Certified Assembly Programming with Embedded Code Pointers

Zhaozhong Ni Zhong Shao
Department of Computer Science, Yale University
New Haven, CT 06520-8285, U.S.A.
{ni-zhaozhong, shao-zhong}@cs.yale.edu

Abstract

Embedded code pointers (ECPs) are stored handles of functions and continuations commonly seen in low-level binaries as well as functional or higher-order programs. ECPs are known to be very hard to support well in Hoare-logic style verification systems. As a result, existing proof-carrying code (PCC) systems have to either sacrifice the expressiveness or the modularity of program specifications, or resort to construction of complex semantic models. In Reynolds's LICS'02 paper, supporting ECPs is listed as one of the main open problems for separation logic.

In this paper we present a simple and general technique for solving the ECP problem for Hoare-logic-based PCC systems. By adding a small amount of syntax to the assertion language, we show how to combine semantic consequence relation with syntactic proof techniques. The result is a new powerful framework that can perform modular reasoning on ECPs while still retaining the expressiveness of Hoare logic. We show how to use our techniques to support polymorphism, closures, and other language extensions and how to solve the ECP problem for separation logic. Our system is fully mechanized. We give its complete soundness proof and a full verification of Reynolds's CPS-style "list-append" example in the Coq proof assistant.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—assertions, logics of programs, mechanical verification; D.2.4 [*Software Engineering*]: Software/Program Verification—correctness proofs, formal methods; D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Languages, Verification

Keywords Hoare Logic, Proof-Carrying Code, Embedded Code Pointers, Higher-Order Functions

1. Introduction

Proof-carrying code (PCC) [29] is a general framework that can, in principle, verify safety properties of arbitrary machine-language programs. Existing PCC systems [11, 20, 10, 12], however, have focused on programs written in type-safe languages [18, 25] or variants of typed assembly languages [27]. Type-based approaches are attractive because they facilitate automatic generation of the safety proofs (by using certifying compilers) and provide great support to modularity and higher-order language features. But they also suffer from several serious limitations. First, types are not expressive enough to specify sophisticated invariants commonly seen in the verification of low-level system software. Recent work on *logic-based type systems* [42, 38, 13, 2, 1] have made types more expressive but they still cannot specify advanced state invariants and assertions [37, 43, 44] definable in a general-purpose predicate logic with inductive definitions [40]. Second, type systems are too weak to prove advanced properties and program correctness, especially in the context of concurrent assembly code [43, 15]. Finally, different style languages often require different type systems, making it hard to reason about interoperability.

An alternative to type-based methods is to use Hoare logic [16, 22]—a widely applied technique in program verification. Hoare logic supports formal reasoning using very expressive assertions and inference rules from a general-purpose logic. In the context of foundational proof-carrying code (FPCC) [4, 43], the assertion language is often unified with the mechanized meta logic (following Gordon [17])—proofs for Hoare logic consequence relation and Hoare-style program derivations are explicitly written out in a proof assistant. For example, Touchstone PCC [30] used Hoare-style assertions to express complex program invariants. Appel *et al* [5, 6] used Hoare-style state predicates to construct a general semantic model for machine-level programs. Yu *et al* [43, 44, 15] recently developed a certified assembly programming framework that uses Hoare-style reasoning to verify low-level system libraries and general multi-threaded assembly programs.

Unfortunately, Hoare logic—as Reynolds [37] observed—does not support higher-order features such as embedded code pointers (ECPs) well. ECPs, based on context and time, are often referred to as computed-gotos, stored procedures, higher-order functions, indirect jumps, continuation pointers, etc. As the variations in its name suggest, ECP has long been an extensively used concept in programming. Because of the ECP problem, PCC systems based on Hoare logic have to either avoid supporting indirect jumps [30, 44], limit the assertions (for ECPs) to types only [19], sacrifice the modularity by requiring whole-program reasoning [43], or resort to construction of complex semantic models [6, 3]. In Reynolds's recent LICS paper [37], supporting ECPs is listed as one of the main open problems for separation logic.

In this paper we present a simple technique for solving the ECP problem in the context of certified assembly programming (CAP) [43]. By adding a small amount of syntax to the assertion language, we show how to combine semantic consequence relation (for assertion subsumption) with syntactic proof techniques. The result is a new powerful framework that can perform modular reasoning on ECPs while still retaining the expressiveness of Hoare logic. We show how to support polymorphism, closures, and other language extensions and how to use our techniques to solve the

POPL'06 January 11–13, 2006, Charleston, South Carolina, USA.
Copyright © 2006 ACM 1-59593-027-2/06/0001...$5.00.

ECP problem for separation logic. Our paper makes the following important new contributions:

- Our new framework provides the first simple and general solution to the ECP problem for Hoare-logic-style PCC systems. Here, by "simple," we mean that our method does not alter the structure of Hoare-style program derivations and assertions—this is in contrast to previous work by Appel and Ahmed *et al* [6, 3] which relies on building indexed semantic models and requires pervasive uses of indices both in program derivations and Hoare assertions. By "general", we mean that our technique can truly handle all kinds of machine-level ECPs, including those hidden inside higher-order closures. Previous work on Hoare logic [33, 28] supports procedure parameters by internalizing Hoare-logic derivations as part of the assertion language and by using stratification: a first order code pointer cannot specify any ECPs in its precondition; an n-th order code pointer can only refer to those lower-order ECPs. Stratification works at the source level and for monomorphic languages [33, 28], but for machine-level programs where ECPs can appear in any part of the memory, tracking the order of ECPs is impossible: ECPs can appear on the stack or in another function's closure (often hidden since closures are usually existentially quantified [26]).

- Our solution to the ECP problem is fully compatible with various extensions to Hoare logic such as separation logic [37] and rely-guarantee-based reasoning for concurrent assembly code [44, 15]. In fact, adding support to ECPs doesn't alter the definition and reasoning of the separation-logic primitives in any way (see Section 5). This is not true for index-based approaches [6, 3] where indices can easily pollute reasoning.

- As far as we know, we are the first to successfully verify Reynolds's CPS-style "list-append" example—which he uses to define the ECP problem for separation logic [37]. This example (see Section 5) may look seemingly simple, but its full verification requires a combination of a rich assertion language (from a general-purpose logic, for correctness proofs), inductive definitions (for defining linked lists), separation logic primitives (for memory management), and support of general ECPs and closures (which often require "impredicative existential types"). Doing such verification using type-based methods would require cramming all these features into a single type system, which we believe is fairly difficult to accomplish.

- Our work gives a precise account on how to extract the key ideas from syntactic techniques (often used in type systems) and incorporate them into Hoare logic-based PCC systems. As we will show later, this requires careful design and is by no means trivial. In fact, even figuring out the precise difference between CAP (as Hoare logic) [43] and typed-assembly language (as type system) [27] has taken us a long time. Finding a way to reconcile the requirements of CAP (*i.e.*, weakest precondition, using the underlying logical implications to support assertion subsumption) with modular support of ECPs using syntactic techniques constitutes one of our main technical contributions (also see Section 6 for a comparison with TAL and state logic [2]).

- We have implemented our system (including its complete soundness proof and a full verification of Reynolds's example) [32] in the Coq proof assistant [40]. Because our new framework does not alter the base structure of the assertion language, we are able to build tactics that exactly mirror those for the built-in meta-logic connectives; writing proofs in the presence of ECPs is same as doing so in the original CAP. Because all logic connectives in our new framework are defined and verified in the underlying meta logic, the trusted computing base (TCB) remains as small as possible.

In the rest of this paper, we first review the CAP framework [43] for Hoare logic and then discuss the ECP problem and its main challenges (Sec 2). We present our base XCAP framework for solving the ECP problem (Sec 3) and show how to extend it to support impredicative polymorphism (Sec 4). We show how to apply the same framework to solve the ECP problem for separation logic and present a full verification of Reynolds's "list-append" example (Sec 5). Finally we discuss related work and then conclude.

2. Background and Challenges

In this section we first present our target machine language and its operational semantics, both of which are formalized inside a mechanized meta logic. We then review the CAP framework [43] for Hoare logic. Finally, we look into the ECP problem in detail and discuss its main difficulties and challenges.

2.1 Target Machine and Mechanized Meta Logic

All verification systems presented in this paper share a common raw target machine TM, as defined in Fig 1. A complete TM program consists of a code heap, a dynamic state component made up of the register file and data heap, and an instruction sequence. The instruction set is minimal but extensions are straightforward. The register file is made up of 32 registers and the data heap can potentially be infinite. The operational semantics of this language (see Fig 2) should pose no surprise. Note that it is illegal to access or free undefined heap locations, or jump to a code label that does not exist; under both cases, the execution gets "stuck."

Both the syntax and the operational semantics of TM are formalized inside a mechanized general-purpose meta logic. In this paper we use the calculus of inductive constructions (CiC) [34] which is a variant of higher-order predicate logic extended with powerful inductive definitions. Although our implementation is done inside the Coq proof assistant[40], we believe that our results can also apply to other proof assistants.

In the rest of this paper we'll use the following syntax to denote terms and predicates in the underlying mechanized meta logic:

$$\begin{aligned}(\textit{Term})\ A,B &::= \mathsf{Set} \mid \mathsf{Prop} \mid \mathsf{Type} \mid x \mid \lambda x{:}A.\,B \mid A\ B\\ &\quad\mid A \to B \mid \textit{ind. def.} \mid \ldots\\ (\textit{Prop})\ p,q &::= \mathsf{True} \mid \mathsf{False} \mid \neg p \mid p \wedge q \mid p \vee q \mid p \supset q\\ &\quad\mid \forall x{:}A.\,p \mid \exists x{:}A.\,p \mid \ldots\end{aligned}$$

To represent TM, machine state can be embedded as a *State* type (which has *Set* sort in Coq); machine instruction sequences and commands can be defined as inductive definitions. General safety policies can be defined as predicates over the entire machine configuration; they will have the *Program* $\to$ *Prop* type. For example, the simple "non-stuckness" safety policy can be defined as follows:

$$\lambda\mathbb{P}.\,\forall n{:}\textit{Nat}.\ \exists\mathbb{P}'{:}\textit{Program}.\ \mathbb{P} \longmapsto^{n} \mathbb{P}'.$$

Here $\longmapsto^{n}$ is the composition of "$\longmapsto$" for n times. Under this setting, if Safe denotes a particular user-defined safety policy, an FPCC package is just a pair of program $\mathbb{P}$ together with a proof object of type $\mathsf{Safe}(\mathbb{P})$, all represented inside the underlying mechanized meta logic.

2.2 The CAP Framework

As suggested by its name (a language for <u>c</u>ertified <u>a</u>ssembly <u>p</u>rogramming), CAP [43] is a Hoare-logic framework for reasoning about assembly programs.

Specification language. First we introduce a construct Ψ (*Code Heap Specification*) for expressing user-defined safety require-

$$
\begin{array}{rll}
(\mathit{Program}) & \mathbb{P} & ::= (\mathbb{C},\mathbb{S},\mathbb{I}) \\
(\mathit{CodeHeap}) & \mathbb{C} & ::= \{\mathtt{f} \leadsto \mathbb{I}\}^* \\
(\mathit{State}) & \mathbb{S} & ::= (\mathbb{H},\mathbb{R}) \\
(\mathit{Heap}) & \mathbb{H} & ::= \{\mathtt{l} \leadsto \mathtt{w}\}^* \\
(\mathit{RegFile}) & \mathbb{R} & ::= \{\mathtt{r} \leadsto \mathtt{w}\}^* \\
(\mathit{Register}) & \mathtt{r} & ::= \{\mathtt{r}_k\}^{k \in \{0\ldots 31\}} \\
(\mathit{Word,Labels}) & \mathtt{w},\mathtt{f},\mathtt{l} & ::= i \ \ (\mathit{nat\ nums}) \\
(\mathit{InstrSeq}) & \mathbb{I} & ::= \mathsf{c};\mathbb{I} \mid \mathsf{jd}\ \mathtt{f} \mid \mathsf{jmp}\ \mathtt{r} \\
(\mathit{Command}) & \mathsf{c} & ::= \mathsf{add}\ \mathtt{r}_d,\mathtt{r}_s,\mathtt{r}_t \mid \mathsf{alloc}\ \mathtt{r}_d,i \mid \mathsf{bgti}\ \mathtt{r}_s,i,\mathtt{f} \\
 & & \quad \mid \mathsf{free}\ \mathtt{r}_s,i \mid \mathsf{ld}\ \mathtt{r}_d,\mathtt{r}_s(i) \mid \mathsf{mov}\ \mathtt{r}_d,\mathtt{r}_s \\
 & & \quad \mid \mathsf{movi}\ \mathtt{r}_d,i \mid \mathsf{st}\ \mathtt{r}_d(i),\mathtt{r}_s
\end{array}
$$

