Conditional Contextual Refinement

YOUNGJU SONG, Seoul National University, Korea & MPI-SWS, 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.

CCS Concepts: • **Theory of computation → Logic and verification**; **Separation logic**.

Additional Key Words and Phrases: contextual refinement, separation logic, Coq, verification

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

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 [Klein et al. 2009; Appel 2014; Gu et al. 2011, 2016]. 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

Authors' addresses: Youngju Song, Seoul National University, Korea & MPI-SWS, SIC, 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, SIC, Germany, msammler@mpi-sws.org; Derek Dreyer, MPI-SWS, SIC, Germany, dreyer@mpi-sws.org.

2475-1421/2023/1-ART39
https://doi.org/10.1145/3571232

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 $\{P\}\ C\ \{Q\}$. 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 $M \sqsubseteq S$.

[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.

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 [Liang and Feng 2016; Turon et al. 2013; Gäher et al. 2022; Frumin et al. 2021a] 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.

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 ($I \sqsubseteq_{\mathsf{ctx}} S$), 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 $I \sqsubseteq_{\mathsf{ctx}} S$ *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 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 $\mathtt{k} \mapsto_{\mathtt{Map}} \mathtt{v}$, asserting that the key `k` is a valid entry of the map and stores the value `v`. With $\mathtt{k} \mapsto_{\mathtt{Map}} \mathtt{v}$, 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 $\mathtt{k} \mapsto_{\mathtt{Map}} \mathtt{0}$ for each entry in the map. Note that the *exclusive token* $\boxed{\textit{pending}}$ is consumed when calling `init` and thus encodes that `init` can only be called once. Then `get(k)` returns `v` given $\mathtt{k} \mapsto_{\mathtt{Map}} \mathtt{v}$, and `set(k,v)` updates $\mathtt{k} \mapsto_{\mathtt{Map}} \mathtt{w}$ to the new value `v`. Note that `set_by_user(k)` updates $\mathtt{k} \mapsto_{\mathtt{Map}} \mathtt{w}$ to an unknown value that is 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 $I_{\mathtt{Map}}$ [Jung et al. 2018]. On the minus side, the

```
(* 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.

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 $I_{\mathtt{Map}}$ using a more abstract program $A_{\mathtt{Map}}$ (the middle column of Fig. 1), which fully captures the observable behavior of $I_{\mathtt{Map}}$. Specifically, this abstraction $A_{\mathtt{Map}}$ 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 $I_{\mathtt{Map}}$ incrementally, in a stepwise fashion. For example, as we will see shortly, a refinement proof of $I_{\mathtt{Map}}$ against $A_{\mathtt{Map}}$, denoted $I_{\mathtt{Map}} \sqsubseteq A_{\mathtt{Map}}$, can be established by introducing an intermediate abstraction $M_{\mathtt{Map}}$ and transitively composing the proofs of $I_{\mathtt{Map}} \sqsubseteq M_{\mathtt{Map}}$ and $M_{\mathtt{Map}} \sqsubseteq A_{\mathtt{Map}}$. 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 $I_{\mathtt{Map}}$ would raise errors while those in $A_{\mathtt{Map}}$ would not.) This condition is of course precisely what the separation logic specification for $I_{\mathtt{Map}}$ enforces.

Conditional contextual 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 $S_{\mathtt{Map}} \vdash I_{\mathtt{Map}} \sqsubseteq A_{\mathtt{Map}}$, which establishes that $I_{\mathtt{Map}}$ refines $A_{\mathtt{Map}}$ under the condition that the module is used according to the separation logic spec $S_{\mathtt{Map}}$. 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 $I_{\mathtt{CL}}$, an abstraction $A_{\mathtt{CL}}$, and conditions $S_{\mathtt{CL}}$. Then we want to modularly verify conditional

refinement for CL only relying on the separation logic specification S_{Map} of Map and *without* needing to reason directly about Map's implementation I_{Map} or abstraction A_{Map}. In other words, we want to prove $S_{\mathsf{Map}} \cup S_{\mathsf{CL}} \vdash I_{\mathsf{CL}} \sqsubseteq A_{\mathsf{CL}}$, which is then composed with $S_{\mathsf{Map}} \vdash I_{\mathsf{Map}} \sqsubseteq A_{\mathsf{Map}}$ to obtain $S_{\mathsf{Map}} \cup S_{\mathsf{CL}} \vdash I_{\mathsf{CL}} \circ I_{\mathsf{Map}} \sqsubseteq A_{\mathsf{CL}} \circ A_{\mathsf{Map}}$ (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 $S_{\mathsf{Map}} \vdash I_{\mathsf{Map}} \sqsubseteq A_{\mathsf{Map}}$ via the following intermediate abstraction M_{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 behaviors, if k is out of range. This facilitates a decomposition of the refinement into two steps.

In the first step, we show that thanks to the range checking, I_{Map} refines M_{Map} as long as init is called at most once, a condition that is enforceable by the following simple specification S^0_{Map}:

$$\begin{array}{lll} \forall \mathtt{sz}.\ \{\boxed{\mathit{pending}_0}\}\ \mathtt{init(sz)} & & \{\top\} \\ \forall \mathtt{k}\ \mathtt{v}. & \{\top\}\ \mathtt{get(k)}, \mathtt{set(k,v)}, \mathtt{set\_by\_user(k)} & \{\top\} \end{array}$$

Here, $\mathit{pending}_0$ is an exclusive token like *pending*, which is used in a similar manner as in the original S_{Map}. (The motivation for differentiating between $\mathit{pending}_0$ and *pending* will be clarified in §2.3). The verification of $S^0_{\mathsf{Map}} \vdash I_{\mathsf{Map}} \sqsubseteq M_{\mathsf{Map}}$ then 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 functional correctness properties specified in S_{Map}.

In the second step, we show that M_{Map} refines A_{Map} under S_{Map}.[2] This 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.

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 $\sqsubseteq_{\mathrm{ctx}}$, between wrapped modules:

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

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:

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

[2] To be precise, we prove that the *wrapped* module $\langle\, S^0_{\mathsf{Map}} \vdash M_{\mathsf{Map}} \,\rangle$ refines A_{Map} under S_{Map}. Wrapping is discussed below.

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 implicit 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 [Song et al. 2022]. Finally, we present an evaluation for our development (§6), discuss related work (§7) and future directions (§8).

2 MAIN IDEAS OF CCR

In this section, we will explain the key ideas behind the central mechanism of CCR, namely the wrapper $\langle S \vdash M \rangle$. Toward this end, we will show (*i*) how the wrapper $\langle S \vdash M \rangle$ is defined (*i.e.,* how the implementation M is instrumented so as to enforce the pre- and postconditions of S operationally), and (*ii*) how we reason about the wrapper in conjunction with a simulation argument in order to establish conditional contextual refinement. Specifically, we will demonstrate that such a conditional refinement indeed enjoys the promised properties: *modular reasoning* (in the sense that each module can be verified independently with separation logic pre- and postconditions), and *incremental verification* in the sense that the verification of each module can be decomposed into multiple stepwise refinements. Finally, we will also present (*iii*) a (global) adequacy theorem—we call it the **Wrapper Elimination Theorem (WET)**—which establishes that the wrappers we introduce as part of conditional contextual refinement proofs can be safely erased at the level of a whole-program verification.

In §2.1, we first discuss these three points with a simplified wrapper that only involves *pure* conditions (*i.e.,* without involving separation logic). Then, in §2.2–§2.4, we move on to the more complex and interesting situation where the wrapper enforces *separation logic* conditions as well.

2.1 Stateless Conditional Refinement

To see how one can encode pure conditions, consider the following contrived yet illustrative example consisting of two function implementations ($I_{\texttt{Sq}}$, $I_{\texttt{Main}}$) and their abstractions ($A_{\texttt{Sq}}$, $A_{\texttt{Main}}$):

```
(* module I_Sq *)
def is_sq(x: int): int ≡
  if (x < 0) error()
  var r := ...
  return r
```

```
(* module A_Sq *)
def is_sq(x: int): int ≡

  var r := ...
  return r
```

```
(* module I_Main *)
def main() ≡
  var x := 16
  var r := is_sq(x)
  output(r)
```

