Conditional Contextual Refinement

YOUNGJU SONG, Seoul National University & MPI-SWS, Korea & Germany
MINKI CHO, Seoul National University, Korea
DONGJAE LEE, Seoul National University, Korea
CHUNG-KIL HUR*, Seoul National University, Korea
MICHAEL SAMMLER, MPI-SWS, Germany
DEREK DREYER, MPI-SWS, Germany

Much work in formal verification of low-level systems is based on one of two approaches: *refinement* or *separation logic*. These two approaches have complementary benefits: refinement supports the use of programs as specifications, as well as transitive composition of proofs, whereas separation logic supports conditional specifications, as well as modular ownership reasoning about shared state. A number of verification frameworks employ these techniques *in tandem*, but in all such cases the benefits of the two techniques remain separate. For example, in frameworks that use relational separation logic to prove contextual refinement, the relational separation logic judgment does not support transitive composition of proofs, while the contextual refinement judgment does not support conditional specifications.

In this paper, we propose **Conditional Contextual Refinement** (or **CCR**, for short), the first verification system to not only combine refinement and separation logic in a single framework but also to truly *marry* them together into a unified mechanism enjoying all the benefits of refinement and separation logic simultaneously. Specifically, unlike in prior work, CCR's refinement specifications are *both* conditional (with separation logic pre- and post-conditions) *and* transitively composable. We implement CCR in Coq and evaluate its effectiveness on a range of interesting examples.

1 INTRODUCTION

In recent years, great progress has been made on the problem of formally verifying correctness of complex, low-level software systems with machine-checked proof [Appel 2014; Gu et al. 2011, 2016; Klein et al. 2009]. Much work in this space is based on one of two approaches: *refinement* or *separation logic*. In this paper, we argue that these two approaches in fact have complementary benefits, and thus it is worth exploring how to marry them together in a single framework. We propose such a framework, which we call **Conditional Contextual Refinement (CCR)**, and we demonstrate its utility on a range of representative examples. But before we get to CCR, let us begin with a brief overview of what refinement and separation logic bring to the table.

1.1 Refinement vs. Separation Logic

Common to essentially all approaches to program verification is the idea that we have a program (or program component) we wish to verify—call it the *implementation*—and we wish to show that it satisfies some formal *specification*. However, two key axes along which different verification methods differ—and in particular, how methods based on refinement vs. separation logic differ—are:

(1) how the *specification* is formalized, and
(2) the sense in which the verification method is *compositional*.

Separation logic. *Separation logic* is an extension of Hoare logic; as such, it specifies program *components* C (rather than whole programs) using a *precondition* P and *postcondition* Q, written

*Chung-Kil Hur is the corresponding author.
Authors' addresses: Youngju Song, Seoul National University & MPI-SWS, Korea & Germany, youngju@mpi-sws.org; Minki Cho, Seoul National University, Korea, minki.cho@sf.snu.ac.kr; Dongjae Lee, Seoul National University, Korea, dongjae.lee@sf.snu.ac.kr; Chung-Kil Hur*, Seoul National University, Korea, gil.hur@sf.snu.ac.kr; Michael Sammler, MPI-SWS, Germany, msammler@mpi-sws.org; Derek Dreyer, MPI-SWS, Germany, dreyer@mpi-sws.org.

$\{P\}\ C\ \{Q\}$. The precondition P specifies the *assumption* that C makes about its program context and the starting state in which it is executed, and the postcondition Q specifies the *guarantee* C makes about the final state after it has executed. A key benefit of this approach is that it enables us to verify the correctness of a component C even if C only satisfies a *conditional specification*—*i.e.,* C only behaves correctly *under certain conditions* (say, when x is a pointer to a well-sorted linked list, or when some initialization routine has been run before C is executed).

In terms of compositionality, separation logic goes beyond Hoare logic by additionally equipping the assertions P and Q with the ability to talk about *ownership* of resources (*e.g.,* memory) that are transferred to C *from* its context (in P) and back *to* its context (in Q). This in turn is essential for supporting *modular reasoning about shared state*: when C operates on a piece of state (*e.g.,* memory) that is shared with its program context, the ownership model of separation logic assertions can dramatically simplify reasoning about potential interference between C and its context. And even ignoring the program context, ownership reasoning can also be helpful in modularly decomposing the verification of C itself—*e.g.,* if, say, C spawns several threads manipulating shared state, each of which we wish to verify separately without considering all concurrent interleavings.

Refinement. In contrast, *refinement* formalizes the specification of a program (or program component[1]) as itself another (higher-level) program: in order to verify that the implementation program I satisfies the specification represented by the program S, we show that the set of possible behaviors exhibited by I *refines* (*i.e.,* is included in) the set of possible behaviors exhibited by S, written $I \sqsubseteq S$. One key benefit of refinement is that, by representing the specification S as a program rather than as a logical formula, refinement supports verification even in cases where we either (1) lack a logic rich enough to express I's behavior or (2) want to express the end-to-end result of our verification in terms that an external user can understand (*i.e.,* using code, rather than an assertion in a bespoke logic known only to verification experts).

In terms of compositionality, an advantage of refinement is *transitive composition of proofs*: one can conduct the verification of $I \sqsubseteq S$ compositionally by introducing n *mediating* programs $M_1, \ldots, M_n$, which *gradually* refine the behavior of the program I until it reaches the specification S—*i.e.,* $I \sqsubseteq M_1 \sqsubseteq \ldots \sqsubseteq M_n \sqsubseteq S$; then, by transitivity, one obtains $I \sqsubseteq S$. The gradual refinement afforded by transitivity lets one focus on orthogonal aspects of I separately—*e.g.,* one step of a refinement proof might deal with how I represents a data structure in memory, while another step might focus on the higher-level functional correctness of I's algorithm. Moreover, transitivity supports proof reuse, since refinement proofs can share "common legs"—the proofs of $I_1 \sqsubseteq S, \ldots, I_n \sqsubseteq S$ might all go through a common mediating M (such that $I_1 \sqsubseteq M, \ldots, I_n \sqsubseteq M$), reusing the proof that $M \sqsubseteq S$.

In summary:

- Separation logic supports *conditional specifications* and *modular reasoning about shared state.*
- Refinement supports *programs as specifications* and *transitive composition of proofs.*

It is therefore quite natural to ask:

Can we marry the complementary benefits of refinement and separation logic in one framework?

Marrying separation logic and refinement. We are certainly not the first to ask this question. In particular, a number of verification frameworks [Frumin et al. 2021a; Gäher et al. 2022; Liang and Feng 2016; Turon et al. 2013] have employed separation logic in conjunction with refinement. However, what the existing work in this space has *not* done so far is to truly synthesize separation logic and refinement into a unified method providing all the benefits each method enjoys individually.

[1]Some refinement-based approaches support verification of modular program components, while others concern only whole programs. We use the term "program" here loosely to refer to both.

```
(* module I_Map *)
private data := NULL

def init(sz: int) ≡
  data := calloc(sz)

def get(k: int): int ≡
  return *(data + k)

def set(k: int, v: int) ≡
  *(data + k) := v

def set_by_user(k: int) ≡
  set(k, input())
```

```
(* module A_Map *)
private map := (fun k => 0)

def init(sz: int) ≡
  skip

def get(k: int): int ≡
  return map[k]

def set(k: int, v: int) ≡
  map := map[k ← v]

def set_by_user(k: int) ≡
  set(k, input())
```

```
(* pre & postconditions S_Map *)
    ∀sz. {pending}
           init(sz)
         {∗_{k∈[0,sz)} k ↦Map 0}
  ∀k v. {k ↦Map v}
           get(k)
         {r. r = v ∧ k ↦Map v}
∀k w v. {k ↦Map w}
           set(k,v)
         {k ↦Map v}
  ∀k w. {k ↦Map w}
           set_by_user(k)
         {∃v. k ↦Map v}
```

Fig. 1. An implementation module $I_{\mathtt{Map}}$, its abstraction $A_{\mathtt{Map}}$, and its pre- and postconditions.

Consider, for example, the main judgment in Simuliris [Gäher et al. 2022]: it takes the form $\{P\}\ I \le S\ \{Q\}$—where here P and Q can talk about (and relate) the states of both I and S. This *relational separation logic* judgment has the advantage that it lets one place precise ownership-based conditions on when I refines S. Furthermore, for certain restricted choices of P and Q, this judgment implies *contextual* refinement ($I \sqsubseteq_{\mathrm{ctx}} S$), a strong property that says I refines S when placed in an arbitrary (well-formed) program context C. Hence, on the one hand, Simuliris uses relational separation logic as an effective technique for establishing contextual refinement. Yet the benefits of separation logic and refinement here are kept separate. The relational separation logic judgment $\{P\}\ I \le S\ \{Q\}$ is a *conditional* refinement, but it does not enjoy transitive composability; in contrast, the contextual refinement $I \sqsubseteq_{\mathrm{ctx}} S$ *is* transitively composable but it is also *un*-conditional (*i.e.,* it does not support placing precise conditions on the program context).

In this paper, we propose **Conditional Contextual Refinement** (or **CCR**, for short), the first verification system to not only combine refinement and separation logic in a single framework but also *fuse* their complementary benefits together in a unified mechanism. Specifically, unlike in prior work, CCR's refinement specifications are *both* conditional (with separation logic pre- and post-conditions) *and* transitively composable. Furthermore, CCR is fully mechanized in the Coq proof assistant. To give a sense of what CCR is capable of, we now present a concrete motivating example.

1.2 Motivating Example

Consider the verification of a simple key-value storage module depicted in Fig. 1. The implementation $I_{\mathtt{Map}}$ uses a pointer `data` to store an array mapping the integer keys to their values. This array is initially `NULL` and initialized by the function `init(sz: int)` with an array consisting of `sz` zeros (returned by `calloc(sz)`). The functions `get` and `set` retrieve and update, respectively, the entry at a given index in the array. Finally, `set_by_user` updates an entry with the value given by the user (*i.e.,* that obtained via the system call *input*`()`).

Now we consider and compare two kinds of specifications of $I_{\mathtt{Map}}$, one using separation logic and the other using refinement. First, in separation logic, we can introduce a points-to predicate $\mathsf{k} \mapsto_{\mathtt{Map}} \mathsf{v}$, asserting that the key k is a valid entry of the map and stores the value v. With $\mathsf{k} \mapsto_{\mathtt{Map}} \mathsf{v}$, the functions of $I_{\mathtt{Map}}$ can be specified in terms of *pre- and postconditions* as shown in the rightmost column of Fig. 1. Here, `init` allocates $\mathsf{k} \mapsto_{\mathtt{Map}} 0$ for each entry in the map. Note that the *exclusive token* $\boxed{pending}$ is consumed when calling `init` and thus encodes that `init` can only be called

once. Then `get(k)` returns `v` given $\texttt{k} \mapsto_{\texttt{Map}} \texttt{v}$, and `set(k,v)` updates $\texttt{k} \mapsto_{\texttt{Map}} \texttt{w}$ to the new value `v`. Note that `set_by_user(k)` updates $\texttt{k} \mapsto_{\texttt{Map}} \texttt{w}$ to an unknown value since any value can be given by the user.

On the plus side, it is well known that this kind of separation logic specification offers powerful modular reasoning principles for verifying clients of $I_{\texttt{Map}}$ [Jung et al. 2018]. On the minus side, the separation logic spec does not fully capture the behavior of the code itself. In particular, the above specification of `set_by_user(k)` does not capture how the function interacts with the user.

Alternatively, under the refinement approach, we can specify $I_{\texttt{Map}}$ using a more abstract program $A_{\texttt{Map}}$ (the middle column of Fig. 1), which fully captures the observable behavior of $I_{\texttt{Map}}$. Specifically, this abstraction $A_{\texttt{Map}}$ adequately retains the implementation's interactions with its environment (*i.e.,* the system call *input*()) while at the same time abstracting away internal implementation details (*i.e.,* it abstracts the low-level memory-based representation of the map into a high-level representation as a mathematical function from `int` to `int`).

On the plus side, thanks to transitivity, the refinement approach allows us to verify $I_{\texttt{Map}}$ incrementally, in a stepwise fashion. For example, as we will see shortly, a refinement proof of $I_{\texttt{Map}}$ against $A_{\texttt{Map}}$, denoted $I_{\texttt{Map}} \sqsubseteq A_{\texttt{Map}}$, can be established by introducing an intermediate abstraction $M_{\texttt{Map}}$ and transitively composing the proofs of $I_{\texttt{Map}} \sqsubseteq M_{\texttt{Map}}$ and $M_{\texttt{Map}} \sqsubseteq A_{\texttt{Map}}$. On the minus side, however—and this is a big minus—the refinement doesn't hold! To be specific, it only holds *under the condition* that `init` is called only once, and that the other functions are only called after the call to `init` and with index arguments that are in range. (Otherwise, the refinement would be broken, since functions in $I_{\texttt{Map}}$ would raise errors while those in $A_{\texttt{Map}}$ would not.) This condition is of course precisely what the separation logic specification for $I_{\texttt{Map}}$ enforces.

Marrying separation logic and refinement. As the above example makes clear, separation logic and refinement are truly complementary methods. Separation logic supports the enforcement of precise conditions on how a module is used, while refinement supports incremental stepwise verification of the module (via transitivity) against a specification represented as code. How can we marry these advantages in one mechanism?

To achieve this, we propose the notion of *conditional contextual refinement* (CCR). At a high level, the idea of CCR is natural: we develop a notion of refinement that allows the imposition of precise separation-logic conditions under which the refinement holds. For instance, in our motivating example, we will be able to prove $S_{\texttt{Map}} \vdash I_{\texttt{Map}} \sqsubseteq A_{\texttt{Map}}$, which establishes that $I_{\texttt{Map}}$ refines $A_{\texttt{Map}}$ under the condition that the module is used according to the separation logic spec $S_{\texttt{Map}}$. This conditional refinement relation satisfies several key desiderata.

First, CCR's conditional refinement supports *modular reasoning* as in separation logic. For example, suppose that we have a client module of `Map`—call it `CL` for "client"—with an implementation $I_{\texttt{CL}}$, an abstraction $A_{\texttt{CL}}$, and conditions $S_{\texttt{CL}}$. Then we want to modularly verify conditional refinement for `CL` only relying on the separation logic specification $S_{\texttt{Map}}$ of `Map` and *without* needing to reason directly about `Map`'s implementation $I_{\texttt{Map}}$ or abstraction $A_{\texttt{Map}}$. In other words, we want to prove $S_{\texttt{Map}} \cup S_{\texttt{CL}} \vdash I_{\texttt{CL}} \sqsubseteq A_{\texttt{CL}}$, which is then composed with $S_{\texttt{Map}} \vdash I_{\texttt{Map}} \sqsubseteq A_{\texttt{Map}}$ to obtain $S_{\texttt{Map}} \cup S_{\texttt{CL}} \vdash I_{\texttt{CL}} \circ I_{\texttt{Map}} \sqsubseteq A_{\texttt{CL}} \circ A_{\texttt{Map}}$ (here, $\circ$ denotes the linking operator on modules). This kind of composition is called *horizontal composition*. Moreover, such modular reasoning should be allowed even in the presence of mutual dependence/recursion between modules.