Figure 1. Syntax of target machine TM

if $\mathbb{I} =$	then $(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathbb{I}) \longmapsto$
$\mathsf{jd}\ \mathtt{f}$	$(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathbb{C}(\mathtt{f}))$ when $\mathtt{f} \in \mathsf{dom}(\mathbb{C})$
$\mathsf{jmp}\ \mathtt{r}$	$(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathbb{C}(\mathbb{R}(\mathtt{r})))$ when $\mathbb{R}(\mathtt{r}) \in \mathsf{dom}(\mathbb{C})$
$\mathsf{bgti}\ \mathtt{r}_s,i,\mathtt{f};\mathbb{I}'$	$(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathbb{I}')$ when $\mathbb{R}(\mathtt{r}_s) \le i$; $(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathbb{C}(\mathtt{f}))$ when $\mathbb{R}(\mathtt{r}_s) > i$
$\mathsf{c};\mathbb{I}'$	$(\mathbb{C},\mathtt{Next}_{\mathsf{c}}(\mathbb{H},\mathbb{R}),\mathbb{I}')$

where

if $\mathsf{c} =$	then $\mathtt{Next}_{\mathsf{c}}(\mathbb{H},\mathbb{R}) =$
$\mathsf{add}\ \mathtt{r}_d,\mathtt{r}_s,\mathtt{r}_t$	$(\mathbb{H},\mathbb{R}\{\mathtt{r}_d \leadsto \mathbb{R}(\mathtt{r}_s)+\mathbb{R}(\mathtt{r}_t)\})$
$\mathsf{mov}\ \mathtt{r}_d,\mathtt{r}_s$	$(\mathbb{H},\mathbb{R}\{\mathtt{r}_d \leadsto \mathbb{R}(\mathtt{r}_s)\})$
$\mathsf{movi}\ \mathtt{r}_d,i$	$(\mathbb{H},\mathbb{R}\{\mathtt{r}_d \leadsto i\})$
$\mathsf{alloc}\ \mathtt{r}_d,i$	$(\mathbb{H}\{\mathtt{l}\leadsto \_,\ldots,\mathtt{l}+i-1\leadsto \_\},\mathbb{R}\{\mathtt{r}_d \leadsto \mathtt{l}\})$ where $\mathtt{l},\ldots,\mathtt{l}+i-1 \notin \mathsf{dom}(\mathbb{H})$ and $\_$ is a random value
$\mathsf{free}\ \mathtt{r}_s,i$	$(\mathbb{H}/\{\mathbb{R}(\mathtt{r}_s),\ldots,\mathbb{R}(\mathtt{r}_s)+i-1\},\mathbb{R})$ when $\mathbb{R}(\mathtt{r}_s),\ldots,\mathbb{R}(\mathtt{r}_s)+i-1 \in \mathsf{dom}(\mathbb{H})$
$\mathsf{ld}\ \mathtt{r}_d,\mathtt{r}_s(i)$	$(\mathbb{H},\mathbb{R}\{\mathtt{r}_d \leadsto \mathbb{H}(\mathbb{R}(\mathtt{r}_s)+i)\})$ when $\mathbb{R}(\mathtt{r}_s)+i \in \mathsf{dom}(\mathbb{H})$
$\mathsf{st}\ \mathtt{r}_d(i),\mathtt{r}_s$	$(\mathbb{H}\{\mathbb{R}(\mathtt{r}_d)+i \leadsto \mathbb{R}(\mathtt{r}_s)\},\mathbb{R})$ when $\mathbb{R}(\mathtt{r}_d)+i \in \mathsf{dom}(\mathbb{H})$

Figure 2. Operational semantics of TM

$$
\begin{array}{rlll}
(\mathit{CdHpSpec}) & \Psi & ::= & \{\mathtt{f} \leadsto \mathtt{a}\}^* \\
(\mathit{Assertion}) & \mathtt{a} & \in & \mathit{State} \to \mathit{Prop} \\
(\mathit{AssertImp}) & \mathtt{a} \Rightarrow \mathtt{a}' & \triangleq & \forall \mathbb{S}.\, \mathtt{a}\, \mathbb{S} \supset \mathtt{a}'\, \mathbb{S}
\end{array}
$$

Figure 3. Assertion language for CAP

$\boxed{\Psi_G \vdash \{\mathtt{a}\}\mathbb{P}}$ ***(Well-formed Program)***

$$\frac{\Psi_G \vdash \mathbb{C}:\Psi_G \qquad (\mathtt{a}\,\mathbb{S}) \qquad \Psi_G \vdash \{\mathtt{a}\}\mathbb{I}}{\Psi_G \vdash \{\mathtt{a}\}(\mathbb{C},\mathbb{S},\mathbb{I})} \ (\text{PROG})$$

$\boxed{\Psi_{IN} \vdash \mathbb{C}:\Psi}$ ***(Well-formed Code Heap)***

$$\frac{\Psi_{IN} \vdash \{\mathtt{a}_i\}\mathbb{I}_i \qquad \forall \mathtt{f}_i}{\Psi_{IN} \vdash \{\mathtt{f}_1 \leadsto \mathbb{I}_1,\ldots,\mathtt{f}_n \leadsto \mathbb{I}_n\}:\{\mathtt{f}_1 \leadsto \mathtt{a}_1,\ldots,\mathtt{f}_n \leadsto \mathtt{a}_n\}} \ (\text{CDHP})$$

$$\frac{\begin{array}{c}\Psi_{IN1} \vdash \mathbb{C}_1:\Psi_1 \qquad \Psi_{IN2} \vdash \mathbb{C}_2:\Psi_2 \qquad \Psi_{IN1}(\mathtt{f}) = \Psi_{IN2}(\mathtt{f}) \\ \mathsf{dom}(\mathbb{C}_1) \cap \mathsf{dom}(\mathbb{C}_2) = \emptyset \qquad \forall \mathtt{f} \in \mathsf{dom}(\Psi_{IN1}) \cap \mathsf{dom}(\Psi_{IN2})\end{array}}{\Psi_{IN1} \cup \Psi_{IN2} \vdash \mathbb{C}_1 \cup \mathbb{C}_2 : \Psi_1 \cup \Psi_2} \ (\text{LINK})$$

$\boxed{\Psi \vdash \{\mathtt{a}\}\mathbb{I}}$ ***(Well-formed Instruction Sequence)***

$$\frac{\begin{array}{c}\mathtt{a} \Rightarrow (\mathtt{a}' \circ \mathtt{Next}_{\mathsf{c}}) \qquad \Psi \vdash \{\mathtt{a}'\}\mathbb{I} \\ \mathsf{c} \in \{\mathsf{add},\mathsf{mov},\mathsf{movi},\mathsf{alloc},\mathsf{free},\mathsf{ld},\mathsf{st}\}\end{array}}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{c};\mathbb{I}} \ (\text{SEQ})$$

$$\frac{\mathtt{a} \Rightarrow \Psi(\mathtt{f}) \qquad \mathtt{f} \in \mathsf{dom}(\Psi)}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{jd}\ \mathtt{f}} \ (\text{JD})$$

$$\frac{\begin{array}{ll}(\lambda(\mathbb{H},\mathbb{R}).\, \mathbb{R}(\mathtt{r}_s) \le i \ \wedge\ \mathtt{a}\,(\mathbb{H},\mathbb{R})) \Rightarrow \mathtt{a}' & \Psi \vdash \{\mathtt{a}'\}\mathbb{I} \\ (\lambda(\mathbb{H},\mathbb{R}).\, \mathbb{R}(\mathtt{r}_s) > i \ \wedge\ \mathtt{a}\,(\mathbb{H},\mathbb{R})) \Rightarrow \Psi(\mathtt{f}) & \mathtt{f} \in \mathsf{dom}(\Psi)\end{array}}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{bgti}\ \mathtt{r}_s,i,\mathtt{f};\mathbb{I}} \ (\text{BGTI})$$

$$\frac{\mathtt{a} \Rightarrow (\lambda(\mathbb{H},\mathbb{R}).\, \mathtt{a}'(\mathbb{H},\mathbb{R}) \wedge \mathbb{R}(\mathtt{r}) \in \mathsf{dom}(\Psi) \wedge \Psi(\mathbb{R}(\mathtt{r})) = \mathtt{a}')}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{jmp}\ \mathtt{r}} \ (\text{JMP})$$

Figure 4. Inference rules for CAP

ments in Hoare-logic style (see Fig 3). A code heap specification associates every code label with an assertion, with the intention that the precondition of a code block is described by the corresponding assertion. CAP programs are written in continuation-passing style because there are no instructions directly in correspondence with function call and return in a high-level language. Hence postconditions in Hoare logic do not have an explicit counterpart in CAP; they are interpreted as preconditions of the continuations.

Following Gordon [17], CAP's assertion language is directly unified with the underlying mechanized meta logic (i.e., shallow embedding). CAP assertions only track the state component in the machine configuration, so any terms of type $\mathit{State} \to \mathit{Prop}$ are valid CAP assertions. For example, an assertion specifying that the registers $\mathtt{r}_1$, $\mathtt{r}_2$, and $\mathtt{r}_3$ store the same value can be written as:

$$\lambda(\mathbb{H},\mathbb{R}).\, \mathbb{R}(\mathtt{r}_1) = \mathbb{R}(\mathtt{r}_2) \wedge \mathbb{R}(\mathtt{r}_2) = \mathbb{R}(\mathtt{r}_3).$$

To simplify the presentation, we lift the propositional implication ($\supset$) to the assertion level ($\Rightarrow$). Note the slight notational abuse for convenience—state meta-variables are reused as state variables.

Inference rules. CAP defines a set of inference rules for proving specification judgments for well-formed programs, code heaps, and instruction sequences (see Fig 4). A TM program is well-formed (rule PROG) under assertion $\mathtt{a}$ if both the global code heap and the current instruction sequence are well-formed and the machine state satisfies the assertion $\mathtt{a}$.

To support separate verification of code modules, we have made some changes to the inference rules for well-formed code heaps from the original CAP. A module is defined as a small code heap which can contain as few as one code block. Each module is associated with an "import" Ψ_{IN} interface and an "export" interface Ψ. A programmer can first establish well-formedness of each individual module via the CDHP rule. Two non-conflicting modules can then be linked together via the LINK rule. All code blocks will eventually be linked together to form a single global code heap with specification Ψ_G (which is used in the well-formed program rule). These two code heap rules provide basic support for modular verification. However, as we'll show later, the modularity breaks down when we reason about ECPs.

The intuition behind well-formed instruction sequence judgment is that if the instruction sequence $\mathbb{I}$ starts execution in a machine state which satisfies the assertion $\mathtt{a}$, then executing $\mathbb{I}$ is safe with respect to the specification Ψ. An instruction sequence preceded by c is safe (rule SEQ) if we can find another assertion a' which serves both as the post-condition of c and as the precondition of the tail instruction sequence. A direct jump is safe (rule JD) if the current assertion can imply the precondition of the target code block as specified in Ψ. An indirect jump is similar (rule JMP) except it refers to the register file for the target code label; unfortunately, this treatment of ECPs requires reasoning about control flow and breaks the modularity (see Sec 2.3).

The soundness theorem below guarantees that given a well-formed program, if it starts with the current instruction sequence, the machine will never get stuck:

Theorem 2.1 (Soundness) If $\Psi_G \vdash \{a\}\mathbb{P}$, then for all natural number n, there exists a program $\mathbb{P}'$ such that $\mathbb{P} \longmapsto^n \mathbb{P}'$.

The proof for this can be established following the syntactic approach of proving type soundness [41] by proving the progress and preservation lemmas (omitted). In order to address the new LINK rule, we need the following code heap typing lemma, whose proof needs the instruction sequence weakening lemma below.

Lemma 2.2 (Code Heap Typing) If $\Psi_{IN} \vdash \mathbb{C}:\Psi$ and $f \in \mathrm{dom}(\Psi)$, then $f \in \mathrm{dom}(\mathbb{C})$ and $\Psi_{IN} \vdash \{\Psi(f)\}\,\mathbb{C}(f)$.

Lemma 2.3 (Instruction Sequence Weakening) If $\Psi \vdash \{a\}\,\mathbb{I}$, $\Psi \subseteq \Psi'$, and $a' \Rightarrow a$ then $\Psi' \vdash \{a'\}\,\mathbb{I}$.

Yu *et al* [43, 44] have also shown that CAP can be easily extended to prove more general safety properties by introducing invariant assertions into the inference rules. Furthermore, by mechanizing the CAP inference rules and the soundness proofs in Coq, we can easily construct FPCC packages for CAP programs [43, 19].

2.3 Embedded Code Pointers

Embedded code pointers (ECPs) are stored handles of functions and continuations commonly seen in low-level binaries as well as higher-order programs. At the assembly level (as in TM), ECPs denote those memory addresses (labels) stored in registers or memory cells, pointing to the start of code blocks. Supporting ECPs is an essential part of assembly code verification. To understand better about the ECP problem, let's take a look at the following example:

```
f1: mov r1, f1 // no assumption
    jd  f2

f2: jmp r1     // r1 stores an ECP with no assumption
```

Here we have defined two assembly code blocks. The first block, labeled f_1, makes no assumption about the state; it simply moves the code label f_1 into register r_1 and then directly jumps to the other code block labeled f_2. The f_2 block requires that upon entering, register r_1 must contain a code pointer that has no assumption about the state; it simply makes an indirect jump to this embedded code pointer and continues execution from there. If the execution initially starts from f_1, the machine will loop forever between the two code blocks and never get stuck. It is important to note that both code blocks are independently written so they are expected to not just work with each other, but with any other code satisfying their specifications as well.