```
(* module A_Main *)
def main() ≡
  var x := 16
  var r := is_sq(x)
  output(1)
```

The function `is_sq(x)` checks if `x` is a square number (elided here in the `...` part), and returns 1 if so and 0 otherwise. In its implementation $I_{\mathtt{Sq}}$, if `x` is negative, it calls the system call *error*, but this check is eliminated in its abstraction $A_{\mathtt{Sq}}$. The `main` function 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, let us see whether we can prove $I_{\mathtt{Sq}} \sqsubseteq_{\mathrm{ctx}} A_{\mathtt{Sq}}$ as a standard *unconditional* contextual refinement. For this, we would need to show that for any value of the argument `x`, the two implementations of `is_sq(x)` in $I_{\mathtt{Sq}}$ and $A_{\mathtt{Sq}}$ are related by the simulation relation $\precsim$ defined by the following rules:

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

Here, $\mathbb{T}$ denotes the "target" (or lower-level) side of the refinement, and $\mathbb{S}$ the "source" (or higher-level) side of the refinement. $\mathbb{T} \hookrightarrow \mathbb{T}'$ 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 on either side and then continue to 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 one can easily see, the simulation between $I_{\mathtt{Sq}}$ and $A_{\mathtt{Sq}}$ 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 will therefore place a *condition* on the contextual refinement. Furthermore, to demonstrate the potential for incremental verification of this example (even though it is not really needed in such a simple example), we will consider two possible conditional specifications. The first is the following spec, $S^0_{\mathtt{Sq}}$, which is the simplest possible condition ensuring that $I_{\mathtt{Sq}}$ refines $A_{\mathtt{Sq}}$ (by ruling out the case where $\mathtt{x} < 0$):

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

The second is the following spec, $S_{\mathtt{Sq}}$, which fully describes the behavior of the `Sq` module and is thus an ideal spec for other modules in the program to rely on:[3]

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

Our goal now is to prove the conditional contextual refinement $S_{\mathtt{Sq}} \vdash I_{\mathtt{Sq}} \sqsubseteq A_{\mathtt{Sq}}$, which establishes that $I_{\mathtt{Sq}}$ refines $A_{\mathtt{Sq}}$ when used by contexts that respect the specification $S_{\mathtt{Sq}}$. Recall that, as discussed in §1.2, $S_{\mathtt{Sq}} \vdash I_{\mathtt{Sq}} \sqsubseteq A_{\mathtt{Sq}}$ is encoded as an ordinary contextual refinement $I_{\mathtt{Sq}} \sqsubseteq_{\mathrm{ctx}} \langle\, S_{\mathtt{Sq}} \vdash A_{\mathtt{Sq}} \,\rangle$, where the "source" side of the refinement *wraps* the abstract implementation $A_{\mathtt{Sq}}$ with the spec $S_{\mathtt{Sq}}$ using the "wrapper" $\langle\, S_{\mathtt{Sq}} \vdash A_{\mathtt{Sq}} \,\rangle$. Our proof strategy is then to prove this refinement, but using the wrapper with $S^0_{\mathtt{Sq}}$ as an intermediate step:

$$I_{\mathtt{Sq}} \sqsubseteq_{\mathrm{ctx}} \langle\, S^0_{\mathtt{Sq}} \vdash A_{\mathtt{Sq}} \,\rangle \sqsubseteq_{\mathrm{ctx}} \langle\, S_{\mathtt{Sq}} \vdash A_{\mathtt{Sq}} \,\rangle$$

Before proving this, though, let us first explain how the wrappers are encoded.

Encoding conditional wrappers. We encode the wrapper $\langle\, S_{\mathtt{Sq}} \vdash A_{\mathtt{Sq}} \,\rangle$ following the approach of Refinement Calculus [Back and Wright 2012] (see §7 for a more detailed comparison). The idea is to encode the pre- and postconditions via **`assume`** and **`assert`** statements. Concretely, $\langle\, S^0_{\mathtt{Sq}} \vdash A_{\mathtt{Sq}} \,\rangle$

[3]Note that the "r." in the postcondition is a binder for the return value of `is_sq(x)`.

and $\langle\, S_{\mathsf{Sq}} \vdash A_{\mathsf{Sq}} \,\rangle$ are encoded as follows.

