^{-1}(o).f)$. From $\sigma|\sigma' \models_\pi G \doteq G$, we have $\sigma(G) \overset{\pi}{\sim} \sigma'(G)$, i.e., $\pi(\sigma(G)) = \sigma'(G)$. So $\pi^{-1}(o) \in \sigma(G)$, and we get $\sigma(\pi^{-1}(o).f) = \tau(\pi^{-1}(o).f)$ from $Agree(\sigma, \tau, \mathrm{rd}\, G\text{`}f)$. □

Lemma C.3. *If* $(\sigma|\sigma') \overset{\pi,\pi'}{\approx} (\tau|\tau')$, *then* $\sigma|\sigma' \models_\rho \mathcal{P}$ *implies* $\tau|\tau' \models_{\pi^{-1};\rho;\pi'} \mathcal{P}$.

Here $\pi^{-1}; \rho; \pi'$ denotes composition of refperms in diagrammatic order, so $(\pi^{-1}; \rho; \pi')(o)$ is $\pi'(\rho(\pi^{-1}(o)))$ if it is defined on o.

Proof. Proof by induction on $\mathcal{P}$. We consider two cases; the other cases are similar or simpler.

Consider the case of $F \doteq F'$, where F, F' are expressions of some class type K. (The argument for type rgn is similar and for base types int and bool straightforward.) Now suppose $\sigma|\sigma' \models_\rho F \doteq F'$, i.e., $\sigma(F) \overset{\rho}{\sim} \sigma'(F')$. For the non-null case, this is equivalent to $\rho(\sigma(F)) = \sigma'(F')$. (We leave the null case to the reader.) We must show $\tau(F) \overset{\pi^{-1};\rho;\pi'}{\sim} \tau'(F)$, i.e., $\pi'(\rho(\pi^{-1}(\tau(F)))) = \tau'(F')$. From $(\sigma|\sigma') \overset{\pi,\pi'}{\approx} (\tau|\tau')$ we have $\sigma \overset{\pi}{\approx} \tau$ and $\sigma' \overset{\pi'}{\approx} \tau'$ by definition. By Lemma 5.6 we get $\sigma(F) \overset{\pi}{\sim} \tau(F)$ and $\sigma'(F') \overset{\pi'}{\sim} \tau'(F')$, which for non-null values means $\pi(\sigma(F)) = \tau(F)$ and $\pi'(\sigma'(F')) = \tau'(F')$. We conclude by using the equations to calculate $\pi'(\rho(\pi^{-1}(\tau(F)))) = \pi'(\rho(\pi^{-1}(\pi(\sigma(F))))) = \pi'(\rho(\sigma(F))) = \pi'(\sigma'(F)) = \tau'(F')$.

Consider the case of $\mathbb{A}G\text{`}f$ where $f$ is a reference type field. Suppose $\sigma|\sigma' \models_\rho \mathbb{A}G\text{`}f$. By semantics and the definitions of *Agree*, *rlocs*, and *Lagree*, this is equivalent to

$$\forall o \in \sigma(G).\ \sigma(o.f) \overset{\rho}{\sim} \sigma'(\rho(o).f). \tag{52}$$

In the rest of the proof, we consider the non-null case, so the body can be rephrased as $\rho(\sigma(o.f)) = \sigma'(\rho(o).f)$. We must show

$$\forall p \in \tau(G).\ \tau(p.f)\ \overset{\pi^{-1};\rho;\pi'}{\approx}\ \tau'(\pi'(\rho(\pi^{-1}(p))).f),$$

i.e., $\pi'(\rho(\pi^{-1}(\tau(p.f)))) = \tau'(\pi'(\rho(\pi^{-1}(p))).f)$. By $\sigma \overset{\pi}{\approx} \tau$, we have $p \in \tau(G)$ iff $\pi^{-1}(p) \in \sigma(G)$, so we reformulate our obligation in terms of $\pi(o)$:

$$\forall o \in \sigma(G).\ \pi'(\rho(\pi^{-1}(\tau(\pi(o).f)))) = \tau'(\pi'(\rho(\pi^{-1}(\pi(o))))).f). \tag{53}$$

By the isomorphisms $\sigma(F) \overset{\pi}{\sim} \tau(F)$ and $\sigma'(F') \overset{\pi'}{\sim} \tau'(F')$, we have $\pi(\sigma(o.f)) = \tau(\pi(o).f)$ and $\pi'(\sigma'(p.f)) = \tau'(\pi'(p).f)$ for any o, p. We prove Equation (53) by calculating for any $o \in \sigma(G)$:

$$\begin{array}{ll}
\pi'(\rho(\pi^{-1}(\tau(\pi(o).f)))) & \\
= \pi'(\rho(\pi^{-1}(\pi(\sigma(o.f))))) & \text{by } \pi(\sigma(o.f)) = \tau(\pi(o).f) \\
= \pi'(\rho(\sigma(o.f))) & \text{by } \pi \text{ bijective} \\
= \pi'(\sigma'(\rho(o).f)) & \text{by } \rho(\sigma(o.f)) = \sigma'(\rho(o).f) \text{ from Equation (52)} \\
= \tau'(\pi'(\rho(o)).f) & \text{by } \pi'(\sigma'(p.f)) = \tau'(\pi'(p).f) \\
= \tau'(\pi'(\rho(\pi^{-1}(\pi(o))))).f) & \text{by } \pi \text{ bijective.} \qquad \square
\end{array}$$

Lemma 8.8 (Refperm Monotonicity). *(i) Any agreement formula is refperm monotonic and so is any refperm independent formula. (ii) Refperm monotonicity is preserved by conjunction, disjunction, and quantification. (iii) Any formula of the form (33), with $\mathcal{R}$ refperm monotonic, is refperm monotonic.*

Proof. (i) To show R is refperm monotonic, we must show for all $\pi, \rho, \sigma, \sigma'$, if $\sigma|\sigma' \models_\pi \mathcal{R}$ and $\rho \supseteq \pi$ then $\sigma|\sigma' \models_\rho \mathcal{R}$. This is immediate in case $\mathcal{R}$ is refperm independent.

There are two general forms for agreement formulas. For the form $F \doteq F'$, we only need to consider F (and thus F') of reference or region type, as otherwise it is refperm independent. For both reference type and region type, we have $\sigma|\sigma' \models_\pi F \doteq F'$ iff $\sigma(F) \overset{\pi}{\sim} \sigma'(F')$ (by semantics, see Figure 25). The latter holds only if $\sigma(F)$ is in the domain of π (for $F : K$) or a subset of the domain (for F : rgn), and *mut. mut.* for $\sigma'(F')$ and the range of π. So $\sigma|\sigma' \models_\pi F \doteq F'$ implies $\sigma|\sigma' \models_\rho F \doteq F'$ for any $\rho \supseteq \pi$.

The other form of agreement formula is $\mathbb{A}LE$ where LE may be a variable x—in which case the meaning is the same as $x \doteq x$ and the above argument applies—or LE has the form $G\text{`}f$. Suppose $\sigma|\sigma' \models_\pi G\text{`}f$. Unfolding the semantics, we have $Agree(\sigma, \sigma', \pi, \mathrm{rd}\, G\text{`}f)$ and $Agree(\sigma', \sigma, \pi^{-1}, \mathrm{rd}\, G\text{`}f)$. That is, $Lagree(\sigma, \sigma', \pi, rlocs(\sigma, \mathrm{rd}\, G\text{`}f))$ and $Lagree(\sigma', \sigma, \pi^{-1}, rlocs(\sigma', \mathrm{rd}\, G\text{`}f)$. This does not entail $\sigma(G) \overset{\pi}{\sim} \sigma'(G)$ (see Section 7.1). But it does entail that $\sigma(G) \subseteq dom(\pi)$ and $\sigma'(G) \subseteq rng(\pi)$ (as already remarked in Section 7.1). So extending π to some $\rho \supseteq \pi$ does not affect the agreements: we have $Lagree(\sigma, \sigma', \rho, rlocs(\sigma, \mathrm{rd}\, G\text{`}f))$ and $Lagree(\sigma', \sigma, \rho^{-1}, rlocs(\sigma', \mathrm{rd}\, G\text{`}f)$ (cf. Equation (21)).

(ii) Conjunction and disjunction are straightforward by definitions. For quantification at a reference type, suppose $\mathcal{R}$ is refperm monotonic and suppose $\sigma|\sigma' \models_\pi \forall x{:}K|x'{:}K'.\ \mathcal{R}$. Thus, by definition (see Figure 37), we have $[\sigma{+}x{:}o]|[\sigma'{+}x'{:}o'] \models_\pi \mathcal{R}$ for all $o \in [\![K]\!]\sigma\backslash\{null\}$ and $o' \in [\![K']\!]\sigma'\backslash\{null\}$. Now, if $\rho \supseteq \pi$ then for any $o \in [\![K]\!]\sigma\backslash\{null\}$ and $o' \in [\![K']\!]\sigma'\backslash\{null\}$ we have $[\sigma{+}x{:}o]|[\sigma'{+}x'{:}o'] \models_\rho \mathcal{R}$ by refperm monotonicity of $\mathcal{R}$. Hence, $\sigma|\sigma' \models_\rho \forall x{:}K|x'{:}K'.\ \mathcal{R}$. For existential quantification, and quantification at type int and type rgn, the argument is the same.

(iii) Suppose $\sigma|\sigma' \models_\pi G \doteq G' \wedge (\forall x{:}K \in G|x{:}K \in G'.\ \mathbb{A}x \Rightarrow \mathcal{R})$. So $\sigma|\sigma' \models_\pi G \doteq G'$, i.e., by semantics $\sigma(G) \overset{\pi}{\sim} \sigma'(G')$. Thus, each element of $\sigma(G)$ (respectively, $\sigma'(G')$) is in the domain (respectively, range) of π. Also by semantics, we have $[\sigma{+}x{:}o]|[\sigma'{+}x{:}o'] \models_\pi \mathcal{R}$, for every $(o, o') \in X$ where $X = \{(o, o') \mid o \in \sigma(G), o' \in \sigma'(G'),$ and $(o, o') \in \pi\}$.

Now suppose $\rho \supseteq \pi$. We have $\sigma|\sigma' \models_\rho G \doteq G'$— As already noted, agreements are refperm monotonic. For the second conjunct, we need $[\sigma{+}x{:}\,o]|[\sigma'{+}x{:}\,o'] \models_\rho \mathcal{R}$ for every (o, o') in the set Y where $Y = \{(o, o') \mid o \in \sigma(G), o' \in \sigma'(G'),$ and $(o, o') \in \rho\}$. But $Y = X$, owing to $\sigma(G) \overset{\pi}{\sim} \sigma'(G')$ hence $o \in dom(\pi)$ and $o' \in rng(\pi)$. So the result follows by refperm monotonicity of $\mathcal{R}$. □

C.2 On Biprogram Semantics

Example C.4. Bi-coms deterministically dovetail unary steps, without regard to the unary control structure. For example, traces of (while 1 do $a; b; c$ | while 1 do d) look like this[46]:

$\langle$(while 1 do $(a; b; c)$ | while 1 do $d)\rangle$
$\langle(a; b; c;$ while 1 do $(a; b; c) \Rsh$ while 1 do $d)\rangle$
$\langle(a; b; c;$ while 1 do $(a; b; c)$ | $d;$ while 1 do $d)\rangle$
$\langle(b; c;$ while 1 do $(a; b; c) \Rsh d;$ while 1 do $d)\rangle$
$\langle(b; c;$ while 1 do $(a; b; c)$ | while 1 do $d)\rangle$
$\langle(c;$ while 1 do $(a; b; c) \Rsh$ while 1 do $d)\rangle$
$\langle(c;$ while 1 do $(a; b; c)$ | $d;$ while 1 do $d)\rangle$
$\langle($while 1 do $(a; b; c) \Rsh d;$ while 1 do $d)\rangle$
$\langle($while 1 do $(a; b; c)$ | while 1 do $d)\rangle$
...

The right side iterated twice, the left once.

Example C.5. In terms of operational semantics, the respective computations of the five biprograms in Equation (12) are as follows, where for clarity, we underline the active command for the underlying unary transition, and abbreviate skip as $\bullet$:

$\langle(\underline{a}; b; c|d; e; f)\rangle\langle(b; c|\underline{\Rsh}\, \underline{d}; e; f)\rangle\langle(\underline{b}; c|e; f)\rangle\langle(c|\underline{\Rsh}\, \underline{e}; f)\rangle\langle(\underline{c}|f)\rangle\langle(\bullet|\underline{\Rsh}\, f)\rangle\langle\lfloor\bullet\rfloor\rangle$
$\langle(\underline{a}; b|d); (c|e; f)\rangle\langle(b|\underline{\Rsh}\, \underline{d}); (c|e; f)\rangle\langle(\underline{b}|\bullet); (c|e; f)\rangle\langle(\underline{c}|e; f)\rangle\langle(\bullet|\underline{\Rsh}\, \underline{e}; f)\rangle\langle(\bullet|\underline{f})\rangle\langle\lfloor\bullet\rfloor\rangle$
$\langle(\underline{a}|d; e); (b; c|f)\rangle\langle(\bullet|\underline{\Rsh}\, \underline{d}; e); (b; c|f)\rangle\langle(\bullet|\underline{e}); (b; c|f)\rangle\langle(\underline{b}; c|f)\rangle\langle(c|\underline{\Rsh}\, \underline{f})\rangle\langle(\underline{c}|\bullet)\rangle\langle\lfloor\bullet\rfloor\rangle$
$\langle(\underline{a}; b; c|\bullet); (\bullet|d; e; f)\rangle\langle(\underline{b}; c|\bullet); (\bullet|d; e; f)\rangle\langle(\underline{c}|\bullet); (\bullet|d; e; f)\rangle\langle(\bullet|\underline{d}; e; f)\rangle\langle(\bullet|\underline{e}; f)\rangle\langle(\bullet|\underline{f})\rangle\langle\lfloor\bullet\rfloor\rangle$
$\langle(\bullet|\underline{d}; e; f); (a; b; c|\bullet)\rangle\langle(\bullet|\underline{\Rsh}\, \underline{e}; f); (a; b; c|\bullet)\rangle\langle(\bullet|\underline{\Rsh}\, \underline{f}); (a; b; c|\bullet)\rangle\langle(\underline{a}; b; c|\bullet)\rangle\langle(\underline{b}; c|\bullet)\rangle\langle(\underline{c}|\bullet)\rangle\langle\lfloor\bullet\rfloor\rangle$

Note that d-steps of the last two examples go by rule bComR0.

Example C.6. In the preceding, we illustrate what happens when the commands do not fault. Now suppose that the transition for c faults but none of the others do. (That is, the c-transitions above do not exist.) Thus, there are unary traces completing actions ab and def, which can be covered by $((a|d; e); (b; c|f))$ and by $((\bullet|d; e; f); (a; b; c|\bullet))$ but not by $(a; b; c|d; e; f)$ or the other rearrangements.

If instead both c and e fault, then both $(a; b|d); (c|e; f)$ and $(a; b; c|\text{skip}); (\text{skip}|d; e; f)$ fault trying to execute c, while the others fault trying to execute e.

Here is an example of the weaving axiom for conditional:

$$(\text{if } E \text{ then } a; b \text{ else } c; d|\text{if } E' \text{ then } e; f \text{ else } g; h) \hookrightarrow \text{if } E|E' \text{ then } (a; b|e; f) \text{ else } (c; d|g; h).$$

Consider a trace of the **left-hand side (lhs)**, where E is true in the left state and E' is false on the right. Absent faults, the trace may look as follows:

[46]The details depend on the unary transition semantics for loops, which is a standard one that takes a step to unfold the loop body. An alternate semantics, e.g., using a stack of continuations, would work slightly differently but the point is the same: bi-com deterministically dovetails the unary executions without regard to unary control structure.

$\langle$(if E then $a; b$ else $c; d$|if E' then $e; f$ else $g; h$)$\rangle$
$\langle$($a; b$|$\lceil$if E' then $e; f$ else $g; h$)$\rangle$
$\langle$($a; b$|$g; h$)$\rangle$
$\langle$(b|$\lceil g; h$)$\rangle$
$\langle$(b|h)$\rangle$
$\langle$(skip|$\lceil h$)$\rangle$
$\langle\lfloor$skip$\rfloor\rangle$

For the **right-hand side (rhs)**, a trace from the same states has only the initial configuration:

$$\langle \text{if } E|E' \text{ then } (a; b|e; f) \text{ else } (c; d|g; h)\rangle.$$

It faults next, an alignment fault due to test disagreement.

Lemma 4.6. $(\overleftarrow{CC}|\overrightarrow{CC}) \mapsto^* CC$ *for any* CC.

Proof. We need the fact that $\mapsto^*$ is a congruence. This is proved by induction on the reflexive-transitive closure, using the congruence rules for $\mapsto$ (Figure 18).

The proof of the lemma proceeds by induction on CC . It's easy to check the lemma holds when CC is of the form $\lfloor A\rfloor$. For the inductive cases, we rely on congruence and transitivity of $\mapsto^*$. For example, consider the case when $CC \equiv DD; EE$. We need to show $(\overleftarrow{DD; EE}|\overrightarrow{DD; EE}) \mapsto^* (DD; EE)$. We have

	$(\overleftarrow{DD; EE}\|\overrightarrow{DD; EE})$	
$\equiv$	$(\overleftarrow{DD}; \overleftarrow{EE}\|\overrightarrow{DD}; \overrightarrow{EE})$	def of projection
$\mapsto$	$(\overleftarrow{DD}\|\overrightarrow{DD}); (\overleftarrow{EE}\|\overrightarrow{EE})$	using $\mapsto$ axiom for sequence
$\mapsto^*$	$DD; (\overleftarrow{EE}\|\overrightarrow{EE})$	congruence and ind hyp $(\overleftarrow{DD}\|\overrightarrow{DD}) \mapsto^* DD$
$\mapsto^*$	$DD; EE$	congruence and ind hyp $(\overleftarrow{EE}\|\overrightarrow{EE}) \mapsto^* EE$.

So $(\overleftarrow{DD; EE}|\overrightarrow{DD; EE}) \mapsto^* DD; EE$ by transitivity. The other cases follow the same pattern. □

Lemma C.7. *For any* C*, we have* $Active(\lfloor\!\lfloor C \rfloor\!\rfloor) = \lfloor\!\lfloor Active(C) \rfloor\!\rfloor$.

The proof is by induction on C using definitions.

Lemma C.8 (Quasi-determinacy of Biprogram Transitions). *Let* φ *be a relational pre-model. Then (a)* $\overset{\varphi}{\Longmapsto}$ *is rule-deterministic. (b) If* $(\sigma|\sigma') \overset{\pi|\pi'}{\approx} (\sigma_0|\sigma_0')$ *and* $\langle CC, \sigma|\sigma', \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \langle DD, \tau|\tau', \nu|\nu'\rangle$ *and* $\langle CC, \sigma_0|\sigma_0', \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \langle DD_0, \tau_0|\tau_0', \nu_0|\nu_0'\rangle$*, then* $DD \equiv DD_0$*,* $\nu = \nu_0$*,* $\nu' = \nu_0'$*, and there are* $\rho \supseteq \pi$ *and* $\rho' \supseteq \pi'$ *such that* $(\tau|\tau') \overset{\rho|\rho'}{\approx} (\tau_0|\tau_0')$*. (c) If* $(\sigma|\sigma') \overset{\pi|\pi'}{\approx} (\sigma_0|\sigma_0')$*, then* $\langle CC, \sigma|\sigma', \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \lightning$ *iff* $\langle CC, \sigma_0|\sigma_0', \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \lightning$.

Proof. Similar to the proof of Lemma A.6. For the one-sided biprogram transition rules like bComL, the argument makes direct use of Lemma A.6. Explicit side conditions of rules bSync and bSyncX ensure that $\lfloor m()\rfloor$ transitions only by bCall, bCallX, or bCall0.

A configuration for $(C|D)$ with $C \not\equiv$ skip takes a step via either bComL or bComLX depending whether C faults or steps; and these are mutually exclusive according to a result about the unary transition relation. A configuration for (skip|D) with $D \not\equiv$ skip goes via either bComR0 or bComRX, depending on whether D faults or not. A configuration for $(C\lceil D)$ goes via bComR or bComRX. The slightly intricate formulation of the rules for bi-com is necessitated by the need for determinacy and liveness.

Similarly, the rules for bi-while in Figure 28 are formulated to be rule deterministic, e.g., bWhR is only enabled if bWhL is not. □

Projection and embedding: between unary and biprogram traces. It is convenient to classify the biprogram transition rules as follows. Leaving aside bSeq and bSeqX, all the other biprogram rules apply to a non-sequence biprogram of some form. Rules bComL and bWhL take ***left-only*** steps, leaving the right side unchanged, whereas bComR, bComR0, and bWhR take ***right-only*** steps. All the other rules are for ***both-sides*** steps or faulting steps.

Lemma 7.8 (Trace Projection). *Suppose φ is a pre-model. Then the following hold. (a) For any step $\langle BB,\ \sigma|\sigma',\ \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \langle CC,\ \tau|\tau',\ \nu|\nu'\rangle$, either*

- $\langle \overleftarrow{BB},\ \sigma,\ \mu\rangle \overset{\varphi_0}{\longmapsto} \langle \overleftarrow{CC},\ \tau,\ \nu\rangle$ *and* $\langle \overrightarrow{BB},\ \sigma',\ \mu'\rangle \overset{\varphi_1}{\longmapsto} \langle \overrightarrow{CC},\ \tau',\ \nu'\rangle$, *or*
- $\langle \overleftarrow{BB},\ \sigma,\ \mu\rangle = \langle \overleftarrow{CC},\ \tau,\ \nu\rangle$ *and* $\langle \overrightarrow{BB},\ \sigma',\ \mu'\rangle \overset{\varphi_1}{\longmapsto} \langle \overrightarrow{CC},\ \tau',\ \nu'\rangle$, *or*
- $\langle \overleftarrow{BB},\ \sigma,\ \mu\rangle \overset{\varphi_0}{\longmapsto} \langle \overleftarrow{CC},\ \tau,\ \nu\rangle$ *and* $\langle \overrightarrow{BB},\ \sigma',\ \mu'\rangle = \langle \overrightarrow{CC},\ \tau',\ \nu'\rangle$.

(b) For any trace T via $\overset{\varphi}{\Longmapsto}$, there are unique traces U via $\overset{\varphi_0}{\longmapsto}$ and V via $\overset{\varphi_1}{\longmapsto}$, and schedule l, r, such that $align(l, r, T, U, V)$.
(c) If $Active(BB) \equiv \lVert B \rVert$ for some B, then $\langle \overleftarrow{BB},\ \sigma,\ \mu\rangle \overset{\varphi_0}{\longmapsto} \langle \overleftarrow{CC},\ \tau,\ \nu\rangle$ and $\langle \overrightarrow{BB},\ \sigma',\ \mu'\rangle \overset{\varphi_1}{\longmapsto} \langle \overrightarrow{CC},\ \tau',\ \nu'\rangle$.

Proof. Part (a) is by case analysis of the biprogram transition rules. For the rules bCallS and bCallX, observe that the condition (unary compatibility) ensures that the unary steps can be taken. For rule bCall0, the biprogram transition is a stutter, with both $\langle \overleftarrow{BB},\ \sigma,\ \mu\rangle = \langle \overleftarrow{CC},\ \tau,\ \nu\rangle$ and $\langle \overrightarrow{BB},\ \sigma,\ \mu\rangle = \langle \overrightarrow{CC},\ \tau,\ \nu\rangle$. Indeed, either the left or right step is in the transition relation (or both), via the unary rule uCall0 for empty model, owing to Lemma 7.5.

In all other cases, it is straightforward to check that the rule corresponds to a unary step on one or both sides, and in case it is a step on just one side the other side remains unchanged. Note that it can happen that a step changes nothing: in the unary transition relation, this happens for empty model of a context call, e.g., biprogram step via bComL using unary transition uCall0.

