Conditional Contextual Refinement (CCR)

Youngju Song  
Seoul National University  
Korea  
youngju.song@sf.snu.ac.kr

Dongjae Lee  
Seoul National University  
Korea  
dongjae.lee@sf.snu.ac.kr

Minki Cho  
Seoul National University  
Korea  
minki.cho@sf.snu.ac.kr

Chung-Kil Hur  
Seoul National University  
Korea  
gil.hur@sf.snu.ac.kr

Abstract
Contextual refinement (CR) is one of the standard notions of specifying open programs. CR has two main advantages: (i) horizontal and vertical compositionality that allows us to decompose a large contextual refinement into many smaller ones enabling modular and incremental verification, and (ii) no restriction on programming features thereby allowing, e.g., mutually recursive, pointer-value passing, and higher-order functions. However, CR has a downside that it cannot impose conditions on the context since it quantifies over all contexts, which indeed plays a key role in support of full compositionality and programming features.

In this paper, we address the problem of finding a notion of refinement that satisfies all three requirements: support of full compositionality, full (sequential) programming features, and rich conditions on the context. As a solution, we propose a new theory of refinement, called CCR (Conditional Contextual Refinement), and develop a verification framework based on it, which allows us to modularly and incrementally verify a concrete module against an abstract module under separation-logic-style pre and post conditions about external modules. It is fully formalized in Coq and provides a proof mode that combines (i) simulation reasoning about preservation of side effects such as IO events and termination and (ii) propositional reasoning about pre and post conditions. Also, the verification results are combined with CompCert, so that we formally establish behavioral refinement from top-level abstract programs, all the way down to their assembly code.

1 Introduction
Contextual refinement (CR) is one of the standard notions of specifying open programs. For an open program $P$ and a more abstract program $A$ given as its specification, we say $P$ contextually refines $A$, denoted $P \preceq_{ctx} A$, if all possible observable behaviors of $P$ under an arbitrary closing context are included in those of $A$ under the same context. Here an observable behavior is a terminating or non-terminating trace of observable events such as input and output events. An important technical benefit is that CR only requires behaviors of closed programs even though CR relates open programs.

CR has two more advantages: support of (i) horizontal and vertical compositionality and (ii) full programming features. First, horizontal compositionality (HComp) allows us to compose modular verification results for different modules, say $M_1$ and $M_2$, as follows:

$$P_{M_1} \preceq_{ctx} A_{M_1} \land P_{M_2} \preceq_{ctx} A_{M_2} \implies P_{M_1 \circ M_2} \preceq_{ctx} A_{M_1 \circ A_{M_2}}$$

where $\circ$ is the linking operator between modules. On the other hand, vertical compositionality (VComp) allows us to compose incremental verification results inside the same module, say $M$, as follows:

$$P_M \preceq_{ctx} I_M \land I_M \preceq_{ctx} A_M \implies P_M \preceq_{ctx} A_M$$

Second, CR imposes no restriction on programming features, so that it allows cyclic structures such as mutual dependence between modules and higher-order functions, and also passing pointer values as argument and return values.

However, CR has a downside that it cannot impose conditions on the context although such conditions are often needed when decomposing a large contextual refinement into smaller ones. The reason is because the definition of CR requires behavioral refinement under all contexts, which indeed plays a crucial role in the general proof of full compositionality without any restriction on programming features.

In this paper, we address the problem of finding a notion of refinement that satisfies the three requirements: full compositionality, full programming features (in particular, cyclic structures), and rich conditions on the context.

A motivating example. Fig. 1 gives a motivating example, where the module $\text{MW}$ is intended for a simple middleware, and $P_{\text{MW}}^{1}$ and $P_{\text{MW}}^{2}$ are two (equivalent) implementations for it performing different optimizations written in green.

To see what $\text{MW}$ does, we look at the common black part of $P_{\text{MW}}^{1}$ and $P_{\text{MW}}^{2}$. The middleware starts with $\text{main}()$, which creates a partial map from $\text{int}_{64}$ to $\text{int}_{64}$ by $\text{Map.new}()$, initializes the application by $\text{App.init}()$, and keeps running it by $\text{App.run}()$. It also provides a map service to the app with $\text{put}(i, v)$, which maps $i$ to $v$ via $\text{Map.update}$ and prints a log, and $\text{get}(i)$, which returns the mapped value at the index $i$ obtained via $\text{Map.get}$ after printing a log.
Then, we see the green parts for optimizations. $P_{MW}^1$ optimizes all accesses to the indices between 0 and 100 by storing their data in the array `arr` of size 100, allocated by `Mem.alloc(100)`. $P_{MW}^2$ optimizes all accesses to the first index given to put by storing the index and its data in the module-local variables `idx` and `data`. It also uses the variable `first` to check whether put is invoked for the first time.

In order to share the common verification of the black code among $P_{MW}^1$ and $P_{MW}^2$, we give an intermediate abstraction $I_{MW}$, that only abstracts the green optimization code of both $P_{MW}^1$ and $P_{MW}^2$, leaving the black code unchanged. For this, it uses two mathematical functions `opt` and `cls` from `int64` to `int64`, where `opt` abstracts the optimized storage by mapping the optimized indices to their data and `cls` assigns to each index its class: 0 for unused indices, 1 for optimized ones, and 2 for the rest. Specifically, `put(i,v)` nondeterministically assigns a class to the index `i`, if it is unused, via `choose([1,2])`; then if `i` is an optimized index, updates `opt`; otherwise, updates `map`. Then `get(i)` assumes $i$ is not unused; then if it is an optimized index, reads from `opt`; otherwise, reads from `map`.

Also we give the final abstraction $A_{MW}$ that further abstracts the middleware. It uses one mathematical function `full` from `int64` to `int64`, which maps all used indices to their data as implemented in `put` and `get`.

Then our goal is to modularly (i.e., separately from other modules) and incrementally verify that $P_{MW}^1$ and $P_{MW}^2$ refine $A_{MW}$ possibly assuming specific conditions about external modules. Concretely, we wish to modularly verify (i) that $P_{MW}^1$ and $P_{MW}^2$ refine $I_{MW}$ by only reasoning about the green optimization code, and then (ii) that $I_{MW}$ refines $A_{MW}$.

One of the benefits of this incremental reasoning is that we can factor out and reuse the common verification that $I_{MW}$ refines $A_{MW}$. Indeed, if we write another optimization, say $P_{MW}^3$, we just need to verify $P_{MW}^3$ refines $I_{MW}$ by only reasoning about the optimization itself. Moreover, in general, such vertical decomposition can provide nice separation of concerns. In particular, well decomposed proofs may be more amenable to proof automation [31].

**Challenges and existing works.** The key question here is to identify the right notion of modular refinement (e.g., what does it mean for $I_{MW}$ to refine $A_{MW}$?). Note that we cannot simply use the (unconditional) contextual refinement because the refinements here are indeed conditional on the context: for example, if `Map.get` incorrectly always returns 0, $I_{MW}$ cannot refine $A_{MW}$ in any sensible way. Moreover, the intended mutual dependence between $MW$ and App makes it hard to define such conditions since they form (non-monotone) cyclic definitions. Specifically, since App uses $MW$, the condition that App behaves well depends

---

**Figure 1.** Two implementations $P_{MW}^1$, $P_{MW}^2$, an intermediate abstraction $I_{MW}$, and a full abstraction $A_{MW}$ for the module $MW$. 

---

1If the assumption fails, it triggers **undefined behavior (UB)** meaning that every possible behavior can happen nondeterministically.
on that $MW$ behaves well, which in turn depends on that $App$ behaves well since $MW$ also uses $App$.

Although the step-indexing technique [2] might be used to solve such cyclic definitions, it is well known that step-indexed relations are hardly transitive (i.e., hardly vertically compositional) [20]. Indeed, among many works using various forms of step-indexed relations [13, 14, 19, 28, 33, 39–41], none of them support transitivity of the relations. Note, however, that some of them [13, 14, 38–41] use step-indexed relations as a means to establish the unconditional CR, which allows VComp but no conditions on contexts.

