# Abstraction Logic: The Marriage of Contextual Refinement and Separation Logic

YOUNGJU SONG, Seoul National University, Korea MINKI CHO, Seoul National University, Korea DONGJAE LEE, Seoul National University, Korea CHUNG-KIL HUR, Seoul National University, Korea

Contextual refinement and separation logics are successful verification techniques that are very different in nature. First, the former guarantees behavioral refinement between a concrete program and an abstract program while the latter guarantees safety of a concrete program under certain conditions (expressed in terms of pre and post conditions). Second, the former does not allow any assumption about the context when locally reasoning about a module while the latter allows rich assumptions.

In this paper, we present a new verification technique, called abstraction logic (AL), that inherently combines contextual refinement and separation logics such as Iris and VST, thereby taking the advantages of both. Specifically, AL allows us to locally verify a concrete module against an abstract module under separation-logic-style pre and post conditions about external modules. AL are fully formalized in Coq and provides a proof mode that supports a combination of simulation-style reasoning using our own tactics and SL-style reasoning using IPM (Iris Proof Mode). Using the proof mode, we verified various examples to demonstrate reasoning about ownership (based on partial commutative monoids) and purity (i.e., termination with no system call), cyclic and higher-order reasoning about mutual recursion and function pointers, and reusable and gradual verification via intermediate abstractions. 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 [Gu et al. 2015] and program logics [Hoare 1969] (most notably, separation logics [O'hearn 2007; Reynolds 2002]) are two successful verification techniques. The former is typically used for compiler verification, where both implementation and specification are given as executable programs and its verification establishes that all possible observable behaviors of the implementation program under an arbitrary context are included in those of the specification program under the same context. The latter are mainly used for verification of a program against its logical specification, where the specification is given as a pair of pre and post conditions and its verification typically establishes that if the program starts with a state satisfying the pre condition, then it executes safely, and if it terminates, the final state satisfies the post condition.

These two techniques are very different in nature and have their own advantages. First, specifications in contextual refinement describe intended dynamic behaviors including interactions with its context/environment and side effects such as fatal errors and non-termination, while those in program logics typically describe sufficient conditions for *safe* executions (*i.e.*, those without fatal errors in case of partial correctness, and in addition without non-termination in case of total correctness). Second, contextual refinement allows to verify one aspect at a time via multiple intermediate (specification) programs since they are transitively composable (*e.g.*, a compiler translation consists of a sequence of optimizations, each of which is separately verified using a possibly

Authors' addresses: Youngju Song, Seoul National University, Korea, youngju.song@sf.snu.ac.kr; 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.

different simulation relation), while program logics require to find and verify safety conditions for the whole program in one step (although the conditions may be refined in further steps). Third, when locally reasoning about a module, in contextual refinement one cannot make any assumptions about external modules because they are completely arbitrary, while in program logics one can make specific assumptions (*i.e.*, safety conditions) about each external module since the verified module is only composed with other *verified* modules instead of *arbitrary* modules.

In this paper, we present a new verification framework, called abstraction logic (AL), that *inherently* combines the two techniques of contextual refinement (CR) and separation logic (SL), thereby taking the advantages of both.

**High-level overview of AL**. We first highlight how AL is different from CR and the standard version of SL using an abstract example (See §8 for comparison with other variations of SL). For this, consider two modules, named M with a function f and named N with a function g, and suppose we have their implementations  $I_1$ ,  $I_2$  and SL-style specifications  $S_1$ ,  $S_2$ .

In SL, one can *locally* verify each implementation  $I_i$  for  $i \in \{1, 2\}$  against its specification  $S_i$  assuming both specifications  $S_1, S_2$ , which we denote by  $S_1, S_2 \vdash I_i : S_i$ . Then SL combines the two verification results as follows:

$$S_1, S_2 \vdash I_1 : S_1$$
  
 $S_1, S_2 \vdash I_2 : S_2$   
 $I_1 \circ I_2 \text{ safe}$ 

It guarantees that the linked program  $I_1 \circ I_2$  only produces safe behaviors (*i.e.*, no fatal errors). For example, if M.f()  $\equiv$  (1+1) / N.g() in  $I_1$ , then by assuming N.g() is safe and returns 1, which is specified in  $S_2$ , we can prove that M.f() is safe and returns 2, which is specified in  $S_1$ . It is important to note that SL in general cannot guarantee anything when verified modules are composed with unverified ones because those unverified may trigger a fatal error. For example, if M.f()  $\equiv$  (L.h(); (1 + 1) / N.g()) in  $I_1$  and the implementation  $I_3$  for the module L is unverified, SL cannot guarantee safety of  $I_1 \circ I_2 \circ I_3$  since  $I_3$  might not be safe.

On the other hand, CR provides verification results that are valid under unverified contexts. Specifically, in CR, one can  $locally^1$  verify  $I_1$  against another more abstract implementation  $A_1$ , simply called *abstraction*, for M, which we denote by  $I_1 \leq_{\text{ctx}} A_1$ . Then by the definition of CR, we have the following result for an arbitrary (even unsafe or abstract) implementation C for N.

$$\frac{I_1 \leq_{\text{ctx}} A_1}{\text{Beh}(I_1 \circ C) \subseteq \text{Beh}(A_1 \circ C)}$$

For example, when  $M.f() \equiv (1+1) / N.g()$  in  $I_1$ , we can verify it against  $A_1$  with  $M.f() \equiv 2 / N.g()$ , which is valid even when N is an unverified arbitrary module because the behavioral refinement preserves even crash or non-termination behaviors between the implementation and abstraction (e.g., if the implementation crashes, so does the abstraction). However, its limitation is that we cannot verify  $I_1$  against more useful abstractions such as  $M.f() \equiv 2$  because N.g() in C is arbitrary and thus may not return 1.

The key innovation in AL is that we overcome the limitation of CR by internalizing, inside a module, SL-style specifications about other modules. To see this, revisit the above problematic example where M.f()  $\equiv$  (1 + 1) / N.g() in  $I_1$  and M.f()  $\equiv$  2 in  $A_1$  and thus  $I_1 \leq_{\text{ctx}} A_1$  does not hold. To solve this problem, from  $A_1$  together with the above specifications  $S_1$ ,  $S_2$  (i.e., saying that M.f() returns 2 and N.g() returns 1), AL derives a special abstraction for M, denoted  $[S_1, S_2 \rtimes A_1 : S_1]$  and called *abspec*, that internalizes  $S_1$ ,  $S_2$  inside  $A_1$ . Then, one can actually prove  $I_1 \leq_{\text{ctx}} [S_1, S_2 \rtimes A_1 : S_1]$ ,

<sup>&</sup>lt;sup>1</sup>In theory, it might be possible to *globally* verify  $I_1 \circ I_2$ , where you can make assumptions about  $I_1$  and  $I_2$ . However, it would sacrifice the power of local reasoning and compositionality, so we aim high to support fully compositional verification.

which essentially amounts to proving that the behaviors of  $I_1$  refines those of  $A_1$  and satisfies  $S_1$  under arbitrary contexts but assuming the specifications  $S_1$  and  $S_2$ . Note that this verification is local to  $I_1$  and  $A_1$  for M just relying on the specifications  $S_1$  and  $S_2$ , so that the refinement can hold under an arbitrary implementation C for N (as required by the definition of  $\leq_{ctx}$ ). Even further, when M.f()  $\equiv$  (L.h(); (1 + 1) / N.g()) in  $I_1$  and M.f()  $\equiv$  (L.h(); 2) in  $A_1$  with the implementation of L unknown (i.e., L may even be unsafe or make arbitrary calls including mutually recursive calls to the modules M and N), we can still prove  $I_1 \leq_{ctx} [S_1, S_2 \rtimes A_1 : S_1]$  for  $S_1, S_2$  the same as above without assuming anything about L.

To combine local verification results, AL provides the following theorem for any  $S_1$ ,  $S_2$ ,  $A_1$ ,  $A_2$ .

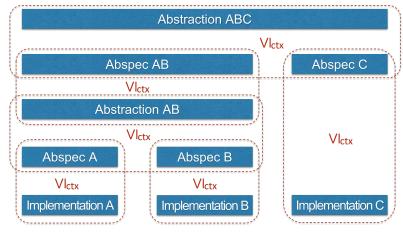
$$[S_1, S_2 \rtimes A_1 : S_1] \circ [S_1, S_2 \rtimes A_2 : S_2] \leq_{\mathsf{ctx}} A_1 \circ A_2$$

Then, by horizontal and vertical compositionality of CR and the definition of CR, we can derive, *e.g.*, the following corollary for any (even unsafe or abstract) implementation *C* for L:

$$\frac{I_1 \leq_{\operatorname{ctx}} [S_1, S_2 \rtimes A_1 : S_1]}{I_2 \leq_{\operatorname{ctx}} [S_1, S_2 \rtimes A_2 : S_2]}$$
$$\underbrace{Beh(I_1 \circ I_2 \circ C) \subseteq Beh(A_1 \circ A_2 \circ C)}$$

We have two advantages from the fact that the resulting abstraction  $A_1 \circ A_2 \circ C$  is also an executable program.

- (1) One can test the abstraction by executing it, so that we can more easily see whether the abstraction works as intended.
- (2) One can treat the abstraction as an implementation and further verify it using AL, which allows gradual abstraction from an implementation to the top-level abstraction in multiple steps and can also increase reusability of verification (See §3.3 for a concrete example). An example of such gradual and modular verification is depicted as follows.



Finally, we remark that AL can be seen as subsuming both CR and SL. The former is achieved by not making any assumptions, and the latter because AL can also be used to prove safety guarantees as follows. For example, for any  $I_1$ ,  $I_2$  and  $S_1$ ,  $S_2$ , by setting  $A_i$  to be a special abstraction, called Safe, we have the following:

$$I_{1} \leq_{\operatorname{ctx}} [S_{1}, S_{2} \rtimes \operatorname{Safe} : S_{1}]$$

$$I_{2} \leq_{\operatorname{ctx}} [S_{1}, S_{2} \rtimes \operatorname{Safe} : S_{2}]$$

$$\operatorname{Beh}(I_{1} \circ I_{2}) \subseteq \operatorname{Beh}(\operatorname{Safe} \circ \operatorname{Safe})$$

where proving  $I_i \leq_{\text{ctx}} [S_1, S_2 \rtimes \text{Safe} : S_i]$  essentially amounts to proving  $S_1, S_2 \vdash I_i : S_i$  in SL. Then since Beh(Safe  $\circ$  Safe) is a set of safe behaviors, we can conclude that  $I_1 \circ I_2$  only produces safe behaviors.

*Contributions.* In this paper, we developed the theory of abstraction logic (currently in a sequential setting) and tools for it including the AL proof mode and a verified compiler for AL down to assembly. All results are fully formalized in Coq [The Coq Development Team 2021] and summarized as follows.

- (1) We developed the first *comprehensive* theory, AL, that enables establishing contextual refinement via powerful local reasoning that allows us to rely on SL-style specifications. In particular, AL allows us to express various ownership via PCMs (Partial Commutative Monoids) [Calcagno et al. 2007] as in the state-of-the-art SLs such as Iris [Jung et al. 2015] and VST [Appel 2011].
- (2) We developed EMS (Executable Module Semantics) by generalizing Interaction Trees [Xia et al. 2019] to support module systems and two kinds of nondeterminism, called *choose* and *take*. In particular, via Coq's extraction mechanism, we can extract an EMS to an executable program in OCaml as done in Interaction Trees. Also, we developed two sub-languages IMP and SPC that are embedded into EMS (via deep embedding for IMP and shallow embedding for SPC).
  - IMP: a C-like language with integer and (function and memory) pointer values, which is
    used to write underlying implementations like I<sub>1</sub> above that are verified against higher-level
    abstractions using AL and also compiled down to assembly via our verified compiler for
    IMP.
  - SPC: a specification language, which is used to write an abspec, which is a combination of an abstraction and SL-style specifications like  $[S_1, S_2 \rtimes A_1 : S_1]$  above.
- (3) We developed the AL proof mode that supports a combination of simulation-style reasoning and SL-style reasoning by allowing smooth switching between the two styles during a single proof, which is essentially necessary because AL inherently combines the two techniques. Specifically, for the former we developed our own tactics that allow to set up a simulation relation (possibly with a module-local relational invariant) between the implementation and abspec, and reason about it stepwise; and for the latter we employed the IPM (Iris Proof Mode) package (*i.e.*, by instantiating it with our AL theory) to streamline the process for reasoning about separating conjunction, magic wand, and PCM resources.
- (4) Using the AL proof mode, we verified various examples written in IMP, which demonstrates reasoning about PCM-based ownership, proving and exploiting *purity* (*i.e.*, termination with no system call), cyclic and higher-order reasoning about recursion and function pointers, and reusable and gradual verification via intermediate abstractions. Note that in spite of supporting cyclic and higher-order reasoning, AL does not rely on any step-indexing techniques [Ahmed 2006].
- (5) Finally, we developed a *verified* compiler for IMP targeting Csharpminor of CompCert [Leroy 2006], which is verified in the style of CompCert's verification. Therefore, we formally establish behavioral refinement from the top-level abstractions of the above examples, all the way down to their assembly code generated by the IMP compiler and CompCert.