For part (b) the proof goes by induction on T and case analysis on the rule by which the last step was taken. Recall that traces are indexed from 0. The base case is T composed of a single configuration, T_0. Let U be $\overleftarrow{T_0}$, V be $\overrightarrow{T_0}$, and let both l and r be the singleton mapping $\{0 \mapsto 0\}$. For the induction step, suppose T has length $n+1$ and let S be the prefix including all but the last configuration T_n. By induction hypothesis, we get l, r, U, V such that $align(l, r, S, U, V)$. There are three sub-cases, depending on whether the step from T_{n-1} to T_n is a left-only step (rule bComL or bWhL), or right-only, or both sides. In the case of left-only, let U' be $U\overleftarrow{T_n}$, let l' be $l \cup \{n \mapsto len(U)\}$, and let r' be $r \cup \{n \mapsto len(V) - 1\}$. Then $align(l', r', T, U', V)$. The other two sub-cases are similar.

Part (c) holds, because one-sided steps are taken only by transition rules bComL, bComR, bComR0, bWhL, and bWhR, none of which are applicable to fully aligned programs. □

Lemma C.9 (Trace Embedding). *Suppose φ is a pre-model. Let cfg be a biprogram configuration. Let U be a trace via φ_0 from $\overleftarrow{cfg}$, and V via φ_1 from $\overrightarrow{cfg}$. Then there is trace T via φ from cfg and traces W from $\overleftarrow{cfg}$ and X from $\overrightarrow{cfg}$ and l, r with $align(l, r, T, W, X)$, such that either*

(a) $U \le W$ and $V \le X$,
(b) $U \le W$ and $X < V$ and W faults next and so does T,
(c) $V \le X$ and $W < U$ and X faults next and so does T,
(d) $W < U$ or $X < V$ and the last configuration of T faults, via one of the rules bCallX, bIfX, or bWhX, i.e., alignment fault.

Proof. First, we make some preliminary observations about the possibilities for a single step. Let cfg be $\langle CC,\ \sigma|\sigma',\ \mu|\mu'\rangle$ such that cfg does not fault next and $CC \not\equiv \lfloor \mathsf{skip} \rfloor$ so there is a next step. By rule determinacy (Lemma C.8(a)), there is a unique applicable transition rule. That rule may

be a left-only, right-only, or both-sides step, as per Lemma 7.8(a). For all but one of the biprogram transition rules, the form of the rule determines whether its transitions are left-, right-, or both-sides. The one exception is bCall0: in case of a transition by this rule, at least one of the unary parts can take a transition, owing to Lemma 7.5, but whether it is left, right, or both depends on the unary models and the states.

For left-only transitions, the applicable rules are bComL and bWhL. In case of bWhL, $\overleftarrow{CC}$ is a loop with test true in σ and $\langle \overleftarrow{CC}, \sigma, \mu\rangle$ takes a deterministic step, unrolling the loop and leaving the state and environment unchanged. In case of bComL, $CC \equiv (C|C')$ for some C, C' with $C \not\equiv$ skip, and $\langle C, \sigma, \mu\rangle$ can step via $\overset{\varphi_0}{\longmapsto}$ to some $\langle D, \tau, \nu\rangle$ where τ may be nondeterministically chosen in case C is an allocation or a context call. (If ν differs from μ, it is because C is a let command and its transition is deterministic.) For any choice of τ, rule bComL allows $\langle (C|C'), \sigma|\sigma', \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \langle (D|\!\!\!\!\lceil C'), \tau|\sigma', \nu|\mu'\rangle$ (or $(D|\text{skip})$ if C' is skip). For right-only transitions, the applicable rules are bComR, bComR0, and bWhR, which are similar to the left-only ones.

The remaining transitions are both-sides. By cases on the many applicable both-sides rules, we find in each case that: (i) the left and right projections have successors under $\overset{\varphi_0}{\longmapsto}, \overset{\varphi_1}{\longmapsto}$, and (ii) if $\langle \overleftarrow{CC}, \sigma, \mu\rangle \overset{\varphi_0}{\longmapsto} \langle D, \tau, \nu\rangle$ and $\langle \overrightarrow{CC}, \sigma', \mu'\rangle \overset{\varphi_1}{\longmapsto} \langle D', \tau', \nu'\rangle$, then there is some DD with $\overleftarrow{DD} \equiv D$, $\overrightarrow{DD} \equiv D'$, and $\langle CC, \sigma|\sigma', \mu|\mu'\rangle \overset{\varphi}{\Longmapsto} \langle DD, \tau|\tau', \nu|\nu'\rangle$. Note that, as in the one-sided cases, τ and/or τ' may be nondeterministically chosen (e.g., in the case of bSync), and any such choices can also be used for the biprogram transition. In case the active command of *cfg* is a sync'd conditional or loop, the applicable rules include ones like bIfTT that have corresponding unary transitions but also the rules bIfX and bWhX in which the biprogram faults although the left and right projections can continue.

For a both-sides step by rule bCallS, we rely on condition (relational compatibility) in Definition 7.4 of pre-model, to ensure that the two unary results τ, τ' can be combined to an outcome $\tau|\tau'$ from $\varphi_2(m)$—since otherwise the biprogram configuration faults via bCallX, contrary to the hypothesis of our preliminary observation above that *cfg* does not fault.

To prove the lemma, we construct T, W, X by iterating the preceding observations, choosing the left and right unary steps in accord with U and V, unless and until those traces are exhausted. If needed, W (respectively, X) is extended beyond U (respectively, V).

Let us describe the construction in more detail, as an iterative procedure in which l, r, W, X, T are treated as mutable variables, and there is an additional variable k. Initialize W, X, T to the singleton traces $\overleftarrow{cfg}$, $\overrightarrow{cfg}$, and *cfg*, respectively. Initially, let $k := 0$. Let l and r both be the singleton mapping $\{0 \mapsto 0\}$. The loop maintains this invariant:

$$align(l, r, T, W, X) \text{ and } (U \le W \vee W \le U) \text{ and } (V \le X \vee X \le V)$$
$$len(T) = k + 1 \text{ and } len(W) = l(k) + 1 \text{ and } len(X) = r(k) + 1$$

Thus, the last configurations of T, W, X are indexed $k, l(k), r(k)$, respectively.

• While ($U \not\le W$ or $V \not\le X$) and neither W, X, nor T faults next, do the following updates, defined by cases on whether T_k is left-only, right-only, or both-sides.

For left-only: update l, r, W, T as follows:

- set $l(k+1) := l(k) + 1$, $r(k+1) := r(k)$,
- if $W < U$, set $W := W \cdot U_{l(k)}$; otherwise extend W by a choosen successor of $W_{l(k)}$,
- set $T := T \cdot cfg'$ where cfg' is determined by the configuration added to W, in accord with the preliminary observations above. Note in particular that T_k does not fault due to failed alignment condition, i.e., by rules bIfX, bCallX, or bWhX, because if it does, then the loop terminates.

For right-only: update l, r, X, T as follows:

- set $l(k+1) := l(k)$, $r(k+1) := r(k) + 1$,
- set $X := X \cdot V_{r(k)}$ if $X < V$, otherwise extend X with a choosen successor of $X_{r(k)}$,
- set $T := T \cdot cfg'$ where cfg' is determined by the configuration added to X.

For both-sides steps, set $l(k+1) := l(k) + 1$, $r(k+1) := r(k) + 1$, and update W, X, T similarly to the preceding cases, in accord with the preliminary observations.

To see that the invariants hold following these updates, note that the invariant implies $\overleftarrow{T_k} = W_{l(k)}$ and $\overrightarrow{T_k} = X_{r(k)}$. Then by construction we get a match for the new configuration: $\overleftarrow{T_{k+1}} = W_{l(k+1)}$ and $\overrightarrow{T_{k+1}} = X_{r(k+1)}$.

The loop terminates, because each iteration decreases the natural number:

$$(2 \times (len(W) \dot{-} len(U)) + (len(X) \dot{-} len(V)) + (1 \text{ if "active cmd is bi-com" else } 0).$$

Here $n \dot{-} m$ means subtraction but 0 if $m > n$. The term (1 if "active cmd is bi-com" else 0) is needed in case $len(W) > len(U)$ and a left-only step must be taken before the next step happens on the right. The factor 2× compensates for that term. (Alternatively, a lexicographic order can be used.)

Now, we can prove the lemma. If the loop terminates because condition $U \nleq W \vee V \nleq X$ is false, then we have condition (a) of the Lemma. If it terminates because W faults next, then we have (b), using invariants $U \leq W \vee W \leq U$ and $V \leq X \vee X \leq V$, noting that we cannot have $W < U$ if W faults next, owing to fault determinacy of unary transitions (a corollary mentioned following Lemma A.6). Similarly, we get (c) if it terminates because X faults next. If it terminates because T faults, but the other cases do not hold, then we have (d) owing to the invariants $U \leq W \vee W \leq U$ and $V \leq X \vee X \leq V$. □

Definition C.10 (Denotation of Biprogram $[\![\Gamma|\Gamma' \vdash CC]\!]$). Suppose CC is wf in $\Gamma|\Gamma'$ and φ is a pre-model that includes all methods called in C. Let $[\![\Gamma|\Gamma' \vdash CC]\!]_\varphi$ to be the function of type $[\![\Gamma]\!] \times [\![\Gamma']\!] \to \mathbb{P}([\![\Gamma]\!] \times [\![\Gamma']\!]) \cup \{\lightning\}$ defined by

$$\begin{array}{rcl} [\![\Gamma|\Gamma' \vdash CC]\!]_\varphi(\sigma|\sigma') & \hat{=} & \{(\tau|\tau') \mid \langle CC,\ \sigma|\sigma',\ \_|\_\rangle \overset{\varphi}{\Longmapsto}{}^* \langle \lfloor \mathsf{skip} \rfloor,\ \tau|\tau',\ \_|\_\rangle\} \\ & & \cup\ (\{\lightning\} \text{ if } \langle CC,\ \sigma|\sigma',\ \_|\_\rangle \overset{\varphi}{\Longmapsto}{}^* \lightning \text{ else } \emptyset). \end{array}$$

Given a pre-model φ, biprogram CC, and relational formula $\mathcal{R}$, and method name m not called in CC and not in $dom(\varphi)$, one can extend the bi-model φ_2 by

$$\dot{\varphi}_2(m)(\sigma|\sigma') \mathrel{\hat{=}} (\{\lightning\} \text{ if } \neg\exists\pi.\ \sigma|\sigma' \models_\pi \mathcal{R} \text{ else } [\![CC]\!]_\varphi(\sigma|\sigma')). \tag{54}$$

To be precise, if precondition $\mathcal{R}$ has spec-only variables $\overline{s}, \overline{s}'$ on left and right, then the condition should say there are no values for these that satisfy: $\neg\exists\pi, \overline{v}, \overline{v}'.\ \sigma|\sigma' \models_\pi \mathcal{R}^{\overline{s},\overline{s}'}_{\overline{v},\overline{v}'}$.

Lemma C.11 (Denoted Relational Model). *(i) Suppose φ is a relational pre-model that includes all the methods in context calls in CC, and suppose m is not in φ. Suppose $\mathcal{R} \Rightarrow \langle\!\!\lbrack R \langle\!\!\lbrack \wedge \rbrack\!\!\rangle R' \rbrack\!\!\rangle$ is valid. Let $\dot{\varphi}$ extend φ with $\dot{\varphi}_2(m)$ given by Equation (54), $\dot{\varphi}_0(m)$ given by Equation (42) for $\overleftarrow{CC}$, R, and $\dot{\varphi}_1(m)$ given by Equation (42) for $\overrightarrow{CC}$, R'. Then $(\dot{\varphi}_0, \dot{\varphi}_1, \dot{\varphi}_2)$ is a pre-model.*

(ii) Suppose, in addition , that $\Phi \models CC : \mathcal{R} \approx\!> \mathcal{S}\ [\eta|\eta']$. Suppose $\dot{\Phi}$ extends Φ with $\dot{\Phi}_0(m) = R \rightsquigarrow S\ [\eta]$, $\dot{\Phi}_1(m) = R' \rightsquigarrow S'\ [\eta']$, and $\dot{\Phi}_2(m) = \mathcal{R} \approx\!> \mathcal{S}\ [\eta|\eta']$ such that $\dot{\Phi}$ is wf. If $\dot{\varphi}_0(m)$ and $\dot{\varphi}_1(m)$ are models for $R \rightsquigarrow S\ [\eta]$ and $R' \rightsquigarrow S'\ [\eta']$, respectively, then $\dot{\varphi}$ is a $\dot{\Phi}$-model.

Proof. (i) To show $\dot{\varphi}_2(m)$ is a pre-model (Definition 7.4), the fault, state, and divergence determinacy conditions follow from quasi-determinacy Lemma C.8 (cf. remark following projection Lemma 7.8).

Next, we show unary compatibility, i.e., $\tau|\tau' \in \dot{\varphi}_2(m)(\sigma|\sigma')$ implies $\tau \in \dot{\varphi}_0(m)(\sigma)$. and $\tau' \in \dot{\varphi}_1(m)(\sigma')$. Now $\tau|\tau' \in \dot{\varphi}_2(m)(\sigma|\sigma')$ iff $\langle CC,\ \sigma|\sigma',\ \_|\_\rangle \stackrel{\varphi}{\Longmapsto}{}^* \langle \lfloor \mathsf{skip} \rfloor,\ \tau|\tau',\ \_|\_\rangle$ and by projection Lemma 7.8 that implies $\langle \overleftarrow{CC},\ \sigma,\ \_\rangle \stackrel{\varphi_0}{\longmapsto}{}^* \langle \mathsf{skip},\ \tau,\ \_\rangle$ whence $\tau \in \dot{\varphi}_0(m)(\sigma)$ provided that $\sigma \models R$ (*mut. mut.* for the right side). Since $\tau|\tau' \in \dot{\varphi}_2(m)(\sigma|\sigma')$, there is some π for which $(\sigma|\sigma')$ satisfies $\mathcal{R}$, and by validity of $\mathcal{R} \Rightarrow \{\!|R|\!\} \wedge \{\!|R'|\!\}$ this implies $\sigma \models R$. Similarly for the right side.

For fault compatibility, suppose $\lightning \in \dot{\varphi}_0(m)(\sigma)$ or $\lightning \in \dot{\varphi}_1(m)(\sigma')$. Then either $\sigma \not\models R$ or $\sigma' \not\models R'$, by definitions, whence $\sigma|\sigma' \not\models \mathcal{R}$ owing to validity of $\mathcal{R} \Rightarrow \{\!|R|\!\} \wedge \{\!|R'|\!\}$. So $\lightning \in \dot{\varphi}_2(m)(\sigma|\sigma')$ as required.

To show relational compatibility, suppose $\tau \in \dot{\varphi}_0(m)(\sigma)$ and $\tau' \in \dot{\varphi}_1(m)(\sigma')$. We need $\dot{\varphi}_2(m)$ to contain either $\lightning$ or $(\tau|\tau')$. If there is no π with $\sigma|\sigma' \models_\pi \mathcal{R}$, then $\dot{\varphi}_2(m)$ is $\{\lightning\}$, and we are done. Otherwise, from $\tau \in \dot{\varphi}_0(m)(\sigma)$ and $\tau' \in \dot{\varphi}_1(m)(\sigma')$, we have traces $\langle C,\ \sigma,\ \_\rangle \stackrel{\varphi_0}{\longmapsto}{}^* \langle \mathsf{skip},\ \tau,\ \_\rangle$ and $\langle C',\ \sigma',\ \_\rangle \stackrel{\varphi_1}{\longmapsto}{}^* \langle \mathsf{skip},\ \tau',\ \_\rangle$. By embedding Lemma C.9, we get that either $\langle CC,\ \sigma|\sigma',\ \_|\_\rangle \stackrel{\varphi}{\Longmapsto}{}^* \langle \lfloor \mathsf{skip} \rfloor,\ \tau|\tau',\ \_|\_\rangle$ or else $\langle CC,\ \sigma|\sigma',\ \_|\_\rangle$ faults due to alignment conditions. Either way, we are done showing that $(\dot{\varphi}_0, \dot{\varphi}_1, \dot{\varphi}_2)$ is a pre-model.

(ii) Suppose that $\Phi \models CC : \mathcal{R} \approx\!> \mathcal{S}\,[\eta|\eta']$. The conditions of Definition 7.9 for $\dot{\varphi}_2(m)$ with respect to $\mathcal{R} \approx\!> \mathcal{S}\,[\eta]$ are direct consequences of $\Phi \models CC : \mathcal{R} \approx\!> \mathcal{S}\,[\eta|\eta']$ and (54). □

Theorem 7.11 (Adequacy). *Consider a valid judgment $\Phi \models_M CC :\ \mathcal{P} \approx\!> \mathcal{Q}\,[\varepsilon|\varepsilon']$. Consider any Φ-model φ and any σ, σ', π with $\sigma|\sigma' \models_\pi \mathcal{P}$. If $\langle \overleftarrow{CC},\ \sigma,\ \_\rangle \stackrel{\varphi_0}{\longmapsto}{}^* \langle \mathsf{skip},\ \tau,\ \_\rangle$ and $\langle \overrightarrow{CC},\ \sigma',\ \_\rangle \stackrel{\varphi_1}{\longmapsto}{}^* \langle \mathsf{skip},\ \tau',\ \_\rangle$, then $\tau|\tau' \models_\pi \mathcal{Q}$. Moreover, all executions from $\langle \overleftarrow{CC},\ \sigma,\ \_\rangle$ and from $\langle \overrightarrow{CC},\ \sigma',\ \_\rangle$ satisfy Safety, Write, R-safe, and Encap in Definition 5.10.*

Proof. Let U, V be the traces and let T be the biprogram trace given by embedding Lemma C.9. The judgment for CC is applicable to T, so cases (b), (c), and (d) in the Lemma are ruled out—T cannot fault. The remaining case is (a), that is, T covers every step of U and V. If U and V are terminated, then so is T, whence the postcondition holds, and the Write condition holds, by validity of the judgment. Regardless of termination, we also get the unary Safety and Encap conditions for U and V, by definitions, since every step is covered by T. □

D RELATIONAL LOGIC AND ITS SOUNDNESS (RE SECTION 8)

Theorem 8.1 (Soundness of Relational Logic). *All the relational proof rules are sound (Figure 30 and Appendix Figure 38).*

Appendix D.1 presents relational proof rules omitted from the body of the article. Section D.2 proves the crucial lockstep alignment lemma. The soundness proofs comprise Appendices D.3–D.11; these are largely independent and need not be read in any particular order.

D.1 Additional Rules

Figure 38 presents the proof rules omitted in the body of the article.

Rule rIf is typical of relational Hoare logics, with the addition of side conditions to ensure encapsulation. Similarly, rules rSeq and rWhile have the same immunity conditions as their unary counterparts. Rules rWhile and rSeq are slightly simplified from the general rules, for clarity. The general rules should include an initial snapshot $r = \mathsf{alloc}$, and region H and field list $\overline{f}$, with conditions to ensure that H contains only freshly allocated objects so writes of $H`\overline{f}$ can be omitted from the frame condition. This caters for writes to locations allocated in the first command of a sequence, or previous iterations of a loop, just as it is done in the unary Seq and While rules (Figure 35). (The details are justified in RLI, though in RLI the rules are slightly more succinct owing to use of freshness effect notation.)

$$\textsc{rEmbS}\ \frac{\Phi_0 \vdash A : P \rightsquigarrow Q\ [\varepsilon] \qquad \Phi_1 \vdash A : P' \rightsquigarrow Q'\ [\varepsilon']}{\Phi \vdash \lfloor A \rfloor : \langle\!|P|\!\rangle \wedge [\![P']\!] \approx\!> \langle\!|Q|\!\rangle \wedge [\![Q']\!]\ [\varepsilon|\varepsilon']}$$

$$\textsc{rSeq}\ \frac{\Phi \vdash CC_1 : \mathcal{P} \approx\!> \mathcal{P}_1\ [\varepsilon_1|\varepsilon_1'] \qquad \Phi \vdash CC_2 : \mathcal{P}_1 \approx\!> \mathcal{Q}\ [\varepsilon_2|\varepsilon_2'] \qquad \varepsilon_2 \text{ is } \overleftarrow{\mathcal{P}}/\varepsilon_1\text{-immune} \qquad \varepsilon_2' \text{ is } \overrightarrow{\mathcal{P}}/\varepsilon_1'\text{-immune}}{\Phi \vdash CC_1 ; CC_2 : \mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon_1, \varepsilon_2|\varepsilon_1', \varepsilon_2']}$$

$$\textsc{rIf}\ \frac{\begin{array}{c}\Phi \vdash_M CC : \mathcal{P} \wedge \langle\!|E|\!\rangle \wedge [\![E']\!] \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon'] \qquad \Phi \vdash_M DD : \mathcal{P} \wedge \langle\!|\neg E|\!\rangle \wedge [\![\neg E']\!] \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon'] \\ \mathcal{P} \Rightarrow E \doteq E' \qquad \delta = (+N \in \Phi, N \neq M.\ bnd(N)) \qquad \delta \cdot\!/.\ r2w(ftpt(E)) \qquad \delta \cdot\!/.\ r2w(ftpt(E'))\end{array}}{\Phi \vdash_M \text{if } E|E' \text{ then } CC \text{ else } DD : \mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon, ftpt(E)|\varepsilon', ftpt(E')]}$$

$$\textsc{rWhile}\ \frac{\begin{array}{c}\Phi \vdash CC : \mathcal{Q} \wedge \neg\mathcal{P} \wedge \neg\mathcal{P}' \wedge \langle\!|E|\!\rangle \wedge [\![E']\!] \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon'] \\ \Phi \vdash (\overleftarrow{CC}|\text{skip}) : \mathcal{Q} \wedge \mathcal{P} \wedge \langle\!|E|\!\rangle \approx\!> \mathcal{Q}\ [\varepsilon|\bullet] \qquad \Phi \vdash (\text{skip}|\overrightarrow{CC}) : \mathcal{Q} \wedge \mathcal{P}' \wedge [\![E']\!] \approx\!> \mathcal{Q}\ [\bullet|\varepsilon'] \\ (+N \in \Phi, N \neq M.\ bnd(N)) \cdot\!/.\ r2w(ftpt(E)) \qquad (+N \in \Phi, N \neq M.\ bnd(N)) \cdot\!/.\ r2w(ftpt(E')) \\ \mathcal{Q} \Rightarrow E \doteq E' \vee (\mathcal{P} \wedge \langle\!|E|\!\rangle) \vee (\mathcal{P}' \wedge [\![E']\!]) \qquad \varepsilon \text{ is } \overleftarrow{\mathcal{Q}}/\varepsilon\text{-immune} \qquad \varepsilon' \text{ is } \overrightarrow{\mathcal{Q}}/\varepsilon'\text{-immune}\end{array}}{\Phi \vdash \text{while } E|E' \cdot \mathcal{P}|\mathcal{P}' \text{ do } CC : \mathcal{Q} \approx\!> \mathcal{Q} \wedge \langle\!|\neg E|\!\rangle \wedge [\![\neg E']\!]\ [\varepsilon, ftpt(E)|\varepsilon', ftpt(E')]}$$

$$\textsc{rIf4}\ \frac{\begin{array}{c}\Phi \vdash_M (C|C') : \mathcal{P} \wedge \langle\!|E|\!\rangle \wedge [\![E']\!] \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon'] \qquad \Phi \vdash_M (C|D') : \mathcal{P} \wedge \langle\!|E|\!\rangle \wedge [\![\neg E']\!] \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon'] \\ \Phi \vdash_M (D|C') : \mathcal{P} \wedge \langle\!|\neg E|\!\rangle \wedge [\![E']\!] \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon'] \qquad \Phi \vdash_M (D|D') : \mathcal{P} \wedge \langle\!|\neg E|\!\rangle \wedge [\![\neg E']\!] \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon'] \\ \delta = (+N \in \Phi, N \neq M.\ bnd(N)) \qquad \delta \cdot\!/.\ r2w(ftpt(E)) \qquad \delta \cdot\!/.\ r2w(ftpt(E'))\end{array}}{\Phi \vdash_M (\text{if } E \text{ then } C \text{ else } D \mid \text{if } E' \text{ then } C' \text{ else } D') : \mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon, ftpt(E)|\varepsilon', ftpt(E')]}$$

$$\textsc{rVar}\ \frac{\Phi \vdash^{\Gamma, x:T|\Gamma', x':T'} CC : \mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon']}{\Phi \vdash^{\Gamma|\Gamma'} \text{var } x{:}T|x'{:}T' \text{ in } CC : \mathcal{P} \wedge \langle\!|x = default(T)|\!\rangle \wedge [\![x' = default(T')]\!] \approx\!> \mathcal{Q}\ [\varepsilon|\varepsilon']}$$

Fig. 38. Relational proof rules omitted from Figure 30.

