Refinement Composition Logic

YOUNGJU SONG, MPI-SWS, Germany
DONGJAE LEE, Seoul National University, Korea

One successful approach to verifying programs is refinement, where one establishes that the implementation (*e.g.*, in C) behaves as specified in its mathematical specification. In this approach, the end result (a whole implementation refines a whole specification) is often established via *composing* multiple "small" refinements.

In this paper, we focus on the task of composing refinements. Our key observation is a novel correspondence between the task of composing refinements and the task of proving entailments in modern separation logic. This correspondence is useful. First, it unlocks tools and abstract constructs developed for separation logic, greatly streamlining the composition proof. Second, it uncovers a fundamentally new verification strategy. We address the key challenge in establishing the correspondence with a novel use of *angelic non-determinism*.

Guided by the correspondence, we develop **RCL (Refinement Composition Logic)**, a logic dedicated to composing refinements. All our results are formalized in Coq.

CCS Concepts: • **Theory of computation → Separation logic**; **Programming logic**; **Abstraction**; **Operational semantics**; **Algebraic semantics**; **Program specifications**; **Program verification**.

Additional Key Words and Phrases: separation logic, refinement framework, angelic nondeterminism, Coq

ACM Reference Format:
Youngju Song and Dongjae Lee. 2024. Refinement Composition Logic. *Proc. ACM Program. Lang.* 8, ICFP, Article 256 (August 2024), 29 pages. https://doi.org/10.1145/3674645

1 Introduction

One prominent approach in formal verification is refinement [Back and Wright 2012; Choi et al. 2017; Dijkstra 1972; Liang and Feng 2016; Wirth 1971], which is used to specify the behavior of a complex and efficient implementation with a much simpler mathematical formulation. Refinements (usually) have a rather simple definition—*e.g.*, a mere set inclusion saying that any observable trace of the implementation can also appear in the specification—and this makes refinement an excellent *end result* of the verification, providing results understandable to a wide audience.

Establishing such an end result is often non-trivial due to both the size of the system and the gap between the implementation and its abstract, mathematical specification: *e.g.*, a hypervisor written in C spans several thousand lines, and exposes low-level programming features of C (such as bitwise manipulations) [Li et al. 2021].

A common approach in establishing such an end result is by *decomposing* it down to multiple pieces of "small" refinements [Gu et al. 2015; Koenig and Shao 2020]. Specifically, the notion of refinement often comes with a calculus offering axioms like *horizontal* and *vertical* compositionality. And, with these, the end result is established in a *module-wise* (*i.e.*, each module is separately verified) and *gradual* (*i.e.*, such verification is accomplished by multiple stepwise refinements) manner.

Authors' Contact Information: Youngju Song, MPI-SWS, Saarland Informatics Campus, Germany, youngju@mpi-sws.org; Dongjae Lee, Seoul National University, Korea, dongjae.lee@sf.snu.ac.kr.

ACM 2475-1421/2024/8-ART256
https://doi.org/10.1145/3674645

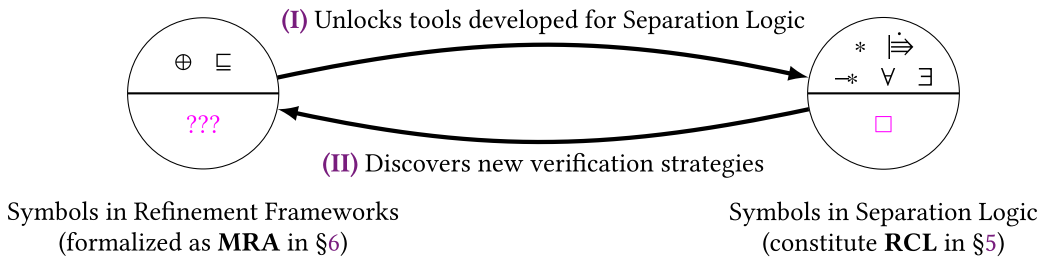
(I) Unlocks tools developed for Separation Logic
(II) Discovers new verification strategies
⊕ ⊑
???
∗ ⇛ −∗ ∀ ∃
□
Symbols in Refinement Frameworks (formalized as MRA in §6)
Symbols in Separation Logic (constitute RCL in §5)

Fig. 1. Contributions of this paper.

In this method, the verification task is twofold:

- Establishing these "small" refinements, and
- *Composing* them—using the axioms of calculus—to derive the end result.

In this paper, we focus on the latter and make two main contributions: **(I)** and **(II)**.

(I) First, we observe that the task of composing refinements could be understood in terms of the assertion language of modern *separation logic*. While there has been a plethora of work on utilizing separation logic for the former task (namely, relational separation logic) [Frumin et al. 2018, 2021; Gäher et al. 2022; Turon et al. 2013; Yang 2007], we are the first to use separation logic for the latter task, to the best of our knowledge. Based on the observation, we soundly lift these "small refinements" into *logical formulae*, and axioms for composing refinements into *logical rules* (of separation logic).

This lifting is practically useful in that it streamlines the composition proof. Most importantly, it unlocks the use of a mature tool [Krebbers et al. 2017] in separation logic to automate tedious steps in the composition proof. Moreover, the resulting logic enjoys a richer structure (*esp.*, having magic wands) than primitive calculus, adding more flexibility in composition proof. Finally, derived constructs developed in separation logic could further streamline the composition proof.

(II) Next, we take one step further and ask the question in reverse:

Can we interpret *all* symbols in separation logic in terms of refinement composition?

Specifically, in this paper, we focus on interpreting *all* symbols of a non-step-indexed version of Iris base logic. Iris base logic, which is at the core of the Iris program logic, suits our purpose well since it comprises just a few primitive symbols and not the programming-language-specific ones (like the points-to connective). Among these symbols in Iris base logic, the lifting in **(I)** already covers all the symbols *but* $\Box$, the *persistence modality* (we henceforth use magenta color for emphasis). Thus, the question is rephrased as:

Can we find an interpretation of $\Box$ in terms of refinement composition?

Unexpectedly, this question guided us to a novel observation that is not only useful for composing refinements but also enables *fundamentally new*—which were not possible before—strategies for formulating "small" refinements.

And here comes our second contribution. While all the existing refinement frameworks *as-is* do not admit persistence modality and its rules, (i) we find a minimal, natural, and elegant extension of the underlying *operational semantics* of existing refinement frameworks that allows us to define persistence modality in the world of refinement composition. Enabled by such an extension, (ii) we find a novel interpretation of persistence modality in refinement composition and thus establish a full correspondence between the task of composing refinements and the task of proving logical entailment in Iris base logic. Finally, using the persistence modality, (iii) we discover fundamentally new verification and composition strategies in refinement.

All in all, in this paper, we introduce **RCL (Refinement Composition Logic)**, a radically new approach to stating and composing refinements. RCL is not tied to a specific refinement framework; it could be instantiated with any notion of module and refinement provided that they satisfy the axioms of what we dubbed **MRA (Module and Refinement Algebra)**. These axioms comprise standard axioms in refinement frameworks and a few more to justify persistence modality.

The overall structure of our contribution is pictorially illustrated in Fig. 1, where the upper half concerns the contribution **(I)** and the lower half, **(II)**.

1.1 Contributions

We discover an unexpected correspondence between modern separation logic assertions and this exact problem of composing refinements. Guided by the correspondence, we develop RCL and present its usefulness. Specifically, this paper makes the following contributions.

- We give a novel interpretation of logical symbols (§3) and derived constructs (§7) of the Iris base logic in the refinement setting and show how it streamlines the proof of refinement composition.
- We show that existing module semantics in refinement frameworks can be naturally and elegantly extended using **angelic non-determinism** to satisfy the axioms of MRA (§4).
- We define MRA, which comprises completely standard axioms in refinement frameworks plus a new operator—revealed by the correspondence—called "core" ($|-|$) and its properties (§5 and §6).
- We derive RCL for any given MRA (§5). Doing so involves choosing a novel quotient set which might be independently useful in Iris (§6).
- We give one instance of MRA by extending an existing module system by Song et al. [2023] (§8).

The results in this paper are fully formalized in the Coq proof assistant [Song and Lee 2024].

2 Background

In this section, we cover some essential background on modules and refinements.

Module. The most basic element in refinement frameworks is a notion of *module*. Though the exact formulation differs by framework, a module is usually a set of function definitions given as a semantic object (*e.g.*, state transition system) that is not bound to a syntax of a specific programming language [Beringer et al. 2014; Song et al. 2019; Stewart et al. 2015]. Note that, in this setting, the state (*e.g.*, memory) appears explicit in the argument.

Modern refinement frameworks often come with an additional notion of ***private state*** (or, local state) [Gu et al. 2015; Koenig and Shao 2020; Sammler et al. 2023; Song et al. 2023]. In this setting, a module is a pair of initial value for the private state and a set of function definitions who can access its own private state but not others'. Such private states do not appear in the implementation modules written with programming languages; they appear only in the mathematical specifications for the purpose of giving a strong, comprehensible specification to the user.

Then, there is a linking operator between modules, $\oplus$, (taking the product of the private states and disjoint union of the functions) and a notion of refinement, $\sqsubseteq$, satisfying compositional properties shown in Fig. 2. The best way to understand refinement is by looking at a specific instance.

Refinement. *Contextual refinement* ($\sqsubseteq_{\text{ctx}}$) is perhaps the most well-known such refinement. It is based on the notion of *trace* and *behavior*, where a trace is a sequence of visible events and behavior is a set of traces. With this, contextual refinement between T and S is defined as follows:

$$T \sqsubseteq_{\text{ctx}} S \quad \triangleq \quad \forall Ctx.\ \text{Beh}(Ctx \oplus T) \subseteq \text{Beh}(Ctx \oplus S)$$

$\boxed{T_{\text{Fct}}}$

```
def f(x: ℕ): ℕ ≡
  if x then x * f(x - 1) else 1
```

$\boxed{T_{\text{Main}}}$

```
def main(): ℤ ≡ let a := f(3) in
  let b := f(4) in a + b
```

$\boxed{S_{\text{Fct}}}$

```
def f(x: ℕ): ℕ ≡ x!
```

$\boxed{S_{\text{Main}}}$

```
def main(): ℤ ≡ 30
```

$$\textsc{ref-refl}\ \frac{}{M \sqsubseteq M} \qquad \textsc{ref-vcomp}\ \frac{T \sqsubseteq M \qquad M \sqsubseteq S}{T \sqsubseteq S} \qquad \textsc{ref-hcomp}\ \frac{T_1 \sqsubseteq S_1 \qquad T_2 \sqsubseteq S_2}{T_1 \oplus T_2 \sqsubseteq S_1 \oplus S_2}$$

Fig. 2. An example showing refinement-based verification (above) and common axioms of refinements (below).

where Ctx is a context module and $\text{Beh}(M)$ is the behavior of a module M. This set inclusion ensures the following property,

$$bug \notin \text{Beh}(Ctx \oplus S_{\text{Hyp}}) \implies bug \notin \text{Beh}(Ctx \oplus T_{\text{Hyp}})$$

saying that: to guarantee the absence of bugs in the actual implementation, it suffices to check the source, which is exactly what clients expect. Also, it is easy to check that such a definition satisfies properties in Fig. 2 (assuming the identity element $\varnothing$ for $\oplus$).

Refinement is particularly suitable for stating the specification of *e.g.*, an operating system or a hypervisor [Gu et al. 2016]. The end result in a hypervisor verification, $T_{\text{Hyp}} \sqsubseteq_{\text{ctx}} S_{\text{Hyp}}$, says that the behavior of T_{Hyp} linked with *arbitrary* guest operating systems (Ctx) refines those of S_{Hyp}. Moreover, S_{Hyp} keeps all the critical data in its private state, not the shared memory; this in turn make it crystal clear that S_{Hyp} operates as expected even in the presence of malicious guests.

The Essence of Refinement-Based Program Verification. Fig. 2 presents, with a contrived example, the essence of refinement-based program verification. At the top, we have an implementation of the whole system comprising two modules, T_{Fct} and T_{Main} (T stands for "target"). A module T_{Fct} is a simple library that offers a single method, `f(x)`, which computes the factorial of `x` via recursion and returns the result. A module T_{Main} is a client that calls `f()` twice with arguments `3` and `4`, computes the sum of the results, and returns it.

In the middle of Fig. 2, we have the abstract specification of the system (again) comprising two modules, S_{Fct} and S_{Main} (S stands for "source" or "specification"). A specification module S_{Fct} simplifies the recursive computation in T_{Fct} to `x!`, using the factorial function in mathematics. S_{Main} replaces the return value by a constant `30`. Then, one could establish the end result (*i.e.*, $T_{\text{Fct}} \oplus T_{\text{Main}} \sqsubseteq S_{\text{Main}}$) by verifying and composing the following small (modular) refinements:

$$i:\ T_{\text{Fct}} \sqsubseteq S_{\text{Fct}} \qquad\qquad ii:\ S_{\text{Fct}} \oplus T_{\text{Main}} \sqsubseteq S_{\text{Fct}} \oplus S_{\text{Main}}$$

In *i*, one verifies that the library implementation meets its specification. In *ii*, one similarly verifies that the client implementation meets its specification, but this time *leveraging* the specification of the library: S_{Fct} appears explicit on the refinement. Thanks to the simplification made by *i*, verifying *ii* is straightforward. Finally, using the axioms in Fig. 2, we compose *i*-*ii* and get the end result:

$$T_{\text{Fct}} \oplus T_{\text{Main}} \sqsubseteq S_{\text{Fct}} \oplus T_{\text{Main}} \sqsubseteq S_{\text{Fct}} \oplus S_{\text{Main}}$$

where we first apply *i* (together with ref-hcomp and ref-refl), and then apply *ii*. These successive applications of refinements (*i.e.*, gradual verification) is allowed by ref-vcomp. These axioms we used are common to refinement frameworks, where ref-vcomp and ref-hcomp are often called *vertical* and *horizontal* compositionality, respectively.

Formulating small refinements is non-trivial. Before we move on, we answer a commonly asked question at this point. Some astute readers have wondered if we can make the composition of refinements completely straightforward by breaking down these small refinements further:

$$\frac{T_{\text{Fct}} \sqsubseteq \ldots \sqsubseteq S_{\text{Fct}} \qquad T_{\text{Main}} \sqsubseteq \ldots \sqsubseteq S_{\text{Main}}}{T_{\text{Fct}} \oplus T_{\text{Main}} \sqsubseteq S_{\text{Fct}} \oplus S_{\text{Main}}}$$

If such a strategy works, its composition is completely trivial and would obviate any need for an advanced calculus or logic like RCL for composing refinements.

However, it turns out that the above approach does *not* work! The issue is that $T_{\text{Main}} \sqsubseteq S_{\text{Main}}$ actually does *not* hold because such a statement would imply a false statement

$$\lightning_{\text{Fct}} \oplus T_{\text{Main}} \sqsubseteq \lightning_{\text{Fct}} \oplus S_{\text{Main}} \qquad \text{(via REF-HCOMP)}$$

where $\lightning_{\text{Fct}}$ is completely broken in that the body of `f()` always renders undefined behavior. Undefined behavior (shortened as **UB**) intuitively stands for an error and is formally defined as a set of all traces. Then, in the above refinement, the target renders **UB** (since T_{Main} calls `f()`), while the source does not (since S_{Main} does not call `f()`); hence, false.