#### 2 KEY IDEAS

We gradually introduce the key ideas behind AL by presenting how to express Hoare logic specifications (§2.1) and separation logic specifications (§2.2).

```
I_{\mathsf{Main}} \coloneqq [\mathsf{Module} \ \mathsf{Main}]
                                                                                 I_{\mathsf{F}} := [\mathsf{Module} \ \mathsf{F}]
           def main() ≡
                                                                                         def f(x) \equiv
              var x := 40:
                                                                                            var r := x*x/4 + x + 1;
               var r := F.f(x);
                                                                                            print(r);
               if (r\%2 == 1) then print(42) else 1/0
S_{Main} := \{ \{True\} Main.main \{True\} \}
                                                                                 S_{\mathsf{F}} := \{ \{ \lambda x. \ x \% \ 4 = 0 \} \ \mathsf{F.f} \ \{ \lambda r. \ r \% \ 4 = 1 \} \}
A_{\mathsf{Main}} := [\mathsf{Module} \ \mathsf{Main}]
                                                                                  A_{\mathsf{F}} \coloneqq [\mathsf{Module} \; \mathsf{F}]
            def main() ≡
                                                                                          def f(x) \equiv
                var x := 40;
                                                                                              var r := |(x/2 + 1)**2|;
                var r := F.f(x);
                                                                                              print(r);
                print(42)
[S_{\mathsf{Main}} \cup S_{\mathsf{F}} \rtimes A_{\mathsf{Main}} : S_{\mathsf{Main}}] :=
                                                                                 [S_{\mathsf{Main}} \cup S_{\mathsf{F}} \rtimes A_{\mathsf{F}} : S_{\mathsf{F}}] :=
   [Module Main]
                                                                                     [Module F]
   def main() \equiv
                                                                                     def f(x: int) \equiv
       var x := 40;
                                                                                        assume(x\%4 == 0);
       guarantee(x%4 == 0);
                                                                                        var r := (x/2 + 1)**2;
       var r := F.f(x);
                                                                                        print(r);
       assume(r%4 == 1);
                                                                                        guarantee(r%4 == 1);
       print(42)
```

Fig. 1. Implementations, HL specifications, Abstractions, and Abspecs for Main and F

# 2.1 Hoare logic specifications in abstraction logic

To demonstrate how to express Hoare logic (HL) specifications in abstraction logic, consider the implementations  $I_{Main}$  and  $I_F$  of the modules Main and F, shown in Fig. 1. Here Main.main() (i) invokes F.f(x) with x = 40, which computes x\*x/4+x+1, prints it out via the system call *print* and returns it; (ii) if the result is an odd number, prints 42; (iii) otherwise, crashes by executing division by zero. In HL, the specification  $S_{Main}$  says that Main.main() runs safely, assuming the specification  $S_F$ , which says that if the argument of F.f is a multiple of 4, it runs safely and returns a number r such that r modulus 4 is 1.

In AL, assuming the specifications  $S_{\text{Main}}$  and  $S_F$  (ignoring the safety), we would like to *locally* verify  $I_{\text{Main}}$  and  $I_F$  against, *e.g.*, the abstractions  $A_{\text{Main}}$  and  $A_F$  in Fig. 1, where the abstracted parts are boxed and \*\* is the exponentiation operator. First, note that without any assumption neither  $I_{\text{Main}} \leq_{\text{ctx}} A_{\text{Main}}$  nor  $I_F \leq_{\text{ctx}} A_F$  holds because their context implementations are arbitrary. For example, 0 may be given for r in Main.main(), and 1 may be given for x in F.f(x), in which cases the implementation and abstraction behave differently. The question here is how to *internalize* the specifications inside  $A_{\text{Main}}$  and  $A_F$ .

The abspecs  $[S_{\text{Main}} \cup S_F \rtimes A_{\text{Main}} : S_{\text{Main}}]$  and  $[S_{\text{Main}} \cup S_F \rtimes A_F : S_F]$  internalizing  $S_{\text{Main}}$ ,  $S_F$  inside  $A_{\text{Main}}$  and  $A_F$  are given in Fig. 1, where we simply insert **assume** and **guarantee** commands according to  $S_{\text{Main}}$  and  $S_F$ . Specifically, F.f(x) assumes its precondition x%4 == 0 at the beginning and guarantees its postcondition r%4 == 1 at the end. Conversely, Main.main() guarantees the precondition of F.f(x) before invoking it and assumes the postcondition of F.f(x) after the invocation. Note that we simply omitted **assume**(true) and **guarantee**(true) corresponding to  $\{True\}$   $\{True\}$  of Main.main() because they do nothing as we will see below.

Now the computational interpretation of **assume** and **guarantee** is given as follows. First, assume(P) for a proposition P does nothing if P holds; otherwise triggers *undefined behavior* (UB), which is a standard notion (e.g., in CompCert) and interpreted as exhibiting all possible

behaviors:

$$\frac{\text{assume}(P)}{\text{e}} \stackrel{\text{def}}{=} \text{if } P \text{ then skip else UB}$$

To see the intuition, consider proving  $I_F \leq_{\text{ctx}} [S_{\text{Main}} \cup S_F \rtimes A_F : S_F]$ . If the argument x to F. f does not satisfy x%4 == 0, the abspec triggers **UB** thereby exhibiting all possible behaviors, which trivially include whatever behavior the implementation may exhibit. Therefore in the verification we do not need to consider the cases where the assume command fails (*i.e.*, we can assume it holds).

Second, **guarantee**(P) does nothing if P holds; otherwise triggers *no behavior* (NB), which appeared in [Kang et al. 2015; Ševčík et al. 2013] and is interpreted as exhibiting no behaviors (see §4 for the formal definition):

$$guarantee(P) \stackrel{\text{def}}{=} if P then skip else NB$$

Again, for  $I_F \leq_{\text{ctx}} [S_{\text{Main}} \cup S_F \rtimes A_F : S_F]$ , suppose the argument x satisfies x%4 == 0 and then both implementation and abspec will compute and print the same value r. Now if r does not satisfy r%4 == 1, the abspec triggers **NB** thereby exhibiting no behaviors, which does not include whatever behavior the implementation may exhibit (unless it also triggers **NB**, which is not the case here). Therefore in the verification we must prove that the guarantee command succeeds (*i.e.*, we should guarantee it holds).

Then, we can actually prove  $I_{Main} \leq_{ctx} [S_{Main} \cup S_F \rtimes A_{Main} : S_{Main}]$  and  $I_F \leq_{ctx} [S_{Main} \cup S_F \rtimes A_F : S_F]$ . For the former, we first prove **guarantee**(x%4 == 0) succeeds since x is 40; then for any given r from F.f(x), we can assume r%4 == 1, which implies r%2 == 1 thereby proving that both implementation and abspec print 42. For the latter, we can assume x%4 == 0 thereby proving that both implementation and abspec compute and print the same value r. Then **guarantee**(r%4 == 1) succeeds since r is (x/2 + 1)\*\*2 and we assumed x%4 == 0. Finally, both return the same r. Finally, we can see that the following hold:

$$Beh([S_{Main} \cup S_F \rtimes A_{Main} : S_{Main}] \circ [S_{Main} \cup S_F \rtimes A_F : S_F]) \subseteq Beh(A_{Main} \circ A_F)$$

which easily follows using the following lemma: for any proposition P and any program with a hole K[-],

$$Beh(K[guarantee(P); assume(P)]) \subseteq Beh(K[skip])$$

This lemma holds trivially if *P* holds; otherwise it holds since the left hand side exhibits no behavior.

# 2.2 Separation logic specifications in abstraction logic

Now to see how we can express separation logic (SL) specifications in abstraction logic, consider the implementations  $I_{\text{Cannon}}$  and  $I_{\text{Main}}$  of the two modules Main and Cannon, shown in Fig. 2. Here Main.main() invokes Cannon.fire() and prints the return value for a fixed number, NUM\_FIRE, of times. The Cannon module consists of (i) the module-local variable powder (i.e., not accessible to other modules), which is initially set to 1 and (ii) the function fire, which sets the variable r to be 1/powder, prints r, decrements powder by 1, and returns r. Note that Cannon.fire() can be safely invoked only once because it will crash due to division by zero at the second invocation; therefore, if NUM\_FIRE is 2, the whole program crashes.

We briefly discuss how one can prove safety of the program when NUM\_FIRE is 1 in SL. First, the SL specification  $S_{Cannon}$  (given in Fig. 2) says that if the resource Ball is logically given, Cannon.fire() safely executes and returns 1 but does not logically give the Ball back (*i.e.*, Ball is *consumed*). Here one can intuitively understand a *resource* as something that is neither duplicable nor creatable out of nothing. In this example, we can design the universe of resources—via a general mechanism based on PCMs (Partial Commutative Monoids)—in such a way that there can be at most one Ball, relying on which we can then *locally* prove the safety of Cannon.fire() since it requires

```
I_{\mathsf{Main}} \coloneqq [\mathsf{Module} \ \mathsf{Main}]
                                                                  I_{Cannon} := [Module Cannon]
          def main() ≡
                                                                               local powder = 1
             repeat NUM_FIRE {
                                                                               def fire() ≡
                var r := Cannon.fire();
                                                                                  var r := 1/powder;
                print(r)
                                                                                  print(r);
                                                                                  powder := powder - 1;
                                                                 S_{\mathsf{Cannon}} := \left\{ \left\{ \left[ \bar{\mathsf{Ball}} \right] \right\} \mathsf{Cannon.fire} \left\{ r. r = 1 \right\} \right\}
S_{\text{Main}} := \{\{ | \text{Ball} | \} | \text{Main.main} \{ | \text{True} \} \}
\sigma_{\text{Main}} := \text{Unit}
                                                                 \sigma_{Cannon} := Ready
A_{\mathsf{Main}} := [\mathsf{Module} \; \mathsf{Main}]
                                                                 A_{Cannon} := [Module Cannon]
           def main() \equiv
                                                                               def fire() ≡
              repeat NUM_FIRE {
                 var r := Cannon.fire();
                                                                                  var r := 1;
                 print(| 1 |)
                                                                                  print( 1 );
[S_{\text{Main}} \cup S_{\text{Cannon}} \rtimes (A_{\text{Main}}, \sigma_{\text{Main}}) : S_{\text{Main}}] :=
                                                                 [S_{\mathsf{Main}} \cup S_{\mathsf{Cannon}} \rtimes (A_{\mathsf{Cannon}}, \sigma_{\mathsf{Cannon}}) : S_{\mathsf{Cannon}}] :=
   [Module Main]
                                                                    [Module Cannon]
   local res_m := Unit
                                                                    local res_m := Ready
   def main() ≡
                                                                    def fire() ≡
                                                                       var res_f := Unit;
      var res_f := Unit;
      ASSUME(\lambdares. res == Ball)
                                                                       ASSUME(\lambdares. res == Ball)
                                                                       var r := 1;
      repeat NUM_FIRE {
                                                                       print(1);
         GUARANTEE(\lambdares. res == Ball)
         var r := Cannon.fire();
                                                                       GUARANTEE(\lambdares. res == Unit && r == 1)
         ASSUME(\lambdares. res == Unit && r == 1)
         print(1)
     GUARANTEE(\lambda res. res == Unit)
ASSUME(Cond) \equiv \{
                                                                GUARANTEE(Cond) \equiv \{
   var res := take(PCM<sub>Cannon</sub>);
                                                                    (res_m, res_f) := fpu(res_m, res_f);
   assume(Cond(res));
                                                                   var res := choose(PCM<sub>Cannon</sub>);
   res_f := res_f + res;
                                                                   guarantee(Cond(res));
   assume(res_m + res_f != Undef);
                                                                    res_f := minus(res_f, res);
```

Fig. 2. Implementations, SL specifications, Initial resources, Abstractions, and Abspecs for Main and Cannon

Ball and thus can be invoked at most once. Second, the SL specification  $S_{Main}$  says that given a Ball, Main.main() safely executes, which is *locally* provable relying on  $S_{Cannon}$  since NUM\_FIRE is 1 and thus Cannon.fire() is invoked only once. Indeed, if NUM\_FIRE was 2, one cannot verify Main.main() against  $S_{Main}$  since there is no Ball left at the second invocation of Cannon.fire(). Note that this kind of reasoning would not be possible in Hoare logic because Cannon.fire() has no arguments, so that it would be hard to express any kind of precondition.

With this intuition, in AL, assuming  $S_{\text{Main}}$  and  $S_{\text{Cannon}}$ , we would like to *locally* verify  $I_{\text{Main}}$  and  $I_{\text{Cannon}}$  against, *e.g.*, the abstractions  $A_{\text{Main}}$  and  $A_{\text{Cannon}}$  in Fig. 2, where the abstracted parts are boxed. Note that without any assumption neither  $I_{\text{Main}} \leq_{\text{ctx}} A_{\text{Main}}$  nor  $I_{\text{Cannon}} \leq_{\text{ctx}} A_{\text{Cannon}}$  holds since their context modules are arbitrary and hence, *e.g.*, 0 may be given for r in Main.main(), and also Cannon.fire() may be invoked twice by the context in which case  $I_{\text{Cannon}}$  crashes but  $A_{\text{Cannon}}$ 

runs successfully. As before, to solve this problem, we will *internalize* the SL specifications inside the abstractions, which are the abspecs given in Fig. 2:

```
[S_{\mathsf{Main}} \cup S_{\mathsf{Cannon}} \rtimes (A_{\mathsf{Main}}, \sigma_{\mathsf{Main}}) : S_{\mathsf{Main}}] \text{ and } [S_{\mathsf{Main}} \cup S_{\mathsf{Cannon}} \rtimes (A_{\mathsf{Cannon}}, \sigma_{\mathsf{Cannon}}) : S_{\mathsf{Cannon}}]
```

To understand the abspecs, we first see how the set of resources  $PCM_{Cannon}$  is defined. Concretely,  $PCM_{Cannon}$  consists of five elements Undef, Unit, Ready, Fired, Ball with a commutative binary operator + defined as follows:

- $\forall p \in PCM_{Cannon}$ , Undef + p = Undef
- $\forall p \in PCM_{Cannon}$ , Unit + p = p
- Ready + Ball = Fired
- Fired + Ready = Fired + Ball = Ready + Ready = Fired + Fired = Ball + Ball = Undef

The intuition here is that Undef represents undefinedness (*i.e.*, *inconsistency*); Unit the *empty resource*, which is the identity for +; Ready the *knowledge* that Cannon is not yet fired; Fired that Cannon is already fired; and Ball the *capability* to invoke Cannon.fire(). The definition of + captures that only a subset of {Ready, Ball} or {Fired} is consistent; in particular, there cannot be two (or more) Balls since Ball + Ball = Undef.

Now we look at the abspecs. The dotted boxes are generated from  $S_{Main}$  and  $S_{Cannon}$ , where ASSUME and GUARANTEE are the macros defined at the bottom of Fig. 2. Also the gray code is boilerplate and the highlighted parts are modules' initial resources, which come from  $\sigma_{Main}$  and  $\sigma_{Cannon}$  of the abspecs.