Another line of work such as CAL (Certified Abstraction Layers) [15] and refinement calculus [4] avoids such cyclic definitions by disallowing those programming features that may introduce cyclicity such as mutual dependence and pointer-value passing between different modules. Indeed, CertiKOS [16], an operating system verified with CAL, clearly shows the merit and limitation of CAL. CertiKOS enjoys HComp and VComp of CAL by modularly and incrementally decomposing the whole refinement into 74 sub-refinements [17, 18]. However, it could not use any dynamic memory allocation since it would require pointer-value passing, and instead had to only use static variables of fixed sizes allocated at the booting time. Koenig and Shao [26, 27] presents a generalization of CAL that supports mutual recursion; however it still lacks support of rich conditions expressing various ownership as in modern separation logics [1, 22].

Yet another line of work such as parametric bisimulation [20], RGSim (Rely-Guarantee-based Simulation) [30] and RUSC (Refinement Under Self-related Contexts) [37] supports only a limited form of conditions on context, although fully supporting compositional and programming features. More precisely, they do not allow specific conditions per module but only global conditions that equally apply to every module.

To sum up, to our best knowledge, no existing work can verify our motivating example modularly and incrementally as outlined above.

**Our approach.** As a solution, we propose CCR (Conditional Contextual Refinement), which uses the (unconditional) contextual refinement as an underlying notion but overcomes its shortcoming by encoding module-specific conditions as executable code added to the abstract module. We will outline how CCR works for the motivating example.

First, although $P_{MW}^i \leq_{ctx, cond} I_{MW}$ does not hold, we can make $P_{MW}^i \leq_{ctx, cond} I_{MW}$ hold for:

\[
P_{MW}^i = I_{MW} + Asm(Mem is used well) + Asm(Mem behaves well) + Grt(MW is used well, simply) + Grt(MW behaves well, simply)
\]

Specifically, $I_{MW}$ extended with four kinds of executable code capturing the needed conditions. The two Grt’s encode guarantees about $MW$’s behavior saying that $MW$ uses Mem well and also behaves well, simply; and the two Asm’s encode assumptions about Mem’s behavior saying that Mem uses $MW$ well, simply, and also behaves well.

There are a few points to note. First, we do not need any conditions about Map and App since the black code using them is not abstracted in this refinement. Second, we only need a simple condition about $MW$ for interaction with Mem, which is why we put “simply.” The condition basically says that Mem does not use $MW$. A full condition, including the guarantee that $MW$. put and $MW$. get behave like a map assuming they are used well, will be needed to reason about interaction with App in the next abstraction.

The most important point here is that technically we assume nothing about external modules and thus do not need any cyclic definitions. Indeed, $P_{MW}^i$ refines $I_{MW}$ even when linked with badly-behaved Mem, which is why the contextual refinement holds. The intuition is that $Asm$ (or Grt) checks whether what has been observed by $MW$ is consistent with the assumed (or guaranteed) behaviors; then if the assumption (or guarantee) fails, it triggers UB (or NB). Here UB is called undefined behavior and interpreted as triggering every possible behavior nondeterministically, so that the refinement holds trivially if UB is triggered; that is, we just need to establish the refinement only when UB does not occur. On the other hand, NB is called no behavior and, roughly speaking, interpreted as empty behavior, so that the refinement cannot hold if NB is triggered; that is, in order to establish the refinement, we have to prove that NB does not occur.

Also we can prove $P_{MW}^i \leq_{ctx, App} I_{MW}$ for:

\[
A_{cond}^i = A_{MW}$ + Grt(Mem is used well) + Asm(Mem behaves well) + Asm(Map is used well) + Asm(App behaves well) + Grt(App behaves well) + Mem(MW is used well) + Grt(MW behaves well)
\]

By VComp of $\leq_{ctx}$, we have $P_{MW}^i \leq_{ctx, App} I_{MW}$ for $i \in \{1, 2\}$.

For other modules, we can prove the following possibly incrementally:

\[
P_{Mem} \leq_{ctx} A_{Mem}^cond, \quad P_{Map} \leq_{ctx} A_{Map}^cond, \quad P_{App} \leq_{ctx} A_{App}^cond
\]

where

\[
A_{Mem}^cond = A_{Mem}$ + Asm(Mem is used well) + Grt(Mem behaves well) + Asm(Map is used well) + Grt(Map behaves well)
\]

\[
A_{Map}^cond = A_{Map}$ + Asm(Mem is used well) + Grt(Mem behaves well) + Asm(Map is used well) + Grt(Map behaves well)
\]

\[
A_{App}^cond = A_{App}$ + Asm(MW is used well) + Asm(MW behaves well) + Asm(App is used well) + Grt(App behaves well)
\]

Note that we have the conditions as above since $P_{Mem}$ uses no external module, $P_{Map}$ uses only Mem, and $P_{App}$ uses only $MW$. Then, by HComp of $\leq_{ctx}$, we have: for $i \in \{1, 2\}$,

\[
P_{MW}^i \circ P_{Mem} \circ P_{Map} \circ P_{App} \leq_{ctx} A_{MW}^cond \circ A_{Mem}^cond \circ A_{Map}^cond \circ A_{App}^cond
\]

Finally, we have the following refinement thanks to our Assumption Cancellation Theorem (ACT) for a closed program,
which cancels out each Asm in the matching Grt and then freely eliminates all Grt’s by definition.

\[ A_{\text{Hof}}^{\text{cond}} \circ A_{\text{Hof}}^{\text{cond}} \circ A_{\text{App}}^{\text{cond}} \leq_{\text{ctx}} A_{\text{Hof}} \circ A_{\text{Hof}} \circ A_{\text{App}} \]

Again, by VComp of \( \leq_{\text{ctx}} \), we have: for \( i \in \{1, 2\} \),

\[ p_i^i \circ p_{\text{Hof}} \circ p_{\text{Hof}} \circ p_{\text{App}} \leq_{\text{ctx}} A_{\text{Hof}} \circ A_{\text{Hof}} \circ A_{\text{App}} \]

Note that conditions in intermediate abstractions such as Grt(MW behaves well, simply) are never used by the cancellation theorem, so that they can be freely chosen regardless of the actual conditions used by external modules.

To summarize, we addressed the challenging problem by developing a novel mechanism (i.e., Asm and Grt) to operationally and module-locally encode rich conditions on context modules. Moreover, Asm and Grt are auto-generated from propositions that can express various ownership via PCMs (Partial Commutative Monoids) [8, 34] as in the state-of-the-art separation logics (SLs) such as Iris [22] and VST [1]. Unlike those SLs, we encode PCM-based ownership without using step-indexing even for mutually dependent modules, which is essential to support VComp. Also, we believe that the key ideas of CCR could be applied to the verification of CertiKOS to lift its current restrictions (i.e., the absence of dynamic allocation and mutual dependence).

**Contributions.** All our results are fully formalized in Coq [3] and summarized as follows.

1. We develop the first theory, CCR, that provides a notion of refinement supporting full compositionality, full (sequential) programming features and rich conditions.

2. We develop EMS (Executable Module Semantics) as a general underlying module semantics for CCR, which is uni-typed with the most general type consisting of all mathematical values, and allows each module to be equipped with an arbitrary small-step operational semantics (expressed in terms of interaction trees [43]) with a given set of events such as primitive events and calls to external functions. EMS also supports two notions of nondeterminism, called demonic and angelic one in the literature [4, 7, 27, 42], which we crucially use to encode rich conditions on context modules.

EMS has two advantages. First, many languages with different type systems can be embedded into EMS since, in particular, typing assumptions or guarantees about argument and return values can be easily expressed. As examples, we develop the following languages and embed them into EMS.

- **IMP:** A C-like language with integer and (function and memory) pointer values, which is used to write implementation code and compiled down to assembly via our verified compiler for IMP together with CompCert.

- **SPC:** A specification language in which one can specify various conditions such as Asm’s and Grt’s above and also write abstract yet executable code.

Second, since modules in EMS are written as interaction trees, they are executable via Coq’s extraction mechanism into OCaml, so that we can test them.