Remark 10. As in the unary While, the frame condition in rWhile needs to include the footprint of the loop tests ($ftpt(E)$, $ftpt(E')$) as the behavior depends on them. Given that the alignment guards $\mathcal{P}$ and $\mathcal{P}'$ influence the bi-while transitions, one may expect that their footprints should also be included. But the dependency of r-respect (Encap) is about execution on one side. The value of E (respectively, E') determines the control state (i.e., unfold the loop body or terminate) at the unary level. By contrast, the value of $\mathcal{P}$ (respectively, $\mathcal{P}'$) determines the biprogram control state. This is reflected in the unary control state, but during a one-sided iteration the other side stutters; and stuttering transitions are removed (by projection, see Lemma 7.8) according to the definition of Encap in Definition 7.10.

Remark 11. Rule rWhile can be slightly strengthened to take into account that in our semantics, to ensure quasi-determinacy, a right iteration only happens when the left guard or test is false. We prefer the more symmetric phrasing of the rule: What matters is that one-sided executions under their designated alignment guard maintain the invariant. The deterministic scheduling is a technical artifact, just like the specific details of the dovetailed execution of the bi-com construct are not important for reasoning.

D.2 Proof of Lockstep Alignment Lemma

Lemma 8.3. *If* $\tau \models snap(\varepsilon)$ *and* $\tau{\rightarrow}v \models \varepsilon$, *then* $wlocs(\tau, \varepsilon)\backslash rlocs(v, \delta^{\oplus}) = rlocs(v, Asnap(\varepsilon)\backslash\delta)$.

Proof. Assume $\tau \models snap(\varepsilon)$ and $\tau{\rightarrow}v \models \varepsilon$. The equality $wlocs(\tau, \varepsilon)\backslash rlocs(v, \delta^{\oplus}) = rlocs(v, Asnap(\varepsilon)\backslash\delta)$ is between sets of locations, i.e., variables and heap locations. We consider the two kinds of location in turn.

For variables, we have $x \in wlocs(\tau, \varepsilon)\backslash rlocs(v, \delta^{\oplus})$ iff $\mathrm{wr}\, x$ is in ε and $\mathrm{rd}\, x$ is not in $\delta^{\oplus}$, by definitions. However, by definition of *Asnap*, we have $x \in rlocs(v, Asnap(\varepsilon)\backslash\delta)$ iff $\mathrm{rd}\, x$ is not in δ and $\mathrm{wr}\, x$ is in ε and $x \not\equiv \mathrm{alloc}$. The conditions are equivalent.

For heap locations, w.l.o.g., we assume ε and δ are in normal form and have exactly one read and one write effect for each field. We are only concerned with writes in ε and reads in δ. Consider any field name f and suppose ε contains $\mathrm{wr}\, G`f$ and $\delta$ contains $\mathrm{rd}\, H`f$ for some G, H. Now for location $o.f$, we have

$$\begin{array}{ll} & o.f \in \mathit{wlocs}(\tau, \varepsilon) \backslash \mathit{rlocs}(v, \delta^{\oplus}) \\ \iff & o \in \tau(G) \backslash v(H) \quad \text{by defs } \mathit{wlocs}, \mathit{rlocs} \text{ and normal form} \\ \iff & o \in \tau(s_{G,f}) \backslash v(H) \quad \text{by } \tau \models \mathit{snap}(\varepsilon), \text{ we have } \tau(s_{G,f}) = \tau(G) \\ \iff & o \in v(s_{G,f}) \backslash v(H) \quad \text{by } \tau \to v \models \varepsilon \text{ and } \mathrm{wr}\, s_{G,f} \notin \varepsilon \text{ have } \tau(s_{G,f}) = v(s_{G,f}) \\ \iff & o \in v(s_{G,f} \backslash H) \quad \text{by semantics of subtraction.} \end{array}$$

However,

$$\begin{array}{ll} & o.f \in \mathit{rlocs}(v, \mathit{Asnap}(\varepsilon) \backslash \delta) \\ \iff & o.f \in \mathit{rlocs}(v, (\mathrm{rd}\, s_{G,f}`f \backslash \mathrm{rd}\, H`f)) \quad \text{by def } \mathit{Asnap} \text{ and assumption about } G, H \\ \iff & o.f \in \mathit{rlocs}(v, \mathrm{rd}\, (s_{G,f} \backslash H)`f) \quad \text{by effect subtraction} \\ \iff & o \in v(s_{G,f} \backslash H) \quad \text{by def } \mathit{rlocs}. \end{array}$$

The conditions are equivalent. □

Lemma 8.9 (Lockstep Alignment). *Suppose*

(i) $\Phi \Rrightarrow \mathit{LocEq}_{\delta}(\Psi)$ and φ is a Φ-model, where $\delta = (+N \in \Psi, N \neq M.\ \mathit{bnd}(N))$,
(ii) $\sigma|\sigma' \models_{\pi} \mathit{pre}(\mathit{locEq}_{\delta}(P \rightsquigarrow Q\ [\varepsilon]))$,
(iii) T is a trace $\langle \llbracket C \rrbracket,\ \sigma|\sigma',\ \_|\_ \rangle \overset{\varphi}{\Longmapsto}{}^{*} \langle BB,\ \tau|\tau',\ \mu|\mu' \rangle$ and C is let-free,
(iv) Let U, V be the projections of T. Then U (respectively, V) is r-safe for $(\Phi_0, \varepsilon, \sigma)$ (respectively, for $(\Phi_1, \varepsilon, \sigma')$) and respects $(\Phi_0, M, \varphi_0, \varepsilon, \sigma)$ (respectively, $(\Phi_1, M, \varphi_1, \varepsilon, \sigma')$).

Then there are B, ρ, with

(v) $BB \equiv \llbracket B \rrbracket$, $\rho \supseteq \pi$, and $\mu = \mu'$,
(vi) $\mathit{Lagree}(\tau, \tau', \rho, (\mathit{freshL}(\sigma, \tau) \cup \mathit{rlocs}(\sigma, \varepsilon) \cup \mathit{wrttn}(\sigma, \tau)) \backslash \mathit{rlocs}(\tau, \delta^{\oplus}))$, and
(vii) $\mathit{Lagree}(\tau', \tau, \rho^{-1}, (\mathit{freshL}(\sigma', \tau') \cup \mathit{rlocs}(\sigma', \varepsilon) \cup \mathit{wrttn}(\sigma', \tau')) \backslash \mathit{rlocs}(\tau', \delta^{\oplus}))$.

Proof. As usual write $\hat{\sigma}, \hat{\sigma}'$ for the extensions of σ, σ' for the spec-only variables of the precondition, as per (ii).

We show that the conditions (v–vii) hold at every step within T, by induction on steps.[47] One might expect that the lemma could be simplified to simply say the conditions hold at every reachable step, without mentioning traces, but we are assuming rather than proving that the r-safety and r-respect conditions hold, so the present formulation seems more clear.

Base Case. For initial configuration $\langle \llbracket C \rrbracket, \sigma|\sigma', \_|\_ \rangle$, we have $\mathit{freshL}(\sigma, \sigma) = \varnothing = \mathit{freshL}(\sigma', \sigma')$ and $\mathit{wrttn}(\sigma, \sigma) = \varnothing = \mathit{wrttn}(\sigma', \sigma')$. From hypothesis (ii) of the Lemma, and the semantics of the agreement formulas in the precondition, we get $\mathit{Agree}(\sigma, \sigma', \pi, \varepsilon_{\delta}^{\leftarrow})$ and $\mathit{Agree}(\sigma', \sigma, \pi^{-1}, \varepsilon_{\delta}^{\leftarrow})$. Unfolding definitions, we have proved the claim with $\rho, \tau, \tau' := \pi, \sigma, \sigma'$.

Induction case. Suppose $\langle \llbracket C \rrbracket, \sigma|\sigma', \_|\_ \rangle \overset{\varphi}{\Longmapsto}{}^{*} \langle BB, \tau|\tau', \mu|\mu' \rangle \overset{\varphi}{\Longmapsto} \langle DD, v|v', \nu|\nu' \rangle$ as a prefix of T. By induction hypothesis, we have $\mu = \mu'$, $BB = \llbracket B \rrbracket$ for some B, and for some $\rho \supseteq \pi$, we have

$$\begin{array}{c} \mathit{Lagree}(\tau, \tau', \rho, (\mathit{freshL}(\sigma, \tau) \cup \mathit{rlocs}(\sigma, \varepsilon) \cup \mathit{wrttn}(\sigma, \tau)) \backslash \mathit{rlocs}(\tau, \delta^{\oplus})), \\ \mathit{Lagree}(\tau', \tau, \rho^{-1}, (\mathit{freshL}(\sigma', \tau') \cup \mathit{rlocs}(\sigma', \varepsilon) \cup \mathit{wrttn}(\sigma', \tau')) \backslash \mathit{rlocs}(\tau', \delta^{\oplus})). \end{array} \tag{55}$$

[47] We are glossing over the local variables introduced by local blocks. To be precise, the initial states are both for Γ and have no extra variables. The Lemma should have additional conclusion that $\mathit{Vars}(\tau) = \mathit{Vars}(\tau')$, which becomes part of the induction hypothesis, to account for possible addition of locals, which will be in *freshL*.

Without loss of generality, we assume that $[\![B]\!] \equiv [\![B_0]\!]; [\![B_1]\!]$, where $Active(B) \equiv B_0$. (Recall by Lemma C.7 that $Active[\![B]\!] = [\![ActiveB]\!]$.)

To find D and an extension of ρ, such that the agreements for $v|v'$ and other conditions hold for the step $\langle BB,\ \tau|\tau',\ \mu|\mu'\rangle \overset{\varphi}{\longmapsto\!\!\!\!\!\Rightarrow} \langle DD,\ v|v',\ \nu|\nu'\rangle$, we go by cases on the possible transition rules. The fault rules are not relevant.

Cases bComL, bComR, bComR0, bWhL, and bWhR are not applicable to $[\![B]\!]$.

Case bSync. So B_0 is an atomic command other than a method call and there are unary transitions $\langle B_0,\ \tau,\ \mu\rangle \overset{\varphi_0}{\longmapsto} \langle \mathsf{skip},\ v,\ \mu\rangle$ and $\langle B_0,\ \tau',\ \mu'\rangle \overset{\varphi_1}{\longmapsto} \langle \mathsf{skip},\ v',\ \mu'\rangle$. The successor configuration has $DD \equiv [\![B_1]\!]$ and $\nu = \mu = \mu' = \nu'$. Because the step is not a method call, the same transitions can be taken via the other models, i.e., we have $\langle B_0,\ \tau,\ \mu\rangle \overset{\varphi_1}{\longmapsto} \langle \mathsf{skip},\ v,\ \mu\rangle$ and $\langle B_0,\ \tau',\ \mu'\rangle \overset{\varphi_0}{\longmapsto} \langle \mathsf{skip},\ v',\ \mu'\rangle$. Moreover, owing to the agreements, we can instantiate the left and right trace's respect condition (hypothesis (iv) of this Lemma). As we are considering a non-call command, the collective boundary for r-respect is $\dot{\delta} = (+N \in (\Psi, \mu), N \neq topm(B, M).\ bnd(N))$. By hypothesis (iii) of the Lemma, C is let-free. So μ is empty. Moreover, there is no ecall in B, there being no environment calls (and as always the starting command has no end markers), so $topm(B, M) = M$. So the collective boundary for r-respect is the δ assumed in the Lemma, i.e., $\delta = (+N \in \Psi, N \neq M.\ bnd(N))$. Both steps satisfy w-respect, i.e., do not write inside the boundary, owing to hypothesis (iv) of the Lemma. Instantiating r-respect twice (with $\tau, \tau', \varphi_0, \rho$ and with $\tau', \tau, \varphi_1, \rho^{-1}$), we have the allowed dependencies $\tau, \tau' \overset{\rho}{\Longrightarrow} v, v' \models^{\sigma}_{\delta} \varepsilon$ and $\tau', \tau \overset{\rho^{-1}}{\Longrightarrow} v', v \models^{\sigma'}_{\delta} \varepsilon$. Even more, r-respects applied to Equation (55) gives some $\dot{\rho}$ and $\dot{\rho}'$ with $\dot{\rho} \supseteq \rho$ and $\dot{\rho}' \supseteq \rho^{-1}$ and the following four conditions:

$$\begin{array}{l} Lagree(v, v', \dot{\rho}, (freshL(\tau, v) \cup wrttn(\tau, v))\backslash rlocs(v, \delta^{\oplus})), \\ \dot{\rho}(freshL(\tau, v)\backslash rlocs(v, \delta)) \subseteq freshL(\tau', v')\backslash rlocs(v', \delta), \\ Lagree(v', v, \dot{\rho}', (freshL(\tau', v') \cup wrttn(\tau', v'))\backslash rlocs(v', \delta^{\oplus})), \\ \dot{\rho}'(freshL(\tau', v')\backslash rlocs(v', \delta)) \subseteq freshL(\tau, v)\backslash rlocs(v, \delta). \end{array} \tag{56}$$

By balanced symmetry Lemma A.3, we get

$$\begin{array}{l} Lagree(v', v, \dot{\rho}^{-1}, (freshL(\tau', v') \cup wrttn(\tau', v'))\backslash rlocs(v', \delta^{\oplus})), \\ \dot{\rho}(freshL(\tau, v)\backslash rlocs(v, \delta)) = freshL(\tau', v')\backslash rlocs(v', \delta). \end{array}$$

We can use preservation Lemma A.4 for these three sets of locations (which are subsets of $locations(\tau)$): $rlocs(\sigma, \varepsilon)\backslash rlocs(\tau, \delta^{\oplus})$, $wrttn(\sigma, \tau)\backslash rlocs(\tau, \delta^{\oplus})$, and $freshL(\sigma, \tau)\backslash rlocs(\tau, \delta^{\oplus})$. By Lemma A.4, we get

$$Lagree(v, v', \dot{\rho}, ((freshL(\sigma, \tau) \cup rlocs(\sigma, \varepsilon) \cup wrttn(\sigma, \tau))\backslash rlocs(\tau, \delta^{\oplus}))\backslash rlocs(v, \delta^{\oplus})).$$

So, by the boundary monotonicity condition of Encap, we have $rlocs(\tau, \delta^{\oplus}) \subseteq rlocs(v, \delta^{\oplus})$. Now from this and Equation (56), using $freshL(\sigma, v) = freshL(\sigma, \tau) \cup freshL(\tau, v)$ and $wrttn(\sigma, v) \subseteq wrttn(\sigma, \tau) \cup wrttn(\tau, v)$, we can combine the agreements together to get

$$Lagree(v, v', \dot{\rho}, (freshL(\sigma, v) \cup rlocs(\sigma, \varepsilon) \cup wrttn(\sigma, v))\backslash rlocs(v, \delta^{\oplus})).$$

With a similar argument, we obtain the symmetric condition

$$Lagree(v', v, \dot{\rho}^{-1}, (freshL(\sigma', v') \cup rlocs(\sigma', \varepsilon) \cup wrttn(\sigma', v'))\backslash rlocs(v', \delta^{\oplus})),$$

which finishes this case for the induction step.

Case bCallS. So B_0 is $m()$ for some m, and $(v|v') \in \varphi_2(m)(\tau|\tau')$. The successor configuration has $DD \equiv [\![B_1]\!]$ and $\nu = \mu = \mu' = \nu'$. Suppose $\Psi(m)$ is $R \rightsquigarrow S\ [\eta]$. By the assumed r-safe condition (hypothesis (iv) of the Lemma), we have $rlocs(\tau, \eta) \subseteq freshL(\sigma, \tau) \cup rlocs(\sigma, \varepsilon)$. Since $\varphi_2(m)(\tau|\tau') \neq \lightning$, there must be values for the spec-only variables $\overline{t}$ of m's spec for which $\tau|\tau'$ satisfy the method's precondition, which by hypothesis (i) of the lemma implies the precondition of $locEq_{\delta}(\Psi(m))$. That

is, there are $\overline{u}$ and $\overline{u}'$ such that $\hat{\tau}|\hat{\tau}' \models_\rho \mathbb{B}R \wedge \mathbb{A}(rds(\eta)\backslash\delta^\oplus) \wedge \mathbb{B}(s^m_{\text{alloc}} = \text{alloc} \wedge snap^m(\eta))$, where $\hat{\tau} = [\tau{+}\overline{t}{:}\overline{u}]$ and $\hat{\tau}' = [\tau'{+}\overline{t}{:}\overline{u}']$. (Apropos the identifier s^m_{alloc}, see Footnote 38.) Since $\varphi \models \Phi$ and $(v|v') \in \varphi_2(m)(\tau|\tau')$, we get the postcondition of $\Phi(m)$, which implies that of $locEq_\delta(\Psi(m))$. Hence, $\hat{v}|\hat{v}' \models_\rho \Diamond(\mathbb{B}Q \wedge \mathbb{A}\eta^{\rightarrow}_\delta)$, where $\hat{v} = [v{+}\overline{t}{:}\overline{u}]$, $\hat{v}' = [v'{+}\overline{t}{:}\overline{u}']$, and

$$\eta^{\rightarrow}_\delta \equiv (\text{rd}\,(\text{alloc}\backslash s^m_{\text{alloc}})\text{`any}, Asnap^m(\eta))\backslash\delta. \tag{57}$$

So by semantics of $\Diamond$ and $\mathbb{A}$ there is $\dot{\rho} \supseteq \rho$ with $Agree(\hat{v}, \hat{v}', \dot{\rho}, \eta^{\rightarrow}_\delta)$ and $Agree(\hat{v}', \hat{v}, \dot{\rho}^{-1}, \eta^{\rightarrow}_\delta)$. We have $freshL(\tau, v) = rlocs(v, \text{rd}\,(\text{alloc}\backslash s^m_{\text{alloc}})\text{`any})$ and $freshL(\tau', v') = rlocs(v', \text{rd}\,(\text{alloc}\backslash s^m_{\text{alloc}})\text{`any})$. We also have $wrttn(\tau, v) \subseteq wlocs(\tau, \eta)$ and $wrttn(\tau', v') \subseteq wlocs(\tau', \eta)$, from $\tau{\rightarrow}v \models \eta$ and $\tau'{\rightarrow}\hat{v}' \models \eta$. Furthermore, by Lemma 8.3, we have

$$wlocs(\tau, \eta)\backslash rlocs(v, \delta^\oplus) = rlocs(v, Asnap^m(\eta)\backslash\delta) \subseteq rlocs(v, \eta^{\rightarrow}_\delta),$$
$$wlocs(\tau', \eta)\backslash rlocs(v', \delta^\oplus) = rlocs(v', Asnap^m(\eta)\backslash\delta) \subseteq rlocs(v', \eta^{\rightarrow}_\delta).$$

So, we have

$$Lagree(v, v', \dot{\rho}, (freshL(\tau, v) \cup wrttn(\tau, v))\backslash rlocs(v, \delta^\oplus)), \tag{58}$$

$$Lagree(v', v, \dot{\rho}^{-1}, (freshL(\tau', v') \cup wrttn(\tau', v'))\backslash rlocs(v', \delta^\oplus)). \tag{59}$$

Thus, we have $\tau, \tau' \overset{\rho}{\Rightarrow} v, v' \models^\sigma_\delta \eta$ and $\tau', \tau \overset{\rho^{-1}}{\Rightarrow} v', v \models^{\sigma'}_\delta \eta$. Since $rlocs(\sigma, \varepsilon)\backslash rlocs(\tau, \delta^\oplus)$, $wrttn(\sigma, \tau)\backslash rlocs(\tau, \delta^\oplus)$ and $freshL(\sigma, \tau)\backslash rlocs(\tau, \delta^\oplus)$ are subsets of $locations(\tau)$, using Lemma A.4, from Equation (55), we get

$$Lagree(v, v', \dot{\rho}, ((freshL(\sigma, \tau) \cup rlocs(\sigma, \varepsilon) \cup wrttn(\sigma, \tau))\backslash rlocs(\tau, \delta^\oplus))\backslash rlocs(v, \delta^\oplus)).$$

By hypothesis (iv) of the Lemma, the steps satisfy boundary monotonicity, i.e., $rlocs(\tau, \delta) \subseteq rlocs(v, \delta)$, which implies $rlocs(\tau, \delta^\oplus) \subseteq rlocs(v, \delta^\oplus)$. Combining this with the agreements of Equation (58), we get

$$Lagree(v, v', \dot{\rho}, (freshL(\sigma, v) \cup rlocs(\sigma, \varepsilon) \cup wrttn(\sigma, v))\backslash rlocs(v, \delta^\oplus)).$$

With a similar argument using Equation (59), we get the symmetric condition

$$Lagree(v', v, \dot{\rho}^{-1}, (freshL(\sigma', v') \cup rlocs(\sigma', \varepsilon) \cup wrttn(\sigma', v'))\backslash rlocs(v', \delta^\oplus)),$$

which completes this case.

Case bCall0. So B_0 is a context call $m()$ that stutters, because the $\varphi_2(m)$ is empty. The agreements are maintained, as nothing changes.

Case bVar. This relies on the additional condition that $Vars(\tau) = Vars(\tau')$, which can be included in the induction hypothesis but is omitted for readability. We have that B_0 is var $x{:}T$ in B_2 for some x, T, B_2, so $\llbracket B_0 \rrbracket \equiv$ var $x{:}T|x{:}T$ in $\llbracket B_2 \rrbracket$. Because $Vars(\tau) = Vars(\tau')$, and using the assumption that *FreshVar* depends only on *Vars*() of the state (Equation (39)), we have some w with $w = FreshVar(\tau) = FreshVar(\tau')$. This ensures $Vars(v) = Vars(v')$, justifying the omitted induction hypothesis; the only other change to variables is by dropping them, by bSync transition for $\lfloor \text{evar}(w) \rfloor$. The step from var $x{:}T|x{:}T$ in $\llbracket B_2 \rrbracket$ goes to $\langle \llbracket B_2 \rrbracket^{x,x}_{w,w}; \lfloor \text{evar}(w) \rfloor; \llbracket B_1 \rrbracket, v|v', \mu|\mu' \rangle$ where $v = [\tau{+}w{:}\,default(T)]$ and $v' = [\tau'{+}w'{:}\,default(T')]$. We get the agreements, because nothing changes except the addition of w with default value. We get the code alignment, because $\llbracket B_2 \rrbracket^{x,x}_{w,w} \equiv \llbracket B_2{}^{x,x}_{w,w} \rrbracket$ by definitions.

Cases bIfTT and bIfFF. So B_0 has the form if E then B_2 else B_3 and the successor configuration has the form either $\llbracket B_2 \rrbracket; \llbracket B_1 \rrbracket$ or $\llbracket B_3 \rrbracket; \llbracket B_1 \rrbracket$. Nothing else changes so the agreements are maintained.

Cases bWhTT and bWhFF. So B_0 has the form while E do B_2 and the successor configuration has the form either $[\![B_2]\!];[\![B_0]\!];[\![B_1]\!]$ (for bWhTT) or $[\![B_1]\!]$. Nothing else changes so the agreements are maintained.

Case bCallE does not occur, because C is let-free.

Case bLet does not occur, because C is let-free.

D.3 Soundness of rLocEq

$$\textsc{rLocEq}\ \frac{\Phi \vdash_M C :\ P \rightsquigarrow Q\ [\varepsilon] \qquad P \models w2r(\varepsilon) \le rds(\varepsilon) \qquad \delta = (+N \in \Phi, N \neq M.\ bnd(N)) \qquad C \text{ is let-free}}{LocEq_\delta(\Phi) \vdash_M [\![C]\!] : locEq_\delta(P \rightsquigarrow Q\ [\varepsilon])}$$

Let $\varepsilon^{\leftarrow}_{\delta} \triangleq rds(\varepsilon)\backslash\delta^{\oplus}$ as in Definition 8.4 of $locEq_\delta(P \rightsquigarrow Q\ [\varepsilon])$. Let φ be a $LocEq_\delta(\Phi)$-model, i.e., φ_0 and φ_1 are Φ-models and φ_2 satisfies Φ_2, which is given by applying the $locEq_\delta$ construction to each spec in Φ as per Definition 8.4. In symbols: $(\varphi_0, \varphi_1, \varphi_2) \models (\Phi, \Phi, locEq_\delta(\Phi))$. Suppose $\overline{s}$ are the spec-only variables of $P \rightsquigarrow Q\ [\varepsilon]$, and suppose σ, σ' satisfy the precondition, for the unique snapshot values $\overline{v}$ and $\overline{v}'$ of $\overline{s}$ on left and right (cf. Lemma C.1). That is,

$$\hat{\sigma}|\hat{\sigma}' \models_\pi \mathbb{B}P \wedge \mathbb{A}\varepsilon^{\leftarrow}_{\delta} \wedge \mathbb{B}(r = \text{alloc} \wedge snap(\varepsilon)) \text{ where } \hat{\sigma} = [\sigma{+}\overline{s}{:}\,\overline{v}] \text{ and } \hat{\sigma}' = [\sigma'{+}\overline{s}{:}\,\overline{v}']. \tag{60}$$

Notice that these assumptions entail hypotheses (i) and (ii) of Lemma 8.9, to which we will appeal repeatedly. We instantiate Φ in the Lemma by $LocEq_\delta(\Phi)$, and the initial states $\sigma|\sigma'$ satisfy the requisite precondition.