Then we see how it works. Each module has a module-local resource, res\_m, initialized with its initial resource and each function has a function local resource, res\_f, initialized with Unit. Then Cannon.fire() assumes its precondition saying that a resource Ball is given at the beginning, and guarantees its postcondition saying that no resource is returned and the return value r is 1 at the end. Main.main() also assumes its precondition saying that a resource Ball is given at the beginning; then guarantees the precondition of Cannon.fire() before invoking it and assumes the postcondition of Cannon.fire() after the invocation; finally guarantees its postcondition at the end.

Now we see the computational interpretation of ASSUME and GUARANTEE, whose macro definitions are given in Fig. 2. First, ASSUME (Cond) takes a resource; assumes it satisfies Cond; adds it to res\_f; and assumes res\_f is consistent with res\_m. Here the **take** operation is a key technique in AL and will be explained below. For now, we simply consider as if **take**(PCM<sub>Cannon</sub>) magically takes the resource that is given by the caller or returned by the callee. Second, GUARANTEE(Cond) (i) nondeterministically updates res\_m and res\_f via **fpu**, called  $frame-preserving update (FPU)^2$ ; (ii) nondeterministically chooses a resource; (iii) guarantees it satisfies Cond; and (iv) subtract it from res\_f. In (i), we are allowed to update the module and function resources before jumping to another function, which however is restricted to FPUs. The definition of **fpu** is given as follows<sup>3</sup>:

where choose(X) nondeterministically picks an element from the given set X. The intuition is that one can update res\_m and res\_f in such a way that consistency is preserved under an arbitrary frame resource res\_frame (capturing possible resources remaining in other modules). In (ii), although a resource res is just nondeterministically chosen, for now we simply consider as if it is magically passed to the callee or returned to the caller. In (iii), if the chosen res does not satisfy

<sup>&</sup>lt;sup>2</sup>The notion of frame-preserving update comes from modern separation logic such as Iris [Jung et al. 2018].

<sup>&</sup>lt;sup>3</sup>Our formal definition of **fpu** is slightly more general as found in SLs (see Fig. 9 for details).

the condition Cond, it triggers no behavior, which means that only that resource satisfying the condition can be chosen. In (iv), we performed the subtraction because we are passing it to another function, which will add it to its own function resource as we have seen above in ASSUME(Cond). The definition of **minus** is given as follows:

Note that when there is no such res\_f', it triggers no behavior because choosing from the empty set does so.

Now we see how the illusion of resource passing between functions works via **choose** and **take**. First of all, note that it does not make sense to *physically* pass any resource information to context modules or receive it from them because context modules are completely arbitrary and thus may not understand such resource information (*e.g.*, those written in IMP). Our key observation, however, is that by defining **take** as the dual operation to **choose**, we can logically make such an illusion. Specifically, the nondeterministic choice **choose** and its dual **take**<sup>4</sup> are defined as follows for any set X (See §4 for formal definitions).

$$Beh(x := choose(X); K[x]) \stackrel{\text{def}}{=} \bigcup_{x \in X} Beh(K[x])$$

$$Beh(x := take(X); K[x]) \stackrel{\text{def}}{=} \bigcap_{x \in X} Beh(K[x])$$

Note that  $choose(\emptyset)$  is NB and  $take(\emptyset)$  is UB. The intuition is that when proving  $Beh(I) \subseteq Beh(res:= choose(PCM_{Cannon}); K[res])$ , it suffices to prove  $Beh(I) \subseteq Beh(K[res])$  for *some* resource res, which allows us to logically (i.e., in the proof) pick a particular resource to pass to the context. On the other hand, when proving  $Beh(I) \subseteq Beh(res:= take(PCM_{Cannon}); K[res])$ , we need to prove  $Beh(I) \subseteq Beh(K[res])$  for *every* resource res, which makes sense because we do not know what resource will be given from the context, and thus we have to prove the refinement whatever resource is given. Then we have the following theorem, which makes the illusion of passing a resource from **choose** to **take**.

Beh(res := choose(PCM<sub>Cannon</sub>); 
$$K[res]$$
; res' := take(PCM<sub>Cannon</sub>);  $K'[res']$ )  $\subseteq$  Beh(res := choose(PCM<sub>Cannon</sub>);  $K[res]$ ;  $K'[res]$ )

This theorem holds simply by instantiating the **take** operation with the chosen resource res.

With this, we can prove  $I_{Main} \leq_{ctx} [S_{Main} \cup S_{Cannon} \rtimes (A_{Main}, \sigma_{Main}) : S_{Main}]$  for NUM\_FIRE = 1. When Main.main() is invoked, by ASSUME( $\lambda$ res. res == Ball) for any taken resource res, we can assume res == Ball and thus add Ball to res\_f yielding Ball. Then for GUARANTEE( $\lambda$ res. res == Ball) the abspec does not update res\_m, res\_f, chooses Ball for res to satisfy the precondition of Cannon.fire(), and successfully subtracts Ball from res\_f yielding Unit. Then both implementation and abspec invoke Cannon.fire() and receive the same return value r. By ASSUME( $\lambda$ res. res == Unit && r == 1) for any taken resource res, we can assume res == Unit && r == 1, and thus add Unit to res\_f yielding Unit. Since r = 1, both implementation and abspec print 1. Finally, we can trivially satisfy GUARANTEE( $\lambda$ res. res == Unit) by not updating the module and function resources and choosing Unit for res. It is important to note that for NUM\_FIRE = 2 the above proof breaks down since at the second iteration we do not have Ball anymore and thus cannot satisfy GUARANTEE( $\lambda$ res. res == Ball).

Similarly, we can prove  $I_{Cannon} \leq_{ctx} [S_{Main} \cup S_{Cannon} \rtimes (A_{Cannon}, \sigma_{Cannon}) : S_{Cannon}]$ . This time we set the module-local relational invariant to be (powder = 1  $\land$  res\_m = Ready)  $\lor$  (powder = 0  $\land$  res\_m = Fired), which initially holds. Then when Cannon.fire() is invoked, by ASSUME( $\land$ res.

<sup>&</sup>lt;sup>4</sup>The dual to (standard or demonic) nondeterminism is called *angelic nondeterminism* in the literature [Back and Wright 2012; Bodik et al. 2010; Koenig and Shao 2020; Tyrrell et al. 2006].

res == Ball) for any taken resource res, we can assume res == Ball and thus add Ball to res\_f yielding Ball. Now due to the relational invariant, we have two cases. First, when powder =  $0 \land \text{res\_m} = \text{Fired}$ , the refinement trivially holds since  $\text{res\_m} + \text{res\_f} = \text{Undef}$  and thus the abspec triggers UB. Second, when powder =  $1 \land \text{res\_m} = \text{Ready}$ , both implementation and abspec assign 1 to r and print 1; the implementation decrements powder by 1 yielding 0 while for GUARANTEE( $\lambda$ res. res == Unit && r == 1) the abspec updates (res\_m, res\_f) to (Fired, Unit), which is frame-preserving, and chooses Unit for res to satisfy the postcondition and successfully subtract Unit from res\_f yielding Unit; finally both return the same value 1 and establish the relational invariant with powder =  $0 \land \text{res\_m} = \text{Fired}$ . It is important to note that invariants relating implementations and abspecs like (powder =  $1 \land \text{res\_m} = \text{Ready}$ )  $\lor$  (powder =  $0 \land \text{res\_m} = \text{Fired}$ ) above do not appear in the abspecs, but only as a part of simulation proofs.

Finally, we can compose the abspecs to erase the specification parts:

```
\operatorname{Beh}([S_{\operatorname{Main}} \cup S_{\operatorname{Cannon}} \rtimes (A_{\operatorname{Main}}, \sigma_{\operatorname{Main}}) : S_{\operatorname{Main}}] \circ [S_{\operatorname{Main}} \cup S_{\operatorname{Cannon}} \rtimes (A_{\operatorname{Cannon}}, \sigma_{\operatorname{Cannon}}) : S_{\operatorname{Cannon}}])
\subseteq \operatorname{Beh}(A_{\operatorname{Main}} \circ A_{\operatorname{Cannon}})
```

We can prove this theorem, called *spec erasure theorem*, as follows. To *discharge* the initial ASSUME of Main.main() (*i.e.*, to replace it with **skip**), we just need to choose an initial resource  $\sigma_{main}$  to Main.main(), which will be Ball here, and show that (*i*) it is consistent with the initial module-local resources (*i.e.*,  $\sigma_{main} + \sigma_{Main} + \sigma_{Cannon}$ != Undef) and (*ii*) it satisfies the precondition of Main.main() (*i.e.*,  $\sigma_{main}$  == Ball). Then each remaining ASSUME(Cond) with a predicate Cond on PCM<sub>Cannon</sub> is discharged by the immediately preceding GUARANTEE(Cond) with the same predicate. To prove this, we first show that consistency of the whole resources (*i.e.*, those stored in all res\_m and res\_f) is invariant: the consistency holds initially because we have shown it by (*i*) above, and is preserved by GUARANTEE(Cond); ASSUME(Cond) because the frame-preserving update via **fpu** preserves it by definition and, by ( $\star$ ) above, the subtracted resource res in GUARANTEE is immediately added back in ASSUME. Then we can complete the proof by discharging all the assumptions in ASSUME, again by ( $\star$ ): **assume**(Cond(res)) is discharged by **guarantee**(Cond(res)), and **assume**(res\_m + res\_f! = Undef) by the above invariant (*i.e.*, consistency of the whole resources). Note that the spec erasure theorem holds in general among any compatible abspecs: for example, even when NUM\_FIRE = 2, the above erasure holds.

To conclude, the most important idea in AL is to give an illusion of passing logical information via **choose** and **take** in an operational way. In the next section, we will see how this powerful mechanism can be used to model various logical features seamlessly.

#### 3 ADVANCED FEATURES AND EXAMPLES OF ABSTRACTION LOGIC

In this section, we will demonstrate a general version of abstraction logic with five advanced features: (i) dealing with unknown contexts, (ii) proving and exploiting purity (i.e., absence of side effects), (iii) decomposing and reusing verification tasks via gradual abstraction, (iv) abstracting function arguments and return values, and (v) reasoning about function pointers. We will introduce them by walking through various examples.

# 3.1 Dealing with unknown contexts

Unlike in the previous section where we composed abspecs for the *whole* modules, here we will see how to compose those for only selected modules, called *friends*, while still allowing them to interact with arbitrary context modules, called *contexts* (*i.e.*, establishing contextual refinement as a result).

```
I_{\mathsf{Mem}} := [\mathsf{Module} \ \mathsf{Mem}]
                                                                                                     A_{\mathsf{Mem}} := [\mathsf{Module} \ \mathsf{Mem}]
                                                                                                                     local mem := ...
              local mem := ...
              def alloc([n: int64]?) \equiv ...
                                                                                                                     def alloc =
              def free([p: ptr]?) \equiv ...
                                                                                                                       friend(_) ≡ TRIVIAL
              def load([p: ptr]?) \equiv ...
                                                                                                                       context([n: int64]?) \equiv ...
              def store([p: ptr, v: val]?) 	≡ ...
\sigma_{\mathsf{Mem}} := \bullet \varepsilon \in Auth(\mathsf{ptr} \to Ex(\mathsf{val})) \subseteq \Sigma
S_{\mathsf{Mem}} := \{\mathsf{Mem.alloc} : \forall n : \mathsf{int64}.
                                                                           \{\lambda x - d \cdot \lceil d = \text{Some } - \land x = \uparrow \lceil n \rceil \land n \ge 0 \rceil \}
                                                                           \{\lambda r \_ \exists p : \mathsf{ptr}, \ell : \mathsf{list} \ \mathsf{val}. \ \lceil r = \uparrow p \land \mathsf{length}(\ell) = n \rceil * (p \mapsto \ell)\},\
                                                                           \{\lambda x - d. \exists p : \mathsf{ptr}, v : \mathsf{val}. \lceil d = \mathsf{Some} - \land x = \uparrow [p] \rceil * (p \mapsto [v])\}
                Mem.free: \forall_{-}:().
                                                                           \{\lambda r \_ . \lceil r \in \uparrow val \rceil\},
                Mem.load: \forall (p,v) : \mathsf{ptr} \times \mathsf{val}. \{ \lambda x - d. \ \lceil d = \mathsf{Some} \ \_ \land x = \uparrow \lceil p \rceil \rceil * (p \mapsto \lceil v \rceil) \}
                                                                           \{\lambda r \_ . \lceil r = \uparrow v \rceil * (p \mapsto [v])\},
                \mathsf{Mem.store} \colon \forall (p,v) : \mathsf{ptr} \times \mathsf{val}. \; \{\lambda \, x \, \_d. \; \lceil d = \mathsf{Some} \, \_ \, \land \, x = \uparrow [p,v] \, \rceil \, * \, \exists v' : \mathsf{val}. \; p \mapsto [v'] \}
                                                                           \{\lambda r \_ . \lceil r \in \uparrow \text{val} \rceil * (p \mapsto [v])\}
                                                                                                                                                                                                        }
```

Fig. 3. An implementation and its abspec for the module Mem.

For this, our first observation is that there is a specification that every module satisfies. Specifically, one can easily see the following holds: for any module *C* in EMS,

$$C \leq_{\operatorname{ctx}} [S_* \rtimes (C, \varepsilon) : S_*]$$

where  $\varepsilon$  is the identity for a PCM in consideration and  $S_*$  has  $s_* = \{\text{True}\}\ \{\text{True}\}\$ for every function f in the scope (*i.e.*, the first  $S_*$  above specifies all the functions invoked by C while the second  $S_*$  all the functions defined by C). The essential reason why *every* module can satisfy the above CR is because specifications in AL do not require *safety*, unlike in SL.

Although we can give specifications to arbitrary contexts as above, we still have to address the problem that the specifications for contexts are not compatible with those for friends. Specifically, the contexts assume  $S_*$  for each module F among the friends while its specification  $S_F$  may not be  $S_*$ . We solve this problem by allowing abspecs to provide two different behaviors: those satisfying intended specifications when invoked by the friends; and those satisfying  $S_*$  when invoked by the contexts. To enable this, we add a special mechanism to EMS that allows a callee to get the caller's module name, so that the callee can behave differently depending on who's the caller.