We introduce a predicate, $\mathsf{cptr}(f,a)$, to state that value f is a *valid* code pointer with precondition a. Following the notations in Sec 2.2, we define the "no assumption" assertion as $a_T \triangleq \lambda\mathbb{S}.\,\mathsf{True}$. The preconditions for f_1 and f_2 can be written as $a_1 \triangleq a_T$ and

$$a_2 \triangleq \lambda(\mathbb{H},\mathbb{R}).\, a_T\ (\mathbb{H},\mathbb{R}) \ \wedge\ \mathsf{cptr}(\mathbb{R}(r_1), a_T).$$

But what should be the definition of $\mathsf{cptr}(f,a)$?

Semantic approach. The semantic approach to this problem is to directly internalize the Hoare derivations as part of the assertion language and to define $\mathsf{cptr}(f,a)$ as valid if there exists a Hoare derivation for the code block at f with precondition a. Using the notation from CAP, it can be informally written as:

$$\mathsf{cptr}(f,a)\ \triangleq\ \Psi \vdash \{a\}\,\mathbb{C}(f).$$

This is clearly ill-formed since Ψ is not defined anywhere and it can not be treated as a parameter of the cptr predicate—the assertion a, which is used to form Ψ, may refer to the cptr predicate again.

Semantic approach: stratification. To break the circularity, one approach is to stratify all ECPs so that only the cptr definitions (and well-formedness proofs) for highly-ranked code blocks can refer to those of lower-ranked ones. More specifically, a first order code pointer does not specify any ECP in its precondition (its code does not make any indirect jump) so we can define cptr over them first; an n-th order code pointer can only refer to those lower-order ECPs so we can define cptr inductively following the same order. The cptr predicate definition would become:

$$\mathsf{cptr}(f,a,k)\ \triangleq\ \Psi_{k-1} \vdash \{a\}\,\mathbb{C}_{k-1}(f).$$

Stratification works for monomorphic languages with simple procedure parameters [33, 28], but for machine-level programs where ECPs can appear in any part of the memory, tracking the orders of ECPs is impossible: ECPs can appear on the stack or in another function's closure (often hidden because closures are usually existentially quantified [26]).

Semantic approach: indexing. Another approach, by Appel *et al* [6, 3, 39], also introduces an parameter k to the cptr predicate. Instead of letting k refer to the depth of nesting ECPs as the stratification approach does, the "index" k now refers to the maximum number of safe future computation steps. Roughly speaking, $\mathsf{cptr}(f,a,k)$ means that it is "safe" to execute the next (at most) $k-1$ instructions, starting from the code block at f with precondition a. Indexing must also be done for assertions and all the Hoare inference rules. For example, the indirect jump rule would have the following shape:

$$\frac{a \Rightarrow \lambda(\mathbb{H},\mathbb{R}).\, a'\ (\mathbb{H},\mathbb{R}) \wedge \mathsf{cptr}(\mathbb{R}(r), a', k-1)}{\Psi \vdash_k \{a\}\mathsf{jmp}\ r}\ .$$

Indexed assertions can only describe safety properties of finite steps. To establish safety properties about infinite future executions, one needs to do induction over the index. Because of the pervasive uses of indices everywhere, indexing dramatically alters the structure of Hoare-logic program derivations and assertions. This makes it hard to use together with other extensions such as separation logic [37] and rely-guarantee-based reasoning [44, 15].

Syntactic approach. Rather than using the semantic methods, CAP takes a syntactic approach and is essentially reasoning about the control flow. Validity of ECPs is established in two steps. In the first step, in indirect jump rule JMP it only requires that we look up the assertion for the target code label stored in register r from the code heap specification Ψ. (The equality of assertions used here is the Coq equality eq which is equivalent to the Leibniz' equality.)

$$\frac{a \Rightarrow (\lambda(\mathbb{H},\mathbb{R}).\, a'\ (\mathbb{H},\mathbb{R})\ \wedge\ \mathbb{R}(r) \in \mathrm{dom}(\Psi)\ \wedge\ \Psi(\mathbb{R}(r)) = a')}{\Psi \vdash \{a\}\,\mathsf{jmp}\ r}\ .$$

Then in the top-level PROG rule the well-formedness of global code heap is checked ($\Psi_G \vdash \mathbb{C}:\Psi_G$) to make sure that every assertion stored in Ψ_G (which is the union of all local Ψ) is indeed a valid precondition for the corresponding code block. The effect of this two steps combined together guarantees that the Hoare derivations internalized in the semantic approach is still obtainable in the syntactic approach for the preservation of whole program. This approach is often used by type systems such as TAL.

But how do we know that such label indeed falls into the domain of Ψ? We reason about the control flow. Take the code block of $\mathtt{f}_2$ as an example, we need to prove:

$$\mathsf{a}_2 \Rightarrow (\lambda(\mathbb{H},\mathbb{R}).\, \mathsf{a}'\ (\mathbb{H},\mathbb{R})\ \wedge\ \mathbb{R}(\mathtt{r}_1)\in\mathsf{dom}(\Psi)\ \wedge\ \Psi(\mathbb{R}(\mathtt{r}_1))=\mathsf{a}')$$

which unfolds to

$$\begin{array}{l}\forall\mathbb{H},\mathbb{R}.\,(\mathsf{a}_T\ (\mathbb{H},\mathbb{R})\ \wedge\ \mathsf{cptr}(\mathbb{R}(\mathtt{r}_1),\mathsf{a}_T))\\ \qquad\supset (\mathsf{a}'\ (\mathbb{H},\mathbb{R})\ \wedge\ \mathbb{R}(\mathtt{r}_1)\in\mathsf{dom}(\Psi)\ \wedge\ \Psi(\mathbb{R}(\mathtt{r}_1))=\mathsf{a}').\end{array}$$

Clearly we should require the following to hold:

$$\forall\mathbb{H},\mathbb{R}.\,\mathsf{cptr}(\mathbb{R}(\mathtt{r}_1),\mathsf{a}_T)\supset(\mathbb{R}(\mathtt{r}_1)\in\mathsf{dom}(\Psi)\ \wedge\ \Psi(\mathbb{R}(\mathtt{r}_1))=\mathsf{a}').$$

If we let the cptr predicate directly refer to Ψ in its definition, assertion and specification become cyclic definitions since Ψ consists of a mapping from code labels to assertions (which can contain cptr).

Previous CAP implementation [43] transforms the above formula into:

$$\forall\mathbb{H},\mathbb{R}.\,\mathsf{cptr}(\mathbb{R}(\mathtt{r}_1),\mathsf{a}_T)\supset\mathbb{R}(\mathtt{r}_1)\in\{\,\mathtt{f}\mid\mathtt{f}\in\mathsf{dom}(\Psi)\wedge\Psi(\mathtt{f})=\mathsf{a}_T\}$$

and statically calculates the address set on the right side using the global code heap specification Ψ_G. It can then define:

$$\mathsf{cptr}(\mathtt{f},\mathsf{a})\ \triangleq\ \mathtt{f}\in\{\,\mathtt{f}'\mid\mathtt{f}'\in\mathsf{dom}(\Psi_G)\wedge\Psi_G(\mathtt{f}')=\mathsf{a}\}.$$

This is clearly not satisfactory as the actual definition of $\mathsf{cptr}(\mathtt{f},\mathsf{a})$ no longer refers to a! Instead, it will be in the form of $\mathtt{f}\in\{\mathtt{f}_1,\ldots,\mathtt{f}_n\}$ which is not modular and very hard to reason about.

Taking the modularity issue more seriously, from the JMP rule again, we notice that all ECPs it can jump to are those contained in the current local Ψ only. Since the CAP language presented in previous section supports separate verification through linking rule LINK, when checking each module's code heap, we do not have the global specification Ψ_G and only have the declared import interface Ψ_{IN}. For our example, checking the code blocks under different organizations of modules would result in different a_2 assertions:

$\lambda(\mathbb{H},\mathbb{R}).\,\mathbb{R}(\mathtt{r}_1)\in\{\mathtt{f}_1\}$ when $\mathtt{f}_1$ and $\mathtt{f}_2$ are in a same module and there is no other block with precondition a_T in it;

$\lambda(\mathbb{H},\mathbb{R}).\,\mathbb{R}(\mathtt{r}_1)\in\{\mathtt{f}_1,\mathtt{f}_3\}$ when $\mathtt{f}_1$ and $\mathtt{f}_2$ are in a same module and there is a block $\mathtt{f}_3$ also with precondition a_T in it;

$\lambda(\mathbb{H},\mathbb{R}).\,\mathbb{R}(\mathtt{r}_1)\in\{\}$ when $\mathtt{f}_1$ and $\mathtt{f}_2$ are not in a same module and there is no other block with precondition a_T in $\mathtt{f}_2$ module;

$\lambda(\mathbb{H},\mathbb{R}).\,\mathbb{R}(\mathtt{r}_1)\in\{\mathtt{f}_3\}$ when $\mathtt{f}_1$ and $\mathtt{f}_2$ are not in a same module and there is a block $\mathtt{f}_3$ also with precondition a_T in $\mathtt{f}_2$ module.

Since we usually do not know the target addresses statically, we cannot put all possible indirect code pointers into the import specification Ψ_{IN}. The syntactic approach used by CAP cannot support general ECPs without resorting to the whole-program analysis. This greatly limits CAP's modularity and expressive power.

2.4 The Main Challenges

Given the ECP problem explained so far, we can summarize some important criteria for evaluating its possible solutions:

- Is it expressive? The new system should retain all expressive power from CAP. In particular, we still want to write assertions as general logic predicates. Ideally, there should be a "type-preserving" translation from CAP into the new system.
- Is it easy to specify? The specifications should be self-explained and can be used to reason about safety properties directly. There should be no need of translating ECP specifications into less informative forms such as indexed assertions or addresses sets as found in approaches previously discussed.
- Is it modular? The specifications should be independently writable and ECPs can be freely passed across the modular boundaries.
- Is it simple? The approach should better not involve overwhelming efforts in design and implementation when compared to CAP. It should not alter or pollute the basic structure of Hoare-style program derivations and assertions.
- Can it support extensions easily? The approach should work smoothly with common language features and popular extensions to Hoare logic.

3. The XCAP Framework

In this section we present our new XCAP framework and show how to use syntactic techniques to perform modular reasoning on ECPs while still retaining the expressive power of Hoare logic. XCAP shares the same target machine TM (see Fig 1 and 2) with CAP.

3.1 Informal Development

To avoid the "circular specification" problem in the syntactic approach (to ECP), we break the loop by adding a small amount of syntax to the assertion language, and then split the syntax of assertions from their "meanings" (or validities). The key idea here is to delay the checking of the well-formedness property of ECPs. Instead of checking them individually at the instruction sequence level using locally available specification Ψ_{IN}, we collect all these checks into one global condition that would only need to be established with respect to the global code heap specification Ψ_G.

Basically cptr is now merely a syntactic constant and can appear in any assertion at any time in form of $\mathsf{cptr}(\mathtt{f},\mathtt{a})$. Its meaning (or validity) is not revealed during the verification of local modules (*i.e.*, the well-formed instruction sequence rules). An "interpretation" will translate the ECP assertion syntax into its meaning (as a meta proposition) when the global code heap specification Ψ_G is finally available in the top-level PROG rule. The PROG rule will then complete the well-formedness check for the whole program. We achieve modularity through this two-stage verification structure.

Note that requiring the global specification Ψ_G in the PROG rule does not break any modularity, since at runtime all code (and their specifications) must be made available before they can be executed. Besides, the top-level rule PROG only needs to be validated once for the initial machine configuration.

3.2 Formalization

Next we give a formal presentation of our new XCAP framework, which has been fully implemented in Coq proof assistant and the implementation is available at the FLINT web site [32].

Extended propositions and the specification language. Figure 5 defines the core of the XCAP specification language which we call *extended logical propositions* (*PropX*). *PropX* can be viewed as a lifted version of the meta logic propositions, extended with an additional cptr constant: the base case $\langle p\rangle$ is just the lifted proposition p (thus *PropX* retains the full expressive power of meta logic propositions); cptr is the constructor for specifying ECP propositions; to interoperate lifted propositions and ECP propositions, we also lift all the logical connectives and quantifiers.

For universal and existential quantifications, we use higher-order abstract syntax (HOAS) [36] to represent them. For example, $\forall\!\!\forall x\!:\!A.\mathtt{P}$ is actually implemented as $\forall\!\!\forall(\lambda x\!:\!A.\mathtt{P})$. The benefit here is that we can utilize the full expressive power of the meta logic (in our case, CiC/Coq) and use a single quantifier to quantify over all possible types (A) such as *Prop*, *State*, and even *State* $\rightarrow$ *Prop*.