3. We verify various examples (including the motivating example) written in IMP, which demonstrates (i) modular and incremental verification via intermediate abstractions (i.e., full compositionality); (ii) cyclic and higher-order reasoning about recursion and function pointers (i.e., full features); and (iii) reasoning about PCM-based ownership (i.e., rich conditions).

For this verification, we use the CCR proof mode in Coq supporting both (i) simulation reasoning about preservation of side effects such as IO events and termination and (ii) propositional reasoning about conditions on contexts, for which we employ the IPM (Iris Proof Mode) package [28] (i.e., by instantiating it with our CCR theory) that streamlines reasoning about PCMs. Note that the details of the simulation technique and CCR proof mode will be published elsewhere.

4. We develop a verified compiler for IMP targeting Csharp-minor of CompCert [29], which is in turn compiled by the verified compiler CompCert to generate assembly code. As a result, we formally establish behavioral refinement from the top-level abstractions of the above examples, all the way down to their compiled assembly code.

## 2 Key ideas

### 2.1 Technical challenges and our solution

To understand the challenges with defining Asm and Grt, we give an example, where we want to abstract \( P_{\text{Once}} \) into \( A_{\text{Once}} \).

\[
P_{\text{Once}} := \text{[Module Once]} \\
\text{local} \quad \text{done} := \text{false} \\
\text{def} \quad \text{do()} \equiv \text{true} \\
\text{if} \quad \text{(done)} \quad \text{print("err")} \\
\text{else} \quad \text{done} := \text{true} \\
A_{\text{Once}} := \text{[Module Once]} \\
\text{def} \quad \text{do()} \equiv \text{skip} \\
\text{if} \quad \text{(done)} \quad \text{print("err")} \\
\text{else} \quad \text{done} := \text{true}
\]

The question here is how to encode and add the condition Asm(do() hasn’t been called) to do() in \( A_{\text{Once}} \), in a way that if a client locally proves Grt(do() hasn’t been called) before calling do(), then they are canceled out when linked.

A naive encoding \( A^{\text{try}}_{\text{Once}} \) might be as follows.

\[
A^{\text{try}}_{\text{Once}} := \text{[Module Once]} \\
\text{local} \quad \text{done} := \text{false} \\
\text{def} \quad \text{do()} \equiv \text{false} \\
\text{assume}(\text{done} = \text{false}) \\
\text{done} := \text{true} \\
\]

\[
P_{\text{Test}} := A_{\text{Test}} := \text{[Module Test]} \\
\text{def} \quad \text{main()} \equiv \text{Once.do()} \\
A_{\text{Test}}^{\text{cond}} := \text{[Module Test]} \\
\text{def} \quad \text{main()} \equiv \text{false} \\
\text{assume}(\text{done} = \text{false}) \\
\text{Grt(...)}; \text{Once.do()}
\]

We can easily see that \( P_{\text{Once}} \leq_{\text{ctx}} A^{\text{try}}_{\text{Once}} \) holds since, when called twice, do() will trigger UB by assume(done = false) rendering all possible behaviors.

However, it is unclear how to state the guarantee condition Grt(...) in \( A^{\text{cond}}_{\text{Test}} \), above, so that it can automatically cancel out assume(done = false) in \( A^{\text{try}}_{\text{Once}} \) when linked. The problem is that the module Test, cannot access the local variable done of Once and even worse, since the context
\[ A^\text{cond}_{\text{Once}} := \begin{cases} \text{Module Once} \\
\end{cases} \]
\[ \text{local mres} := \varepsilon \]
\[ \text{def do()} \equiv \]
\[ \begin{array}{ll}
\text{var frm} := & \text{Asm}(\lambda \sigma. \sigma \geq \text{Do}, \varepsilon) \\
\text{skip} & \text{Grt}(\lambda \sigma. \text{True}, \text{frm})
\end{array} \]
\[ A^\text{cond}_{\text{Test}i} := \begin{cases} \text{Module Test}i \\
\end{cases} \]
\[ \text{local mres} := \varepsilon \]
\[ \text{def main()} \equiv \]
\[ \begin{array}{ll}
\text{var frm} := & \text{Asm}(\lambda \sigma. \sigma \geq \text{Do}, \varepsilon) \\
\text{repeat} \ i \ {\}} & \begin{array}{l}
\text{frm} := \text{Grt}(\lambda \sigma. \sigma \geq \text{Do}, \text{frm}) \\
\text{Once.do()} \\
\text{frm} := \text{Asm}(\lambda \sigma. \text{True}, \text{frm}) \\
\text{Grt}(\lambda \sigma. \text{True}, \text{frm})
\end{array}
\end{array} \]
\[ \text{Asm}(\text{Cond}, \text{lres}) \equiv \{ \\
\} \]
\[ \text{Grt}(\text{Cond}, \text{eres}) \equiv \{ \\
\} \]
\[ \text{var (eres, } \sigma) := \text{take}(\Sigma \times \Sigma) \]
\[ \text{assume(Cond } \sigma) \]
\[ \text{assume(} \forall V(\text{mres} + \text{lres} + \text{eres} + \sigma) \]
\[ \text{eres} \} \]
\[ \text{lres} \} \]

**Figure 2.** Conditional abstractions \( A^\text{cond}_{\text{Once}} \cdot A^\text{cond}_{\text{Test}i} \cdot A^\text{cond}_{\text{Test}j} \)

The module Once is arbitrary in \( P_{\text{Test}i} \leq_{ctx} A^\text{cond}_{\text{Test}i} \), it may not have such a local variable at all.

To understand the requirement for the encoding more clearly, consider the following variation.

\[ P_{\text{Test}2} := A^\text{cond}_{\text{Test}2} \]
\[ \text{def main()} \equiv \]
\[ \begin{array}{ll}
\text{Once.do()} \\
\text{Once.do()}
\end{array} \]

It is clear that something should go wrong here because Once.do() is called twice. Here what we desire is that since actual proofs should be done module-localy in \( P_{\text{Once}} \leq_{ctx} A^\text{cond}_{\text{Once}} \) and \( P_{\text{Test}i} \leq_{ctx} A^\text{cond}_{\text{Test}i} \), the cancellation of syntactically matched Asm and Grt should hold unconditionally without requiring reasoning about the conditions; that is, the following should hold:

\[ A^\text{try}_{\text{Once}} \circ A^\text{cond}_{\text{Test}i} \leq_{ctx} A^\text{cond}_{\text{Once}} \circ A^\text{try}_{\text{Test}i} \]

Instead, the local reasoning of \( P_{\text{Test}i} \leq_{ctx} A^\text{cond}_{\text{Test}i} \) should go wrong since \( A^\text{cond}_{\text{Test}2} \cdot \text{main()} \) is blamable due to its two calls of Once.do(). From this, it follows that the second Grt(...) in \( A^\text{cond}_{\text{Test}2} \) should effectively prevent the second call to Once.do(). The reason is because invoking Once.do() twice triggers UB in \( A^\text{try}_{\text{Once}} \circ A^\text{cond}_{\text{Test}2} \) while \( A^\text{cond}_{\text{Once}} \circ A^\text{cond}_{\text{Test}2} \) does not.

To sum up, the challenge here is that the Asm in Once.do() and the Grt in Test2.main() are completely independent and local computations since there is no argument passing or secrete channel; however, they should affect each other and prevent undesired computation in the right place.