As an example, consider the module Mem given in Fig. 3. The implementation  $I_{\text{Mem}}$  is directly written in EMS, which provides a CompCert-like memory model for the IMP language. Here, to see what, e.g., [p: ptr, v: val]? in store means, note that in EMS every function takes a value of type Any, which can be seen as the set of all mathematical values, and also returns an Any value, while the IMP language supports values of type val<sup>5</sup> consisting of 64-bit integers (int64) and function and memory pointers (ptr). Therefore, when defining the semantics of IMP in EMS, we pack arguments to a function into a single value of type list val and upcast it into Any, and conversely downcast and unpack a given Any argument to a list of values of expected types, where we trigger UB if the downcast or the unpacking fails. The notation [p: ptr, v: val]? denotes such downcast and unpacking of an Any argument into a list of two values of types ptr and val. Now we see what  $I_{\text{Mem}}$  does. alloc(n) allocates a memory block consisting of n cells, each of which can store a value of type val, and returns the pointer pointing to the beginning of the block; free(p) deallocates the cell pointed to by p; load(p) reads a value from the cell pointed to by p and returns it; and store(p, v) stores the value v in the cell pointed to by p.

<sup>&</sup>lt;sup>5</sup>To support a compilation to CompCert, val also contains the special value **undef**.

Now we see how the abspec for Mem, given in Fig. 3, is defined, where each function consists of two definitions, **friend** and **context**. Concretely, the **friend** definitions are all TRIVIAL, which can be simply understood as **skip** for now (see §3.2 for details), while the **context** definitions are identical to their implementations. The intention is that the former defines its behaviors when invoked by friends and the latter by contexts. Although we do not know which modules will be friends yet, we can still locally verify the following CR for any specification S with  $\sigma_{\text{Mem}}$ ,  $S_{\text{Mem}}$  given in Fig. 3,

$$I_{\text{Mem}} \leq_{\text{ctx}} [S \rtimes (A_{\text{Mem}}, \sigma_{\text{Mem}}) : S_{\text{Mem}}]$$

Here the EMS semantics of each function f in  $[S \rtimes (A_{Mem}, \sigma_{Mem}) : S_{Mem}]$  is given by *intersecting* the semantics of **friend** and that of **context**. Specifically, we can intersect them by:

var b = take(bool); if (b) then 
$$C_{friend}$$
 else  $C_{context}$ 

where the semantics  $C_{\texttt{friend}}$  for f is generated from the friend definition of f in  $A_{\texttt{Mem}}$  together with its specification  $S \rtimes S_{\texttt{Mem}}$  (and  $\sigma_{\texttt{Mem}}$ ) in the way we have seen in the previous section; similarly for  $C_{\texttt{context}}$  but with the context definition and  $S \rtimes S_*$ . Such intersection makes sense because it is essential to establish refinement between  $I_{\texttt{Mem}}$  and  $A_{\texttt{Mem}}$  for both friends and contexts. It is important to note that the specification  $S \rtimes S_*$  for  $C_{\texttt{context}}$  means that when we prove  $C_{\texttt{context}}$  satisfies  $S_*$ , we can still rely on the intended assumptions S about friends when it invokes their functions. We will see such examples in §3.2 and §3.4.

For such abspecs with **friend** and **context**, we have a general spec erasure theorem yielding contextual refinement (*i.e.*, under arbitrary contexts).

Theorem 3.1. Given a global PCM  $\Sigma$  (including all PCMs of interest) and abspecs  $[S \rtimes (A_i, \sigma_i) : S_i]$  w.r.t.  $\Sigma$  for  $i \in \{1, \ldots, n\}$  with any  $S \supseteq S_1 \cup \ldots \cup S_n$ , suppose that their module names (i.e., friends) are  $N = \{\text{name}_1, \ldots, \text{name}_n\}$ , and for any argument value v to Main.main, there is an initial resource  $\sigma$  to Main.main such that  $\sigma + \sigma_1 + \ldots + \sigma_n = 0$ . Indeed, if Main.main is among the friends,  $(v, \sigma)$  satisfies its precondition. Then we have the following:

$$[S \times (A_1, \sigma_1) : S_1] \circ \ldots \circ [S \times (A_n, \sigma_n) : S_n] \leq_{ctr} [A_1]_N \circ \ldots \circ [A_n]_N$$

Here we can understand  $S \supseteq S'$  as  $S \supseteq S'$  though it has a slightly more general definition (see §4 for details). Also the semantics of a function f in  $[A_i]_N$  is defined by combining its **friend** semantics  $C_{\texttt{friend}}$  and its **context** semantics  $C_{\texttt{context}}$  in  $A_i$  as follows:

if 
$$(get\_caller() \in N)$$
 then  $C_{friend}$  else  $C_{context}$ 

where  $get\_caller()$  is supported by EMS and returns the module name of the caller. Also we henceforth call such  $A_i$  pre-abstraction and such  $[A_i]_N$  abstraction. Note that since the theorem establishes contextual refinement, the contexts are completely unrestricted (e.g., they may be unsafe, make system calls, or make mutually recursive calls to the friends).

Now we discuss the details of  $S_{\text{Mem}}$ . Here we ignore the measure parameter d (used to prove termination) and  $\_$  in  $S_{\text{Mem}}$ , which will be discussed in §3.2 and §3.4. First,  $\uparrow$  is the *upcast* operator (*i.e.*,  $\uparrow x$  is a value of type Any for x of any type X) and  $\uparrow$ val is  $\{\uparrow v \mid v \in \text{val}\}$ . The parameters x and r are bound to argument and return values. Then  $S_{\text{Mem}}$  is a standard specification for such memory operations that one would write in modern separation logics such as Iris [Jung et al. 2018]. Specifically, the pre and post conditions define predicates on resources (*i.e.*,  $\Sigma \to \text{Prop}$ , which we call rProp) when values for the quantifiers and argument are given; and the separating conjunction \*, magic wand -\*, lifting  $\ulcorner \urcorner$  of rProp to rProp, and existential and universal quantifiers for rProp are defined in the standard way. Also, the rPCM rAuth ( $r\text{ptr} \to \text{Ex}(\text{val})$ ) is a standard authoritative rPCM and the points-to predicate rprop is rprop.

beginning of consecutive cells that contain the values in  $\ell$ , is derived in the standard way satisfying the following law:

$$p \mapsto v :: \ell \iff (p \mapsto [v]) * (p + 8 \mapsto \ell)$$

Note that in AL, universal quantifiers such as  $\forall (p,v): \mathsf{ptr} \times \mathsf{val}$  above are also modeled via **choose** and **take**. The reason is because values for those quantifiers are essentially determined by the callers, and therefore the caller *chooses* a value for the quantifier and the callee *takes* the value as we have seen in the previous section.

Now we briefly discuss how to verify  $I_{Mem}$ : for any specification S,

$$I_{\text{Mem}} \leq_{\text{ctx}} \left[ S \rtimes (A_{\text{Mem}}, \sigma_{\text{Mem}}) : S_{\text{Mem}} \right]$$
 (1)

As usual, we first set up a module-local relational invariant and prove refinement between  $I_{Mem}$ and  $[S \rtimes (A_{Mem}, \sigma_{Mem}) : S_{Mem}]$ , which is split into two cases because the abspec is defined as the intersection of friend and context: proving (i) that  $I_{Mem}$  refines the friend definitions of  $A_{Mem}$ under  $S \rtimes S_{Mem}$  and (ii) that  $I_{Mem}$  refines the **context** definitions of  $A_{Mem}$  under  $S \rtimes S_*$ , where both proofs involve preservation of the (common) relational invariant, which essentially captures that the **friend** and **context** definitions work in harmony (i.e., they do not interfere each other's reasoning in any interleaved invocations of the two definitions). Specifically, the invariant says that the blocks allocated in the implementation are split into two groups such that in the abspec, one of the groups is allocated at the same addresses<sup>6</sup> (performed by **context**) and the other resides in the module-local resource (but not in the memory) in terms of the points-to predicate (performed by **friend**). The reason why this invariant is preserved even when the **context** definitions are invoked with arbitrary pointers is essentially because when a context tries to access the blocks allocated by friends by forging their addresses, the invariant guarantees that in the abspec no blocks are allocated at those addresses so that such accesses always trigger UB, which immediately completes the refinement. Except the preservation of the invariant, the proof of (ii) is straightforward because it establishes refinement between identical definitions; and that of (i) essentially amounts to a standard SL proof for those specifications together with a termination proof, which will be discussed in the following section.

# 3.2 Proving and exploiting purity

Now we discuss how to express, prove and exploit *purity* of an abspec of a function. Note that it is possible that even though an implementation has impurity, if the impurity only changes the module's local state, its abspec can be made pure by migrating the impurity to its specification (*i.e.*, pre and post conditions). Indeed this is the case for the module Mem and we will see how we can do it in this section.

To see this clearly, we need another example, which is the module Stack given in Fig. 4 and implemented using the module Mem. Concretely,  $I_{Stack}$  presents the EMS semantics of an IMP program<sup>7</sup> that implements stacks using linked lists, where new() creates a new stack; push(stk, v) pushes the value v into the stack stk; and pop(stk) pops a value from stk and returns it if stk is nonempty; otherwise returns 0.

Then  $A^1_{\text{Stack}}$  presents an abstract version of the functions, which *module-locally* manage a pool of mathematical lists using ptr values as their handles (defined as ptr  $\rightarrow$  option (list val)). Concretely, new() nondeterministically chooses an *unused* handle (via **choose** and **guarantee**), registers the empty list with the handle in the pool and returns the handle; push(handle, v) gets the list with handle from the pool (if fails, trigger **UB**) by unopt?(pool handle) and then updates

<sup>&</sup>lt;sup>6</sup>This is possible because our allocator is nondeterministic.

<sup>&</sup>lt;sup>7</sup>The downcast to and upcast from val (with **UB** in case of failure) around a function call are omitted for syntactic clarity.

```
A^1_{\mathsf{Stack}} \coloneqq [\mathbf{Module} \ \mathsf{Stack}]
I_{\mathsf{Stack}} := [\mathsf{Module} \ \mathsf{Stack}]
         def new([]?) \equiv
                                                            local pool := (\lambda_{-}. None)
            var stk := Mem.alloc(1);
                                                               : ptr \rightarrow option (list val)
            Mem.store(stk, NULL);
                                                            def new =
                                                              friend, context([]?) ≡
                                                               var handle := choose(ptr);
         def push([stk: val, v: val]?) ≡
            var node := Mem.alloc(2);
                                                               guarantee(pool handle == None);
                                                              pool := pool[handle := Some []];
            var hd := Mem.load(stk);
            Mem.store(node, v);
                                                               handle
            Mem.store(node+8, hd);
                                                            def push =
                                                             friend,context([handle: ptr, v: val]?) =
            Mem.store(stk,node)
                                                               var stk := unopt?(pool handle);
         def pop([stk: val]?) ≡
            var hd := Mem.load(stk);
                                                               pool := pool[handle := Some (x::stk)]
            if hd == NULL then 0
                                                            def pop =
            else
                                                              friend, context([handle: ptr]?) =
               var v := Mem.load(hd);
                                                               var stk := unopt?(pool handle);
               var next := Mem.load(hd+8);
                                                               match stk with | [] => 0
               Mem.store(stk, next);
                                                               | v :: stk' =>
               Mem.free(hd); Mem.free(hd+8);
                                                                 pool := pool[handle := Some stk'];
S^1_{\texttt{Stack}} := \{ \; \texttt{Stack.new} : s_*, \; \texttt{Stack.push} : s_*, \; \texttt{Stack.pop} : s_* \; \}
                                                                                         \sigma^1_{\text{Stack}} := \varepsilon
where s_* = \forall_- : (). \{\lambda x x_a d. \lceil d = \text{None } \land x = x_a \rceil \} \{\lambda r r_a. \lceil r = r_a \rceil \}
```

Fig. 4. An implementation and its first abspec for the module Stack.

the list by adding the value v; and pop(handle) also gets the list with handle by unopt?(pool handle), and if it is empty, returns  $\emptyset$ ; otherwise returns the head of the list after eliminating it from the list. Note that in the pre-abstraction, the **friend** and **context** definitions coincide and so do their specifications (*i.e.*, both are  $s_*$ , the details of which will be explained in §3.4).

First of all, we start by noting the problem that it actually does not make sense to prove CR between the implementation and the abspec for Stack because there are several calls to Mem in  $I_{\text{Stack}}$  while there are no such calls in  $A^1_{\text{Stack}}$ . Even though Mem is specified in the abspec, when locally establishing CR for Stack, Mem is still arbitrary and thus can make arbitrary side effects such as fatal errors or making system calls. Therefore completely abstracting away those calls in the pre-abstraction would not allow us to establish the desired CR.

Our approach to address this problem is three-fold: (i) giving a way of enforcing and specifying purity (i.e., the absence of side effects), (ii) giving an illusion of abstracting away those calls to functions specified as pure, simply called pure calls, instead of actually eliminating them in the abspec, and then (iii) truly eliminating them in the abstractions after applying the spec erasure theorem.

We discuss (ii) first. To give an illusion of eliminating pure calls in local reasoning as seen in  $A^1_{\mathsf{Stack}}$ , we introduce a mechanism, called *implicit pure calls (IPC)*. The idea is to implicitly introduce all possible pure calls (according to the given specification) since they will be eliminated by the spec erasure theorem anyway. Specifically, an IPC nondeterministically makes an unbounded but finite number of all possible pure calls with all possible (physical and logical) arguments and we implicitly insert an IPC at each line (e.g., the semicolon in  $A^1_{\mathsf{Stack}}$  is interpreted as making an IPC). Therefore without explicitly writing down specific pure calls, we can freely rely on the specification of any pure function during the proof.

Now we discuss (i): how to enforce and specify purity in AL. The notion of purity of a function f, saying that f does not produce any side effects, can be mostly enforced by defining the preabstraction of f as an IPC. Since IPC only allows invoking pure functions, verifying against IPC essentially amounts to proving the absence of side effects except for non-termination. Therefore, the remaining questions are how to enforce termination, how to specify purity and how to make a pure call.