Second, CCR's conditional refinement allows incremental verification via transitive composition (sometimes known in the literature as *vertical composition*). For example, consider proving $S_{\texttt{Map}} \vdash I_{\texttt{Map}} \sqsubseteq A_{\texttt{Map}}$ via the following intermediate abstraction $M_{\texttt{Map}}$, which simply adds the field

size and the range checking code **assume**(0 ≤ k < size) to A_{Map}:

```
(* module M_Map *)
private map := (fun k => 0)
private size := 0
```

```
def init(sz: int) ≡
  size := sz

def get(k: int): int ≡
  assume(0 ≤ k < size)
  return map[k]
```

```
def set(k: int, v: int) ≡
  assume(0 ≤ k < size)
  map := map[k ⟵ v]

def set_by_user(k: int) ≡
  set(k, input())
```

Here, the command **assume**(0 ≤ k < size) triggers *undefined behavior*, rendering all possible observable behaviors, if the index is out of range. This facilitates a modular decomposition of the refinement $S_{\mathsf{Map}} \vdash I_{\mathsf{Map}} \sqsubseteq A_{\mathsf{Map}}$ as follows. Thanks to the range checking, the first refinement between I_{Map} and M_{Map} holds as long as `init` is called at most once, which is captured by the following simple conditions, which we will denote as S^0_{Map}:

$$\begin{array}{lll} \forall \mathtt{sz}.\ \{\boxed{\mathit{pending}}\}\ \mathtt{init(sz)} & & \{\top\} \\ \forall \mathtt{k}\ \mathtt{v}. & \{\top\}\ \mathtt{get(k), set(k,v), set\_by\_user(k)} & \{\top\} \end{array}$$

Specifically, the verification of $S^0_{\mathsf{Map}} \vdash I_{\mathsf{Map}} \sqsubseteq M_{\mathsf{Map}}$ amounts to only proving data abstraction from the memory-based representation of the map into the function-based representation assuming that `init` is called at most once, but *without* bothering to prove that the module satisfies the modular reasoning principles of S_{Map}. Then, the second refinement $S_{\mathsf{Map}} \vdash M_{\mathsf{Map}} \sqsubseteq A_{\mathsf{Map}}$ actually amounts to proving that the module satisfies S_{Map} but based on the higher-level function-based representation rather than the lower-level memory-based representation. Vertical composition then means that $S^0_{\mathsf{Map}} \vdash I_{\mathsf{Map}} \sqsubseteq M_{\mathsf{Map}}$ and $S_{\mathsf{Map}} \vdash M_{\mathsf{Map}} \sqsubseteq A_{\mathsf{Map}}$ should be composable to yield $S_{\mathsf{Map}} \vdash I_{\mathsf{Map}} \sqsubseteq A_{\mathsf{Map}}$.

In order to cleanly formalize our notion of conditional refinement, as well as prove its horizontal and vertical composition properties, CCR employs *separation logic wrappers*, a novel mechanism for "operationalizing" the enforcement of separation logic specs. Concretely, CCR defines a notion of a *wrapper*, written $\langle S \vdash M \rangle$, which converts M into a module that "self-enforces" the pre- and postconditions of S at the points where M interacts with its program context. With these wrappers in hand, CCR then can define conditional refinement as just a mode of use of the standard notion of contextual refinement, denoted $\sqsubseteq_{\mathsf{ctx}}$, between wrapped modules:

$$S \vdash I \sqsubseteq A \quad \triangleq \quad I \sqsubseteq_{\mathsf{ctx}} \langle S \vdash A \rangle$$

This allows us to easily establish the horizontal and vertical composition of conditional refinement by leveraging the fact that contextual refinement enjoys these properties by construction. Concerning our example, we can verify I_{Map} via the following chain of refinements, whose transitive composition follows directly from the transitivity of contextual refinement:

$$I_{\mathsf{Map}} \quad \sqsubseteq_{\mathsf{ctx}} \quad \langle S^0_{\mathsf{Map}} \vdash M_{\mathsf{Map}} \rangle \quad \sqsubseteq_{\mathsf{ctx}} \quad \langle S_{\mathsf{Map}} \vdash A_{\mathsf{Map}} \rangle$$

Of course, this leaves the question of how exactly to define these separation logic wrappers. The key challenge in defining such a wrapper is that separation logic reasoning involves non-trivial cooperation across interacting modules, such as a transfer of resource ownership, which is not readily observable in the program state. To tackle this challenge, we use a combination of *angelic* and *demonic* non-determinism which is often called *dual non-determinism*. In prior work, this was mainly studied in the context of game semantics [Back and Wright 2012; Koenig and Shao 2020]. In CCR, we apply this idea instead as a way to express resource transfer between the caller and callee of a function, using non-deterministic choices of both parties.

In summary, we develop the theory of CCR, which fully fuses together the benefits of refinement and separation logic in a unified mechanism. In this paper, we present the ideas and formalization

of CCR in detail, along with a variety of motivating examples (and others in the supplementary material) involving shared-memory reasoning, mutual recursion, function pointers, and termination.

Structure of the paper. The rest of the paper is structured as follows. We first give an overview of the main ideas of CCR by showing (semi-formally) how it applies to our motivating example (§2). Next, we explain how CCR is formalized as a general verification framework. This is done in two steps: we first present a general, language-agnostic module system we developed (§3), and then develop the key definitions and meta-theory of the CCR framework (§4). The framework presented in §4 is self-contained and sufficient to handle our motivating example. However, the full-fledged CCR framework has additional features, which we motivate with further examples (§5). The formalization for the full framework is given in the appendix [Author(s) 2022]. Finally, we present an evaluation for our development (§6), discuss related work (§7) and future directions (§8).

2 MAIN IDEAS OF CCR

This section is devoted to explaining how CCR, in particular the wrapper $\langle S \vdash M \rangle$, works. For this, we will show (*i*) how pre- and postconditions are encoded in $\langle S \vdash M \rangle$; (*ii*) how the encoding affects the simulation argument for proving refinement; and (*iii*) how we formalize them more generally in terms of contextual refinement. For presentation purposes, we first discuss how *pure* conditions (*i.e.,* without involving separation logic) are encoded (§2.1) and then more generally how separation logic conditions are encoded (§2.2).

2.1 Stateless Conditional Refinement

To see how one can encode pure conditions, consider the following two functions:

```
(* module I_Sq *)                (* module A_Sq *)                (* module I_Main *)              (* module A_Main *)
def is_sq(x: int): int ≡          def is_sq(x: int): int ≡          def main() ≡                  def main() ≡
  if (x < 0) error()                                                  var x := 16                   var x := 16
  var r := ...                      var r := ...                      var r := is_sq(x)             var r := is_sq(x)
  return r                          return r                          output(r)                     output(1)
```

On the left, `is_sq(x)` checks if `x` is a square number (elided here in the `...` part), and returns 1 if so and 0 otherwise. Also, in its implementation $I_{\mathtt{Sq}}$, if `x` is negative, it calls the system call *error*, but this check is eliminated in its abstraction $A_{\mathtt{Sq}}$. On the right, `main` invokes `is_sq(16)` and outputs its result, which is abstracted to 1 in its abstraction $A_{\mathtt{Main}}$ since 16 is in fact a square number.

Now, we see whether the standard simulation argument for the (unconditional) refinement $I_{\mathtt{Sq}} \sqsubseteq_{\mathrm{ctx}} A_{\mathtt{Sq}}$ works. For this, we need to show that for any value for the argument `x`, the two codes `is_sq(x)` in $I_{\mathtt{Sq}}$ and $A_{\mathtt{Sq}}$ are related by the simulation relation $\precsim$ defined by the following rules:

$$\text{(STL)}\ \frac{T \hookrightarrow T' \quad T' \precsim S}{T \precsim S} \qquad \text{(STR)}\ \frac{S \hookrightarrow S' \quad T \precsim S'}{T \precsim S} \qquad \text{(CALL)}\ \frac{\forall w.\ \mathtt{r}:=w; T \precsim \mathtt{r}:=w; S}{\mathtt{r}:=f(\vec{v}); T \precsim \mathtt{r}:=f(\vec{v}); S} \qquad \text{(RET)}\ \frac{}{\mathbf{return}\ v \precsim \mathbf{return}\ v}$$

Here, $T \hookrightarrow T'$ denotes the silent (deterministic) step of the code (or program state) according to its small-step operational semantics. The first two rules say that one can freely take silent steps in either side and show simulation between the resulting states. The third rule says that at a function call point both sides should call the same function f with the same arguments $\vec{v}$ and the resulting states for the same arbitrary return value w should be simulated. The last rule says that at a return point both sides should return the same value. As easily seen, the simulation does not hold when `x` is negative since *error*() is called in $I_{\mathtt{Sq}}$ but not in $A_{\mathtt{Sq}}$.

To make the refinement hold, we can consider the following condition S^0_{Sq}:

$$\forall \mathtt{x}.\ \{\mathtt{x} \geq 0\}\ \mathtt{is\_sq(x)}\ \{\top\}$$

Moreover, to provide a modular reasoning principle to other modules, we can strengthen it to S_{Sq}:

$$\forall \mathtt{x}.\ \{\mathtt{x} \geq 0\}\ \mathtt{is\_sq(x)}\ \{r.\ r = 1 \iff \exists i.\ \mathtt{x} = i * i\}$$

Encoding conditions. Now we see how we can define the wrapper $\langle\, S_{\mathsf{Sq}} \vdash A_{\mathsf{Sq}} \,\rangle$ following the approach of Refinement Calculus [Back and Wright 2012]. The idea is to encode the pre- and postconditions via **assume** and **assert** statements. Concretely, $\langle\, S^0_{\mathsf{Sq}} \vdash A_{\mathsf{Sq}} \,\rangle$ and $\langle\, S_{\mathsf{Sq}} \vdash A_{\mathsf{Sq}} \,\rangle$ are encoded as follows.

```
def is_sq(x: int): int ≡
  assume(x ≥ 0)
  var r := ...
  assert(⊤)
  return r
```

```
def is_sq(x: int): int ≡
  assume(x ≥ 0)
  var r := ...
  assert(r = 1 ⟺ ∃i. x = i * i)
  return r
```

The simulation rules for **assume** and **assert** are given as follows.

$$\text{(ASMR)}\ \frac{P \implies T \lesssim S}{T \lesssim \mathbf{assume}(P); S} \qquad \text{(ASTR)}\ \frac{P \qquad T \lesssim S}{T \lesssim \mathbf{assert}(P); S} \qquad \text{(ASML)}\ \frac{P \qquad T \lesssim S}{\mathbf{assume}(P); T \lesssim S} \qquad \text{(ASTL)}\ \frac{P \implies T \lesssim S}{\mathbf{assert}(P); T \lesssim S}$$

Proving refinement incrementally. With the simulation rules, we can prove $I_{\mathsf{Sq}} \sqsubseteq_{\mathsf{ctx}} \langle\, S^0_{\mathsf{Sq}} \vdash A_{\mathsf{Sq}} \,\rangle$ and $\langle\, S^0_{\mathsf{Sq}} \vdash A_{\mathsf{Sq}} \,\rangle \sqsubseteq_{\mathsf{ctx}} \langle\, S_{\mathsf{Sq}} \vdash A_{\mathsf{Sq}} \,\rangle$, which can then be combined to yield $I_{\mathsf{Sq}} \sqsubseteq_{\mathsf{ctx}} \langle\, S_{\mathsf{Sq}} \vdash A_{\mathsf{Sq}} \,\rangle$. The former can be proven as follows: for any value of x, by (ASMR) with **assume**(x ≥ 0), we can assume the value x is non-negative; by (STL), we can skip the if-statement without calling *error*() since x ≥ 0; by applying (STL) and (STR) in lock steps without any serious reasoning we can reach after **var** r := ... with the same value for r since the codes in both sides are identical; by (ASTR), we can simply skip **assert**(⊤); finally we can conclude by (RET) since the return values are the same. Note that here the only non-trivial verification is to prove the absence of *error*() relying on the assumption x ≥ 0.

The latter refinement $\langle\, S^0_{\mathsf{Sq}} \vdash A_{\mathsf{Sq}} \,\rangle \sqsubseteq_{\mathsf{ctx}} \langle\, S_{\mathsf{Sq}} \vdash A_{\mathsf{Sq}} \,\rangle$ is proven as follows: for any value for x, by (ASMR) we can assume x ≥ 0 and then by (ASML) we need to prove x ≥ 0, which immediately follows from the assumption we just made; similarly as before we can easily reach after **var** r := ... with the same value for r by applying (STL) and (STR) in lock steps; by (ASTL) we can skip **assert**(⊤); then by (ASTR) we need to prove r = 1 ⟺ ∃*i*. x = *i* ∗ *i* holds, which is basically to prove that the code in ... correctly checks whether x is a square when x ≥ 0; finally we can conclude by (RET). Here the only non-trivial verification is to prove the **assert** statement by (ASTR), which essentially amounts to verification of A_{Sq} against S_{Sq} in Hoare logic.

Using pre- and postconditions modularly. Now for $S_{\mathtt{Main}} = \{\ \{\top\}\ \mathtt{main()}\ \{\top\}\ \}$, we can modularly prove that $I_{\mathtt{Main}}$ refines $\langle\, S_{\mathsf{Sq}} \cup S_{\mathtt{Main}} \vdash A_{\mathtt{Main}} \,\rangle$, which is defined as follows:

```
def main() ≡
  assume(⊤); var x := 16
  assert(x ≥ 0); var r := is_sq(x); assume(r = 1 ⟺ ∃i. x = i * i)
  output(1); assert(⊤)
```

Note that the precondition of is_sq is **assert**ed before the call and its postcondition is **assume**d after the call, which means that during the simulation proof of $I_{\mathtt{Main}} \sqsubseteq_{\mathsf{ctx}} \langle\, S_{\mathsf{Sq}} \cup S_{\mathtt{Main}} \vdash A_{\mathtt{Main}} \,\rangle$, by (ASTR) the condition x ≥ 0 needs to be *proven*, which trivially holds since x = 16, and by

(ASMR) the condition $\mathtt{r} = 1 \iff \exists i.\ \mathtt{x} = i * i$ can be *assumed*, from which $\mathtt{r} = 1$ follows since $\mathtt{x} = 16 = 4 * 4$ and thus both sides call $\mathit{output}(1)$.