```
(* ⟨ S⁰_Sq ⊢ A_Sq ⟩ *)
def is_sq(x: int): int ≡
  assume(x ≥ 0)
  var r := ...
  assert(⊤)
  return r
```

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

The wrapper adds an **assume** with the precondition at the start of the function and an **assert** with the postcondition at the end. The behavior of **assume** and **assert** is formally described by the following simulation rules:

$$\text{(ASMR)}\ \frac{P \implies \mathbb{T} \precsim \mathbb{S}}{T \precsim \mathbf{assume}(P); \mathbb{S}} \qquad \text{(ASTR)}\ \frac{P \qquad \mathbb{T} \precsim \mathbb{S}}{\mathbb{T} \precsim \mathbf{assert}(P); \mathbb{S}}$$

$$\text{(ASML)}\ \frac{P \qquad \mathbb{T} \precsim \mathbb{S}}{\mathbf{assume}(P); \mathbb{T} \precsim \mathbb{S}} \qquad \text{(ASTL)}\ \frac{P \implies \mathbb{T} \precsim \mathbb{S}}{\mathbf{assert}(P); \mathbb{T} \precsim \mathbb{S}}$$

The intuition behind these rules and their use in our wrappers is easiest to grasp from considering the case—covered by rules (ASMR) and (ASTR)—where the **assume** or **assert** appears on the source (right-hand) side of a refinement. In that case, **assume**(P) lets one assume P, which is why we use **assume** in the encoding of wrappers to model preconditions; whereas **assert**(P) turns P into a proof obligation, which is why we use **assert** in the encoding of wrappers to model postconditions. Dually, as shown in rules (ASML) and (ASTL), these operators swap their roles (**assume** becoming a proof obligation and **assert** becoming an assumption) when appearing on the target (left-hand) side of a refinement. These rules are validated w.r.t. a trace-based model of computation in §3.2. For now, however, it is easiest to understand the behavior of **assume** and **assert** axiomatically in terms of the refinements they enable.

Proving refinement incrementally. As explained above, we are going to prove the desired refinement for the Sq module in two stages: $I_{\mathsf{Sq}} \sqsubseteq_{\mathsf{ctx}} \langle\, S^0_{\mathsf{Sq}} \vdash A_{\mathsf{Sq}} \,\rangle$ 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 $I_{\mathsf{Sq}} \sqsubseteq_{\mathsf{ctx}} \langle\, S_{\mathsf{Sq}} \vdash A_{\mathsf{Sq}} \,\rangle$.

For the first refinement, $I_{\mathsf{Sq}} \sqsubseteq_{\mathsf{ctx}} \langle\, S^0_{\mathsf{Sq}} \vdash A_{\mathsf{Sq}} \,\rangle$, the intuitive idea of the proof is that the presence of the precondition `x ≥ 0` in S^0_{Sq} ensures that the dynamic check `x ≥ 0` in I_{Sq} succeeds. More formally, the proof proceeds 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 step without any interesting reasoning we can reach after **var** `r := ...` with the same value for `r` since the implementations on 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`.

For the second refinement $\langle\, S^0_{\mathsf{Sq}} \vdash A_{\mathsf{Sq}} \,\rangle \sqsubseteq_{\mathsf{ctx}} \langle\, S_{\mathsf{Sq}} \vdash A_{\mathsf{Sq}} \,\rangle$, the assumptions on the two sides of the refinement match, so the only interesting part is showing that the result satisfies the postcondition in S_{Sq}. More formally, the proof proceeds as follows: for any value of `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 **var** `r := ...` with the same value for `r` by applying (STL) and (STR) in lock step; by (ASTL) we can skip **assert**`(⊤)`; then by (ASTR) we need to prove `r = 1 ⟺ ∃i. x = i * i` holds, which means basically proving 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 consider what happens when we want to verify the `Main` module (see the beginning of this section), which is a client of the `Sq` module. With the spec $S_{\texttt{Sq}}$ in hand, we can verify `Main` *modularly* (*i.e.,* relying only on $S_{\texttt{Sq}}$ and without needing to have access to its implementation). To do so, we will prove the conditional refinement $I_{\texttt{Main}} \sqsubseteq_{\text{ctx}} \langle\, S_{\texttt{Sq}} \cup S_{\texttt{Main}} \vdash A_{\texttt{Main}} \,\rangle$, where the source side of the refinement wraps the abstraction $A_{\texttt{Main}}$ with two specs: the spec $S_{\texttt{Main}} = \{\ \{\top\}\ \texttt{main()}\ \{\top\}\ \}$ for `Main`, as well as the spec $S_{\texttt{Sq}}$ that is assumed for `Sq`. This wrapping is encoded 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 $I_{\texttt{Main}} \sqsubseteq_{\text{ctx}} \langle\, S_{\texttt{Sq}} \cup S_{\texttt{Main}} \vdash A_{\texttt{Main}} \,\rangle$, by (ASTR) the condition $\texttt{x} \geq \texttt{0}$ needs to be *proven* (this is trivial since $\texttt{x} = 16$), and by (ASMR) the condition $\texttt{r} = 1 \iff \exists i.\ \texttt{x} = i * i$ can be *assumed*, from which $\texttt{r} = 1$ follows (since $\texttt{x} = 16 = 4 * 4$); thus, both sides of the refinement call *output*(1), and the proof is done.

Eliminating wrappers. Now that we have verified the `Sq` module and the `Main` module separately, we want to be able to put the proofs together into a verification of the whole program—*i.e.,* to prove $I_{\texttt{Sq}} \circ I_{\texttt{Main}} \sqsubseteq_{\text{beh}} A_{\texttt{Sq}} \circ A_{\texttt{Main}}$, where $\sqsubseteq_{\text{beh}}$ is a notion of whole-program behavioral refinement (roughly, trace refinement—see §3.2). To do so, we first rely on the *horizontal* compositionality of contextual refinement, which says that we can compose our proofs of $I_{\texttt{Sq}} \sqsubseteq_{\text{ctx}} \langle\, S_{\texttt{Sq}} \vdash A_{\texttt{Sq}} \,\rangle$ and $I_{\texttt{Main}} \sqsubseteq_{\text{ctx}} \langle\, S_{\texttt{Sq}} \cup S_{\texttt{Main}} \vdash A_{\texttt{Main}} \,\rangle$ to obtain a refinement for the linked program (where $\circ$ denotes linking):

$$I_{\texttt{Sq}} \circ I_{\texttt{Main}} \sqsubseteq_{\text{ctx}} \langle\, S_{\texttt{Sq}} \vdash A_{\texttt{Sq}} \,\rangle \circ \langle\, S_{\texttt{Sq}} \cup S_{\texttt{Main}} \vdash A_{\texttt{Main}} \,\rangle$$

And since these are closed programs, this implies also the same with $\sqsubseteq_{\text{beh}}$ instead of $\sqsubseteq_{\text{ctx}}$:

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

But this is not quite what we want yet, because the source side of the refinement is cluttered with wrappers. Thus, to get to our end goal, we need to eliminate the wrappers by proving:

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

Fortunately, this falls out as an instance of a more general "adequacy" result of CCR, which we call the **Wrapper Elimination Theorem (WET)**.

The intuition behind the WET is simple: we should be able to eliminate these wrappers because we have already shown that both modules satisfy their specs. A bit more formally, the key idea of the proof is that in the linked program $\langle\, S_{\texttt{Sq}} \vdash A_{\texttt{Sq}} \,\rangle \circ \langle\, S_{\texttt{Sq}} \cup S_{\texttt{Main}} \vdash A_{\texttt{Main}} \,\rangle$, every **assume**$(P)$ will get executed immediately after a corresponding **assert**(P)[4]: either P is the precondition of a function `f` and the **assume**(P) occurs at the beginning of the body of `f`, in which case the caller of `f` must have done **assert**(P) right before calling it; or P is the postcondition of a function `f` and the **assume**(P) occurs right after a call to `f` (on the caller's side), in which case the body of `f` must have done **assert**(P) right before returning to the caller. Thus, both can be eliminated by repeatedly applying the following refinement (where K is an arbitrary evaluation context):

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

This refinement is easily provable by applying (ASTL) followed by (ASML) at the condition statements and otherwise applying (STL) and (STR) in lock step. CCR includes the machinery to automatically apply this cancellation and thus eliminate the wrappers.

[4]Except for the trivial precondition $\top$ of `Main`.

Note that the Wrapper Elimination Theorem is only applicable to closed programs (and behavioral refinement)—*not* open programs and contextual refinement—because for open programs it is not sound in general to eliminate the conditions on the context. We describe the WET more formally in Theorem 4.1 (§4.2).

2.2 Stateful Conditional Refinement via Separation Logic

We have just given an overview of how CCR works in the simple case where pre- and postconditions in specs are pure propositional formulae. In this and the following subsections, we describe how to generalize this technique to handle conditions expressed in *separation logic*.

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. Recall, for instance, the example from §1.2, in which we wanted to prove $I_{\mathsf{Map}} \sqsubseteq_{\mathsf{ctx}} \langle\, S^0_{\mathsf{Map}} \vdash M_{\mathsf{Map}} \,\rangle$. The wrapper on the right will be encoded by adding ASSUME and ASSERT statements, so that the `init` function (for example) will look *roughly* as follows:

$$\texttt{def init(sz: int)} \equiv \mathrm{ASSUME}(\boxed{\mathit{pending}_0}); \texttt{size := sz;}\ \mathrm{ASSERT}(\top)$$

This may look simple enough, but the devil is in the details: in particular, what does it even mean—operationally—to assume or assert a separation logic condition?! Before we can explain this, though, let us first begin by giving a short review of how separation logic assertions are modeled.

Model of separation logic. Consider the assertion $\boxed{\mathit{pending}_0}$. This assertion denotes ownership of an exclusive *resource* called $\mathit{pending}_0$, and in some sense the whole point of separation logic is to provide a rich set of proof principles for reasoning modularly about ownership of such resources.

In separation logic, resources are usually modeled as (variants of) *Partial Commutative Monoids* (PCMs). For our purposes, a PCM Σ[5] 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) $\forall a, b.\, \mathcal{V}(a+b) \implies \mathcal{V}(a)$. Since the PCMs can be naturally composed via Cartesian product, each module can pick its own PCM and then the whole system can be instantiated with the *global PCM*, which is just a product of the PCMs used by each module.

A PCM Σ yields a notion of separation logic proposition which we call $\mathbf{rProp}_\Sigma$[6], which 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 $a \in \Sigma$, and a $\mathbf{Prop}$ R, the connectives in our example have the following definitions (where $r \geq a \triangleq \exists b. r = a + b$):

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

For instance, in our running example, we use a PCM Σ^0_{Map} (for S^0_{Map}) with the following elements:

$$\mathit{pending}_0 \mid \varepsilon \mid \lightning$$

Here, 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 $\mathit{pending}_0 + \mathit{pending}_0$ to result in $\lightning$; thus, $\boxed{\mathit{pending}_0} * \boxed{\mathit{pending}_0}$ is false, which is the essence of what is meant when we say that $\mathit{pending}_0$ is an "exclusive" resource: if one module in the program owns $\mathit{pending}_0$, then no other module can own another copy.

With our resource model in hand, we can state the fundamental invariant of separation logic:

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

This invariant is the core rely/guarantee principle of separation logic. Specifically, a user of separation logic can **rely** on the summation of all current resources being valid, which means that if they are verifying a module which locally owns a resource r, then they know that whatever "frame"

[5]In the development, we use a custom version of resource algebra ([Jung et al. 2018]) without step-indexing.
[6]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 *)
```

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

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

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

```
(* ⟨ S⁰_Map ⊢ M_Map ⟩ *)
private map := (fun k => 0)
private size := 0
private mrs: Σ := ε

def init(sz: int) ≡
  var (frs, ctx) := (ε, ε)
  ASSUME(pending₀)
  size := sz
  ASSERT(⊤)

def get(k: int): int ≡
  var (frs, ctx) := (ε, ε)

  ASSUME(⊤)
  assume(0 ≤ k < size)
  var r := map[k]
  ASSERT(⊤)
  return r
```