We can answer all the question by adding the *measure parameter d* to preconditions, which can be passed from a caller to the callee via **choose** and **take**. First, we define the type of measures to be option ord with ord the set of ordinals<sup>8</sup> with a well-founded order < and define a relation  $\Box$  between them by the following two cases:

Some 
$$o \sqsubseteq \text{Some } o' \text{ with } o < o' \in \text{ord}$$
  $d \sqsubseteq \text{None with } d \in \text{option ord}$ 

Second, invoking a function with a measure Some  $\_$  is considered as a pure call and the EMS semantics of a function in an abspec, when invoked with Some  $\_$ , is defined to be an IPC. Third, to enforce termination, when a function f is invoked with a measure d, for each function call made inside the invocation we add the **guarantee** that it should pass (*i.e.*, **choose**) a measure d' with  $d' \sqsubseteq d$ . Then it is guaranteed that a pure call (*i.e.*, with Some o) can only make pure calls as sub calls (thereby producing no side effects other than non-termination) and should terminate because any measure  $d \sqsubseteq$  Some o should be Some o' with o' < o. Note that an impure call (*i.e.*, with the measure None) can still make any calls because we have  $d \sqsubseteq$  None for any measure d including None.

Then, we achieve (iii): the spec erasure theorem can soundly eliminate all IPCs in the resulting abstractions. Also, note that the **friend** definition of f in an abspec only describe the impure behavior of f since its pure behavior is defined to be an IPC.

With this in mind, we revisit the module Mem. First, all the specifications have  $d = \text{Some}_{-}$  in their preconditions, which implies that only pure calls to those functions can be made. Second, since the functions in Mem do not allow any impure calls to them, their **friend** definitions are never executed, which we can guarantee by defining them as **NB** (*i.e.*, TRIVIAL= **NB**). Then, in the abstraction  $[A_{\text{Mem}}]_N$  for Mem after applying the spec erasure theorem with friends N, we have the functions that trigger **NB** if invoked by the friends, and do the original jobs otherwise. This implies that the friends are guaranteed not to invoke any memory operations (*i.e.*, all the calls to Mem by the friends are eliminated in their abstractions). Note that using the fact that **NB** contextually refines any possible definitions, we can also revert the abstraction  $[A_{\text{Mem}}]_N$  back to  $I_{\text{Mem}}$  (*i.e.*,  $[A_{\text{Mem}}]_N \leq_{\text{ctx}} I_{\text{Mem}}$ ). Also note that as discussed above, the implementation  $I_{\text{Mem}}$  can be seen as impure because, *e.g.*, calls to **store** have impact on subsequent calls to **load**, while its pre-abstraction, IPC, for friends are indeed pure and thus can be eliminated.

Finally, we can verify  $I_{\text{Stack}}$ : for any  $S \supseteq S_{\text{Mem}}$ ,

$$I_{\mathsf{Stack}} \leq_{\mathsf{ctx}} \left[ S \rtimes (A^1_{\mathsf{Stack}}, \sigma^1_{\mathsf{Stack}}) : S^1_{\mathsf{Stack}} \right]$$
 (2)

For this, we set up the module-local invariant saying that the lists in the pool are matched with the linked lists stored in the module-local resource of Stack in terms of the points-to predicate, which are obtained via IPCs to the memory functions with  $S_{Mem}$ . Then, one can easily establish a simulation proof preserving the invariant.

# 3.3 Decomposing and reusing verification tasks via gradual abstraction

Although the abstraction  $[A^1_{\text{Stack}}]_{\text{Mem,Stack}}$ , obtained by applying the spec erasure theorem for the friends Mem and Stack, can be directly used by other modules, it would be better to provide useful logical specifications for Stack like those we provided for Mem.

<sup>&</sup>lt;sup>8</sup>We developed our own Coq library for ordinals, which will be published elsewhere.

```
A_{\rm Stack}^2 \coloneqq [ \mbox{Module Stack} ] \\ \mbox{local pool} \ := (\lambda_-. \ \mbox{None})
                                                                                                             def new =
                                                                                                                friend(_{-}) \equiv NB
                             : ptr \rightarrow option (list val)
                                                                                                               context([]?) \equiv ...
\sigma_{\mathsf{Stack}}^{2\mathsf{A}} := \bullet \varepsilon \in Auth(\mathsf{ptr} \to Ex(\mathsf{list}\ \mathsf{val})) \subseteq \Sigma
S_{\text{Stack}}^{2A} := \{ \text{Stack.new: } \forall_- : ().
                                                                                                                          \{\lambda x - d \cdot \lceil d = \text{Some } - \land x = \uparrow \lceil \rceil \rceil \}
                                                                                                                          \{\lambda r \perp \exists h : \mathsf{ptr.} \ \lceil r = \uparrow h \rceil * \mathsf{is\_stk} \ h \ []\},
                       Stack.push: \forall (h, v, \ell) : ptr \times val \times list val. \{ \lambda x - d. \lceil d = Some - \land x = \uparrow [h, v] \rceil * is_stk h \ell \}
                                                                                                                          \{\lambda r \_ . \ \lceil r \in \uparrow \text{val} \ \rceil * \text{is\_stk } h \ (v :: \ell)\},
                       Stack.pop: \forall (h, \ell) : ptr \times list val.
                                                                                                                         \{\lambda x - d. \lceil d = \text{Some } - \land x = \uparrow \lceil h \rceil \rceil * \text{is\_stk } h \ell \}
                                                                                                                          \{\lambda r ... \Gamma r = \uparrow \text{head}(\ell, 0) \rceil * \text{is\_stk } h \text{ tail}(\ell)\}
                                                                                                                                                                                                                                 }
\sigma_{\mathsf{Stack}}^{2B} \coloneqq \bullet \varepsilon \in \operatorname{Auth}\left(\mathsf{ptr} \to \operatorname{Option}\left(\operatorname{Ag}\left(\mathcal{P}(\mathsf{val})\right)\right)\right) \subseteq \Sigma
S_{\mathsf{Stack}}^{2\mathsf{B}} := \{\mathsf{Stack.new:} \ \forall P: \mathcal{P}(\mathsf{val}).
                                                                                                        \{\lambda x - d \cdot \lceil d = \text{Some } - \land x = \uparrow \lceil \rceil \rceil \}
                                                                                                         \{\lambda r \perp \exists h : \mathsf{ptr.} \ \lceil r = \uparrow h \rceil * \mathsf{is\_bag} \ h \ P\},\
                       Stack.push: \forall (h, v, P) : ptr \times val . \{\lambda x \_ d . \lceil d = Some \_ \land x = \uparrow \lceil h, v \rceil \land v \in P \rceil * is\_bag h P \}
                                                                                                     \{\lambda r \_ . \ \lceil r \in \uparrow \text{val} \ \rceil * \text{ is\_bag } h P\},
                                                                             \times \mathcal{P}(\text{val})
                       Stack.pop: \forall (h, P) : ptr \times \mathcal{P}(val). \{ \lambda x - d. \ ^{r}d = Some \ _{\land} x = \uparrow [h] \ ^{r} * is\_bag \ h \ P \}
                                                                                                         \{\lambda r \perp \exists v : \text{val. } \lceil r = \uparrow v \land (v = 0 \lor v \in P) \rceil * \text{is\_bag } h P\} \}
```

Fig. 5. Two abspecs on top of the first abstraction for the module Stack

Fig. 5 shows two such specifications  $S_{\sf Stack}^{2A}$  and  $S_{\sf Stack}^{2B}$  for Stack. Then we can separately verify the previous abstraction against the two specifications together with the *pure* pre-abstraction  $A_{\sf Stack}^2$ . Specifically, we prove the following: for any specification S,

$$[A_{\mathsf{Stack}}^1]_{\mathsf{Mem,Stack}} \leq_{\mathsf{ctx}} [S \rtimes (A_{\mathsf{Stack}}^2, \sigma_{\mathsf{Stack}}^{2A}) : S_{\mathsf{Stack}}^{2A}] \tag{3}$$

$$[A_{\mathsf{Stack}}^1]_{\{\mathsf{Mem},\mathsf{Stack}\}} \leq_{\mathsf{ctx}} [S \bowtie (A_{\mathsf{Stack}}^2,\sigma_{\mathsf{Stack}}^{2B}):S_{\mathsf{Stack}}^{2B}] \tag{4}$$

Here both  $S_{\rm Stack}^{2A}$  and  $S_{\rm Stack}^{2B}$  only allow pure calls to the functions by requiring  $d={\rm Some}$  and the pre-abstraction  $A_{\rm Stack}^2$  is the same as  $A_{\rm Stack}^1$  for **context**, and **NB** for **friend**, as we have done for Mem. The two specifications provide different benefits to the client: the former precisely tracks the contents of a stack via the predicate is\_stk h  $\ell$  saying that the stack with handle h in the pool coincides with  $\ell$ , while the latter maintains a certain property for a stack via the predicate is\_bag h P saying that all the elements in the stack with handle h satisfy the property P and furthermore allows duplicating the resource thereby permitting multiple modules to update the stack at the same time as long as the pushed elements satisfy P. The proof structures for these two verifications are similar to that for the verification for Mem: is\_stk and is\_bag are defined similarly as the points-to predicate using standard authoritative PCMs; the module-local relational invariant for the former says each list in the pool is matched with the corresponding list stored in the module-local resource in terms of is\_stk; and that for the latter says all the elements of each list in the pool satisfy the corresponding property stored in the module-local resource in terms of is\_bag.

Benefits of such gradual abstraction for Stack are two-fold. First, we can achieve separation of concerns via gradual abstraction. For example, in the first abstraction for Stack, the verification focused on abstracting linked lists into mathematical lists without thinking about providing useful specifications to the client, while in the second abstractions, the verifications focused on providing such specifications based on mathematical lists. Second, gradual abstraction increases reusability of verification results. For example, for Stack, we essentially reused the first verification result turning linked lists into mathematical lists, in the two verifications providing different specifications to the client. Also note that in final abstractions obtained by applying the spec erasure theorem either with the is\_stk specification or with the is\_bag specification, the result of the first abstraction,

```
I_{\mathsf{Echo}} :=
[Module Echo]
                                               def input([stk: val]?) ≡
                                                                                                   def output([stk: val]?) ≡
def echo([]?) \equiv
                                                  var v := IO.getint();
                                                                                                      var v := Stack.pop(stk);
   var stk := Stack.new();
                                                  if (v == 0) then 0
                                                                                                      if (v == 0) then 0
                                                                                                      else { IO.putint(v);
                                                  else { Stack.push(stk, v);
   Echo.input(stk);
   Echo.output(stk)
                                                              Echo.input(stk) }
                                                                                                                 Echo.output(stk) }
A_{\mathsf{Echo}} :=
[Module Echo]
                                     def input =
                                                                                         def output =
def echo =
                                       friend(stk:! list int64) =
                                                                                          friend(stk:! list int64) =
 friend,context([]?) =
                                         var v:? int64 := IO.getint(); match stk with | [] => ()
   var stk:! list int64
                                         if (v == 0) then stk
                                                                                            | hd::tl => IO.putint(hd);
                                                                                                               Echo.output(tl) end
       := Echo.input([]);
                                         else Echo.input(v::stk)
   Echo.output(stk)
                                       context(_{-}) \equiv UB
                                                                                          context(_{-}) \equiv UB
\sigma_{\mathsf{Echo}} \coloneqq \varepsilon
S_{\mathsf{Echo}} := \{\mathsf{Echo.echo:} \ \ s_*,
            Echo.input: \forall h:ptr. \{\lambda x x_a d. \exists \ell : \text{list int64.} \ \lceil d = \text{None} \land x = \uparrow [h] \land x_a = \uparrow \ell \rceil * \text{is\_estk } h \ell \}
                                            \{\lambda r r_a. \exists \ell : \text{list int64.} \ \lceil r \in \uparrow \text{val} \land r_a = \uparrow \ell \rceil * \text{is\_estk } h \ell \},
             Echo.output:\forall h:ptr. \{\lambda x \, x_a \, d. \, \exists \ell : \text{list int64.} \, \ulcorner d = \text{None } \land \, x = \uparrow \llbracket h \rrbracket \, \land \, x_a = \uparrow \ell \urcorner * \text{is\_estk } h \, \ell \}
                                           \{\lambda r r_a. \exists \ell : \text{list int64.} \ \lceil r \in \uparrow \text{val} \land r_a = \uparrow \ell \rceil * \text{is\_estk } h \ell \}
where is_estk h \ell = \lceil \text{nonzero}(\ell) \rceil * \text{is\_stk } h \ell
```

Fig. 6. An implementation and its abspec for the module Echo.

 $A^1_{Stack}$ , is reused to provide the **context** behaviors (*i.e.*, those arising when invoked by contexts), which is based on mathematical lists instead of linked lists.

# 3.4 Abstracting function arguments and return values

Since friends are enforced to respect each other's specification by the spec erasure theorem, it would make sense to abstract even function arguments and return values among the friends. Indeed, in abstraction logic, such abstraction can be easily achieved.

To see this, we consider the example given in Fig. 6. In the implementation  $I_{\mathsf{Echo}}^{\,\,\,\,\,}$ , Echo. echo() creates a new stack, stk; invokes Echo. input(stk), which repeatedly gets an integer via I0. getint and pushes it into stk until getting 0; and then invokes Echo. output(stk), which repeatedly pops an integer from stk and outputs it via I0. putint until the stack is empty (i.e., 0 is returned). The pre-abstraction  $A_{\mathsf{Echo}}$  directly uses mathematical lists instead of using the module Stack. Concretely, the **friend** definitions of input and output take a mathematical list as an argument, instead of a val value. The argument (stk:! list int64) denotes that the Any value given as an argument is downcast to list int64 and if it fails, triggers NB, which guarantees that every friend invokes them with a mathematical list. On the other hand, we will treat the I0 module as an unrestricted context (i.e., specified as  $S_*$ ) and thus trigger UB if the downcast from an Any value given by I0. getint() to int64 fails, which is denoted by var v:? int64 := I0.getint().

It is important to note that the **context** definitions of input and output immediately trigger **UB**, which specifies that contexts are *not allowed* to invoke them because they are intended to be such *internal* functions whose arbitrary invocation may interfere the behavior of the module Echo. On the other hand, echo has the same definition for **friend** and **context** with  $s_*$ , which specifies that it can be freely invoked by both friends and contexts since it does not interfere the behavior of Echo.