Cancelling wrappers. Now we see how we can compose the modular verification results and eliminate the wrappers by cancelling them against each other. The first step is to horizontally compose the two refinements $I_{\mathsf{Sq}} \sqsubseteq_{\mathsf{ctx}} \langle\, S_{\mathsf{Sq}} \vdash A_{\mathsf{Sq}} \,\rangle$ and $I_{\mathsf{Main}} \sqsubseteq_{\mathsf{ctx}} \langle\, S_{\mathsf{Sq}} \cup S_{\mathsf{Main}} \vdash A_{\mathsf{Main}} \,\rangle$, as shown in the first of the two refinements below:

$$I_{\mathsf{Sq}} \circ I_{\mathsf{Main}} \sqsubseteq_{\mathsf{beh}} \langle\, S_{\mathsf{Sq}} \vdash A_{\mathsf{Sq}} \,\rangle \circ \langle\, S_{\mathsf{Sq}} \cup S_{\mathsf{Main}} \vdash A_{\mathsf{Main}} \,\rangle \sqsubseteq_{\mathsf{beh}} A_{\mathsf{Sq}} \circ A_{\mathsf{Main}}$$

Note that $I_{\mathsf{Sq}} \circ I_{\mathsf{Main}}$ is a closed program so we use whole program refinement $\sqsubseteq_{\mathsf{beh}}$ (defined in §3.2) instead of contextual refinement $\sqsubseteq_{\mathsf{ctx}}$. The second step is to eliminate the wrappers by automatically cancelling them against each other, as shown in the second refinement above. This second refinement can be proven *automatically* by observing that in the linked program $\langle\, S_{\mathsf{Sq}} \vdash A_{\mathsf{Sq}} \,\rangle \circ \langle\, S_{\mathsf{Sq}} \cup S_{\mathsf{Main}} \vdash A_{\mathsf{Main}} \,\rangle$, each $\mathtt{assume}(P)$ is immediately after a corresponding $\mathtt{assert}(P)$ (except for the trivial ones with $\top$ at the very beginning and end). Thus, both can be eliminated via repeatedly applying the following refinement:

$$K[\mathtt{assert}(P); \mathtt{assume}(P)] \quad \sqsubseteq_{\mathsf{beh}} \quad K[\mathtt{skip}]$$

This is easily provable by applying (ASTL) followed by (ASML) at the condition statements and otherwise applying (STL) and (STR) in lock steps. CCR includes the machinery to automatically apply this cancellation and thus eliminate the wrappers. We call this the **Assumption Cancellation Theorem (ACT)** and formally describe it in Theorem 4.1 (§4.2).

2.2 Stateful Conditional Refinement via Separation Logic

We now describe how CCR extends the technique described in the previous section to separation logic conditions. In particular, we will see how to define the wrapper $\langle\, S \vdash A \,\rangle$ where S contains the separation logic pre- and postconditions of the module A. The high-level idea is simple: We define more elaborate versions of **assume** and **assert**—which we call ASSUME and ASSERT—that work on separation logic assertions instead of pure propositions. However, the devil is in the details, so we first need to give a short introduction to the model of separation logic before we can introduce ASSUME and ASSERT.

Separation logic reasoning. Consider the example $I_{\mathsf{Map}} \sqsubseteq_{\mathsf{ctx}} \langle\, S^0_{\mathsf{Map}} \vdash M_{\mathsf{Map}} \,\rangle$ from §1.2, where $\langle\, S^0_{\mathsf{Map}} \vdash M_{\mathsf{Map}} \,\rangle$ is defined as follows:

```
(* ⟨ S⁰_Map ⊢ M_Map ⟩ *)
private map := (fun k => 0)   def init(sz: int) ≡                    ...
private size := 0               var (frs, ctx) := (ε, ε)
private mrs: Σ := ε             ASSUME(pending); size := sz; ASSERT(⊤)
```

The separation logic specification S^0_{Map} requires the caller to provide the resource *pending* when calling `init` and it is consumed by the specification (*i.e.,* not given back in the postcondition). The key property of the resource *pending* is that it cannot be duplicated and thus there can be at most one *pending* in existence. With this in mind, the verification goes as follows. Initially, the module $\langle\, S^0_{\mathsf{Map}} \vdash A_{\mathsf{Map}} \,\rangle$ does not own any resource. The first time `init` is called, the module adds the resource *pending* to its privately owned state (since it does not need to give it back). If `init` is called again, the module observes that there are two *pending*s: one given in the precondition and one in the ownership of M. However, we know that there can be at most one *pending* and thus there is a contradiction that concludes the refinement proof. In other words, we just need to prove the refinement assuming that `init` is called at most once.

Model of separation logic. With this intuition at hand, we can now consider how the separation logic reasoning above works formally. Consider how the resource *pending* is defined. In separation logic, resources are usually modeled as (variants of) *Partial Commutative Monoids* (PCMs). For our purposes, a PCM Σ is a set equipped with a commutative and associative binary operator $+$ on Σ, called *addition*, an identity element ε, and a validity predicate $\mathcal{V}$ on Σ satisfying (i) $\mathcal{V}(\varepsilon)$ and (ii) $\forall a, b.\, \mathcal{V}(a+b) \implies \mathcal{V}(a)$ (aka monotonicity). Since PCM is closed on the Cartesian product, each module can pick its own PCM and the whole system can be instantiated with the global PCM which is a product of all PCMs used.

A PCM Σ yields a notion of separation logic proposition which we call $\mathbf{rProp}_\Sigma$[2] , and it is defined simply as $\Sigma \to \mathbf{Prop}$. We define the logical connectives on $\mathbf{rProp}_\Sigma$ following Jung et al. [2018]. Specifically, for an arbitrary $\mathbf{rProp}_\Sigma$ P and Q, a resource a, an element x of an arbitrary set, and a $\mathbf{Prop}$ R, the connectives in our example have the following definitions (using $r \geq a \triangleq \exists b. r = a + b$):

$$\boxed{a} \triangleq \lambda r.\, r \geq a \qquad P * Q \triangleq \lambda r.\, \exists\, a\, b.\, r = a + b \wedge P\, a \wedge Q\, b \qquad \exists x.\, P \triangleq \lambda r.\, \exists x.\, P\, r \qquad \ulcorner R \urcorner \triangleq \lambda \_.\, R$$

For our example, we use the PCM with the following elements:

$$\mathit{pending} \mid \varepsilon \mid \lightning$$

where ε is the identity element and all elements except $\lightning$ are valid (*i.e.,* satisfy $\mathcal{V}$). The crucial part is the definition of addition: We define $\mathit{pending} + \mathit{pending}$ to result in $\lightning$ and thus having *pending* twice would violate the validity predicate $\mathcal{V}$ and lead to the contradiction described above.

Formally, this contradiction arises due to the core principle of separation logic:

*"The summation of all resources always **are** and **should remain** valid."*

This principle is the core rely/guarantee principle of separation logic: A user of separation logic (i) can rely on the summation of all current resources being valid (which in turn implies each constituent resource is also valid thanks to monotonicity) but also (ii) needs to guarantee when updating a resource that the summation stays valid. We have actually seen both parts in action: The rely condition (i) ensures that having $\mathcal{V}(\mathit{pending} + \mathit{pending})$ leads to a contradiction, while the guarantee condition (ii) enforces validity is upheld which disallows duplication of the *pending* resource.

ASSUME and ASSERT. So how does this rely/guarantee principle of separation logic help with defining ASSUME and ASSERT? The idea is simple: We use ASSUME to encode the rely condition, while ASSERT encodes the guarantee condition. In particular, ASSUME **`assume`**s the validity of the summation of all separation logic resources while ASSERT **`assert`**s their validity. These definitions of ASSUME and ASSERT are shown at the top of Fig. 2. The key component of these definitions is the **`assume`** (resp. **`assert`**) on line L4 (resp. R4) that assumes (resp. asserts) the validity of the summation of "all" resources. To make this intuition more precise, we consider three questions: (i) What constitutes "all" resources? (ii) How does the wrapper transform the code to allow ASSUME and ASSERT to track these resources? (iii) How does the definition of ASSUME and ASSERT work?

The first question (i) is what constitutes "all" resources. The answer is that the summation on line L4 / R4 consists of the following resources:

- A *module resource* `mrs`: This resource corresponds to the ownership of the current module and it is used to state invariants about the private state of the module.
- A *function resource* `frs`: This resource corresponds to the ownership of the current function and is used to store ownership across function calls.

[2]We omit Σ when it is a referring to a global PCM.

```
ASSUME(Cond) ≡ {
  var σ := take(Σ)                    (* L1 *)
  assume(Cond σ)                      (* L2 *)
  ctx := take(Σ)                      (* L3 *)
  assume(V(mrs + frs + σ + ctx)) }    (* L4 *)

ASSERT(Cond) ≡ {
  var σ := choose(Σ)                  (* R1 *)
  assert(Cond σ)                      (* R2 *)
  (mrs, frs) := choose(Σ × Σ)         (* R3 *)
  assert(V(mrs + frs + σ + ctx)) }    (* R4 *)
```

```
(* ⟨ S_Map ⊢ A_Map ⟩ *)
private map := (fun k => 0)
private mrs: Σ := •ε

def init(sz: int) ≡
  var (frs, ctx) := (ε, ε)
  ASSUME(pending)
  skip
  ASSERT(∗_{k∈[0,sz)} k ↦Map 0)

def get(k: int): int ≡
  var (frs, ctx) := (ε, ε)
  var v := take(int)
  ASSUME(k ↦Map v)
  var r := map[k]
  ASSERT(r = v ∧ k ↦Map v)
  return r

...
```

```
(* I_Main *)
private mrs:Σ := ε
def main() ≡

  var sz := 100

  init(sz)

  var k := 42

  r := get(k)

  output(r)
```

```
(* A_Main *)
private mrs:Σ := ε
def main() ≡

  var sz := 100

  init(sz)

  var k := 42

  r := get(k)

  output(0)
```

```
(* ⟨ S_Map ∪ S_Main ⊢ A_Main ⟩ *)
private mrs: Σ := ε
def main() ≡
ASSUME(pending)
var sz := 100

ASSERT(pending)
init(sz)
ASSUME(∗_{k∈[0,sz)} k ↦Map 0)

var k := 42

var v := choose(int)
ASSERT(k ↦Map v)
r := get(k)
ASSUME(r = v ∧ k ↦Map v)

output(0)
ASSERT(⊤)
```

Fig. 2. The condition-wrapped abstractions for Map and the client module, Main.

- A *call/return resource* σ: This resource corresponds to the ownership of the separation logic proposition *Cond* that is assumed or asserted. Line L2 (resp. R2) assumes (resp. asserts) *Cond* σ to state that σ corresponds to the ownership of *Cond*.
- A *context resource*, ctx: This resource corresponds to the summation of all other resources. In particular, this includes module resources of other modules and function resources of the caller.

This directly leads to the second question (ii), *i.e.,* how the wrapper transforms the code to track these resources. The output of the wrapper $\langle S_{\mathsf{Map}} \vdash A_{\mathsf{Map}} \rangle$ is shown in Fig. 2. Consider `init`: Similar to the stateless wrapper described in §2.1, the wrapper inserts an ASSUME statement to assume the precondition (*i.e.,* pending) and an ASSERT statement to guarantee the postcondition (*i.e.,* $*_{k\in[0,sz)}$ k ↦Map 0). Additionally, the wrapper inserts some boilerplate code (shown background-colored) to track the module, function and context resources. Concretely, the wrapper introduces a module-scoped private variable to store the module resource mrs throughout the whole program's lifetime, which is initialized to the initial ownership of the module.[3] Also, the wrapper introduces function-scoped variables to store the function resource frs and the context resource ctx throughout the current function's lifetime, which are initialized to the unit of the PCM.

Now we can consider question (iii) and look at the definition of ASSUME and ASSERT in detail. To understand their definition, it is important to realize that each ASSUME on the callee side will by matched by a ASSERT on the caller side, similar to **assume** and **assert** in §2.1. With this in mind, let us look at the definitions of ASSUME and ASSERT in Fig. 2. Conceptually, the lines L1-L4 of ASSUME(*Cond*) "take" a call/return resource σ from the caller/callee (L1) and a context resource ctx from the whole environment (L3) and assume that σ satisfies *Cond* (L2) and the summation of

[3]The initial resource $\bullet\varepsilon \in \mathit{Auth}(\mathtt{int} \to \mathit{Ex}(\mathtt{int}))$ should be specified in S_{Map} but is omitted for simplicity of presentation.

all resources is valid (L4). Intuitively, **take** returns a value that has been chosen by the caller. We will make this more precise in a moment. On the other hand, the lines R1-R4 of ASSERT(*Cond*) "choose" a call/return resource σ to give to the callee/caller (R1) and update the module and function resources, `mrs` and `frs`, to chosen resources (R3), and assert that σ satisfies *Cond* (R2) and the summation of all (updated) resources is valid (R4).[4] Intuitive, **choose** provides a value for the callee. Finally, we need consider **take** and **choose** allow passing resources between caller and callee. As we will see in the next section, the answer to this question is dual non-determinism.

2.3 Implicit Value Passing via Dual Non-determinism

Before we can understand how dual non-determinism helps with defining **choose** and **take**, let us look at the open question in more detail: How do we obtain, for example, the call resource σ in the definition of ASSUME and ASSERT? Intuitively, the caller of a function should provide the call resource when it proves the precondition. Thus, a natural idea would be to add an additional argument to each function that corresponds to the σ resource. However, adding a new explicit argument does not work as adding an argument is not a transformation allowed by contextual refinement. Thus instead, CCR employs a mechanism to enable passing arguments *implicitly* via dual non-determinism.

At a high level, our insight is the following: As we have seen in §2.1, **assume** and **assert** can be seen as allowing implicit passing of proofs of pre- and postconditions (*i.e.,* the fact that the condition holds) between caller and callee. The downside of **assume** and **assert** is that they can only be used to pass logical proofs, not actual values. Luckily, we can generalize the mechanism of **assume** and **assert** to a more powerful one, *dual non-determinism*, that can be used to "assume" and "assert" actual values, including separation logic resources!