Extended propositions can be used to construct "extended" predicates using the abstraction facility in the underlying meta

$$
\begin{array}{lllll}
(\mathit{PropX}) & \mathtt{P},\mathtt{Q} & ::= & \langle p\rangle & \mathit{lifted\ meta\ proposition}\\
 & & | & \mathsf{cptr}(\mathtt{f},\mathtt{a}) & \mathit{embedded\ code\ pointer}\\
 & & | & \mathtt{P}\,\mathbb{\wedge}\,\mathtt{Q} & \mathit{conjunction}\\
 & & | & \mathtt{P}\,\mathbb{\vee}\,\mathtt{Q} & \mathit{disjunction}\\
 & & | & \mathtt{P}\twoheadrightarrow\mathtt{Q} & \mathit{implication}\\
 & & | & \mathbb{\forall} x{:}A.\,\mathtt{P} & \mathit{universal\ quantification}\\
 & & | & \mathbb{\exists} x{:}A.\,\mathtt{P} & \mathit{existential\ quantification}\\
(\mathit{CdHpSpec}) & \Psi & ::= & \{\mathtt{f}\rightsquigarrow\mathtt{a}\}^{*} & \\
(\mathit{Assertion}) & \mathtt{a} & \in & \mathit{State}\rightarrow\mathit{PropX} & \\
(\mathit{AssertImp}) & \mathtt{a}\Rightarrow\mathtt{a}' & \triangleq & \forall\Psi,\mathbb{S}.\,[\![\mathtt{a}]\!]_{\Psi}\,\mathbb{S}\supset[\![\mathtt{a}']\!]_{\Psi}\,\mathbb{S} &
\end{array}
$$

Figure 5. Extended propositions and assertion language for XCAP

$$
\begin{array}{rcl}
[\![\mathtt{a}]\!]_{\Psi} & \triangleq & \lambda\mathbb{S}.\,[\![\mathtt{a}\,\mathbb{S}]\!]_{\Psi}\\
\\
[\![\langle p\rangle]\!]_{\Psi} & \triangleq & p\\
[\![\mathsf{cptr}(\mathtt{f},\mathtt{a})]\!]_{\Psi} & \triangleq & \mathtt{f}\in\mathsf{dom}(\Psi)\wedge\Psi(\mathtt{f})=\mathtt{a}\\
[\![\mathtt{P}\,\mathbb{\wedge}\,\mathtt{Q}]\!]_{\Psi} & \triangleq & [\![\mathtt{P}]\!]_{\Psi}\wedge[\![\mathtt{Q}]\!]_{\Psi}\\
[\![\mathtt{P}\,\mathbb{\vee}\,\mathtt{Q}]\!]_{\Psi} & \triangleq & [\![\mathtt{P}]\!]_{\Psi}\vee[\![\mathtt{Q}]\!]_{\Psi}\\
[\![\mathtt{P}\twoheadrightarrow\mathtt{Q}]\!]_{\Psi} & \triangleq & [\![\mathtt{P}]\!]_{\Psi}\supset[\![\mathtt{Q}]\!]_{\Psi}\\
[\![\mathbb{\forall} x{:}A.\,\mathtt{P}]\!]_{\Psi} & \triangleq & \forall B{:}A.\,[\![\mathtt{P}[B/x]]\!]_{\Psi}\\
[\![\mathbb{\exists} x{:}A.\,\mathtt{P}]\!]_{\Psi} & \triangleq & \exists B{:}A.\,[\![\mathtt{P}[B/x]]\!]_{\Psi}
\end{array}
$$

Figure 6. Interpretations of extended propositions & assertions

logic. For example, the following extended state predicate of type *State*$\rightarrow$*PropX* says registers $\mathtt{r}_1$, $\mathtt{r}_2$, and $\mathtt{r}_3$ store the same value:

$$\lambda(\mathbb{H},\mathbb{R}).\,\langle\mathbb{R}(\mathtt{r}_1)=\mathbb{R}(\mathtt{r}_2)\wedge\mathbb{R}(\mathtt{r}_2)=\mathbb{R}(\mathtt{r}_3)\rangle.$$

Extended predicates are not limited to be over machine states only. For example, the following extended value predicate resembles the code pointer type found in type systems. (Here $\mathtt{a}$, the precondition of the code block pointed to by $\mathtt{f}$, is an extended state predicate.)

$$\mathsf{code}\ \mathtt{a}\ \triangleq\ \lambda\mathtt{f}.\,\mathsf{cptr}(\mathtt{f},\mathtt{a})$$

Using extended logical propositions and predicates, we can also define code heap specifications (Ψ), assertions ($\mathtt{a}$), and the assertion subsumption relation ($\Rightarrow$) accordingly (see Fig 5).

Extended propositions adds a thin layer of syntax over meta propositions. Fig 6 presents an **interpretation** of their validity in meta logic. It is defined as a meta function. A lifted proposition $\langle p\rangle$ is valid if p is valid in the meta logic. Validity of ECP propositions can only be testified with a code heap specification Ψ, so we make it a parameter of the interpretation function; Ψ is typically instantiated by (but not limited to) the global specification Ψ_{G}. Interpretation of $\mathsf{cptr}(\mathtt{f},\mathtt{a})$ tests the equality of $\mathtt{a}$ with $\Psi(\mathtt{f})$. Here we use the inductively defined Coq equality `eq` (equivalent to the Leibniz' equality) extended with extensionality of functions (a safe extension commonly used in the Coq [21] community). Note that the use of equality predicate in logic is different from the use of equality function in programming—a programmer must supply the proper equality proofs in order to satisfy the ECP interpretation. Extended logical connectives and quantifiers are interpreted in the straight-forward way. Note that HOAS [36] is used in the interpretation of extended implications.

$\boxed{\Psi_{G}\vdash\{\mathtt{a}\}\mathbb{P}}$ ***(Well-formed Program)***

$$\frac{\Psi_{G}\vdash\mathbb{C}{:}\Psi_{G}\qquad([\![\mathtt{a}]\!]_{\Psi_{G}}\ \mathbb{S})\qquad\Psi_{G}\vdash\{\mathtt{a}\}\mathbb{I}}{\Psi_{G}\vdash\{\mathtt{a}\}(\mathbb{C},\mathbb{S},\mathbb{I})}\ (\text{PROG})$$

$\boxed{\Psi_{IN}\vdash\mathbb{C}{:}\Psi}$ ***(Well-formed Code Heap)***

$$\frac{\Psi_{IN}\vdash\{\mathtt{a}_i\}\mathbb{I}_i\qquad\forall\mathtt{f}_i}{\Psi_{IN}\vdash\{\mathtt{f}_1\rightsquigarrow\mathbb{I}_1,\ldots,\mathtt{f}_n\rightsquigarrow\mathbb{I}_n\}:\{\mathtt{f}_1\rightsquigarrow\mathtt{a}_1,\ldots,\mathtt{f}_n\rightsquigarrow\mathtt{a}_n\}}\ (\text{CDHP})$$

$$\frac{\begin{array}{c}\Psi_{IN1}\vdash\mathbb{C}_1{:}\Psi_1\qquad\Psi_{IN2}\vdash\mathbb{C}_2{:}\Psi_2\qquad\Psi_{IN1}(\mathtt{f})=\Psi_{IN2}(\mathtt{f})\\ \mathsf{dom}(\mathbb{C}_1)\cap\mathsf{dom}(\mathbb{C}_2)=\emptyset\qquad\forall\mathtt{f}\in\mathsf{dom}(\Psi_{IN1})\cap\mathsf{dom}(\Psi_{IN2})\end{array}}{\Psi_{IN1}\cup\Psi_{IN2}\vdash\mathbb{C}_1\cup\mathbb{C}_2:\Psi_1\cup\Psi_2}\ (\text{LINK})$$

$\boxed{\Psi\vdash\{\mathtt{a}\}\mathbb{I}}$ ***(Well-formed Instruction Sequence)***

$$\frac{\begin{array}{c}\mathtt{a}\Rightarrow(\mathtt{a}'\circ\mathtt{Next}_{\mathtt{c}})\qquad\Psi\vdash\{\mathtt{a}'\}\mathbb{I}\\ \mathtt{c}\in\{\mathsf{add},\mathsf{mov},\mathsf{movi},\mathsf{alloc},\mathsf{free},\mathsf{ld},\mathsf{st}\}\end{array}}{\Psi\vdash\{\mathtt{a}\}\,\mathtt{c};\mathbb{I}}\ (\text{SEQ})$$

$$\frac{\mathtt{a}\Rightarrow\Psi(\mathtt{f})\qquad\mathtt{f}\in\mathsf{dom}(\Psi)}{\Psi\vdash\{\mathtt{a}\}\,\mathsf{jd}\ \mathtt{f}}\ (\text{JD})$$

$$\frac{\begin{array}{l}(\lambda(\mathbb{H},\mathbb{R}).\,\langle\mathbb{R}(\mathtt{r}_s)\le i\rangle\ \mathbb{\wedge}\ \mathtt{a}\ (\mathbb{H},\mathbb{R}))\Rightarrow\mathtt{a}'\qquad\Psi\vdash\{\mathtt{a}'\}\mathbb{I}\\ (\lambda(\mathbb{H},\mathbb{R}).\,\langle\mathbb{R}(\mathtt{r}_s)>i\rangle\ \mathbb{\wedge}\ \mathtt{a}\ (\mathbb{H},\mathbb{R}))\Rightarrow\Psi(\mathtt{f})\quad\mathtt{f}\in\mathsf{dom}(\Psi)\end{array}}{\Psi\vdash\{\mathtt{a}\}\,\mathsf{bgti}\ \mathtt{r}_s,i,\mathtt{f};\mathbb{I}}\ (\text{BGTI})$$

$$\frac{\mathtt{a}\Rightarrow(\lambda(\mathbb{H},\mathbb{R}).\,\mathtt{a}'\ (\mathbb{H},\mathbb{R})\ \mathbb{\wedge}\ \mathsf{cptr}(\mathbb{R}(\mathtt{r}),\mathtt{a}'))}{\Psi\vdash\{\mathtt{a}\}\,\mathsf{jmp}\ \mathtt{r}}\ (\text{JMP})$$

$$\frac{(\lambda\mathbb{S}.\,\mathsf{cptr}(\mathtt{f},\Psi(\mathtt{f}))\,\mathbb{\wedge}\,\mathtt{a}\,\mathbb{S})\Rightarrow\mathtt{a}'\quad\mathtt{f}\in\mathsf{dom}(\Psi)\quad\Psi\vdash\{\mathtt{a}'\}\mathbb{I}}{\Psi\vdash\{\mathtt{a}\}\mathbb{I}}\ (\text{ECP})$$

Figure 7. Inference rules for XCAP

Inference rules. To reason about TM programs in XCAP, just as we did for CAP (in Fig 4), we present a similar set of inference rules for well-formed programs, code heaps, and instruction sequences in Fig 7. Other than the differences in the assertion language, we only modified the PROG rule and the JMP rule, and added a new ECP rule for introducing new ECP propositions into assertions on the fly. For the unchanged rules, readers can refer to Sec 2.2 for explanations.

The difference for the PROG rule is minor but important. Here we use the global specification Ψ_{G} in the interpretation of assertions which may contain ECP propositions. For state $\mathbb{S}$ to satisfy assertion $\mathtt{a}$, we require a proof for the meta proposition $([\![\mathtt{a}]\!]_{\Psi_{G}}\ \mathbb{S})$. This is the only place in the XCAP inference rules where validity of assertions (with ECP propositions) needs to be established. All other rules only require subsumption between assertions.

For the JMP rule, instead of looking up the target code blocks' preconditions $\mathtt{a}'$ from the current (local) specification Ψ, we require the current precondition $\mathtt{a}$ to guarantee that the target code label $\mathbb{R}(\mathtt{r})$ is a valid ECP with $\mathtt{a}'$ as its precondition. Combined with the $([\![\mathtt{a}]\!]_{\Psi_{G}}\ \mathbb{S})$ condition established in the PROG rule, we can deduce that $\mathtt{a}'$ is indeed the one specified in Ψ_{G}.

If we call the JMP rule "consumer" of ECP propositions, then the ECP rule can be called "producer." It is essentially a "cast" rule—it allows us to introduce new ECP propositions $\mathsf{cptr}(\mathtt{f}, \Psi(\mathtt{f}))$ about any code label $\mathtt{f}$ found in the current code heap specification Ψ into the new assertion a'. This rule is often used when we move a constant code label into a register to create an ECP.

Combining the JMP and ECP rules, ECP knowledge can be built up by one module at the time of ECP creation and be passed around and get used for indirect jumps in other modules. For example, given the following code where we assume register $\mathtt{r}_{30}$ contains the return value and register $\mathtt{r}_{31}$ contains the return code pointer:

```
plus: add r30, r0, r1; // fun plus (a, b)
      jmp r31          //        = a + b

app2: mov r3, r0;      // fun app2(f, a, b)
      mov r0, r1;      //        = f(a, b)
      mov r1, r2;
      jmp r3
```

we can assign them with the following XCAP specifications and make safe (higher-order) function calls such as *app2(plus,1,2)*.

$$\begin{aligned}\{\mathsf{plus}\ \rightsquigarrow\ & \lambda(\mathbb{H},\mathbb{R}).\,\exists a,b,ret.\\ &\quad \langle \mathbb{R}(\mathtt{r}_0)=a\wedge\mathbb{R}(\mathtt{r}_1)=b\wedge\mathbb{R}(\mathtt{r}_{31})=ret\rangle\\ &\quad \mathbin{\wedge\!\!\!\wedge}\, \mathsf{cptr}(ret,\ \lambda(\mathbb{H}',\mathbb{R}').\,\langle\mathbb{R}'(\mathtt{r}_{30})=a+b\rangle)\}\end{aligned}$$

$$\begin{aligned}\{\mathsf{app2}\rightsquigarrow\ & \lambda(\mathbb{H},\mathbb{R}).\,\exists f,a,b,ret.\\ &\quad \langle \mathbb{R}(\mathtt{r}_1)=a\wedge\mathbb{R}(\mathtt{r}_2)=b\wedge\mathbb{R}(\mathtt{r}_0)=f\wedge\mathbb{R}(\mathtt{r}_{31})=ret\rangle\\ &\quad \mathbin{\wedge\!\!\!\wedge}\, \mathsf{cptr}(f,\ \lambda(\mathbb{H}',\mathbb{R}').\,\exists a',b',ret'.\\ &\qquad\qquad \langle\mathbb{R}'(\mathtt{r}_0)=a'\wedge\mathbb{R}'(\mathtt{r}_1)=b'\wedge\mathbb{R}'(\mathtt{r}_{31})=ret'\rangle\\ &\qquad\qquad \mathbin{\wedge\!\!\!\wedge}\, \mathsf{cptr}(ret',\ \lambda(\mathbb{H}'',\mathbb{R}'').\,\langle\mathbb{R}''(\mathtt{r}_{30})=a'+b'\rangle))\\ &\quad \mathbin{\wedge\!\!\!\wedge}\, \mathsf{cptr}(ret,\ \lambda(\mathbb{H}',\mathbb{R}').\,\langle\mathbb{R}'(\mathtt{r}_{30})=a+b\rangle)\}\end{aligned}$$

Soundness. The soundness of XCAP is proved in the same way as for CAP. We give the main lemmas and a proof sketch here; the fully mechanized proof in the Coq proof assistant can be found in our implementation [32].