3 A Tour of RCL

In this section, we give a brief tour to our main contributions: **(I)** in §3.1, and **(II)** in §3.2.

3.1 (I) Interpreting Refinement Composition with Symbols in Separation Logic

In this subsection, we will see what it means to understand refinement composition in terms of separation logic and how it could be useful in streamlining composition proof (discussed further in §7). To see this, consider a similar verification scenario with three modules where we have the following small refinements (*i-iii*) and the end result (*goal*):

$$\begin{array}{lrcl}
i: & T_A & \sqsubseteq & S_A \\
ii: & S_A \oplus T_B & \sqsubseteq & S_A \oplus S_B \qquad\qquad goal: \quad T_A \oplus T_B \oplus T_C \sqsubseteq S_A \oplus S_B \oplus S_C \\
iii: & S_A \oplus T_C & \sqsubseteq & S_A \oplus S_C
\end{array}$$

Composing the verification conditions *i-iii* to form the end result is conceptually straightforward, but actually comprises some tedious steps. To begin with, let us take a look at additional common axioms in refinement frameworks, shown at the top of Fig. 3. Here, REF-COMM and REF-ASSOC states commutativity and associativity, and REF-ASSOCR is derived from these two.

Then, the proof script for proving the *goal* in Coq proof assistant is shown on the left of the figure. It utilizes the setoid mechanism to automatically discharge many of the tedious steps, but there are still a few.

In this proof, lines `L1`-`L3` apply the refinements *i-iii* each. After `L2`, our business with S_B is over, and we would like to remove it from both the left-hand side and the right-hand side of the goal. While conceptually straightforward, it requires multiple tedious applications of various axioms. Here, we use REF-HCOMP (along with REF-REFL) to remove S_B, but for this, we need to use a series of REF-COMM, REF-ASSOC, and REF-ASSOCR before and after to align the modules properly. What is worse is that we often need to manually specify where to apply such axioms (the magenta colored part).

Basics of RCL. In RCL, we have a much more streamlined proof shown on the right of Fig. 3. This is built on top of the following key observation in RCL. We lift the notion of refinements into *logical formulae* and the primitive axioms into *logical rules* of separation logic assertion language.

To begin with, the statement $T \sqsubseteq S$ is expressed in RCL as follows:

$$\boxed{T} \vdash |\dot{\Rrightarrow} \boxed{S}$$

REF-COMM
$M_1 \oplus M_2 \sqsubseteq M_2 \oplus M_1$

REF-ASSOC
$M_1 \oplus (M_2 \oplus M_3) \sqsubseteq M_1 \oplus M_2 \oplus M_3$

REF-ASSOCR
$M_1 \oplus M_2 \oplus M_3 \sqsubseteq M_1 \oplus (M_2 \oplus M_3)$

```
(* $T_A \oplus T_B \oplus T_C \sqsubseteq S_A \oplus S_B \oplus S_C$ *)
L1: rewrite i.
(* $S_A \oplus T_B \oplus T_C \sqsubseteq S_A \oplus S_B \oplus S_C$ *)
L2: rewrite ii.
(* $S_A \oplus S_B \oplus T_C \sqsubseteq S_A \oplus S_B \oplus S_C$ *)
    rewrite REF-COMM with ($M_2$ := $T_C$).
    rewrite <- REF-COMM with ($M_1$ := $S_C$).
    rewrite REF-ASSOC. rewrite <- REF-ASSOCR.
    apply REF-HCOMP; [|apply REF-REFL].
(* $T_C \oplus S_A \sqsubseteq S_C \oplus S_A$ *)
L3: rewrite REF-COMM with ($M_2$ := $S_A$).
    rewrite <- REF-COMM with ($M_2$ := $S_C$).
(* $S_A \oplus T_C \sqsubseteq S_A \oplus S_C$ *)
    rewrite iii. apply REF-REFL.
```

```
(* $\boxed{T_A} * \boxed{T_B} * \boxed{T_C} \vdash \Rrightarrow(\boxed{S_A} * \boxed{S_B} * \boxed{S_C})$ *)
R1:  iDestruct (i with [$]) as >?.
(* $\boxed{S_A} * \boxed{T_B} * \boxed{T_C} \vdash \Rrightarrow(\boxed{S_A} * \boxed{S_B} * \boxed{S_C})$ *)
R2:  iDestruct (ii with [$]) as >[? $].

(* $\boxed{T_C} * \boxed{S_A} \vdash \Rrightarrow(\boxed{S_A} * \boxed{S_C})$ *)
R3:  iDestruct (iii with [$]) as $.
```

Fig. 3. Additional common axioms for shuffling modules (top), usual composition proof with these axioms (left), and the same proof in RCL using IPM (right).

This logical formula uses three symbols, which we only give some intuition here (the full definitions and rules will be given in §5). First, $\boxed{T}$ specifies a singleton set where the only element is the module T. Second, the logical entailment $P \vdash Q$ means that P is a subset of Q. Finally, the *refines* modality $\Rrightarrow P$ specifies a set of modules that refine the module specified by P. With these intuitions, $\boxed{T} \vdash \Rrightarrow \boxed{S}$ is translated into $\{T\} \subseteq \{M \mid M \sqsubseteq S\}$, which in turn means $T \sqsubseteq S$.

Next, we define the separating conjunction $(*)$ to satisfy $\boxed{T_1} * \boxed{T_2} \equiv \boxed{T_1 \oplus T_2}$; lifting the notion of module linking $(\oplus)$ into the logical world. Separating conjunction is commutative and associative, meaning that $A * B \equiv B * A$ and $A * (B * C) \equiv A * B * C$.

With these, the original "small" refinements and the end result is rephrased as follows:

$$\begin{aligned} i &: \boxed{T_A} \vdash \Rrightarrow \boxed{S_A} \\ ii &: \boxed{S_A} * \boxed{T_B} \vdash \Rrightarrow(\boxed{S_A} * \boxed{S_B}) \\ iii &: \boxed{S_A} * \boxed{T_C} \vdash \Rrightarrow(\boxed{S_A} * \boxed{S_C}) \end{aligned} \qquad goal : \boxed{T_A} * \boxed{T_B} * \boxed{T_C} \vdash \Rrightarrow(\boxed{S_A} * \boxed{S_B} * \boxed{S_C})$$

Same proof in RCL. Now we explain the proof script on the right in Fig. 3. The proof here utilizes an existing well-polished tool called Iris Proof Mode (IPM) [Krebbers et al. 2017] for proving logical entailments in separation logic. Among many other features, IPM excels at automatic applications of the *frame rule* and commutativity/associativity.

As before, the lines R1-R3 apply the refinements *i*-*iii* each, and the comments show the current goal at each point. While the proof *script* may seem cryptic—and we do not intend to explain them in detail in this paper—the way the proof *goal* changes should be easy to follow.

The point here is twofold.

First, IPM's automatic applications of *frame rule* replace manual applications of REF-HCOMP (along with REF-REFL). Note the use of `[? $]` at the end of R2. These symbols indicate whether each conjunct of the right-hand-side of *ii* should be *framed away* or remain in the proof: `?` means it should remain, and `$` means it should be framed away. Therefore, in R2, S_A remains in the proof and S_B is framed away. IPM automatically applies all the needed REF-HCOMP and REF-REFL (which could be multiple in general) to achieve this action.

Second, IPM's automatic applications of commutativity and associativity completely obviate the need for manual applications of REF-COMM, REF-ASSOC, and REF-ASSOCR. To see this, take a look at R3.

$\boxed{T_{\text{Suc}}}$
```
def succ(x: ℤ): ℤ ≡
  x + 1
```

$\boxed{T_{\text{Put}}}$
```
def put(x: ℤ): ℤ ≡
  print(x); x
```

$\boxed{T_{\text{Rpt}}}$
```
def rpt(p: ℙ,n: ℕ,x: ℤ): ℤ ≡
  match n with
  | 0 ⇒ x
  | S m ⇒
    let y := *p(x) in
    rpt(p, m, y)
```

$\boxed{S_{\text{Rpt}}\ fn\ f}$
```
def rpt(p: ℙ,n: ℕ,x: ℤ): ℤ ≡
  assume(p == fn);
  (fⁿ x)
```

$\boxed{T_{\text{Main}}}$
```
def main(): ℤ ≡
  let a := rpt(succ,1,1) in
  rpt(put,a,a); 0
```

$\boxed{M_{\text{Main}}}$
```
def main(): ℤ ≡
  rpt(put,2,2); 0
```

$\boxed{S_{\text{Main}}}$
```
def main(): ℤ ≡
  print(2); print(2); 0
```

Fig. 4. An example using function pointers.

After applying *ii* in R2, the order of modules in the premise is shuffled (T_C appears before S_A). In L3, we needed to apply REF-COMM and align the modules before we apply *iii*. In R3, we directly apply *iii*: IPM automatically inserts all the necessary REF-COMM, REF-ASSOC, and REF-ASSOCR to align the goal.

To wrap up, in this subsection, we have seen how we interpret refinements in terms of separation logic formulae and why it is beneficial (unlocking the use of tools developed for separation logic).

3.2 (II) New Verification Strategies Revealed by the Correspondence

In this subsection, we will investigate a verification strategy newly envisioned by the correspondence. To see this, consider an example shown in Fig. 4. The module T_{Rpt} has a single function `rpt`, which is a well-known "repeat" function: it takes a function pointer `p`, a natural number `n`, and an integer `x` as an argument, and applies the function pointed to by `p` to the argument `x` for `n` times. For example, T_{Rpt} computes $x + n$ when the argument `p` is `succ`. Note that, following recent literature [Gäher et al. 2022; Wang et al. 2022], we use string literals as function pointers.

The specification of T_{Rpt} could be given as $S_{\text{Rpt}}\ fn\ f$. Note that this module is parameterized by a function name fn and a body f: the spec $S_{\text{Rpt}}\ fn\ f$ is intended to serve only the function pointer for fn. The body of $S_{\text{Rpt}}\ fn\ f$ first checks, with the **assume** instruction, if the argument `p` is a function pointer for fn. If the condition is not met, it directly renders **UB**, which means that the caller is calling `rpt` in an unintended way. If the condition is met, **assume** does nothing and proceed with the rest of the body, *i.e.*, returning f^n `x`. The operator $(-)^n$ is a semantic iterator, which simplifies the manual recursive call in T_{Rpt}.

With this specification, similar to *ii* above, the verification of Rpt would be as follows:

$$fn \mapsto f \oplus T_{\text{Rpt}} \sqsubseteq fn \mapsto f \oplus S_{\text{Rpt}}\ fn\ f \sqsubseteq S_{\text{Rpt}}\ fn\ f \qquad (\text{VERIF-RPT})$$

for any given fn and f. Here, $fn \mapsto f$ denotes a module with only one function with name fn and body f. The first refinement holds because: if `p` is equal to fn, both sides essentially carry out the same computation, and if `p` is not equal to fn, the source renders **UB**. Next, in this example, we will never be using the module $fn \mapsto f$ again, so we drop it for simplicity in the second refinement. The second refinement holds trivially because when a function fn is invoked, the source may render **UB**—because the definition is missing—while the target does not (this is discussed further in §4).

On one hand, this result is good: module Rpt is verified modularly (independent of the client) and it can be used by any client by instantiating fn and f as they wish. On the other hand, this result is bad: such a refinement could not serve multiple clients *at the same time*.

To see this clearly, consider a Main module given at the bottom of Fig. 4 where we wish to establish the following whole-program refinement as an end result:

$$(T_{\text{Suc}} \oplus T_{\text{Put}}) \oplus T_{\text{Rpt}} \oplus T_{\text{Main}} \sqsubseteq S_{\text{Main}}$$

Since the set of observable behaviors of both sides exactly coincide, this refinement holds. However, establishing such a result in a modular and gradual way, using VERIF-RPT, is tricky.

$$(T_{\text{Suc}} \oplus T_{\text{Put}}) \oplus T_{\text{Rpt}} \oplus T_{\text{Main}} \sqsubseteq T_{\text{Put}} \oplus (S_{\text{Rpt}}\ \texttt{succ}\ \_) \oplus T_{\text{Main}} \sqsubseteq T_{\text{Put}} \oplus (S_{\text{Rpt}}\ \texttt{succ}\ \_) \oplus M_{\text{Main}} \not\sqsubseteq S_{\text{Main}} \quad (\oslash)$$

We could first apply VERIF-RPT with succ and its body (abbreviated as _). Then, using this, we can easily abstract T_{Main} into M_{Main} where M_{Main} has its first call to rpt inlined. However, we have arrived at a dead end now: the resulting program cannot refine our end goal anymore! The reason is simple: executing $T_{\text{Put}} \oplus (S_{\text{Rpt}}\ \texttt{succ}\ \_) \oplus M_{\text{Main}}$ results in **UB**. Note that $(S_{\text{Rpt}}\ \texttt{succ}\ \_)$ serves *only* the single client, succ, and if it gets called with a function pointer other than succ it triggers **UB**.

This illustrates why it is challenging to verify Fig. 4 in a *modular* way using the basic notions of refinement. While it would be possible to verify Fig. 4 if we compromise some modularity—*i.e.*, we could verify *globally* as follows: $(T_{\text{Suc}} \oplus T_{\text{Put}}) \oplus T_{\text{Rpt}} \sqsubseteq S_{\text{RptSucPut}}$ where $S_{\text{RptSucPut}}$ serves all the clients—this is clearly less ideal.

The new component: persistence modality. Our solution to this problem is revealed in the effort to establish the full correspondence between the Iris base logic and the refinements. In particular, $\square$ **(persistence) modality**, which finds a novel interpretation in the refinement setting in this paper, is the key to our solution.

In Iris base logic, the rules for $\square$ are as follows. We have $\square P \vdash P$, meaning that $\square P$ behaves like P, but it has an additional property. That is, we have $\square P \vdash \square P * \square P$, meaning that $\square P$ is *duplicable*. Finally, as usual in all modalities, $\square$ is monotone: *i.e.*, $P \vdash Q \implies \square P \vdash \square Q$ holds.

In RCL, we adopt $\square$ modality with exactly the same structural rules. For expository purposes, we postpone its *semantics* in RCL and why such semantics justifies the above rules in the refinement setting until §4. Here, we proceed assuming that $\boxed{T_{\text{Rpt}}} \vdash \square \boxed{T_{\text{Rpt}}}$ holds.