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

private mrs: Σ := •λ_.None

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
```

Fig. 2. The implementation and condition-wrapped abstractions for Map (excerpt).

resource f is owned by the rest of the program, it must be the case that r is "compatible" with f (*i.e.*, $\mathcal{V}(r + f)$). At the same time, they must also **guarantee** that if they update the module's local resource to r', then they can only do so if r' remains compatible with f (*i.e.*, $\mathcal{V}(r' + f)$). This is known in the separation logic literature as a "frame-preserving update" [Jung et al. 2018].

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 do we define ASSUME and ASSERT?

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 is the resource owned privately by the current *module*—it is used to state invariants about the private state of the module. This resource is scoped module-locally.
- A *function resource* `frs`: This is the resource owned privately by the current *function*—it is used when reasoning about function calls (to *other* functions) to keep track of the local resources that should be framed around the function call. This resource is scoped function-locally.
- A *call/return resource* σ: This is the resource that is transferred to a function when calling it, or back from it when the function returns. 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 "frame"—it represents the summation of all other resources owned by other modules/functions besides the one we are currently verifying.

Now onto the second question (ii): 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.,* $\boxed{\textit{pending}}$) and an ASSERT statement to guarantee the postcondition (*i.e.,* $\ast_{k\in[0,sz)}\ \mathsf{k} \mapsto_{\mathsf{Map}} \mathtt{0}$). 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.[7] 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 are ready to 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 (resp. caller) side will be matched by a ASSERT on the caller (resp. callee) side that directly precedes it in the execution of the program. (These matching ASSUME/ASSERT pairs will eventually be cancelled out with the WET as described in §2.1.) With this in mind, let us look at the definitions of ASSUME and ASSERT in Fig. 2, which employ two new operators, **take** and **choose**. Intuitively, the lines L1-L4 of ASSUME(*Cond*) **take** a call/return resource σ that the callee (resp. caller) is *receiving* from the caller (resp. callee) (L1), as well as a context resource `ctx` representing the "frame" (L3) and *assume* that σ satisfies *Cond* (L2) and that the summation of all resources is valid (L4). Dually, the lines R1-R4 of ASSERT(*Cond*) **choose** a call/return resource σ that the caller (resp. callee) is *sending* to the callee (resp. caller) (R1), as well as updated values for the caller's (resp. callee's) module- and function-local resources, `mrs` and `frs` (R3), so long as we can *guarantee* that σ satisfies *Cond* (R2) and that the summation of all (updated) resources remains valid (R4).

We will soon see how these definitions of ASSUME and ASSERT play out when proving conditional contextual refinements, but we first need to understand **take** and **choose** more formally—*i.e.,* what these operators do and how they allow resource transfer between the caller and callee.

Implicit value passing via dual non-determinism. As we described above, the definitions of ASSUME and ASSERT make use of **choose** and **take** in order to transfer resources back and forth between caller and callee. The reader may wonder, however, why we don't simply pass the resources as explicit arguments instead of introducing these new **choose** and **take** operators. Indeed, a natural idea would be to add an additional argument to each function that corresponds to the call/return resource σ. However, recall that we are establishing contextual refinement: if we are making any change to the argument/return value, the context will be able to differentiate it and thus contextual refinement cannot hold. For instance, the following refinement where the source adds explicit σ argument does not hold

$$\texttt{def f(x) } \equiv \texttt{ 10} \not\sqsubseteq_{\mathsf{ctx}} \texttt{ def f(x, } \sigma\texttt{) } \equiv \texttt{ 10}$$

because of the following context: `def bad_ctx() ≡ f(x, ε)`. This context linked with the target side of the refinement leads to an undefined behavior since the number of arguments does not match, but it can be linked with the source side without a problem.

Thus, what we need here is a mechanism that effectively gives an *illusion* of passing an additional resource argument, but where the resource argument is not actually passed as a parameter. At a high level, our insight is the following: As we have seen in §2.1, **assume** and **assert** can be seen as

[7] To be precise, the initial resource of each module (ε for M_{Map} and $(\bullet\lambda\_.\mathsf{None})$ for A_{Map}) should be given as an additional parameter to the wrapper. For brevity, we will omit them in the wrapper notation when they are made explicit in the figure.

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. However, 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!

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, which are analogous to the ones for **assume** and **assert**:

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

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

The rules, which we will validate w.r.t. an underlying trace model in §3.2, can be interpreted as follows. 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.

2.3 Incremental and Modular Verification of the Motivating Example

To properly illustrate how **take** and **choose** work, we now revisit the example from §1.2 and demonstrate our wrappers in action: we will sketch how to incrementally prove $I_{\texttt{Map}} \sqsubseteq \langle\, S^0_{\texttt{Map}} \vdash M_{\texttt{Map}} \,\rangle \sqsubseteq \langle\, S_{\texttt{Map}} \vdash A_{\texttt{Map}} \,\rangle$, and how to modularly verify a client of this library using the separation logic spec, $S_{\texttt{Map}}$. In the course of doing so, we will also see the role that each of the resources in the encoding of ASSUME and ASSERT plays in the refinement verification. But first, before we dive into the proof, let us set up some preliminaries.

Simulation with relational invariant. Since modules are now equipped with their own private states, we extend the previous (stateless) simulation relation with a notion of *module-local relational invariant*. Specifically, when proving a simulation between a pair of modules, one can fix up front a relation *Inv* between the possible private states of the two (*i.e., Inv* $\in \mathcal{P}(\text{state}_{\text{tgt}} \times \text{state}_{\text{src}})$). Then, the simulation (i) allows one to rely on *Inv* whenever this pair of modules acquires control (*i.e.,* at the beginning of the function and after a function call), and (ii) obligates one to guarantee *Inv* holds whenever this pair of modules releases control (*i.e.,* at the end of the function and before a function call). Note that the private state of the modules includes both their physical private state (*i.e.,* module-local variables, like `data` and `map` in the `Map` example) *and* their module resources (`mrs`, which is a kind of module-local ghost state that ASSUME and ASSERT manipulate). Indeed, as we will see concretely below, a key purpose of the module-local relational invariant *Inv* is to dictate how the modules' private physical and ghost state relate to each other.

First refinement for Map. To prove $I_{\texttt{Map}} \sqsubseteq \langle\, S^0_{\texttt{Map}} \vdash M_{\texttt{Map}} \,\rangle$, we use the following module-local relational invariant Inv_0 between the private states of the two modules (*i.e.,* `data` from the former and `map`, `size`, `mrs` from the latter):

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

The invariant is a disjunction of two cases: the former (*i.e.,* `size` $= 0 \land$ `map` $=$ (`fun k => 0`)) states the relation before `init` is called, and the latter states the relation after `init` is called. In the former case, the $pending_0$ token is not contained within the module resource `mrs`, which means a client of `Map` may have it and can use it to invoke `init`. In the latter case, the module resource `mrs` *must* contain $pending_0$—thus preventing a client from owning it and trying to invoke `init` again—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* $\mapsto_{\texttt{Mem}}$ 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_0, we can prove the refinement for each function by applying the simulation rules and doing a case analysis on Inv_0. We first note that, for the functions in `Map` module, one can completely ignore the `frs`; it will be initialized as ε and remain the same all the time (*i.e.,* we will always choose it to be ε in R3). To see where `frs` gets used, see the proof of `Main` below.

Consider `init`. At the beginning, by the relational invariant we have `mrs` which satisfies Inv_0, and by simulation argument we are given some resource σ (L1, TKR) which satisfies $\boxed{pending_0}$ (L2, ASMR), so $\sigma \geq pending_0$. We are then also given $\mathcal{V}(\texttt{mrs} + \sigma)$ (L4, ASMR), which means that `mrs` cannot *also* contain $pending_0$, so it must be in the uninitialized state (left disjunct). At the end of `init`, we update σ to ε and `mrs` to $pending_0$ (R1, R3, CHR) so that Inv_0 and the postcondition are satisfied (R2, ASTR). We also check that this update maintains the validity of the sum of all resources (L3, TKR, R4, ASTR)—*i.e.,* that it is a "frame-preserving update".

For `get` and `set`, a high-level proof is as follows: we know that the module is in the initialized state (*i.e.,* the latter case of Inv_0) since in the former case with `size = 0`, the range checking **`assume`**`(0 ≤ k < size)` fails and thus the refinement holds trivially. Then, thanks to the ownership `data` $\mapsto_{\texttt{Mem}}$ `map[0 : size]`, we can prove that both source and target (i) (in case of `get`) retrieve the same value, and (ii) (in case of `set`) update `data` and `map` equivalently, reestablishing Inv_0.

Second refinement for Map. We now explain how to prove $\langle\, S^0_{\texttt{Map}} \vdash M_{\texttt{Map}} \,\rangle \sqsubseteq \langle\, S_{\texttt{Map}} \vdash A_{\texttt{Map}} \,\rangle$, the second refinement. The structure of the proof is largely similar to that of the first refinement, but there is one new twist because the precondition of the spec $S^0_{\texttt{Map}}$ on the target side of the refinement poses a bit of a challenge: we must somehow discharge its precondition (in this case, $\boxed{pending_0}$). One way to discharge it would be to simply take $\boxed{pending_0}$ as the precondition of $S_{\texttt{Map}}$ as well: that way, we would get to ASSUME ownership of $pending_0$ on the source side and then use it to discharge the ASSUME on the target. However, if we did that, we could no longer actually *use* the $pending_0$ token in the remainder of the proof, which would be a problem: we would have no way to reason (in *this* proof) that `init` is only called once.

To solve this problem, we define $S_{\texttt{Map}}$ so that its precondition is the separating conjunction of $\boxed{pending_0}$ (the precondition of $S^0_{\texttt{Map}}$) and a second token $pending_1$, which enjoys the same exclusive property (*i.e.,* $pending_1 + pending_1$ is invalid). We then define $pending$ to be $pending_0 + pending_1$. By using $\boxed{pending}$ (instead of $\boxed{pending_0}$) as the precondition for the source side, we ensure that (i) we can discharge the ASSUME on the target side using the $pending_0$ component of $pending$; but (ii) even after giving up $pending_0$, we will still be left with an exclusive token $pending_1$ that (together with the relational invariant we are about to define) we can use to establish that the module is in the uninitialized state. Note, however, that a client of **`Map`** need not know this internal technical detail of how $pending$ is defined: to the client, it is just an exclusive resource.

We define the relational invariant Inv between the private states of the two modules (*i.e.,* $\texttt{map}_\texttt{M}$, `size`, $\texttt{mrs}_\texttt{M}$ from the former and $\texttt{map}_\texttt{A}$, $\texttt{mrs}_\texttt{A}$ from the latter) as follows:

$$[\![\, \boxed{\texttt{mrs}_\texttt{M}} * \ulcorner \texttt{map}_\texttt{M} = \texttt{map}_\texttt{A} \urcorner * \bullet(\texttt{map}_\texttt{A}\langle 0 : \texttt{size}\rangle) * (\ulcorner \texttt{size} = 0 \urcorner \lor \boxed{pending_1}) \,]\!](\texttt{mrs}_\texttt{A})$$