Dual non-determinism. In particular, we consider two kinds of non-determinism [Back and Wright 2012]: On the one hand, there is so-called *demonic non-determinism*, corresponding to **assert**, which we denote as **choose**(X). On the other hand, there is so-called *angelic non-determinism*, corresponding to **assume**, which we denote as **take**(X). The easiest way to gain intuition for them is to consider their simulation relation rules:

$$\text{(CHR)}\quad \frac{\exists v \in X.\ T \precsim \texttt{var x := } v\texttt{; } S}{T \precsim \texttt{var x := choose}(X)\texttt{; } S} \qquad \text{(TKR)}\quad \frac{\forall x \in X.\ T \precsim \texttt{var x := } v\texttt{; } S}{T \precsim \texttt{var x := take}(X)\texttt{; } S}$$

$$\text{(CHL)}\quad \frac{\forall x \in X.\ \texttt{var x := } v\texttt{; } T \precsim S}{\texttt{var x := choose}(X)\texttt{; } T \precsim S} \qquad \text{(TKL)}\quad \frac{\exists x \in X.\ \texttt{var x := } v\texttt{; } T \precsim S}{\texttt{var x := take}(X)\texttt{; } T \precsim S}$$

Choosing on the right side of the refinement (CHR) requires *providing* a value for the choice—just like an **assert** on the right side (ASTR) requires *proving* the assertion. Taking on the right side (TKR) means *receiving* a value for the choice—*i.e.,* similar to how **assume** on the right side (ASMR) means *assuming* the assertion. As before, the rules for the left side are dual to those for the right.

Eliminating wrappers via implicit value passing. As in §2.1, the effect of implicit value passing happens when we compose compatible wrapped modules and eliminate the wrappers: $\langle S \vdash M_1 \rangle \circ \langle S \vdash M_2 \rangle \sqsubseteq M_1 \circ M_2$. Before we proceed, note that CCR provides this refinement as a general theorem, called ACT (Assumption Cancellation Theorem), so that the user of CCR does

[4] Experts in separation logic might realize that (L3-4) and (R3-4) encode the notion of a *frame-preserving update*. Given any context resource (L3) such that it is consistent with the local resources (L4), we update the local resources (R3) in such a way that they are consistent with the context resource (R4).

not need to prove it (see Theorem 4.1). Note also that this refinement amounts to eliminating the matching conditions ASSERT(*Cond*); ASSUME(*Cond*) and we can prove it in two steps.

First, we *pass* the resource σ from ASSERT(*Cond*) to ASSUME(*Cond*) and eliminate **assume**(*Cond* σ) in ASSUME(*Cond*) as shown below. The simulation proof proceeds mostly by applying corresponding rules in lock steps except for the cases where ASSERT(*Cond*); ASSUME(*Cond*) occurs:

$$
\begin{array}{l}
\quad \texttt{var}\ \sigma := \texttt{choose}(\Sigma);\ \texttt{assert}(\mathit{Cond}\ \sigma);\ (\texttt{mrs}, \texttt{frs}) := \texttt{choose}(\Sigma \times \Sigma);\ \texttt{assert}(\mathcal{V}(\texttt{mrs} + \texttt{frs} + \sigma + \texttt{ctx})); \\
\quad \texttt{var}\ \sigma' := \texttt{take}(\Sigma);\ \texttt{assume}(\mathit{Cond}\ \sigma');\ \texttt{ctx}' := \texttt{take}(\Sigma);\ \texttt{assume}(\mathcal{V}(\texttt{mrs}' + \texttt{frs}' + \sigma' + \texttt{ctx}'));\ \ldots \\
\precsim\ \texttt{var}\ \sigma := \texttt{choose}(\Sigma);\ \texttt{assert}(\mathit{Cond}\ \sigma);\ (\texttt{mrs}, \texttt{frs}) := \texttt{choose}(\Sigma \times \Sigma);\ \texttt{assert}(\mathcal{V}(\texttt{mrs} + \texttt{frs} + \sigma + \texttt{ctx})); \\
\quad \texttt{var}\ \sigma' := \sigma; \qquad\qquad\qquad\qquad \texttt{ctx}' := \texttt{take}(\Sigma);\ \texttt{assume}(\mathcal{V}(\texttt{mrs}' + \texttt{frs}' + \sigma' + \texttt{ctx}'));\ \ldots
\end{array}
$$

Note that we primed the variable names of ASSUME(*Cond*) to distinguish them from those of ASSERT(*Cond*). We can process the first four statements by applying corresponding rules in lock steps. Then we apply (TKL) and (ASML) with the resource σ and the proof of *Cond* σ that are given in the previous steps, which essentially makes an illusion of passing σ. The remaining statements can then be processed in lock steps.

Second, we can eliminate the remaining parts of ASSERT(*Cond*); ASSUME(*Cond*) by a simulation proof but with a rather global construction. Roughly speaking, by (TKL) we always set $\texttt{ctx}'$ to be the summation of all the other module and function resources (in the whole system) than the current ones $\texttt{mrs}'$ and $\texttt{frs}'$. Then by construction, we have $\texttt{mrs}+\texttt{frs}+\sigma+\texttt{ctx} = \texttt{mrs}'+\texttt{frs}'+\sigma'+\texttt{ctx}'$ since both are the summation of all the module and function resources at the same point and therefore can discharge **assume**$(\mathcal{V}(\texttt{mrs}' + \texttt{frs}' + \sigma' + \texttt{ctx}'))$ from **assert**$(\mathcal{V}(\texttt{mrs} + \texttt{frs} + \sigma + \texttt{ctx}))$.

Overall this leads to the following observation:

"Dual non-determinism can give **an illusion of value passing** *among cooperative modules."*

2.4 Incremental and modular verification of the running example

Now we revisit the example from §1 and sketch how to incrementally prove $I_{\mathsf{Map}} \sqsubseteq \langle\, S^0_{\mathsf{Map}} \vdash M_{\mathsf{Map}} \,\rangle$ and $\langle\, S^0_{\mathsf{Map}} \vdash M_{\mathsf{Map}} \,\rangle \sqsubseteq \langle\, S_{\mathsf{Map}} \vdash A_{\mathsf{Map}} \,\rangle$, and how to modularly verify a client of this library using S_{Map}. Before we proceed, we make two important remarks.

First, our strategy to incremental verification in general is as follows. To incrementally verify $I \sqsubseteq M$ with condition S_M and $M \sqsubseteq A$ with condition S_A, we verify $I \sqsubseteq \langle\, S_M \vdash M \,\rangle$ and $\langle\, S_M \vdash M \,\rangle \sqsubseteq \langle\, S_M * S_A \vdash A \,\rangle$ where $S_M * S_A$ is simply obtained by taking the separating conjunction $*$ of the conditions of S_M and S_A for each pre/post-condition. Our observation is that proving $\langle\, S_M \vdash M \,\rangle \sqsubseteq \langle\, S_M * S_A \vdash A \,\rangle$ essentially amounts to proving $M \sqsubseteq \langle\, S_A \vdash A \,\rangle$ since S_M on both sides can be easily canceled out. Then by transitivity we have $I \sqsubseteq \langle\, S_M * S_A \vdash A \,\rangle$. Furthermore, instead of providing $S_M * S_A$ as reasoning principles to the client, we can also reformulate $S_M * S_A$ into a more abstract form using abstract predicates.

In the running example, for this purpose, we make two copies of the pending PCM via Cartesian-product of it with itself and use the first one, say $\mathit{pending}_0$, for the first abstraction and the second one, say $\mathit{pending}_1$, for the second abstraction. Concretely, we use $\{\boxed{\mathit{pending}_0}\}\texttt{init(sz)}\{\top\}$ for S^0_{Map} and $\{\boxed{\mathit{pending}_0} * \boxed{\mathit{pending}_1}\}\texttt{init(sz)}\{*_{k\in[0,\texttt{sz})}\, \mathsf{k} \mapsto_{\mathsf{Map}} \texttt{0}\}$ for S_{Map}. Furthermore, to hide the unnecessary details from the client, we define *pending* as $\mathit{pending}_0 + \mathit{pending}_1$, which can be seen as a conceptually single resource that is not duplicable.

Second, we extend the previous simulation relation with a notion of *module-local relational invariant*. Basically, when proving simulation between two modules, one can fix a relation *Inv* between the possible private states of the two and then (i) prove the invariant *Inv* at the beginning and (ii) assuming *Inv* whenever getting control, prove *Inv* whenever giving control.

First refinement for Map. To prove $I_{\mathtt{Map}} \sqsubseteq \langle\, S^0_{\mathtt{Map}} \vdash M_{\mathtt{Map}} \,\rangle$, we use the following relational invariant *Inv* between the private states of the two modules (*i.e.,* `data` from the former and `map`, `size`, `mrs` from the latter):

$$[\![\ulcorner \mathtt{size} = 0 \land \mathtt{map} = (\mathtt{fun\ k\ =>\ 0}) \urcorner \lor (\boxed{\mathit{pending}_0} * \mathtt{data} \mapsto_{\mathtt{Mem}} \mathtt{map}[0:\mathtt{size}])]\!](\mathtt{mrs})$$

The invariant consists of two cases: the former (*i.e.,* `size = 0 ∧ map = (fun k => 0)`) states the relation before `init` is called and the latter that after `init` is called. The latter says that the module resource `mrs` should contain $\mathit{pending}_0$ and the pointer `data` should point to an array with contents `map[0 : size]` (*i.e.,* `map[0]`, ..., `map[size-1]`). Intuitively, the *points-to predicate* $\mapsto_{\mathtt{Mem}}$ gives exclusive ownership of the memory it points to and thus rules out interference by other modules. The way we model memory accesses and modularly reason about them is presented in §5.2.

With the invariant *Inv*, we can prove the refinement for each function by applying the simulation rules by doing a case analysis on *Inv*. At a high level, for `init`, *Inv* together with its precondition $\mathit{pending}_0$ says that the module should be in the uninitialized state (*i.e.,* the former case of *Inv*) since the initialized state (*i.e.,* the latter case) is conflicted with its precondition due to two copies of $\mathit{pending}_0$. Therefore, one can easily proceed with the simulation rules and at the return point one can reestablish *Inv* by the latter case of *Inv* since `data` is properly initialized and we can own $\mathit{pending}_0$ because the postcondition does not require it.

For `get` and `set`, we can assume that the module is in the initialized state (*i.e.,* the latter case of *Inv*) since in the former case with `size` = 0, the range checking **`assume`**`(0 ≤ k < size)` fails and thus the refinement is completed trivially. Then thanks to the ownership `data` $\mapsto_{\mathtt{Mem}}$ `map[0 : size]`, we can prove that both implementation and abstraction retrieve the same value from `data` and `map` (in case of `get`) and update them equivalently (in case of `set`), and therefore reestablish the latter case of *Inv*, in particular, `data` $\mapsto_{\mathtt{Mem}}$ `map[0 : size]`.

A more low-level proof of this refinement is given in the appendix [Author(s) 2022].

Second refinement for Map. To prove $\langle\, S^0_{\mathtt{Map}} \vdash M_{\mathtt{Map}} \,\rangle \sqsubseteq \langle\, S_{\mathtt{Map}} \vdash A_{\mathtt{Map}} \,\rangle$, we use the following relational invariant *Inv* between the private states of the two modules (*i.e.,* $\mathtt{map}_{\mathrm{M}}$, `size`, $\mathtt{mrs}_{\mathrm{M}}$ from the former and $\mathtt{map}_{\mathrm{A}}$, $\mathtt{mrs}_{\mathrm{A}}$ from the latter):

$$[\![\boxed{\mathtt{mrs}_{\mathrm{M}}} * \ulcorner \mathtt{map}_{\mathrm{M}} = \mathtt{map}_{\mathrm{A}} \urcorner * \bullet(\mathtt{map}_{\mathrm{A}}\langle 0:\mathtt{size}\rangle) * (\ulcorner \mathtt{size} = 0 \urcorner \lor \boxed{\mathit{pending}_1})]\!](\mathtt{mrs}_{\mathrm{A}})$$

where $\mathtt{map}_{\mathrm{A}}\langle 0:\mathtt{size}\rangle \triangleq$ `(fun k => (0 ≤ k < size) ? Some map(k) : None)`. *Inv* says that (i) $\mathtt{mrs}_{\mathrm{A}}$ contains $\mathtt{mrs}_{\mathrm{M}}$, which is needed to systematically cancel out the statements about $\mathit{pending}_0$ from both sides, (ii) $\mathtt{map}_{\mathrm{M}}$ and $\mathtt{map}_{\mathrm{A}}$ coincide, (iii) $\mathtt{mrs}_{\mathrm{A}}$ contains the resource $\bullet(\mathtt{map}_{\mathrm{A}}\langle 0:\mathtt{size}\rangle)$, which means that only `k` $\mapsto_{\mathtt{Map}}$ `0` for $0 \le k <$ `size` is *valid* (*i.e.,* $\mathcal{V}(\mathtt{k} \mapsto_{\mathtt{Map}} 0 * \bullet(\mathtt{map}_{\mathrm{A}}\langle 0:\mathtt{size}\rangle))$), and (iv) `size = 0` or $\mathtt{mrs}_{\mathrm{A}}$ contains $\mathit{pending}_1$. With this invariant, we can prove simulation for each function. Since $M_{\mathtt{Map}}$ and $A_{\mathtt{Map}}$ are identical except for the range checking, the verification essentially amounts to the usual separation logic reasoning to prove that $A_{\mathtt{Map}}$ satisfies $S_{\mathtt{Map}}$, plus a few easy reasonings to rule out failure of the range checking in $M_{\mathtt{Map}}$ and to cancel out $\mathit{pending}_0$. Here we give high-level intuitions about this.

First, we discuss how we cancel out reasoning about $\mathit{pending}_0$. The key idea is that given a context resource for $A_{\mathtt{Map}}$ (by (TKR)), we construct the context resource for $M_{\mathtt{Map}}$ (by (TKL)) in a particular way so that what is happening in $\langle\, S^0_{\mathtt{Map}} \vdash M_{\mathtt{Map}} \,\rangle$, which we cannot control, regarding its resources including $\mathit{pending}_0$ and its module resource does not interfere with what we are doing in $\langle\, S_{\mathtt{Map}} \vdash A_{\mathtt{Map}} \,\rangle$, which we can control, regarding $\mathit{pending}_1$ and its module resource. Then, we can *replay* what happened in $\langle\, S^0_{\mathtt{Map}} \vdash M_{\mathtt{Map}} \,\rangle$ in the abstraction side $\langle\, S^0_{\mathtt{Map}} \vdash M_{\mathtt{Map}} \,\rangle$ discharging necessary reasoning about $\mathit{pending}_0$. For this, we maintain $\mathtt{mrs}_{\mathrm{M}}$ in $\mathtt{mrs}_{\mathrm{A}}$ (the first component of

Inv) to make sure that the context $\texttt{ctx}_A$ given to $A_{\texttt{Map}}$ that is disjoint from $\texttt{mrs}_A$ is also disjoint from $\texttt{mrs}_M$, which allows us to put $\texttt{ctx}_A$ into the context $\texttt{ctx}_M$ for $M_{\texttt{Map}}$.