Lemma 3.1 (Progress) If $\Psi_G \vdash \{\mathsf{a}\}\,\mathbb{P}$, then there exists a program $\mathbb{P}'$ such that $\mathbb{P} \longmapsto \mathbb{P}'$.

Proof Sketch: Suppose $\mathbb{P} = (\mathbb{C},\mathbb{S},\mathbb{I})$, by inversion we obtain $\Psi_G \vdash \{\mathsf{a}\}\,\mathbb{I}$. The proof is by induction over this derivation. ■

Lemma 3.2 (Preservation) If $\Psi_G \vdash \{\mathsf{a}\}\,\mathbb{P}$ and $\mathbb{P} \longmapsto \mathbb{P}'$ then there exists an assertion a' such that $\Psi_G \vdash \{\mathsf{a}'\}\,\mathbb{P}'$.

Proof Sketch: Suppose $\mathbb{P} = (\mathbb{C},\mathbb{S},\mathbb{I})$; by inversion we obtain $\Psi_G \vdash \mathbb{C}:\Psi_G$, $([\![\mathsf{a}]\!]_{\Psi_G}\ \mathbb{S})$, and $\Psi_G \vdash \{\mathsf{a}\}\,\mathbb{I}$. We do induction over derivation $\Psi_G \vdash \{\mathsf{a}\}\,\mathbb{I}$. The only interesting cases are the JMP and ECP rules.

For the JMP rule case, let $\mathbb{S}$ be $(\mathbb{H},\mathbb{R})$. By the implication $\mathsf{a} \Rightarrow (\lambda(\mathbb{H},\mathbb{R}).\,\mathsf{cptr}(\mathbb{R}(\mathtt{r}),\mathsf{a}'))$ and the interpretation of cptr it follows that $\mathsf{a}' = \Psi_G(\mathbb{R}(\mathtt{r}))$ and $\mathbb{R}(\mathtt{r}) \in \mathsf{dom}(\Psi_G)$. Then by the same code heap typing lemma as discussed in Sec 2.2, it follows that $\Psi_G \vdash \{\mathsf{a}'\}\,\mathbb{C}(\mathbb{R}(\mathtt{r}))$. Finally by $\mathsf{a} \Rightarrow \mathsf{a}'$ it follows that $\mathsf{a}'(\mathbb{H},\mathbb{R})$.

For the ECP case, by the code heap typing lemma and by $(\lambda\mathbb{S}.\,\mathsf{cptr}(\mathtt{f},\Psi_G(\mathtt{f})) \mathbin{\wedge\!\!\!\wedge} \mathsf{a}\ \mathbb{S}) \Rightarrow \mathsf{a}'$ it follows that $[\![\,\mathsf{a}'\ \mathbb{S}]\!]_{\Psi_G}$. Also we have $\Psi_G \vdash \{\mathsf{a}'\}\,\mathbb{I}$. Then we use the induction hypothesis to finish the proof. ■

Theorem 3.3 (Soundness) If $\Psi_G \vdash \{\mathsf{a}\}\,\mathbb{P}$, then for all natural number n, there exists a program $\mathbb{P}'$ such that $\mathbb{P} \longmapsto^n \mathbb{P}'$.

3.3 Discussion

The main idea of XCAP is to support Hoare-style reasoning of ECPs by extending the assertion language with a thin layer of syntax. Next we review XCAP using the criteria given in Sec 2.4. The following theorem presents a simple "type-preserving" translation from CAP to XCAP and shows that XCAP is at least as powerful as CAP. To avoid confusion, we use $\vdash_{CAP}$ and $\vdash_{XCAP}$ to represent CAP judgments (as defined in Fig 4) and XCAP ones (as defined in Fig 7). See our implementation [32] for the detailed proofs.

Theorem 3.4 (CAP to XCAP Translation)
We define the lifting of CAP assertions and specifications as:

$$\ulcorner \mathsf{a} \urcorner \triangleq \lambda\mathbb{S}.\,\langle \mathsf{a}\ \mathbb{S}\rangle$$

$$\text{and } \ulcorner\{\mathtt{f}_1 \rightsquigarrow \mathsf{a}_1,\ldots,\mathtt{f}_n \rightsquigarrow \mathsf{a}_n\}\urcorner \triangleq \{\mathtt{f}_1 \rightsquigarrow \ulcorner\mathsf{a}_1\urcorner,\ldots,\mathtt{f}_n \rightsquigarrow \ulcorner\mathsf{a}_n\urcorner\}.$$

1. If $\Psi_G \vdash_{CAP} \{\mathsf{a}\}\,\mathbb{P}$ then $\ulcorner\Psi_G\urcorner \vdash_{XCAP} \{\ulcorner\mathsf{a}\urcorner\}\,\mathbb{P}$;
2. if $\Psi_{IN} \vdash_{CAP} \mathbb{C}:\Psi$ then $\ulcorner\Psi_{IN}\urcorner \vdash_{XCAP} \{\mathbb{C}\}\,\ulcorner\Psi\urcorner$;
3. if $\Psi \vdash_{CAP} \{\mathsf{a}\}\,\mathbb{I}$ then $\ulcorner\Psi\urcorner \vdash_{XCAP} \{\ulcorner\mathsf{a}\urcorner\}\,\mathbb{I}$.

Proof Sketch: (1) and (2) are straight-forward and based on (3). For (3), as CAP's JMP rule only reasons about preconditions in the current Ψ, we use ECP rule on all pairs of code label and precondition in Ψ and use XCAP's JMP rule to finish the proof. ■

The specification written in XCAP assertion language is close to a typical Hoare assertion and thus is easy to write and reason about. From the user's perspective, there is no need to worry about the "meaning" of the ECP propositions because they are treated abstractly almost all the time.

XCAP is still lightweight because the lifted propositions $\langle p\rangle$ and their reasoning are shallowly embedded into the meta logic, which is the same as CAP. The added component of ECP propositions as well as other lifted connectives and quantifiers are simple syntactic constructs and do not involve complex constructions.

As we will show in later sections, the XCAP framework can be easily extended to support other language features and popular extensions of Hoare logic.

4. XCAP with Impredicative Polymorphism

The XCAP system presented earlier enjoys great expressive power from its underlying meta logic. As the mini examples have shown, features such as data polymorphism can be easily supported. However, to support modular verification, when composing specifications, it is important to be able to abstract out and quantify over (part of) the specifications themselves. This is especially important for ECPs, since very often the specification for the target code is only partially disclosed to the callers. In this section we show how to support this kind of **impredicative polymorphism** in XCAP.

Impredicative polymorphism can be easily supported in type systems. Take TAL [27] as an example, it allows quantifications over value types, which correspond to value predicates in XCAP. Since XCAP predicates are much more flexible than types, we choose to support universal and existential quantifications over arbitrary extended predicates of type $A \to PropX$ (where A does not contain $PropX$). We reuse the quantifiers defined in XCAP in Sec 3 and write **impredicative extended propositions** as follows:

$$\forall\!\!\!\forall \alpha\!:\!A \to PropX.\,\mathtt{P} \qquad \text{and} \qquad \exists\!\!\!\exists \alpha\!:\!A \to PropX.\,\mathtt{P}.$$

In the implementation of these impredicative quantifiers, the HOAS technique used for predicative ones no longer works because of the negative-occurrence restriction for inductive definitions. We use the de Bruijn notations [14] to encode the impredicative quantifiers. For more details, see Appendix A.

The next task is to find a way to establish the **validity of impredicative extended propositions**. One obvious idea is to take the previously defined interpretation function in Fig 6 and directly apply it on the impredicative quantification cases as follows:

$\boxed{\Gamma \vdash_\Psi \mathtt{P}}$ ***(Validity of Extended Propositions)*** *(The following presentation omits the Ψ in judgment $\Gamma \vdash_\Psi$ P.)*

$$(\mathit{env})\quad \Gamma ::= \cdot \mid \Gamma, \mathtt{P} \qquad \frac{\mathtt{P} \in \Gamma}{\Gamma \vdash \mathtt{P}}\ (\text{ENV}) \qquad \frac{p}{\Gamma \vdash \langle p \rangle}\ (\langle\rangle\text{-I}) \qquad \frac{\Gamma \vdash \langle p \rangle \qquad p \supset (\Gamma \vdash \mathtt{Q})}{\Gamma \vdash \mathtt{Q}}\ (\langle\rangle\text{-E}) \qquad \frac{\Psi(\mathtt{f}) = \mathtt{a}}{\Gamma \vdash \mathsf{cptr}(\mathtt{f}, \mathtt{a})}\ (\text{CP-I})$$

$$\frac{\Gamma \vdash \mathsf{cptr}(\mathtt{f}, \mathtt{a}) \qquad (\Psi(\mathtt{f}) = \mathtt{a}) \supset (\Gamma \vdash \mathtt{Q})}{\Gamma \vdash \mathtt{Q}}\ (\text{CP-E}) \qquad \frac{\Gamma \vdash \mathtt{P} \qquad \Gamma \vdash \mathtt{Q}}{\Gamma \vdash \mathtt{P} \mathbin{\wedge\!\!\!\wedge} \mathtt{Q}}\ (\mathbin{\wedge\!\!\!\wedge}\text{-I}) \qquad \frac{\Gamma \vdash \mathtt{P} \mathbin{\wedge\!\!\!\wedge} \mathtt{Q}}{\Gamma \vdash \mathtt{P}}\ (\mathbin{\wedge\!\!\!\wedge}\text{-E1}) \qquad \frac{\Gamma \vdash \mathtt{P} \mathbin{\wedge\!\!\!\wedge} \mathtt{Q}}{\Gamma \vdash \mathtt{Q}}\ (\mathbin{\wedge\!\!\!\wedge}\text{-E2})$$

$$\frac{\Gamma \vdash \mathtt{P}}{\Gamma \vdash \mathtt{P} \mathbin{\vee\!\!\!\vee} \mathtt{Q}}\ (\mathbin{\vee\!\!\!\vee}\text{-I1}) \qquad \frac{\Gamma \vdash \mathtt{Q}}{\Gamma \vdash \mathtt{P} \mathbin{\vee\!\!\!\vee} \mathtt{Q}}\ (\mathbin{\vee\!\!\!\vee}\text{-I2}) \qquad \frac{\Gamma \vdash \mathtt{P} \mathbin{\vee\!\!\!\vee} \mathtt{Q} \qquad \Gamma, \mathtt{P} \vdash \mathtt{R} \qquad \Gamma, \mathtt{Q} \vdash \mathtt{R}}{\Gamma \vdash \mathtt{R}}\ (\mathbin{\vee\!\!\!\vee}\text{-E}) \qquad \frac{\Gamma, \mathtt{P} \vdash \mathtt{Q}}{\Gamma \vdash \mathtt{P} \twoheadrightarrow \mathtt{Q}}\ (\twoheadrightarrow\text{-I})$$

$$\frac{\Gamma \vdash \mathtt{P} \twoheadrightarrow \mathtt{Q} \qquad \Gamma \vdash \mathtt{P}}{\Gamma \vdash \mathtt{Q}}\ (\twoheadrightarrow\text{-E}) \qquad \frac{\Gamma \vdash \mathtt{P}[B/x] \qquad \forall B{:}A}{\Gamma \vdash \forall\!\!\!\forall x{:}A.\mathtt{P}}\ (\forall\!\!\!\forall\text{-I1}) \qquad \frac{\Gamma \vdash \forall\!\!\!\forall x{:}A.\mathtt{P} \qquad B{:}A}{\Gamma \vdash \mathtt{P}[B/x]}\ (\forall\!\!\!\forall\text{-E1}) \qquad \frac{B{:}A \qquad \Gamma \vdash \mathtt{P}[B/x]}{\Gamma \vdash \exists\!\!\!\exists x{:}A.\mathtt{P}}\ (\exists\!\!\!\exists\text{-I1})$$

$$\frac{\Gamma \vdash \exists\!\!\!\exists x{:}A.\mathtt{P} \qquad \Gamma, \mathtt{P}[B/x] \vdash \mathtt{Q} \qquad \forall B{:}A}{\Gamma \vdash \mathtt{Q}}\ (\exists\!\!\!\exists\text{-E1}) \qquad \frac{\Gamma \vdash \mathtt{P}[\mathtt{a}/\alpha] \qquad \forall \mathtt{a}{:}A \to \mathit{PropX}}{\Gamma \vdash \forall\!\!\!\forall \alpha{:}A \to \mathit{PropX}.\mathtt{P}}\ (\forall\!\!\!\forall\text{-I2}) \qquad \frac{\mathtt{a}{:}A \to \mathit{PropX} \qquad \Gamma \vdash \mathtt{P}[\mathtt{a}/\alpha]}{\Gamma \vdash \exists\!\!\!\exists \alpha{:}A \to \mathit{PropX}.\mathtt{P}}\ (\exists\!\!\!\exists\text{-I2})$$

Figure 8. Validity rules for impredicative extended propositions

$$[\![\forall\!\!\!\forall \alpha{:}A \to \mathit{PropX}.\mathtt{P}]\!]_\Psi \triangleq \forall \mathtt{a}{:}A \to \mathit{PropX}.\,[\![\mathtt{P}[\mathtt{a}/\alpha]]\!]_\Psi$$
$$[\![\exists\!\!\!\exists \alpha{:}A \to \mathit{PropX}.\mathtt{P}]\!]_\Psi \triangleq \exists \mathtt{a}{:}A \to \mathit{PropX}.\,[\![\mathtt{P}[\mathtt{a}/\alpha]]\!]_\Psi$$

Unfortunately (but not surprisingly), the recursive call parameter $\mathtt{P}[\mathtt{a}/\alpha]$ may be larger than the original ones as a can bring in unbounded new sub-formulas. The interpretation function no longer terminates, and thus is not definable in the meta logic.