Repairing composition proof in RCL. Now we see how we verify Fig. 4 modularly in RCL. The basic strategy is the same as before ($\oslash$), but this time, we utilize $\square$ modality to fix the proof. We begin with enhancing VERIF-RPT as follows:

$$\boxed{T_{\text{Rpt}}} \vdash \square\, \mathit{REF} \qquad\qquad \mathit{REF} \triangleq (\forall \mathit{fn}, f.\ \boxed{\mathit{fn} \mapsto f} \mathrel{-\!\!*} |\!\!\Rrightarrow \boxed{S_{\text{Rpt}}\ \mathit{fn}\ f}) \qquad (\text{VERIF-RPT-RCL})$$

The symbols here ($\forall$, $-\!\!*$) obey the same rule as in standard separation logic. To get VERIF-RPT-RCL, we begin with lifting VERIF-RPT into the logical form: $\boxed{T_{\text{Rpt}}} \vdash \mathit{REF}$. Then, assuming $\boxed{T_{\text{Rpt}}} \vdash \square \boxed{T_{\text{Rpt}}}$, VERIF-RPT-RCL follows directly from the monotonicity of $\square$ modality: $\boxed{T_{\text{Rpt}}} \vdash \square \boxed{T_{\text{Rpt}}} \vdash \square\, \mathit{REF}$.

Now, we prove the following four "small refinements". We verify each client *modularly* as follows:

$$\boxed{T_{\text{Suc}}} * \square\, \mathit{REF} \vdash \boxed{T_{\text{Suc}}} * \mathit{REF} \vdash |\!\!\dot{\Rrightarrow} \boxed{S_{\text{Rpt}}\ \texttt{succ}\ \_} \qquad\qquad \boxed{T_{\text{Put}}} * \square\, \mathit{REF} \vdash \boxed{T_{\text{Put}}} * \mathit{REF} \vdash |\!\!\dot{\Rrightarrow} \boxed{S_{\text{Rpt}}\ \texttt{put}\ \_}$$

Also, for the Main module, we prove the following two gradual refinements:

$$\boxed{S_{\text{Rpt}}\ \texttt{succ}\ \_} * \boxed{T_{\text{Main}}} \vdash |\!\!\dot{\Rrightarrow} \boxed{M_{\text{Main}}} \qquad\qquad \boxed{S_{\text{Rpt}}\ \texttt{put}\ \_} * \boxed{M_{\text{Main}}} \vdash |\!\!\dot{\Rrightarrow} \boxed{S_{\text{Main}}}$$

Recall that $\boxed{T} \vdash |\!\!\dot{\Rrightarrow} \boxed{S}$ equals $T \sqsubseteq S$. It is easy to check that these corresponding refinements hold.

Finally, we compose all the entailments (VERIF-RPT-RCL and four "small refinements") and get the end result as follows (we omit some intermediate steps automatically handled by IPM, especially the elimination of $\dot{\Rrightarrow}$ modalities; the rules will be articulated in §5):

$$
\begin{aligned}
&\boxed{T_{\text{Suc}}} * \boxed{T_{\text{Put}}} * \boxed{T_{\text{Rpt}}} * \boxed{T_{\text{Main}}} \ \vdash\ \boxed{T_{\text{Suc}}} * \boxed{T_{\text{Put}}} * \square\, \mathit{REF} * \boxed{T_{\text{Main}}} \\
&\vdash\ (\boxed{T_{\text{Suc}}} * \square\, \mathit{REF}) * (\boxed{T_{\text{Put}}} * \square\, \mathit{REF}) * \boxed{T_{\text{Main}}} && (1)\\
&\vdash\ \dot{\Rrightarrow}\boxed{S_{\text{Rpt}}\ \texttt{succ}\ \_} * \dot{\Rrightarrow}\boxed{S_{\text{Rpt}}\ \texttt{put}\ \_} * \boxed{T_{\text{Main}}} && (2)\\
&\vdash\ \dot{\Rrightarrow}\boxed{S_{\text{Rpt}}\ \texttt{put}\ \_} * \dot{\Rrightarrow}\boxed{M_{\text{Main}}} \ \vdash\ \dot{\Rrightarrow}\boxed{S_{\text{Main}}} && (3)
\end{aligned}
$$

We begin the proof by applying VERIF-RPT-RCL. In (1), we *duplicate* $\square\, \mathit{REF}$ and distribute it to two clients. In (2), we apply two modular refinements for two clients. In (3), we apply two gradual refinements for the main module, and get the end result.

Contrast this to the dead end we arrived before (⊘). Before, $\boxed{T_{\text{Rpt}}}$ served only one client because the application of VERIF-RPT changed the definition of `rpt` which affected every client. Here, $\boxed{T_{\text{Rpt}}}$ effectively serves both clients thanks to the *duplicability* of the persistence modality.

We can understand this in two ways. In terms of *logic*, the fundamentally new thing that RCL brings to the table is the persistence modality. In terms of *refinement*, this makes a natural extension from a *definition-wise* refinement into a *call-wise* refinement. The granularity of refinement ($\sqsubseteq$) is *definition-wise* and it was the root of the failure in ⊘ (refining `rpt` to a specification affects *all* calls to `rpt`). On the contrary, the proof above essentially does a *call-wise* refinement in the sense that the two calls to `rpt` (in `main`) are refined to different specifications using different refinement results.

To our best knowledge, all existing frameworks have a fundamental issue in supporting it; *i.e.*, the verification above is fundamentally new. The issue is that the use of persistence modality results in what we call a ***polysemic*** program, by which we mean programs containing multiple different definitions for a single function name. Indeed, under the hood, the above verification unfolds to the following sequence of refinements (with 2 in the middle):

$$T_{\text{Suc}} + T_{\text{Put}} + T_{\text{Rpt}} + T_{\text{Main}} \ \sqsubseteq\ \ldots\ \sqsubseteq\ (S_{\text{Rpt}}\ \texttt{succ}\ \_) + (S_{\text{Rpt}}\ \texttt{put}\ \_) + T_{\text{Main}} \ \sqsubseteq\ \ldots\ \sqsubseteq\ S_{\text{Main}}$$

Here, we have a polysemic program in the middle—note that `rpt` is defined twice in $(S_{\text{Rpt}}\ \texttt{succ}\ \_) + (S_{\text{Rpt}}\ \texttt{put}\ \_)$—whereas the left (target) and the right (source) programs are *monosemic* (*i.e.*, not polysemic). In order for this refinement to hold, it is essential to give proper semantics to polysemic programs, and this is the key challenge in this work. Note that we cannot simply give trivial semantics to polysemic programs, like top (undefined behavior) or bottom (empty behavior) because that must break either the right or the left refinement.

To wrap up, we have seen how the use of persistence modality allows a new verification strategy with Fig. 4. In the next section (§4), we will see the key challenge and our key idea to tackle it.

3.3 Position of This Work

Since RCL spans both refinement and separation logic assertions in a new way, let us position RCL within these contexts to avoid potential confusion.

Relation with separation logic. While we are borrowing symbols and rules from Iris base logic, in RCL we give a completely different interpretation: a logical proposition means *a set of modules* instead of *a set of program states* as in its original use as a program logic.

Note that RCL is not meant to replace or compete with separation-logic-as-a-program-logic; rather, our intention is to use them *in conjunction*. In RCL, we only consider *composing* these small refinements and actually establishing such a small refinements (like $T_{\text{Fct}} \sqsubseteq S_{\text{Fct}}$) is left to the user. Relational program logics [Frumin et al. 2018, 2021; Gäher et al. 2022; Turon et al. 2013] are powerful techniques for *establishing* such small refinements. Thus, relational program logics and RCL complement each other.

Relation with refinements. As mentioned earlier, RCL is not tied to a specific refinement framework; it could be instantiated with any notion of module and refinement provided that they satisfy the axioms of MRA. These axioms comprise standard axioms we have seen so far and a few more to justify persistence modality. While these latter axioms are not satisfied in existing refinement frameworks as-is, we find a minimal and natural extension that validates the latter.

4 Extending Operational Semantics for Persistence Modality

In this section, we first discuss the meaning of persistence modality in the world of refinement and modules (§4.1). Then, we see why this interpretation is sound with respect to the rules for $\square$ (§4.2). Finally, we articulate the key challenge and the key idea on formalizing $\square$ (§4.3).

By the end of this section, it should be clear why $\boxed{T_{\text{Rpt}}} \vdash \square \boxed{T_{\text{Rpt}}}$—which we reserved explanation for in §3.2—holds.

4.1 Interpretation of Persistence Modality ($\square$) in the World of Refinement

We begin with the following question: what kind of module could possibly be duplicated? We encourage readers to ponder this before continuing.

The answer is: modules without private state (*stateless* modules for short). In fact, statelessness almost precisely[1] captures duplicability, and it is intuitively understandable as follows (will further be clarified in §4.3). First, a stateless module (which may well contain other computations like let-bindings, function calls, or system calls) should be okay to duplicate; each copy will behave the same way regardless of how many copies there are. Second, statefulness easily breaks duplicability.

Example 1. *Consider the following stateful module (S_{Ctr}). If it is duplicable, there is a contradiction.*

$\boxed{S_{Ctr}}$	$\boxed{T_{ClntX}}$	$\boxed{S_{ClntX}}$
`private v: ℤ := 0`		`private v: ℤ := 0`
	`def fooX(): 1 ≡`	`def fooX(): 1 ≡ v := v + 1; print(v)`
`def incr(): ℤ ≡ v := v + 1; v`	`  print(incr())`	`def incr() ≡ triggerUB`

Proof. Here, S_{Ctr} is a simple counter module with a single method `incr` that increments the module-private counter `v` and returns its value. Also, there are *two* identical clients of this counter module, T_{Clnt1} and T_{Clnt2} (we are using $\mathcal{X}$ as a macro for 1 and 2), each having a single method `foo1` and `foo2` which simply calls `incr` and prints the result.

Now, if S_{Ctr} is duplicable, we can prove:

$$S_{\text{Ctr}} \oplus T_{\text{Clnt1}} \oplus T_{\text{Clnt2}} \sqsubseteq (S_{\text{Ctr}} \oplus T_{\text{Clnt1}}) \oplus (S_{\text{Ctr}} \oplus T_{\text{Clnt2}}) \sqsubseteq S_{\text{Clnt1}} \oplus S_{\text{Clnt2}}$$

where we first duplicate S_{Ctr}, distribute it to the two clients, and then do the usual refinement (*i.e.*, inlining the call to `incr`) to derive specifications for each clients, S_{Clnt1} and S_{Clnt2}. However, such a refinement should *not* hold: the target and the source have different behaviors. `def main() ≡ foo1(); foo2()` prints 1 and 2 in the target ($S_{\text{Ctr}} \oplus T_{\text{Clnt1}} \oplus T_{\text{Clnt2}}$) whereas the same program

[1]In theory, modules that are "effectively" stateless—modules that have state but for which there exists a semantically equivalent version that is stateless—could also be duplicable. Such modules would include a module that only reads from the state or a module that uses the state just as a cache for the computation. Since these modules can be refined to a "syntactically" stateless version, we focus only on the "syntactically" stateless modules in this paper.

prints 1 twice in the source ($S_{\text{Clnt1}} \oplus S_{\text{Clnt2}}$). The reason is that the state of the module S_{Ctr} has a single copy in the target, whereas in the source, there are two. ■

Similarly to Example 1, statelessness properly captures the ability to do call-wise refinement.

Example 2. *Consider the following stateful module (T_{Ctr}) and its refinement ($T_{Ctr} \sqsubseteq S_{Ctr}$). If such a refinement could be applied call-wise on the client side, there is a contradiction.*

$\boxed{T_{Ctr}}$
```
private v: ℤ := 0
def incr(): ℤ ≡ v := v-1; -v
```

$\boxed{S_{Ctr}}$
```
private v: ℤ := 0
def incr(): ℤ ≡ v := v+1; v
```

$\boxed{T_{Main}}$
```
def main(): 1 ≡
  print(incr()); print(incr())
```

Proof. Here, executing an implementation $T_{\text{Ctr}} \oplus T_{\text{Main}}$ prints 1 and 2 (note that T_{Ctr} decrements the counter and then returns `-v`, not `v`). Now, since $T_{\text{Ctr}} \sqsubseteq T_{\text{Ctr}}$ and $T_{\text{Ctr}} \sqsubseteq S_{\text{Ctr}}$ holds, suppose that we did a call-wise refinement and reached an imaginary program where Main first calls T_{Ctr} and then S_{Ctr}. Then, such a program prints 1 twice, which is wrong. ■

As a final remark, statelessness is not overly restrictive. As mentioned in §2, the implementation modules are always stateless: they only operate on global state—memory—which either appears explicit in the argument or appears as a primitive module [Song et al. 2023]. And, as shown in the verification of our running example (Fig. 4), we only need implementations ($\boxed{T_{\text{Rpt}}}$) to be stateless.

4.2 Rules Related with Persistence Modality (□).

With the understanding of statelessness, we can now (intuitively) interpret $\Box P$ as P in conjunction with the fact that modules specified by P are stateless. Now, let us examine the rules for this persistence—or, *stateless*—modality. Relevant rules are shown at Fig. 5. We have already seen the first three rules (PERS-MONO, PERS-DUP, and PERS-E) in §3.2. PERS-MONO is self-explanatory. PERS-DUP holds since the modules specified in P are all stateless, and thus duplicable. PERS-E simply forgets the fact about statelessness.

The rule CORE-PERS says that a *core* of any given module ($|a|$) is stateless: core operator *extracts* the stateless computations from a given module by simply interpreting all state accesses as **UB**. Therefore, if a module b is already stateless, $|b|$ results in the same module as b. Now, since T_{Rpt} is stateless, we have $|T_{\text{Rpt}}| \equiv T_{\text{Rpt}}$. This, together with CORE-PERS, implies the desired $\boxed{T_{\text{Rpt}}} \vdash \Box \boxed{T_{\text{Rpt}}}$.

Affinity. Now we see the final rule: AFFINE. The rule is very simple, but its consequence is significant: it makes RCL an *affine* logic, meaning that one can freely drop *any* proposition in the premise. That is, the following holds:

$$P * Q \vdash P * \mathsf{True} \vdash P \qquad \text{(WEAKENING)}$$

This is really useful since we want to mindlessly duplicate propositions via PERS-DUP, and we do not want to keep excess ones in our final result. Moreover, we often want to just drop modules that are not used anymore, just as we did in VERIF-RPT.

However, if we see the interpretation of WEAKENING in refinements, it is rather exotic. On the one hand, such a rule makes sense: if one could prove $a \sqsubseteq c$, then $a \oplus b \sqsubseteq c$ would also hold because one can utilize more modules (b) in establishing the latter refinement. On the other hand, this also entails the following exotic result: $(\texttt{foo} \mapsto \lambda\_.\, 1) \oplus (\texttt{foo} \mapsto \lambda\_.\, 2) \sqsubseteq (\texttt{foo} \mapsto \lambda\_.\, 1)$.

In fact, this rule and PERS-DUP both are closely related to the ***key challenge*** in this paper: applications of these rules easily result in polysemic programs, and the polysemic semantics we give must justify these rules.

PERS-MONO
$$\frac{P \vdash Q}{\square P \vdash \square Q}$$

PERS-DUP
$$\square P \vdash \square P * \square P$$

PERS-E
$$\square P \vdash P$$

CORE-PERS
$$\boxed{a} \vdash \square \boxed{|a|}$$

AFFINE
$$P * \mathsf{True} \dashv\vdash P$$

Fig. 5. Selected rules related with $\square$.

In the next subsection, we will see how we give operational semantics to these polysemic programs in a way that is consistent with the desired rules.