Encap. Consider any trace T from $\langle [\![C]\!],\ \sigma|\sigma',\ \_|\_\rangle$. Recall that $(LocEq_\delta(\Phi))_0 = \Phi$ and $(LocEq_\delta(\Phi))_1 = \Phi$. So according to Definition 7.10, we must prove that the projections U (respectively, V) of T (by projection Lemma 7.8) satisfy r-safe for $(\Phi, \varepsilon, \sigma)$ (respectively, $(\Phi, \varepsilon, \sigma')$), and respect for $(\Phi, M, \varphi_0, \varepsilon, \sigma)$ (respectively, $(\Phi, M, \varphi_1, \varepsilon, \sigma')$). These are both traces of C from P-states, and φ_0, φ_1 are Φ-models, so we get r-safe and respect by two instantiations of the premise.

Write. A terminated trace via φ provides terminated unary traces via φ_0 and φ_1 The initial states satisfy the precondition P of the premise, and we get the Write property directly from two instantiations of the premise.

Safety. Suppose $\langle [\![C]\!],\ \sigma|\sigma',\ \_|\_\rangle \overset{\varphi}{\Longrightarrow}{}^* \langle BB,\ \tau|\tau',\ \mu|\mu'\rangle \overset{\varphi}{\Longrightarrow} \lightning$. We can apply Lemma 8.9 to the trace ending in BB. The lemma requires the trace to satisfy exactly the r-safe and respects conditions that are established above for Encap. By Lemma 8.9 there are B, ρ with $BB \equiv [\![B]\!]$, $\rho \supseteq \pi, \mu = \mu'$,

$$\begin{array}{l} Lagree(\tau, \tau', \rho, (freshL(\sigma, \tau) \cup rlocs(\sigma, \varepsilon) \cup wrttn(\sigma, \tau))\backslash rlocs(\tau, \delta^{\oplus})), \\ Lagree(\tau', \tau, \rho^{-1}, (freshL(\sigma', \tau') \cup rlocs(\sigma', \varepsilon) \cup wrttn(\sigma', \tau'))\backslash rlocs(\tau', \delta^{\oplus})). \end{array} \tag{61}$$

We show that $\langle BB,\ \tau|\tau',\ \mu|\mu'\rangle$ does not fault, by contradiction, going by cases on the possible transition rules that yield fault.

- bSyncX would give a unary fault via φ_0 or φ_1, contrary to the premise.
- bCallX applies if $\lightning$ is returned by $\varphi_2(m)$, and because φ_2 is a context model, that means $\tau|\tau'$ falsifies the precondition for m. Suppose that $\Phi(m) = R \rightsquigarrow S\ [\eta]$. The precondition includes $\mathbb{B}(s^m_{\text{alloc}} = \text{alloc} \wedge snap^m(\eta))$, which uses spec-only variables that do not occur in R, δ, or η, and which can be satisfied by values determined by $\tau|\tau'$. So for the precondition to be false there must be no $\rho, \overline{u}, \overline{u}'$ such that $\rho \supseteq \pi$ and $\hat{\tau}|\hat{\tau}' \models_\rho \mathbb{B}R \wedge \mathbb{A}rds(\eta)\backslash\delta^{\oplus}$ where $\hat{\tau} = [\tau{+}\overline{t}{:}\,\overline{u}]$ and $\hat{\tau}' = [\tau'{+}\overline{t}{:}\,\overline{u}']$. From fault and relational compatibility (Definition 7.4), we have

$$\lightning \in \varphi_0(m)(\tau) \vee \lightning \in \varphi_1(m)(\tau') \vee (v \in \varphi_0(m)(\tau) \wedge v' \in \varphi_1(m)(\tau')).$$

From the premise, it is not the case that $\lightning \in \varphi_0(m)(\tau)$ or $\lightning \in \varphi_1(m)(\tau')$, so there must be $\overline{u}$ and $\overline{u}'$ such that $\hat{\tau} \models R \wedge \hat{\tau}' \models R$ (with $\hat{\tau}, \hat{\tau}$, as above). (Note that $\overline{u}, \overline{u}'$ are uniquely determined, by Lemma 5.1.) Thus, there is no $\rho \supseteq \pi$ with $\hat{\tau}|\hat{\tau}' \models_\rho \mathbb{A}rds(\eta)\backslash\delta^\oplus$. But from R-safe condition of the premise, we know that $rlocs(\tau, \eta) \subseteq freshL(\sigma, \tau) \cup rlocs(\sigma, \varepsilon)$ and $rlocs(\tau', \eta) \subseteq freshL(\sigma', \tau') \cup rlocs(\sigma', \varepsilon)$. So (61) implies $Agree(\tau, \tau', \rho, \eta\backslash(\delta, \text{rd alloc}))$ and $Agree(\tau', \tau, \rho^{-1}, \eta\backslash(\delta, \text{rd alloc}))$, which is a contradiction.

- In case BIFX, B has the form (if E then D_0 else D_1); D_2 for some D_0, D_1, D_2.

  To show that BIFX does not apply, we show that $\tau(E) \neq \tau'(E)$ cannot happen, by contradiction. Suppose $\tau(E) = \text{true}$ and $\tau'(E) = \text{false}$ (a symmetric argument handles the case $\tau(E) = \text{false}$ and $\tau'(E) = \text{true}$). By unary semantics, we have $\langle \text{if } E \text{ then } D_0 \text{ else } D_1; D_2,\ \tau,\ \mu\rangle \overset{\varphi_0}{\longmapsto} \langle D_0; D_2,\ \tau,\ \mu\rangle$ and $\langle \text{if } E \text{ then } D_0 \text{ else } D_1; D_2,\ \tau',\ \mu\rangle \overset{\varphi_1}{\longmapsto} \langle D_1; D_2,\ \tau',\ \mu\rangle$. The latter step can also be taken via φ_0 as it is not a call. By Equation (61), we have

$$Lagree(\tau, \tau', \rho, (freshL(\sigma, \tau) \cup rlocs(\sigma, \varepsilon^{\leftarrow}_\delta))\backslash rlocs(\tau, \delta^\oplus)).$$

  The r-respects condition for the left step is for the collective boundary $(+N \in (\Phi, \mu), N \neq topm(B, M).\ bnd(N))$, but because C is let-free, μ is empty and $topm(B, M)$ is M, so this simplifies to δ. So, we have the agreement in the antecedent for r-respects, and the other antecedent is $Agree(\tau', \tau', \delta)$, which holds. So by r-respect from the premise, and instantiating the alternate step as the one from τ', we can obtain $D_0; D_2 \equiv D_1; D_2$. This is false, because we assume all subcommands are uniquely labeled and thus the label on D_0 is distinct from the one on D_1. (See footnote 19 in Definition 3.3.)
- For BWHX, B has the form while E do $D_0; D_1$ so $\|B\|$ is while $E|E \cdot \text{false}|\text{false}$ do $D_0; \|D_1\|$. As the alignment guards are false, rule BWHX applies just if $\tau(E) \neq \tau'(E)$. We can show this contradicts the premise for the same reasons as in the argument above for BIFX in the case $D_0 \not\equiv D_1$, i.e., the conditional branches differ. We do not have to consider the situation where the branches go different ways but the code is the same: if $\tau(E) = \text{true}$ and $\tau'(E) = \text{false}$ then $\langle \text{while } E \text{ do } D_0; D_1,\ \tau,\ \mu\rangle \overset{\varphi_0}{\longmapsto} \langle D_0; \text{while } E \text{ do } D_0; D_1,\ \tau,\ \mu\rangle$ and $\langle \text{while } E \text{ do } D_0; D_1,\ \tau',\ \mu\rangle \overset{\varphi_1}{\longmapsto} \langle D_1,\ \tau',\ \mu\rangle$—the code is different, as needed to contradict r-respects in the premise.

Post. Consider terminated trace $\langle CC, \sigma|\sigma', \_|\_\rangle \overset{\varphi}{\Longrightarrow}{}^* \langle \lfloor \text{skip} \rfloor, \tau|\tau', \_|\_\rangle$, for states τ, τ'. We must prove $\hat{\tau}, \hat{\tau}' \models_\pi \Diamond(\mathbb{B}Q \wedge \mathbb{A}\varepsilon^{\rightarrow}_\delta)$, where $\varepsilon^{\rightarrow}_\delta \triangleq (\text{rd}\,(\text{alloc}\backslash r)\text{`any}, Asnap(\varepsilon))\backslash\delta$ with $\hat{\tau} = [\tau{+}\overline{s}{:}\,\overline{v}]$ and $\hat{\tau}' = [\tau'{+}\overline{s}{:}\,\overline{v}']$ (with $\overline{v}, \overline{v}'$ as defined following Equation (60)).

Recall that we have $\hat{\sigma}|\hat{\sigma}' \models_\pi \mathbb{B}P \wedge \mathbb{A}\varepsilon^{\leftarrow}_\delta \wedge \mathbb{B}(s_{\text{alloc}} = \text{alloc} \wedge snap(\varepsilon))$, where $\varepsilon^{\leftarrow}_\delta \triangleq rds(\varepsilon)\backslash\delta^\oplus$ (see Equation (60)). From Equation (61), we get allowed dependencies

$$\sigma, \sigma' \overset{\pi}{\Rightarrow} \tau, \tau' \models^{\sigma}_{\delta} \varepsilon \text{ and } \sigma', \sigma \overset{\pi^{-1}}{\Rightarrow} \tau', \tau \models^{\sigma'}_{\delta} \varepsilon. \tag{62}$$

Also, from Lemma 7.8 (projection lemma), we get two terminated traces of the premise. Thus, we have $\hat{\tau} \models Q$ and $\hat{\tau}' \models Q$. From $\hat{\sigma}|\hat{\sigma}' \models_\pi \mathbb{A}\varepsilon^{\leftarrow}_\delta$ and $\hat{\sigma}|\hat{\sigma}' \models_\pi \mathbb{B}P$ and side condition $P \models w2r(\varepsilon) \leq rds(\varepsilon)$ we get $\hat{\sigma}|\hat{\sigma}' \models_\pi \mathbb{A}w2r(\varepsilon)\backslash\delta^\oplus$. This means, by semantics of $\mathbb{A}$ and definitions (noting that spec-only variables are not among the agreeing locations) that

$$Lagree(\sigma, \sigma', \pi, wlocs(\sigma, \varepsilon)\backslash rlocs(\sigma, \delta^\oplus)),$$
$$Lagree(\sigma', \sigma, \pi^{-1}, wlocs(\sigma', \varepsilon)\backslash rlocs(\sigma', \delta^\oplus)).$$

Now using Equation (62), by preservation Lemma A.4, we get

$$Lagree(\tau, \tau', \rho, wlocs(\sigma, \varepsilon)\backslash rlocs(\sigma, \delta^\oplus)\backslash rlocs(\tau, \delta^\oplus)),$$
$$Lagree(\tau', \tau, \rho^{-1}, wlocs(\sigma', \varepsilon)\backslash rlocs(\sigma', \delta^\oplus)\backslash rlocs(\tau', \delta^\oplus)).$$

From Encap boundary monotonicity condition of the premise we get $rlocs(\sigma,\delta) \subseteq rlocs(\tau,\delta)$ and $rlocs(\sigma',\delta) \subseteq rlocs(\tau',\delta)$. Thus, the preceding agreements simplify to

$$Lagree(\tau,\tau',\rho, wlocs(\sigma,\varepsilon)\backslash rlocs(\tau,\delta^{\oplus})),$$
$$Lagree(\tau',\tau,\rho^{-1}, wlocs(\sigma',\varepsilon)\backslash rlocs(\tau',\delta^{\oplus})).$$

Furthermore, by Lemma 8.3, we have $wlocs(\sigma,\varepsilon)\backslash rlocs(\tau,\delta^{\oplus}) = rlocs(\tau, Asnap(\varepsilon)\backslash\delta)$ and also $wlocs(\sigma',\varepsilon)\backslash rlocs(\tau',\delta^{\oplus}) = rlocs(\tau', Asnap(\varepsilon)\backslash\delta)$. Thus, we get

$$Lagree(\tau,\tau',\rho, rlocs(\tau, Asnap(\varepsilon)\backslash\delta)),$$
$$Lagree(\tau',\tau,\rho^{-1}, rlocs(\tau', Asnap(\varepsilon)\backslash\delta)).$$

This means $\hat{\tau}|\hat{\tau}' \models_{\rho} \mathbb{A} Asnap(\varepsilon)\backslash\delta$.

Since $freshL(\tau,v) = rlocs(v, \mathrm{rd}\,(\mathrm{alloc}\backslash r)\text{`any})$ and $freshL(\tau',v') = rlocs(v', \mathrm{rd}\,(\mathrm{alloc}\backslash r)\text{`any})$, we can use the agreements on fresh locations given by Equation (62) to get $\hat{\tau}|\hat{\tau}' \models_{\rho} \mathbb{A}(\mathrm{rd}\,(\mathrm{alloc}\backslash r)\text{`any})\backslash\delta$.

Combining what is proved above and using ρ as witness of the existential in the semantics of $\diamond$, we conclude the proof of Post: $\hat{\tau}|\hat{\tau}' \models_{\pi} \diamond(\mathbb{B}Q \wedge \mathbb{A}(\mathrm{rd}\,(\mathrm{alloc}\backslash r)\text{`any}, Asnap(\varepsilon)\backslash\delta))$.

R-safe. By projection Lemma 7.8(c) there are unary executions that take the same unary steps. The R-safe condition from the premise applies on both sides and yields R-safety for the conclusion.

D.4 Soundness of RSOF

$$\textsc{rSOF}\ \frac{\begin{array}{c} LocEq_{\delta}(\Phi,\Theta) \vdash_M [\![C]\!] : locEq_{\delta}(P \rightsquigarrow Q\ [\varepsilon]) \\ \models bnd(N)|bnd(N)\ \mathrm{frm}\ \mathcal{N} \qquad \mathcal{N} \Rrightarrow \Box\mathcal{N} \qquad N \neq M \\ N \in \Theta \qquad \forall m \in \Phi.\ mdl(m) \not\leq N \qquad \delta = (+L \in (\Phi,\Theta), L \neq M.\ bnd(L)) \qquad C \text{ is let-free} \end{array}}{LocEq_{\delta}(\Phi),\ LocEq_{\delta}(\Theta) \mathbin{\triangle} \mathcal{N} \vdash_M [\![C]\!] \ :\ locEq_{\delta}(P \rightsquigarrow Q\ [\varepsilon]) \mathbin{\triangle} \mathcal{N}}$$

Before studying the following, readers are advised to be familiar with Sections D.2 and D.3.

To show soundness of RSOF, suppose the side conditions hold and the premise of the rule is valid:

$$LocEq_{\delta}(\Phi,\Theta) \models_M [\![C]\!] : locEq_{\delta}(P \rightsquigarrow Q\ [\varepsilon]). \tag{63}$$

We must prove validity of the conclusion:

$$LocEq_{\delta}(\Phi), (LocEq_{\delta}(\Theta) \mathbin{\triangle} \mathcal{N}) \models_M [\![C]\!] \ :\ locEq_{\delta}(P \rightsquigarrow Q\ [\varepsilon]) \mathbin{\triangle} \mathcal{N}. \tag{64}$$

To that end, consider an arbitrary model φ^{+} of the relational context $LocEq_{\delta}(\Phi), LocEq_{\delta}(\Theta) \mathbin{\triangle} \mathcal{N}$. To make use of the premise, we define a model, φ^{-}, of $LocEq_{\delta}(\Phi,\Theta)$.

For m in Φ, the definition is unchanged: $\varphi_i^{-}(m) = \varphi_i^{+}(m)$ for $i \in \{0,1,2\}$. For methods m of Θ, we first define $\varphi_2^{-}(m)$. For that, we need some notation. Suppose $\Theta(m) = R \rightsquigarrow S\ [\eta]$. Let $\mathcal{R}$ be the local equivalence precondition

$$\mathcal{R} \ \hat{=}\ \mathbb{B}R \wedge \mathbb{A} rds(\eta)\backslash\delta^{\oplus} \wedge \mathbb{B}(s^m_{\mathrm{alloc}} = \mathrm{alloc} \wedge snap^m(\eta)). \tag{65}$$

Let $\bar{t}$ be the spec-only variables, including s^m_{alloc} and the $snap^m$ ones. Note that $\mathcal{N}$ depends on no spec-only variables, by the side condition that it is framed by dynamic boundary $bnd(N)$. For any states τ and τ', define

$$\varphi_2^{-}(m)(\tau|\tau') \hat{=} \begin{cases} \{\lightning\} & \forall\pi,\overline{u},\overline{u}'.\ \tau|\tau' \models_{\pi} \neg\mathcal{R}^{\bar{t}|\bar{t}}_{\overline{u}|\overline{u}'}, \\ \varnothing & (\exists\pi,\overline{u},\overline{u}'.\ \tau|\tau' \models_{\pi} \mathcal{R}^{\bar{t}|\bar{t}}_{\overline{u}|\overline{u}'},) \wedge (\forall\pi,\overline{u},\overline{u}'.\ \tau|\tau' \models_{\pi} \mathcal{R}^{\bar{t}|\bar{t}}_{\overline{u}|\overline{u}'}, \Rightarrow \tau|\tau' \not\models_{\pi} \mathcal{N}), \\ \varphi_2^{+}(m)(\tau|\tau') & \exists\pi,\overline{u},\overline{u}'.\ \tau|\tau' \models_{\pi} \mathcal{R}^{\bar{t}|\bar{t}}_{\overline{u}|\overline{u}'}, \wedge\mathcal{N}. \end{cases}$$

One might hope that $(\varphi_0^+, \varphi_1^+, \varphi_2^-)$ is a model for $LocEq_\delta(\Phi, \Theta)$ but this may fail for m in Φ if $\varphi_0^+(m)(\tau)$ or $\varphi_1^+(m)(\tau')$ is non-empty for $\tau|\tau'$ that satisfy $\mathcal{R}$ but not $\mathcal{N}$—because then the relational compatibility condition for pre-model fails (Definition 7.4, which is a pre-requisite for Definition 7.9).

To solve this problem, we define $\varphi_0^-(m)$ and $\varphi_1^-(m)$ like $\varphi_0^+(m)$ and $\varphi_1^+(m)$ but yielding empty outcome sets for such τ, τ'. To see why this works, we make the following observations about the definitions of pre-model and model for unary specs. For any pre-model $\varphi(m)$ and states τ, σ, if $\tau \in \varphi(m)(\sigma)$ and $\varphi'(m)$ is defined identically to $\varphi(m)$ except that $\varphi'(m)(\sigma) = (\varphi(m)(\sigma))\backslash\{\tau\}$, then φ' is a pre-model. Moreover, if $\varphi(m)$ is a context model for some spec and σ satisfies the precondition, then φ' is a context model. Now, for any τ, define $\varphi_0^-(m)(\tau) \triangleq \varnothing$ if there is τ' such that the conditions of the second case for φ_2^- hold for $\tau|\tau'$, that is

$$\left(\exists \pi, \overline{u}, \overline{u}'.\ \tau|\tau' \models_\pi \mathcal{R}^{\overline{t}|\overline{t}}_{\overline{u}|\overline{u}'},\right) \text{ and } \left(\forall \pi, \overline{u}, \overline{u}'.\ \tau|\tau' \models_\pi \mathcal{R}^{\overline{t}|\overline{t}}_{\overline{u}|\overline{u}'}, \Rightarrow \tau|\tau' \not\models_\pi \mathcal{N}\right).$$

Otherwise, define $\varphi_0^-(m)(\tau) \triangleq \varphi_0(m)(\tau)$. The displayed condition implies that τ satisfies the unary precondition R, so $\varphi_0^-(m)$ is a model for $\Theta(m)$ as observed above. Define $\varphi_1^-(m)$ the same way but existentially quantifying the left state: $\varphi_1^-(m)(\tau) \triangleq \varnothing$ if there is τ such that $(\exists \pi, \overline{u}, \overline{u}'.\ \tau|\tau' \models_\pi \mathcal{R}^{\overline{t}|\overline{t}}_{\overline{u}|\overline{u}'},)$ and $(\forall \pi, \overline{u}, \overline{u}'.\ \tau|\tau' \models_\pi \mathcal{R}^{\overline{t}|\overline{t}}_{\overline{u}|\overline{u}'}, \Rightarrow \tau|\tau' \not\models_\pi \mathcal{N})$; otherwise define $\varphi_1^-(m)(\tau) \triangleq \varphi_1(m)(\tau)$. We leave it to the reader to check that $(\varphi_0^-, \varphi_1^-, \varphi_2^-)$ satisfies all the conditions to be a relational pre-model and to be a context model of $LocEq_\delta(\Phi, \Theta)$. The latter means φ_0^- and φ_1^- are (Φ, Θ)-models, and $\varphi_2^-(m)$ models $locEq_\delta(\Phi, \Theta)(m)$ for all m.

Now, we return to the proof of validity of the conclusion, (64). Having fixed an arbitrary context model φ^+, we now consider any σ, σ', π that satisfy the precondition of the conclusion, i.e., the precondition of $locEq_\delta(P \rightsquigarrow Q\ [\varepsilon]) \mathbin{\mathbb{O}} \mathcal{N}$. That is, we assume

$$\hat{\sigma}|\hat{\sigma}' \models_\pi \mathbb{B}P \wedge \mathbb{A}rds(\varepsilon)\backslash\delta^\oplus \wedge \mathbb{B}(s_{\text{alloc}} = \text{alloc} \wedge snap(\varepsilon)) \wedge \mathcal{N}, \tag{66}$$

where $\overline{s}$ are the spec-only variables (which are the same on both sides of these specs), $\hat{\sigma} = [\sigma{+}\overline{s}{:}\,\overline{v}]$, $\hat{\sigma}' = [\sigma'{+}\overline{s}{:}\,\overline{v}']$ for some $\overline{v}, \overline{v}'$. (Recall that $\overline{v}, \overline{v}'$ are uniquely determined, by Lemma C.1.)

To finish the soundness proof, we need the following claim involving σ, σ', π and the context model φ^- derived from φ^+.

> Claim. *If* $\langle [\![C]\!],\ \sigma|\sigma',\ \_|\_\rangle \overset{\varphi^+}{\Longmapsto}{}^* \langle BB,\ \tau|\tau',\ \mu|\mu'\rangle$, *then there are* B *and* ρ *such that*
> *(a)* $\langle [\![C]\!],\ \sigma|\sigma',\ \_|\_\rangle \overset{\varphi^-}{\Longmapsto}{}^* \langle BB,\ \tau|\tau',\ \mu|\mu'\rangle$,
> *(b)* $\tau|\tau' \models_\rho \mathcal{N}$,
> *(c)* $\rho \supseteq \pi$ *and* $BB \equiv [\![B]\!]$ *and* $\mu = \mu'$,
> *(d)* $Lagree(\tau, \tau', \rho, (freshL(\sigma, \tau) \cup rlocs(\sigma, \varepsilon) \cup wrttn(\sigma, \tau))\backslash rlocs(\tau, \delta^\oplus))$, *and*
> *(e)* $Lagree(\tau', \tau, \rho^{-1}, (freshL(\sigma', \tau') \cup rlocs(\sigma', \varepsilon) \cup wrttn(\sigma', \tau'))\backslash rlocs(\tau', \delta^\oplus))$.