Second, using the second component $\texttt{map}_M = \texttt{map}_A$, one can trivially prove that the two identical code does the same thing to map and get the same value from map, and also trivially reestablish $\texttt{map}_M = \texttt{map}_A$ after the update.

Finally, note that the resource $\bullet(\texttt{map}_A\langle 0 : \texttt{size}\rangle) * (\ulcorner \texttt{size} = 0 \urcorner \vee \boxed{\mathit{pending}_1})$ is essentially the same as what one would use to prove $A_{\texttt{Map}}$ against $S_{\texttt{Map}}$ in separation logic. Concretely, one would use the existentially quantified version of it there: $\exists \texttt{size}.\ \bullet(\texttt{map}_A\langle 0 : \texttt{size}\rangle) * (\ulcorner \texttt{size} = 0 \urcorner \vee \boxed{\mathit{pending}_1})$. Also, the reasoning in CCR in this case corresponds to that in separation logic.

A more low-level proof of this refinement is given in the appendix [Author(s) 2022].

Using pre- and postconditions modularly. Now we see how to modularly reason about a client of Map using $S_{\texttt{Map}}$. In Fig. 2, the function main in the implementation $I_{\texttt{Main}}$ initializes the Map module with size 100, retrieves the 42nd value, and outputs the result. In the abstraction $A_{\texttt{Main}}$, the output value is abstracted into the constant 0. We define $S_{\texttt{Main}}$ as follows:

$$\{\boxed{\mathit{pending}}\}\ \texttt{main}()\ \{\top\}$$

Then the verification of $I_{\texttt{Main}} \sqsubseteq \langle\, S_{\texttt{Map}} \cup S_{\texttt{Main}} \vdash A_{\texttt{Main}} \,\rangle$ can be easily done by a similar simulation reasoning as before. Here we just make two notable remarks. First, in the wrapped code for get and main in Fig. 2, we pass the logical value v from main to get, which is only used in the precondition of get but not in its code, via the illusion of value passing enabled by **choose** and **take**. Second, the key reasoning in main is done as follows at a high level. From $*_{k\in[0,100)}\ \texttt{k} \mapsto_{\texttt{Map}} \texttt{0}$ given after init(100), we get $42 \mapsto_{\texttt{Map}} \texttt{0}$ since $0 \le 42 < 100$, which we provide to get(42), after which we are given $\texttt{r} = 0 \wedge 42 \mapsto_{\texttt{Map}} \texttt{0}$. Therefore, output(r) and output(0) outputs the same value 0.

3 EXECUTABLE MODULE SEMANTICS (EMS)

Before we present the formal definition of CCR in the next section, we present the base setting first: our module system and its semantics.

3.1 Module and Contextual Refinement

Recall that, in our examples, the central notion we used is a notion of *module*, and we will present how we define our module system which we dubbed EMS (Executable Module Semantics). Before we get there, we first briefly review *interaction trees* [Xia et al. 2019] which we use extensively in our formalization.

Interaction Trees. For a given event type E : **Set** $\to$ **Set** and a return type T, an interaction tree of type itree E T could be seen as an open small-step semantics that can (i) take a silent deterministic step, (ii) terminate with a return value of type T, or (iii) trigger an event in $E(X)$ for some X and, continues execution for each possible return value in X. Since itree E forms a monad for any E, we henceforth use the monad notations: $x \leftarrow i; k$ and $i \gg\!= k$ for bind and **ret** v for return.

We enjoy the following benefits of interaction trees: (i) they can be extracted to executable programs in OCaml (thus "Executable"), (ii) they provide useful combinators and theorems, and (iii) the monad notation serves as a shallow-embedded programming language in Coq, with which we write the semantics of abstractions.

Function and Module. Now we see how we define the notion of module, given in Fig. 3. We use $X|_{cond}$ to denote a conditionally non-empty set. First, fundef(E) is the semantic domain for a function, which takes a value in Any as an argument and gives an itree w.r.t. the event type E and

$$\mathrm{fundef}(E) \triangleq \mathtt{Any} \to \mathtt{itree}\ E\ \mathtt{Any} \qquad X|_{cond} \triangleq \text{if } cond \text{ holds, then } X \text{ else } \emptyset$$
$$\mathrm{E_P}(X) \triangleq \{\mathtt{Choose}\} \uplus \{\mathtt{Take}\} \uplus \{\mathtt{Obs}\ fn\ arg \mid fn \in \mathtt{string},\ arg \in \mathtt{Any}\}|_{X=\mathtt{Any}}$$
$$\mathrm{E_{EMS}}(X) \triangleq \mathrm{E_P}(X) \uplus \{\mathtt{Call}\ fn\ arg | fn \in \mathtt{string},\ arg \in \mathtt{Any}\}|_{X=\mathtt{Any}} \uplus \{\mathtt{Put}\ a \mid a \in \mathtt{Any}\}|_{X=()} \uplus \{\mathtt{Get}\}|_{X=\mathtt{Any}}$$
$$\mathrm{Mod} \triangleq \left\{(\mathtt{init}, \mathtt{funs}) \in \mathtt{Any} \times (\mathtt{string} \xrightarrow{\mathrm{fin}} \mathrm{fundef}(\mathrm{E_{EMS}}))\right\}$$

$$\mathrm{Mods} \triangleq \mathtt{list}\ \mathrm{Mod} \qquad \circ \in \mathrm{Mods} \to \mathrm{Mods} \to \mathrm{Mods} \triangleq \mathtt{append}$$
$$M \sqsubseteq_{\mathrm{beh}} M' \triangleq \mathrm{Beh}(Ms) \subseteq \mathrm{Beh}(Ms') \qquad M \sqsubseteq_{\mathrm{ctx}} M' \triangleq \forall C \in \mathrm{Mods}.\ C \circ M \sqsubseteq_{\mathrm{beh}} C \circ M'$$

Fig. 3. Definitions of module and contextual refinement.

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

Fig. 4. Definitions of trace and behavior.

the return type Any, where Any can be understood as the set of all mathematical values. Mod is the semantic domain for a module, which is given by (i) the initial value of the module local state, init, and (ii) the definitions of the module's functions, funs, with the event type $\mathrm{E_{EMS}}$. $\mathrm{E_{EMS}}$ is the event type for EMS consisting of (i) Choose and Take for nondeterministically *choosing* and *taking* a value from any given set X, (ii) Obs for triggering observable events (*e.g., input*, *output*), (iii) Call for making a call to (internal or external) functions, (iv) Get and Put for accessing the module local state of type Any, Now, the instructions **choose**(X) and **take**(X) are simply an itree triggering Choose(X) and Take(X), respectively. We use **call** *fn x* to denote an itree triggering Call *fn x*, and similarly for **put**, **get** and **obs**. Also, **tau** denotes the interaction tree taking a silent step and immediately returns the unit value of the unit type.

Contextual Refinement. Fig. 3 shows formal definition for the (whole-program) behavioral refinement and contextual refinement. We say modules (Mods) to simply refer to a list of modules and linking $\circ$ between them is the list append. Throughout the paper we use implicit casting from Mod to Mods as a singleton list.

Behavioral refinement between two modules M and M' is defined simply as a set inclusion between the set of possible traces given by Beh(-), which will be explained shortly (§3.2). Contextual refinement between two modules M and M' is defined as behavioral refinement under an arbitrary context modules C. As expected, this definition enjoys both vertical and horizontal compositionality:

$$\text{(Vertical)} \quad I \sqsubseteq_{\mathrm{ctx}} M \wedge M \sqsubseteq_{\mathrm{ctx}} A \implies I \sqsubseteq_{\mathrm{ctx}} A$$
$$\text{(Horizontal)} \quad I_1 \sqsubseteq_{\mathrm{ctx}} A_1 \wedge I_2 \sqsubseteq_{\mathrm{ctx}} A_2 \Rightarrow (I_1 \circ I_2) \sqsubseteq_{\mathrm{ctx}} (A_1 \circ A_2)$$

3.2 Traces and Behavior

As promised, we see how we define the set of possible traces for a list of modules, given in Fig. 4.

Traces. To give the notion of behavior, we first define the set of traces, Trace, coinductively. A trace is a finite or infinite sequence of ObsEvent (*i.e.,* pairs of an observable event and its return value) that can possibly end with one of the four cases: (i) normal termination with an Any value, (ii) silent divergence without producing any events, (iii) erroneous termination, or (iv)

partial termination. The notion of trace is mostly equivalent to that of CompCert, except for the partial termination. Partial termination will serve as a dual of erroneous termination: erroneous termination is terminating due to an error in the program, and partial termination is due to the user (*e.g.,* by pressing Ctrl+C).

Behavior. The behavior $\mathrm{Beh}(Ms)$ for a given list of module is defined in two steps. First, we concatenate the computations described in each function semantics to create a single, large itree using standard combinators of interaction tree (concat). Then, we define a set of possible traces for such an itree (beh). The predicate $\mathrm{beh}(-)$ is defined by a mixed induction coinduction as follows, where box denotes coinduction and dash-box denotes induction. For a given itree i, $\mathrm{beh}(i)$ includes the partial termination (Partial) since the program can be terminated by the user at any point; the divergence (Diverge) if i is *divergent* according to the predicate div defined below; and the following depending on the first step of i: (i) if i executes **tau**, the behaviors of its continuation; (ii) if i executes **choose**, the union of the behaviors of each chosen continuation; (iii) if i executes **take**, the intersection of the behavior of each taken continuation; (iv) if i executes an observable event with fn and arg, the union of the behaviors of each continuation $k(ret)$ prefixed by $(\mathtt{Obs}\ fn\ arg, ret)$ for each *valid* return value ret satisfying valid_obs $fn\ arg\ ret$[5]; and (v) if i returns a value v, the normal termination (Term v). Note that the erroneous termination (Error) can only occur by **take**$(\emptyset)$. The divergence predicate div coinductively defines the set of those itrees that take infinite steps without triggering any observable events, as shown in Fig. 4.

Commands and Operators. Now, we define and discuss several derived commands/operators that we use throughout the paper. First, **UB** and **NB** are defined as **take**$(\emptyset)$ and **choose**$(\emptyset)$, respectively. Note that beh(**UB**) includes all the traces including Error while beh(**NB**) includes only Partial. Also we have the following duality:

$$\begin{array}{lll} \text{(Prefix-closed)} & \forall i, t_0, t_1. & t_0 +\!\!+ t_1 \in \mathrm{beh}(i) \implies t_0 +\!\!+ \mathrm{Partial} \in \mathrm{beh}(i) \\ \text{(Postfix-closed)} & \forall i, t_0, t_1. & t_0 +\!\!+ \mathrm{Error} \in \mathrm{beh}(i) \implies t_0 +\!\!+ t_1 \in \mathrm{beh}(i) \end{array}$$

Next, we define the following operators:

$$\begin{array}{ll} \mathbf{assume}(P) \triangleq \mathbf{if}\ P\ \mathbf{then}\ ()\ \mathbf{else}\ \mathbf{UB} & \mathtt{x?} \triangleq \mathbf{match}\ \mathtt{x}\ \mathbf{with}\ |\ \mathtt{Some}(c) \Rightarrow c\ |\ \_ \Rightarrow \mathbf{UB}\ \mathbf{end} \\ \mathbf{assert}(P) \triangleq \mathbf{if}\ P\ \mathbf{then}\ ()\ \mathbf{else}\ \mathbf{NB} & \mathtt{x!} \triangleq \mathbf{match}\ \mathtt{x}\ \mathbf{with}\ |\ \mathtt{Some}(c) \Rightarrow c\ |\ \_ \Rightarrow \mathbf{NB}\ \mathbf{end} \end{array}$$

For a proposition P, we define **assume** (resp. **assert**) to trigger **UB** (resp. **NB**) if P does not hold. Also, two unwrap operators that pull out the internal value of an option-typed value are defined as follows: the postfix operator **?** and **!** execute **UB** and **NB**, respectively, upon failure.

3.3 Simulation Relation

In CCR, we establish contextual refinement using a standard simulation technique. We have a common simulation relation which relates a pair of an interaction tree (of type $\mathtt{itree}\ \mathrm{E}_{\mathrm{EMS}}$) together with its module-private state (of type Any). Specifically, it allows imposing relational invariants, $\mathbb{I}$, on the module-private states of both side. Also, it additionally allows $\mathbb{I}$ to depend on Kripke-style possible worlds, which could be of any type $\mathcal{W}$ equipped with a preorder $(\sqsubseteq_{\mathcal{W}})$. With these, the simulation relation $\lesssim_w$[6] at a given world $w \in \mathcal{W}$ is coinductively (greatest fixpoint) defined with constructors (rules) shown in Fig. 5.