4.3 Operational Semantics for Polysemic Programs

As mentioned in §1, refinement frameworks *as-is* do not admit persistence modality. The problem is that duplicating modules results in a polysemic program.

In existing refinement frameworks, a polysemic program is considered invalid (often syntactically), and thus, its operational semantics is not properly given. This is a sensible design: such a polysemic program is considered syntactically invalid in programming languages (*e.g.*, C).

In RCL, such polysemic programs would very well appear as a result of PERS-DUP (where each copy can thereafter be refined to different definitions, *e.g.*, as in 2). Note, however, that such polysemic programs appear only in the *intermediate* proof states, and do not appear in the *end result* (recall the end result in §3.2).

Goal of the subsection. In this subsection, we demonstrate how the operational semantics of a polysemic program can be *adequately* defined by ***extending*** existing operational semantics in a simple and elegant way. By extending, we mean that we are only giving semantics to programs that have been considered invalid (polysemic programs), and the semantics of valid programs will remain exactly the same. Thus, this extension does *not* affect the end result.

The extension we are making is *adequate*, in the sense that it validates all the rules in Fig. 5. As we will see in §6, this boils down to the following two axioms in MRA:

$$\forall a.\ a \sqsubseteq |a| \oplus a \qquad\qquad \forall a.\ a \sqsubseteq \varnothing$$

where $\varnothing$ is the unit of the linking operator $\oplus$. The former axiom is meant to justify PERS-DUP, and the latter the AFFINE. Validating these axioms is precisely our goal here.

Background: operational semantics and behavior of a program. Before we proceed, we quickly cover some relevant background to avoid confusion. As we have seen in §2, we denote the behavior (which is a set of traces) of a module with $\mathrm{Beh}(-) \in M \to \mathcal{P}(\mathit{Trace})$ where M is the type of module. Note well that the module linker $\oplus \in M \to M \to M$ links the *modules*, not the set of traces. Then, a behavior of a linked program $M_0 \oplus M_1$ is given as $\mathrm{Beh}(M_0 \oplus M_1)$, *not* $\mathrm{Beh}(M_0) \oplus \mathrm{Beh}(M_1)$ (which would not type-check). When we discuss the behavior of a program, we consider the behavior of a closed (fully linked) whole program.

For a closed program, the behavior is straightforwardly computed (co-)recursively. For instance, consider the following simple program:

$$\mathtt{main} \mapsto \mathbf{let}\ \mathtt{x} := \mathbf{pick}(\mathbb{N})\ \mathbf{in}\ \mathtt{f(x)} \qquad \oplus \qquad \mathtt{f} \mapsto \lambda \mathtt{x}.\ \mathit{print}(\mathtt{x})$$

where **pick** randomly picks (also called demonic non-determinism) one of the possible values in the given set. Then, the behavior of this program is (intuitively) computed as follows:

$$\mathrm{Beh}(\mathtt{main()}) = \mathrm{Beh}(\mathbf{let}\ \mathtt{x} := \mathbf{pick}(\mathbb{N})\ \mathbf{in}\ \mathtt{f(x)}) = \bigcup_{x \in \mathbb{N}} \mathrm{Beh}(\mathtt{f}(x)) = \bigcup_{x \in \mathbb{N}} \{\mathtt{Obs}\ \mathit{print}\ x\}$$

where $\mathtt{Obs}\ \mathit{print}\ x$ is a single-length trace emitted from the program code, $\mathit{print}(\mathtt{x})$.

Table 1. Semantics of polysemic programs.

Approach \ Duplications	0	1	2+
Existing frameworks	UB	$\mathbf{B}_0$	Invalid
Angelic	$\bigcap \mathbf{B}$	$\bigcap \mathbf{B}$	$\bigcap \mathbf{B}$

Semantics of polysemic programs. Suppose that we are calling a function `foo`, and there are multiple definitions for it. Executing each definition results in possibly different behavior and we denote each of them using an index: ith definition's behavior is denoted as $\mathbf{B}_i$. We use $\mathbf{B}$ to denote the set containing all $\mathbf{B}_i$; *i.e.*, it is a set of set of traces.

To begin with, consider how the existing frameworks define the semantics of calling `foo` (Table. 1). If `foo` is not defined (*i.e.*, there is no function to execute), it is considered **UB**. If `foo` is defined exactly once, that one is executed with behavior $\mathbf{B}_0$. Finally, if `foo` has multiple definitions, it is considered invalid. Exactly how it is considered invalid differs by frameworks: many of them (i) syntactically reject such a program, (ii) some just choose the leftmost one and keep executing ($\mathbf{B}_0$) and (iii) some consider it as **UB**. However, none of them work for us: (ii) breaks the commutativity of linking, and (iii) breaks affinity (affinity yields `main` $\mapsto$ **`skip`** $\oplus$ `main` $\mapsto$ **`skip`** $\sqsubseteq$ `main` $\mapsto$ **`skip`**, which is false because the target triggers **UB** while the source not).

Our solution is to define the semantics as the ***intersection*** of all possible behaviors, using so-called *angelic non-determinism*; and not just for the polysemic case, but for all three cases! Before we proceed, first note that $\bigcap \mathbf{B}$ equals the usual semantics when the function is not defined or defined once.[2] This suggests that defining the polysemic case the same as $\bigcap \mathbf{B}$ is a natural *extension*, unifying all three cases.

Now we see if such a definition justifies the two axioms, $\forall a.\ a \sqsubseteq \varnothing$ and $\forall a.\ a \sqsubseteq |a| \oplus a$. It is easy to see that the former holds: having *more* definition will always result in less behavior because the intersection of behaviors monotonically decreases.

On the other hand, the latter axiom is more subtle. Consider a function call `foo()` where `foo` is defined in a and thus also in $|a|$. We use $\mathbf{b0}$ to refer to the behavior of calling `foo()` of a, $\mathbf{b1}$ for that of $|a|$, and $\mathbf{B}$ for those of the outside modules (they can very well have definitions for `foo` too). Then, we need to show $\bigcap(\{\mathbf{b0}\} \cup \mathbf{B}) \subseteq \bigcap(\{\mathbf{b0}, \mathbf{b1}\} \cup \mathbf{B})$. Using the distributive law, it suffices to show $\bigcap\ \{\mathbf{b0}\} \subseteq \bigcap\ \{\mathbf{b0}, \mathbf{b1}\}$; *i.e.*, $\mathbf{b0} \subseteq \mathbf{b0} \cap \mathbf{b1}$. Now, there are two cases: if `foo` does *not* access the state, $\mathbf{b0}$ equals $\mathbf{b1}$ and the inclusion holds. If `foo` *does* access the state, $\mathbf{b1}$ results in **UB**—recall that the core operator maps all state access to **UB**. Then, since **UB** is an identity element for $\cap$—recall that **UB** is the set of all traces—the inclusion holds again.

Example: Composing refinements of Rpt. The extended semantics still validates all the reasoning principles we have seen so far. As a sanity check, we revisit the entailment between 2 and 3 in our running example (Fig. 4). Under the hood, the refinement involved in this entailment is:

$$(S_{\text{Rpt}}\ \texttt{succ}\ \ldots) \oplus (S_{\text{Rpt}}\ \texttt{put}\ \ldots) \oplus T_{\text{Main}} \sqsubseteq (S_{\text{Rpt}}\ \texttt{put}\ \ldots) \oplus M_{\text{Main}}$$

As we can see, the left-hand side (LHS) of this refinement is polysemic. In this refinement, the behavior of LHS is exactly the same as the behavior of the right-hand side (RHS). Specifically, the behavior of LHS is computed as follows where $K[X] \triangleq$ **`let`** `a := ` X **`in`** `rpt(put,a,a); 0`.

$$\begin{aligned}&\text{Beh}(LHS) = \text{Beh}(K[\texttt{rpt(succ,1,1)}]) = \\ &\text{Beh}(K[\textbf{assume}\texttt{(succ == succ); }(f_{\texttt{succ}}\ \texttt{1})]) \bigcap \text{Beh}(K[\textbf{assume}\texttt{(succ == put); }(f_{\texttt{put}}\ \texttt{1})]) = \\ &\text{Beh}(K[f_{\texttt{succ}}\ \texttt{1}]) \bigcap \textbf{UB} = \text{Beh}(K[f_{\texttt{succ}}\ \texttt{1}]) = \text{Beh}(K[\texttt{2}]) = \text{Beh}(RHS)\end{aligned}$$

[2]In the former case, $\bigcap \emptyset$ equals **UB** since $\forall x.\ (x \in \bigcap \emptyset) \Leftrightarrow (\forall \mathbf{B}_i \in \emptyset.\ x \in \mathbf{B}_i) \Leftrightarrow \top \Leftrightarrow (x \in \textbf{UB})$. The latter case is trivial.

Definition 1. *A MRAS is a tuple* $(M, (\sqsubseteq) \in M \to M \to \textbf{\textit{Prop}}, (\oplus) \in M \to M \to M, |-| \in M \to M, \varnothing \in M)$ *satisfying:*

$$\forall a.\ a \sqsubseteq a \qquad \text{(MRAS-REF-REFL)}$$

$$\forall a, b, c.\ a \sqsubseteq b \land b \sqsubseteq c \Rightarrow a \sqsubseteq c \qquad \text{(MRAS-REF-VCOMP)}$$

$$\forall a_0, a_1, b_0, b_1.\ a_0 \sqsubseteq b_0 \land a_1 \sqsubseteq b_1 \Rightarrow a_0 \oplus a_1 \sqsubseteq b_0 \oplus b_1 \qquad \text{(MRAS-REF-HCOMP)}$$

$$\forall a, b, c.\ (a \oplus b) \oplus c = a \oplus (b \oplus c) \qquad \text{(MRAS-LINK-ASSOC)}$$

$$\forall a, b.\ a \oplus b = b \oplus a \qquad \text{(MRAS-LINK-COMM)}$$

$$\forall a.\ a = |a| \oplus a \qquad \text{(MRAS-CORE-INTRO)}$$

$$\forall a.\ a \sqsubseteq \varnothing \qquad \text{(MRAS-REF-AFFINE)}$$

$$\forall a.\ a \oplus \varnothing = a \qquad \text{(MRAS-LINK-UNIT)}$$

$$\forall a.\ ||a|| = |a| \qquad \text{(MRAS-CORE-IDEM)}$$

$$|\varnothing| = \varnothing \qquad \text{(MRAS-CORE-UNIT)}$$

$$\forall a, b.\ |a \oplus b| = |a| \oplus |b| \qquad \text{(MRAS-CORE-COMMUTE)}$$

Fig. 6. Definition of MRAS.

In the first line, we simply unfold the definition of `main`. In the second line, the call to `rpt` is computed as the intersection of two possible cases: executing (S_{Rpt} `succ` ...) or (S_{Rpt} `put` ...). In the third line, we continue computation in both cases. In the former, the condition in **`assume`** is met and thus stripped away. In the latter, however, the condition in **`assume`** is not met, and thus it triggers **`UB`**. Since **`UB`** is an identity element for $\bigcap$, we remove it and decide that the "right" branch is the former one. Continuing execution on the former case, we reach $K[2]$ which is the body of RHS.

We conclude this section with a final remark. A reader familiar with angelic non-determinism might be concerned that its misuse could inadvertently lead to an *empty behavior*. However, in our use of angelic non-determinism, it does not happen. Recall that the implementation of the whole program—say T—is monosemic (*i.e.*, not polysemic), ensuring its behavior is never an empty set. Then, in any end result $T \sqsubseteq S$ we establish, the behavior of the S also cannot be an empty set: the very definition of $\sqsubseteq$ guarantees that the set of behavior of S is larger than that of T, which is non-empty. In essence, while our use of angelic non-determinism provides fundamentally new strategies for formulating "small refinements", it is only relevant in intermediate proof states and does not influence the end result.

5 Refinement Composition Logic

In this section, we give formal definitions and rules of RCL. For expository purposes, we first formalize RCL assuming **MRAS** (**MRA S**trengthened), a slightly stronger version of MRA. This greatly simplifies presentation *and* formalization.[3] The gap between MRAS and MRA is filled in §6.

5.1 MRAS and Definitions of RCL

We first define MRAS as shown in Fig. 6. The colored axioms are the strengthened ones, having equality instead of refinements ($\sqsubseteq$).

We have already seen the first seven axioms. These each correspond to (in order): REF-REFL, REF-VCOMP, REF-HCOMP, REF-COMM, REF-ASSOC, and the two axioms in §4.3. The last four axioms are also easy to understand. MRAS-LINK-UNIT simply states that $\varnothing$ is the unit for $\oplus$. Finally, recall the intuition behind the $|-|$ operator (§4.2): it simply maps all state accesses to **`UB`**. With this intuition, the last three rules (MRAS-CORE-IDEM, MRAS-CORE-UNIT, and MRAS-CORE-COMMUTE) are straightforward.

[3]Specifically, MRAS is closer to the Iris RA (Resource Algebra), and it allows defining each symbol in the same way as in (non-step-indexed) Iris.

$$\begin{aligned}
[\![\mathsf{True} \in \mathit{mProp}]\!] &\triangleq \lambda a.\, \top \\
[\![\mathsf{False} \in \mathit{mProp}]\!] &\triangleq \lambda a.\, \bot \\
[\![(\forall x \in X.\, P\, x) \in \mathit{mProp}]\!] &\triangleq \lambda a.\, \forall x \in X.\, ([\![P\, x \in \mathit{mProp}]\!]\; a) \\
[\![(\exists x \in X.\, P\, x) \in \mathit{mProp}]\!] &\triangleq \lambda a.\, \exists x \in X.\, ([\![P\, x \in \mathit{mProp}]\!]\; a) \\
[\![\boxed{b}]\!] &\triangleq \lambda a.\, b \preccurlyeq a \\
[\![\models\!\dot{\Rrightarrow} P]\!] &\triangleq \lambda a.\, \exists b.\; a \sqsubseteq b \wedge [\![P]\!]\, b \\
[\![P * Q]\!] &\triangleq \lambda a.\, \exists b, c.\; a = b \oplus c \wedge [\![P]\!]\, b \wedge [\![Q]\!]\, c \\
[\![P \mathrel{-\!\!*} Q]\!] &\triangleq \lambda a.\, \forall b \in [\![P]\!].\, [\![Q]\!]\, (a \oplus b) \\
[\![\square\, P]\!] &\triangleq \lambda a.\, [\![P]\!]\, |a|
\end{aligned}$$

Fig. 7. Definitions of symbols in RCL.

Semantic domain of mProp. Now, given an arbitrary MRAS, we will define RCL. We begin with the semantic domain for propositions in RCL: *mProp* (abbreviation of *module propositions*). As said, *mProp* is defined as the set of modules, and (now in full technical detail) a condition that ensures affinity. Specifically, we have:

$$[\![\mathit{mProp}]\!] \triangleq \{P \in M \longrightarrow \textbf{\textit{Prop}} \mid \forall a \preccurlyeq b,\, P\, a \Rightarrow P\, b\} \qquad\qquad [\![P \vdash Q]\!] \in \textbf{\textit{Prop}} \triangleq P \subseteq Q$$

where ***Prop*** stands for meta-level propositions (*e.g.*, that of mathematics or Coq) and $a \preccurlyeq b \triangleq \exists c.\; b = a \oplus c$. Such a relation $\preccurlyeq$ is called an *inclusion relation* and is reflexive and transitive. With this, *mProp* is upward-closed with $\preccurlyeq$. This ensures affinity: *i.e.*, AFFINE.