Item (a) says a trace via the conclusion's φ^+ can be taken via the premise's φ^-. Item (b) says $\mathcal{N}$ holds at every step (outside context calls). Items (c), (d), and (e) are the same as the conclusions (v), (vi), and (vii) of the lockstep alignment Lemma 8.9, for refperm ρ that additionally truthifies $\mathcal{N}$ according to item (b).

We do not directly apply Lemma 8.9 in the following argument, because it gives us no good way to establish $\tau|\tau' \models_\rho \mathcal{N}$. However, we will establish (c)–(e) by similar arguments to the proof (Section D.2) of Lemma 8.9, in which the conclusions (v)–(vii) are proved by induction on a given trace. In short, we will apply the induction step of that proof. Whereas the lemma connects an initial π with a refperm $\rho \supseteq \pi$ for a given reachable configuration, the proof of the induction step of the lemma does exactly what we need: Given a current ρ with $\rho \supseteq \pi$, it yields a $\dot{\rho}$ with $\dot{\rho} \supseteq \rho$, for the next step of the trace. We can reason the same way, for (c)–(e), but also add that $\dot{\rho}$ satisfies $\mathcal{N}$.

One could factor out the induction step of the lemma as a separate result, and then apply it directly here. We refrain from spelling that out explicitly, but we do need to be clear how we are instantiating the assumptions of Lemma 8.9. For the unary spec Ψ in the Lemma, we take (Φ, Θ). For the relational spec Φ in the Lemma, we take $(LocEq_\delta(\Phi), LocEq_\delta(\Theta))$, which is the same as $LocEq_\delta(\Phi, \Theta)$. For the context model φ, we take φ^-. So, we have assumption (i) of the Lemma. We also have (ii), as direct consequence of Equation (66). For (iii), we will consider a trace via φ^- given by (a) in the Claim. For (iv), i.e., r-safety and respect for that trace, we will appeal to the premise (63).

Proof of Claim, by induction on steps.

Base Case. For initial configuration $\langle [\![C]\!],\ \sigma|\sigma',\ \_|\_\rangle$, take $\rho := \pi$. We have $\sigma|\sigma' \models_\pi \mathcal{N}$ by assumption (66); the rest follows.

Induction Case. Suppose

$$\langle [\![C]\!],\ \sigma|\sigma',\ \_|\_\rangle \overset{\varphi^+}{\Longmapsto}{}^* \langle BB,\ \tau|\tau',\ \mu|\mu'\rangle \overset{\varphi^+}{\Longmapsto} \langle DD,\ v|v',\ \nu|\nu'\rangle. \tag{67}$$

By induction hypothesis there is ρ such that the conditions (a)–(e) of the Claim hold for the configuration with τ, τ'—including $\rho \supseteq \pi$, $\tau|\tau' \models_\rho \mathcal{N}$, BB has the form $[\![B]\!]$ for some B, and $\langle [\![C]\!],\ \sigma|\sigma',\ \_|\_\rangle \overset{\varphi^-}{\Longmapsto}{}^* \langle BB,\ \tau|\tau',\ \mu|\mu'\rangle$. We must show there is $\dot\rho$ such that $\dot\rho \supseteq \pi$, $v|v' \models_{\dot\rho} \mathcal{N}$, $\langle [\![B]\!],\ \tau|\tau',\ \mu|\mu'\rangle \overset{\varphi^-}{\Longmapsto} \langle DD,\ v|v',\ \nu|\nu'\rangle$, and the other conditions of the Claim for $\dot\rho, v, v'$. We write (ȧ), (ḃ), and so on, to indicate those conditions instantiated for $\dot\rho, v, v'$.

To find $\dot\rho$ and show the conditions of the Claim for v, v' we distinguish three cases:

Case *Active*(B) is not a context call. Because the step is not a call, it is independent of model, so we have

$$\langle [\![B]\!],\ \tau|\tau',\ \mu|\mu'\rangle \overset{\varphi^-}{\Longmapsto} \langle DD,\ v|v',\ \nu|\nu'\rangle, \tag{68}$$

which takes care of part (ȧ) of the Claim. Moreover, this together with Equation (66) lets us instantiate the premise Equation (63), so (by Encap) we have that the left and right projections of the whole trace Equation (67) satisfy respect for $((\Phi, \Theta), M, \varphi_0^-, \varepsilon, \sigma)$ and $((\Phi, \Theta), M, \varphi_1^-, \varepsilon, \sigma')$, respectively. Thus, we have the assumption (iv) of Lemma 8.9 applied to the trace Equation (67). By direct application of the Lemma, we get that $\nu = \nu'$ and there is some D with $DD \equiv [\![D]\!]$. Direct application would also yield agreements for some $\dot\rho \supseteq \pi$, but that is not enough. Instead, we apply the induction step of the Lemma's proof, which yields $\dot\rho$ such that $\dot\rho \supseteq \rho$ and (ḋ) and (ė) hold. Finally, from the Encap condition of premise of the rule, we also know that unary steps on left and right of Equation (68) w-respect $bnd(N)$, so we get $Agree(\tau, v, bnd(N))$ and $Agree(\tau', v', bnd(N))$. So from side condition $\models bnd(N)|bnd(N)$ frm $\mathcal{N}$, by Definition 7.1 of the relational framing judgment, using (b), we get $v|v' \models_\rho \mathcal{N}$. By $\dot\rho \supseteq \rho$ and the side condition $\mathcal{N} \Rightarrow \Box\mathcal{N}$ of rSOF, we get $v|v' \models_{\dot\rho} \mathcal{N}$, proving (ḃ) and concluding the induction step for this case.

Note that the induction step in the proof of Lemma 8.9 goes by cases on transition rules. The preceding paragraph covered all the transition rules except for context call.

Case *Active*(B) is a context call to some m in Φ. The step can be taken via φ^-, because $\varphi_2^-(m)$ is defined to be $\varphi_2^+(m)$, so we have (ȧ). As in the preceding case, we can apply the induction step of Lemma 8.9 to get $\dot\rho \supseteq \rho$ with (ċ)–(ė). As in the preceding case, we appeal to w-respect for premise (63), and $\models bnd(N)|bnd(N)$ frm $\mathcal{N}$, to get (ḃ).

In our appeal to the proof of Lemma 8.9, we are here using the cases of transition rules bCallS and bCall0.

Case *Active*(B) is a context call to some m in Θ. So B has the form $B \equiv m(); B_2$ for some B_2. The transition can go by either bCall0 or bCallS. In the case of bCall0, we get the Claim directly from the induction hypothesis: taking $\dot\rho := \rho$ we get (ȧ)–(ė) from (a)–(e).

Now consider the case of bCallS. Suppose $\Theta(m) = R \rightsquigarrow S\ [\eta]$ and $\overline{t}$ is spec-only variables of R and of the snapshot variables of $locEq_\delta(R \rightsquigarrow S\ [\eta])$ tagged for m. Since we are in the case bCallS, the precondition of m for φ^+ holds, for some refperm; $\varphi^-(m)$ is defined the same way (last case in its definition) and the transition can be taken via φ^-, so we have (ȧ). It remains to find some $\dot{\rho} \supseteq \pi$ satisfying (ḃ)–(ė) for v, v'. For (ċ), by bCallS the method environments are unchanged and DD has the form $[\![B_2]\!]$.

Let us spell out what it means that the precondition of m for φ^+ (i.e., the precondition of $locEq_\delta(R \rightsquigarrow S\ [\eta])$) holds for some ρ_1: We have

$$\hat{\tau}|\hat{\tau}' \models_{\rho_1} (\mathbb{B}R \wedge \mathbb{A}\eta_\delta^{\leftarrow} \wedge \mathbb{B}(s^m_{\text{alloc}} = \text{alloc} \wedge snap^m(\eta)))^{\overline{t}|\overline{t}}_{\overline{u}|\overline{u}'}, \wedge \mathcal{N}, \tag{69}$$

where $\hat{\tau} \triangleq [\tau{+}\overline{s}{:}\overline{v}]$ and $\hat{\tau}' \triangleq [\tau{+}\overline{s}{:}\overline{v}']$, where $\overline{v}, \overline{v}'$ are the unique values for the spec-only variables $\overline{s}$ defined in connection with Equation (66), and $\overline{u}, \overline{u}'$ are the unique values for the spec-only variables $\overline{t}$ for $\Theta(m)$. We can write $\mathcal{N}$ outside the substitutions, because it has no spec-only variables, but this is not important. What is important is that $\overline{v}, \overline{v}', \overline{u}, \overline{u}'$ are uniquely determined, independent of the refperm, by Lemma C.1. Let $\widehat{\tau} \triangleq [\hat{\tau}{+}\overline{t}{:}\overline{u}]$ and $\widehat{\tau}' \triangleq [\hat{\tau}{+}\overline{t}{:}\overline{u}']$. So Equation (69) can be written

$$\widehat{\tau}|\widehat{\tau}' \models_{\rho_1} \mathbb{B}R \wedge \mathbb{A}\eta_\delta^{\leftarrow} \wedge \mathbb{B}(s^m_{\text{alloc}} = \text{alloc} \wedge snap^m(\eta)) \wedge \mathcal{N}. \tag{70}$$

Now, $\mathbb{B}R \wedge \mathbb{B}(s^m_{\text{alloc}} = \text{alloc} \wedge snap^m(\eta))$ is refperm independent. So using induction hypothesis (b), we have $\widehat{\tau}|\widehat{\tau}' \models_\rho \mathbb{B}R \wedge \mathbb{B}(s^m_{\text{alloc}} = \text{alloc} \wedge snap^m(\eta)) \wedge \mathcal{N}$. We can get $\widehat{\tau}|\widehat{\tau}' \models_\rho \mathbb{A}\eta_\delta^{\leftarrow}$ from induction hypothesis (d) and (e), as follows. First, we have Encap and r-safety for the trace up to τ, τ', by induction hypothesis (a) and the premise. Now $\eta_\delta^{\leftarrow}$ is $rds(\eta)\backslash\delta^{\oplus}$, i.e., $rds(\eta)\backslash(\delta, \text{rd alloc})$. By r-safety, we have $rlocs(\tau, \eta_\delta^{\leftarrow}) \subseteq (freshL(\sigma, \tau) \cup rlocs(\sigma, \varepsilon))\backslash rlocs(\tau, \delta^{\oplus}))$ and $rlocs(\tau', \eta_\delta^{\leftarrow}) \subseteq (freshL(\sigma', \tau') \cup rlocs(\sigma', \varepsilon))\backslash rlocs(\tau', \delta^{\oplus}))$. So by semantics of $\mathbb{A}\eta_\delta^{\leftarrow}$ and induction hypothesis (d) and (e) we get $\widehat{\tau}|\widehat{\tau}' \models_\rho \mathbb{A}\eta_\delta^{\leftarrow}$.

Having established that the precondition (70) holds for $\rho_1 := \rho$, we can instantiate the spec of m with ρ and obtain the postcondition (in accord with Definition 7.9 of relational context model):

$$\widehat{v}|\widehat{v}' \models_\rho \Diamond(\mathbb{B}S \wedge \mathbb{A}\eta_\delta^{\rightarrow} \wedge \mathcal{N}).$$

By semantics, this implies there is $\dot{\rho} \supseteq \rho$ with $v|v' \models_{\dot{\rho}} \mathbb{B}S \wedge \mathbb{A}\eta_\delta^{\rightarrow} \wedge \mathcal{N}$. So, we have (ḃ) and (ċ). Finally, $\dot{\rho}$ satisfies the agreements of (ḋ) and (ė); this follows from $v|v' \models_{\dot{\rho}} \mathbb{A}\eta_\delta^{\rightarrow}$ for reasons that are spelled out in detail in proving the induction step of Lemma 8.9 in the case of bCallS, starting around the displayed formula (57).

Having proved the Claim, we prove validity of the conclusion (64) of rSOF.

Safety. Suppose $\langle [\![C]\!],\ \sigma|\sigma',\ \_|\_\rangle \overset{\varphi^+}{\Longrightarrow}{}^* \langle BB,\ \tau|\tau',\ \mu|\mu'\rangle$. We show by contradiction the latter configuration cannot fault.

Case: fault by a non-call step. Then the faulting step can also be taken via φ^-, and it is reached via φ^- owing to the Claim (a), but a faulting trace via φ^- contradicts the premise (63).

Case: fault by a context call to some m in Φ. Then the step can also be taken via φ^-, again contradicting the premise.

Case: fault by a context call to some m in Θ. Let the spec of m be $R \rightsquigarrow S\ [\eta]$, so the relational precondition is $\mathcal{R} \wedge \mathcal{N}$ where $\mathcal{R}$ is given by Equation (65). Because φ^+ is a context model, the call only faults if there are no $\dot{\rho}, \overline{u}, \overline{u}'$ such that $\tau|\tau' \models_{\dot{\rho}} \mathcal{R}^{\overline{t}|\overline{t}}_{\overline{u}|\overline{u}'}, \wedge \mathcal{N}$ (see transition rule bCallX). By the snapshot uniqueness Lemma C.1, values $\overline{u}, \overline{u}'$ exist and are uniquely determined by τ, τ'. So the call only faults if there is no $\dot{\rho}$ such that $\hat{\tau}|\hat{\tau}' \models_{\dot{\rho}} \mathcal{R} \wedge \mathcal{N}$ where $\hat{\tau}, \hat{\tau}'$ are the states extended with $\overline{u}, \overline{u}'$ for the snapshot variables. But, we have ρ and can show $\hat{\tau}|\hat{\tau}' \models_\rho \mathcal{R} \wedge \mathcal{N}$

as follows. We have $\hat{\tau}|\hat{\tau}' \models_\rho \mathcal{N}$ by Claim (b). We have $\hat{\tau}|\hat{\tau}' \models_\rho \mathbb{B}(s^m_{\text{alloc}} = \text{alloc} \wedge \mathit{snap}^m(\eta))$ in accord with our choice of the correct snapshot values. To show the conjunct $\hat{\tau}|\hat{\tau}' \models_\rho \mathbb{B}R$, we can apply the premise, in particular Safety: there must be some refperm for which $\hat{\tau}|\hat{\tau}'$ satisfy $\mathbb{B}R$, because otherwise the call would fault via φ^-, contrary to the premise Equation (63). Now, we get $\hat{\tau}|\hat{\tau}' \models_\rho \mathbb{B}R$, because $\mathbb{B}R$ is refperm independent. It remains to show the conjunct $\hat{\tau}|\hat{\tau}' \models_\rho \mathbb{A}\eta^{\leftarrow}_\delta$, that is, $\hat{\tau}|\hat{\tau}' \models_\rho \mathbb{A}\mathit{rds}(\eta)\backslash\delta^{\oplus}$. We have r-safety for the trace up to τ, τ', by Claim (a) and the premise. By r-safety, we have $\mathit{rlocs}(\tau, \eta^{\leftarrow}_\delta) \subseteq (\mathit{freshL}(\sigma, \tau) \cup \mathit{rlocs}(\sigma, \varepsilon))\backslash\mathit{rlocs}(\tau, \delta^{\oplus}))$ and $\mathit{rlocs}(\tau', \eta^{\leftarrow}_\delta) \subseteq (\mathit{freshL}(\sigma', \tau') \cup \mathit{rlocs}(\sigma', \varepsilon))\backslash\mathit{rlocs}(\tau', \delta^{\oplus}))$. So by Claim (d) and (e) we get $\hat{\tau}|\hat{\tau}' \models_\rho \mathbb{A}\eta^{\leftarrow}_\delta$.

Post. For all τ, τ' such that $\langle [\![C]\!],\ \sigma|\sigma',\ \_|\_\rangle \overset{\varphi^+}{\Longrightarrow}{}^* \langle \lfloor \text{skip} \rfloor,\ \tau|\tau',\ \_|\_\rangle$, we must show $\tau|\tau' \models_\pi \Diamond(\mathbb{B}Q \wedge \mathbb{A}\varepsilon^{\rightarrow}_\delta \wedge \mathcal{N})$. Applying the Claim to this trace, we obtain ρ such that conditions (a)–(e) hold for τ, τ'. We will show $\tau|\tau' \models_\rho \mathbb{B}Q \wedge \mathbb{A}\varepsilon^{\rightarrow}_\delta \wedge \mathcal{N}$; our obligation then follows by semantics of $\Diamond$, using $\rho \supseteq \pi$ from (b).

We have $\tau|\tau' \models_\rho \mathcal{N}$ by (b). By (a), we can instantiate the premise Equation (63), which yields $\tau|\tau' \models_\pi \Diamond(\mathbb{B}Q \wedge \mathbb{A}\varepsilon^{\rightarrow}_\delta)$. This implies $\tau|\tau' \models_\rho \mathbb{B}Q$, because $\mathbb{B}Q$ is refperm independent. Finally, we get $\tau|\tau' \models_\rho \mathbb{A}\varepsilon^{\rightarrow}_\delta$ as a consequence of (d) and (e) by essentially the same argument as the one spelled out in the proof of Post for rule rLocEq (Section D.3).

Write, R-safe, and Encap. These are obtained directly from the premise, using the Claim. Note that $\Phi, \Theta \between \mathcal{N}$ has the same methods, and thus the same modules, as Φ, Θ has, so the Encap conditions have exactly the same meaning for the conclusion of the rule as for the premise.

D.5 Soundness of rPoss, rDisj, and rConj

For rPoss, assume validity of the premise: $\Phi \models_M CC : \mathcal{P} \twoheadrightarrow \mathcal{Q}\ [\varepsilon|\varepsilon']$. To prove validity of the conclusion $\Phi \models_M CC : \Diamond\mathcal{P} \twoheadrightarrow \Diamond\mathcal{Q}\ [\varepsilon|\varepsilon']$, consider any Φ-model φ. Consider any σ, σ', π such that $\sigma|\sigma' \models_\pi \Diamond\mathcal{P}$. By formula semantics, there is $\rho \supseteq \pi$ such that $\sigma|\sigma' \models_\rho \mathcal{P}$. The Safety, Write, and Encap conditions now follow by instantiating the premise with φ and ρ. For Post, the premise yields that for terminal state pair $\tau|\tau'$, we have $\tau|\tau' \models_\rho \mathcal{Q}$. This implies $\tau|\tau' \models_\pi \Diamond\mathcal{Q}$, since $\rho \supseteq \pi$.

For rDisj, suppose φ is a Φ-model and suppose $\sigma|\sigma' \models_\pi \mathcal{P}_0 \vee \mathcal{P}_1$. By semantics of formulas, either $\sigma|\sigma' \models_\pi \mathcal{P}_0$ or $\sigma|\sigma' \models_\pi \mathcal{P}_1$, so we can instantiate one of the premises using φ. It is straightforward to check that the conditions of Definition 7.10 for the conclusion follow directly from the premise. Note that the propositional connectives have classical semantics in relational formulas, as they do in unary formulas.

For rConj the argument is similar.

D.6 Soundness of rFrame

All conditions except Post are easy consequences of the premise. For Post, suppose $\sigma|\sigma' \models_\pi \mathcal{P} \wedge \mathcal{R}$ and $\langle CC,\ \sigma|\sigma',\ \_|\_\rangle \overset{\varphi}{\Longrightarrow}{}^* \langle \lfloor \text{skip} \rfloor,\ \tau|\tau',\ \_|\_\rangle$. By Write, we have $\sigma{\rightarrow}\tau \models \varepsilon$ and $\sigma'{\rightarrow}\tau' \models \varepsilon'$ (as well as $\sigma \hookrightarrow \tau$ and $\sigma' \hookrightarrow \tau'$ of course). By the rule's condition $\mathcal{P} \wedge \mathcal{R} \Rightarrow \{\!|\eta \cdot/. \varepsilon|\!\} \wedge \{\!|\eta' \cdot/. \varepsilon'|\!\}$, we can use fact (29) to get $\mathit{Agree}(\sigma, \tau, \eta)$ and $\mathit{Agree}(\sigma', \tau', \eta')$. So by $\mathcal{P} \models \eta|\eta'$ frm $\mathcal{R}$ and semantics of this judgment we get $\tau|\tau' \models_\pi \mathcal{R}$. We have $\tau|\tau' \models_\pi \mathcal{Q}$ by Post for the premise.

D.7 Soundness of rEmb and rEmbS

We prove rEmb (Figure 30). The argument for rEmbS (Figure 38) is similar.

Suppose $\Phi_0 \models_M C : P \rightsquigarrow Q\,[\varepsilon]$ and $\Phi_1 \models_M C' : P' \rightsquigarrow Q'\,[\varepsilon']$. To show validity of the conclusion, $\Phi \models_M (C|C') : \{\!|P|\!\} \wedge \{\!|P'|\!\} \twoheadrightarrow \{\!|Q|\!\} \wedge \{\!|Q'|\!\}\ [\varepsilon|\varepsilon']$, consider any Φ-model φ and any σ, σ', π such

that $\sigma|\sigma' \models_\pi \{\!|P^{\bar{s}}_{\bar{v}}|\!\} \wedge [\!|P'^{\bar{s}'}_{\bar{v}'}|\!]$. By biprogram semantics, $(C|C')$ goes by dovetailed steps of C via φ_0 (rule bComL) and steps of C' via φ_1 (rules bComR and bComR0). All reached configurations are in the bi-com form. For Safety, observe that if fault is reached it is by bComLX or bComRX, so by projection, we obtain a faulting trace either of C or of C', contrary to the premises. For Post and Write, suppose $\langle (C|C'),\ \sigma|\sigma',\ \_|\_\rangle \overset{\varphi}{\Longrightarrow}{}^* \langle \lfloor \mathsf{skip} \rfloor,\ \tau|\tau',\ \_|\_\rangle$. Then by projection, we obtain terminated traces (via φ_0 and φ_1, respectively) to which the premises apply. This yields $\sigma \rightarrow \tau \models \varepsilon$ and $\sigma' \rightarrow \tau' \models \varepsilon'$ (proving Write) and $\tau \models Q^{\bar{s}}_{\bar{v}}$ and $\tau' \models Q'^{\bar{s}'}_{\bar{v}'}$ so that $\tau|\tau' \models_\pi \{\!|Q^{\bar{s}}_{\bar{v}}|\!\} \wedge [\!|Q'^{\bar{s}'}_{\bar{v}'}|\!]$ (proving Post). For every trace from $\langle (C|C'), \sigma|\sigma', \_|\_\rangle$ consider its projections, which are unary traces from $\langle C,\ \sigma,\ \_\rangle$ via φ_0 and $\langle C',\ \sigma',\ \_\rangle$ via φ_1. Then both R-safe and Encap follow using R-safe and Encap for the unary traces to which the premises apply.

D.8 Soundness of rCall

$$\textsc{rCall}\ \frac{\Phi_0 \vdash m() : \Phi_0(m) \qquad \Phi_1 \vdash m() : \Phi_1(m)}{\Phi \vdash \lfloor m() \rfloor : \Phi_2(m)}$$

Let the current module be N in all three judgments.

Suppose $\Phi_2(m)$ is $m : \mathcal{P} \approx\!> Q\ [\varepsilon]$. Let φ be a Φ-model and suppose $\sigma, \sigma' \models_\pi \mathcal{P}$. Because φ is a Φ-model (Definition 7.9), $\varphi_2(m)(\sigma|\sigma')$ does not contain $\lightning$. Moreover, execution from $\langle \lfloor m() \rfloor,\ \sigma|\sigma',\ \_|\_\rangle$ either goes by bCallS to a terminated state, or by bCall0 repeating the configuration $\langle \lfloor m() \rfloor,\ \sigma|\sigma',\ \_|\_\rangle$ unboundedly. So Safety holds. We also get Post and Write by definition of context model. R-safety requires $rlocs(\sigma, \eta) \subseteq rlocs(\sigma, \eta)$ and $rlocs(\sigma', \eta') \subseteq rlocs(\sigma', \eta')$, which hold.

Encap is more interesting, as it is not a direct consequence of φ being a context model. Encap imposes conditions on the unary projections of every trace from $\langle \lfloor m() \rfloor,\ \sigma|\sigma',\ \_|\_\rangle$. By projection Lemma 7.8, or indeed by unary compatibility of the context model, the premises of rCall apply to these traces—and yield all the Encap conditions.