```
(* I_Main *)

def main() ≡


  var sz := 100
  init(sz)
  var k := 42

  r := get(k)
  output(r)
```

```
(* ⟨ S_Map ∪ S_Main ⊢ A_Main ⟩ *)
private mrs: Σ := ε
def main() ≡
  var (frs, ctx) := (ε, ε)
  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(∗_{k∈[0,sz)} k ↦_Map 0)
```

Fig. 3. An implementation and its condition-wrapped abstraction for the client module, `Main`.

Here, let $\mathtt{map}_A\langle 0 : \mathtt{size}\rangle \triangleq$ `(fun k => if (0 ≤ k < size) then Some map(k) else None)`. *Inv* says several things: (i) $\mathtt{mrs}_A \geq \mathtt{mrs}_M$—as we will see shortly, this is needed to discharge the validity condition in the target-side ASSUME; (ii) $\mathtt{map}_M$ and $\mathtt{map}_A$ coincide; (iii) $\mathtt{mrs}_A$ contains the resource $\bullet(\mathtt{map}_A\langle 0 : \mathtt{size}\rangle)$, which means that $\mathtt{k} \mapsto_{\mathtt{Map}} \mathtt{v}$ is only valid for $0 \leq k < \mathtt{size}$ and its value v is equal to $\mathtt{map}_A[\mathtt{k}]$; and (iv) either $\mathtt{size} = 0$ or $\mathtt{mrs}_A$ contains $pending_1$ (analogous to the corresponding condition in Inv_0).

With this invariant in hand, we can prove simulation for each function. At a high level, the proof is straightforward: 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 reasoning steps to rule out failure of the range checking in $M_{\mathtt{Map}}$. The most interesting bit is how the connection between $\mathtt{mrs}_M$ and $\mathtt{mrs}_A$ is handled.

For space reasons, we just sketch the proof of `init`: After executing the ASSUME(*pending*) in the source and quantifying over $\mathtt{mrs}_A$ which satisfies the relational invariant, we learn that $\mathtt{mrs}_A \geq \mathtt{mrs}_M + \bullet(\mathtt{map}_A\langle 0 : 0\rangle)$ along with the validity condition $\mathcal{V}(\mathtt{mrs}_A + pending + \mathtt{ctx}_A)$. We then execute ASSUME in the target by picking σ_M to be $pending_0$, proving (trivially) that it satisfies the precondition *pending*$_0$, picking $\mathtt{ctx}_M$ to be $\mathtt{ctx}_A + pending_1 + \bullet(\mathtt{map}_A\langle 0 : 0\rangle)$, and proving the validity condition for the target—$\mathcal{V}(\mathtt{mrs}_M + \sigma_M + \mathtt{ctx}_M)$—which is implied directly from the assumed validity condition for the source since all we did was shuffle the resources around.

At the end of the function, we execute ASSERT in the target, which gives us the updated $\mathtt{mrs}'_M$ and the validity condition $\mathcal{V}(\mathtt{mrs}'_M + \mathtt{ctx}_M)$ (we ignore the return resource since the target post is $\top$). We then execute ASSERT in the source by picking σ_A to be the resource satisfying $\ast_{k\in[0,\mathtt{size})}\ \mathtt{k} \mapsto_{\mathtt{Map}} 0$, updating $\mathtt{mrs}'_A$ to be $\mathtt{mrs}'_M + pending_1 + \bullet(\mathtt{map}_A\langle 0 : \mathtt{size}\rangle)$, and proving the validity condition $\mathcal{V}(\mathtt{mrs}'_A + \sigma_A + \mathtt{ctx}_A)$. This validity condition is implied by the validity condition from the target and the fact that the allocation of $\ast_{k\in[0,\mathtt{size})}\ \mathtt{k} \mapsto_{\mathtt{Map}} 0$ (and corresponding update to $\bullet(\mathtt{map}_A\langle 0 : \mathtt{size}\rangle)$) are frame-preserving updates (*i.e.,* they preserve validity of composition with any frame context). Finally, we have to reestablish the invariant, which is straightforward since $\mathtt{map}_M$ and $\mathtt{map}_A$ are not modified and $\mathtt{mrs}'_A$ contains $\mathtt{mrs}'_M$, $\bullet(\mathtt{map}_A\langle 0 : \mathtt{size}\rangle)$, and $pending_1$.

Using pre- and postconditions modularly. Let us see now how to reason modularly about a client of Map using $S_{\mathtt{Map}}$. The interesting bit here is that this proof involves an analogue of the "frame rule" of separation logic, which (as we will see shortly) is operationalized in our wrappers via the *function resource* `frs`.

The client module we consider here, given in Fig. 3, consists of a single function `main`. In the implementation $I_{\mathtt{Main}}$, `main` initializes the `Map` module with size 100, retrieves the 42nd value, and outputs the result. In the abstraction $A_{\mathtt{Main}}$, the output value is abstracted into the constant 0. We

define $S_{\texttt{Main}}$ as follows:

$$\{\boxed{\textit{pending}}\}\ \texttt{main()}\ \{\ast_{k\in[0,100)}\ \texttt{k} \mapsto_{\texttt{Map}} \texttt{0}\}$$

In proving the refinement $I_{\texttt{Main}} \sqsubseteq \langle\, S_{\texttt{Map}} \cup S_{\texttt{Main}} \vdash A_{\texttt{Main}} \,\rangle$, there is a point where we need to reason about the call to `get(42)`, which has the following spec:

$$\{42 \mapsto_{\texttt{Map}} \texttt{0}\}\ \texttt{get(42)}\ \{r.\ (\ulcorner r = \texttt{0} \urcorner \wedge 42 \mapsto_{\texttt{Map}} \texttt{0})\}$$

However, at that point in the proof, we know not only that $42 \mapsto_{\texttt{Map}} \texttt{0}$ but that $\texttt{k} \mapsto_{\texttt{Map}} \texttt{0}$ for all $0 \le k < 100$. So, what we want to do is frame $F \triangleq \ast_{k\in[0,42)\cup[43,100)}\ \texttt{k} \mapsto_{\texttt{Map}} \texttt{0}$ around the call to `get(42)`, effectively relying on the spec:

$$\{F * 42 \mapsto_{\texttt{Map}} \texttt{0}\}\ \texttt{get(42)}\ \{r.\ F * (\ulcorner r = \texttt{0} \urcorner \wedge 42 \mapsto_{\texttt{Map}} \texttt{0})\}$$

We represent this "frame" in CCR using the *function resource* `frs`, which is a function-scoped local variable. Concretely, R3 of the Fig. 2 says that when executing the ASSERT($42 \mapsto_{\texttt{Map}} \texttt{0}$) in the source (as precondition of `get(42)`), one needs to split its "current resources" (resources that are disjoint from `ctx`) into: (i) σ satisfying the condition $42 \mapsto_{\texttt{Map}} \texttt{0}$, (ii) `mrs` satisfying the relational invariant, and (iii) `frs`, which will be defined as the frame F above. Then, when executing the following ASSUME(Q) in the source (where Q is the postcondition $r = \texttt{0} \wedge 42 \mapsto_{\texttt{Map}} \texttt{0}$), one is given a new σ' (satisfying Q), a new module resource `mrs`$'$ (satisfying the relational invariant), a new context resource `ctx`$'$, and the fact that `mrs`$' + $ `frs` $+ \sigma' +$ `ctx`$'$ is valid. The "current resources" are thus reconstituted, with `frs` and σ' coming together to form ownership of the full key map again.

One point that we glossed over in the above explanation is how we handle auxiliary variables in function specifications. In the case of `get`, for instance, the actual spec for `get` quantifies universally over the value `v` that is stored at the `k`-th index. Here, `v` is an *auxiliary* variable in the spec because it does not appear in the code (`get(k)`). We handle such a universally quantified auxiliary variable again using dual non-determinism: on the caller side, that means we get to non-deterministically **`choose`** the right value for `v` before ASSERTing the precondition. Dually, on the callee side, `v` is chosen using **`take`** (see the body of `get`).

2.4 Wrapper Elimination

Recall that in §2.1, we concluded our discussion of stateless wrapper with a global adequacy theorem, the Wrapper Elimination Theorem (WET), which shows how wrappers can be eliminated once we have a closed whole program (*e.g.*, $\langle\, S \vdash M_1 \,\rangle \circ \langle\, S \vdash M_2 \,\rangle \sqsubseteq_{\text{beh}} M_1 \circ M_2$). Similarly, we conclude this section by sketching the proof of WET for separation logic wrappers. For expository purposes, we show the proof in multiple gradual refinements.

In the first step, we eliminate R1/R2 and L1/L2 of neighboring ASSERT(P) and ASSUME(P) statements (Fig. 2) with *local* reasoning as follows:

$$\begin{array}{lll}
 & K[\text{ASSERT}(\mathit{Cond});] & \text{ASSUME}(\mathit{Cond})] \\
\equiv & K[\textbf{var}\ \sigma := \textbf{choose}(\Sigma);\ \textbf{assert}(\mathit{Cond}\ \sigma);\ R34[\sigma];\ \textbf{var}\ \sigma' := \textbf{take}(\Sigma);\ \textbf{assume}(\mathit{Cond}\ \sigma');\ L34[\sigma']] \\
\sqsubseteq_{\text{beh}} & K[\textbf{var}\ \sigma := \textbf{choose}(\Sigma);\ \textbf{assert}(\mathit{Cond}\ \sigma);\ R34[\sigma]; & \textbf{assume}(\mathit{Cond}\ \sigma);\ L34[\sigma]] \\
\sqsubseteq_{\text{beh}} & K[\textbf{var}\ \sigma := \textbf{choose}(\Sigma); & R34[\sigma]; \qquad L34[\sigma]]
\end{array}$$

We first (i) unfold the definitions, where $R34$ and $L34$ refer to the combined lines R3/R4 and L3/L4, (ii) turn implicit value passing between **`choose`** and **`take`** to *explicit* value passing so that the resource σ coincides on both sides (the proof is a simple application of the (TKL) rule), and (iii) eliminate the matching **`assert`** and **`assume`** as done in §2.1.

$$\begin{array}{ll}
\text{fundef}(E) \triangleq \texttt{Any} \to \texttt{itree}\ E\ \texttt{Any} & X|_{cond} \triangleq \text{if } cond \text{ holds, then } X \text{ else } \emptyset \\
\text{E}_\text{P}(X) \triangleq \{\texttt{Choose}\} \uplus \{\texttt{Take}\} \uplus \{\texttt{Obs}\ fn\ arg \mid fn \in \texttt{string}, arg \in \texttt{Any}\}|_{X=\texttt{Any}} & \\
\text{E}_\text{EMS}(X) \triangleq \text{E}_\text{P}(X) \uplus \{\texttt{Call}\ fn\ arg | fn \in \texttt{string}, arg \in \texttt{Any}\}|_{X=\texttt{Any}} \uplus \{\texttt{Put}\ a \mid a \in \texttt{Any}\}|_{X=()} \uplus \{\texttt{Get}\}|_{X=\texttt{Any}} & \\
\text{Mod} \triangleq \left\{(\texttt{init}, \texttt{funs}) \in \texttt{Any} \times (\texttt{string} \xrightarrow{\text{fin}} \text{fundef}(\text{E}_\text{EMS}))\right\} &
\end{array}$$

$$\begin{array}{ll}
\text{Mods} \triangleq \texttt{list Mod} & \circ \in \text{Mods} \to \text{Mods} \to \text{Mods} \triangleq \text{append} \\
Ms \sqsubseteq_\text{beh} Ms' \triangleq \text{Beh}(Ms) \subseteq \text{Beh}(Ms') & M \sqsubseteq_\text{ctx} M' \triangleq \forall C \in \text{Mods}.\ C \circ M \sqsubseteq_\text{beh} C \circ M'
\end{array}$$

Fig. 4. Definitions of module and contextual refinement.

In the second step, we eliminate the remaining R3/R4 and L3/L4 with *global* reasoning as follows:

$$\begin{array}{ll}
 & K[\textbf{var}\ \sigma := \textbf{choose}(\Sigma); R34[\sigma]; L34[\sigma]] \\