**Our solution.** Fig. 2 shows how we encode the conditions for \( A^\text{Once} \) and \( A^\text{Test}i \) for \( i \in \{1, 2\} \). Before we proceed, we remark an interesting point. In fact, the two conditional abstractions \( A^\text{try}_{\text{Once}} \) and \( A^\text{cond}_{\text{Once}} \) are contextually equivalent:

\[ A^\text{try}_{\text{Once}} \leq_{ctx} A^\text{cond}_{\text{Once}} \text{ and } A^\text{cond}_{\text{Once}} \leq_{ctx} A^\text{try}_{\text{Once}} \]

Thus the following contextual refinements are equivalent.

\[ A^\text{cond}_{\text{Once}} \circ A^\text{cond}_{\text{Test}i} \leq_{ctx} A^\text{cond}_{\text{Once}} \circ A^\text{cond}_{\text{Test}i} \]

\[ \Leftrightarrow A^\text{try}_{\text{Once}} \circ A^\text{cond}_{\text{Test}i} \leq_{ctx} A^\text{cond}_{\text{Once}} \circ A^\text{cond}_{\text{Test}i} \]

However, the former is much easier to prove than the latter. Specifically, the former proof just requires to check that the Asm’s and Grt’s are syntactically matched without requiring any reasoning about the conditions specified in them, while the latter proof essentially requires non-trivial reasoning about the conditions. It is important to note that such difference is made due to the specific ways they are computed, not due to their observable behaviors.

Inspired by the approach of modern SLs [1, 22], we also encode various ownership representing certain capability and knowledge using partial commutative monoids (PCMs), but operationally and module-locally (i.e., independently from external modules). Note that a PCM \( \Sigma \) is a set equipped with a commutative and associative binary operator + on \( \Sigma \), called addition, an identity element \( \varepsilon \), and a validity predicate \( V \) on \( \Sigma \) satisfying (i) \( V(\varepsilon) \) and (ii) \( \forall a, b. V(a + b) \Rightarrow V(a) \). Since invalid elements are considered undefined, + can be seen as a partial operator. Elements of \( \Sigma \) are called resources and we say \( a \geq b \) for \( a, b \in \Sigma \) if there exists \( \sigma \in \Sigma \), \( a = b + c \).

In \( A^\text{cond}_{\text{Once}} \), we add conditions basically saying that do() should be invoked at most once. For this, we can use any PCM \( \Sigma \) involving a resource, Do, such that Do + Do is undefined (i.e., whose result is invalid), which represents inconsistency. Then, Do represents the capability to invoke do() and we can see that there is at most one Do since Do + Do is undefined.

With this intuition, we look at do() in \( A^\text{cond}_{\text{Once}} \). The Asm at the beginning encodes the assumption that Do is given to do() by the caller, which forms Asm(do() hasn’t been called); the Grt at the end says that we guarantee nothing. The special module-local variable mres is only used by Asm and Grt and initialized with a specific resource for each module, which is \( \varepsilon \) for every module in this example. The intuition here is that invoking do() requires the resource Do, which is consumed (i.e., not returned) so that do() cannot be invoked anymore. Similarly, main() requires Do initially and guarantees to give Do before each invocation of Once.do().

Now we see the computational interpretation of Asm and Grt, whose definitions are given at the bottom of Fig. 2. Note that each function of a module (decorated with conditions) maintains four kinds of resources at every interaction point with a caller/callee:

- a module resource stored in the module-local variable mres that is owned by the module (i.e., shared among all functions in the module),
- a local resource, lres, that is locally owned by the current invocation of the function,
- an external resource, eres, that is conceptually the summation of all the other local and module resources in the whole system at the moment,
• a call resource, σ, that is conceptually passed from the caller/callee or will be passed to the caller/callee.

Then, \( \text{Asm}([\text{Cond}, \, \text{1res}]) \) for \( \text{Cond} \) a predicate on \( \Sigma \) and \( \text{1res} \) the current local resource, which is \( \varepsilon \) at the beginning of each function invocation, computes as follows. It magically takes an external resource \( \text{eres} \) and a call resource \( \sigma \) via \( \text{take} \) (which are conceptually passed from somewhere but technically out-of-thin-air, which we will explain soon); \( \text{assumes} ([\text{Cond} \, \sigma]) \) holds (i.e., if unsuccessful, triggers UB rendering all possible behaviors); \( \text{assumes} \) all four resources are consistent (i.e., their summation is valid); and returns \( \text{eres} \). As in \( \text{do}() \) and \( \text{main}(\cdot) \), the returned \( \text{eres} \) is stored at the variable \( \text{frm} \) and passed to the next \( \text{Grt} \).

Then, \( \text{Grt}([\text{Cond}, \, \text{eres}]) \) for \( \text{eres} \) the current external resource computes as follows. It nondeterministically chooses a module resource and updates the module-local variable \( \text{mres} \) with it and further chooses a local resource \( \text{lres} \) and a call resource \( \sigma \) via \( \text{choose} \) (which are conceptually passed to somewhere but technically nowhere, which we will explain soon); \( \text{guarantees} ([\text{Cond} \, \sigma]) \) holds (i.e., if unsuccessful, triggers NB, called no behavior, basically rendering empty behavior); \( \text{guarantees} \) all four resources are consistent; and returns \( \text{lres} \). The returned \( \text{lres} \) is stored at the variable \( \text{frm} \) and passed to the next \( \text{Asm} \).

Now we see how \( \text{take} \) and \( \text{choose} \) can make an illusion of receiving and sending any ghost information, including a resource, from and to a caller/callee. Indeed, a function cannot physically receive or send any ghost information in the setting of contextual refinement because context modules are completely arbitrary so that they may well be just physical modules such as those written in IMP. However, by using the standard (a.k.a. demonic) nondeterministic choice function \( \text{choose} \) and its dual (a.k.a. angelic [4, 7, 27, 42]) one \( \text{take} \), we can logically make such an illusion. Basically, we define the operational behaviors of \( \text{choose}(X) \) and \( \text{take}(X) \) for any set \( X \) as follows:

\[
\text{Beh}(\text{var} \, x := \text{choose}(X); \, K[x]) \overset{\text{def}}{=} \bigcup_{x \in X} \text{Beh}(K[x])
\]

\[
\text{Beh}(\text{var} \, x := \text{take}(X); \, K[x]) \overset{\text{def}}{=} \bigcap_{x \in X} \text{Beh}(K[x])
\]

It is important to note that even though defining angelic nondeterminism for \( \text{open} \) programs is non-trivial [27], we can avoid such a problem and give a simple operational semantics as above since we only need to define it for \( \text{closed} \) programs, thanks to the use of contextual refinement.

### The example revisited.

Then we discuss how such an illusion can actually make refinement between \( \text{P}_{\text{Once}} \) and \( A_{\text{Once}}^{\text{cond}} \) hold for any context including badly behaved ones (i.e., \( \text{P}_{\text{Once}} \preceq_{\text{ctx}} A_{\text{Once}}^{\text{cond}} \)). When \( \text{do}() \) is invoked for the first time, we need to establish the refinement for any taken \( \text{eres} \) and \( \sigma \) that satisfy both \( \text{assume} \) in \( \text{Asm}(\lambda \sigma. \, \sigma \geq \text{Do}, \, \varepsilon) \) by definition of \( \text{take} \). Note that an unsuccessful \( \text{assume} \) triggers UB rendering all possible behaviors and thus the refinement trivially holds. Now, for any such taken \( \text{eres} \) and \( \sigma \), both \( \text{P}_{\text{Once}} \) and \( A_{\text{Once}}^{\text{cond}} \) return, where we successfully \( \text{choose} \) to update \( \text{mres} \) with \( \text{Do} \) by executing \( \text{Grt}(\lambda \sigma. \, \text{True}, \, \text{frm}) \) since the \( \text{frm} \) given by \( \text{Asm}(\lambda \sigma. \, \sigma \geq \text{Do}, \, \varepsilon) \) cannot include \( \text{Do} \). Since then, whenever \( \text{do}() \) is invoked again (badly), the refinement trivially holds since \( \text{mres} \) contains \( \text{Do} \) and thus \( \text{Asm}(\lambda \sigma. \, \sigma \geq \text{Do}, \, \varepsilon) \) must trigger UB in \( A_{\text{Once}}^{\text{cond}} \) since the condition cannot be met. Note that this informal argument can be made formal by using a simple simulation relation between \( \text{P}_{\text{Once}} \) and \( A_{\text{Once}}^{\text{cond}} \) with the following relational invariant:

\[
(\text{done} = \text{false} \land \text{mres} = \varepsilon) \lor (\text{done} = \text{true} \land \text{mres} = \text{Do})
\]

Similarly, one can easily see \( \text{P}_{\text{Test}}, \preceq_{\text{ctx}} A_{\text{Test}}^{\text{cond}}, \) also follows.