 $<sup>^{9}</sup>$ It shows the semantics of an IMP program, omitting the downcast and upcast around function calls.

Now we see how we support the abstraction of argument and return values of input and output from val to list int64. For this, AL adds an extra argument  $x_a$ , called *abstract argument*, and an extra return value  $r_a$ , called *abstract return value*, to the specifications. Then we can specify relationship between concrete ones and abstract ones in the pre and post conditions. For example, the specifications of input and output say that the concrete argument x contains a handle h, the abstract one  $x_a$  a list  $\ell$ , and they satisfy is\_estk h  $\ell$  (defined as  $\lceil \text{nonzero}(\ell) \rceil * \text{is_stk } h$   $\ell$ ) stating that  $\ell$  only has non-zero elements and h is a handle for a stack containing  $\ell$ ; and similarly for concrete and abstract return values.

Then we can define the abspec semantics to give an illusion of passing abstract values as follows. When we make a call to f with an abstract value (e.g., Echo. input(v::stk)), we choose a concrete value, guarantee that they are related as specified in the precondition of f, and pass the concrete value as a real argument. Conversely, in the **friend** definition of f, we first receive a concrete value as a real argument, and then take an abstract value, assume that they are related as specified in the precondition of f and pass the abstract value as an argument to the **friend** definition. The abspec semantics also does similarly for return values. From these constructions, it follows that the spec erasure theorem can soundly eliminate concrete values and directly pass abstract ones in the final abstractions.

For verification of Echo, we use  $S_{\sf Stack}^{\sf 2A}$  as a specification for the module Stack and prove

$$I_{\mathsf{Echo}} \leq_{\mathsf{ctx}} \left[ S \times (A_{\mathsf{Echo}}, \sigma_{\mathsf{Echo}}) : S_{\mathsf{Echo}} \right]$$
 (5)

for any  $S \supseteq S_{\sf Echo} \cup S^{\sf 2A}_{\sf Stack} \cup S_*$  with  $S_*$  for the module I0. The proof is quite straightforward: the local simulation argument directly follows from the precondition of each function without using any module-local invariant or resource. Note that this proof essentially involves cyclic reasoning since Echo. input and Echo. output make recursive calls; however, it does not require any special treatment.

Now we compose all the verification results so far as follows. By applying the spec erasure theorem to Equations (1) and (2), and to Equations (3) and (5), we have

$$\begin{array}{lll} I_{\mathsf{Mem}} \circ I_{\mathsf{Stack}} & \leq_{\mathsf{ctx}} & [A_{\mathsf{Mem}}]_{\{\mathsf{Mem},\mathsf{Stack}\}} \circ [A^1_{\mathsf{Stack}}]_{\{\mathsf{Mem},\mathsf{Stack}\}} \\ [A^1_{\mathsf{Stack}}]_{\{\mathsf{Mem},\mathsf{Stack}\}} \circ I_{\mathsf{Echo}} & \leq_{\mathsf{ctx}} & [A^2_{\mathsf{Stack}}]_{\{\mathsf{Stack},\mathsf{Echo}\}} \circ [A_{\mathsf{Echo}}]_{\{\mathsf{Stack},\mathsf{Echo}\}} \\ \end{array}$$

Then by horizontal and vertical compositionality of CR, we have

$$I_{\mathsf{Mem}} \circ I_{\mathsf{Stack}} \circ I_{\mathsf{Echo}} \leq_{\mathsf{ctx}} [A_{\mathsf{Mem}}]_{\{\mathsf{Mem},\mathsf{Stack}\}} \circ [A_{\mathsf{Stack}}^2]_{\{\mathsf{Stack},\mathsf{Echo}\}} \circ [A_{\mathsf{Echo}}]_{\{\mathsf{Stack},\mathsf{Echo}\}}$$

Furthermore, the following three CRs hold trivially by exploiting **NB** in **friend** and **UB** in **context**.

 $[A_{\mathsf{Mem}}]_{\{\mathsf{Mem},\mathsf{Stack}\}} \leq_{\mathsf{ctx}} I_{\mathsf{Mem}}, \quad [A_{\mathsf{Stack}}^2]_{\{\mathsf{Stack},\mathsf{Echo}\}} \leq_{\mathsf{ctx}} [A_{\mathsf{Stack}}^1]_{\{\}}, \quad [A_{\mathsf{Echo}}]_{\{\mathsf{Stack},\mathsf{Echo}\}} \leq_{\mathsf{ctx}} [A_{\mathsf{Echo}}]_{\{\mathsf{Echo}\}}$  Therefore, we can further simplify the top-level abstraction as follows.

$$I_{\mathsf{Mem}} \circ I_{\mathsf{Stack}} \circ I_{\mathsf{Echo}} \leq_{\mathsf{ctx}} I_{\mathsf{Mem}} \circ \big[A^1_{\mathsf{Stack}}\big]_{\{\}} \circ \big[A_{\mathsf{Echo}}\big]_{\{\mathsf{Echo}\}}$$

We conclude the sequence of examples shown so far with a few remarks. First, the module I0 is a part of the context, so that it can be arbitrary. For example, it is completely valid for I0.getint and I0.putint to make mutually recursive calls to Echo.echo although calls to Echo.input or Echo.output by I0 will trigger **UB**. Second, it is possible to leave useful assumptions and guarantees in the final abstractions, which may be necessary or helpful for further abstractions. For example, **guarantee**(pool handle == None) in  $A^1_{\text{Stack}}$  can be seen as such a guarantee. As a further example, in  $A_{\text{Echo}}$ , we can also insert **assume**(is\_prime(v)) after the call I0.getint() and **guarantee**(is\_prime(hd)) before the call I0.putint(hd). Third, the definition  $s_*$  (given in  $S^1_{\text{Stack}}$  of Fig. 4) says that it must be an impure call and the concrete and abstract arguments (also return values) coincide, which is needed to prove that every EMS function semantics satisfies  $s_*$ . Finally,

```
I_{\mathsf{RP}} := [\mathsf{Module} \ \mathsf{RP}]
                                                                                                  I_{SC} := [Module SC]
          def repeat([f:ptr, n:int64, m:int64]?) =
                                                                                                            def succ([m:int64]?) \equiv m + 1
                                                                                                  I_{\mathsf{AD}} \coloneqq [\mathsf{Module}\ \mathsf{AD}]
             if n \le 0 then m
                                                                                                            def add([n:int64, m:int64]?) =
             else { var v := (*f)(m);
                                                                                                                RP.repeat(&SC.succ, n, m)
                          RP.repeat(f, n-1, v) }
A_{\mathsf{RP}} \coloneqq [\mathsf{Module} \ \mathsf{RP}]
                                                                                                 A_{AD} := [Module AD]
           def repeat = friend(_) ≡ NB context UB
                                                                                                            def add = friend,context
A_{SC} := [Module SC]
                                                                                                               ([n:int64, m:int64]?) \equiv
           def succ = friend(_) ≡ NB context UB
                                                                                                                assume(n >= 0); n + m
H_{\mathsf{RP}}(S_{\mathsf{f}}) := \{\mathsf{RP.repeat} : \forall (f, n, m, f_{\mathsf{sem}}) : \mathsf{ptr} \times \mathsf{int64} \times \mathsf{int64} \times (\mathsf{int64} \to \mathsf{int64}).
                   \{\lambda x - d. \ \lceil x = \uparrow [f, n, m] \land n \ge 0 \land d \ge \text{Some } (\omega + n) \land \}
                                S_f \supseteq \{ *f : \forall m : \mathsf{int64}, \{ \lambda x \_ d. \ulcorner x = \uparrow [m] \land d = \mathsf{Some} \ \omega \urcorner \} \{ \lambda r \_. \ulcorner r = \uparrow (f_{\mathsf{Sem}}(m)) \urcorner \} \} \urcorner \}
                   \{\lambda r \_ . \lceil r = \uparrow (f_{\text{sem}}^n(m)) \rceil \}
S_{SC} := \{SC.succ : \forall m : int64. \{\lambda x \_ d. \lceil x = \uparrow [m] \land d = Some \_ \rceil \} \{\lambda r \_. \lceil r = \uparrow (m+1) \rceil \} \}
S_{AD} := \{AD.add : s_*\}
```

Fig. 7. Implementations and their specifications for the modules RP, SC, AD

Echo.echo() may terminate or not depending on the behavior of the IO module, which may even behave nondeterministically, and thus its termination may be nondeterministic as well. However, this does not cause any problem in AL because we are not proving termination or non-termination for Echo; rather we guarantee preservation of termination: if the abstraction terminates, so does the implementation.

# 3.5 Reasoning about function pointers

In this section, we present a general pattern for doing higher-order reasoning in AL without requiring any special support. For this, consider the simple example given in Fig. 7. The function RP.repeat(f,n,m) in  $I_{\rm RP}$  recursively apply \*f, n times, to m, where \*f is the function pointed to by the pointer value f. The definitions in  $I_{\rm SC}$  and  $I_{\rm AD}$  are straightforward to understand except that &SC. succ is the pointer value pointing to the function SC. succ. For RP and SC, we give the pure pre-abstractions  $A_{\rm RP}$  and  $A_{\rm SC}$ , where we maximally simplified them for presentation purposes. For AD, the pre-abstractions  $A_{\rm AD}$  turns the call to RP.repeat into the native addition with the non-negativity assumption about the first argument.

To specify RP. repeat, we essentially need to embed expected specifications for argument functions f inside the specification of RP. repeat. Directly supporting this would make the definition of specification 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 specification  $H_{\mathsf{RP}}$  to the module RP, given in Fig. 7, which is given as a function from specifications to specifications. We can understand that the input specification  $S_f$  includes the specifications for all the functions that are passed to RP. repeat by friends. Then  $H_{\mathsf{RP}}(S_f)$  gives a specification for RP that only allow those functions in  $S_f$  to be given as an argument to RP. repeat. With this intuition, we see the definition of  $H_{\mathsf{RP}}(S_f)$ : for arguments f, n, m and a mathematical function  $f_{\mathsf{sem}}$ , we require  $S_f$  to include the expected specification for  $f_{\mathsf{sem}}$  for any argument  $f_{\mathsf{sem}}$  is the smallest ordinal bigger than every natural number and thus we can allow  $f_{\mathsf{sem}}$  to have any finite recursion depth. Also we require RP. repeat to be pure with measure at least Some  $f_{\mathsf{sem}}$ 0 because RP. repeat makes recursive calls with depth  $f_{\mathsf{n}}$ 1 followed by a call to  $f_{\mathsf{n}}$ 2.

 $<sup>^{10}</sup>d \ge$ Some o is defined as  $\exists o' \ge o$ . d =Some o'.

```
fundef(E) \stackrel{\text{def}}{=} Any \rightarrow itree E Any
  (cond \Rightarrow X) \stackrel{\text{def}}{=} \text{if } cond \text{ holds, then } X \text{ else } \emptyset
 E_{\text{nrim}}(X) \stackrel{\text{def}}{=} \{\text{Choose}\} \uplus \{\text{Take}\} \uplus (X = \text{Any} \Rightarrow \{\text{Obs } \textit{fn } \textit{args} \mid \textit{fn} \in \text{string}, \textit{args} \in \text{Any}\})
 \begin{split} \mathbf{E}_{\mathrm{EMS}}(X) &\stackrel{\mathrm{def}}{=} \mathbf{E}_{\mathrm{prim}}(X) \uplus (X = \mathsf{Any} \Rightarrow \{\mathsf{Call} \; \mathit{fn} \; \mathit{args} \; | \; \mathit{fn} \in \mathsf{string}, \; \mathit{args} \in \mathsf{Any}\}) \uplus \\ & (X = \mathsf{Any} \Rightarrow \{\mathsf{Get}\}) \uplus (X = () \Rightarrow \{\mathsf{Put} \; a \; | \; a \in \mathsf{Any}\}) \uplus (X = \mathsf{string} \Rightarrow \{\mathsf{GetCaller}\}) \end{split}
                                                   \stackrel{\text{def}}{=} \left\{ (\text{name, init, funs}) \in \text{string} \times \text{Any} \times (\text{string} \xrightarrow{\text{fin}} \text{fundef}(E_{EMS})) \right\}
 EMS
 E_{PAbs}(X) \stackrel{\text{def}}{=} E_{EMS}(X) \uplus (X = () \Longrightarrow \{IPC\})
                                                      \overset{\text{def}}{=} \big\{ (\text{name, init, funs}) \in \text{string} \times \text{Any} \times (\text{string} \xrightarrow{\text{fin.}} (\text{fundef}(E_{PAbs}) \times \text{fundef}(E_{PAbs}))) \big\} 
 PAbs
 PCM \stackrel{\text{def}}{=} \{ (Res, +, \mathcal{V}, \varepsilon) \in (Set \times (Res \rightarrow Res \rightarrow Res) \times \mathcal{P}(Res) \times Res) \mid + \text{ is commutative and associative,} 
                                                  \varepsilon is an identity element, \mathcal{V}(\varepsilon) holds, and \mathcal{V} is monotone with respect to addition.
 \mathbf{rProp}_{\Sigma} \stackrel{\text{def}}{=} \Sigma \to \mathbf{Prop} \quad \text{ for } \Sigma \in \mathrm{PCM}
Spec_{\Sigma} \stackrel{def}{=} \{(A,c) \mid A \in Type \land c \in A \rightarrow (Any \rightarrow Any \rightarrow option \ ord \rightarrow rProp_{\Sigma}) \times (Any \rightarrow Any \rightarrow rProp_{\Sigma})\}
Specs_{\Sigma} \stackrel{def}{=} string \xrightarrow{fin} Spec_{\Sigma}
\begin{array}{ll} s_1 \sqsupseteq s_0 \in \operatorname{Spec} & \stackrel{\operatorname{def}}{=} \forall a_0 \in (s_0.\mathbb{A}) \,.\, \exists a_1 \in (s_1.\mathbb{A}) \,.\, \exists (P_0,Q_0) = s_0.\mathtt{c}(a_0) \,.\, \exists (P_1,Q_1) = s_1.\mathtt{c}(a_1) \,.\, \\ & (\forall \, x \, x_a \, d. \, (P_0 \, x \, x_a \, d) \, \vdash \, \biguplus (P_1 \, x \, x_a \, d)) \, \land \, (\forall \, r \, r_a. \, (Q_1 \, r \, r_a) \, \vdash \, \biguplus (Q_0 \, r \, r_a)) \end{array}
 S_1 \sqsupseteq S_0 \in \operatorname{Specs} \stackrel{\operatorname{def}}{=} \forall fn \in \operatorname{string}. \ \forall s_0 \in \operatorname{Spec}. \ S_0(fn) = \operatorname{Some} s_0 \implies \exists s_1 \in \operatorname{Spec}, \ S_1(fn) = \operatorname{Some} s_1 \wedge s_1 \sqsupseteq s_0 = \operatorname{Spec} s_1(fn) = \operatorname{Some} s_1 \wedge s_1 \supseteq s_0 = \operatorname{Spec} s_1(fn) = \operatorname{Spec} s_1(
 \operatorname{Mod} \stackrel{\operatorname{def}}{=} \operatorname{LD} \times (\operatorname{LD} \to \operatorname{EMS}) \operatorname{Mods} \stackrel{\operatorname{def}}{=} \operatorname{list} \operatorname{Mod} \circ \in \operatorname{Mods} \to \operatorname{Mods} \to \operatorname{Mods} \to \operatorname{Mods} = \operatorname{append}
 Trace \stackrel{\text{coind}}{=} \{e :: tr \mid e \in \text{ObsEvent}, tr \in \text{Trace}\} \uplus \{\text{Term } v \mid v \in \text{Any}\} \uplus \{\text{Diverge}\} \uplus \{\text{Error}\} \uplus \{\text{Partial}\}
 Beh \in Mods \rightarrow \mathbb{P}(\text{Trace}) \stackrel{\text{def}}{=} \dots
 M_i \leq_{\operatorname{ctx}} M_{\operatorname{a}} \in \operatorname{Mods} \stackrel{\operatorname{def}}{=} \forall C \in \operatorname{Mods} . \operatorname{Beh}(C \circ M_i) \subseteq \operatorname{Beh}(C \circ M_{\operatorname{a}})
```