\equiv & K[\textbf{var}\ \sigma := \textbf{choose}(\Sigma); (\texttt{mrs}, \texttt{frs}) := \textbf{choose}(\Sigma \times \Sigma);\ \textbf{assert}(\mathcal{V}(\texttt{mrs} + \texttt{frs} + \sigma + \texttt{ctx})); \\
 & \quad \texttt{ctx}' := \textbf{take}(\Sigma);\ \textbf{assume}(\mathcal{V}(\texttt{mrs}' + \texttt{frs}' + \sigma + \texttt{ctx}'))] \\
\sqsubseteq_\text{beh} & K[\textbf{var}\ \sigma := \textbf{choose}(\Sigma); (\texttt{mrs}, \texttt{frs}) := \textbf{choose}(\Sigma \times \Sigma);\ \textbf{assert}(\mathcal{V}(\texttt{allrs}));\ \textbf{assume}(\mathcal{V}(\texttt{allrs}))] \\
\sqsubseteq_\text{beh} & K[\textbf{skip}]
\end{array}$$

In this proof, we maintain the global invariant (see §2.2) that the summation of all module resources in the program is valid. Moreover, while a module is executing, we enforce the invariant that its context resource `ctx` is equal to the summation of the `mrs` and `frs` resources from all *other* modules in the program. Then, when control is transferred to a different module in the program (*e.g.,* at the point where `ctx`′ is updated to **take**(Σ) above), we correspondingly update *that* module's context resource `ctx`′ (TKL) so that the global invariant is maintained. This means that, in the proof above, both `mrs` + `frs` + σ + `ctx` of the **assert**er and `mrs`′ + `frs`′ + σ + `ctx`′ of the **assume**r can be guaranteed to be equal to the summation of all resources in the system at the given moment, named `allrs`. With this invariant in place, we can eliminate the matching **assert** and **assume** as before, and conclude by removing the now-unused **choose**s.

In summary:

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

3 EXECUTABLE MODULE SEMANTICS (EMS)

Before discussing the formal definition of the wrapper and the wrapper elimination theorem in the next section, this section introduces CCR's module system and its semantics.

3.1 Module and Contextual Refinement

As seen in the examples in the previous sections, programs in CCR are organized into *modules* that combine functions with module-local state. We call this module system EMS (Executable Module Semantics). Before we can introduce EMS formally, we must first review *interaction trees* [Xia et al. 2019] which are used extensively in the definition of EMS.

Interaction trees. For a given event type E : **Set** → **Set** and a return type T, an interaction tree of type `itree` E T can 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.

Interaction trees provide the following benefits: (*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.

$$\text{ObsEvent} \triangleq \{(\texttt{Obs}\ \mathit{fn}\ \mathit{arg}, \mathit{ret}) \mid \mathit{fn} \in \texttt{string},\ \mathit{arg}, \mathit{ret} \in \texttt{Any}\}$$

$$\text{Trace} \overset{\text{coind}}{=} \{e :: tr \mid e \in \text{ObsEvent}, tr \in \text{Trace}\} \uplus \{\text{Term}\ v \mid v \in \texttt{Any}\} \uplus \{\text{Diverge}\} \uplus \{\text{Error}\} \uplus \{\text{Partial}\}$$

$$\text{Beh}(Ms) \in \mathbb{P}(\text{Trace}) \triangleq \text{beh}(\text{concat}(Ms)) \qquad \text{concat}(Ms) \in \texttt{itree}\ \text{E}_\text{P}\ \texttt{Any} \triangleq \ldots$$

$$\text{beh} \in \texttt{itree}\ \text{E}_\text{P}\ \texttt{Any} \to \mathbb{P}(\text{Trace}) \triangleq \lambda i.\ \{\text{Partial}\} \cup \{\text{Diverge}\}|_{i \in \text{div}} \cup$$

$$\begin{array}{l}
\quad \textbf{match}\ i\ \textbf{with} \\
\quad \mid \textbf{tau} \gg\!= k \Rightarrow \boxed{\text{beh}(k())} \mid \textbf{choose}(X) \gg\!= k \Rightarrow \bigcup_{x \in X} \boxed{\text{beh}(k(x))} \mid \textbf{take}(X) \gg\!= k \Rightarrow \bigcap_{x \in X} \boxed{\text{beh}(k(x))} \\
\quad \mid \textbf{obs}\ \mathit{fn}\ \mathit{arg} \gg\!= k \Rightarrow \bigcup_{\mathit{ret} \in \texttt{Any}} (\texttt{Obs}\ \mathit{fn}\ \mathit{arg}, \mathit{ret}) :: \boxed{\text{beh}(k(\mathit{ret}))}|_{\text{valid_obs}\ \mathit{fn}\ \mathit{arg}\ \mathit{ret}} \mid \textbf{ret}\ v \Rightarrow \{\text{Term}\ v\}\ \textbf{end}
\end{array}$$

$$\text{div} \in \mathbb{P}(\texttt{itree}\ \text{E}_\text{P}\ \texttt{Any}) \overset{\text{coind}}{=} \{\ \textbf{tau} \gg\!= k \mid k() \in \text{div}\ \} \cup \{\ \textbf{choose}(X) \gg\!= k \mid \exists x \in X.\, k(x) \in \text{div}\ \} \cup \{\ \textbf{take}(X) \gg\!= k \mid \forall x \in X.\, k(x) \in \text{div}\ \}$$

Fig. 5. Definitions of trace and behavior.

Function and module. Now we see how we define the notion of module, given in Fig. 4. The semantic domain of EMS functions fundef(E) is given by meta-level functions that take an arbitrary value (denoted by the type Any) as an argument and return an itree w.r.t. the event type E and the return type Any. (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 E_EMS. 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, The instructions **choose**(X) and **take**(X) are defined as 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. 4 shows formal definition for the (whole-program) behavioral refinement and contextual refinement. We say modules (Mods) to refer to a list of modules and linking $\circ$ of modules is defined as 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 as 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:

$$\begin{array}{ll}
\text{(Vertical)} & I \sqsubseteq_\text{ctx} M \wedge M \sqsubseteq_\text{ctx} A \Rightarrow I \sqsubseteq_\text{ctx} A \\
\text{(Horizontal)} & I_1 \sqsubseteq_\text{ctx} A_1 \wedge I_2 \sqsubseteq_\text{ctx} A_2 \Rightarrow (I_1 \circ I_2) \sqsubseteq_\text{ctx} (A_1 \circ A_2)
\end{array}$$

3.2 Traces and Behavior

As promised, we now present how we define the set of possible traces for a list of modules (Fig. 5).

Traces. To give the notion of behavior, we first define the set of traces, denoted 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: intuitively, erroneous termination is terminating due to an error in the program, while partial termination is due to the user (*e.g.,* by killing the process via Ctrl+C).

Behavior. The behavior Beh(Ms) for modules is defined in two steps. First, we concatenate the computations described in each function semantics to create a single, large itree using the standard concat combinator on interaction trees. Then, we define a set of possible traces for such an itree (beh). The predicate beh(−) is defined by a mixed induction coinduction[8] as follows, where a dashed box denotes coinduction and a solid box denotes induction. For a given itree i, 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 (Obs *fn arg*, *ret*) for each *valid* return value *ret* satisfying valid_obs *fn arg ret*[9]; and (v) if i returns a value v, the normal termination (Term v). Note that the erroneous termination (Error) can only arise from **take**(∅). The divergence predicate div coinductively defines the set of those itrees that take infinite steps without triggering any observable events, as shown in Fig. 5.

Though it is not our main contribution, our definition of beh(−) is novel in the sense that it addresses (possibly) infinite traces and dual non-determinism at the same time. Previous work, to our knowledge, considered dual non-determinism only for finite traces [Back and Wright 2012; Koenig and Shao 2020], or considered (possibly) infinite traces but without dual non-determinism (including the trace interpretation of itree [Xia et al. 2019]).

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**(∅) and **choose**(∅), respectively. Note that beh(**UB**) includes all the traces including Error, while beh(**NB**) includes only Partial. Also we have the following duality:

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

The prefix-closed property holds because Partial appears in the constructor of beh(−) unconditionally (Fig. 5). The postfix-closed property holds because Error never appears explicitly in the definition of beh(−) but can only arise implicitly from executing **take**(∅) = **UB**. In particular, **take**(∅) denotes the set of all traces (*i.e.,* the unit of intersection), so if Error is a possible trace, all other traces must be possible as well.

Next, we define the following operators:

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

For a proposition P, we define **assume** (resp. **assert**) to trigger **UB** (resp. **NB**) if P does not hold. The two unwrap operators **?** and **!** extract the internal value of an option-typed value and result in **UB** resp. **NB** on 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 `itree` E_{EMS}) together with its module-private state (of type `Any`). Specifically, the simulation allows imposing relational invariants, $\mathbb{I}$, on the module-private states of both sides. An invariant $\mathbb{I}$ can depend on

[8] We use Paco library [Hur et al. 2013] in the formalization.

[9] Following CompCert, the parameter predicate valid_obs specifies the possible return values of each observable event.

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

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

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

$$\frac{w_0 \sqsubseteq_{\mathcal{W}} w_1 \qquad \mathbb{I}_{w_1}\ st_\mathtt{t}\ st_\mathtt{s}}{(st_\mathtt{t}, \mathbf{ret}\ r) \lesssim_{w_0} (st_\mathtt{s}, \mathbf{ret}\ r)} \qquad \frac{\mathbb{I}_{w_1}\ st_\mathtt{t}\ st_\mathtt{s} \qquad \forall r, w_2, st'_\mathtt{t}, st'_\mathtt{s}.\ w_1 \sqsubseteq_{\mathcal{W}} w_2 \wedge \mathbb{I}_{w_2}\ st'_\mathtt{t}\ st'_\mathtt{s} \Rightarrow (st'_\mathtt{t}, K_\mathtt{t}\ r) \lesssim_{w_0} (st'_\mathtt{s}, K_\mathtt{s}\ r)}{(st_\mathtt{t}, r \leftarrow \mathbf{call}\ f\ x;\ K_\mathtt{t}\ r) \lesssim_{w_0} (st_\mathtt{s}, r \leftarrow \mathbf{call}\ f\ x;\ K_\mathtt{s}\ r)}$$

Fig. 6. Constructors for our common simulation relation (simplified).

Kripke-style possible worlds, *i.e.,* an arbitrary type $\mathcal{W}$ equipped with a preorder ($\sqsubseteq_{\mathcal{W}}$). With these, the simulation relation $\lesssim_w$[10] at a given world $w \in \mathcal{W}$ is coinductively defined (*i.e.,* as a greatest fixpoint) with constructors (rules) shown in Fig. 6.