Our solution is to define the interpretation of extended propositions as the set of inductively defined validity rules shown in Fig 8. The judgment, $\Gamma \vdash_\Psi \mathtt{P}$, means that P is valid under environment Γ (which is a set of extended propositions) and code heap specification Ψ. An extended proposition is valid if it is in the environment. Constructors of extended propositions have their introduction and elimination rules. The introduction rules of lifted proposition $\langle p \rangle$ and ECP proposition $\mathsf{cptr}(\mathtt{f}, \mathtt{a})$ require that p and $\Psi(\mathtt{f}) = \mathtt{a}$ be valid in the meta logic. Their elimination rules allow full meta-implication power in constructing derivations of validity of the new extended propositions. The rules for other constructors are standard and require little explanation.

The **interpretation of extended propositions** can be now be simply defined as their validity under the empty environment.

$$[\![\mathtt{P}]\!]_\Psi \triangleq \cdot \vdash_\Psi \mathtt{P}$$

Given the above definitions of interpretation and validity, we have proved the following soundness theorem (with respect to CiC/Coq) using the syntactic strong normalization proof method by Pfenning [35]. For proof details, see Appendix B.

Theorem 4.1 (Soundness of *PropX* Interpretation)

1. If $[\![\langle p \rangle]\!]_\Psi$ then p;
2. if $[\![\mathsf{cptr}(\mathtt{f}, \mathtt{a})]\!]_\Psi$ then $\Psi(\mathtt{f}) = \mathtt{a}$;
3. if $[\![\mathtt{P} \mathbin{\wedge\!\!\!\wedge} \mathtt{Q}]\!]_\Psi$ then $[\![\mathtt{P}]\!]_\Psi$ and $[\![\mathtt{Q}]\!]_\Psi$;
4. if $[\![\mathtt{P} \mathbin{\vee\!\!\!\vee} \mathtt{Q}]\!]_\Psi$ then either $[\![\mathtt{P}]\!]_\Psi$ or $[\![\mathtt{Q}]\!]_\Psi$;
5. if $[\![\mathtt{P} \twoheadrightarrow \mathtt{Q}]\!]_\Psi$ and $[\![\mathtt{P}]\!]_\Psi$ then $[\![\mathtt{Q}]\!]_\Psi$;
6. if $[\![\forall\!\!\!\forall x{:}A.\mathtt{P}]\!]_\Psi$ and $B{:}A$ then $[\![\mathtt{P}[B/x]]\!]_\Psi$;
7. if $[\![\exists\!\!\!\exists x{:}A.\mathtt{P}]\!]_\Psi$ then there exists $B{:}A$ such that $[\![\mathtt{P}[B/x]]\!]_\Psi$;
8. if $[\![\forall\!\!\!\forall \alpha{:}A \to \mathit{PropX}.\mathtt{P}]\!]_\Psi$ and $\mathtt{a}{:}A \to \mathit{PropX}$ then $[\![\mathtt{P}[\mathtt{a}/\alpha]]\!]_\Psi$;
9. if $[\![\exists\!\!\!\exists \alpha{:}A \to \mathit{PropX}.\mathtt{P}]\!]_\Psi$ then there exists $\mathtt{a}{:}A \to \mathit{PropX}$ such that $[\![\mathtt{P}[\mathtt{a}/\alpha]]\!]_\Psi$.

Corollary 4.2 (Consistency) $[\![\langle \mathsf{False} \rangle]\!]_\Psi$ is not provable.

To make impredicative extended propositions easy to use, in the implementation of *PropX* and its interpretation, we define additional concrete syntax and proof tactics to hide the de Bruijn representation and the interpretation detail. A user can mostly manipulate *PropX* objects in the same way as with *Prop* objects.

Inference rules and soundness. We change the indirect jump rule JMP from the one presented in Fig 7 to the following:

$$\frac{\mathtt{a} \Rightarrow (\lambda(\mathbb{H}, \mathbb{R}).\ \exists\!\!\!\exists \mathtt{a}'.\ (\mathtt{a}'\ (\mathbb{H}, \mathbb{R}) \mathbin{\wedge\!\!\!\wedge} \mathsf{cptr}(\mathbb{R}(\mathtt{r}), \mathtt{a}')))}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{jmp}\ \mathtt{r}}\ (\text{JMP}).$$

The existential quantification over the assertion $\mathtt{a}'$ (for the target code block) is moved from (implicitly) over the whole rule to be after the assertion subsumption ($\Rightarrow$). This change is important to support polymorphic code—the target assertion $\mathtt{a}'$ can now depend on the current assertion a.

All other inference rules of XCAP remain unchanged. The soundness of XCAP inference rules (Theorem 3.3) and the CAP to XCAP translation (Theorem 3.4) only need trivial modifications in the case of indirect jump. We do not restate them here. This impredicative version of XCAP has been implemented in Coq proof assistant (see our implementation [32] for details).

With impredicative quantifications, ECP can now be specified and used with great flexibility. For example, the *app*2 function in previous section can now be assigned with the following more general specification. Instead of being restricted to an argument with the "plus" functionality, any functions that take two arguments a and b and return a value satisfying (unrestricted) assertion $\mathtt{a}_{ret}(a, b)$ can be passed to *app*2.

$$\begin{aligned}
\{\mathsf{app2} \rightsquigarrow \lambda(\mathbb{H}, \mathbb{R}).\ &\exists\!\!\!\exists f, a, b, ret, \mathtt{a}_{ret}.\\
&\langle \mathbb{R}(\mathtt{r}_1) = a \wedge \mathbb{R}(\mathtt{r}_2) = b \wedge \mathbb{R}(\mathtt{r}_0) = f \wedge \mathbb{R}(\mathtt{r}_{31}) = ret \rangle\\
&\mathbin{\wedge\!\!\!\wedge} \mathsf{cptr}(f,\ \lambda(\mathbb{H}', \mathbb{R}').\ \exists\!\!\!\exists a', b', ret'.\\
&\qquad \langle \mathbb{R}'(\mathtt{r}_0) = a' \wedge \mathbb{R}'(\mathtt{r}_1) = b' \wedge \mathbb{R}'(\mathtt{r}_{31}) = ret' \rangle\\
&\qquad \mathbin{\wedge\!\!\!\wedge} \mathsf{cptr}(ret',\ \mathtt{a}_{ret}(a', b')))\\
&\mathbin{\wedge\!\!\!\wedge} \mathsf{cptr}(ret,\ \mathtt{a}_{ret}(a, b))\}
\end{aligned}$$

Subtyping on ECP propositions. The ECP proposition has a very rigid interpretation. To establish the validity of $\mathsf{cptr}(\mathtt{f}, \mathtt{a})$, $\Psi(\mathtt{f})$ must be "equal" to a. This is simple for the system, but is restrictive in usage and differs from typical type systems where subtyping can be used to relax code types. With support of impredicative quantifications, instead of directly using cptr, we can define a more flexible predicate for ECPs:

$$\mathsf{codeptr}(\mathtt{f},\mathsf{a}) \triangleq \exists\,\mathsf{a}'.\,(\mathsf{cptr}(\mathtt{f},\mathsf{a}') \wedge\!\!\!\wedge \forall\!\!\!\forall\mathbb{S}.\,\mathsf{a}\;\mathbb{S} \twoheadrightarrow \mathsf{a}'\;\mathbb{S}).$$

We can define the following subtyping lemma for ECP predicates.

Lemma 4.3 (Subtyping of ECP Propositions)
If $[\![\mathsf{codeptr}(\mathtt{f},\mathsf{a}')]\!]_\Psi$ and $[\![\forall\!\!\!\forall\mathbb{S}.\,\mathsf{a}\;\mathbb{S} \twoheadrightarrow \mathsf{a}'\;\mathbb{S}]\!]_\Psi$ then $[\![\mathsf{codeptr}(\mathtt{f},\mathsf{a})]\!]_\Psi$.

Proof: From $[\![\mathsf{codeptr}(\mathtt{f},\mathsf{a}')]\!]_\Psi$ it follows that

$$[\![\exists\,\mathsf{a}''.\,(\mathsf{cptr}(\mathtt{f},\mathsf{a}'') \wedge\!\!\!\wedge \forall\!\!\!\forall\mathbb{S}.\,\mathsf{a}'\;\mathbb{S} \twoheadrightarrow \mathsf{a}''\;\mathbb{S})]\!]_\Psi.$$

By the soundness of interpretation theorem it follows that

$$\exists \mathsf{a}''.\,[\![\mathsf{cptr}(\mathtt{f},\mathsf{a}'')]\!]_\Psi \wedge [\![\forall\!\!\!\forall\mathbb{S}.\,\mathsf{a}'\;\mathbb{S} \twoheadrightarrow \mathsf{a}''\;\mathbb{S}]\!]_\Psi.$$

Using the $\forall\!\!\!\forall$-I1, $\forall\!\!\!\forall$-E1, $\twoheadrightarrow$-I, and $\twoheadrightarrow$-E rules it follows that

$$[\![\forall\!\!\!\forall\mathbb{S}.\,\mathsf{a}\;\mathbb{S} \twoheadrightarrow \mathsf{a}''\;\mathbb{S}]\!]_\Psi.$$

Using the $\wedge\!\!\!\wedge$-I and $\exists$-I2 rules it follows that

$$[\![\exists\,\mathsf{a}''.\,(\mathsf{cptr}(\mathtt{f},\mathsf{a}'') \wedge\!\!\!\wedge \forall\!\!\!\forall\mathbb{S}.\,\mathsf{a}\;\mathbb{S} \twoheadrightarrow \mathsf{a}''\;\mathbb{S})]\!]_\Psi.$$

Which is $[\![\mathsf{codeptr}(\mathtt{f},\mathsf{a})]\!]_\Psi$. ■

Discussion. In Fig 8 we have only included the introduction rules for the two impredicative quantifiers. This could cause confusion because from the logic perspective, missing the two elimination rules would raise questions related to the completeness of the logic. However, despite its name, *PropX* is **not** designed to be a general (complete) logic; it is purely a level of syntax laid upon the meta logic. While its expressive power comes from the lifted propositions $\langle p\rangle$, the modular handling of ECPs and impredicative polymorphism follows syntactic types.

To certify the examples in this paper (or any polymorphic TAL programs), what we need is to establish the assertion subsumption relation $\Rightarrow$ between XCAP assertions. According to its definition, assertion subsumption is merely a meta-implication between validities of XCAP propositions. Although in certain cases it is possible to first do all the subsumption reasoning in *PropX* and prove $[\![\mathtt{P} \twoheadrightarrow \mathtt{Q}]\!]_\Psi$, and then obtain the subsumption proof $[\![\mathtt{P}]\!]_\Psi \supset [\![\mathtt{Q}]\!]_\Psi$ by Theorem 4.1, it is not always possible due to the lack of completeness for *PropX*, and is not the way *PropX* should be used. Instead, one can always follow the diagram below in proving subsumption relations (we use the impredicative existential quantifier as an example):

$$\begin{tikzcd}
{[\![\exists\alpha.\,\mathtt{P}]\!]_\Psi} \arrow[r, dashed, "implication"] \arrow[d, Rightarrow, "Theorem\ 4.1"] & {[\![\exists\alpha.\,\mathtt{Q}]\!]_\Psi} \\
\exists \mathsf{a}.\,[\![\mathtt{P}[\mathsf{a}/\alpha]]\!]_\Psi \arrow[r, "meta\text{-}implication"] & \exists \mathsf{a}.\,[\![\mathtt{Q}[\mathsf{a}/\alpha]]\!]_\Psi \arrow[u, Rightarrow, "rule\ \exists\text{-I2}"']
\end{tikzcd}$$

To prove the intended "implication" relation (the top one), we first use Theorem 4.1 to turn the source proposition's existential quantification into the meta one, from which we can do (flexible) meta implications. Then we reconstruct the existential quantification of the target proposition via the introduction rule. This way, the construction of subsumption proof in meta logic does not require the reasoning at the *PropX* level.