The logical entailment between *mProp*s, $P \vdash Q$, is simply defined as a set inclusion.

Definitions of symbols. The primitive symbols in RCL are as follows:[4]

$$\begin{aligned}
a \in M &::= a \mid a \oplus a \mid |a| \\
P \in \mathit{mProp} &::= \boxed{a} \mid \models\!\dot{\Rrightarrow} P \mid P * P \mid \square\, P \mid \\
&\qquad\quad P \mathrel{-\!\!*} P \mid \exists x \in X.\, P \mid \forall x \in X.\, P \mid \mathsf{False} \mid \mathsf{True} \\
\Phi \in \textbf{\textit{Prop}} &::= P \vdash P \mid P \dashv\vdash P \mid \ldots
\end{aligned}$$

The definitions of these symbols are given in Fig. 7. These are either already explained ($\boxed{-}$, $\models\!\dot{\Rrightarrow}$, $*$, and $\square$) or standard ($\mathrel{-\!\!*}$, quantifiers and truth values). We mention two points.

First, the definition of $\square\, P$ uses core operator, specifying the set of modules that satisfy P *statelessly*. Then, the axioms of $\square$ are direct consequences of the properties of the core operator. Specifically, MRAS-CORE-INTRO and MRAS-CORE-IDEM entails $\forall a.\; |a| = |a| \oplus |a|$, which justifies PERS-DUP.

Second, the definition of $\boxed{b}$ is now extended to a set of modules that "includes" b. This is needed to satisfy the condition on affinity. As we will see next, it satisfies all the rules we need.

5.2 Rules of RCL

Now we see the logical rules in RCL. To understand them, it is best to begin our journey with the adequacy of RCL: the rules LOGIC-I and LOGIC-E. These rules say that $\boxed{a} \vdash \models\!\dot{\Rrightarrow} \boxed{b}$ precisely captures the proposition $a \sqsubseteq b$, and these rules act as the main entry/exit point of RCL, respectively.

To see why LOGIC-I and LOGIC-E holds, we begin with the following lemma:

LEMMA 5.1 (INCL-REF). $\forall a\, b, a \preccurlyeq b \longrightarrow b \sqsubseteq a$

PROOF. By unfolding $\preccurlyeq$ and applying MRAS-LINK-UNIT, MRAS-REF-HCOMP, and MRAS-REF-AFFINE. ■

[4] Logical connectives like $\wedge$ and $\vee$ could be derived from quantifiers ($\forall$ and $\exists$, respectively), thus omitted.

$$\frac{a \sqsubseteq b}{\boxed{a} \vdash |\!\dot{\Rrightarrow} \boxed{b}}\ \textsc{logic-i} \qquad \frac{\boxed{a} \vdash |\!\dot{\Rrightarrow} \boxed{b}}{a \sqsubseteq b}\ \textsc{logic-e} \qquad \frac{}{P \vdash P}\ \textsc{entails-refl} \qquad \frac{P \vdash Q \quad Q \vdash R}{P \vdash R}\ \textsc{entails-trans}$$

$$\frac{P \vdash Q}{|\!\dot{\Rrightarrow} P \vdash |\!\dot{\Rrightarrow} Q}\ \textsc{ref-mono} \qquad \frac{P \vdash Q}{P * R \vdash Q * R}\ \textsc{star-mono} \qquad \frac{Q \vdash P \quad R \vdash S}{P \mathrel{-\!\!*} R \vdash Q \mathrel{-\!\!*} S}\ \textsc{wand-mono} \qquad \frac{P \vdash Q}{\Box P \vdash \Box Q}\ \textsc{pers-mono}$$

$$\frac{}{P \vdash \mathsf{True}}\ \textsc{true-i} \qquad \frac{}{\mathsf{False} \vdash P}\ \textsc{false-e} \qquad \frac{\forall \mathbf{x} \in X.\ (P \vdash Q)}{P \vdash \forall x \in X.\ Q}\ \textsc{univ-i} \qquad \frac{t \in X \quad P \vdash \forall x \in X.\ Q}{P \vdash Q[t/x]}\ \textsc{univ-e}$$

$$\textsc{star-link}\ \ \boxed{a} * \boxed{b} \dashv\vdash \boxed{a \oplus b} \qquad \textsc{star-comm}\ \ P * Q \vdash Q * P \qquad \textsc{star-assoc}\ \ (P * Q) * R \vdash P * (Q * R)$$

$$\textsc{ref-i}\ \ P \vdash |\!\dot{\Rrightarrow} P \qquad \textsc{ref-e}\ \ |\!\dot{\Rrightarrow} |\!\dot{\Rrightarrow} P \vdash |\!\dot{\Rrightarrow} P \qquad \textsc{ref-frame}\ \ |\!\dot{\Rrightarrow} P * Q \vdash |\!\dot{\Rrightarrow}(P * Q)$$

$$\frac{P * Q \vdash R}{P \vdash Q \mathrel{-\!\!*} R}\ \textsc{wand-i} \qquad \frac{P \vdash Q \mathrel{-\!\!*} R}{P * Q \vdash R}\ \textsc{wand-e} \qquad \textsc{pers-star}\ \ \Box P * \Box Q \vdash \Box(P * Q) \qquad \textsc{pers-idemp}\ \ \Box P \vdash \Box\Box P$$

Fig. 8. Selected rules for RCL.

Then, we have the following theorem:

Theorem 5.2 (adequacy). *logic-i and logic-e holds.*

Proof. By unfolding definitions, the goal becomes: $a \sqsubseteq b \Leftrightarrow (\forall c.(a \preccurlyeq c \Rightarrow \exists d.c \sqsubseteq d \wedge b \preccurlyeq d))$.
($\Rightarrow$) Apply Lemma 5.1 to a $\preccurlyeq$ c, instantiate d with b, and then apply mras-ref-vcomp.
($\Leftarrow$) Instantiate c with a, apply Lemma 5.1 to b $\preccurlyeq$ d, and then apply mras-ref-vcomp. ■

All the rules in Fig. 8 follow from a few steps of unfolding and applying axioms in MRAS. We quickly go over rather standard parts and focus on interesting observations.

Entailments, monotonicity, quantifiers, truths, and separating conjunction. Entailment forms a preorder (entails-refl and entails-trans). Given $P \vdash Q$, the monotonicity rules (the ones in the second row; ref-mono, star-mono, wand-mono, and pers-mono) allow any positive occurrences of P in RCL to be substituted into Q (and in reverse order for negative occurrences).

Rules for truth values, True/False (denoting a full set/an empty set), are self-explanatory (true-i/false-e). Rules for quantifiers, $\forall/\exists$, are also standard (univ-i and univ-e/rules for $\exists$ are omitted).

Separating conjunction (star) is just a reflection of $\oplus$ in the logic level. star-link equates $\oplus$ in the domain with $*$ in the logic. Here, $P \dashv\vdash Q$ is just a shorthand for $(P \vdash Q) \wedge (Q \vdash P)$. We also have commutativity and associativity of $*$ (star-comm and star-assoc).

Correspondence between $\sqsubseteq$ and $|\!\dot{\Rrightarrow}$. The refines modality ($|\!\dot{\Rrightarrow}$) is originally called an *update* modality in the original Iris, having the three primitive rules ref-i, ref-e, and ref-frame. It turns out, with the definition of refines modality above, these three rules precisely correspond to ref-refl, ref-vcomp, and ref-hcomp! To the best of our knowledge, such a correspondence between compositionality theorems for $\sqsubseteq$ and primitive rules for $|\!\dot{\Rrightarrow}$ has not been observed before.[5]

[5]Even the authors of DimSum [Sammler et al. 2023] and CCR [Song et al. 2023]—recent projects that span both refinement and Iris—were not aware of such a correspondence.

It is relatively easy to check that theorems of $\sqsubseteq$ imply the primitives rules of $\dot{\Rrightarrow}$ by unfolding the definition. We present the reverse direction: we use the latter in RCL to derive the former.

$$\dfrac{\boxed{a} \vdash \dot{\Rrightarrow}\boxed{a}}{a \sqsubseteq a} \qquad \dfrac{\dfrac{\dfrac{a \sqsubseteq b}{\boxed{a} \vdash \dot{\Rrightarrow}\boxed{b}} \quad \dfrac{b \sqsubseteq c}{\boxed{b} \vdash \dot{\Rrightarrow}\boxed{c}}}{\boxed{a} \vdash \dot{\Rrightarrow}\boxed{b} \vdash \dot{\Rrightarrow}\dot{\Rrightarrow}\boxed{c} \vdash \dot{\Rrightarrow}\boxed{c}}}{a \sqsubseteq c} \qquad \dfrac{\dfrac{\dfrac{a \sqsubseteq b}{\boxed{a} \vdash \dot{\Rrightarrow}\boxed{b}} \quad \dfrac{a' \sqsubseteq b'}{\boxed{a'} \vdash \dot{\Rrightarrow}\boxed{b'}}}{\boxed{a} * \boxed{a'} \vdash \dot{\Rrightarrow}(\boxed{b} * \boxed{a'}) \vdash \dot{\Rrightarrow}\dot{\Rrightarrow}(\boxed{b} * \boxed{b'}) \vdash \dot{\Rrightarrow}(\boxed{b} * \boxed{b'})}}{a \oplus a' \sqsubseteq b \oplus b'}$$

The first proof derives REF-REFL by instantiating REF-I with $\boxed{a}$ and then applying LOGIC-E. The second proof derives REF-VCOMP as follows: we first lift each refinement into the logic with LOGIC-I, compose them with REF-MONO, remove duplicated $\dot{\Rrightarrow}$ with REF-E, and get the final result with LOGIC-E. The third proof derives REF-HCOMP in a similar way: we first lift each refinement with LOGIC-I, compose them with STAR-MONO, REF-MONO and REF-FRAME, remove duplicated $\dot{\Rrightarrow}$ with REF-E, and get the final result with LOGIC-E.

Magic wand. *Magic wand*, $P \mathrel{-\!\!*} Q$, is a standard symbol in separation logic. One way to understand wand is as a (right) adjoint of the separating conjunction, satisfying the rules WAND-I and WAND-E. The wand is also often called separating implication, thanks to the following rule: $P * (P \mathrel{-\!\!*} Q) \vdash Q$ (derived by STAR-COMM, WAND-I and ENTAILS-REFL).

In the context of separation logic, it is known that while the wand might not necessarily give additional proof power, it is very handy in various ways to streamline proofs [Charguéraud 2023]; hence its widespread use.

However, the wand is exotic in refinement frameworks. Indeed, the adjoint for $\oplus$ operator (say, $-\!\oplus$) is not necessarily well-defined: *e.g.*, $(\texttt{foo} \mapsto \lambda\_.\, \mathit{print}(1)) \mathrel{-\!\oplus} (\texttt{foo} \mapsto \lambda\_.\, \mathit{print}(2))$ is not well-defined (*i.e.*, there is no module that translates $\mathit{print}(1)$ into $\mathit{print}(2)$ when linked). However, $-\!\!*$ is well-defined in the logical system of RCL where the semantic domain comprises both modules and logical propositions; intuitively, a nonsensical wand like the one above just results in False.

A more practical way to understand wand is as a drop-in replacement of logical entailment ($\vdash$). Given two *mProp*s, logical entailment returns a meta-level proposition, ***Prop***, whereas wand returns an *mProp* again, which still resides inside the logic.

Persistence (stateless) modality. We have already discussed the most important rules of persistence modality in §4.2. We have just two more rules here. PERS-STAR basically says that if we have two stateless modules, the linked one is also stateless. PERS-IDEMP ensures that the persistence modality is idempotent; with PERS-E, we have $\square P \dashv\vdash \square\square P$.

6 More on Algebra: MRAS, MRA, and Connection to Iris

In this section, we discuss the underlying algebra more. The key challenge we tackle here is bridging the gap between MRAS and MRA. Our key idea is to give a novel homomorphism to connect these.

In §6.1, we present MRA and give a homomorphism from MRA to MRAS. As a result, we expose MRA (which is easier to satisfy) to the user, translate MRA into MRAS using a homomorphism, and instantiate RCL with MRAS. Such a homomorphism is achieved by taking a novel quotient on the given MRA.

In §6.2, we present some bonus results. Readers unfamiliar with Iris can safely skip this subsection. We connect RCL with Iris by giving a translation from MRAS to Iris RA (Resource Algebra). Actually, the resulting RA is slightly stronger than the original RA (*i.e.*, MRAS assumes one axiom that is not required in RA), and we call this RAC (RA where Core commutes). It turns out that we can use the same technique as in §6.1 to *weaken* the axioms of RAC: we call this RACW (RAC Weakened). This could potentially be useful for future constructions of Iris RA.

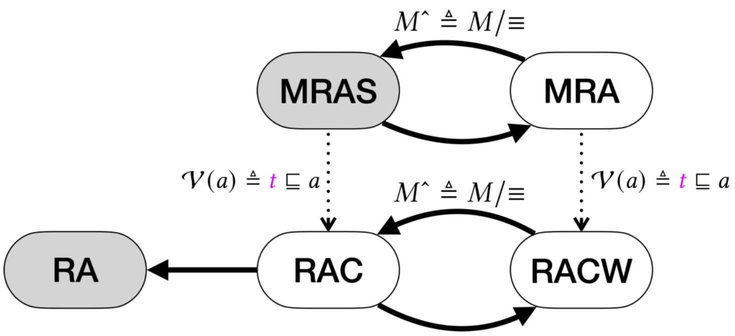
$M^\hat{} \triangleq M/\equiv$
MRAS
MRA
$\mathcal{V}(a) \triangleq t \sqsubseteq a$
$M^\hat{} \triangleq M/\equiv$
$\mathcal{V}(a) \triangleq t \sqsubseteq a$
RA
RAC
RACW

Fig. 9. A diagram showing relations between algebras.

In summary, we prove the diagram shown in Fig. 9. Solid arrows are homomorphisms where the text above them denotes the quotient used on such; those without text are trivial homomorphisms (from an algebra with stronger axioms to weaker axioms). Dotted arrows are translation of operators in RCL into Iris. Grey-colored boxes are the algebras we actually use for instantiating the logic.

6.1 Weakening the Axioms of MRAS

Now we define MRA, containing the set of axioms that actually appears to the user.

Definition 2. *A MRA is a MRAS with axioms* MRAS-CORE-INTRO, MRAS-LINK-COMM, *and* MRAS-LINK-ASSOC *weakened as follows:*

$$\forall a.\ a \sqsubseteq |a| \oplus a \qquad \text{(MRA-CORE-INTRO)}$$

$$\forall a, b.\ a \oplus b \sqsubseteq b \oplus a \qquad \text{(MRA-LINK-COMM)}$$

$$\forall a, b, c.\ (a \oplus b) \oplus c \sqsubseteq a \oplus (b \oplus c) \qquad \text{(MRA-LINK-ASSOC)}$$

Note that MRAS-CORE-INTRO (using equality) is weakened to MRA-CORE-INTRO (using $\sqsubseteq$) that is actually satisfied by the polysemic semantics we discussed (§4.3). Also, MRAS-LINK-COMM and MRAS-LINK-ASSOC are weakened correspondingly.