The definition comprises constructors for: (i) executing a tau step, **choose**, and **take** in the left side (first row), (ii) executing the same for the right side (second row), (iii) executing **put** and **get** (third row), and (iv) executing function return and call (fourth row). (i) and (ii) are largely the

[5]Following CompCert, the parameter predicate valid_obs specifies the possible return values of each observable event.
[6]We omit stuttering index for brevity.

$$\frac{K \xhookrightarrow{\tau} K' \qquad (st, K') \lesssim_{w_0} \mathbb{S}}{(st, K) \lesssim_{w_0} \mathbb{S}} \qquad \frac{\forall x \in X.\ (st, K\ x) \lesssim_{w_0} \mathbb{S}}{(st, x \leftarrow \textbf{choose}(X); K\ x) \lesssim_{w_0} \mathbb{S}} \qquad \frac{\exists x \in X.\ (st, K\ x) \lesssim_{w_0} \mathbb{S}}{(st, x \leftarrow \textbf{take}(X); K\ x) \lesssim_{w_0} \mathbb{S}}$$

$$\frac{\mathbb{T} \lesssim_{w_0} (st, K') \qquad K \xhookrightarrow{\tau} K'}{\mathbb{T} \lesssim_{w_0} (st, K)} \qquad \frac{\exists x \in X.\ \mathbb{T} \lesssim_{w_0} (st, K\ x)}{\mathbb{T} \lesssim_{w_0} (st_s, x \leftarrow \textbf{choose}(X); K\ x)} \qquad \frac{\forall x \in X.\ \mathbb{T} \lesssim_{w_0} (st, K\ x)}{\mathbb{T} \lesssim_{w_0} (st, x \leftarrow \textbf{take}(X); K\ x)}$$

$$\frac{(st', K) \lesssim_{w_0} \mathbb{S}}{(st, \textbf{put}\ st';\ K) \lesssim_{w_0} \mathbb{S}} \qquad \frac{(st, K\ st) \lesssim_{w_0} \mathbb{S}}{(st, \textbf{get} \gg\!= K) \lesssim_{w_0} \mathbb{S}} \qquad \frac{\mathbb{T} \lesssim_{w_0} (st', K)}{\mathbb{T} \lesssim_{w_0} (st, \textbf{put}\ st';\ K)} \qquad \frac{\mathbb{T} \lesssim_{w_0} (st, K\ st)}{\mathbb{T} \lesssim_{w_0} (st, \textbf{get} \gg\!= K)}$$

$$\frac{w_0 \sqsubseteq_{\mathcal{W}} w_1 \qquad \mathbb{I}_{w_1}\ st_t\ st_s}{(st_t, \textbf{ret}\ r) \lesssim_{w_0} (st_s, \textbf{ret}\ r)} \qquad \frac{\mathbb{I}_{w_1}\ st_t\ st_s \qquad \forall r, w_2, st_t', st_s'.\ w_1 \sqsubseteq_{\mathcal{W}} w_2 \wedge \mathbb{I}_{w_2}\ st_t'\ st_s' \Rightarrow (st_t', K_t\ r) \lesssim_{w_0} (st_s', K_s\ r)}{(st_t, r \leftarrow \textbf{call}\ f\ x;\ K_t\ r) \lesssim_{w_0} (st_s, r \leftarrow \textbf{call}\ f\ x;\ K_s\ r)}$$

Fig. 5. Constructors for our common simulation relation (simplified)

same as before, and (*iii*) is straightforward. For (*iv*) those constructors are now equipped with worlds following the standard open simulations[Song et al. 2019]. That is, when returning one needs to assert that $\mathbb{I}$ holds for some future world, w_1, and as a result, after a function call one can rely on that there is some future world, w_2, such that $\mathbb{I}$ holds.

The common simulation relation satisfies the following adequacy theorem.

Theorem 3.1 (Adequacy). *For a pair of modules M_t and M_s, a possible world $\mathcal{W}$, and a relational invariant $\mathbb{I}$ w.r.t. $\mathcal{W}$, and the simulation relation $\lesssim$ w.r.t. $\mathcal{W}$ and $\mathbb{I}$ (defined in Fig. 5), if we have (i) $\exists w_0.\ \mathbb{I}_{w_0}\ M_t.\texttt{init}\ M_s.\texttt{init}$, (ii) $dom(M_t.\texttt{funs}) = dom(M_s.\texttt{funs})$, and (iii) for each pair of function f_t and f_s with the same name, $\forall\, v\ w\ st_t\ st_s.\ (st_t, f_t\ v) \lesssim_w (st_s, f_s\ v)$, the following holds:*

$$M_t \sqsubseteq_{ctx} M_s$$

4 CCR FRAMEWORK, SIMPLIFIED

In this section, we present the CCR framework, which formalizes ideas presented in §2. Specifically, we show how we formally define the wrapper and the ACT theorem for a basic version of CCR. Other advanced features will be presented in subsequent sections.

4.1 Condition Wrapped Abstractions

At the heart of CCR framework is the wrapper, $\langle\, S \vdash_\alpha M \,\rangle$, and we first see its formal definition given in Fig. 6. The whole framework is parameterized with a global PCM, Σ. For each function, we specify its *condition* $s \in \mathrm{Cond}$ consisting of three components $(\mathtt{W}, \mathtt{P}, \mathtt{Q})$ each standing for the type of auxiliary variable and pre- and postconditions. The auxiliary variable w [Kleymann 1999; Schreiber 1997] is shared between $\mathtt{P}$ and $\mathtt{Q}$ (*e.g.,* $\mathtt{v}$ in the spec of `get` in Fig. 1). The passing of $w \in \mathtt{W}$ from a caller to a callee is also encoded via **choose** and **take** as we have seen in Fig. 2 for $\mathtt{v}$ in the spec of `get`. $\mathtt{P}(w)/\mathtt{Q}(w)$, given $w \in \mathtt{W}$, specifies a pre/post condition on (*i*) a concrete (those physically passed) argument/return value, and (*ii*) an argument/return resource. A *collection of conditions* $S \in \mathrm{Conds}$ simply consists of such conditions for a finite set of functions. The wrapping $\langle\, S \vdash_\alpha M \,\rangle$ for a module M, conditions S, and an initial module resource α is again a module with its initial private state now *paired*[7] with the initial module resource α, and its functions wrapped via WrapF. In Fig. 6 and hereafter, we will implicitly cast between `Any` and a certain type such as Σ. Casting failures in the wrapper are technically defined as **UB**, but they are spurious (never actually happens) and gets eliminated in the ACT.

[7]The `Any` type provides a *pair* operator of type $\mathtt{Any} \to \mathtt{Any} \to \mathtt{Any}$ and a *split* operator of type $\mathtt{Any} \to \mathtt{option}(\mathtt{Any} \times \mathtt{Any})$.

$\mathbf{rProp} \triangleq \Sigma \to \mathbf{Prop}$ for $\Sigma \in \mathrm{PCM}$
$\mathrm{Cond} \ni s \triangleq \{(\mathtt{W},\mathtt{P},\mathtt{Q}) \mid \mathtt{W} \in \mathbf{Set} \wedge \mathtt{P},\mathtt{Q} \in \mathtt{W} \to \mathtt{Any} \to \mathbf{rProp}\}$
$\mathrm{Conds} \ni S \triangleq \mathtt{string} \xrightarrow{\mathrm{fin}} \mathrm{Cond}$
$\langle S \vdash_\alpha M \rangle \triangleq ((M.\mathtt{init}, \alpha), \lambda\, fn \in \mathrm{dom}(M.\mathtt{funs}).\mathrm{WrapF}(S,\ S\ fn,\ M.\mathtt{funs}\ fn))$
(* defined only when $\mathrm{dom}(M.\mathtt{funs}) \subseteq \mathrm{dom}(S)$ *)

```
WrapC((W, P, Q), ctx, fn, x) ≜
(*C1*) w ← choose(W);
(*C2*) frs ← ASSERT(P(w),x,ctx);
(*C3*) r ← call fn x;
(*C4*) ctx ← ASSUME(Q(w),r,frs);
(*C5*) ret (r, ctx)
```

```
WrapF(S, (W, P, Q), f ∈ fundef(E_EMS)) ≜ λx.
(*F1*)  w ← take(W); ctx ← ASSUME(P(w), x, ε);
(*F2*)  (r, ctx) ← f(x)[Call fn x ↦ λctx. WrapC((S fn)!, ctx, fn, x),
(*F3*)                  Put mps  ↦ λctx. (_, mrs) ← get; put(mps, mrs); ret ((), ctx),
(*F4*)                  Get      ↦ λctx. (mps, _) ← get; ret (mps, ctx)              ](ctx);
(*F5*)  (_) ← ASSERT(Q(w), r, ctx); ret r
```

```
ASSUME(Cond, xr, frs) ≡
 σ ← take(Σ);
 assume(Cond xr σ);
 ctx ← take(Σ); (_, mrs) ← get;
 assume(V(mrs + frs + ctx + σ));
 ret ctx
```

```
ASSERT(Cond, xr, ctx) ≡
 σ ← choose(Σ);
 assert(Cond xr σ);
 (mrs, frs) ← choose(Σ × Σ); (mps, _) ← get; put(mps, mrs);
 assert(V(mrs + frs + ctx + σ));
 ret frs
```

Fig. 6. Definition of the wrapper.

Inserting conditions. In Fig. 6, we wrap (i) each function definition by inserting a precondition at the beginning and a postcondition at the end (WrapF), and (ii) each function call by inserting a precondition before the call and a postcondition after the call (ASCall). At a high level, WrapF with conditions S for function calls, its own condition $(\mathtt{W},\mathtt{P},\mathtt{Q})$ and its own function definition f **take**s the auxiliary variable, ASSUMEs the precondition (F1), executes the function body f with the given argument x (F2-4), and ASSERTs the postcondition and returns (F5).

In lines F2-4, we use a combinator of interaction trees with type:

$$\mathtt{itree}\ E\ T \to (\forall X.\ E(X) \to ST \to \mathtt{itree}\ E'\ (X \times ST)) \to ST \to \mathtt{itree}\ E'\ (T \times ST)$$

It takes an itree $i \in \mathtt{itree}\ E\ T$, adds a local state of type ST, and interprets each event in E as an itree in a new event type E' that can access and update the local state. In our case, such a local state will store the context resource `ctx`, which was stored in a function-local variable in pseudocode of the previous examples. We use the notation $i[e_1 \mapsto \lambda s.\ t_1, \ldots, e_n \mapsto \lambda s.\ t_n](s_0)$ to denote the resulting itree when the combinator is applied to an itree i, with an initial local state s_0, by interpreting each event e_i to an itree t_i for a given local state s. We omit those events that are interpreted identically, and the state component when it is the unit type.

Now we discuss more details of lines F2-4. In line F2, whenever a function call (*i.e.,* `Call` event) is made, it is wrapped by the wrapper WrapC with the callee's condition in S. Specifically, WrapC simply **choose**s the auxiliary variable (C1), ASSERTs the precondition (C2), makes the intended function call (C3), ASSUMEs the postcondition (C4), and returns (C5). The lines F3-4 are simple. Recall that a module's private state is now a pair of a physical state (used by the module) and a module resource (used by the wrapper). The lines F3-4 simply convert the `Put`/`Get` to access the first element of the pair.

Encoding conditions. The formal definition of ASSUME and ASSERT in Fig. 6 are basically the same as those presented in Fig. 2. The only difference is that `ctx` and `frs`, which were stored in function-local variables in the pseudocode, are now explicitly threaded through. Concretely, the `ctx` taken at the ASSUME in line F1, is passed to the ASSERT in line C2 or F5 via the interpretation combinator. Similarly, the `ctx` taken at the ASSUME in line C4 is passed to line C2 or F5. Also, `frs` is explicitly passed from line C2 to line C4.

4.2 Key theorems of CCR

Now we are ready to formally state the ACT theorem (described in §2) that removes those wrappers.

Theorem 4.1 (Assumption Cancellation Theorem (ACT)). *For a global PCM Σ, wrapped abstractions $\langle S \vdash_{\alpha_i} A_i \rangle$ for $i \in \{1, \ldots, n\}$ and an initial resource α to* `main` *that satisfies its precondition and $\mathcal{V}(\alpha + \alpha_1 + \ldots + \alpha_n)$, if $A_1 \circ \ldots \circ A_n$ is a closed program:*

$$\langle S \vdash_{\alpha_1} A_1 \rangle \circ \ldots \circ \langle S \vdash_{\alpha_n} A_n \rangle \sqsubseteq_{beh} A_1 \circ \ldots \circ A_n$$

The validity condition ensures that the summation of all resources at the beginning of the program is valid. We also have the following extensionality theorem.

Theorem 4.2 (Extensionality). *For any S, S', A, α, S_A, the following holds:*

$$S \subseteq S' \implies \langle S \vdash_{\alpha} A \rangle \sqsubseteq_{ctx} \langle S' \vdash_{\alpha} A \rangle$$

Although Theorem 4.1 can be applied to arbitrary abstractions, if we consider the special case where the abstractions are trivially safe programs, CCR can be seen as playing the usual separation logic. To be specific, we define a special module $\mathrm{Safe}(ns_{\mathrm{in}}, ns_{\mathrm{out}})$, which defines functions with their names in ns_{out} to only nondeterministically invoke arbitrary functions in ns_{in} with arbitrary arguments for any (finite or infinite) number of times. Then, we have:

Lemma 4.3 (Safety). *For $ns \subseteq ns_1 \uplus \ldots \uplus ns_n$, $Safe(ns, ns_1) \circ \ldots \circ Safe(ns, ns_n)$ produces no Error.*

This holds since what the composed whole program actually does is only calling each other. Combining Lemma 4.3 and Theorem 4.1 leads to the following corollary, showing how CCR can be used to prove safety of programs:

Corollary 4.4 (SL). *Given a global PCM Σ, (I_i, α_i) for $i \in \{1, \ldots, n\}$, $ns \subseteq dom(I_1.\texttt{funs}) \uplus \ldots \uplus dom(I_n.\texttt{funs})$, and an initial resource α to* `main` *satisfying its precondition and $\mathcal{V}(\alpha + \alpha_1 + \ldots + \alpha_n)$,*

$$(\forall i.\ I_i \sqsubseteq_{ctx} \langle S \vdash_{\alpha_i} Safe(ns, dom(I_i.\texttt{funs})) \rangle) \implies I_1 \circ \ldots \circ I_n \text{ produces no Error.}$$

Proving $I_i \sqsubseteq_{\mathrm{ctx}} \langle S \vdash_{\alpha_i} \mathrm{Safe}(ns, \mathrm{dom}(I_i.\texttt{funs})) \rangle$ essentially amounts to proving I_i against S in separation logic, and we also get the same result by this corollary.

5 MORE EXAMPLES AND FEATURES

In this section, we present advanced features of CCR with motivating examples. The formalization of the full version of CCR is given in the appendix ([Author(s) 2022]).

5.1 Cancellable Calls

Consider the following example where, in the implementation side (left), function `f` calls `fib` with argument 10 and outputs the result, and in the abstraction side (right), `f` directly outputs 55.

$$\texttt{def f() } \equiv \texttt{ var x := fib(10); } \mathit{output}(\texttt{x}) \not\sqsubseteq_{\mathrm{ctx}} \langle S \vdash \texttt{def f() } \equiv \mathit{output}(55) \rangle$$

One would expect this contextual refinement to hold if one assumes a suitable Hoare triple S for `fib`, *e.g.*, stating that it returns the n-th fibonacci number. However, there is a problem: This refinement eliminates a call to an unknown function (`fib`), which may interact with the user (*e.g.*, via *output*), and thus may not hold in general. This is not a fundamental limitation since we can always put matching function calls in the abstraction. For example, if we change the code of the abstraction into `fib(10); output(55)`, the refinement would hold. However, it would be better to eliminate those function calls that are specified as *cancellable*.