Fig. 8. Formal definitions of abstraction logic

Then we can easily verify RP: for any  $S_f$  (*i.e.*, no restriction for f) and any  $S \supseteq (S_f \cup H_{RP}(S_f))$  (since RP. repeat makes a call to \*f and itself), we prove

$$I_{\mathsf{RP}} \leq_{\mathsf{ctx}} [S \rtimes (A_{\mathsf{RP}}, \varepsilon) : H_{\mathsf{RP}}(S_{\mathsf{f}})]$$

Also, we can easily verify SC: for any S, we prove

$$I_{SC} \leq_{ctx} [S \rtimes (A_{SC}, \varepsilon) : S_{SC}]$$

Then, we can easily verify AD: for any  $S_f \supseteq S_{SC}$  (since SC. succ is passed to RP.repeat) and any  $S \supseteq H_{RP}(S_f)$  (since AD. add makes a call to RP.repeat), we prove

$$I_{AD} \leq_{ctx} [S \rtimes (A_{AD}, \varepsilon) : S_{AD}]$$

Finally, we can instantiate the above CRs with  $S_f = S_{SC}$  and  $S = H_{RP}(S_{SC}) \cup S_{SC} \cup S_{AD}$  and apply the spec erasure theorem to them as follows:

```
I_{RP} \circ I_{SC} \circ I_{AD}
\leq_{ctx} [S \rtimes (A_{RP}, \varepsilon) : H_{RP}(S_{SC})] \circ [S \rtimes (A_{SC}, \varepsilon) : S_{SC}] \circ [S \rtimes (A_{AD}, \varepsilon) : S_{AD}] \text{ (by compositionality of CR)}
\leq_{ctx} [A_{RP}]_{RP,SC,AD} \circ [A_{SC}]_{RP,SC,AD} \circ [A_{AD}]_{RP,SC,AD}  (by spec erasure thm.)
```

As a more advanced example, we also verified Landin's knot [Birkedal and Bizjak 2020], which can be found in our Coq development [Author(s) 2021]. Since we do not know of any practical higher-order example for which our approach fails, we believe it is general enough in practice.

#### 4 FORMAL DEFINITIONS OF ABSTRACTION LOGIC

In this section, we present formal definitions, key properties, and the embedding of abspecs and abstraction into EMS. First of all, our Coq formalization largely relies on interaction trees [Xia et al. 2019]. Intuitively, the interaction tree itree  $E\ T$  for an event type E (defining a set of events E(X) for each set X) and a return type T can be understood as a small-step operational semantics that can take a silent step, terminate with a value in T, or trigger an event in E(X) for any set X, in which case its continuation nondeterministically receives each value in X. We enjoy two benefits of interaction trees: (i) they provide useful combinators, which made our various constructions straightforward, and (ii) they can be extracted to executable programs in OCaml. In particular, all the language constructs shown in the examples so far are simply combinators for itrees. The key combinator we use is the *interpretation* function with states in ST, which has type:

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

where we use the notation  $t[e_1 \mapsto t_1, \dots, e_n \mapsto t_n]$  to denote the interpretation of the events  $e_i$  to the (state-indexed) itree  $t_i$  in the itree t and implicitly drop identical interpretations essentially mapping e to itself and also drop the state component when ST = ().

Fig. 8 shows the formal definitions of AL. First, we define two notations  $(cond \Rightarrow X)$  for conditional construction of a set and fundef(E) for semantics of a function that takes a value in Any and executes by possibly triggering events in the event type E. We first define the primitive events  $E_{\rm prim}$  consisting of Choose and Take for any type X, and Obs for triggering observable events such as system calls that pass and receive an Any value; and the events  $E_{\rm EMS}$  extends  $E_{\rm prim}$  with Call for making a function call, Get and Put for reading from and writing to the module local state, and GetCaller for getting the caller's module name. Then EMS for semantics of a module is defined as the set of triples consisting of a module's name, an initial module-local state and function semantics triggering events in  $E_{\rm EMS}$  for (a finite set of) module functions. The events  $E_{\rm PAbs}$  extends  $E_{\rm EMS}$  with IPC for triggering an IPC, and PAbs for pre-abstraction of a module is defined similarly as EMS except that PAbs has a pair of function semantics (for **friend** and **context**) triggering events in  $E_{\rm PAbs}$  for each module function.

PCM, the set of PCMs, is defined in a standard way [Jung et al. 2018], where the predicate  $\mathcal{V}$  indicates whether a resource is defined or not. A specification (for a function) in  $\operatorname{Spec}_{\Sigma}$ , parameterized by a global PCM  $\Sigma$ , consists of a set A over which the universal quantifier in the specification quantifies, and a condition c that, given a value  $a \in A$ , gives a pair of pre and post conditions. A precondition takes concrete and abstract arguments with a measure and gives a resource proposition, which is a predicate on  $\Sigma$ ; and similarly for a postcondition but without measure. A collection of specifications in  $\operatorname{Specs}_{\Sigma}$  is a finite map from function names to function specifications. We also define the strengthening relation  $\square$  between specifications, which generalizes the simple inclusion relation following Iris [Jung et al. 2018].

Mod gives a notion of code for a single module (*i.e.*, before loading), which is parameterized by a notion of loading data 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 function in LD  $\rightarrow$  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, which forms basic units in contextual refinement, and linking  $\circ$  between them is the list append.

Then we coinductively define the set of traces, Trace. A trace is a finite or infinite sequence of observable events in ObsEvents, defined as  $\{(\text{Obs }fn\ args,r)\mid fn\in \text{string},\ args,r\in \text{Any}\}$ , possibly ended with one of the four cases: (i) normal termination with an Any value, (ii) divergence without producing any observable events, (iii) fatal error, or (iv) partial termination. The notion of partial termination can be intuitively understood as terminating the execution at the user's will

such as pressing Ctrl+C, which is dual to fatal error (*i.e.*, termination due to the program's fault). This will be clarified in the definition of Beh that gives the set of those traces that can possibly arise when loading and executing a given modules code. The definition (omitted for brevity; see our Coq development [Author(s) 2021] for details) is defined as usual except that (i) the partial termination, Partial, may occur nondeterministically at any point during execution (capturing that the user can stop the program at any time), and (ii) triggering a Choose (or Take) event is interpreted as taking the union (or intersection) of the behaviors of all possible continuations. Note that triggering **UB** (i.e., taking from the empty set) can produce all possible traces, which is dual to the interpretation of triggering **NB** (i.e., choosing from the empty set) that can only immediately terminate with Partial without any other possible traces (capturing that there is no behavior caused by the program after **NB** is triggered). Finally, we define contextual  $refinement \leq_{ctx}$  between modules codes as usual.

In AL, we establish CR using a standard simulation technique that allows module-local relational invariants, I, to depend on Kripke-style possible worlds equipped with a preorder ( $\sqsubseteq_w$ ). Then, the (coinductively-defined) greatest simulation relation  $\lesssim_w$  at a given world w relates target states (*i.e.*, the implementation side) and source states (*i.e.*, the abspec side), both of which consist of a module-local state and an itree<sup>11</sup>. Two notable cases are the return and call cases shown below: in the former one has to prove that the invariant I holds at the current or a future world; and relying on that, in the latter, one can assume the invariant holds after every function call.

$$\frac{w \sqsubseteq_{\mathcal{W}} w' \quad I \, w' \, st_i \, st_a}{(st_i, \mathsf{ret} \, r) \lesssim_{\mathcal{W}} (st_a, \mathsf{ret} \, r)} \qquad \frac{w \sqsubseteq_{\mathcal{W}} w' \quad I \, w' \, st_i \, st_a}{\forall r, w'', st_i', st_a' \cdot w' \sqsubseteq_{\mathcal{W}} w'' \wedge I \, w'' \, st_i' \, st_a' \Rightarrow (st_i', K_i[r]) \lesssim_{\mathcal{W}} (st_a, K_a[r])}{(st_i, \mathsf{var} \, r \colon = \mathsf{call} \, f \, x; \, K_i[r]) \lesssim_{\mathcal{W}} (st_a, \mathsf{var} \, r \colon = \mathsf{call} \, f \, x; \, K_a[r])}$$

Fig. 9 formally presents our translation of abspecs into EMS and that of pre-abstractions into EMS (*i.e.*, abstractions). These translations are done in the same way as we have explained except that, as we mentioned before, instead of requiring frame-preserving updates we allow to update local resources as long as they are consistent with the frame resource given at the latest interaction point. Also note that we used the same macros ASSUME and GUARANTEE for both pre and post conditions although the latter lacks the measure parameter. Here we implicitly cast postconditions to the type of preconditions by making them to ignore the measure parameter.

Now we present our core theorems. The most important one – spec erasure theorem – is already presented in Theorem 3.1. Then we present adequacy of the simulation relation.

Theorem 4.1 (Adequacy). For a given pair of module  $M_i$  and  $M_a$ , a possible world W equipped with  $\sqsubseteq_W$ , and a module-local relational invariant I, if each pair of functions with the same name for  $M_i$  and  $M_a$  are related by the simulation relation  $\lesssim$  for any argument and any module-local states satisfying I, then we have the contextual refinement between them:

$$[M_i] \leq_{ctx} [M_a]$$

We also use the following strengthening theorem.

THEOREM 4.2 (STRENGTHENING). For any  $S, S', A, \sigma, S_A$ , the following holds:

$$S' \supseteq S \implies [S \rtimes A, \sigma : S_A] \leq_{ctx} [S' \rtimes A, \sigma : S_A]$$

Finally, we briefly discuss how to define the abstraction Safe mentioned in  $\S1$ . The module Safe (ns, ns') defines each function in ns' whose semantics is simply defined to nondeterministically invoke an arbitrary function in ns with arbitrary arguments any number of times (even infinitely many). Then we have the following theorem.

 $<sup>^{11}</sup>$ We also support the stuttering index using our ordinal library, but omit it here for brevity.

```
[A]_N \stackrel{\text{def}}{=} (A.\text{name}, A.\text{init}, \lambda fn. \text{ toAbs}(N, A.\text{funs } fn)) abspecCall(S, d, frm, fn, x_a) \stackrel{\text{def}}{=}
                                                                     var (A, PQ) := S(fn);
toAbs(N, (frd, ctx)): fundef(E_{EMS}) \stackrel{\text{def}}{=} \lambda x.
                                                                    var a := choose(A); var (P, Q) := PQ(a);
  var mn := get_caller();
                                                                    var(x,d',res_f) := GUARANTEE(P,x_a,frm);
   if(mn \in N) frd(x)[IPC \mapsto skip]
                                                                    guarantee(d' \sqsubseteq d);
                   ctx(x)[IPC \mapsto skip]
                                                                     var r := call fn x;
[S \times A, \sigma : S_a] \stackrel{\text{def}}{=} (A.\mathsf{name}, (A.\mathsf{init}, \uparrow \sigma),
                                                                    var(r_a, \_, frm) := ASSUME(Q, r, res_f);
                  \lambda fn. toAbspec(S, A.funs fn, S<sub>a</sub> fn))
                                                                     (r_a, \text{ frm})
                                                                  abspecIPC(S, d, frm) \stackrel{\text{def}}{=}
toAbspec(S, (frd, ctx), s): fundef(E<sub>EMS</sub>) \stackrel{\text{def}}{=} \lambda x.
                                                                    var i := choose(ord);
  var is_friend := take(bool);
                                                                    while(choose(bool))
   if(is_friend) abspecFun(S, s, frd)(x)
                                                                       var (fn, x_a) := choose(string \times Any);
                      abspecFun(S, s_*, ctx)(x)
                                                                       (\_, frm) := abspecCall(S, d, frm, fn, x_a);
abspecFun(S, s, fun: fundef(E_{PAbs})) \stackrel{\text{def}}{=} \lambda x.
                                                                       i := choose(\{i' \in ord \mid i' < i\})
  var (A, PQ) := s;
                                                                     ((), frm)
  var a := take(A); var (P, Q) := PQ(a);
   var (x_a, d, frm) := ASSUME(P, x, \varepsilon);
                                                                  ASSUME(Cond, xr, res_f) \equiv \{
  {\sf match}\ d\ {\sf with}
                                                                     \operatorname{var}(xr_a, d, \operatorname{res}, \operatorname{frm}) := \operatorname{take}(\_);
   | None \Rightarrow var (r_a, \text{ frm}) :=
                                                                     var (res_m, _) := get;
     abspecBody(S, d, frm, fun(x_a))
                                                                     assume(Cond xr xr_a d res);
   I Some o =>
                                                                     assume(V(res + res_f + res_m + frm));
     var (\_, frm) := abspecIPC(S, d, frm);
                                                                     (xr_a, d, frm) }
     var r_a := choose(Any) end;
                                                                  GUARANTEE (Cond, xr_a, frm) \equiv {
  var (r, \_, \_) := GUARANTEE(Q, r_a, frm);
                                                                     var(xr,d',res,res_f,res_m) := choose(_);
                                                                     guarantee(Cond xr xr_a d' res);
abspecBody(S, d, frm, body) \stackrel{\text{def}}{=} body[
                                                                     guarantee(V(res + res_f + res_m + frm));
 Call fn x_a \mapsto \lambda \text{frm. abspecCall}(S, d, \text{frm}, \text{fn}, x_a),
                                                                     var (_, st) := get; put(res_m, st);
                                                                     (xr, d', res_f) }
           IPC \mapsto \lambda \text{frm. abspecIPC}(S, d, \text{frm})
```

Fig. 9. Translations of abspecs and pre-abstractions into EMS.