Then, we will *internally* cast MRA into MRAS and instantiate RCL with MRAS, but the resulting logic behaves the same way as if we were instantiating the logic with the operators and notion of module given in MRA; let us refer to this *hypothetical* logic as a RCL$_2$. We do this by establishing a homomorphism from MRA to MRAS. That is, given an MRA, we want to derive an MRAS with $M^\hat{}$ as a notion of module—with embedding $(-)^\hat{} \in M \rightarrow M^\hat{}$—and operators for $M^\hat{}$ (we overload the notations) that satisfies all the axioms in MRAS and also the following properties about homomorphism:

$$a = b \Rightarrow a^\hat{} = b^\hat{} \qquad (a \oplus b)^\hat{} = a^\hat{} \oplus b^\hat{} \qquad |a|^\hat{} = |a^\hat{}| \qquad a^\hat{} \sqsubseteq b^\hat{} \Leftrightarrow a \sqsubseteq b \qquad \text{(HOM-MRA)}$$

The above conditions, basically saying that everything commutes, are sufficient to *simulate* any proof of RCL$_2$ in RCL: given a proposition P in RCL$_2$, we map this into a proposition $P^\hat{}$ in RCL by simply changing all $\boxed{a}$ into $\boxed{a^\hat{}}$. Then, every rule application in RCL$_2$ can be simulated in RCL while preserving such a simulation-invariant. For example, consider STAR-LINK: $\boxed{a \oplus b} \dashv\vdash \boxed{a} * \boxed{b}$ in RCL$_2$ is simulated in RCL as $\boxed{(a \oplus b)^\hat{}} \dashv\vdash \boxed{a^\hat{} \oplus b^\hat{}} \dashv\vdash \boxed{a^\hat{}} * \boxed{b^\hat{}}$. Also, the LOGIC-I in RCL$_2$ is simulated in RCL as follows: $a \sqsubseteq b \Rightarrow a^\hat{} \sqsubseteq b^\hat{} \Rightarrow \boxed{a^\hat{}} \vdash \models\!\!\dot{\Rightarrow} \boxed{b^\hat{}}$ and vice versa for LOGIC-E: $\boxed{a^\hat{}} \vdash \models\!\!\dot{\Rightarrow} \boxed{b^\hat{}} \Rightarrow a^\hat{} \sqsubseteq b^\hat{} \Rightarrow a \sqsubseteq b$.

Now, we are left with actually building such a homomorphism. We first define $\equiv$ as follows:

$$a \equiv b \triangleq a \sqsupseteq\!\sqsubseteq b \land |a| \sqsupseteq\!\sqsubseteq |b| \qquad M^\hat{} \triangleq M/\equiv \qquad a^\hat{} \triangleq [a]_\equiv$$

where $P \sqsupseteq\sqsubseteq Q$ denotes $P \sqsubseteq Q \land Q \sqsubseteq P$. $\equiv$ forms an equivalence relation over M and we use $M/\equiv$ as $M^\wedge$. $[a]_\equiv$ is an equivalence class over $\equiv$ containing a. $\equiv$ respects all the operators, $\sqsubseteq$, $\oplus$, and $|-|$:

$$\forall a, a', b, b'.\ a \equiv a' \Rightarrow b \equiv b' \Rightarrow (a \sqsubseteq b \Leftrightarrow a' \sqsubseteq b')$$
$$\forall a, a', b, b'.\ a \equiv a' \Rightarrow b \equiv b' \Rightarrow a \oplus b \equiv a' \oplus b'$$
$$\forall a, a'.\ a \equiv a' \Rightarrow |a| \equiv |a'|$$

and induces the operators $\sqsubseteq$, $\oplus$, $|-|$ for $M^\wedge$ for free as follows:

$$[a]_\equiv \sqsubseteq [b]_\equiv \triangleq a \sqsubseteq b \qquad [a]_\equiv \oplus [b]_\equiv \triangleq [a \oplus b]_\equiv \qquad |[a]_\equiv| \triangleq [|a|]_\equiv$$

Such definitions satisfy all the axioms for MRAS and properties for homomorphism (hom-mra). We only discuss the proof of mras-core-intro here. For any given a, we need to prove:

$$a \equiv |a| \oplus a \Leftrightarrow a \sqsupseteq\sqsubseteq |a| \oplus a \land |a| \sqsupseteq\sqsubseteq ||a| \oplus a| \Leftrightarrow |a| \sqsupseteq\sqsubseteq ||a| \oplus a| \Leftrightarrow |a| \sqsupseteq\sqsubseteq ||a|| \oplus |a|$$

where we first unfold $\equiv$, discharge the first condition with mra-core-intro and mras-ref-affine, simplify with mras-core-commute. ($\sqsubseteq$) By applying mra-core-intro with $|a|$. ($\sqsupseteq$) By applying mras-ref-affine. ■

6.2 Embedding MRA into Iris RA

Now, we show how one can embed MRA into the original RA in Iris. This could potentially be useful when enriching RCL with more advanced protocols. Readers unfamiliar with Iris can safely skip this subsection.

We will be translating refinement in MRA into *frame-preserving update* in Iris RA. For this, we parameterize over one additional argument, $t \in M$: the whole target program in the verification of our interest. Then, given a MRA and $t \in M$, we can define a (unital) resource algebra as $(M, \mathcal{V}, |-|, (\oplus), \emptyset)$ where $\mathcal{V}$ is defined as follows:

$$\mathcal{V}(a) \triangleq t \sqsubseteq a$$

Then, we have the following rules corresponding to logic-e and logic-i:

$$\overset{\text{IRIS-I}}{\frac{a \sqsubseteq b}{\boxed{a} \vdash \models\!\dot{\Rrightarrow} \boxed{b}}} \qquad\qquad \overset{\text{IRIS-E}}{\frac{\boxed{t} \vdash \models\!\dot{\Rrightarrow} \boxed{b}}{t \sqsubseteq b}}$$

Note that in iris-e, the target side is fixed as t, but this should be okay since we will be using such a rule only for the end result (the result of composing all the small refinements).

Such an embedding satisfies all the axioms of RA. In fact, the resulting RA is slightly *stronger* than the original RA that it satisfies mras-core-commute, whereas in the original Iris RA the required axiom is a slightly weaker version:

$$\forall a, b.\ a \preccurlyeq b \Rightarrow |a| \preccurlyeq |b| \qquad \text{(RA-CORE-MONO)}$$

Let us name a RA with ra-core-mono strengthened to mras-core-commute as RAC (RA where Core commutes). Then, our construction results in RAC.

Weakening the axioms of RA? The above weakening and embedding led us to a question of whether we could weaken the axiom of original RA in the same way we did for MRA. As it turns out, there is not much room to squeeze in RA, but we can still have an interesting result.

First, note that RA does *not* have the axiom mras-core-commute which was used crucially in the validation of mras-core-intro above. Thus, the same strategy as above would not work for RA.

However, if we consider *RAC*, we can weaken its axioms using the above strategy. This is still an interesting result because axiom mras-core-commute—the only difference between RA and RAC—is commonly satisfied by RA instances. Indeed, to the best of our knowledge, all the instances of RA

Table 2. Summary of refinement frameworks in terms of building blocks and theories for composition.

	Vanilla Refinements	Layered Refinements	RCL
Building block	$T \sqsubseteq S$	$S_1 \vDash T_2 : S_2$	Logical formula
Theories for composition	REF-REFL, REF-VCOMP, REF-HCOMP	Layer Calculus	Logical rules

$$P \Rrightarrow\!\!\ast\, Q \triangleq P \mathrel{-\!\!*} \mathrel{|\!\!\Rrightarrow} Q \qquad \text{VS-REFL}\ \frac{}{P \Rrightarrow\!\!\ast\, P} \qquad \text{VS-TRANS}\ \frac{(P \Rrightarrow\!\!\ast\, Q) * (Q \Rrightarrow\!\!\ast\, R)}{P \Rrightarrow\!\!\ast\, R} \qquad \text{VS-FRAME}\ \frac{(P \Rrightarrow\!\!\ast\, P') * (Q \Rrightarrow\!\!\ast\, Q')}{(P * Q) \Rrightarrow\!\!\ast\, (P' * Q')}$$

$$\text{LAYER-HCOMP}\ \frac{S_1 \vDash T_2 : S_2 \quad S_2 \vDash T_3 : S_3}{S_1 \vDash T_2 \oplus T_3 : S_3} \qquad \text{LAYER-VCOMP}\ \frac{S_1 \sqsubseteq S_1' \quad S_1' \vDash T_2 : S_2' \quad S_2' \sqsubseteq S_2}{S_1 \vDash T_2 : S_2} \qquad \text{LAYER-TCOMP}\ \frac{S_1 \vDash T_2 : S_2 \quad S_x \vDash T_y : S_y}{S_1 \oplus S_x \vDash T_2 \oplus T_y : S_2 \oplus S_y}$$

Fig. 10. Definition of view shift operator and its rules (above), and the essence of Layer Calculus (below).

in the official Iris except for the one (that encodes state transition systems [Jung et al. 2015]) satisfy MRAS-CORE-COMMUTE.

Specifically, we define a weakened version of RAC, called RACW (RAC Weakened), as follows:

Definition 3. *A RACW is a RAC with following five axioms weakened by changing equality into* $\rightsquigarrow$ *and* $\leftrightsquigarrow$*:*

$$\forall a.\ a \rightsquigarrow |a| \oplus a \qquad \text{(RACW-CORE-INTRO)}$$
$$\forall a.\ ||a|| \rightsquigarrow |a| \qquad \text{(RACW-CORE-IDEM)}$$
$$\forall a, b.\ a \oplus b \rightsquigarrow b \oplus a \qquad \text{(RACW-ADD-COMM)}$$
$$\forall a, b, c.\ (a \oplus b) \oplus c \rightsquigarrow a \oplus (b \oplus c) \qquad \text{(RACW-ADD-ASSOC)}$$
$$\forall a, b.\ |a \oplus b| \leftrightsquigarrow |a| \oplus |b| \qquad \text{(RACW-CORE-COMMUTE)}$$

where $a \leftrightsquigarrow b$ is a syntactic sugar for $a \rightsquigarrow b \wedge b \rightsquigarrow a$. Then, a homomorphism from RACW into RAC could be constructed in the same way as above but with the following equivalence relation:

$$a \equiv b \triangleq a \leftrightsquigarrow b \wedge |a| \leftrightsquigarrow |b|$$

The resulting quotient satisfies all axioms of RA and also the following rules for homomorphism:

$$a = b \Rightarrow a^\wedge = b^\wedge \qquad (a \oplus b)^\wedge = a^\wedge \oplus b^\wedge \qquad |a|^\wedge = |a^\wedge| \qquad \mathcal{V}^\wedge a^\wedge \Leftrightarrow \mathcal{V} a \qquad a^\wedge \rightsquigarrow^\wedge b^\wedge \Leftrightarrow a \rightsquigarrow b \quad \text{(HOM-RA)}$$

This result could potentially be useful for future constructions of RA that it obligates, assuming one stronger axiom, multiple weaker axioms to the user.

7 Derived Constructs and Their Applications

An important virtue of modern separation logic including Iris is that they have a *minimal core* [Pym et al. 2004]: tremendous efforts have been made to reduce the logic into minimal primitive mechanisms. On the other hand, this means that a plethora of constructs are *derived* on top of these minimal primitives. This section reports on our efforts to understand some of these important constructs in the RCL side; *i.e.*, finding their correspondence in the refinement setting.

7.1 View Shift Operator

View shift operator ($\Rrightarrow\!\!\ast$) is a construct widely used in Iris originating from [Dinsdale-Young et al. 2010]. As presented in Fig. 10, it is just a syntactic sugar for a combination of magic wand and update (refines) modality. View shifts are reflexive (VS-REFL) and transitive (VS-TRANS), and also horizontally composable (VS-FRAME); they follow directly from the rules in Fig. 8.

Interestingly, view shift operator reveals an alternative approach to define "layered refinement" and the Layer Calculus inside RCL. Table. 2 summarizes these different approaches in terms of *building blocks* and theories for composition. Let us begin by reviewing the essence of layered refinements.

Layer Calculus. The idea of layered refinement was popularized by CAL (Certified Abstraction Layers) [Gu et al. 2015], and its key idea is now widely adopted in various frameworks [Lee et al. 2023; Sammler et al. 2023][6]. In this section, we discuss a general pattern among these.

Actually, we have already seen the key idea behind layered refinement with Fig. 2. That is, layered refinement is simply the following abstraction built on top of vanilla refinements:

$$S_1 \vDash T_2 : S_2 \quad \triangleq \quad S_1 \oplus T_2 \sqsubseteq S_2$$

which says that the "client" T_2 refines S_2 while using the specification of the "library" S_1.

What makes layered refinement particularly useful is the Layer Calculus—a set of composition rules for composing these building blocks—shown in Fig. 10. These rules allow modular (via LAYER-HCOMP) and gradual (via LAYER-VCOMP) verification. The final rule, LAYER-TCOMP, allows composing two layered refinements that do not depend on each other.

Layer Calculus in RCL. We define layered refinement in RCL as follows, along with adequacy:

$$(P \vDash Q : R) \in \mathit{mProp} \triangleq (P * Q) \mathrel{-\!\!*} \mathrel{|\!\!\dot{\Rrightarrow}} R \qquad \overset{\text{LAYER-I}}{\frac{a \vDash b : c}{\boxed{a} \vDash \boxed{b} : \boxed{c}}} \qquad \overset{\text{LAYER-E}}{\frac{\boxed{a} \vDash \boxed{b} : \boxed{c}}{a \vDash b : c}}$$

we are overriding the notation $\vDash$ for the layered refinement in RCL (having the type *mProp*). This definition is equivalent to $(P \vDash Q : R) \triangleq Q \mathrel{-\!\!*} (P \mathrel{\Rrightarrow\!\!*} R)$. Our layered refinement in RCL satisfies all the rules of Layer Calculus (Fig. 10); the proofs are straightforward, and we omit them here.

Now, consider an example where we compose three layered refinements (for C_0, C_1, and C_2 each) in the original layer calculus:

$$\frac{(L_0 \vDash C_0 : L_0 * L_1) \land (L_1 * L_0 \vDash C_1 * C_2 : L_2 * L_3) \implies (L_0 \vDash C_0 * C_1 * C_2 : L_2 * L_3)}{(L_0 \vDash C_0 : L_0 * L_1) \land (L_1 \vDash C_1 : L_2) \land (L_0 \vDash C_2 : L_3) \implies (L_0 \vDash C_0 * C_1 * C_2 : L_2 * L_3)}$$

Here, we apply LAYER-TCOMP for the latter two (C_1 and C_2) and then apply LAYER-HCOMP to conclude the proof.