### Assumption Cancellation.

Finally, we discuss how the assumptions are canceled out by the matching guarantees when the two conditional abstractions are linked:

\[
A_{\text{Once}}^{\text{cond}} \circ A_{\text{Test}}^{\text{cond}}, \preceq_{\text{ctx}} A_{\text{Once}} \circ A_{\text{Test}}^{\text{cond}}.
\]

The assumption cancellation theorem (ACT) is proven in general for any \( \text{closed} \) program as follows.

To discharge the initial \( \text{Asm} \) of \( \text{main}(\cdot) \) (i.e., replacing it with \( \text{skip} \)), we just need to take an initial resource \( \sigma \) to \( \text{main}(\cdot) \), here \( \text{Do} \), that (i) is consistent with all the initial module resources, here \( \varepsilon \) for both \( A_{\text{Once}}^{\text{cond}} \) and \( A_{\text{Test}}^{\text{cond}} \), and (ii) satisfies the precondition of \( \text{main}(\cdot) \), here \( \sigma \geq \text{Do} \). For \( \text{eres} \), we take the summation of the initial resources of all external modules.

Then, every subsequent \( \text{Asm} \) is automatically discharged by carefully taking its resources depending on the resources chosen by the matching (i.e., immediately preceding) \( \text{Grt} \). Specifically, we take (i) \( \varepsilon \) to be the same \( \sigma \) chosen by the \( \text{Grt} \) and (ii) \( \text{eres} \) to be the summation of all the local and module resources in the system except for its own \( \text{lres} \) and \( \text{mres} \). Then the two \( \text{assume} \)’s in the \( \text{Asm} \) are discharged by the two \( \text{guarantee} \)’s in the \( \text{Grt} \) because \( \text{mres} + \text{lres} + \text{eres} \) in \( \text{Asm} \) is equal to that in \( \text{Grt} \). This is because both summations coincide with the summation of all the local and module resources in the system by construction.
We conclude with two remarks. First, in our Coq formalization, we slightly generalize the CCR framework to support the ACT theorem for open programs, which cancels out every assumption by the matching guarantee even in the presence of unverified (logically orthogonal) contexts. Second, even after applying the ACT theorem (in particular to a selected set of verified modules using the generalized CCR), we can further abstract the resulting abstraction (possibly together with other unverified modules) using the (generalized) CCR. The technical report [3] presents the details of the generalized CCR framework and the examples for the above two cases in §3.1 and §3.3, respectively.

### 2.2 Incremental verification

To present the key ideas behind incremental verification, we give a simple implementation $P_{App}$, shown in Fig. 3, for the module $App$ of the motivating example. We incrementally verify it against $A_{App}$ via $I_{App}$ just for presentation purposes.

In $P_{App}$, the init() function first checks whether $App$ is already initialized or not using the module-local variable initialized; if so, prints an error; otherwise, assigns true to the variable and put 42 at index 0 using MW.put. The run() function also checks whether $App$ is initialized; if not, prints an error; otherwise, fetches the value at index 0 using MW.get and prints it. In $r_{App}^{cond}$, we abstract away the green error checking code of $P_{App}$, and add the following conditions:

$$\text{Asm}(\text{App is used well, simply}) + \text{Grt}(\text{App behaves well, simply})$$

Note that we do not need any condition on $MW$ since the black code using $MW$ is not abstracted. In $A_{App}^{cond}$, we further optimize the function run by replacing the fetched value $v$ with 42 and add the following conditions:

$$\text{Grt}(\text{$MW$ is used well}) + \text{Asm}(\text{$MW$ behaves well}) + \text{Asm}(\text{App is used well}) + \text{Grt}(\text{App behaves well})$$

**First abstraction.** In $r_{App}^{cond}$, we add conditions basically saying that init() should be invoked once and then run() can be invoked many times. For this, we can use any PCM $\Sigma$ involving three resources, Init, Run and Both, with the following law:

$$\text{Init} + \text{Run} = \text{Both}$$

All the other additions among them are undefined (i.e., whose results are invalid), which represents inconsistency. Then, init and Run represent the capability to invoke init() and run(), respectively. From the laws of $\Sigma$, we can see that there are at most one Init and one Run since Init + Init and Run + Run are undefined.

With this intuition, we look at init() and run() in $r_{App}^{cond}$. The Asm’s at the beginning of them encode the assumptions that Init / Run is given to init() / run() respectively by the caller, which form $\text{Asm}(\text{App is used well, simply})$. On the other hand, the Grt’s at the end of them encode the guarantees that Run is returned to the caller of init() / run(), which form $\text{Grt}(\text{App behaves well, simply})$. The Grt and Asm around MW.put and MW.get explicitly say that we guarantee and assume nothing about them. The special module-local variable $mres$ is only used by Asm and Grt and initialized with Run. The intuition behind these conditions is as follows: (i) only init() can be invoked initially by an external module since Run is kept in $mres$ of App; (ii) once init() is invoked with Init, it consumes Init and returns Run instead, so that init() cannot be invoked anymore, but instead (iii) run() can be invoked with Run, which returns Run back to the caller, so that run() can be invoked again.

Then, similarly as before, one can informally check that $P_{App} \subseteq_{ctx} A_{App}^{cond}$ holds, which can be formalized by a simulation relation with the following invariant.

$$(\text{initialized} = \text{false} \land mres = \text{Run}) \lor (\text{initialized} = \text{true} \land mres = \text{Init})$$

**Further abstraction.** In $A_{App}^{cond}$, we add further conditions about $MW$ to reason about the propagation of 42 from init() to run(). For this, we use new resources $\text{MWhas}(f)$ for $f \in \text{int}_{42}$ to $\text{int}_{42}$, where $\text{int}_{42}$ is a shorthand for option $\text{int}_{42}$. Intuitively, $\text{MWhas}(f)$ captures (i) the knowledge that the module $MW$ currently contains the partial map $f$ and (ii) the capability to invoke $MW$.put and $MW$.get. Technically, these resources are defined using the standard PCM combinators $\text{Auth}$ and $\text{Ex}$ [21] (see §4 for the definition of $\text{MWhas}$).

Then init() assumes Init and $\text{MWhas}(fi)$ for some $fi$ at the beginning, and guarantees Run and $\text{MWhas}(fi|0 ← 42)$ at the end. Here it is important to note that the variable $fi$ connecting the assumption and guarantee, also called auxiliary variable in the literature [25, 35], is taken via take making an illusion of receiving the information from the caller. Similarly, run() assumes $\text{MWhas}(fi)$ with $fi(0) =$ Some 42 for some $fi$ together with Run at the beginning, and guarantees the same at the end. These conditions form $\text{Asm}(\text{App is used well})$ and $\text{Grt}(\text{App behaves well})$.

On the other hand, in init(), we guarantee $\text{MWhas}(fe)$ for some $fe$ before the call $\text{MW}.\text{put}(0, 42)$ and assumes $\text{MWhas}(fe[0 ← 42])$ after the call. Similarly, in run(), we guarantee $\text{MWhas}(fe)$ with $fe(0) =$ Some ve for some $fe$ and ve before the call $v := \text{MW}.\text{get}(0)$ and assumes $\text{MWhas}(fe)$ with $v = \text{ve}$ after the call. These form $\text{Grt}(\text{MW is used well})$ and $\text{Asm}(\text{MW behaves well})$. Note that this time the auxiliary variables $fe$ and ve are chosen via choose making an illusion of sending the information to the callee.