In fact, the subtyping relation found in TAL can be simulated by the subsumption relation in XCAP (with only the introduction rules for the two impredicative quantifiers). What the missing "elimination rules" would add here is the ability to support a notion of "higher-order subtyping" between "impredicative types", which does not appear in practical type systems such as TAL, FLINT, or ML. Although it could be nice to include such feature in XCAP, we did not do so since that would require a complex semantic strong normalization proof instead of the simple syntactic one used for Theorem 4.1 (see Appendix B).

5. Solving Reynolds's ECP Problem

Separation logic [37] is a recent Hoare-logic framework designed for reasoning about shared mutable data structures. Reynolds [37] listed supporting ECPs as a major open problem for separation logic. In this section, we show how to solve this problem in the XCAP framework (with impredicative polymorphism).

XCAP directly supports separation logic specifications and reasoning by defining their constructs and inference rules in the assertion language and meta logic as macros and lemmas. For example, the following are some separation logic primitives defined over the data heap (assuming $\uplus$ is the disjoint union):

$$\begin{aligned}
\mathsf{emp} &\triangleq \lambda\mathbb{H}.\,\langle \mathsf{dom}(\mathbb{H}) = \{\}\rangle \\
\mathtt{l} \mapsto \mathtt{w} &\triangleq \lambda\mathbb{H}.\,\langle \mathsf{dom}(\mathbb{H}) = \{\mathtt{l}\} \wedge \mathbb{H}(\mathtt{l}) = \mathtt{w}\rangle \\
\mathtt{l} \mapsto \_ &\triangleq \lambda\mathbb{H}.\,\langle \exists \mathtt{w}.\,(\mathtt{l} \mapsto \mathtt{w}\;\mathbb{H})\rangle \\
\mathsf{a}_1 * \mathsf{a}_2 &\triangleq \lambda\mathbb{H}.\,\exists\mathbb{H}_1,\mathbb{H}_2.\,\langle \mathbb{H}_1 \uplus \mathbb{H}_2 = \mathbb{H}\rangle \wedge\!\!\!\wedge \mathsf{a}_1\;\mathbb{H}_1 \wedge\!\!\!\wedge \mathsf{a}_2\;\mathbb{H}_2 \\
\mathtt{l} \mapsto \mathtt{w}_1,\ldots,\mathtt{w}_n &\triangleq \mathtt{l} \mapsto \mathtt{w}_1 * \ldots * \mathtt{l}{+}n{-}1 \mapsto \mathtt{w}_n
\end{aligned}$$

The frame rule can be defined as lemmas (derived rules) in XCAP:

$$\frac{\mathsf{a} \Rightarrow (\mathsf{a}' \circ \mathtt{Next}_\mathtt{c})}{(\mathsf{a} * \mathsf{a}'') \Rightarrow ((\mathsf{a}' * \mathsf{a}'') \circ \mathtt{Next}_\mathtt{c})} \;(\text{FRAME-INSTR})$$

$$\frac{\{\mathtt{f}_1 \rightsquigarrow \mathsf{a}_1,\ldots,\mathtt{f}_n \rightsquigarrow \mathsf{a}_n\} \vdash \{\mathsf{a}\}\,\mathbb{I}}{\{\mathtt{f}_1 \rightsquigarrow \mathsf{a}_1 * \mathsf{a}',\ldots,\mathtt{f}_n \rightsquigarrow \mathsf{a}_n * \mathsf{a}'\} \vdash \{\mathsf{a} * \mathsf{a}'\}\,\mathbb{I}} \;(\text{FRAME-ISEQ})$$

ECP formulas can appear freely in these assertions and rules, thus it is very convenient to write specifications and reason about shared mutable data structures and embedded code pointers simultaneously. So our XCAP framework easily carries to separation logic.

Example: destructive list-append function in CPS. To demonstrate the above point, we verify a destructive version of the list-append example which Reynolds [37] used to define the ECP open problem. Following Reynolds, our destructive list-append function is written in continuation passing style (CPS):

```
append(x, y, rk) =
        if x == NULL then rk(y)
        else let k(z) = ([x+1] := z; rk(x))
             in append([x+1], y, k)
```

Here the *append* function takes three arguments: two lists x and y and a return continuation rk. If x is an empty list, it calls rk with list y. Otherwise, it first creates a new continuation function k which takes an (appended) list z, makes list x's head node link to z, and passes the newly formed list (which is pointed to by x) to the return continuation rk. Variables x and rk form the closure environment for continuation function k. The *append* function then recursively calls itself with the tail of list x, list y, and the new continuation k. For node x, $[x]$ is its data and $[x+1]$ is the link to the next node.

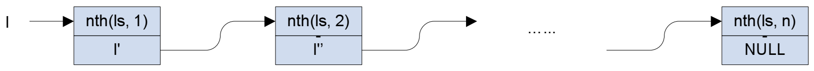
We do closure conversion and translate *append* into TM assembly code. The complete code, specification, and illustration of the destructive list-append function is in Fig 9. In the presentation, we often write a for assertion $\lambda(\mathbb{H},\mathbb{R}).\,\exists x_1\!:\!A_1,\ldots,x_n\!:\!A_n.\,\mathsf{a}$, so all free variables in a are existentially quantified right after the lambda abstraction. Formulas such as $(\mathsf{a} * \mathsf{a}'\;\mathbb{H})$ and $\mathbb{R}(\mathtt{r}) = \mathtt{w}$ are also simplified to be written as $\mathsf{a} * \mathsf{a}'$ and $\mathtt{r} = \mathtt{w}$.

Predicate $(\mathsf{list}\;ls\;\mathtt{l})$ describes a linked list pointed to by $\mathtt{l}$ where the data cell of each node stores the value in ls respectively. Here ls is a mathematical list where nil, $\mathtt{w} :: ls$ and $ls {+\!\!+} lt$ stand for the cases of empty list, cons, and append respectively. Predicate $(\mathsf{cont}\;\mathsf{a}_{env}\;ls\;\mathtt{f})$ requires $\mathtt{f}$ to point to a continuation code block which expects an environment of type a_{env} and a list which stores ls. Predicate $(\mathsf{clos}\;ls\;\mathtt{l})$ describes a continuation closure pointed to by $\mathtt{l}$; this closure is a pair (cnt, env) where cnt is a continuation

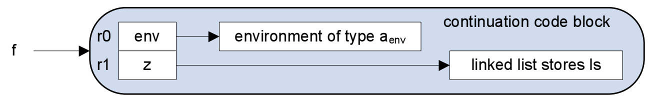
In this figure, all undefined identifiers are the names of (implicitly) existentially quantified variables.

$$\mathsf{list\ nil}\ \mathtt{l} \triangleq \mathsf{emp} \Lbag \langle \mathtt{l} = \mathsf{NULL} \rangle$$

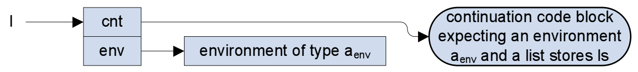
$$\mathsf{list}\ (\mathtt{w} :: ls)\ \mathtt{l} \triangleq \exists \mathtt{l}'.\, \mathtt{l} \mapsto \mathtt{w}, \mathtt{l}' * \mathsf{list}\ ls\ \mathtt{l}'$$

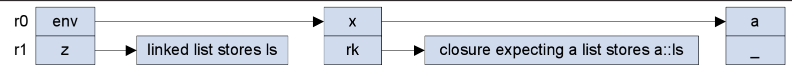


$$\mathsf{cont}\ \mathsf{a}_{env}\ ls\ \mathtt{f} \triangleq \mathsf{codeptr}(\mathtt{f},\ \lambda \mathbb{S}. \exists env, z.\ \langle \mathtt{r}_0 = env \wedge \mathtt{r}_1 = z \rangle \Lbag \mathsf{a}_{env}\ env * \mathsf{list}\ ls\ z)$$

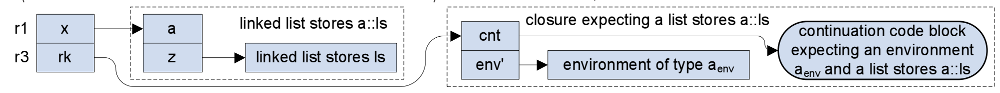


$$\mathsf{clos}\ ls\ \mathtt{l} \triangleq \exists \mathsf{a}_{env}, cnt, env.\ \mathtt{l} \mapsto cnt, env * \mathsf{a}_{env}\ env \Lbag \mathsf{cont}\ \mathsf{a}_{env}\ ls\ cnt$$

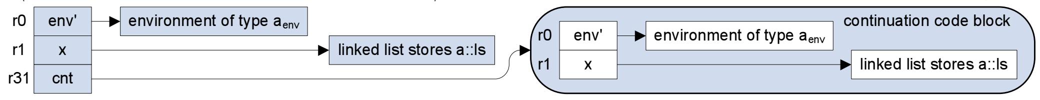

l
cnt
env
environment of type a_env
continuation code block expecting an environment a_env and a list stores ls



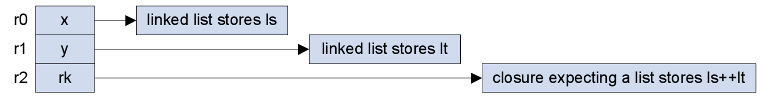
```
k:      ld r2, r0(0)     {⟨r0=env  ∧r1=z                          ⟩ ⩕ list ls z ∗ x↦a,_ ∗ clos (a::ls) rk ∗ env↦x,rk}
        ld r3, r0(1)     {⟨··················∧r2=x                 ⟩ ⩕ ································· ∗ env↦_,rk}
        free r0, 2       {⟨···························∧r3=rk        ⟩ ⩕ ································· ∗ env↦_,_}
        st r2(1), r1     {⟨          ·························     ⟩ ⩕ ·································}
        mov r1, r2       {⟨                 ·················      ⟩ ⩕ ······· ∗ x↦a,z ∗ ·············}
```


r1
x
r3
rk
a
z
linked list stores a::ls
linked list stores ls
closure expecting a list stores a::ls
cnt
env'
environment of type a_env
continuation code block expecting an environment a_env and a list stores a::ls

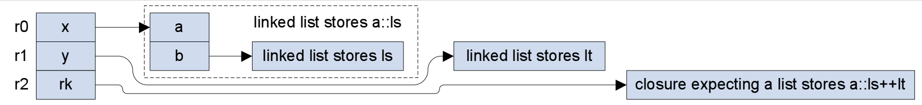
```
        ld r31, r3(0)    {⟨          r1=x          ∧r3=rk          ⟩ ⩕  list (a::ls) x   ∗ a_env env' ⩕ cont a_env (a::ls) cnt ∗ rk↦cnt,env'}
        ld r0, r3(1)     {⟨          ·······       ∧········∧r31=cnt⟩ ⩕ ····················································· ∗ rk↦_,env'}
        free r3, 2       {⟨r0=env'∧·······         ∧·················⟩ ⩕ ····················································· ∗ rk↦_,_}
```



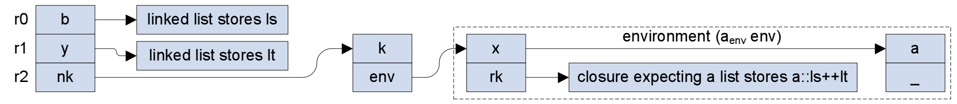
```
        jmp r31          {⟨r0=env'∧r1=x                   ∧r31=cnt⟩ ⩕  list (a::ls) x   ∗ a_env env' ⩕ cont a_env (a::ls) cnt}
```



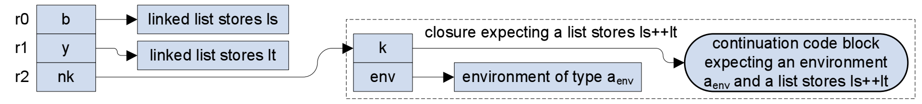
```
append: bgti r0, 0, else {⟨r0=x   ∧r1=y∧r2=rk               ⟩ ⩕ list ls x ∗ list lt y ∗ clos (ls++lt) rk}
                         // the following ‘‘then’’ branch is almost same as the second half of function k
        ld r31, r2(0)    {⟨           r1=y∧r2=rk            ⟩ ⩕          list lt y ∗ a_env env ⩕ cont a_env lt cnt ∗ rk↦cnt,env}
        ld r0, r2(1)     {⟨           ···············∧r31=cnt⟩ ⩕          ···································· ∗ rk↦_,env}
        free r2, 2       {⟨r0=env∧··························⟩ ⩕          ···································· ∗ rk↦_,_}
        jmp r31          {⟨r0=env∧r1=y            ∧r31=cnt⟩ ⩕          list lt y ∗ a_env env ⩕ cont a_env lt cnt}
```



```
else:   alloc r3, 2      {⟨r0=x∧r1=y∧r2=rk               ⟩ ⩕ list ls b ∗ list lt y ∗ clos (a::ls++lt) rk ∗ x↦a,b}
        st r3(0), r0     {⟨························∧r3=env⟩ ⩕ ··········································· ∗ env↦_,_}
        st r3(1), r2     {⟨································⟩ ⩕ ··········································· ∗ env↦x,_}
        ld r0, r0(1)     {⟨··············        ∧········⟩ ⩕ ··········································· ∗ env↦x,rk}
        alloc r2, 2      {⟨r0=b∧······           ∧········⟩ ⩕ ·································· ∗ x↦a,_ ∗ ··········}
        st r2(1), r3     {⟨··············∧r2=nk∧········⟩ ⩕ ·························································· ∗ nk↦_,_}
        movi r3, k       {⟨························       ⟩ ⩕ ·························································· ∗ nk↦_,env}
        st r2(0), r3     {⟨························∧r3=k  ⟩ ⩕ ·······································································}
```



```
        // (ecp) rule    {⟨r0=b∧r1=y∧r2=nk               ⟩ ⩕ list ls b ∗ list lt y ∗ clos (a::ls++lt) rk ∗ x↦a,_ ∗env↦x,rk ∗ nk↦k,env}
```


r0
b
linked list stores ls
r1
y
linked list stores lt
r2
nk
k
env
closure expecting a list stores ls++lt
environment of type a_env
continuation code block expecting an environment a_env and a list stores ls++lt

```
        jd append        {⟨r0=b∧r1=y∧r2=nk               ⟩ ⩕ list ls b ∗ list lt y ∗ clos (ls++lt) nk}
                         // where the a_env being packed is defined as a_env env ≜ clos (a::ls++lt) rk ∗ x↦a,_∗env↦x,rk
```

Figure 9. Complete code, specification, and illustration of destructive list append function in CPS

function pointer and *env* points to an environment for *cnt*. The environment predicate a_{env} is hidden inside the closure predicate.