Of course, we can prove this the same way in RCL using the same rules. Moreover, the layered refinement defined in RCL enables (arguably) a more streamlined proof using the rules of RCL:

$$\frac{(L_0 \mathrel{\Rrightarrow\!\!*} L_0 * L_1) * (L_1 \mathrel{\Rrightarrow\!\!*} L_2) * (L_0 \mathrel{\Rrightarrow\!\!*} L_3) \vdash (L_0 \mathrel{\Rrightarrow\!\!*} L_2 * L_3)}{(L_0 \vDash C_0 : L_0 * L_1) * (L_1 \vDash C_1 : L_2) * (L_0 \vDash C_2 : L_3) \vdash (L_0 \vDash C_0 * C_1 * C_2 : L_2 * L_3)}$$

In this proof, we start by pulling out $C_0 * C_1 * C_2$ from the goal and feed them into each layered refinement in the premise. Then, we are left with composing view shifts, which are easier to work with since the modules C_i are already discharged. Usual rules for layer composition still work with view shifts: *e.g.*, LAYER-HCOMP correspond to VS-TRANS and LAYER-TCOMP to VS-FRAME.

Such flexibility comes from the use of magic wand, which is (as discussed in §5) a new feature enabled by RCL.

$\boxed{S_{\text{Mem}}}$

```
private next: 𝕃 := 0
private mem: 𝕃 → Option (List 𝕍) :=
  λ_. None
def calloc(sz: int64): ℙ ≡
  var d := pick(𝕃)
  var next := next + d + 1
  mem := mem[next ← init_list sz 0]
  (next, 0)
def load(p: ℙ) ≡
  var vs := unwrapUB(mem[fst p])
  var r  := unwrapUB( vs[snd p])
  r
def store(p: ℙ, v: 𝕍): 1 ≡ ...
```

$\boxed{T_{\text{Var}}}$

```
private p: ℙ

def init(): 1 ≡
  p := calloc(1)

def get(): int64 ≡
  load(p)

def set(w: int64): 1 ≡
  store(p, w)
```

$\boxed{S_{\text{Var}}}$

```
private v: Option (int64)
  := None
def init(): 1 ≡
  v := Some(0)

def get(): int64 ≡
  unwrapUB(v)

def set(w: int64): 1 ≡
  v := unwrapUB(w)
```

$\mathbb{V} \triangleq \mathbb{P} \uplus$ `int64` (pointer and integer) $\quad \mathbb{P} \triangleq (\mathbb{L} \times \mathbb{N}$ (block and offset) $\uplus$ String (function pointers)) $\quad \mathbb{L} \triangleq \mathbb{N}$

`unwrapUB`$(x \in \text{Option } (X)) \in X \triangleq$ **match** x **with** Some$(x) \Rightarrow x \mid$ None $\Rightarrow$ **triggerUB**

Fig. 11. An example to demonstrate fancy update modality.

7.2 Fancy Update

One pattern in refinement frameworks is what we call a (memory) *stealing* pattern [Gu et al. 2015; Song et al. 2023]. While the exact detail differs by the framework, its essence could be captured with a minimal contrived example shown in Fig. 11.

On the left is a memory module S_{Mem}, which offers a usual block-based memory model [Leroy 2006] to client modules. It manages a "memory", a map from blocks of type $\mathbb{L}$ to contents of type Option (List $\mathbb{V}$), as a module private state, and provides a standard interface of memory to the clients. For instance, `calloc` allocates a fresh block of the required size, initialized to 0s, and `load` returns the value stored in the memory pointed to by the argument pointer, where a pointer is a pair of a block and an offset for the content ($\mathbb{L} \times \mathbb{N}$). Also, observe that these functions utilize some degree of non-determinism to lubricate the composition of refinements: `calloc` nondeterministically chooses the next block, and `load` becomes undefined if the pointer is not pointing to a valid location.

On the right are an implementation and a specification of a Var module, which stores a single value of type `int64`. The module offers three functions: `init` to initialize and `get`/`set` to get/set the variable, respectively. The implementation T_{Var} uses Mem module to store the data, but S_{Var} abstracts away the calls to Mem and directly stores the data in its module-private variable `v`.

Then, we have the following refinement:

$$S_{\text{Mem}} \oplus T_{\text{Var}} \sqsubseteq S_{\text{Mem}} \oplus S_{\text{Var}}$$

where Var effectively *steals* the memory block for `p` from Mem and put it into its module-private state, *i.e.*, the private variable `v`. The rationale behind such refinement is that it facilitates modular proofs by *separation of concerns*. In general, memory is a shared state among numerous modules, and considering all the interactions in a monolithic manner induces unnecessary complexity—in the case of T_{Var}, the private storage `p` should be the only concern, not all the interactions happening in the shared memory. The refinement at hand precisely captures this intuition, separating T_{Var} from irrelevant interactions by abstracting it into S_{Var}.

Looking at the above refinement again, note that the same memory module—S_{Mem}—appears again on the source side. This allows multiple different modules to employ the same pattern on their verification. In some sense, the module Mem is behaving like an *invariant* in this pattern:

[6]DimSum used layered refinement in program verification, not the compiler verification.

$$\vDash\!\!\Rrightarrow_I P \triangleq I \mathrel{\Rrightarrow\!\!\!\circledast} (I * P)$$

$$\textsc{fupd-mono}\ \frac{P \vdash Q}{\vDash\!\!\Rrightarrow_I P \vdash \vDash\!\!\Rrightarrow_I Q} \qquad \textsc{fupd-i}\ \dot{\vDash\!\!\Rrightarrow} P \vdash \vDash\!\!\Rrightarrow_I P \qquad \textsc{fupd-e}\ \vDash\!\!\Rrightarrow_I \vDash\!\!\Rrightarrow_I P \vdash \vDash\!\!\Rrightarrow_I P \qquad \textsc{fupd-frame}\ \vDash\!\!\Rrightarrow_I P * Q \vdash \vDash\!\!\Rrightarrow_I (P * Q)$$

Fig. 12. Definition of fancy update and its rules.

every module can access it in their refinement, but they should put it back so that others can also use it. We also note that such a pattern is not specific to Mem; any module with a similar structure (*i.e.*, key-value data storage) could employ a similar pattern.

In RCL, such a pattern is captured with a derived notion $\vDash\!\!\Rrightarrow_I$ shown in Fig. 12. This modality, $\vDash\!\!\Rrightarrow_I$, is a degenerated version of the so-called *fancy update* modality in Iris.[7] It is known [Jung et al. 2018] (and easy to check) that this modality satisfies standard rules shown in Fig. 12. The correspondence is interesting on its own and could potentially be useful in streamlining the composition proof of multiple such refinements using the stealing pattern.

For instance, consider the following case:

$$\begin{array}{rl} i: & \boxed{S_{\text{Mem}}} * \boxed{T_{\text{Var}}} \vdash \dot{\vDash\!\!\Rrightarrow}(\boxed{S_{\text{Mem}}} * \boxed{S_{\text{Var}}}) \\ ii: & \boxed{S_{\text{Mem}}} * \boxed{T_{\text{Var2}}} \vdash \dot{\vDash\!\!\Rrightarrow}(\boxed{S_{\text{Mem}}} * \boxed{S_{\text{Var2}}}) \end{array} \quad goal: \boxed{S_{\text{Mem}}} * \boxed{T_{\text{Var}}} * \boxed{T_{\text{Var2}}} \vdash \dot{\vDash\!\!\Rrightarrow}(\boxed{S_{\text{Mem}}} * \boxed{S_{\text{Var}}} * \boxed{S_{\text{Var2}}})$$

Here, just (horizontally) composing *i* and *ii* would not yield the result we want. With fancy updates, we can rephrase the problem as follows:

$$\begin{array}{rl} i: & \boxed{T_{\text{Var}}} \vdash \vDash\!\!\Rrightarrow_M \boxed{S_{\text{Var}}} \\ ii: & \boxed{T_{\text{Var2}}} \vdash \vDash\!\!\Rrightarrow_M \boxed{S_{\text{Var2}}} \end{array} \qquad goal: \boxed{T_{\text{Var}}} * \boxed{T_{\text{Var2}}} \vdash \vDash\!\!\Rrightarrow_M (\boxed{S_{\text{Var}}} * \boxed{S_{\text{Var2}}})$$

where M is a shorthand for $\boxed{S_{\text{Mem}}}$. With this rephrasing, we can directly (horizontally) compose *i* and *ii* to prove the *goal*.

7.3 Accessor Pattern

Now we see the final pattern, so-called an *accessor pattern* in Iris. For the sake of space, we focus on understanding the notion itself and its corresponding interpretation in the RCL side; specific example (along with its rules) is given in [Song and Lee 2024].

The accessor pattern is defined as follows with $\dot{\subseteq}$ notation:

$$P \mathrel{\dot{\subseteq}} Q \triangleq Q \mathrel{\Rrightarrow\!\!\!\circledast} (P * (P \mathrel{\Rrightarrow\!\!\!\circledast} Q))$$

Intuitively, $P \mathrel{\dot{\subseteq}} Q$ means that Q "contains" P but in a fancier way. A naive containment relation could be defined as $P \subseteq Q \triangleq \exists R.\, P * R \dashv\vdash Q$, but this is often too strong.

To see this, consider a module b that implements a `compare` method. In many module systems,

$$\exists c.\, b = (\texttt{compare} \mapsto \ldots) \oplus c \qquad\qquad \exists c.\, b \sqsupseteq\sqsubseteq (\texttt{compare} \mapsto \ldots) \oplus c$$

the proposition on the left would not hold, but instead, a weaker proposition on the right would hold. In other words, splitting a module results in a pair of modules that are definitionally different but behaviorally equivalent (and similarly for merging). That means, in RCL:

$$\boxed{\texttt{compare} \mapsto \ldots} \subseteq \boxed{b} \qquad\qquad \boxed{\texttt{compare} \mapsto \ldots} \mathrel{\dot{\subseteq}} \boxed{b}$$

the proposition on the left would not hold but the one on the right holds.

[7]In original Iris, fancy update modality has access to Iris *invariants* which is a much more powerful and sophisticated tool than the simple pattern shown here. Iris invariants are enabled by step-indexing, which we do not consider in RCL (see: §10).

$$
\begin{array}{ll}
\mathrm{fundef}(E) \triangleq \mathtt{Any} \to \mathtt{itree}\ E\ \mathtt{Any} & X|_{cond} \triangleq \text{if } cond \text{ holds, then } X \text{ else } \emptyset \\
\mathrm{E_V}(X) \triangleq \{\mathtt{Pick}\} \uplus \{\mathtt{Take}\} \uplus \{\mathtt{Obs}\ fn\ arg \mid fn \in \mathrm{String},\ arg \in \mathtt{Any}\}|_{X=\mathtt{Any}} & \\
\mathrm{E_L}(X) \triangleq \mathrm{E_V}(X) \uplus \{\mathtt{Call}\ fn\ arg | fn \in \mathrm{String},\ arg \in \mathtt{Any}\}|_{X=\mathtt{Any}} \uplus \{\mathtt{Put}\ a \mid a \in \mathtt{Any}\}|_{X=()} \uplus \{\mathtt{Get}\}|_{X=\mathtt{Any}} & \\
\mathrm{Mod} \triangleq \{(\mathtt{init}, \mathtt{funs}) \in \mathtt{Any} \times (\mathrm{alist\ String\ fundef}(\mathrm{E_L}))\} &
\end{array}
$$

$$
M \sqsubseteq_{\mathrm{beh}} M' \triangleq \mathrm{Beh}(M) \subseteq \mathrm{Beh}(M') \qquad M \sqsubseteq M' \triangleq \forall C \in \mathrm{Mod}\,.\, C \oplus M \sqsubseteq_{\mathrm{beh}} C \oplus M'
$$

$$
\begin{array}{l}
\mathrm{ObsEvent} \triangleq \{(\mathtt{Obs}\ fn\ arg, ret) \mid fn \in \mathrm{String},\ arg, ret \in \mathtt{Any}\} \\
\mathrm{Trace} \overset{\mathrm{coind}}{=} \{e :: tr \mid e \in \mathrm{ObsEvent}, tr \in \mathrm{Trace}\} \uplus \{\mathrm{Term}\ v \mid v \in \mathtt{Any}\} \uplus \{\mathrm{Diverge}\} \uplus \{\mathrm{Error}\} \uplus \{\mathrm{Partial}\} \\
\mathrm{Beh}(M) \in \mathbb{P}(\mathrm{Trace}) \triangleq \mathrm{beh}(\mathrm{concat}(M)) \qquad \mathrm{concat}(M) \in \mathtt{itree}\ \mathrm{E_V}\ \mathtt{Any} \triangleq \ldots \\
\mathrm{beh} \in \mathtt{itree}\ \mathrm{E_V}\ \mathtt{Any} \to \mathbb{P}(\mathrm{Trace}) \triangleq \lambda i.\ \{\mathrm{Partial}\} \cup \{\mathrm{Diverge}\}|_{i \in \mathrm{div}} \cup \\
\quad \mathbf{match}\ i\ \mathbf{with} \\
\quad \mid \mathbf{tau} \gg\!= k \Rightarrow \boxed{\mathrm{beh}(k())} \mid \mathbf{pick}(X) \gg\!= k \Rightarrow \bigcup_{x \in X} \boxed{\mathrm{beh}(k(x))} \mid \mathbf{take}(X) \gg\!= k \Rightarrow \bigcap_{x \in X} \boxed{\mathrm{beh}(k(x))} \\
\quad \mid \mathbf{obs}\ fn\ arg \gg\!= k \Rightarrow \bigcup_{ret \in \mathtt{Any}} (\mathtt{Obs}\ fn\ arg, ret) :: \boxed{\mathrm{beh}(k(ret))} \mid \mathbf{ret}\ v \Rightarrow \{\mathrm{Term}\ v\}\ \mathbf{end} \\
\mathrm{div} \in \mathbb{P}(\mathtt{itree}\ \mathrm{E_V}\ \mathtt{Any}) \overset{\mathrm{coind}}{=} \{\ \mathbf{tau} \gg\!= k \mid k() \in \mathrm{div}\ \} \cup \{\ \mathbf{pick}(X) \gg\!= k \mid \exists x \in X.\, k(x) \in \mathrm{div}\ \} \cup \\
\qquad \{\ \mathbf{take}(X) \gg\!= k \mid \forall x \in X.\, k(x) \in \mathrm{div}\ \}
\end{array}
$$

Fig. 13. Definitions of the module system and behavior (copied from CCR [Song et al. 2023] and modified).

This is useful when writing a specification that involves some form of polymorphism. For example, we may want to verify a `sort` function that serves any client module P that implements `compare`. In such cases, we can use the specification like this (simplified):

$$\boxed{\mathrm{Sort}_0} \vdash \Box((\boxed{\mathrm{Cmp}\ldots} \dot{\subseteq} P) \twoheadrightarrow (\boxed{\mathrm{Sort}_1 \ldots}))$$

where $\boxed{\mathrm{Cmp}\ldots} \dot{\subseteq} P$ indicates that P implements expected features (See: [Song and Lee 2024]).