We conclude with a few remarks on the verification of $r_{App}^{cond} \subseteq_{ctx} A_{App}^{cond}$. First, all proof obligations about Init and Run are automatically discharged by the CCR proof mode using the fact that conditions about them are preserved between $r_{App}^{cond}$ and $A_{App}^{cond}$. Second, by choosing $fe$ to be the same as $fi$ (and $ve$ to be 42) in init() and run(), all the guarantees about $\text{MWhas}$ are trivially discharged by the immediately preceding assumptions. Third, from the assumption after the call $v := \text{MW}.\text{get}(0)$ in $A_{App}^{cond}$, it follows that $v = 42$, which,
We present the underlying semantics EMS, the specification

$$X \in \text{coind} \text{ObsEvent}$$

def run() :=

def init() :=

if \{initialized\} {
    print("error: init")
} else {
    initialized := true
    MW.put(0, 42)
}

def run() :=
if \{!initialized\} {
    print("error: run")
} else {
    var v := MW.get(0)
    print("val:"+str(v))
}

Figure 3. An implementation $P_{\text{App}}$, an intermediate conditional abstraction $r^{\text{cond}}_{\text{App}}$, a full conditional abstraction $A^{\text{cond}}_{\text{App}}$ for $\text{App}$

$$X|_{\text{cond}} \overset{\text{def}}{=} \text{if cond holds, then } X \text{ else } \emptyset$$

fundef($E$) $\overset{\text{def}}{=} \text{Any} \rightarrow \text{itree E Any}$

$E_{\text{EMS}}(X) \overset{\text{def}}{=} \begin{cases} \text{Obs fn ars} \mid \text{fn in string, ars in Any} | X = \text{Any} \lor \text{Call fn ars} \mid \text{fn in string, ars in Any} | X = \text{Any} \lor \\
\text{Get} | X = \text{Any} \lor \text{Put a | a in Any} | X = 1 \lor \\
\text{Choose} \lor \text{Take} \end{cases}$

$\text{EMS} \overset{\text{def}}{=} \{(\text{init, funs}) \in \text{Any} \times (\text{string fn) fundef(E_{\text{EMS}}))}

$\text{Mod} \overset{\text{def}}{=} \text{LD} \times (\text{LD} \rightarrow \text{EMS}) \quad \text{Mods} \overset{\text{def}}{=} \text{list Mod}$

$\circ \in \text{Mods} \rightarrow \text{Mods} \rightarrow \text{Mod} \overset{\text{def}}{=} \text{append}$

$M \subseteq M' \overset{\text{def}}{=} \forall N \in \text{Mods}. \text{Beh}(M \circ N) \subseteq \text{Beh}(M' \circ N)$

$\text{ObsEvent} \overset{\text{def}}{=} \{(\text{Obs fn ars}, r) \mid \text{fn in string, ars in Any} \}$

$\text{Trace}^{\text{cond}} \overset{\text{def}}{=} \{e \in \text{Trace} \mid \text{Term v | v in Any} \lor \text{Diverge} \lor \text{Error} \lor \text{Partial} \}$

$\text{Beh} \in \text{ Mods} \rightarrow \text{P(Trace)} \overset{\text{def}}{=} \ldots$

Figure 4. Definitions of Executable Module Semantics (EMS)

together with the fact that the return value $v$ from the same
call should be the same in $r^{\text{cond}}_{\text{App}}$ and $A^{\text{cond}}_{\text{App}}$, proves that the
same string is passed to $\text{print}$ in $\text{run()}$ of $r^{\text{cond}}_{\text{App}}$ and $A^{\text{cond}}_{\text{App}}$.

3 Formal definitions and key theorems

We present the underlying semantics EMS, the specification
language SPC with its shallow embedding into EMS, key
theorems of CCR and our verified compiler for IMP.

3.1 EMS (Executable Module Semantics)

Interaction trees. First of all, our Coq formalization largely
relies on interaction trees [43]. Intuitively, an itree in itree $E T$
for an event type $E : \text{Set} \rightarrow \text{Set}$ (consisting of a set of events

$E(X)$ whose return type is $X$) and a return type $T$ for the
itree can be understood as an open small-step operational
semantics that can (i) take a silent step, (ii) terminate with
a return value of type $T$, or (iii) trigger an event in $E(X)$
for some $X$ and, as a continuation, give an itree for each
possible return value in $X$. We enjoy two benefits of interaction
trees: (i) they are extracted to executable programs in OCaml,
and (ii) they provide useful combinators, which
made our various constructions straightforward.

We mainly use the interpretation combinator with the type:

$$\text{itree } E T \rightarrow (\forall X. E(X) \rightarrow ST \rightarrow \text{itree } E' (X \times ST)) \rightarrow ST \rightarrow \text{itree } E' (X \times ST)$$

It takes an itree $t$ in 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.
This combinator is useful when adding the code encoding
conditions (i.e., those written in red and blue in the
previous examples) by interpreting each call event as the
same call with conditions around it. We use the notation
$t[\epsilon_1 \mapsto \lambda.s.t_1, \ldots, \epsilon_n \mapsto \lambda.s.t_n](s_0)$ to denote the resulting itree
when the combinator is applied to an itree $t$, with an initial
local state $s_0$, by interpreting each event $\epsilon_i$ to an itree $t_i$ for
a given local state $s$. We omit those events that are interpreted
identically, and the state component when it is the unit type.
Since $\text{itree } E$ forms a monad for any $E$, we henceforth use
the monad notations: $x ::= t_1; t_2$ for bind and $\text{ret } t$ for return.

EMS. Fig. 4 shows the formal definition of EMS, where we use $X|_{\text{cond}}$ to denote a conditionally non-empty set. First, fundef($E$) is the semantic domain for a function, which takes
a value in Any as an argument and gives an itree w.r.t. the
event type $E$ and the return type Any, where Any can be
understood as the set of all mathematical values. $E_{\text{EMS}}$ is
The event type for EMS consisting of (i) Obs for triggering observable events such as system calls, (ii) Call for making a call to (internal or external) functions, (iii) Get and Put for accessing the module local state of type Any, and (iv) Choose and Take for nondeterministically choosing and taking a value from any given set X. EMS is the semantic domain for a module (after loading), 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 EEMS.

Mod gives a notion of module code (i.e., before loading) for a global loading data type LD, which happens to be required to form a PCM to combine loading data from all modules and express consistency between them. A module code consists of its own loading data in LD and a loading function in LD → EMS that, given the global loading data gathered from all the modules, returns its module semantics. A modules code in Mods is simply a list of module codes and linking o between them is the list append. Note that we require function names to be globally unique when loading modules, which is not too strong since function names can include their module name as a prefix. Then we define contextual refinement between two modules codes M and M’ as behavioral refinement under an arbitrary context modules code N.

### Observable behavior.

To give the notion of behavior, we first define the set of traces, Trace, coinductively. A trace is a finite or infinite sequence of ObsEvent (i.e., pairs of an observable event and its return value) that can possibly end with one of the four cases: (i) normal termination with an Any value, (ii) silent divergence without producing any events, (iii) erroneous termination, or (iv) partial termination. The notion of partial termination is interesting, which is used to define NB (to be shown below). It can be intuitively understood as stopping the execution at the user’s will such as pressing Ctrl+C, which is dual to erroneous termination (i.e., termination due to the program’s fault).

The predicate Beh(M) defines all possible traces of the modules code M in a standard way except for the following. First, the partial termination, Partial, can occur nondeterministically at any point during execution (capturing that the user can stop the program at any time). Second, the behaviors of choose and take are defined as follows:

\[
\overline{\text{Beh}}(x \leftarrow \text{choose}(X); K[x]) \equiv_{\text{cond}} \text{Partial} \cup \bigcup_{x \in X} \overline{\text{Beh}}(K[x])
\]

\[
\overline{\text{Beh}}(x \leftarrow \text{take}(X); K[x]) \equiv_{\text{cond}} \text{Partial} \cup \bigcup_{x \in X} \overline{\text{Beh}}(K[x])
\]

where \( \overline{\text{Beh}} \) is coinductively defined for the large (closed) itsre of all functions of all modules in M (see [3, Fig. 9-10] for definition). Third, assume and guarantee are defined as follows.
\[
\text{assume}(P) \overset{\text{def}}{=} \text{if } (P) \text{ skip else take}(\emptyset)
\]
\[
\text{guarantee}(P) \overset{\text{def}}{=} \text{if } (P) \text{ skip else choose}(\emptyset)
\]

Note that \text{assume}(\text{False}) exhibits all observable behaviors (i.e., Trace), understood as UB, and \text{guarantee}(\text{False}) exhibits only the partial termination (i.e., \{Partial\}), understood as NB. Thanks to the partial termination, we can avoid the completely empty behavior, which may cause a trouble since it can eliminate previously triggered events.