D.9 Soundness of rIf

$$\textsc{rIf}\ \frac{\begin{array}{c}\Phi \vdash_M CC :\ \mathcal{P} \wedge \{\!|E|\!\} \wedge [\!|E'|\!] \approx\!> Q\ [\varepsilon|\varepsilon'] \\ \Phi \vdash_M DD :\ \mathcal{P} \wedge \{\!|\neg E|\!\} \wedge [\!|\neg E'|\!] \approx\!> Q\ [\varepsilon|\varepsilon'] \qquad \mathcal{P} \Rightarrow E \doteqdot E' \\ \delta = (+N \in \Phi, N \neq M.\ bnd(N)) \qquad \delta \cdot\!/.\ r2w(ftpt(E)) \qquad \delta \cdot\!/.\ r2w(ftpt(E'))\end{array}}{\Phi \vdash_M \mathsf{if}\ E|E'\ \mathsf{then}\ CC\ \mathsf{else}\ DD :\ \mathcal{P} \approx\!> Q\ [\varepsilon, ftpt(E)|\varepsilon', ftpt(E')]}$$

As in the unary rule If, the separator $(+N \in \Phi, N \neq M.\ bnd(N)) \cdot\!/.\ r2w(ftpt(E))$ and its counterpart simplify to true or false. In virtue of condition $\mathcal{P} \Rightarrow E \doteqdot E'$, every biprogram trace from states satisfying $\mathcal{P}$ begins with a step going to CC via bIfT or a step going to DD via bIfF; it cannot fault via bIfX, which is for tests that disagree. Subsequent steps satisfy all the conditions Safety, Post, Write, R-safe, because these are the same as the conditions for the premises CC and DD. Encap for the conclusion is almost the same condition as for the premise, the only difference being that the frame condition $\varepsilon|\eta'$ for the premise is a subeffect of the one for the conclusion. So Encap for the conclusion follows from the premises by an argument like that for soundness of rule rConseq.

The first step clearly satisfies Safety, Post, Write, and R-safe. To show the first step satisfies Encap, boundary monotonicity and w-respect are immediate, because the step does not change the state. For r-respect, we need that alternate executions follow the same control path—and this is ensured by separator conditions, for reasons spelled out in detail in the proof of If.

D.10 Soundness of RLINK

$$\textsc{rLink}\ \frac{\begin{array}{c}\Phi,\Theta \vdash_\bullet [\![C]\!] : \mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon] \\ \Phi,\Theta \vdash_{mdl(m)} (B|B') : \Theta_2(m) \qquad \Phi_0,\Theta_0 \vdash_{mdl(m)} B : \Theta_0(m) \qquad \Phi_1,\Theta_1 \vdash_{mdl(m)} B' : \Theta_1(m) \\ \delta = (+L \in (\Phi,\Theta).\ bnd(L)) \qquad (\Phi,\Theta) \Rrightarrow LocEq_\delta(\dot\Phi,\dot\Theta) \qquad \mathcal{P} \Rightarrow pre(locEq_\delta(P \rightsquigarrow Q\ [\varepsilon])) \\ \forall N \in \Phi, L \in \Theta.\ N \not\prec L \qquad \forall N, L.\ N \in \Theta \wedge N \prec L \Rightarrow L \in (\Phi,\Theta) \qquad C \text{ is let-free}\end{array}}{\Phi \vdash_\bullet \text{let } m = (B|B') \text{ in } [\![C]\!] : \mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon]}$$

The rule caters for different specs on left and right, subject to the constraints of Definition 4.1. For RMLINK, we instantiate $\Theta_2(m)$ to something of the form $locEq(\ldots) \mathbin{⍉} \mathcal{M}$, for coupling relation $\mathcal{M}$, and the operation $⍉\mathcal{M}$ conjoins $\overleftarrow{\mathcal{M}}$ and $\overrightarrow{\mathcal{M}}$ to the unary specs. Some unary ingredients appear in the premises and side conditions but are not directly used in the conclusion: P, Q, and $\dot\Phi$ and $\dot\Theta$. These ensure that the specs are strengthenings of a local equivalence spec.

Remark 12. This version of the rule includes unary premises for B and B'. These are used only to obtain unary models (of $\Theta_0(m)$ and $\Theta_1(m)$), which are formally required to define a full context model of Θ (using Lemma C.11). As the proof shows, execution of $[\![C]\!]$ remains fully aligned (except during environment calls to m) and all calls are sync'd, so the unary models have no influence on the traces used in the proof. In future work, we expect to eliminate these unary premises by revisiting the definitions of compatibility for context models (Definition 7.4), and adjusting the well-formedness conditions for contexts (Definition 4.1) and definition of covariant implication (Definition 8.5) for a better fit with compatibility.

In the following proof of RLINK, we assume there are no recursive calls in B or B'. To allow recursion, one should use a fixpoint construction for the denotational semantics (as in proof of linking for impure methods in RLIII) and an extra induction on calling depth (as in the linking proofs in RLII and RLIII). This adds complication but does not shed light; and there are plenty other complications that do deserve to be spelled out carefully.

As in the unary semantics, we say a biprogram trace is ***m-truncated*** iff the last configuration does not contain ecall(m). In general, there may be unary environment calls and ecall(m) may occur inside a bi-com, as in (skip|B; ecall(m); C); DD.

Consider any Φ-model φ. Let $\theta_0(m)$ and $\theta_1(m)$ be the models of $\Theta_0(m)$ and $\Theta_1(m)$ from the denotations of B and B', by Lemma A.8, using the unary premises for B and B', and side conditions about imports. Let θ be the bi-model of m given by Lemma C.11(i) for the denotation of $(B|B')$ in φ, for which we use that each method's relational precondition implies its unary preconditions (which holds, because Φ is wf; see Definition 4.1). Owing to validity of $\Phi,\Theta \vdash_N (B|B') : \Theta_2(m)$, we have that (φ,θ) is a (Φ,Θ)-model by Lemma C.11(ii).

In the rest of the proof, no further use is made of the unary premises for B and B'.

To introduce identifiers for the relational spec of m, suppose $\Phi_2(m)$ is $\mathcal{R} \approx\!> \mathcal{S}\ [\eta|\eta']$. For clarity, we follow a convention also used the in proof of unary LINK: environments that contain m have dotted names like $\dot\mu$ and the corresponding environment without m has the same name without dot.

CLAIM. *Let σ,σ',π be such that $\hat\sigma|\hat\sigma' \models_\pi \mathcal{P}$, where $\hat\sigma$ is $[\sigma{+}\overline{s}{:}\,\overline{v}]$ and $\hat\sigma'$ is $[\sigma{+}\overline{s}'{:}\,\overline{v}']$ for the unique values $\overline{v},\overline{v}'$ determined by σ,σ' for the spec-only variables $\overline{s},\overline{s}'$ of $\mathcal{P}$. Suppose*

$$\langle [\![C]\!],\ \sigma|\sigma',\ [m{:}B]|[m{:}B'] \rangle \overset{\varphi}{\Longmapsto}{}^* \langle DD,\ \tau|\tau',\ \dot\mu|\dot\mu' \rangle$$

is m-truncated (for some $DD,\tau,\tau',\dot\mu,\dot\mu'$). Then $\langle [\![C]\!],\ \sigma|\sigma',\ \_|\_ \rangle \overset{\varphi\theta}{\Longmapsto}{}^* \langle DD,\ \tau|\tau',\ \mu|\mu' \rangle$, where $\mu = \dot\mu \restriction m$ and $\mu' = \dot\mu' \restriction m$, and $DD = [\![D]\!]$ for some D. Moreover, if $D \equiv m(); D_0$ for some D_0, then there is ρ such that $\tau|\tau' \models_\rho \mathcal{R}$.

Proof of Claim. By induction on the number of completed top-level calls of m. (Since we are not considering recursion, all calls are top level.) The steps taken in code of $\llbracket C \rrbracket$ can be taken via $\stackrel{\varphi\theta}{\Longrightarrow}$, because the two transition relations are identical except for calls to m. By induction hypothesis, any call is in sync'd form, and a completed call from $\lfloor m() \rfloor$ amounts to a terminated execution of $(B|B')$. Thus, a completed call gives rise to a single step via (φ, θ) with the same outcome, because $\theta_2(m)$ is defined to be the denotation of $(B|B')$, which is defined directly in terms of executions of $(B|B')$—provided that the precondition $\mathcal{R}$ of m holds. The premise for $\llbracket C \rrbracket$ is applicable to the trace via φ, θ, so the precondition $\mathcal{R}$ must hold—because otherwise that trace could fault, contrary to the premise for $\llbracket C \rrbracket$. It remains to show that at DD is $\llbracket D \rrbracket$ for some D. For this, we appeal to lockstep alignment Lemma 8.9. Let U and V be the unary projections of this trace. By validity of the premise for $\llbracket C \rrbracket$, we get that U (respectively, V) satisfies r-safe for $((\Phi_0, \Theta_0), \varepsilon, \sigma)$ (respectively, $((\Phi_1, \Theta_1), \varepsilon, \sigma')$) and respect for $((\Phi_0, \Theta_0), \bullet, (\varphi_0, \theta_0), \varepsilon, \sigma)$ (respectively, $((\Phi_1, \Theta_1), \bullet, (\varphi_1, \theta_1), \varepsilon, \sigma')$). By side condition of rLink, C is let-free. Thus, the assumptions are satisfied for the instantiation $\Phi \triangleq (\Phi, \Theta)$ of Lemma 8.9, which yields that DD is $\llbracket D \rrbracket$ for some D. The Claim is proved.

Post. Consider any $\varphi, \sigma, \sigma', \pi$ with $\hat{\sigma}|\hat{\sigma}' \models_\pi \mathcal{P}$ (where $\hat{\sigma}$ is $[\sigma{+}\overline{s}{:}\,\overline{v}]$ and $\hat{\sigma}'$ is $[\sigma{+}\overline{s}'{:}\,\overline{v}']$ for the unique values $\overline{v}, \overline{v}'$ determined by σ, σ' for the spec-only variables $\overline{s}, \overline{s}'$ of $\mathcal{P}$). A terminated trace of the linked program has the form

$$\begin{aligned}\langle \mathsf{let}\ m = (B|B')\ \mathsf{in}\ \llbracket C \rrbracket,\ \sigma|\sigma',\ \_|\_\rangle &\stackrel{\varphi}{\Longrightarrow} \langle \llbracket C \rrbracket; \lfloor \mathsf{elet}(m) \rfloor,\ \sigma|\sigma',\ [m{:}B]|[m{:}B']\rangle \\ &\stackrel{\varphi}{\Longrightarrow}{}^* \langle \lfloor \mathsf{elet}(m) \rfloor,\ \tau|\tau',\ [m{:}B]|[m{:}B']\rangle \\ &\stackrel{\varphi}{\Longrightarrow} \langle \lfloor \mathsf{skip} \rfloor,\ \tau|\tau',\ \_|\_\rangle.\end{aligned}$$

By semantics, we obtain $\langle \llbracket C \rrbracket,\ \sigma|\sigma',\ [m{:}B]|[m{:}B']\rangle \stackrel{\varphi}{\Longrightarrow}{}^* \langle \lfloor \mathsf{skip} \rfloor,\ \tau|\tau',\ [m{:}B]|[m{:}B']\rangle$. This is m-truncated. By the Claim, we have $\langle \llbracket C \rrbracket,\ \sigma|\sigma',\ \_|\_\rangle \stackrel{\varphi\theta}{\Longrightarrow}{}^* \langle \lfloor \mathsf{skip} \rfloor,\ \tau|\tau',\ \_|\_\rangle$. By the premise for $\llbracket C \rrbracket$, we get $\hat{\tau}|\hat{\tau}' \models_\pi \mathcal{Q}$, where $\hat{\tau}, \hat{\tau}'$ are the extensions using $\overline{v}, \overline{v}'$.

Write. Very similar to the argument for Post.

Safety. As the steps for let and elet do not fault, a faulting execution gives one of the form

$$\langle \llbracket C \rrbracket,\ \sigma|\sigma',\ [m{:}B]|[m{:}B']\rangle \stackrel{\varphi}{\Longrightarrow}{}^* \langle DD,\ \tau|\tau',\ \dot{\mu}|\dot{\mu}'\rangle \stackrel{\varphi}{\Longrightarrow} \lightning.$$

We show this contradicts the premises, by cases on whether the trace up to DD is m-truncated.

Case m-truncated. The active command of D (equivalently, of $\llbracket D \rrbracket$) is not a call to m, because an environment call does not fault on its first step; it goes by rule bCallE. By the Claim, we have $\langle \llbracket C \rrbracket,\ \sigma|\sigma',\ \_|\_\rangle \stackrel{\varphi\theta}{\Longrightarrow}{}^* \langle DD,\ \tau|\tau',\ \mu|\mu'\rangle$. Because the active command is not a call to m, the step $\langle DD,\ \tau|\tau',\ \dot{\mu}|\dot{\mu}'\rangle \stackrel{\varphi}{\Longrightarrow} \lightning$ can also be taken via $\stackrel{\varphi\theta}{\Longrightarrow}$. But then we have a faulting trace that contradicts the premise for $\llbracket C \rrbracket$.

Case not m-truncated. A trace with an incomplete call of m has the following form. (Here, we rely on the Claim to write parts in fully aligned form.)

$$\begin{aligned}\langle \llbracket C \rrbracket,\ \sigma|\sigma',\ [m{:}B]|[m{:}B']\rangle &\stackrel{\varphi}{\Longrightarrow}{}^* \langle \lfloor m() \rfloor; \llbracket D_0 \rrbracket,\ \tau_0|\tau_0',\ \dot{\mu}|\dot{\mu}'\rangle \\ &\stackrel{\varphi}{\Longrightarrow} \langle (B|B'); \llbracket D_0 \rrbracket,\ \tau_0|\tau_0',\ \dot{\mu}_0|\dot{\mu}_0'\rangle \stackrel{\varphi}{\Longrightarrow}{}^* \langle BB_0; \llbracket D_0 \rrbracket,\ \tau|\tau',\ \dot{\mu}|\dot{\mu}'\rangle \stackrel{\varphi}{\Longrightarrow} \lightning,\end{aligned}$$

with $BB_0 \not\equiv \lfloor \mathsf{skip} \rfloor$. Applying the Claim to the m-truncated prefix, we get $\tau_0|\tau_0' \models_\rho \mathcal{R}$ for some ρ. By semantics, we get $\langle (B|B'),\ \tau_0|\tau_0',\ \dot{\mu}_0|\dot{\mu}_0'\rangle \stackrel{\varphi}{\Longrightarrow}{}^* \langle BB_0,\ \tau|\tau',\ \dot{\mu}|\dot{\mu}'\rangle \stackrel{\varphi}{\Longrightarrow} \lightning$. Now, $(B|B')$ has no calls to m—because we are proving soundness assuming there is no recursion. So the same

transitions can be taken via $\stackrel{\varphi\theta}{\longmapsto}$. But then we get a faulting trace that contradicts the premise for $(B|B')$.

R-safety. For any trace T of let $m=(B|B')$ in $\llbracket C \rrbracket$ from σ, σ' satisfying $\mathcal{P}$, we must show that the left projection U and right projection V is r-safe for $(\Phi_0, \varepsilon, \sigma)$ and $(\Phi_1, \varepsilon, \sigma')$, respectively. Observe that the premises for $\llbracket C \rrbracket$ and for $(B|B')$ give r-safety of their left projections, for $((\Phi_0, \Theta_0), \varepsilon, \sigma)$, and r-safety of their right projection for $((\Phi_1, \Theta_1), \varepsilon, \sigma')$). For methods of Φ, by definition of r-safety, these are the same conditions as r-safety for $(\Phi_0, \varepsilon, \sigma)$ and for $(\Phi_1, \varepsilon, \sigma')$. Let us consider U, as the argument for V is symmetric. We must show the r-safety condition for any configuration, say U_i. Let $\dot{T}$ the prefix of T such that U_i is aligned (by projection Lemma) with the last configuration of $\dot{T}$. Now go by cases on whether $\dot{T}$ is m-truncated.

case $\dot{T}$ is m-truncated. If the last configuration is calling m, then there is nothing to prove. Otherwise, that configuration is not within a call of m, so by the Claim, we get from $\dot{T}$ a trace $\ddot{T}$ of $\llbracket C \rrbracket$ via $\stackrel{\varphi}{\Longmapsto}$ that ends with the same configuration. Now can appeal to r-safety from the premise for $\llbracket C \rrbracket$, and we are done. (The claim does not address the first step of let $m = (B|B')$ in $\llbracket C \rrbracket$, but that satisfies r-safety by definition.)

case $\dot{T}$ is not m-truncated. So a suffix of $\dot{T}$ is an incomplete environment call of m, say at position j. By the Claim, the call is sync'd (and m's relational precondition holds), so the code of $\dot{T}_j$ has the form $\lfloor m() \rfloor; DD$ for some continuation code DD, and the following steps execute starting from $(B|B'); DD$ (by transition rule bCallE). By dropping "$; DD$" from each configuration, we obtain a trace of $(B|B')$ that includes configuration $\dot{T}_j$. Now, we can appeal to r-safety from the premise for $(B|B')$, and we are done.

Encap. For any trace of let $m=(B|B')$ in $\llbracket C \rrbracket$ from σ, σ' satisfying $\mathcal{P}$, we must show that the left projection respects $(\Phi_0, \bullet, \varphi_0, \varepsilon, \sigma)$ and the right respects $(\Phi_1, \bullet, \varphi_1, \varepsilon, \sigma')$. The proof is structured similarly to the proof of R-safe, though it is a bit more intricate.

Observe that the premises yield respect of $((\Phi_0, \Theta_0), \bullet, (\varphi_0, \theta_0), \varepsilon, \sigma)$ and $((\Phi_1, \Theta_1), \bullet, (\varphi_1, \theta_1), \varepsilon, \sigma')$. By contrast with the argument above for r-safety, where the meaning of the condition for the conclusion is very close to its meaning for the premises, for respect there are two significant differences. First, the respect condition depends on the current module $\bullet$, and the judgment for $(B|B')$ is for a possibly different module. Second, respect depends on the modules in context, and by side conditions of the rule the modules of Φ are not the same as those of (Φ, Θ). Fortunately, these differences are exactly the same in the setting of rule Link. The proof Encap for Link (Section B.10) shows in detail how respect, for traces of let $m = B$ in C, follows from respect for traces of B and for traces of C in which calls to m are context calls.

Now, we proceed to prove Encap. For any trace T of let $m=(B|B')$ in $\llbracket C \rrbracket$ from σ, σ' satisfying $\mathcal{P}$, consider its left projection U (the right having a symmetric proof), which is a trace of let $m=B$ in C. Consider any step in U, say U_{i-1} to U_i.

If the step is an environment call to m, i.e., the call is the active command of U_{i-1}, then it satisfies respect of $(\Phi_0, \bullet, \varphi_0, \varepsilon, \sigma)$ by definitions and semantics. If the active command is ecall(m), then again we get respect by definitions and semantics. Otherwise, let $\dot{T}$ be the prefix of T such that the last configuration corresponds with U_i, and go by cases on whether $\dot{T}$ is m-truncated.

case $\dot{T}$ is m-truncated. So the step is not within a call of m, and is present in the trace $\ddot{T}$ given by the Claim. So, we can appeal to the premise for $\llbracket C \rrbracket$. We get that the step respects $(\Phi_0, \bullet, \varphi_0, \varepsilon, \sigma)$, using the arguments in the Link proof to connect with respect of $((\Phi_0, \Theta_0), \bullet, (\varphi_0, \theta_0), \varepsilon, \sigma)$ in accord with the premise for $\llbracket C \rrbracket$.

case $\dot{T}$ is not m-truncated. As in the r-safety argument, we obtain a trace of $(B|B')$ that includes the step in question, and it respects $(\Phi_0, \bullet, \varphi_0, \varepsilon, \sigma)$, using the arguments in the Link proof to connect with respect of $((\Phi_0, \Theta_0), mdl(m), (\varphi_0, \theta_0), \varepsilon, \sigma)$ in accord with the premise for $(B|B')$.

D.11 Soundness of rWeave

$$\textsc{rWeave}\ \frac{\Phi \vdash DD : \mathcal{P} \approx\!> \mathcal{Q}\,[\varepsilon|\varepsilon'] \qquad CC \hookrightarrow^* DD}{\Phi \vdash CC : \mathcal{P} \approx\!> \mathcal{Q}\,[\varepsilon|\varepsilon']}$$

Remark 13. In general, $\Phi \models DD : \mathcal{P} \approx\!> \mathcal{Q}\,[\varepsilon]$ and $DD \hookrightarrow CC$ do not imply $\Phi \models CC : \mathcal{P} \approx\!> \mathcal{Q}\,[\varepsilon]$, for one reason: CC may assert additional test agreements that do not hold.

The crux of the soundness proof for rule rWeave is soundness for a single weaving step, $CC \hookrightarrow DD$, which is Lemma D.4 below. Using the lemma, we can prove soundness of rWeave by induction on the number of weaving steps $CC \hookrightarrow^* DD$. In case of zero steps, $CC \equiv DD$ and the result is immediate. In case of more than one steps, apply Lemma D.4 and the induction hypothesis.

Before proving Lemma D.4, we prove preliminary results.

Lemma D.1 (Weave and Project). *If $CC \hookrightarrow DD$, then $\overleftarrow{CC} \equiv \overleftarrow{DD}$ and $\overrightarrow{CC} \equiv \overrightarrow{DD}$.*

Proof. By induction on the rules for $\hookrightarrow$ (Figure 18), making straightforward use of the definitions of the syntactic projections. As an example, for the if-else axiom, we have $\overleftarrow{(\text{if } E \text{ then } C \text{ else } D \mid \text{if } E' \text{ then } C' \text{ else } D')} \equiv \text{if } E \text{ then } \overleftarrow{C|C'} \text{ else } \overleftarrow{D|D'} \equiv \overleftarrow{\text{if } E|E' \text{ then } (C|C') \text{ else } (D|D')}$. As an example inductive case, for the rule from $BB \hookrightarrow CC$ infer $BB; DD \hookrightarrow CC; DD$, we have $\overleftarrow{BB;DD} \equiv \overleftarrow{BB};\overleftarrow{DD} \equiv \overleftarrow{CC};\overleftarrow{DD} \equiv \overleftarrow{CC;DD}$ where the middle step is by induction hypothesis. □

Lemma D.2 (Trace Coverage). *Suppose $\Phi \models DD : \mathcal{P} \approx\!> \mathcal{Q}\,[\varepsilon]$ and let φ be a Φ-model. Consider any π and any σ, σ' such that $\sigma|\sigma' \models_\pi \mathcal{P}$. Let U and V be traces from $\langle \overleftarrow{DD},\ \sigma,\ \_\rangle$ and $\langle \overrightarrow{DD},\ \sigma',\ \_\rangle$, respectively. Then there is a trace T from $\langle DD,\ \sigma|\sigma',\ \_|\_\rangle$, with projections W, X such that $U \leq W$ and $V \leq X$.*

Proof. Apply embedding Lemma C.9 to U, V to obtain T, W, X satisfying one of the conditions (a), (b), (c), or (d) in that Lemma. Conditions (b), (c), and (d) contradict the premise, specifically Safety for DD. That leaves condition (a), which completes the proof. □

Lemma D.3 (Weave and Trace). *Suppose $\Phi \models DD : \mathcal{P} \approx\!> \mathcal{Q}\,[\varepsilon]$ and $CC \hookrightarrow DD$ or $DD \hookrightarrow CC$. Consider any Φ-model φ. Consider any π and any σ, σ' such that $\sigma|\sigma' \models_\pi \mathcal{P}$. Consider any trace S from $\langle CC,\ \sigma|\sigma',\ \_|\_\rangle$ and let U, V be the projections of S according to the projection Lemma 7.8. Then there is a trace T from $\langle DD,\ \sigma|\sigma',\ \_|\_\rangle$, with projections W, X such that $U \leq W$ and $V \leq X$.*

Proof. Using $CC \hookrightarrow DD$ or $DD \hookrightarrow CC$, by Lemma D.1, we have $\overleftarrow{\langle DD,\ \sigma|\sigma',\ \_|\_\rangle} = \langle \overleftarrow{CC},\ \sigma,\ \_\rangle$ and $\overrightarrow{\langle DD,\ \sigma|\sigma',\ \_|\_\rangle} = \langle \overrightarrow{CC},\ \sigma,\ \_\rangle$, so we get the result by Lemma D.2. □

Finally, we proceed to prove soundness for a single weaving step. The hard case is Safety, for reasons explained in the proof.