8 A Concrete Instance of MRA

In this section, we give a concrete instance of MRA; this gives us confidence that the axioms of MRA are reasonable. To this end, we start by directly borrowing the module system of CCR [Song et al. 2023] (§8.1), and then (with some adjustments) extend it with the core operator and polysemic semantics (§8.2). We choose this module system for several reasons: (i) it already satisfies all the axioms except those for the core, (ii) it already supports angelic non-determinism, and (iii) its event-based semantics (all state accesses are via events) make the definition of $|-|$ straightforward.

We remark that the Fig. 13 is copied directly from CCR with the authors' permission, and slightly modified for presentation purposes. We include this figure for self-containedness.

8.1 Module System and Behavior

We first briefly explain the module system of CCR and how contextual refinement is defined. For interested readers, a detailed explanation can be found in the third section of Song et al. [2023].

CCR's module system heavily uses *interaction trees* [Xia et al. 2019], which can be understood as a fancy small-step semantics, parameterized over an event type E : **Set** $\to$ **Set** and a return type T. Interaction trees can take a silent step, return with a value of type T, or trigger an event in $E(X)$ for given X. It is also a monad, and we use the usual monad notations such as $\gg\!=$ for bind.

In the first row of Fig. 13 is the set Mod, consisting of an initial value for module-private state (`init`) and an association list with string (function names) as the key and function definitions as the value. Functions themselves reside in fundef($\mathrm{E_L}$), where the type `Any` can be seen as the set of every mathematical value, and the event type $\mathrm{E_L}$ is composed of (a) `Pick`/`Take` for (demonically/angelically) nondeterministically picking a value from a given set X, (b) `Obs` for observable events such as I/O, (c) `Call` for a function call, and (d) `Put`/`Get` to access the module's local state. These events have corresponding instructions in `itrees`, which triggers the matching event.

The second row defines the contextual refinement between Mod. Then the contextual refinement $\sqsubseteq$ is defined as expected (and also satisfies the three composition rules), where behavioral (whole-program) refinement is defined in terms of Beh$(-)$ explained in the last row of the figure.

The last row defines the notion of trace (Trace) and defines the behavior of a program as the set of possible traces. A trace is simply a finite or infinite sequence of observable events (in ObsEvent) with four terminal cases: normal return, silent divergence, termination with an error, and partial termination. Behavior of a whole program is computed by first concatenating all `Call` to the matching function definitions. Then, from the resulting `itree`, we extract a set of possible traces, via the predicate beh(itr). This predicate is a mixed inductive-coinductive definition, where a $\boxed{\text{solid box}}$ means a recursion and a dashed box means a corecursion.

8.2 Core Operator and Angelic Semantics

The core of a module simply takes the core of every function definition in the module. Then, the core of a function definition is defined as follows:

$$|f \in \text{fundef}(\mathrm{E_L})| \triangleq f[\texttt{Put}\_ \mapsto \texttt{triggerUB}, \texttt{Get} \mapsto \texttt{triggerUB}]$$

where it simply maps all the state access into **`triggerUB`**. This definition trivially satisfies MRAS-CORE-COMMUTE, MRAS-CORE-UNIT and MRAS-CORE-IDEM. Now we are left with the most interesting axiom, MRAS-CORE-INTRO. To get there, we need to extend the semantics to handle polysemic programs.

We begin with the existing semantics. When "booting" the program, the module system first checks whether there are multiple definitions and triggers **`UB`** if so. Then it continues the execution, and whenever a function is called, the module system finds the corresponding definition with findDef. We simply remove the check at the booting phase and extend findDef as follows:

$$\text{findDef}\ (\text{prog} \in \text{alist String fundef}(\mathrm{E_L}))\ (n \in \text{String}) \in \text{fundef}(\mathrm{E_L}) \triangleq \texttt{unwrapUB}(\text{alist\_find}\ n\ \text{prog})$$
$$\text{findDef prog}\ n \triangleq i \leftarrow \texttt{Take}(\{i \in \mathbb{N} \mid \text{prog}[i] = \text{Some}(n, \_)\}); (\text{prog}[i])\#2$$

Above is the original definition. The findDef looks at function definitions with the name n in the whole program (prog), triggers **`UB`** if a definition is missing, and executes a definition if there is. Below is the extended definition. It *angelically* picks (`Take`) a function definition with the matching name and executes it; note that the semantics of `Take` is defined as an intersection (Fig. 13).

With the above extension, we have the following theorem where the proof follows the idea described in §4.3 (details could be found in our formalization [Song and Lee 2024]):

THEOREM 8.1. *The extended module system satisfies all the axioms in MRA.*

9 Evaluation

Our Coq development comprises 19,054 SLOC (significant line of code) of Coq, where we reuse a large portion of existing developments in CCR. We also use a recently developed FreeSim library [Cho et al. 2023].

To assess the benefit of **(I)**, we have extracted the essence of the most interesting composition proof of CCR and Fair Operational Semantics [Lee et al. 2023] projects (CCR and FOS in the table, respectively). We also assess an imaginary verification scenario where both are combined together (CCR+FOS in the table).

Here, the main tactics mean applications of "small refinements", and auxiliary tactics mean all others: applications of commutativity, associativity, horizontal compositionality, vertical compositionality, and reflexivity. As the number shows, the proof in RCL is much shorter. In general, the

Table 3. Number of required tactics (in Coq) for refinement composition.

	#Main Tactics	#Auxiliary Tactics	#Total Tactics
CCR	12	17	29
CCR@RCL	12	2	14
FOS	3	22	25
FOS@RCL	3	2	5
CCR+FOS	2	33	35
CCR+FOS@RCL	2	2	4

difference gets more drastic as the example gets larger. Moreover, these three proofs have the same simple proof structure in RCL, unlike existing proofs.

We believe RCL will be especially useful in *heterogeneous* scenarios like CCR+FOS. Existing refinement frameworks tend to (if not always) impose a simple, uniform (*i.e.*, *homogeneous*) style in refinement composition (*e.g.*, in CAL and FOS, a client module always merges a library module in its refinement and in CCR, each module is completely independently refined). While this uniformity has been effective in taming some complexity of refinement composition, more interesting verification scenarios may necessitate mixing different styles, and the simplicity RCL brings will be highly beneficial there.

10 Related Work

We are not aware of any work that used separation logic assertions in this exact problem of composing refinements. Nevertheless, we distinguish RCL from works that share common keywords.

Separation logic. As discussed in §3.3, *relational* separation logics and RCL are complementary to each other. On the other hand, there are also *unary* separation logics that establish safety (or functional correctness) rather than refinement. These are not directly relevant here because they establish a weaker result than refinements.

CCR [Song et al. 2023] spans both refinement and separation logic in a novel way. The verification conditions in CCR are stated in terms of refinement and we expect them to be easily integrable with RCL. CCR supports reasoning for function pointers to some extent, but cannot verify Fig. 4 modularly as we did. Their verification of `rpt` function requires a global "specification table".

Call-wise reasoning. As we have seen in §3.2, the key benefit of $\Box$ in RCL is that it allows call-wise *refinement*. In a broader context, call-wise *reasoning* (the callee provides multiple specifications, and the caller specializes it) itself is a rather common theme across different contexts: *e.g.*, type systems, unary logic, and relational logic all support it through a simple universal quantification. The fundamental difference between them and us lies in the fundamental difference between the notion of types/Hoare Triples/Hoare quadruples and the notion of refinement: only the refinement has the notion of vertical compositionality.

Layered refinements and CAL. The names of the rules in Fig. 10 are modified for uniformity in presentation: LAYER-VCOMP to CONSEQ and LAYER-HCOMP to VCOMP. Their rule HCOMP is only used for composing refinements *inside* the module and does not appear in the granularity we presented. LAYER-TCOMP was missing in the original CAL and was added in its recent "modernization".

The modernization of CAL by Koenig [2020] removes many of its limitations and provides a much more general and elegant foundation for CAL. Its semantic domain is very close to that of CCR (§8) in that it supports dual (both angelic and demonic) non-determinism and also algebraic effects. It will be interesting future work to apply the extension we made in §8 to this setting. One final difference with CAL is that they use the simulation as a notion of underlying refinement and this appears in the notion of layered refinement.

Game Semantics and Linear Logic. Game semantics and linear logic are very closely related, and there has been some work that relate game semantics to refinement. Perhaps the most relevant to ours is RBGS (refinement-based game semantics) [Koenig and Shao 2020]. It used game semantics as a semantic domain for modules and imported some symbols from linear logic. However, the object these symbols capture is fundamentally different from ours. In our setting, a logical atom $\boxed{a}$ specifies a particular module with its function definitions, and in RBGS an atom denotes *signature* of functions. RBGS uses bang modality (!) to duplicate function calls and differs from $\square$ in RCL which concerns modules. In RCL, affinity is an important feature and it was not discussed in RBGS (linear logic usually does not have affinity).

Correspondence between logic and programming. While there is a plethora of literature that connects logic (especially with linear logic) with programming [Caires and Pfenning 2010; Frumin et al. 2022], they tend to establish correspondence with *programming features*. The notion of the module discussed in the paper is more coarse-grained and not specific to programming languages.

11 Conclusion and Future works

In this paper, we have identified an unexpected correspondence between the task of *composing* refinements and *separation* logic assertions, and developed RCL with the guidance of the correspondence. We believe the correspondence on its own is an idea worth spreading to the community.

We believe there is more to be discovered under the slogan that:

Separation and composition are two sides of the same coin.

We leave it as future work to further investigate derived constructs and patterns of separation logics other than Iris [Appel 2014; Ley-Wild and Nanevski 2013] and to connect them to RCL.

Acknowledgments

We thank Derek Dreyer for critical feedback on our presentation. We thank anonymous reviewers (including those in previous submissions), Ralf Jung, and Lennard Gäher for valuable comments on our draft. We thank Michael Sammler, Simon Spies, and Emanuele D'Osualdo for fruitful discussions on the early stage of this work. Youngju Song is the corresponding author.

References

Andrew W. Appel. 2014. *Program Logics for Certified Compilers.* Cambridge University Press. https://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/program-logics-certified-compilers

Ralph-Johan Back and Joakim Wright. 2012. *Refinement calculus: a systematic introduction.* Springer Science & Business Media.

Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. 2014. Verified Compilation for Shared-Memory C. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems (ESOP 2014).*

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR 2010 - Concurrency Theory*, Paul Gastin and François Laroussinie (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 222–236.

Arthur Charguéraud. 2023. Software Foundations Vol. 6, Chapter Wand. https://softwarefoundations.cis.upenn.edu/slf-current/Wand.html

Minki Cho, Youngju Song, Dongjae Lee, Lennard Gäher, and Derek Dreyer. 2023. Stuttering for Free. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 281 (oct 2023), 28 pages. https://doi.org/10.1145/3622857

Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. *Proc. ACM Program. Lang.* 1, ICFP, Article 24 (aug 2017), 30 pages. https://doi.org/10.1145/3110268

Edsger W. Dijkstra. 1972. *Chapter I: Notes on Structured Programming.* Academic Press Ltd., GBR, 1–82.

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP 2010 – Object-Oriented Programming*, Theo D'Hondt (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 504–528.

Dan Frumin, Emanuele D'Osualdo, Bas van den Heuvel, and Jorge A. Pérez. 2022. A Bunch of Sessions: A Propositions-as-Sessions Interpretation of Bunched Implications in Channel-Based Concurrency. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 155 (oct 2022), 29 pages. https://doi.org/10.1145/3563318

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*. 442–451.

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *Logical Methods in Computer Science* Volume 17, Issue 3 (Jul 2021). https://doi.org/10.46298/lmcs-17(3:9)2021

Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: a separation logic framework for verifying concurrent program optimizations. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–31.

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015)*.

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016)*.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018).

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. *ACM SIGPLAN Notices* 50, 1 (2015), 637–650.

Jérémie Koenig. 2020. Refinement-Based Game Semantics for Certified Components. https://flint.cs.yale.edu/flint/publications/koenig-phd.pdf

Jérémie Koenig and Zhong Shao. 2020. Refinement-Based Game Semantics for Certified Abstraction Layers. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science* (Saarbrücken, Germany) *(LICS '20)*. Association for Computing Machinery, New York, NY, USA, 633–647. https://doi.org/10.1145/3373718.3394799

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 205–217. https://doi.org/10.1145/3009837.3009855

Dongjae Lee, Minki Cho, Jinwoo Kim, Soonwon Moon, Youngju Song, and Chung-Kil Hur. 2023. Fair Operational Semantics. *Proc. ACM Program. Lang.* 7, PLDI, Article 139 (jun 2023), 24 pages. https://doi.org/10.1145/3591253

Xavier Leroy. 2006. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006)*.

Ruy Ley-Wild and Aleksandar Nanevski. 2013. Subjective auxiliary state for coarse-grained concurrency. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 561–574.

Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021. A Secure and Formally Verified Linux KVM Hypervisor. In *IEEE Symposium on Security and Privacy*. IEEE, 1782–1799. https://doi.org/10.1109/SP40001.2021.00049

Hongjin Liang and Xinyu Feng. 2016. A Program Logic for Concurrent Objects under Fair Scheduling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 385–399. https://doi.org/10.1145/2837614.2837635

David J. Pym, Peter W. O'Hearn, and Hongseok Yang. 2004. Possible worlds and resources: the semantics of BI. *Theoretical Computer Science* 315, 1 (2004), 257–305. https://doi.org/10.1016/j.tcs.2003.11.020 Mathematical Foundations of Programming Semantics.

Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. 2023. DimSum: A Decentralized Approach to Multi-Language Semantics and Verification. *Proc. ACM Program. Lang.* 7, POPL, Article 27 (jan 2023), 31 pages. https://doi.org/10.1145/3571220

Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2019. CompCertM: CompCert with C-Assembly Linking and Lightweight Modular Verification. *Proc. ACM Program. Lang.* 4, POPL, Article 23 (Dec. 2019), 31 pages. https://doi.org/10.1145/3371091

Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 2023. Conditional Contextual Refinement. *Proc. ACM Program. Lang.* 7, POPL, Article 39 (jan 2023), 31 pages. https://doi.org/10.1145/3571232

Youngju Song and Dongjae Lee. 2024. Refinement Composition Logic: Technical documentation and Coq development. https://github.com/CCR-project/RCL

Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 275–287. https://doi.org/10.1145/2676726.2676985

Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*. 377–390.

Yuting Wang, Ling Zhang, Zhong Shao, and Jérémie Koenig. 2022. Verified Compilation of C Programs with a Nominal Memory Model. *Proc. ACM Program. Lang.* 6, POPL, Article 25 (jan 2022), 31 pages. https://doi.org/10.1145/3498686

Niklaus Wirth. 1971. Program Development by Stepwise Refinement. *Commun. ACM* 14, 4 (apr 1971), 221–227. https://doi.org/10.1145/362575.362577

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2019. Interaction Trees: Representing Recursive and Impure Programs in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 51 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371119

Hongseok Yang. 2007. Relational separation logic. *Theor. Comput. Sci.* 375, 1-3 (2007), 308–334. https://doi.org/10.1016/j.tcs.2006.12.036