Indeed CCR supports such a feature: (*i*) we support a mechanism to specify whether a function call is "cancellable" and remove those in the ACT, and (*ii*) we also allow the user to omit those

cancellable function calls when writing an abstraction. Note that the same function may be cancellable or not depending on its argument value and thus cancellability is a property of a function call instead of a function definition. First, we consider what makes a function call cancellable. Pure function calls, *i.e.,* function calls that does not trigger any visible event (including Diverge) and does not modify any state, are clearly cancellable. However, the class of cancellable functions is slightly larger: we allow cancellable calls to modify *resources* which anyway gets removed by ACT. In other words, cancellable calls are those function calls that become pure after eliminating conditions and resources. Then, those pure calls can also be removed by ACT.

Now, how do we specify and enforce the notion of cancellability? To enforce that a function does triggers visible events or modifies the physical state, CCR imposes a simple syntactic restriction (to be described in more detail shortly). A more interesting question here is how we enforce a function call to terminate. Note that diverging function calls are not cancellable. For this, in Cond (Fig. 6), we will add a new component $\mathtt{D}(w)$ which, given $w \in \mathtt{W}$, specifies the maximum call depth. Specifically, a depth $d \in \mathrm{Depth}$ is either ∞ denoting the call is not specified as cancellable, or an ordinal $\langle o \rangle$ denoting the call is cancellable and has a maximum call depth o. This means a call with depth $\langle o \rangle$ is only allowed to call functions with depth *strictly less* than o. Those conditions together allows ACT to remove those cancellable calls, solving the issue (i) above.

As an example, the above `fib` function can have the following specification:

$$\forall n : \mathbb{N}.\ \{\lambda x.\ \ulcorner n < \mathtt{INTMAX} \ \wedge\ x = n \urcorner\}\ \ \{\lambda r.\ \ulcorner r = \mathit{fibmath}(n) \urcorner\}\ \ \{\langle n \rangle\}$$

where $\mathbb{N}$ and the three components in the curly brackets correspond to $(\mathtt{W}, \mathtt{P}, \mathtt{Q}, \mathtt{D}) \in \mathrm{Cond}$, respectively. The pre-/postconditions state that given a non-negative $n \in \mathbb{N}$, the return value for `fib(n)` is specified as the `n`-th fibonacci number (denoted by a mathematical function).[8] The last bracket is a newly added component which says `fib(n)` is cancellable and its maximum call depth is `n`.

After seeing how the notion of the cancellable call is defined, we now turn to the issue (ii) above. For this, first observe that the abstraction after ACT does not change even if the wrapper adds arbitrary cancellable calls to the abstractions since they will get removed by ACT anyway. Thus, the wrapper implicitly and automatically inserts the following boilerplate code at every line.

```
var n := choose(Ordinal); repeat n { ASSERT(...); □; ASSUME(...) }
```

We call this construction ACC (Arbitrary Cancellable Calls). ACC executes the □ part for a nondeterministically chosen number of times, where the □ part makes a nondeterministically chosen cancellable call according to the given spec. In other words, an ACC is an over-approximation of possible cancellable calls in the implementation. With this, the refinement of `f` above now holds because there is an automatically inserted ACC on the abstraction side which one can instantiate `n` as 1 and then instantiate □ as `fib(10)`. Note that if there is no cancellable call to be matched in the implementation, one can simply instantiate `n` to be 0 to skip the ACC in the simulation proof.

Finally, the aforementioned syntactic enforcement is simply made as follows: we enforce the body of a cancellable call to be an ACC. This captures the notion of cancellable call well since the only thing a cancellable call is supposed to do is to make other cancellable calls (with *strictly decreasing* depth) with their conditions (which could modify the module resource).

5.2 Memory as a Module

Now we see how we handle memory as promised in §2.2. When it comes to handling memory (or a global state in general), it is common in other module systems[Gu et al. 2015; Song et al. 2019] to just pass the memory as an additional argument (return value) each time a function gets invoked (returns). In CCR, we explore a rather different design: we handle memory as a *module*.

[8] There are implicit castings from $\mathbb{N}$ to `int` and ordinal.

$\alpha_{\texttt{Mem}} := \bullet\varepsilon \in \mathit{Auth}(\texttt{ptr} \rightharpoonup \mathit{Ex}(\texttt{val})) \subseteq \Sigma$
$S_{\texttt{Mem}} := \{$

`calloc`:	$\forall n : \texttt{int}.$	$\{\lambda x.\ \ulcorner x = [n] \land n \geq 0 \urcorner\}\{\lambda r.\ \exists p{:}\texttt{ptr}.\ (p \mapsto_{\texttt{Mem}} (\mathit{repeat}\ 0\ n)) * \ulcorner r = p \urcorner\}\{\langle 0 \rangle\},$
`load`:	$\forall (p, v) : \texttt{ptr} \times \texttt{val}.$	$\{\lambda x.\ (p \mapsto_{\texttt{Mem}} [v]) * \ulcorner x = [p] \urcorner\}\{\lambda r.\ (p \mapsto_{\texttt{Mem}} [v]) * \ulcorner r = v \urcorner\}\{\langle 0 \rangle\},$
`store`:	$\forall (p, v) : \texttt{ptr} \times \texttt{val}.$	$\{\lambda x.\ (p \mapsto_{\texttt{Mem}} [\_]) * \ulcorner x = [p, v] \urcorner\}\{\lambda r.\ (p \mapsto_{\texttt{Mem}} [v]) * \ulcorner r \in \texttt{val} \urcorner\}\{\langle 0 \rangle\},$
`free`:	$\forall \_ : ().$	$\{\lambda x.\ \exists p{:}\texttt{ptr}.\ (p \mapsto_{\texttt{Mem}} [\_]) * \ulcorner x = [p] \urcorner\}\{\lambda r.\ \ulcorner r \in \texttt{val} \urcorner\}\{\langle 0 \rangle\}$ $\}$

Fig. 7. Selected specifications for `Mem` module.

In particular, we define a module, `Mem`, representing memory and implement each memory operation as a function of this module. The benefits of this approach are as follows: First, defining memory as a module allows us to reuse CCR's existing mechanism for specifying pre- and post-conditions on functions. In particular, we can give a standard separation logic pre- and post-conditions for memory operations[Reynolds 2002] involving the points-to predicate $\mapsto_{\texttt{Mem}}$. Second, defining memory as a module makes it easy to support different memory allocators and memory models as they can be defined independently. This means CCR does not "bake-in" the memory itself as a primitive in the framework, but its notion of modules allows encoding of memory. These together means that we do not need to extend the framework at all to handle memory.

The memory module we use, $I_{\texttt{Mem}}$, is defined using a simplified version of the CompCert memory model. Specifically, its private state consists of a finite partial mapping from pointers to values ($\texttt{mem} : \texttt{ptr} \xrightarrow{\text{fin}} \texttt{val}$). Its specification, $S_{\texttt{Mem}}$ in Fig. 7, follows the usual style of specifying memory operations in separation logic. In particular, it is specified using a points-to predicate $p \mapsto_{\texttt{Mem}} l$ denoting the ownership of a list of data, l, stored in the memory location from p to $p + \mathit{len}(l)$. This predicate is defined using the same kind of PCM as the `Map` module described in §2.4.

One interesting aspect of the specifications in $S_{\texttt{Mem}}$ is that those calls are cancellable, as manifested by the depth $\langle 0 \rangle$. This might be surprising since a call to *e.g.*, `store` in $I_{\texttt{Mem}}$ modifies the memory `mem`, which is not a pure operation. However, in $\langle\, S_{\texttt{Mem}} \vdash_{\alpha_{\texttt{Mem}}} A_{\texttt{Mem}} \,\rangle$, `mem` is abstracted to a module resource (*i.e.,* $A_{\texttt{Mem}}$ has a module-private state of type unit), and the operations like `store` modify the module resource instead. As described in §5.1, cancellable function calls are allowed to modify module resources as these resources are eliminated by the ACT. This allows the memory operations to be cancellable and thus be eliminated through refinement. For instance, in the running example of Fig. 1, we do not need to write calls to the memory module in the abstraction $A_{\texttt{Map}}$ since they are implicitly inserted and removed.

5.3 Abstraction of Arguments and Return Values

Consider the following example where, in the implementation module (left), there is one function, `popall`, which takes a pointer (`h`) to a linked-list containing integer values (stored in memory), then pops all the elements while printing it along the way. In the abstraction (right), it basically does the same thing but now it takes a mathematical list (`l`).

```
def popall(h: ptr) ≡ if h then print(pop(h)); popall(h) else skip ⋢ctx
def popall(l: list ℤ) ≡ match l with | hd::tl ⇒ print(hd); popall(tl) | _ ⇒ skip end
def main() ≡ var h := newlist(); push(h, 9); popall(h) ⋢ctx def main() ≡ popall([9])
```

Here, the issue is that this seemingly sensible contextual refinement does not hold because the type of the argument has changed. As seen in §2.1, in contextual refinement all the arguments and return values in both sides should be and expected to be equal. For the same reason, the refinement for a client module containing one function `main` also does not hold: the implementation (lower left) calls `popall` with a linked-list containing 9 in the memory and the abstraction (lower right) calls `popall` with a singleton mathematical list containing 9.

```
(* module I_RP *)
def repeat(f:ptr, n:int, m:int) ≡
  if n ≤ 0 then return m
  else { var v := (*f)(m)
         return repeat(f, n-1, v) }
```

```
(* module I_SC *)
def succ(m:int) ≡
  m + 1
```

```
(* module I_AD *)
def main() ≡
  var n := getint()
  print(str(repeat(&succ,n,n)))
```

$H_{\mathsf{RP}}(S_{\mathsf{f}}) := \{\texttt{repeat} : \forall(f, n, m, f_{\mathrm{sem}}) : \texttt{ptr} \times \texttt{int} \times \texttt{int} \times (\texttt{int} \to \texttt{int}).$
$\{\lambda x.\ \ulcorner x = [f, n, m] \urcorner \wedge n \geq 0 \wedge S_{\mathsf{f}} \sqsupseteq \{*f : \forall m : \texttt{int}, \{\lambda x.\ \ulcorner x = [m] \urcorner\}\{\lambda r.\ \ulcorner r = f_{\mathrm{sem}}(m) \urcorner\}\}\urcorner\}$
$\{\lambda r.\ \ulcorner r = {f_{\mathrm{sem}}}^{n}(m) \urcorner\}\}$
$S_{\mathsf{SC}} := \{\texttt{succ}: \forall m : \texttt{int}.\ \{\lambda x.\ \ulcorner x = [m] \urcorner\}\{\lambda r.\ \ulcorner r = m + 1 \urcorner\}\}$
$S_{\mathsf{AD}} := \{\texttt{main}: \forall \_ : ().\ \{\lambda x.\ \ulcorner x = [] \urcorner\}\{\lambda \_.\ \top\}\}$

Fig. 8. An example of higher-order reasoning

We extend CCR to support this kind of refinement. Again, the idea is to use dual non-determinism to give an illusion of value passing discussed in §2.3. That is, the wrapper will automatically insert **choose** and **take** adequately so that the user can write abstractions *as if* they are sending/receiving those **abstract** values (*e.g.,* `[9]` and `l`) around, but under the hood the wrapper adjusts it so that it physically sends/receives the same value as in the implementation (*e.g.,* `h` and `h`), which is needed for the module-wise contextual refinement to hold.

Those abstract values (either an argument or a return value) are *illusory* things at the wrapped-abstractions, just like resources. However, the ACT will now do one additional task: it will materialize those abstract values so that, after the cancellation, they get *physically* passed around. In the above example, what will be left after the ACT removes the wrapper will exactly be the abstraction on the right hand sides, now physically passing those abstract values (*e.g.,* `[9]` and `l`). Note the difference between the notion of resources and those abstract values where the former gets erased in the cancellation, and the later gets materialized.

For all those mechanisms to make sense, at least the relation between abstract values and physical values should somehow be specified so that the wrapper can make an illusion with respect to it. To this end, we extend our pre- and post-conditions to have one additional parameter, x_{a} and r_{a}, meaning an abstract argument and an abstract return value, respectively. With this, the specification for the above `popall` could be written as follows:

$$\forall h : \texttt{ptr}.\ \{\lambda x\, x_{\mathrm{a}}.\ \exists \ell : \texttt{list}\ \mathbb{Z}.\ \ulcorner x = [h]\uparrow \urcorner \wedge x_{\mathrm{a}} = \ell\uparrow \urcorner * \mathrm{is\_list}\ h\ \ell\}\ \ \{\lambda r\, r_{\mathrm{a}}.\ \top\}\ \ \{\infty\}$$

saying that (i) in the implementation a pointer `h` is passed (x), (ii) in the abstraction a mathematical list will be passed (x_{a}), and (iii) `h` is pointing to a linked list containing the values of `l`. The postcondition is simply true, and this function is not cancellable since it makes visible effects.

Now we see how we make such an illusion of abstract value passing, again with dual non-determinism. When sending a value to another module, the abstraction (user writes) will send an abstract value and the wrapper will change it to a physical value **choose**n with respect to the condition. On the other hand, when receiving a value from another module, the physical value will be received and the wrapper will change it to an abstract value **take**n with respect to the condition. The formal definition of such wrapping is given in [Author(s) 2022].

When $x_{\mathrm{a}}/r_{\mathrm{a}}$ in the pre/post-condition is omitted, it means they are equal to x/r, respectively. We conclude this section with the following remark: the abstract argument can contain essentially more information – that was only available in the ghost resource – than the argument in the implementation. In this example, the `h` itself does not have any information about the contents, but `l` carries such information.

5.4 Function Pointers

Finally, we present how we can do higher-order reasoning involving function pointers of C-like languages without extending the framework. The idea is simple. As already known in the literature[Charguéraud 2020] – "Nested triples are naturally supported by shallow embeddings of Separation Logic in higher-order logic proof assistants." – we can use higher-order quantification of the meta-logic, Coq. In our setting, since the collection of specifications (Conds in Fig. 6) themselves are an object in the meta-logic, Coq, they can be made higher order in the meta-logic.

Concretely, consider the example given in Fig. 8. The function `repeat(f,n,m)` in I_{RP} recursively apply `*f`, `n` times, to `m`, where `*f` is the function pointed to by the pointer value `f`. The definitions in I_{SC} and I_{AD} are straightforward to understand except that `&succ` is the pointer value pointing to the function `succ`. The abstraction A_{AD}, omitted in the figure, turns the call to `repeat` into the addition, `(n + n)`.