Lemma D.4 (One Weave Soundness). *Suppose $\Phi \models DD : \mathcal{P} \approx\!> \mathcal{Q}\,[\varepsilon]$ and $CC \hookrightarrow DD$. Then $\Phi \models CC : \mathcal{P} \approx\!> \mathcal{Q}\,[\varepsilon]$.*

Proof. Suppose $\Phi \models DD : \mathcal{P} \approx\!> \mathcal{Q}\,[\varepsilon]$ and $CC \hookrightarrow DD$. To show the conclusion $\Phi \models CC : \mathcal{P} \approx\!> \mathcal{Q}\,[\varepsilon]$, consider any Φ-model φ. Consider any π and any σ, σ' such that $\sigma|\sigma' \models_\pi \mathcal{P}$.

R-safe. Consider any trace S of CC from σ, σ'. By Lemma D.3, there is a trace T of DD such that every unary step in S is covered by a step in T. So r-safety follows from r-safety of the premise.

Encap. Similar to R-safe.

Write and Post. By Lemma D.3, a terminated trace of CC gives rise to one of DD with the same final states, to which the premise applies.

Safety. This requires additional definitions and results. Faults by CC may be alignment faults (rules bCallX, bIfX, bWhX) or due to unary faults (bSyncX, bComLX, bComRX). The latter can be ruled out by reasoning similar to the above, but alignment faults pose a challenge, because weaving rearranges the alignment of execution steps. We proceed to develop some technical notions about alignment faults, and use them to prove Safety.

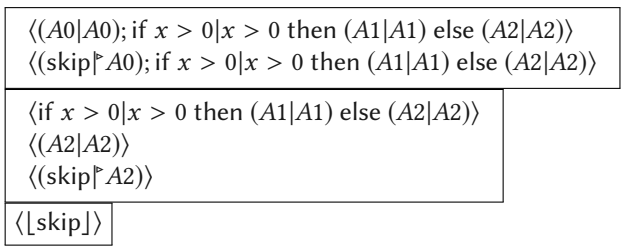
In most of this article, we only need to consider traces from initial configurations $\langle CC, \sigma|\sigma', \mu|\mu'\rangle$ where the environments are empty (written _) and the code has no endmarkers. In the following definitions, we need to consider non-empty initial environments, and CC may be an extended biprogram; in particular, CC may include endmarkers. (It turns out that we will not have occasion to consider an initial biprogram CC that contains a right-bi-com.) This is needed because, in the proof of Lemma D.5 below, specifically the case of weaving the body of a bi-let, we apply the induction hypothesis to a trace in which the initial environments are non-empty. The initial configuration of a trace must still be well formed: free variables in CC should be in the states, and methods called in CC must be in either the context or the environment and not in both.

Define a ***sync point*** in a biprogram trace T to be a position i, $0 \le i < len(T)$, such that one of the following holds:

- $i = 0$ (i.e., T_i is the initial configuration),
- The configuration T_i is terminal, i.e., has code $\lfloor \mathsf{skip} \rfloor$,
- $Active(T_i)$ is not a bi-com, i.e., neither $(-|-)$ nor $(-|^{\triangleright}-)$. Thus, $Active(T_i)$ may be $\lfloor - \rfloor$, bi-if, bi-while, bi-let, or bi-var (by definition, the active biprogram is not a sequence),
- $i > 0$ and the step from T_{i-1} to T_i completed the first part of a biprogram sequence. That is, the code in T_{i-1} has the form $CC; DD$ with CC the active command, and the code in T_i is DD. Such a transition is a transition from CC to $\lfloor \mathsf{skip} \rfloor$ that is lifted to $CC; DD$ by rule bSeq.[48] Later, we refer to this kind of step as a "semi-colon removal."

A ***segment*** of a biprogram trace is just a list of configurations that occur contiguously in the trace. A ***segmentation*** of trace T is a list L of nonempty segments, the catenation of which is T. Thus, indexing the list L from 0, the configuration $(L_i)_j$ is T_{n+j} where $n = \Sigma_{0 \le k < i} len(L_k)$. An ***alignment segmentation*** of T is a segmentation L such that each segment in L begins with a sync point of T.

For an example, using abbreviations $A0 \triangleq x := 0$, $A1 \triangleq x := 1$, $A2 \triangleq x := 2$ and omitting states/environments from the configurations, here is a trace with one of its alignment segmentations depicted by boxes:



Every trace has a minimal-length alignment segmentation consisting of the trace itself—a single segment—and also a maximal-length alignment segmentation (which has a segment for each sync

[48] One could make this more explicit by dropping the identification of $\lfloor \mathsf{skip} \rfloor; DD$ with DD and instead having a separate transition from $\lfloor \mathsf{skip} \rfloor; DD$ to DD, but this would make extra cases in other proofs.

point). (Keep in mind that we define traces to be finite.) The above example, with three segments, is maximal.

As another example, here is a trace that faults next (because $x > 0$ is false on the left but true on the right), with its maximal alignment segmentation.

$$\boxed{\begin{array}{l}\langle (x := 0 | x := 1); \mathsf{if}\ x > 0 | x > 0\ \mathsf{then}\ \lfloor A1 \rfloor\ \mathsf{else}\ \lfloor A2 \rfloor\rangle \\ \langle (\mathsf{skip} \triangleright x := 1); \mathsf{if}\ x > 0 | x > 0\ \mathsf{then}\ \lfloor A1 \rfloor\ \mathsf{else}\ \lfloor A2 \rfloor\rangle\end{array}}$$
$$\boxed{\langle \mathsf{if}\ x > 0 | x > 0\ \mathsf{then}\ \lfloor A1 \rfloor\ \mathsf{else}\ \lfloor A2 \rfloor\rangle}$$

Note that a segment can begin with a configuration that contains end-markers whose beginning was in a previous segment. For example,

$$\boxed{\begin{array}{l}\langle \mathsf{var}\ x : T | x' : T'\ \mathsf{in}\ (a|b); (c|d)\rangle \\ \langle (a|b); (c|d); (\mathsf{evar}(x) | \mathsf{evar}(x'))\rangle \\ \langle (\mathsf{skip} \triangleright b); (c|d); (\mathsf{evar}(x) | \mathsf{evar}(x'))\rangle\end{array}}$$
$$\boxed{\begin{array}{l}\langle (c|d); (\mathsf{evar}(x) | \mathsf{evar}(x'))\rangle \\ \langle (\mathsf{skip} \triangleright d); (\mathsf{evar}(x) | \mathsf{evar}(x'))\rangle \\ \langle (\mathsf{evar}(x) | \mathsf{evar}(x'))\rangle \\ \langle (\mathsf{skip} \triangleright \mathsf{evar}(x'))\rangle \\ \langle \lfloor \mathsf{skip} \rfloor\rangle\end{array}}$$

In the following, we sometimes refer to the left and right sides of a weaving as lhs and rhs. A weaving $lhs \rightsquigarrow rhs$ introduces sync points in the biprogram's traces, but it does not remove sync points of lhs. Moreover, though it rearranges the order in which the underlying unary steps are taken, it does not change the states that appear at sync points. This is made precise in the following lemma, which gives a sense in which weaving is directed (i.e., not commutative).

Lemma D.5 (Weaving Preserves Sync Points). *Consider any pre-model φ. Consider any biprograms CC and DD such that $CC \rightsquigarrow DD$. Let S be a trace (via φ) of CC from some initial states and environments. (No assumption is made about the initial states, and non-empty method environments are allowed.) Let L be the maximal alignment segmentation of S. Then there is a trace T of DD from the same states and environments, such that either*

(i) the last configuration of T can fault next, by alignment fault; or
(ii) there is an alignment segmentation M of T such that M has the same length as L and for all i, segment M_i and segment L_i begin with the same states, same environments, and same underlying unary programs, that is

$$\overleftarrow{(L_i)_0} = \overleftarrow{(M_i)_0} \text{ and } \overrightarrow{(L_i)_0} = \overrightarrow{(M_i)_0}. \tag{71}$$

Note that M in Lemma D.5 need not be the maximal segmentation. Typically T will have additional sync points, but these are not relevant to the conclusion of the lemma. What matters is that T covers the sync points of S. (Note that T need not cover all the steps of S.) As an example of the lemma, consider a biprogram of the form $\langle (A0|A0); \mathsf{if}\ x > 0 | x > 0\ \mathsf{then}\ (A1|A1)\ \mathsf{else}\ (A2|A2)\rangle$. It relates by $\rightsquigarrow$ to $\langle (A0|A0); \mathsf{if}\ x > 0 | x > 0\ \mathsf{then}\ (A1|A1)\ \mathsf{else}\ \lfloor A2 \rfloor\rangle$ (by an axiom and the congruence rules for sequence and conditional). From the same initial states (and empty environments), the latter biprogram has a shorter trace (owing to sync'd execution of $A2$) but that trace can still be segmented in accord with the lemma. Its second segment has three configurations:

$$\langle \mathsf{if}\ x > 0 | x > 0\ \mathsf{then}\ (A1|A1)\ \mathsf{else}\ \lfloor A2 \rfloor\rangle\langle \lfloor A2 \rfloor\rangle\langle \lfloor \mathsf{skip} \rfloor\rangle.$$

We defer the proof of Lemma D.5 and use it to finish the proof of Lemma D.4 by completing the proof of Safety. As before, we assume $\Phi \models DD : \mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon]$ and $CC \hookrightarrow DD$. To show the Safety condition for $\Phi \models CC : \mathcal{P} \approx\!> \mathcal{Q}\ [\varepsilon]$, consider any Φ-model φ. Consider any π and any σ, σ' such that $\sigma|\sigma' \models_\pi \mathcal{P}$. Suppose CC has a trace S from σ, σ' (and empty environments). If S faults next by a unary fault, then let its unary projections be U, V (one of which faults next). Then by Lemma D.3 the trace T from U, V must also fault next—and this contradicts the assumed judgment for DD.

Finally, suppose S faults next by alignment fault. Consider the maximal alignment segmentation of S and let T be the trace from DD given by Lemma D.3. By Lemma D.5 there is a segmentation of T that covers each sync point of S, including the last configuration of S, which faults. But then T faults next, contrary to the premise for DD.

This concludes the proof of Lemma D.4 and thus soundness of rWeave.

Proof. (Of Lemma D.5). By induction on the derivation of the weaving relation $CC \hookrightarrow DD$, and by cases on the definition of $\hookrightarrow$ starting with the axioms (Figure 18).

Case weaving axiom $(A|A) \hookrightarrow \lfloor A \rfloor$. For most atomic commands A, a trace S of the lhs consists of an initial configuration $\langle (A|A),\ \sigma|\sigma',\ \mu|\mu' \rangle$, possibly a second one with code $(\mathsf{skip}|^{r} A)$, and possibly a third one that is terminated (i.e., has code $\lfloor \mathsf{skip} \rfloor$). However, because the lemma allows non-empty environments, there is also the case that A is an environment call to some m in the domain of μ and of μ'. In that case, if $\mu(m) = B$ and $\mu'(m) = B'$, then there are traces of the form $\langle (m()|m()) \rangle \langle (B; \mathsf{ecall}(m)|^{r} m()) \rangle \langle (B; \mathsf{ecall}(m)|B'; \mathsf{ecall}(m)) \rangle \ldots$. Traces of the $\lfloor m() \rfloor$ can have the form $\langle \lfloor m() \rfloor \rangle \langle (B|B') \rangle \ldots$ but also, if $B' \equiv B$, the form $\langle \lfloor m() \rfloor \rangle \langle \lVert B \rVert \rangle \ldots$ (see rule bCallE and Figure 20). The latter is susceptible to alignment faults.

In any case, the only sync points in S are the initial configuration and, if present, the terminated one. If S is not terminated, then it has only the initial sync point, so L has only a single segment. This can be matched by the trace T consisting of the one configuration $\langle \lfloor A \rfloor,\ \sigma|\sigma',\ \mu|\mu' \rangle$, which also serves as the single segment for T. (The lemma does not require T to cover all steps of S, only the sync points of S.)

If S terminated, then by projection and then embedding Lemma C.9, $\langle \lfloor A \rfloor, \sigma|\sigma', \mu|\mu' \rangle$ has a trace T that either terminates, covering the steps of S, or faults. It cannot have a unary fault, because S did not. If it has an alignment fault, which would be via context call transition bCallX or by some step of an environment call executing $\lVert B \rVert$, then we are done. Otherwise, T can be segmented to match the segmentation L: One segment including all of T except the last configuration, followed by that configuration as a segment.

Case weaving axiom $(C; D \mid C'; D') \hookrightarrow (C|C'); (D|D')$. A trace S of the lhs may make several steps, and may eventually terminate. If terminated, then it has two sync points, initial and final; otherwise, only the initial configuration is a sync point. If not terminated, then the initial configuration for $(C|C'); (D|D')$ provides the trace T and its single segment. If S terminated, then by projection and embedding, we obtain a trace T that either terminates in the same states or has an alignment fault. So, we either get a matching segmentation of T or an alignment fault.

Cases the other weaving axioms. The argument is the same as above, in all cases. The rhs of weaving has additional sync points, which are of no consequence, except that they can give rise to alignment faults. Like the preceding cases, bi-if and bi-while introduce the possibility of alignment fault; bi-let and bi-var weavings do not.

Having dispensed with the base cases, we turn to the inductive cases, which each have as premise that $BB \hookrightarrow CC$ (Figure 18). The induction hypothesis is that for any trace S of BB and any alignment segmentation L of S, there is a trace T of CC such that either its last configuration can alignment-fault or there is a segmentation M of T that covers the segmentation of S.

Case $BB; DD \hookrightarrow CC; DD$.

A trace S of $BB; DD$ may include only execution of BB or may continue to execute DD.

- In case S never starts DD, the trace S determines a trace S^+ of BB by removing the trailing "$; DD$" from every configuration. (In the special case that CC is run to completion in S, i.e., its last configuration has exactly the code DD, then the last configuration of S^+ has $\lfloor \mathsf{skip} \rfloor$.) (Note that S may have sync points besides the initial one, as BB is an arbitrary biprogram.) By induction, we obtain trace T of CC and either alignment fault or segmentation of T that covers the segmentation of S. Adding $; DD$ to every configuration of T yields the requisite segmentation of S.
- Now consider the other case: S includes at least one step of DD, so there is some $i > 0$ such that S_{i-1} has code $BB'; DD$ for some BB' that steps to $\lfloor \mathsf{skip} \rfloor$, and S_i has code DD. Because L is the maximal segmentation of S, it includes a segment that starts with the configuration S_i. Now we can proceed as in the first bullet, to obtain trace T of CC and either alignment fault or segmentation for the part of S up to but not including position i. Catenating this segmentation with the one for the trace of DD from i yields the result.

Case $DD; BB \rightarrowtail DD; CC$. For a trace S that never reaches BB, the result is immediate by taking $T := S$ and $M := L$. Otherwise, the given trace S can be segmented into an execution of DD that terminates, followed by a terminating execution of BB. By maximality, the segmentation breaks at the semicolon, and we obtain the result using the induction hypothesis similarly to the preceding case.

Case $\mathsf{if}\ E|E'\ \mathsf{then}\ BB\ \mathsf{else}\ DD \rightarrowtail \mathsf{if}\ E|E'\ \mathsf{then}\ CC\ \mathsf{else}\ DD$. If the given trace S has length one, then we immediately obtain a length-one trace and segmentation that satisfies the same-projection condition (71).

If $len(S) > 1$, then the first step does not fault, i.e., the tests agree. Let S^+ be the trace starting at position 1, which is a trace of BB or of DD depending on whether the tests are initially true or false. If the tests are false, then catenating the initial configuration for $\mathsf{if}\ E|E'\ \mathsf{then}\ CC\ \mathsf{else}\ DD$ with S^+ provides the requisite T, and also its segmentation. If the tests are true, then apply the induction hypothesis to obtain a trace T for CC, and segmentation (if not alignment fault); and again, prefixing the initial configuration to T and to its first segment yields the result.

Case $\mathsf{if}\ E|E'\ \mathsf{then}\ DD\ \mathsf{else}\ BB \rightarrowtail \mathsf{if}\ E|E'\ \mathsf{then}\ DD\ \mathsf{else}\ CC$. Symmetric to the preceding case.

Case $\mathsf{while}\ E|E' \cdot \mathcal{P}|\mathcal{P}'\ \mathsf{do}\ BB \rightarrowtail \mathsf{while}\ E|E' \cdot \mathcal{P}|\mathcal{P}'\ \mathsf{do}\ CC$. A trace S of lhs can be factored into a series of zero or more iterations possibly followed by an incomplete iteration of left/right/both. Note that a completed iteration ends with a "semi-colon removal" step (the left-, right-, or both-sides loop body finishes and was followed by the bi-loop). Because the segmentation L is maximal, it has a separate segment for each iteration.

Now the argument goes by induction on the number of iterations. The inner induction hypothesis yields segmentation for rhs up to the last iteration, which in turn ensures that lhs and rhs agree on whether the last iteration is left-only, right-only, or both-sides. In the one-sided cases there are no sync points. In the both-sides case, the main induction hypothesis for $BB \rightarrowtail CC$ can be used in a way similar to the argument for sequence weaving above.

Case $\mathsf{let}\ m = (B|B')\ \mathsf{in}\ BB \rightarrowtail \mathsf{let}\ m = (B|B')\ \mathsf{in}\ CC$. Suppose S is a trace from $\langle \mathsf{let}\ m = (B|B')\ \mathsf{in}\ BB, \sigma|\sigma', \mu|\mu' \rangle$, with segmentation L. If S has length one, then the rest is easy. Otherwise, S takes at least one step, to $\langle BB; \lfloor \mathsf{elet}(m) \rfloor, \sigma|\sigma', \hat{\mu}|\hat{\mu}' \rangle$ where $\hat{\mu}$ and $\hat{\mu}'$ extend μ, μ' with $m{:}B$ and $m{:}B'$, respectively. We obtain trace S^+ of $\langle BB; \lfloor \mathsf{elet}(m) \rfloor, \sigma|\sigma', \hat{\mu}|\hat{\mu}' \rangle$ by omitting the first configuration of S—and here we use a trace where the initial environments are non-empty. Applying the induction hypothesis, we obtain trace T^+ for S^+, and either alignment fault or matching segmentation M^+. Prefixing the configuration $\langle \mathsf{let}\ m = (B|B')\ \mathsf{in}\ CC, \sigma|\sigma', \mu|\mu' \rangle$ yields the requisite trace T. If there is alignment fault, then we are done. Otherwise, if BB begins with an aligning bi-program, i.e., if S_1 is

a sync point in S, then let segmentation M consist of the singleton $\langle \text{let } m = (B|B') \text{ in } CC,\ \sigma|\sigma',\ \mu|\mu' \rangle$ followed by the elements of M^+. Finally, if S_1 is not a sync point in S, then we obtain M by prefixing $\langle \text{let } m = (B|B') \text{ in } CC,\ \sigma|\sigma',\ \mu|\mu' \rangle$ to the first segment in M^+.

Case var $x{:}T|x'{:}T'$ in $BB \leftrightarrowtail$ var $x{:}T|x'{:}T'$ in CC. By semantics and induction hypothesis, similar to the preceding case for bi-let.

E GUIDE TO IDENTIFIERS AND NOTATIONS

The prime symbol, like σ', is consistently used for right side in a pair of commands, states, and so on. Other decorations, like $\dot{\sigma}$ and $\ddot{\tau}$, are used for fresh identifiers in general.

Table 1. Use of Identifiers

A	atomic command	Figure 5
B, C, D	command	Figure 5
BB, CC, DD	biprogram	Figure 5
E	program expression	Figure 5
G, H	region expression	Figure 5
F	either program or region expression	Figure 5
f, g	field name	Figure 5, Equation (6)
K	reference type	Figure 5
M, N, L	module name	
T	data type	Figure 5
T, U, V, W	trace (unary or biprogram)	
P, Q, R	formula	Figure 9
$\mathcal{P}, \mathcal{Q}, \mathcal{R}, \mathcal{M}, \mathcal{N}$	relation formula	Figure 14
x, y, z, r, s	program variable	
$\varepsilon, \eta, \delta$	effect expression	Equation (6)
Γ	typing context	
$\Phi, \Theta, \Psi,$	unary or relational hypothesis context	Sections 3.4 and 4.3
φ, θ, ψ	unary or relational context model	Sections 5.4 and 7.4
Φ_0, Φ_1, Φ_2	components of relational context	see preceding Definition 4.2
σ, τ, υ	state	Section 5.1
$\hat{\sigma}$	state with spec-only vars	
π, ρ	refperm	Section 5.2

Table 2. Use of Symbols

$\cdot/.$	separator function	Equation (29)
$\bullet$	default/main module	Section 3.2
$\bullet$	empty effect	Equation (6)
$\varepsilon \backslash \eta$	effect subtraction	following Definition 3.1
$(+ - . -)$	combination of effects	following Definition 3.1
$\text{`}f$	image in region expression or effect	Figure 5, Equation (6)
$\#$	disjoint regions	Figure 9
$\preceq \ \prec$	module import	Section 3.2
$\doteq$	equal reference or region, modulo refperm	Figure 14, Figure 25
$\mathbb{A}x \ \ \mathbb{A}G\text{`}f$	agreement formulas	Figure 14, Figure 25
$\{\!\{ - \}\!\}, \ \{\!\| - \}\!\}, \ \mathbb{B}-$	embed unary formula (left, right, both)	Figure 14, Figure 25
$\{\!\{ - \}\!\}, \ \{\!\| - \}\!\}$	embed unary expression	Figure 14, Figure 25
$\Diamond$	possibly (in an extended refperm)	Figure 14, Figure 25
$\textcircled{\wedge}$	conjoin invariant	Definition 4.7
$\llbracket - \rrbracket$	full alignment of command	Figure 20
$\hookrightarrow$	weave biprogram	Figure 18
$[\sigma + x\colon v]$	extend state to map x to v	Section 5.1
$[\sigma \mid x\colon v]$	update value of x	Section 5.1
$\sigma \restriction x$	drop variable x from state	Section 5.1
$\hookrightarrow$	can succeed	Section 5.2
$\delta^{\oplus}$	abbreviates effect δ, rd alloc	preceding Definition 5.10
$\overset{\pi}{\sim}$	equiv modulo refperm	Section 5.2
$\overset{\pi\mid\pi'}{\approx} \ \cong_{\pi\mid\pi'}$	state pair isomorphism	Definition 7.3
$\overset{\pi}{\approx} \ \cong_{\pi}$	state isomorphism, outcome equivalence	Definition 5.5
$\overset{\varphi}{\longmapsto} \ \overset{\varphi}{\longmapsto}{}^*$	unary transitions	Figures 22 and 34
$\overset{\varphi}{\Longmapsto} \ \overset{\varphi}{\Longmapsto}{}^*$	biprogram transitions	Figures 27 and 28
$(C \restriction C')$	r-bi-com biprogram	Section 7.3
$\sigma \rightarrow \tau \models \varepsilon$	allows change	Section 5.2
$\tau, \tau' \overset{\pi}{\Rightarrow} v, v' \models^{\sigma}_{\delta} \varepsilon$	allowed dependence	Definition A.2
$P \models \varepsilon \leq \eta$	subeffect judgment	Equation (26)
$P \models P \ \mathsf{frm} \ \varepsilon$	framing of a formula	Equation (27)
$\mathcal{P} \models \eta \mid \eta' \ \mathsf{frm} \ \mathcal{Q}$	framing of a relation	Section 7
$\Phi \vdash_M C : P \rightsquigarrow Q \ [\varepsilon]$	correctness judgment	Definition 3.3, Definition 5.10
$\Phi \vdash_M CC : \mathcal{P} \twoheadrightarrow \mathcal{Q} \ [\varepsilon \mid \varepsilon']$	relational correctness judgment	Definition 4.2, Definition 7.10
$locEq_{\delta}(P \rightsquigarrow Q \ [\varepsilon]) \quad LocEq_{\delta}(\Phi)$	local equivalence specs	Definition 8.4
$\Rrightarrow$	covariant spec implication	Definition 8.5

ACKNOWLEDGMENTS

We thank the anonymous TOPLAS reviewers for their insightful technical feedback and stuctural suggestions, which have improved the exposition. We thank Andrew Myers for his encouragement and diligent editing throughout the reviewing process. Stephen Sondheim's lyrics "Perpetual anticipation is good for the soul/But it's bad for the heart" provided perspective as we worked through multiple review iterations.