Theorem 4.3 (Safety). For  $ns = ns_1 \uplus ... \uplus ns_n$ , Safe  $(ns, ns_1) \circ ... \circ Safe(ns, ns_n)$  is safe (i.e., never produces an Error).

#### 5 PROOFMODE

AL proof mode consists of two modes supporting simulation reasoning and separation logic reasoning.

# 5.1 Simulation Reasoning

**Reduction Tactics.** We establish CR using a simulation technique, which relates an implementation state and an abspec state  $^{12}$ . To take steps, we provide reduction tactics that reduces the current itree (either in implementation or in abspec) into the head normal form by, *e.g.*, converting  $(i \gg j) \gg k$  into  $i \gg (j \gg k)$ . Also, we made the tactics extensible: when embedding a new language with new effects and a new handler H into EMS, we can register new reduction lemmas about H so that they are used by the reduction tactics (*e.g.*, converting  $H(i \gg j)$  into  $Hi \gg Hj$ ).

**Simulation Tactics.** We provide tactics that automate common parts of simulation reasoning. Specifically, the tactics repeatedly apply the rules given below (*i.e.*, converting the conclusion into the premise), where we omitted the state component and the world index in the simulation

<sup>&</sup>lt;sup>12</sup>In fact, our simulation tactics generally work for relating any two EMS semantics.

relation for brevity. Note that the dash-boxed rules are *complete* (*i.e.*, the premise and conclusion are equivalent) while the others not. Then we provide three tactics: **steps** automatically applies complete rules as many as possible, **force\_i** applies any available rule once in the implementation side, and **force\_a** does the same for the abspec side.

$$\frac{\exists x \in X. \ T_i \lesssim K_a[x]}{T_i \lesssim x := \operatorname{choose}(X); \ K_a[x]} \qquad \frac{\forall x \in X. \ T_i \lesssim K_a[x]}{T_i \lesssim x := \operatorname{take}(X); \ K_a[x]} \qquad \frac{P \wedge (T_i \lesssim T_a)}{T_i \lesssim \operatorname{guarantee}(P); \ T_a} \qquad \frac{P \Longrightarrow T_i \lesssim T_a}{T_i \lesssim \operatorname{assume}(P); \ T_a}$$
 
$$\frac{\forall x \in X. \ K_i[x] \lesssim T_a}{x := \operatorname{choose}(X); \ K_i[x] \lesssim T_a} \qquad \frac{\exists x \in X. \ K_i[x] \lesssim T_a}{x := \operatorname{take}(X); \ K_i[x] \lesssim T_a} \qquad \frac{P \Longrightarrow T_i \lesssim T_a}{\operatorname{guarantee}(P); \ T_i \lesssim T_a} \qquad \frac{P \wedge (T_i \lesssim T_a)}{\operatorname{assume}(P); \ T_i \lesssim T_a}$$

## 5.2 Separation Logic Reasoning

$$\forall x. \ P \mid \exists x. \ P \mid \lceil P \rceil \mid P \land Q \mid P \lor Q \mid P \Rightarrow Q \mid P \ast Q \mid P \ast Q \mid [a : \Sigma] \mid \Rightarrow P \mid P \vdash Q$$

We support the SL tactics provided by IPM (Iris Proof Mode). For this, we define the standard Iris connectives shown above on **rProp**, instead of **iProp** of Iris with step indices, and prove the lemmas that IPM (Iris Proof Mode) requires, which allows us to use IPM when proving logical entailment. Note that we omitted the later modality ( $\triangleright$ ) because we do not use step-indexing.

At Interaction Points. We have three tactics – slinit, slcall, and slret – for each interaction points. slinit is used at the beginning of a function and it moves the precondition (in rProp) into Coq hypothesis (current\_rProps). slcall is used when calling a function — where source have its call decorated with abspec\_call (Fig. 9) and target not. It asks the user to specify a subset of current\_rProps that will be consumed when proving (with IPM) both (i) callee's precondition and (ii) the relational invariant. After that, the user proceeds the proof with fresh rProps satisfying (i) callee's postcondition and (ii) the relational invariant. In other words, while the mechanism is implemented with various assume/guarantees in low-level (e.g., abspec\_call), we give a high-level interface with these tactics. slret is used at the end of the function and is similar to slcall.

**In Between.** We also support operations on current\_rProps in the middle of the simulation proof. Notably, we support (i) eliminating connectives like  $\forall$ ,  $\exists$ ,  $\lor$ ,  $\land$ , \*,  $\not\models$ ,  $\ulcorner \urcorner$ , (ii) **Assert** tactic that consumes specified rProps to establish specified goal (using IPM), and (iii) **OwnV** tactic that gives  $\mathcal{V}(\sigma)$  for given  $\lceil \overline{\sigma} \rceil$ . It is worth noting that, despite such functionalities, it suffices to put updates and validity checking only at the interaction points (as in §2.2), not in between.

#### 6 IMP AND COMPILATION TO COMPCERT

The IMP language, extended from Imp presented in [Xia et al. 2019], has the following syntax and its semantics is defined in EMS.

```
 \begin{array}{ll} x,f,g \in String & fp \in Pointer \\ e \in Expr ::= x \mid v : \mathbb{Z} \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 Stmt ::= \mathrm{skip} \mid x := e \mid s_1; s_2 \mid \mathrm{if} \ (e) \ \mathrm{then} \ \{s_1\} \ \mathrm{else} \ \{s_2\} \mid x = f(e_1,...,e_n) \mid x = fp(e_1,...,e_n) \mid x = g \mid x = \mathrm{malloc}(e) \mid \mathrm{free}(e) \mid x = \mathrm{load}(e) \mid \mathrm{store}(e_1,e_2) \mid x = \mathrm{cmp}(e_1,e_2) \\ \end{array}
```

We have developed a verified compiler<sup>13</sup> from IMP to Csharpminor of CompCert [Leroy 2006], which is then composed with CompCert to give a compiler C from IMP to CompCert's assembly.

 $<sup>^{13}</sup>$ As a simple solution to resolve a subtle mismatch between CompCert's memory model and our simplified one, we compile the free instruction to skip.

THEOREM 6.1 (SEPARATE COMPILATION CORRECTNESS). Let  $(I_1, Asm_1), \ldots, (I_n, Asm_n)$  be pairs of IMP and Asm modules such that  $C(I_i) = Some(Asm_i)$  for all i. Then, the following holds:

$$Beh(Asm_1 \bullet \cdots \bullet Asm_n)^{14} \subseteq Beh(I_{Mem} \circ I_1 \circ \cdots \circ I_n)$$
.

Note that CompCert assemblies are linked with the *syntactic linking* operator (●) and we followed the lightweight verification approach of [Kang et al. 2016].

#### 7 EVALUATION

The right table shows the SLOC of the whole development (counted by coqwc). EMS and its meta-theory amount to 5609 SLOC, SL-specific theory (including PAbs, proof mode, translations, and spec erasure theorem) to 9341 SLOC, IMP-specific theory (including syntax, semantics, its compiler, and compiler verification) to 7899 SLOC, verification examples (Cannon, Mem, Stack, Echo, Repeat, and Landin's knot) to 6136 SLOC, and general purpose Coq libraries to 3539 SLOC. In

Portion	Def	Proof
EMS	2584	3025
SL-specific	5709	3632
IMP-specific	3770	4129
Examples	3082	3054
Coq libraries	2689	850
Total	17834	14690

total, our Coq definitions amount to 17834 SLOC and proofs to 14690 SLOC.

The proof effort for verifying each example in AL is split into two: (i) that for reasoning about SL entailment, which would be comparable to that in Iris because we use IPM and prove essentially similar goals, and (ii) that for simulation reasoning, which was quite straightforward in all the examples thanks to our tactics.

Vertical compositionality of CR (*i.e.*, gradual abstraction) also simplified the proof of meta theory of AL. For instance, the spec erasure theorem (Theorem 3.1) is established by transitively composing six contextual refinements, two major ones of which are (i) removing ASSUME and GUARANTEE while concretizing the measure information and (ii) removing pure calls by proving their termination using the measure information.

All the examples in the paper are extracted to OCaml programs, and we indeed found bugs by testing before verification. For extraction, all the events are handled inside Coq except for  $E_{\rm prim}$ , which are handled by special handlers written in OCaml. Specifically, we wrote a few handlers doing IO for 0bs and a handler for Choose and Take, which asks the user for a nondeterministic choice (currently only supports int64). For testing, we extracted both implementations and abstractions, executed them and compared the results. Interestingly, we found two mis-downcast bugs in the abstraction of Echo by testing before verification. Also we can extract and run even the abspecs although it would introduce so much nondeterminism.

# 8 RELATED WORK

There have been many works in proving safety and refinement of programs in various directions but occasionally with certain restrictions. Abstraction logic can be seen as a unifying theory, based on elementary mechanisms, that can subsume most of the works without such restrictions. We will discuss those works and their restrictions if any.

*Specifications as programs.* Refinement calculus [Back and Wright 2012] understands Hoare-style specifications as programs and provides refinement between them, which enjoys fully compositionality as in CR. [Koenig 2020; Koenig and Shao 2020] recently made advances in this line of research, where they also employ dual-nondeterminism and algebraic effects (similar to interaction trees but without extraction) like AL. However, unlike AL, they do not support SL-style specifications.

<sup>&</sup>lt;sup>14</sup>We cast CompCert's events into 0bs events in EMS.

**Separation Logics.** First of all, compared to the state-of-the-art separation logics such as Iris and VST, abstraction logic does not support concurrency yet although we plan to extend AL to support it following their approaches.

There have been works based on separation logics that go beyond safety such as CaReSL[Turon et al. 2013] and ReLoC[Frumin et al. 2018]. Like AL, they establish contextual refinement using SL but in a restricted setting that does not allow transformations essentially relying on logical specifications of external modules.

Using Iris, [Sammler et al. 2019] establishes guarantee of desired properties on observable traces (*i.e.*, a sequence of system calls), instead of safety guarantee, in the presence of unverified contexts, but in a restricted setting that does not allow the contexts to invoke system calls.

**Contextual Refinement.** Certified Abstraction Layers (CAL)[Gu et al. 2015, 2018] proved effectiveness of contextual refinement in large scale verification by verifying a realistic operating system. Compared to AL, although it supports concurrency, CAL is limited in a few aspects. For example, CAL does not support SL-style specifications and thus does not allow implementations to use shared resources across modules. Also it does not allow mutual recursion between modules.

## 9 CONCLUSION AND FUTURE WORK

We present a comprehensive theory combining the benefits of contextual refinement and separation logic, together with practical tools, using the key idea of **choose** and **take** that gives an illusion of passing any information to anyone without involving physical operations. As future works, we plan to extend abstraction logic to support concurrency, and also develop testing tools that can efficiently find bugs that breaks desired contextual refinement, which may also give a certain level of confidence without verification.

#### **ACKNOWLEDGMENTS**

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

#### REFERENCES

Amal Ahmed. 2006. Step-indexed syntactic logical relations for recursive and quantified types. In European Symposium on Programming. Springer, 69–83.

Andrew W. Appel. 2011. Verified Software Toolchain. In Proceedings of the 20th European Symposium on Programming (ESOP 2011).

Anonymous Author(s). 2021. Supplementary material for this paper available to reviewers.

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

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

Rastislav Bodik, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, 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.

Cristiano Calcagno, Peter W O'Hearn, and Hongseok Yang. 2007. Local action and abstract separation logic. In 22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007). IEEE, 366–378.

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.

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, Jieung Kim, Xiongnan Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananadro. 2018. Certified concurrent abstraction layers. *ACM SIGPLAN Notices* 53, 4 (2018), 646–661.

Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580. Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018).

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

Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. 2015. A formal C memory model supporting integer-pointer casts. ACM SIGPLAN Notices 50, 6 (2015), 326–335.

Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2016. Lightweight Verification of Separate Compilation. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016).* 

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

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

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

Peter W O'hearn. 2007. Resources, concurrency, and local reasoning. *Theoretical computer science* 375, 1-3 (2007), 271–307. 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.

Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. 2019. The high-level benefits of low-level sandboxing. Proceedings of the ACM on Programming Languages 4, POPL (2019), 1–32.

Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A verified compiler for relaxed-memory concurrency. *Journal of the ACM (JACM)* 60, 3 (2013), 1–50.

The Coq Development Team. 2021. The Coq Proof Assistant 8.13.2 Reference Manual. https://coq.github.io/doc/V8.13.2/refman/.

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.

Malcolm Tyrrell, Joseph M Morris, Andrew Butterfield, and Arthur Hughes. 2006. A lattice-theoretic model for an algebra of communicating sequential processes. In *International Colloquium on Theoretical Aspects of Computing*. Springer, 123–137.

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