To specify `repeat`, we essentially need to embed expected conditions for argument functions `f` inside the condition of `repeat`. First, we give a higher-order condition H_{RP} to the module `RP`, given in Fig. 8, which is given as a function from conditions to conditions. Concretely, given $S_{\mathtt{f}}$, for arguments f, n, m and a mathematical function f_{sem}, the condition $H_{\mathsf{RP}}(S_{\mathtt{f}})$ assumes $S_{\mathtt{f}}$ to include the expected specification for $*f$ (saying that $*f$ returns $f_{\mathrm{sem}}(m)$ for any argument m), and then guarantees that the return value is ${f_{\mathrm{sem}}}^n(m)$. We have omitted the Depth parameter for those conditions since the notion of cancellable calls are orthogonal to higher-order reasoning.

Then we verify `RP`. For any $S_{\mathtt{f}}$ and any $S \supseteq_{\mathsf{S}} (S_{\mathtt{f}} \cup H_{\mathsf{RP}}(S_{\mathtt{f}}))$ (since `repeat` calls $*f$ and itself), we prove:

$$I_{\mathsf{RP}} \sqsubseteq_{\mathrm{ctx}} \langle\, S \vdash_{\varepsilon} A_{\mathsf{RP}} \,\rangle.$$

Also, we verify `SC`. For any $S \supseteq_{\mathsf{S}} S_{\mathsf{SC}}$, we prove:

$$I_{\mathsf{SC}} \sqsubseteq_{\mathrm{ctx}} \langle\, S \vdash_{\varepsilon} A_{\mathsf{SC}} \,\rangle.$$

Also, we verify `AD`. For any $S_{\mathtt{f}} \supseteq_{\mathsf{S}} S_{\mathsf{SC}}$ (since `succ` is passed to `repeat`) and any $S \supseteq_{\mathsf{S}} (H_{\mathsf{RP}}(S_{\mathtt{f}}) \cup S_{\mathsf{AD}})$ (since `add` makes a call to `repeat`), we prove:

$$I_{\mathsf{AD}} \sqsubseteq_{\mathrm{ctx}} \langle\, S \vdash_{\varepsilon} A_{\mathsf{AD}} \,\rangle.$$

Finally, we instantiate those proofs with $S_{\mathtt{f}} = S_{\mathsf{SC}}$ and $S = H_{\mathsf{RP}}(S_{\mathsf{SC}}) \cup S_{\mathsf{SC}} \cup S_{\mathsf{AD}}$ and apply ACT:

$$I_{\mathsf{RP}} \circ I_{\mathsf{SC}} \circ I_{\mathsf{AD}} \sqsubseteq_{\mathrm{beh}} \langle\, S \vdash_{\varepsilon} A_{\mathsf{RP}} \,\rangle \circ \langle\, S \vdash_{\varepsilon} A_{\mathsf{SC}} \,\rangle \circ \langle\, S \vdash_{\varepsilon} A_{\mathsf{AD}} \,\rangle \sqsubseteq_{\mathrm{beh}} A_{\mathsf{RP}} \circ A_{\mathsf{SC}} \circ A_{\mathsf{AD}}$$

As an advanced example, we also verify Landin's knot [Birkedal and Bizjak 2020] (see our Coq development [Author(s) 2022]).

6 IMPLEMENTATION AND EVALUATION

6.1 Imp and its Verified Compiler

For an end-to-end verification, we develop a deep-embedded language, IMP, for implementing the modules. The IMP language is extended from `Imp` [Xia et al. 2019], and has standard syntax as follows:

$$\begin{array}{l} x \in \mathrm{LVarName} \qquad f \in \mathrm{GlobName} \\ e \in \mathit{Expr} ::= x \mid i : \mathtt{int}_{64} \mid e_1 \ == \ e_2 \mid e_1 < e_2 \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \\ s \in \mathit{Stmt} ::= \mathtt{skip} \mid x := e \mid s_1; s_2 \mid \mathtt{if}\ e\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2 \mid x = \&f \mid x = f(e_1, ..., e_n) \mid x = (*e)(e_1, ..., e_n) \mid \\ \qquad\qquad x = \mathtt{malloc}(e) \mid \mathtt{free}(e) \mid x = \mathtt{load}(e) \mid \mathtt{store}(e_1, e_2) \end{array}$$

The semantics is also standard except that the memory operations are interpreted as function calls to `Mem` module (§5.2). The notion of value consists of 64-bit integers and pointers.

We also develop a verified compiler from IMP to Csharpminor of CompCert [Leroy 2006], which is then composed with CompCert to give a verified compiler $(\!|-|\!)$ from IMP to assembly.

Theorem 6.1 (Separate Compilation Correctness). *Given* (I_i, Asm_i) *with* $(\!|I_i|\!) = \mathsf{Some}\ Asm_i$ *for* $i \in \{1, \ldots, n\}$,

$$Asm_1 \bullet \cdots \bullet Asm_n{}^{9} \sqsubseteq_{beh} I_{\mathtt{Mem}} \circ I_1 \circ \cdots \circ I_n$$

Here $\bullet$ is the *syntactic linking* operator of CompCert, and $I_{\mathtt{Mem}}$ is an EMS module (directly written as itrees) that implements our memory model (*i.e.,* a simplified version of CompCert's).

6.2 Evaluation

Our development comprises 42,794 SLOC of Coq (counted by `coqwc`), including 12,925 SLOC for the examples. The examples reason about various representative features of C-like languages including shared memory, mutual recursion, function pointers, (non-)termination, and interaction with the user. Further explanations for most of these examples could be found in the technical report [Author(s) 2022].

As already mentioned (§2.2), vertical compositionality played a crucial role in simplifying the proof of the ACT (Theorem 4.1). Specifically, the theorem is established by transitively composing six refinements, where major ones of which are (i) removing `ASSUME` and `GUARANTEE` while materializing the abstract arguments (§5.3) and (ii) removing cancellable calls (§5.1) by proving their termination using the depth information.

Since our formalization is built on top of Interaction Trees, all the examples in the paper and appendix can be extracted to OCaml and run. Note that all the `itree` events are handled inside Coq except for the primitive events, $\mathrm{E_P}$. $\mathrm{E_P}$ gets extracted to OCaml and is handled by special handlers written in OCaml. Specifically, we wrote a few handlers doing IO for `Obs` and a handler for `Choose` and `Take`, which asks the user for a nondeterministic choice (currently only supports `int`).

The extraction allows differential testing between implementations and abstractions (*i.e.,* executing both and comparing the results). Interestingly, we found two mis-downcast bugs in one of our example (the `Echo` example [Author(s) 2022, §3.4]) by testing it before verification.

7 DISCUSSION AND RELATED WORK

As explained in the introduction, we are not the first to consider how to combine separation logic and refinement in a single framework, but prior work in this direction does not fully marry the benefits of separation logic and refinement in a unified mechanism. We compare here with the most closely related work.

Contextual refinement. In general, refinement techniques may or may not be modular in the structure of a program (*i.e.,* they may require whole-program reasoning). *Contextual refinement* is a variant of refinement that is *inherently modular*: component I contextually refines S (written $I \sqsubseteq_{\mathrm{ctx}} S$) if $C[I] \sqsubseteq C[S]$ under all closing program contexts C. It is also *inherently transitive* by definition. Since contextual refinement is typically difficult to establish directly (due to the quantification over all contexts C), many techniques have been developed for proving it *locally* (*i.e.,* without explicitly reasoning about the context), including some based on separation logic [Frumin et al. 2021a; Gäher et al. 2022; Turon et al. 2013]. A key limitation of contextual refinement, however, is that it is in a certain sense *too* strong: it only applies to refinements that hold under *all* program contexts, thus excluding refinements that hold only under contexts that satisfy some conditions. Although some formulations of contextual refinement restrict the context—*e.g.,* to be well-typed—this still does not provide a very fine-grained method of expressing the precise conditions on C under which $C[I] \sqsubseteq C[S]$.

[9] We cast CompCert's events into `Obs` events in EMS.

Relational separation logics for contextual refinement. There has been a long line of work on using *relational* separation logics [Benton 2004; Yang 2007] as a tool for effectively proving contextual refinement in higher-order, imperative, and concurrent languages [Dreyer et al. 2010; Frumin et al. 2018, 2021b; Gäher et al. 2022; Turon et al. 2013]. In these frameworks, separation logic plays a critical role as a way of modularizing the proof of the contextual refinement itself, and contextual refinement (by virtue of its transitivity) plays a critical role of enabling the verification of the program to be performed in a stepwise, incremental fashion. But as explained in the introduction, the benefits of the two mechanisms remain separate: they offer no way to express refinements that are both *conditional* (with separation logic conditions) and *transitively composable*, as CCR refinements are.

Simulation versus behavioral refinement. It is perfectly valid to take a simulation relation (Fig. 3)—instead of contextual refinement—as a universal building block. However, we advocate here for using contextual refinement as a building block since (i) it gives vertical compositionality for free, and more importantly, (ii) it is extensional: it specifies the property at a higher level without mentioning how a pair of modules are simulated internally. This extensional definition is beneficial because there could be multiple different simulation relations (implying contextual refinement), and it is unclear whether there is a universal simulation relation that can be used for all examples.

Moreover, such an extensional nature of behavioral refinement can make gluing different projects together easier. Specifically, in our end-to-end verification, the simulation being used in program verification (Fig. 5), IMP Compiler (§6.1), and CompCert[Leroy 2006] are vastly different. However, we can still compose them by first converting each of them to behavioral refinement; the notion of behavior remains (almost) the same among these. This is in contrast to simulation-based frameworks (*e.g.,* [Gu et al. 2016]) where a uniform simulation is used across the compiler and program verifications.

Hierarchical refinement. Another popular approach to refinement, as a program verification technique, is what we call *hierarchical refinement*. Here, we first prove some notion of refinement for the lowest-level (*i.e.,* has no dependence on other modules) library module I_1 against its abstraction: $I_1 \sqsubseteq \ldots \sqsubseteq A_1$. Then, we prove that a client module I_2 refines its abstraction A_2, as follows: $A_1 \oplus I_2 \sqsubseteq \ldots \sqsubseteq A_2$. Note that all the functions and private state of A_1 are *inlined* into its client module. Next, we prove $A_2 \oplus I_3 \sqsubseteq \ldots \sqsubseteq A_3$—where A_2 serves as a library module this time—and this process is repeated until the whole system is verified.

This rather simple and elegant idea was popularized by Gu et al. [2015] in their work on Certified Abstraction Layers (CAL), and it has proven to be powerful enough to verify both the CertiKOS concurrent OS kernel [Gu et al. 2016] and the SeKVM hypervisor [Li et al. 2021]. Specifically, it enjoys full compositionality—both vertical and horizontal—and it supports conditional refinement proofs in a specific sense: the refinement proof for a client of a library module can depend conditionally on the specification of the library module [Lorch et al. 2020] because the proof can literally inline the (abstracted) code of the library module.

However, in terms of modular reasoning, the CAL approach also has limitations: (i) it does not support mutual recursion—since there is a strict order between modules, imposed by dependence—and (ii) its support for modular reasoning about shared state is limited compared to that of separation logic. In particular, if the private state of a library module is shared among multiple client modules—as in our running example(Fig. 1)—one needs to employ non-local reasoning across the client modules. We believe the idea of CCR could potentially be applied in this setting to overcome the second limitation.

Dual non-determinism. The notion of dual angelic/demonic non-determinism—which is central to how we operationally enforce separation-logic specifications on modules—is an old idea, but has mainly been studied in the context of game semantics. The most recent and relevant work to ours in this space is the work on Refinement-Based Game Semantics (RBGS) [Koenig 2020; Koenig and Shao 2020]. However, their focus was to unify the notions of refinement, game semantics, and algebraic effects, and they do not consider the interaction with separation logic.

8 LIMITATIONS AND FUTURE WORK

At the moment, CCR does not support any form of concurrency. While we believe the approach used in §5.4 should be applicable for most programming patterns in C, we surely do not support full features of higher-order separation logic[Jung et al. 2016], which was used crucially for higher-order languages like Rust or OCaml.

Since CCR is a new framework that spans refinement-style verification, logic-style verification, and testing, there are various future research directions: (*i*) supporting concurrency in the style of Iris [Jung et al. 2018]; (*ii*) developing property-based testing tools for efficient differential testing between an implementation and its abstraction; and (*iii*) integrating the idea of Parametric Bisimulations [Hur et al. 2012] to support general higher-order languages.

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

Anonymous Author(s). 2022. CCR: Technical documentations and Coq developments.

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

Nick Benton. 2004. Simple Relational Correctness Proofs for Static Analyses and Program Transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. Association for Computing Machinery, New York, NY, USA, 14–25. https://doi.org/10.1145/964001.964003

Lars Birkedal and Aleš Bizjak. 2020. Lecture notes on iris: Higher-order concurrent separation logic. https://iris-project.org/tutorial-material.html

Arthur Charguéraud. 2020. Separation Logic for Sequential Programs (Functional Pearl). *Proc. ACM Program. Lang.* 4, ICFP, Article 116 (aug 2020), 34 pages. https://doi.org/10.1145/3408998

Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. 2010. A Relational Modal Logic for Higher-Order Stateful ADTs. *SIGPLAN Not.* 45, 1 (jan 2010), 185–198. https://doi.org/10.1145/1707801.1706323

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. 2021a. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *Log. Methods Comput. Sci.* 17, 3 (2021). https://doi.org/10.46298/lmcs-17(3:9)2021

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021b. 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. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. https://doi.org/10.1145/3498689

Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo. 2011. CertiKOS: A Certified Kernel for Secure Cloud Computing. In *Proceedings of the 2nd ACM SIGOPS Asia-Pacific Workshop on Systems (APSys 2011)*.

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)*.

Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. 2012. The Marriage of Bisimulations and Kripke Logical Relations. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012)*.

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-Order Ghost State. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 256–269. https://doi.org/10.1145/2951913.2951943

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).

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal verification of an OS kernel. In *SOSP*. ACM, 207–220. https://doi.org/10.1145/1629575.1629596

Thomas Kleymann. 1999. Hoare Logic and Auxiliary Variables. *Form. Asp. Comput.* 11, 5 (dec 1999), 541–566. https://doi.org/10.1007/s001650050057

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

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)*.

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 *POPL*. 385–399. https://doi.org/10.1145/2837614.2837635

Jacob R Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R Wilcox, and Xueyuan Zhao. 2020. Armada: low-effort verification of high-performance concurrent programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 197–210.

John C Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 55–74.

Thomas Schreiber. 1997. Auxiliary Variables and Recursive Procedures. In *Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development (TAPSOFT '97)*. Springer-Verlag, Berlin, Heidelberg, 697–711.

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

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.

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