The ideas in this article arose from discussions between Banerjee and Naumann during a long walk at PLDI 2009 in Dublin, following which, Naumann jotted down initial thoughts at a cafe. The discussions spurred a long-term research program that has produced substantial intermediate results (RLI–RLIII) that have culminated in this article. For arranging presentations of the work at various stages of its development, and for their comments and encouragement, we thank Nina Amla, Lennart Beringer, Lars Birkedal, Stephen Chong, Rance Cleaveland, Matthias Felleisen, Philippa Gardner, Neil Immerman, Patricia Johann, Assaf Kfoury, Shriram Krishnamurthi, César

Kunz, Gary Leavens, David Liu, Aleks Nanevski, Minh Ngo, Noam Rinetzky, Mooly Sagiv, Don Sannella, Gordon Stewart, and Jan Vitek.

We thank the organizers and participants of the Dagstuhl Seminar 18151 on Program Equivalence. The stimulating atmosphere of the seminar and Dagstuhl's salubrious environs (which naturally inspired us to take many long walks) aided technical progress at a crucial stage. Naumann acknowledges Manuel Hermenegildo for arranging an enjoyable and fruitful stay at the IMDEA Software Institute in 2011, and Andrew Appel for arranging an engaging stay at Princeton in 2017-18. Finally, we thank our families for their continuing and steadfast support.

REFERENCES

[1] Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. 2019. A relational logic for higher-order programs. *J. Funct. Program.* 29 (2019), e16. https://doi.org/10.1017/S0956796819000145

[2] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent representation independence. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. ACM, 340–353. https://doi.org/10.1145/1480881.1480925

[3] T. Amtoft, S. Bandhakavi, and A. Banerjee. 2006. A logic for information flow in object-oriented programs. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. ACM, 91–102. https://doi.org/10.1145/1111037.1111046

[4] Torben Amtoft and Anindya Banerjee. 2007. Verification condition generation for conditional information flow. In *Proceedings of the ACM Workshop on Formal Methods in Security Engineering (FMSE'07)*, Peng Ning, Vijay Atluri, Virgil D. Gligor, and Heiko Mantel (Eds.). ACM, 2–11. https://doi.org/10.1145/1314436.1314438

[5] Timos Antonopoulos, Eric Koskinen, Ton Chanh Le, Ramana Nagasamudram, David A. Naumann, and Minh Ngo. 2022. An algebra of alignment for relational verification. Retrieved from https://arxiv.org/abs/2202.04278.

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

[7] 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

[8] Anindya Banerjee and David A. Naumann. 2005. Stack-based access control and secure information flow. *J. Funct. Program.* 15, 2 (2005), 131–177. https://doi.org/10.1017/S0956796804005453

[9] 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

[10] Anindya Banerjee and David A. Naumann. 2013. State based encapsulation for modular reasoning about behavior-preserving refactorings. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Lecture Notes in Computer Science, Vol. 7850. Springer, 319–365. https://doi.org/10.1007/978-3-642-36946-9_12

[11] Anindya Banerjee, David A. Naumann, and Mohammad Nikouei. 2016. Relational logic with framing and hypotheses. In *Proceedings of the 36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (LIPIcs, Vol. 65)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 11:1–11:16. https://doi.org/10.4230/LIPIcs.FSTTCS.2016.11 Technical report at http://arxiv.org/abs/1611.08992.

[12] 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.* 40, 2 (2018), 6:1–6:90. https://doi.org/10.1145/3174801

[13] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. 2008. Expressive declassification policies and modular static enforcement. In *Proceedings of the 29th IEEE Symposium on Security and Privacy*. IEEE Computer Society, 339–353. https://doi.org/10.1109/SP.2008.20

[14] 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

[15] Yuyan Bao, Gary T. Leavens, and Gidon Ernst. 2018. Unifying separation logic and region logic to allow interoperability. *Formal Aspects Comput.* 30, 3–4 (2018), 381–441. https://doi.org/10.1007/s00165-018-0455-5

[16] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational verification using product programs. In *Proceedings of the 17th International Symposium on Formal Methods (FM'11) (Lecture Notes in Computer Science, Vol. 6664)*. Springer, 200–214. https://doi.org/10.1007/978-3-642-21437-0_17

[17] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2013. Beyond 2-safety: Asymmetric product programs for relational program verification. In *Proceedings of the International Symposium on Logical Foundations of Computer Science (LFCS'13) (Lecture Notes in Computer Science, Vol. 7734)*. Springer, 29–43. https://doi.org/10.1007/978-3-642-35722-0_3

[18] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2016. Product programs and relational program logics. *J. Log. Algebraic Methods Program.* 85, 5 (2016), 847–859. https://doi.org/10.1016/j.jlamp.2016.05.004

[19] 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)*. IEEE Computer Society, 100–114. https://doi.org/10.1109/CSFW.2004.17

[20] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2011. Secure information flow by self-composition. *Math. Struct. Comput. Sci.* 21, 6 (2011), 1207–1252. https://doi.org/10.1017/S0960129511000193

[21] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. 2013. EasyCrypt: A tutorial. In *Proceedings of the 7th Tutorial Lectures on Foundations of Security Analysis and Design (FOSAD'13) (Lecture Notes in Computer Science, Vol. 8604)*, Alessandro Aldini, Javier López, and Fabio Martinelli (Eds.). Springer, 146–166. https://doi.org/10.1007/978-3-319-10082-1_6

[22] Gilles Barthe, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2017. Coupling proofs are probabilistic product programs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, (POPL'17)*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 161–174. https://doi.org/10.1145/3009837.3009896

[23] Gilles Barthe and Tamara Rezk. 2005. Non-interference for a JVM-like language. In *Proceedings of the ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI'05)*, J. Gregory Morrisett and Manuel Fähndrich (Eds.). ACM, 103–112. https://doi.org/10.1145/1040294.1040304

[24] Bernhard Beckert and Mattias Ulbrich. 2018. Trends in relational program verification. In *Principled Software Development—Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday*, Peter Müller and Ina Schaefer (Eds.). Springer, 41–58. https://doi.org/10.1007/978-3-319-98047-8_3

[25] N. Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. ACM, 14–25. https://doi.org/10.1145/964001.964003

[26] Nick Benton, Martin Hofmann, and Vivek Nigam. 2014. Abstract effects and proof-relevant logical relations. In *Proceedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'14)*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 619–632. https://doi.org/10.1145/2535838.2535869

[27] Lennart Beringer. 2011. Relational decomposition. In *Proceedings of the 2nd International Conference on Interactive Theorem Proving (ITP'11) (Lecture Notes in Computer Science, Vol. 6898)*, Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk (Eds.). Springer, 39–54. https://doi.org/10.1007/978-3-642-22863-6_6

[28] Lennart Beringer. 2021. Verified software units. In *Proceedings of the 30th European Symposium on Programming (ESOP'21), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS'21) (Lecture Notes in Computer Science, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer, 118–147. https://doi.org/10.1007/978-3-030-72019-3_5

[29] Lennart Beringer and Andrew W. Appel. 2019. Abstraction and subsumption in modular verification of C programs. In *Proceedings of the 3rd World Congress on Formal Methods (FM'19) (Lecture Notes in Computer Science, Vol. 11800)*, Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira (Eds.). Springer, 573–590. https://doi.org/10.1007/978-3-030-30942-8_34

[30] Lars Birkedal and Hongseok Yang. 2008. Relational parametricity and separation logic. *Log. Methods Comput. Sci.* 4, 2 (2008). https://doi.org/10.2168/LMCS-4(2:6)2008

[31] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A separation logic tool to verify correctness of C programs. *J. Autom. Reason.* 61, 1–4 (2018), 367–422. https://doi.org/10.1007/s10817-018-9457-5

[32] Arthur Charguéraud and François Pottier. 2019. Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. *J. Autom. Reason.* 62, 3 (2019), 331–365. https://doi.org/10.1007/s10817-017-9431-7

[33] Andrey Chudnov, George Kuan, and David A. Naumann. 2014. Information flow monitoring as abstract interpretation for relational logic. In *Proceedings of the IEEE 27th Computer Security Foundations Symposium (CSF'14)*. IEEE, 48–62. https://doi.org/10.1109/CSF.2014.12

[34] Berkeley R. Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 1027–1040. https://doi.org/10.1145/3314221.3314596

[35] Martin Clochard, Claude Marché, and Andrei Paskevich. 2020. Deductive verification with ghost monitors. *Proc. ACM Program. Lang.* 4 (2020), 2:1–2:26. https://doi.org/10.1145/3371070

[36] Karl Crary. 2017. Modules, abstraction, and parametric polymorphism. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 100–113. https://doi.org/10.1145/3009837.3009892

[37] Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. 2010. A relational modal logic for higher-order stateful ADTs. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10)*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 185–198. https://doi.org/10.1145/1706299.1706323

[38] Mnacho Echenim, Radu Iosif, and Nicolas Peltier. 2019. The Bernays-Schönfinkel-ramsey class of separation logic on arbitrary domains. In *Proceedings of the 22nd International Conference on Foundations of Software Science and Computation Structures (FOSSACS'19), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS'19) (Lecture Notes in Computer Science, Vol. 11425)*, Mikolaj Bojanczyk and Alex Simpson (Eds.). Springer, 242–259. https://doi.org/10.1007/978-3-030-17127-8_14

[39] Marco Eilers, Peter Müller, and Samuel Hitz. 2020. Modular product programs. *ACM Trans. Program. Lang. Syst.* 42, 1 (2020), 3:1–3:37. https://doi.org/10.1145/3324783

[40] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating regression verification. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE'14)*, Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher (Eds.). ACM, 349–360. https://doi.org/10.1145/2642937.2642987

[41] Jean-Christophe Filliâtre. 2021. Simpler proofs with decentralized invariants. *J. Log. Algebraic Methods Program.* 121 (2021), 100645. https://doi.org/10.1016/j.jlamp.2021.100645

[42] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. 2016. The spirit of ghost code. *Formal Methods Syst. Design* 48, 3 (2016), 152–174. https://doi.org/10.1007/s10703-016-0243-x

[43] Nissim Francez. 1983. Product properties and their direct verification. *Acta Informatica* 20 (1983), 329–344. https://doi.org/10.1007/BF00264278

[44] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A mechanised relational logic for fine-grained concurrency. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'18)*, Anuj Dawar and Erich Grädel (Eds.). ACM, 442–451. https://doi.org/10.1145/3209108.3209174

[45] Thibaut Girka, David Mentré, and Yann Régis-Gianas. 2017. Verifiable semantic difference languages. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*, Wim Vanhoof and Brigitte Pientka (Eds.). ACM, 73–84. https://doi.org/10.1145/3131851.3131870

[46] Benny Godlin and Ofer Strichman. 2008. Inference rules for proving the equivalence of recursive procedures. *Acta Informatica* 45, 6 (2008), 403–439. https://doi.org/10.1007/s00236-008-0075-2

[47] Niklas Grimm, Kenji Maillard, Cédric Fournet, Catalin Hritcu, Matteo Maffei, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, and Santiago Zanella Béguelin. 2018. A monadic framework for relational verification: Applied to information security, program equivalence, and optimizations. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP'18)*, June Andronick and Amy P. Felty (Eds.). ACM, 130–145. https://doi.org/10.1145/3167090

[48] Walter Guttmann. 2018. Verifying minimum spanning tree algorithms with Stone relation algebras. *J. Log. Alg. Methods Program.* 101 (2018), 132–150. https://doi.org/10.1016/j.jlamp.2018.09.005

[49] 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

[50] Chris Hawblitzel, Ming Kawaguchi, Shuvendu K. Lahiri, and Henrique Rebêlo. 2013. Towards modularly comparing programs using automated theorem provers. In *Proceedings of the 24th International Conference on Automated Deduction (CADE'13) (Lecture Notes in Computer Science, Vol. 7898)*, Maria Paola Bonacina (Ed.). Springer, 282–299. https://doi.org/10.1007/978-3-642-38574-2_20

[51] C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

[52] C. A. R. Hoare. 1972. Proofs of correctness of data representations. *Acta Informatica* 1 (1972), 271–281. https://doi.org/10.1007/BF00289507

[53] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[54] Ioannis T. Kassios. 2006. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *Proceedings of the 14th International Symposium on Formal Methods (FM'06) (Lecture Notes in Computer Science, Vol. 4085)*, Jayadev Misra, Tobias Nipkow, and Emil Sekerinski (Eds.). Springer, 268–283. https://doi.org/10.1007/11813040_19

[55] Moritz Kiefer, Vladimir Klebanov, and Mattias Ulbrich. 2018. Relational program reasoning using compiler IR—Combining static verification and dynamic analysis. *J. Autom. Reason.* 60, 3 (2018), 337–363. https://doi.org/10.1007/s10817-017-9433-5

[56] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SYMDIFF: A language-agnostic semantic diff tool for imperative programs. In *Proceedings of the 24th International Conference on Computer Aided*

Verification (CAV'12) (Lecture Notes in Computer Science, Vol. 7358), P. Madhusudan and Sanjit A. Seshia (Eds.). Springer, 712–717. https://doi.org/10.1007/978-3-642-31424-7_54

[57] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. 2013. Differential assertion checking. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, Bertrand Meyer, Luciano Baresi, and Mira Mezini (Eds.). ACM, 345–355. https://doi.org/10.1145/2491411.2491452

[58] Shuvendu K. Lahiri, Andrzej S. Murawski, Ofer Strichman, and Mattias Ulbrich. 2018. Program equivalence (dagstuhl seminar 18151). *Dagstuhl Reports* 8, 4 (2018), 1–19.

[59] Leslie Lamport and Fred B. Schneider. 2021. Verifying hyperproperties with TLA. In *Proceedings of the 34th IEEE Computer Security Foundations Symposium (CSF'21)*. IEEE, 1–16. https://doi.org/10.1109/CSF51468.2021.00012

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

[61] Gary T. Leavens and David A. Naumann. 2015. Behavioral subtyping, specification inheritance, and modular reasoning. *ACM Trans. Program. Lang. Syst.* 37, 4 (2015), 13:1–13:88. https://doi.org/10.1145/2766446

[62] K. Rustan M. Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'10) (Lecture Notes in Computer Science, Vol. 6355)*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

[63] K. Rustan M. Leino and Michał Moskal. 2010. Usable auto-active verification. In *Proceedings of the Usable Verification Workshop*, Thomas Ball, Natarajan Shankar, and Lenore Zuck (Eds.). 4 pages. Retrieved from http://fm.csl.sri.com/UV10/submissions/uv2010_submission_20.pdf.

[64] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. 2002. Using data groups to specify and check side effects. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*, Jens Knoop and Laurie J. Hendren (Eds.). ACM, 246–257. https://doi.org/10.1145/512529.512559

[65] Kenji Maillard, Catalin Hritcu, Exequiel Rivas, and Antoine Van Muylder. 2020. The next 700 relational program logics. *Proc. ACM Program. Lang.* 4 (2020), 4:1–4:33. https://doi.org/10.1145/3371072

[66] Anshuman Mohan, Wei Xiang Leow, and Aquinas Hobor. 2021. Functional correctness of C implementations of dijkstra's, kruskal's, and prim's algorithms. In *Proceedings of the 33rd International Conference on Computer Aided Verification (CAV'21) (Lecture Notes in Computer Science, Vol. 12760)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 801–826. https://doi.org/10.1007/978-3-030-81688-9_37

[67] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2017. Viper: A verification infrastructure for permission-based reasoning. In *Dependable Software Systems Engineering*, Alexander Pretschner, Doron Peled, and Thomas Hutzelmann (Eds.). NATO Science for Peace and Security Series D: Information and Communication Security, Vol. 50. IOS Press, 104–125. https://doi.org/10.3233/978-1-61499-810-5-104

[68] Adithya Murali, Lucas Peña, Christof Löding, and P. Madhusudan. 2020. A first-order logic with frames. In *Proceedings of the 29th European Symposium on Programming (ESOP'20), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS'20) (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 515–543. https://doi.org/10.1007/978-3-030-44914-8_19

[69] Ramana Nagasamudram and David A. Naumann. 2021. Alignment completeness for relational hoare logics. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'21)*. IEEE, 1–13. https://doi.org/10.1109/LICS52264.2021.9470690 Extended version at https://arxiv.org/abs/2101.11730.

[70] 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

[71] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating state transition systems for fine-grained concurrent resources. In *Proceedings of the 23rd European Symposium on Programming (ESOP'14), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS'14) (Lecture Notes in Computer Science, Vol. 8410)*, Zhong Shao (Ed.). Springer, 290–310. https://doi.org/10.1007/978-3-642-54833-8_16

[72] David A. Naumann. 2006. From coupling relations to mated invariants for checking information flow. In *Proceedings of the 11th European Symposium on Research in Computer Security (ESORICS'06) (Lecture Notes in Computer Science, Vol. 4189)*, Dieter Gollmann, Jan Meier, and Andrei Sabelfeld (Eds.). Springer, 279–296. https://doi.org/10.1007/11863908_18

[73] David A. Naumann. 2007. Observational purity and encapsulation. *Theoret. Comput. Sci.* 376, 3 (2007), 205–224. https://doi.org/10.1016/j.tcs.2007.02.004

[74] David A. Naumann. 2020. Thirty-seven years of relational hoare logic: Remarks on its principles and history. In *Proceedings of the 9th International Symposium on Leveraging Applications of Formal Methods (ISoLA'20) (Lecture*

Notes in Computer Science, Vol. 12477), Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 93–116. https://doi.org/10.1007/978-3-030-61470-6_7

[75] Mohammad Nikouei. 2019. *A Logical Analysis of Relational Program Correctness*. Ph. D. Dissertation. Stevens Institute of Technology.

[76] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic (CSL'01) (Lecture Notes in Computer Science, Vol. 2142)*, Laurent Fribourg (Ed.). Springer, 1–19. https://doi.org/10.1007/3-540-44802-0_1

[77] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. 2009. Separation and information hiding. *ACM Trans. Program. Lang. Syst.* 31, 3 (2009), 1–50. https://doi.org/10.1145/964001.964024

[78] Susan S. Owicki and David Gries. 1976. An axiomatic proof technique for parallel programs I. *Acta Informatica* 6 (1976), 319–340. https://doi.org/10.1007/BF00268134

[79] Lauren Pick, Grigory Fedyukovich, and Aarti Gupta. 2018. Exploiting synchrony and symmetry in relational verification. In *Proceedings of the 30th International Conference on Computer Aided Verification (CAV'18), Held as Part of the Federated Logic Conference (FloC'18) (Lecture Notes in Computer Science, Vol. 10981)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 164–182. https://doi.org/10.1007/978-3-319-96145-3_9

[80] Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2013. Automating separation logic using SMT. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV'13) (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 773–789. https://doi.org/10.1007/978-3-642-39799-8_54

[81] Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014. GRASShopper—Complete heap verification with mixed specifications. In *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS'14) (Lecture Notes in Computer Science, Vol. 8413)*, Erika Ábrahám and Klaus Havelund (Eds.). Springer, 124–139. https://doi.org/10.1007/978-3-642-54862-8_9

[82] François Pottier. 2008. Hiding local state in direct style: A higher-order anti-frame rule. In *Proceedings of the 23rd Annual IEEE Symposium on Logic in Computer Science (LICS'08)*. IEEE Computer Society, 331–340. https://doi.org/10.1109/LICS.2008.16

[83] Ivan Radicek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. 2018. Monadic refinements for relational cost analysis. *Proc. ACM Program. Lang.* 2 (2018), 36:1–36:32. https://doi.org/10.1145/3158124

[84] John C. Reynolds. 1983. Types, abstraction and parametric polymorphism. In *Proceedings of the IFIP 9th World Computer Congress on Information Processing*, R. E. A. Mason (Ed.). North-Holland/IFIP, 513–523.

[85] Martin Rinard. 1999. *Credible Compilation*. Technical Report MIT-LCS-TR-776. MIT. Retrieved from https://people.csail.mit.edu/rinard/paper/credibleCompilation.html.

[86] Martin Rinard and Darko Marinov. 1999. Credible compilation with pointers. In *Proceedings of the FLoC Workshop on Run-Time Result Verification*. 20 pages. Retrieved from https://people.csail.mit.edu/rinard/paper/credibleCompilation.html.

[87] Stan Rosenberg, Anindya Banerjee, and David A. Naumann. 2012. Decision procedures for region logic. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'12) (Lecture Notes in Computer Science, Vol. 7148)*, Viktor Kuncak and Andrey Rybalchenko (Eds.). Springer, 379–395. https://doi.org/10.1007/978-3-642-27940-9_25

[88] Robert Sedgewick and Kevin Wayne. 2011. *Algorithms*, 4th ed. Addison-Wesley.

[89] Ron Shemer, Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. 2019. Property directed self composition. In *Proceedings of the 31st International Conference on Computer Aided Verification (CAV'19) (Lecture Notes in Computer Science, Vol. 11561)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 161–179. https://doi.org/10.1007/978-3-030-25540-4_9

[90] Jan Smans, Bart Jacobs, and Frank Piessens. 2009. Implicit dynamic frames: Combining dynamic frames and separation logic. In *Proceedings of the 23rd European Conference on Object-oriented Programming (ECOOP'09) (Lecture Notes in Computer Science, Vol. 5653)*, Sophia Drossopoulou (Ed.). Springer, 148–172. https://doi.org/10.1007/978-3-642-03013-0_8

[91] Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. 2010. Automatic verification of Java programs with dynamic frames. *Formal Aspects Comput* 22, 3–4 (2010), 423–457. https://doi.org/10.1007/s00165-010-0148-1

[92] Kristina Sojakova and Patricia Johann. 2018. A general framework for relational parametricity. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'18)*, Anuj Dawar and Erich Grädel (Eds.). ACM, 869–878. https://doi.org/10.1145/3209108.3209141

[93] Marcelo Sousa and Isil Dillig. 2016. Cartesian hoare logic for verifying k-safety properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16)*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 57–69. https://doi.org/10.1145/2908080.2908092

[94] Marcelo Sousa, Isil Dillig, and Shuvendu K. Lahiri. 2018. Verified three-way program merge. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 165:1–165:29. https://doi.org/10.1145/3276535

[95] Christopher S. Strachey. 2000. Fundamental concepts in programming languages. *High. Order Symb. Comput.* 13, 1/2 (2000), 11–49. https://doi.org/10.1023/A:1010000313106

[96] Jacob Thamsborg, Lars Birkedal, and Hongseok Yang. 2012. Two for the price of one: Lifting separation logic assertions. *Log. Methods Comput. Sci.* 8, 3 (2012). https://doi.org/10.2168/LMCS-8(3:22)2012

[97] Hiroshi Unno, Tachio Terauchi, and Eric Koskinen. 2021. Constraint-based relational verification. In *Proceedings of the Conference on Computer Aided Verification (Lecture Notes in Computer Science, Vol. 12759)*. Springer, 742–766. https://doi.org/10.1007/978-3-030-81685-8_35

[98] Mark Allan Weiss. 2010. *Data Structures and Problem Solving Using Java*, 4th ed. Addison-Wesley.

[99] Tim Wood, Sophia Drossopoulou, Shuvendu K. Lahiri, and Susan Eisenbach. 2017. Modular verification of procedure equivalence in the presence of memory allocation. In *Proceedings of the 26th European Symposium on Programming (ESOP'17), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS'17) (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 937–963. https://doi.org/10.1007/978-3-662-54434-1_35

[100] Hongseok Yang. 2007. Relational separation logic. *Theoret. Comput. Sci.* 375, 1–3 (2007), 308–334. https://doi.org/10.1016/j.tcs.2006.12.036

[101] Anna Zaks and Amir Pnueli. 2008. CoVaC: Compiler validation by program analysis of the cross-product. In *Proceedings of the 15th International Symposium on Formal Methods (FM'08) (Lecture Notes in Computer Science, Vol. 5014)*, Jorge Cuéllar, T. S. E. Maibaum, and Kaisa Sere (Eds.). Springer, 35–51. https://doi.org/10.1007/978-3-540-68237-0_5

[102] Lenore D. Zuck, Amir Pnueli, Benjamin Goldberg, Clark W. Barrett, Yi Fang, and Ying Hu. 2005. Translation and run-time validation of loop transformations. *Formal Methods Syst. Des.* 27, 3 (2005), 335–360. https://doi.org/10.1007/s10703-005-3402-z

Received 13 October 2020; revised 28 March 2022; accepted 3 July 2022