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 unchanged from the presentation in §2, and (*iii*) is straightforward. For (*iv*) those constructors are now equipped with worlds following the standard open simulations [Song et al. 2019]. That is, at each call or return one needs to guarantee that $\mathbb{I}$ holds for some future world w_1. In return, at the beginning of a function and after a function call, one can rely on the fact that there is some (future) world w_2 such that $\mathbb{I}$ holds. Note that the simulation relation is meant to imply contextual refinement where the context is completely arbitrary and can be reentrant to the module being verified (*i.e.,* performing mutual recursion). This is precisely the reason why $\mathbb{I}$ needs to be reestablished before all function calls (see the first precondition of the rule for calls). Consequently, CCR supports mutual recursion between different modules.[11]

The common simulation relation satisfies the following adequacy theorem.

Theorem 3.1 (Adequacy). *For a pair of modules $M_\mathtt{t}$ and $M_\mathtt{s}$, a possible world $\mathcal{W}$, and a relational invariant $\mathbb{I}$ w.r.t. $\mathcal{W}$, if we have (i) $\exists w_0.\ \mathbb{I}_{w_0}\ M_\mathtt{t}.\mathtt{init}\ M_\mathtt{s}.\mathtt{init}$, (ii) $dom(M_\mathtt{t}.\mathtt{funs}) = dom(M_\mathtt{s}.\mathtt{funs})$, and (iii) for each pair of function $f_\mathtt{t}$ and $f_\mathtt{s}$ with the same name, $\forall v\ w\ st_\mathtt{t}\ st_\mathtt{s}.\ \mathbb{I}_w\ st_\mathtt{t}\ st_\mathtt{s} \implies (st_\mathtt{t}, f_\mathtt{t}\ v) \lesssim_w (st_\mathtt{s}, f_\mathtt{s}\ v)$, the following holds:*

$$M_\mathtt{t} \sqsubseteq_{ctx} M_\mathtt{s}$$

4 CCR FRAMEWORK, SIMPLIFIED

In this section, we present the CCR framework, which formalizes ideas presented in §2. Specifically, we show how to formally define the wrapper and the WET theorem for a basic (yet expressive

[10] We omit the stuttering index for brevity.

[11] It could seem restrictive that the invariant should be reestablished at every function call: for some function calls that are known to not be reentrant, one may want to temporarily break the invariant. Fortunately, such reasoning is also supported in CCR—without any extension to the core mechanism—by employing a well-known trick from separation logic (*e.g.,* used by the masks of Iris invariants [Jung et al. 2015, 2018]). That is, one can add an exclusive token X as a precondition to the functions of a module M, which means that only functions with ownership of X can call functions of M. So if a function of M calls another function without X in the precondition, there cannot be reentrancy and one does not have to reestablish the invariant of M (technically, this works by adding a disjunction with X to the invariant of M so it can be trivially reestablished by giving up X).

$\mathbf{rProp} \triangleq \Sigma \to \mathbf{Prop}$ 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. 7. Definition of the wrapper.

enough to handle the running example) version of CCR. Additional advanced features will be presented in subsequent sections.

4.1 Condition Wrapped Abstractions

At the heart of CCR framework is the wrapper, $\langle\, S \vdash_\alpha M \,\rangle$. Its formal definition is given in Fig. 7. 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 [Schreiber 1997; Kleymann 1999] is shared between P and Q (*e.g.*, v in the specification 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 v in the specification of `get`. $\mathtt{P}(w)$ resp. $\mathtt{Q}(w)$, given $w \in \mathtt{W}$, specify a separation logic pre- resp. postcondition on the concrete argument resp. return value. A *collection of conditions* $S \in \mathrm{Conds}$ collects such conditions for a finite set of functions. The wrapper $\langle\, S \vdash_\alpha M \,\rangle$ for a module M, conditions S, and an initial module resource α is again a module with its initial private state now paired[12] with the initial module resource α, and its functions wrapped via WrapF. In Fig. 7 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 (they never acutually happen) and get eliminated in the WET.

Inserting conditions. In Fig. 7, 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 (WrapC). WrapF is parametrized by the conditions S for outgoing function calls, the condition $(\mathtt{W},\mathtt{P},\mathtt{Q})$ of the function to wrap, and the definition f of the function. WrapF generates a function that **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

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

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 the events that are interpreted identically, and the state component when it is the unit type.

Now we can discuss lines F2-4 in more detail. 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 **choose**s the auxiliary variable (C1), ASSERTs the precondition (C2), makes the intended function call (C3), ASSUMEs the postcondition (C4), and returns (C5). Lines F3-4 make sure that f accesses the correct private state. 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 definitions of ASSUME and ASSERT in Fig. 7 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 WET theorem (described in §2) that removes those wrappers.

Theorem 4.1 (Wrapper Elimination Theorem (WET)). *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 behaves similarly to a standard unary separation logic. To be specific, we define a special module $\mathrm{Safe}(ns_{\mathrm{in}}, ns_{\mathrm{out}})$, which defines functions with names in ns_{out} that non-deterministically invoke arbitrary functions in ns_{in} with arbitrary arguments an arbitrary (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 lemma holds since the whole program only consists of internal function calls and the precondition ensures that all invoked functions exist. 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 $i \in \{1, \ldots, n\}$, $ns \subseteq dom(I_1.\mathtt{funs}) \uplus \ldots \uplus dom(I_n.\mathtt{funs})$, and an initial resource α to* `main` *satisfying its precondition and $\mathcal{V}(\alpha + \alpha_1 + \ldots + \alpha_n)$,*

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

Proving $I_i \sqsubseteq_{\mathrm{ctx}} \langle S \vdash_{\alpha_i} \mathrm{Safe}(ns, \mathrm{dom}(I_i.\mathtt{funs})) \rangle$ essentially amounts to verifying I_i against S in separation logic, and Corollary 4.4 mirrors the corresponding adequacy result of separation logic.

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 [Song et al. 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.

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

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. One workaround for this problem is to 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, we would like to eliminate such spurious function calls at the top-level.

CCR supports this with the following features: (i) we support a mechanism to specify whether a function call is "cancellable" (defined below) and remove those in the WET, and (ii) we 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 and thus cancellability is a property of a function call instead of a function definition. First, let us 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. The class of cancellable functions is slightly larger: we allow cancellable calls to modify *resources* which anyway get removed by the WET. 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 WET.

Now, how do we specify and enforce the notion of cancellability? To enforce that a function does not trigger visible events or modify the physical state, CCR imposes a simple syntactic restriction (to be described in more detail shortly). A more interesting question is how to enforce that a cancellable function call terminate, as diverging function calls are not cancellable. For this, we add a new component $\mathtt{D}(w)$ to Cond (Fig. 7), where $\mathtt{D}(w)$, 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. In particular, a call with depth $\langle o \rangle$ is only allowed to call functions with depth *strictly smaller* than o. Those conditions together allows WET 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- and 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).[13] The last bracket is a newly added component saying `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 WET does not change even if the wrapper adds arbitrary cancellable calls to the abstractions since the cancellable calls are removed by WET. Thus,

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

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

`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. 8. Selected specifications for `Mem` module.

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 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 (other than pure computations) 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 pass the memory as an additional argument resp. return value each time a function gets invoked resp. returns. In CCR, we explore a rather different design: we handle memory as a *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 postconditions on functions. In particular, we can give a standard separation logic pre- and postconditions 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 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. 8, 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 §2.3.

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 WET. 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.

This section describes how to 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 §2.2. 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 WET will now do one additional task: it will materialize those abstract values so that, after the elimination of wrappers, they get *physically* passed around. In the above example, what will be left after the WET 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 WET, 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:

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

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 [Song et al. 2022].