3.2 SPC and its embedding into EMS

We define a language, SPC, where one can specify conditions and write executable code for modules, which are then shallowly embedded into EMS (i.e., auto-generating the code that encodes the conditions). Fig. 5 shows the formal definition of SPC and its embedding.

**SPC.** In SPC, for each function, we can specify a pair of pre and post conditions \( s \in \text{Cond}_\Sigma \), which is parameterized by a global PCM \( \Sigma \). Concretely, a condition \( s \in \text{Cond}_\Sigma \) consists of four components \( (w, D, P, Q) \) and a collection of conditions \( S \in \text{Cond}_\Sigma \) consists of such conditions for a finite set of functions. Here \( w \) defines the type of the auxiliary variable \( w \) (e.g., \( f_1 \) and \( f_2 \) in Fig. 3) that is shared among \( D, P \) and \( Q \), which are explained below.

\( D(w) \), given \( w \in W \), specifies the maximum call depth. This component is used to specify **pure calls**, which can be automatically eliminated by the ACT theorem since they (i) always terminate (ii) without triggering any observable event. For example, in Fig. 1, calls to \( \text{Mem} \) and \( \text{Map} \) are all eliminated in \( \text{App} \) since they are pure. Specifically, a depth \( d \in D \) is either \( \infty \) denoting (potential) impurity, or an ordinal \( (o) \) denoting purity (with a maximum call depth \( o \)). We then give a well-founded ordering \( < \) on Depth.

Now we see how we can \textit{locally} impose purity (i.e., termination and absence of observable events). First, we require the depth to \textit{strictly} decrease for a pure call (see line 3 in APCDef). From this (at the point of applying ACT) it follows that any chain of pure calls always terminates. Second, when a function is invoked with depth \( (o) \) (i.e., a pure call), its body is replaced with APCDef that can only nondeterministically make a finite number of arbitrary pure calls (see line 11 in FunDef). From this (at the point of applying ACT) it follows by construction that pure calls do not trigger any observable event. Note that purity of a function may depend on its argument (e.g., printing an error for an invalid argument but otherwise behaving purely). The technical report [3, §3.2] presents more detailed explanation about purity with an example.

\( P(w)/Q(w) \), given \( w \in W \), specifies a pre/post condition on (i) a concrete argument/return value, (ii) an abstract argument/return value, and (iii) an argument/return resource. The notions of concrete and abstract values are used to abstract values passed between functions. For example, we can abstract a function taking a pointer to a linked list (i.e., concrete value) into that taking a mathematical list (i.e., an abstract value). The technical report [3, §3.4] presents such an example.

It is important to note that even though it does not make sense to send or receive abstract values to and from contexts since the contexts are arbitrary, we can again make such an illusion using \text{choose} and \text{take}. Specifically, when an abstract function passes an abstract value to a context, we \text{choose} a concrete value that satisfies the required pre/post condition together with the abstract value and the chosen resource; conversely, when a context passes a concrete value, we \text{take} an abstract value satisfying the required condition and pass it to the abstract function.

Note also that SPC allows us to specify conditions as \text{rProp}_\Sigma-level predicates following Iris [21] and supports the IPM (Iris Proof Mode) [28] for reasoning about them.

In SPC, we can also write abstract yet executable code for each function as an itree, a collection of which form a **pre-abstraction** \( A \in \text{PAbs} \). Concretely, \( A \) is the same as an EMS module except that it can trigger an extra event, APC, which is interpreted as nondeterministically making arbitrary pure calls (w.r.t. the input conditions) in conditional abstractions but eliminated after applying the ACT theorem. Note that APC is implicitly inserted at each line (via a macro expansion for the bind operator) because they will be freely eliminated by ACT. For example, in Fig. 1, when \( \text{Mem}.\text{store} \) or \( \text{Mem}.\text{load} \) is invoked in \( \text{P}_w^\text{cond} \), the same call can be made in \( \text{rProp}_w^\text{cond} \) via APC since they are pure calls.

**Embedding into EMS.** A pre-abstraction \( A \) together with conditions are translated into EMS in two ways as follows.

- \([A] \in \text{EMS}, \text{called abstraction}, \) is obtained by eliminating all APC events (i.e., replacing them by \text{ret} ()).
- \([S_m \times A, \sigma : S_{\text{out}}] \in \text{EMS}, \text{called conditional abstraction}, \) has \( (A,\text{init}, \sigma) \) as an initial module state together with function definitions generated by FunDef, where \( S_m \) is conditions about the functions that \( A \) invokes, \( S_{\text{out}} \) conditions about the functions that \( A \) defines, and \( \sigma \) an initial module resource of \( A \).

For example, \( I_{\text{hmf}}, A_{\text{hmf}}, I_{\text{app}}, \text{and } A_{\text{app}} \) (with implicit APC’s) are (pre)abstractions and \( I_{\text{cond}}, A_{\text{cond}}, I_{\text{cond}}^\text{app}, \text{and } A_{\text{cond}}^\text{app} \) are conditional abstractions.

FunDef is a formal definition of what we have explained so far, where by using the interpretation function for itrees, we introduce the \text{frm} variable and replace (i) each call with \text{CallDef}, which adds the condition for the callee around the call; (ii) each \text{APC} with APCDef, which makes a finite number of arbitrary pure calls, and (iii) \text{Put} and \text{Get} with accessing the first component of the module local state since it is extended with \text{mres} in the second component. Note that \( (S_m \text{fn})! \) triggers \text{NB} when \( \text{fn} \not\in \text{dom}(S_m) \), and \( \text{ret}_\Sigma^\text{Any} \times \Sigma \) triggers UB when the assigned value is not of type \text{Any} \times \Sigma.
3.3 Key theorems of CCR

**Theorem 3.1 (Assumptions).**

For a global PCM $\Sigma$, conditional abstractions $[S \triangleright ((A_1, \sigma_1) : S_1) \circ \ldots \circ (A_n, \sigma_n) : S_n] \leq \text{ctx}$ for $i \in \{1, \ldots, n\}$ with $S \subseteq S_1 \cup \ldots \cup S_n$, and an initial resource $\sigma$ to satisfy its precondition and $\forall (\sigma + \sigma_1 + \ldots + \sigma_n)$,

$[S \triangleright ((A_1, \sigma_1) : S_1) \circ \ldots \circ (A_n, \sigma_n) : S_n] \leq \text{ctx}$

Here we remark that as a corollary of the ACT theorem, CCR can also serve as a framework for modern separation logics, but in an operational style without step-indexing. For this, we define special pre-abstractions $\text{Safe}(n_{\text{in}}, n_{\text{out}})$, which define functions with their sizes in $n_{\text{out}}$ to only nondeterministically invoke arbitrary functions in $n_{\text{in}}$ with arbitrary arguments for any (finite or infinite) number of times.

**Lemma 3.2 (Safety).** For $n_s \subseteq n_{\text{s1}} \cup \ldots \cup n_{\text{sn}}$, $\text{Safe}(n_s, n_{\text{s1}}) \circ \ldots \circ \text{Safe}(n_s, n_{\text{sn}})$ does not produce an error.

**Corollary 3.3 (SL).** Given a global PCM $\Sigma$, $(P_i, S_i, \sigma_i)$ for $i \in \{1, \ldots, n\}$ with $S \subseteq S_1 \cup \ldots \cup S_n$, and an initial resource $\sigma$ to satisfy its precondition and $\forall (\sigma + \sigma_1 + \ldots + \sigma_n)$,

$[V_i : P_i \leq \text{ctx} [S \triangleright \text{Safe}(\text{dom}(S), \text{dom}(S_i)), \sigma_i : S_i]]$  

$\implies P_1 \circ \ldots \circ P_n$ does not produce an error.