```
(* 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)))
```

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

Fig. 9. An example of higher-order reasoning.

When $x_{\mathrm{a}}/r_{\mathrm{a}}$ in the pre/postcondition is omitted, it means they are equal to x/r, respectively. We conclude this section with a few remarks. First, 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. Second, while the mechanism is used here to abstract the values of implementation language into the values of specification (language), the mechanism is more general than that and we anticipate its wider applications. As a case study, we show how to do CompCertM-style compiler verification using this mechanism in [Song et al. 2022]. There, the target memory is passed as a physical value, and the source memory is passed as an abstract value.

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. 7) 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. 9. The function `repeat(f,n,m)` in I_{RP} recursively applies `*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 abstractions are simple and omitted in the figure: A_{RP} and A_{SC} directly returns an arbitrary integer – which is then enforced to satisfy their postcondition by ASSERT – and A_{AD} prints `(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. 9, 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 $S \supseteq_{\mathrm{S}} (S_{\mathtt{f}} \cup H_{\mathsf{RP}}(S_{\mathtt{f}}))$ (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 $S \supseteq_{\mathrm{S}} S_{\mathsf{SC}}$, we prove:

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

Also, we verify `AD`. For any $S_{\mathtt{f}} \supseteq_{\mathrm{S}} S_{\mathsf{SC}}$ (since `succ` is passed to `repeat`) and any $S \supseteq_{\mathrm{S}} (H_{\mathsf{RP}}(S_{\mathtt{f}}) \cup S_{\mathsf{AD}})$ (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 $S_{\mathsf{f}} = S_{\mathsf{SC}}$ and $S = H_{\mathsf{RP}}(S_{\mathsf{SC}}) \cup S_{\mathsf{SC}} \cup S_{\mathsf{AD}}$ and apply WET:

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

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

6 IMPLEMENTATION AND EVALUATION

6.1 Imp and its Verified Compiler

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

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

As with `Imp`, IMP is semantically interpreted into an itree (*i.e.,* EMS module here). The semantics is also standard except that the memory operations are interpreted as function calls to the `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* $(\!|I_i|\!) = \mathtt{Some}\ \mathit{Asm}_i$ *for* $i \in \{1, \ldots, n\}$,

$$\mathit{Beh}_{CC}(\mathit{Asm}_1 \bullet \cdots \bullet \mathit{Asm}_n) \subseteq \mathit{Beh}(I_{\mathtt{Mem}} \circ I_1 \circ \cdots \circ I_n)$$

Here $\bullet$ is the *syntactic linking* operator of CompCert, and $\mathrm{Beh}_{\mathrm{CC}}$ computes a set of CompCert traces for a given CompCert program, which are then implicitly casted into EMS traces. $I_{\mathtt{Mem}}$ 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. In these examples, we use the Iris Proof Mode [Krebbers et al. 2017] when proving logical entailments. Further explanations for most of these examples are in the appendix [Song et al. 2022].

As already mentioned (§2.2), vertical compositionality played a crucial role in simplifying the proof of the WET (Theorem 4.1). Specifically, the theorem is established by transitively composing six refinements, where major ones of which are (i) removing ASSUME and ASSERT 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 [Song et al. 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 $I \sqsubseteq_{\text{ctx}} S$) 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 [Turon et al. 2013; Frumin et al. 2021a; Gäher et al. 2022]. 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 $C[I] \sqsubseteq C[S]$.

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; Turon et al. 2013; Frumin et al. 2018, 2021b; Gäher et al. 2022]. 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.

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.

Simulation versus behavioral refinement. It is perfectly valid to take a simulation relation (Fig. 4)—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. 6), 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.*, CAL) where a uniform simulation is used across the compiler and program verifications.

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. Refinement Calculus [Back 1981; Back and Wright 2012] was a pioneer in this direction in that they employed `assume` and `assert` statements and dual non-determinism (which we borrow from them) to write specifications as programs, which in turn allows incremental verification. However, they only considered a simple language with global state (no module-private states), and also did not consider the interaction with separation logic. The most recent and relevant work to ours in this space is the work on Refinement-Based Game Semantics (RBGS) [Koenig and Shao 2020; Koenig 2020]. They extend Refinement Calculus into a setting similar to ours where there is a notion of layer (akin to module) and its local state. However, their focus was on unifying the notions of refinement, game semantics, and algebraic effects, and they also did 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 do not yet support all the features of higher-order concurrent separation logic [Jung et al. 2018], which have proven useful in reasoning about higher-order, imperative, and concurrent languages like Rust and OCaml.

Since CCR is a new framework that spans refinement-style verification, Hoare-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.

Finally, we have focused here on developing a "model" that unifies separation logic and contextual refinement, and the proofs we presented (in §2) work directly on the model level. In the future, we plan to develop higher-level proof techniques which can hide low-level details of the model.

ACKNOWLEDGMENTS

We thank Ralf Jung and Simon Spies for helpful feedback. Chung-Kil Hur is the corresponding author. This research was funded in part by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT2102-03, a Google PhD Fellowship (Sammler), and awards from Android Security's ASPIRE program and from Google Research.

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

R.J.R. Back. 1981. On correct refinement of programs. *J. Comput. System Sci.* 23, 1 (1981), 49–68. https://doi.org/10.1016/0022-0000(81)90005-2

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. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. https://doi.org/10.1145/1706299.1706323

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)*.

Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. 2013. The Power of Parameterization in Coinductive Proof. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 193–206. https://doi.org/10.1145/2429069.2429093

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), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. https://doi.org/10.1145/2676726.2676980

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

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

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

Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 2022. CCR: Technical documentation and Coq development. https://sf.snu.ac.kr/ccr/

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

Received 2022-07-07; accepted 2022-11-07