This corollary can be seen as a separation logic because $P_i \leq \text{ctx} [S \triangleright \text{Safe}(\text{dom}(S), \text{dom}(S_i)), \sigma_i : S_i]$ essentially amounts to proving, in SL, that $P_i$ satisfies the pre and post conditions of $S_i$ assuming other modules satisfy those of $S$. Moreover, by employing the Iris Proof Mode [28], the actual proofs of these refinements in CCR look similar to those in Iris.

3.4 Imp and its verified compiler

The IMP language, extended from IMP [43], has standard syntax and semantics built on a simplified version of the CompCert memory model. In particular, IMP computes with the set of values, val, consisting of 64-bit integers, int, and memory and function pointers, ptr. When embedding IMP into EMS, we cast back and forth between val and Any, and if the downcast from Any to val fails, trigger UB.

We also develop a verified compiler2 from IMP to CSharpMinor of CompCert [29], which is then composed with CompCert to give a verified compiler $\downarrow$ from IMP to assembly.

**Theorem 3.4 (Separate Compilation Correctness).** Given $(P_i, \text{Asm}_i)$ with $[P_i] = \text{Asm}_i$ for $i \in \{1, \ldots, n\}$, $\text{Beh}(\text{Asm}_1 \circ \ldots \circ \text{Asm}_n) \subseteq \text{Beh}(P_{\text{imp}} \circ P_1 \circ \ldots \circ P_n)$.

Here $\circ$ is the syntactic linking operator of CompCert, and $P_{\text{imp}}$ is an EMS module (directly written as an iset) that implements our memory model (i.e., a simplified version of CompCert’s).

4 Examples

Fig. 6 shows the conditions for the modules used in the motivating example of Fig. 1, where we use the notation $\forall w : W. \{d \vdash x. P\} \{r. Q\}$ as a shorthand for the condition

$\forall w : W. \{d \vdash x. P\} \{r. Q\}$

2As a simple solution to resolve a subtle mismatch between CompCert’s memory model and ours, we compile the free instruction to skip for now. Also we support separate compilation following the approach of [23].

3We cast CompCert’s events into 0bs events in EMS.
5 Reasoning about function pointers

We present a general pattern for doing higher-order reasoning in CCR without requiring any special support. For this, consider the simple example given in Fig. 7. The function repeat(f, n, m) in Prop recursively apply *f, n times, to m, where *f is the function pointed to by the pointer value f. The definitions in Prop and AD are straightforward to understand except that &SC.succ is the pointer value pointing to the function SC.succ. The pre-abstractions &Prop and &SC are empty since they are pure. The pre-abstraction &AD turns the call to RP.repeat into the addition.

To specify RP.repeat, we essentially need to embed expected conditions for argument functions f inside the condition of RP.repeat. Directly supporting this would make the definition of condition more involved since we need to solve a recursive equation to define it. Although such an equation could be solved by employing the step-indexing technique, here we propose a more elementary solution that does not introduce any cyclic definition.

Now we see how to do it. First, we give a higher-order condition &Dyn to the module RP, given in Fig. 7, which is given as a function from conditions to conditions. Concretely, given St, for arguments f, n, and m and a mathematical function fsem, the condition &Dyn(St) assumes St to include the expected specification for *f (saying that *f is pure with measure (ω + n)) and returns fsem(m) for any argument m, and then guarantees that the return value is fsem"(m). Here ω is the smallest ordinal bigger than every natural number and thus *f is allowed to have any finite recursion depth. Also we require RP.repeat to be pure with measure (ω + n) because it makes recursive calls with depth n followed by a call to *f.

Then we verify RP. For any St and any S ⊇ (St & HDyn(St)) (since RP.repeat makes a call to *f and itself), we prove: PProp ≤cxs [S ∗ (PProp, ε) : HDyn(St)]. Also, we verify SC. For any S, we prove: PProp ≤cxs [S ∗ (AS, ε) : SC].

Finally, we instantiate the CRs with St = &SC and S = HDyn(SC) ∪ SC ∪ &AD and apply ACT to them with ε to main():

PProp ◦ PProp ◦ PProp ≤cxs [S ∗ (PProp, ε) : HDyn(SC)] ◦ [S ∗ (AS, ε) : SC] ◦ [S ∗ (AD, ε) : AD]

As an advanced example, we also verify Landin’s knot [6] (see our Coq development [3]). We believe this pattern is a general solution to higher-order reasoning in practice since it applies to all practical examples we can think of.

6 Evaluation, related and future work

Evaluation. Our development comprises 37,329 SLOC of Coq (counted by coqwc), including 10,100 SLOC for all the examples in the paper and technical report [3]. For differential testing, we ran each example in two ways (by extracting both implementation and abstraction to OCaml) and compared the results. Interestingly, we found two mis-downcast bugs in the ECHO example [3, §3.4] by testing it before verification.
Related work. Main related works are discussed in §1. Here we discuss other related works.

Relational Hoare/separation logics [5, 44] establish refinement (or equivalence) between programs. However, they do not support conditions on external functions but only on input and output states. More specifically, they do not allow invoking unknown functions relying on their specifications. Other logics [24, 36] proving a form of refinement also only establish unconditional refinement.

There are non-relational program logics (i.e., proving safety not refinement or equivalence) that support higher-order specifications without using step-indexing. First, the key idea of CFML [9–11] to avoid step-indexing is essentially similar to our idea presented in §5, although applied in a different setting (i.e., unary program logic instead of conditional refinement). Second, XCAP [32] and Bedrock [12] use a syntactic technique to avoid step-indexing, where higher-order predicates are treated as syntactic objects.

Future work. Since CCR is a new framework that spans refinement-style verification, logic-style verification, and testing, there are various future research directions: (i) supporting (relaxed-memory) concurrency in the style of Iris [21], (ii) embedding assembly into EMS in the style of CompCertM [37], enabling verification of software composed of C and assembly, which can be lowered to assembly via compilation with CompCertM, and (iii) developing property-based testing tools for efficient differential testing between an implementation and its abstraction.

References

[1] Andrew W. Appel. 2011. Verified Software Toolchain. In Proceedings of the 20th European Symposium on Programming (ESOP 2011).
[2] Andrew W. Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. ACM Trans. Program. Lang. Syst. 23, 5 (sep 2001), 657–683. https://doi.org/10.1145/504709.504712
[3] Anonymous Author(s). 2021. CCR: Technical documentations and Coq developments.
[4] Ralph-Johan Back and Joakim Wright. 2012. Refinement calculus: a systematic introduction. Springer Science & Business Media.
[5] 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
[6] Lars Birkedal and Aleš Bizjak. 2020. Lecture notes on iris: Higher-order concurrent separation logic. https://iris-project.org/tutorial-material.html
[7] Rastislav Bodik, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tang, Shaon Barman, and Casey Rodarmor. 2010. Programming with angelic nondeterminism. In Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 339–352.
[8] Cristiano Calcagno, Peter W O’Hearn, andHongseok Yang. 2007. Local action and abstract separation logic. In 22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007). IEEE, 366–378.
[9] Arthur Charguéraud. 2011. Characteristic Formulae for the Verification of Imperative Programs. In Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (Tokyo, Japan) (ICFP ’11). Association for Computing Machinery, New York, NY, USA, 418–430. https://doi.org/10.1145/2034773.2034828
[10] Arthur Charguéraud. 2016. Higher-Order Representation Predicates in Separation Logic. In Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs (St. Petersburg, FL, USA) (CPP 2016). Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/2854065.2854068
[11] Arthur Charguéraud. 2020. Separation Logic for Sequential Programs (Functional Pearls). Proc. ACM Program. Lang. 4, ICFP, Article 116 (aug 2020), 34 pages. https://doi.org/10.1145/3408998
[12] Adam Chlipala. 2013. The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier. In Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (Boston, Massachusetts, USA) (ICFP ’13). Association for Computing Machinery, New York, NY, USA, 391–402. https://doi.org/10.1145/2500365.2500592
[13] 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.
[14] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. Logical Methods in Computer Science Volume 17, Issue 3 (Jul 2021). https://doi.org/10.46298/lmcs-17(3:9)2021
[15] 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).
[16] 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).