In Fig 9 we list the precondition for each instruction on its right side. The instruction (or the comment on the left side) determines which well-formed instruction rule to use at each step. State diagrams are drawn before all the interesting steps.

Our specification of the *append* function guarantees that the return continuation *rk* will get a **correctly** appended list (*i.e.*, the list contents are exactly same as the two input lists). Furthermore, even though the function contains a large amount of heap allocation, mutation, and deallocation (which are used to build and destroy continuation closures and to append two lists on the fly), memory safety is **fully** guaranteed (no garbage, no illegal free operation). The complete Coq proof is also available in our implementation [32].

6. Related Work and Conclusion

Hoare logic. Earlier work such as Yu *et al* [45, 8] encountered the problem of "functional parameters" when checking the correctness of MC68020 object code. They first prove correctness of the program assuming some constraints about function parameters, and then apply the correctness theorem of the program repeatedly by substituting the functional parameters with concrete functions matching the imposed constraints. This approach is very close to CAP's syntactic approach to ECP as the global code heap specification is used when forming the "constraints", and was described by Yu *et al.* [45] as "extremely difficult."

Reynolds [37] identified the ECP problem for separation logic and described ECPs as "difficult to describe in the first-order world of Hoare logic." He speculated that a potential solution lies in marrying separation logic with continuation-passing style and by adding a reflection operator into the logic. The idea was only described briefly and informally. Sec 5 of this paper solved the ECP problem for separation logic.

In addition to the related work presented in Sections 1 and 2.3, there are many other attempts made toward the ECP problem, though they all use some forms of stratification or indexing. Recently, borrowing ideas from typed π-calculus, Honda and Yoshida [23] presented a formal reasoning system for a polymorphic version of PCF—they can support high-order functions but their assertion language requires testing whether a computation is a bottom (*i.e.*, termination). Their subsequent work [24] built a compositional program logic which captures observational semantics (standard contextual congruence) of a basic high-level programming language, based on the suggestions from the encoding of the language into the pi-calculus. Another subsequent work [7] added aliasing pointers to their framework. It is unclear whether their framework can be adapted to machine-level languages.

Proof-carrying code. Other than the semantic FPCC system discussed in Sec 2.3, similar ECP problems surfaced in some other PCC systems as well. In particular, Configurable PCC (CPCC) systems, as proposed by Necula and Schneck [31], statically check program safety using symbolic predicates which are called "continuations." For checking the safety of an indirect jump instruction which transfers the program control given a code pointer, a trusted "decoder" generates an "indirect continuation" whose safety needs to be verified; this continuation is indirect because the target address cannot be determined by the decoder statically. For verification purpose, an untrusted "VCGen extension" is responsible for proposing some "direct continuations" (direct meaning that the target addresses are known statically) whose safety implies the safety of the "indirect continuation" given by the decoder. In practice, the extension works by listing all the possible values of the code pointer (essentially replacing the code pointer in the continuations with all concrete functions that it could stand for), which requires whole-program analysis and is same as CAP's syntactic approach to ECP.

Chang *et al* [9] presented a refined CPCC system in which "local invariants" refine "continuations." A local invariant essentially consists of two related components—an "assumption" of the current state and a list of "progress continuations" which are used for handling code pointers. To allow the VCGen extension to manipulate predicates using first-order logic, only a syntactically restricted form of invariants are used. Although this is necessary for automatic proof construction for type-safety, it is insufficient in handling embedded code pointers in general. As a result, these local invariants are only used to handle more gracefully certain fixed patterns of code pointers, such as return pointers. Other situations, such as virtual dispatch, would still require whole-program analysis for the VCGen extension to discharge the decoder's indirect continuations. In particular, it is unclear how this approach extends to support arbitrary safety policies and nested continuations.

TAL and assembly language with state logic. Both TAL [27] and assembly language with state logic (SL) [2] also use syntactic techniques to support ECPs, but they are rather different from the CAP-based Hoare-logic systems. TAL types are simple syntactic entities while CAP assertions are general-purpose logical predicates over the entire machine state. SL formulas are more expressive than TAL types but SL is still a specialized logic for reasoning about adjacency, separation, and aliasing of memory blocks.

CAP specifications and deduction rules precisely track the machine-state changes at each instruction (based on weakest preconditions or strongest postconditions) and rely on semantic consequence relation to support assertion subsumption. In other words, assertion subsumption in CAP is implemented merely as logical implications over extended propositions in the meta logic. TAL and SL, on the other hand, use pure syntactic subsumption (or subtyping) rules; TAL and SL specifications are not expressive enough to accurately track arbitrary state changes. Naively adapting the TAL and SL framework to CAP would require us to axiomatize the entire general-purpose predicate logic with inductive definitions (as in Coq [40]) inside our mechanized meta logic. We didn't take this approach because it requires huge amount of work and is definitely nontrivial and probably not even practical.

SL can also benefit from using the "semantic" subsumption relation as we designed for XCAP, essentially by reasoning about the SL formula sub-typing directly using the model built in a mechanized meta logic. Under this setup, SL formulas become user-defined logical predicates; all the "syntactic" SL logical deduction rules become lemmas (thus are no longer necessary). It would result in a simpler, more extensible, and more expressive system.

Index-based approaches. In Sections 1 and 2.3, we have given a detailed comparison between our XCAP framework and the index-based approach [6, 3, 39]. One remaining question is on how to compare the expressive power of these two systems. To address this formally, we need to first formalize an assembly language using the index-based approach based on step-counting. This is a big and nontrivial task itself—even the Princeton FPCC group has not written any such paper (this fact itself shows the complexity of indexing): Appel and McAllester [6] does not support memory updates; Amal Ahmed's excellent PhD thesis [3] provides a formalization of the semantic approach but only at the high level for a lambda-calculus-like language; the actual FPCC proofs are still not publicly available. We will thus leave this question as future work.

Conclusion. We presented a simple but powerful technique for solving the ECP problem for Hoare logic in the context of certified assembly programming. We show how to combine semantic consequence relation (for assertion subsumption) with syntactic proof techniques. The result is a new powerful framework that can perform modular reasoning on embedded code pointers while still retaining the expressiveness of Hoare logic. Our new framework can be applied to support other language features and is orthogonal to other Hoare-logic extensions such as separation logic [37] and CCAP [44]. We plan to evolve it into a general but simple system for reasoning about machine-level programs.

Acknowledgments

We thank Hai Fang, Xinyu Feng, Nadeem A. Hamid, Andrew McCreight, Valery Trifonov, David Walker, Dachuan Yu, and anonymous referees for suggestions and comments on an earlier version of this paper. This research is based on work supported in part by gifts from Intel and Microsoft, and NSF grants CCR-0208618 and CCR-0524545. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] A. Ahmed, L. Jia, and D. Walker. Reasoning about hierarchical storage. In *Proc. 18th IEEE Symposium on Logic in Computer Science*, pages 33–44, June 2003.

[2] A. Ahmed and D. Walker. The logical approach to stack typing. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 74–85. ACM Press, 2003.

[3] A. J. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

[4] A. W. Appel. Foundational proof-carrying code. In *Proc. 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–258, June 2001.

[5] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proc. 27th ACM Symposium on Principles of Programming Languages*, pages 243–253, Jan. 2000.

[6] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, Sept. 2001.

[7] M. Berger, K. Honda, and N. Yoshida. A logical analysis of aliasing in imperative higher-order functions. In *Proc. 10th ACM SIGPLAN International Conference on Functional Programming*, pages 280–293, Sept. 2005.

[8] R. S. Boyer and Y. Yu. Automated proofs of object code for a widely used microprocessor. *J. ACM*, 43(1):166–192, 1996.

[9] B.-Y. E. Chang, G. C. Necular, and R. R. Schneck. Extensible code verification. Unpublished manuscript, 2003.

[10] J. Chen, D. Wu, A. W. Appel, and H. Fang. A provably sound tal for back-end optimization. In *Proc. 2003 ACM Conference on Programming Language Design and Implementation*, pages 208–219. ACM Press, 2003.

[11] C. Colby, P. Lee, G. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *Proc. 2000 ACM Conference on Programming Language Design and Implementation*, pages 95–107, New York, 2000. ACM Press.

[12] K. Crary. Toward a foundational typed assembly language. In *Proc. 30th ACM Symposium on Principles of Programming Languages*, page 198, Jan. 2003.

[13] K. Crary and J. C. Vanderwaart. An expressive, scalable type theory for certified code. In *Proc. 7th ACM SIGPLAN International Conference on Functional Programming*, pages 191–205, 2002.

[14] N. G. de Bruijn. Lambda calculus notation with nameless dummies. *Indagationes Mathematicae*, 34:381–392, 1972.

[15] X. Feng and Z. Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *Proc. 2005 International Conference on Functional Programming*, pages 254–267, Sept. 2005.

[16] R. W. Floyd. Assigning meaning to programs. *Communications of the ACM*, Oct. 1967.

[17] M. Gordon. A mechanized Hoare logic of state transitions. In A. W. Roscoe, editor, *A Classical Mind—Essays in Honour of C.A.R. Hoare*, pages 143–160. Prentice Hall, 1994.

[18] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[19] N. A. Hamid and Z. Shao. Interfacing hoare logic and type systems for foundational proof-carrying code. In *Proc. 17th International Conference on Theorem Proving in Higher Order Logics*, volume 3223 of *LNCS*, pages 118–135. Springer-Verlag, Sept. 2004.

[20] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. In *Proc. 17th Annual IEEE Symposium on Logic in Computer Science*, pages 89–100, July 2002.

[21] H. Herbelin, F. Kirchner, B. Monate, and J. Narboux. Faq about coq. `http://pauillac.inria.fr/coq/doc/faq.html#htoc38`.

[22] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, Oct. 1969.

[23] K. Honda and N. Yoshida. A compositional logic for polymorphic higher-order functions. In *Proc. 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 191–202, Aug. 2004.

[24] K. Honda, N. Yoshida, and M. Berger. An observationally complete program logic for imperative higher-order functions. In *Proc. 20th IEEE Symposium on Logic in Computer Science*, June 2005.

[25] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997.

[26] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Proc. 23rd ACM Symposium on Principles of Programming Languages*, pages 271–283. ACM Press, 1996.

[27] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *Proc. 25th ACM Symposium on Principles of Programming Languages*, pages 85–97. ACM Press, Jan. 1998.

[28] D. A. Naumann. Predicate transformer semantics of a higher-order imperative language with record subtyping. *Science of Computer Programming*, 41(1):1–51, 2001.

[29] G. Necula. Proof-carrying code. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 106–119, New York, Jan. 1997. ACM Press.

[30] G. Necula. *Compiling with Proofs*. PhD thesis, School of Computer Science, Carnegie Mellon Univ., Sept. 1998.

[31] G. C. Necula and R. R. Schneck. A sound framework for untrustred verification-condition generators. In *Proceedings of IEEE Symposium on Logic in Computer Science*, pages 248–260. IEEE Computer Society, July 2003.

[32] Z. Ni and Z. Shao. Implementation for certified assembly programming with embedded code pointers. `http://flint.cs.yale.edu/publications/xcap.html`, Oct. 2005.

[33] P. W. O'Hearn and R. D. Tennent. *Algol-Like Languages*. Birkhauser, Boston, 1997.

[34] C. Paulin-Mohring. Inductive definitions in the system Coq—rules and properties. In M. Bezem and J. Groote, editors, *Proc. TLCA*, volume 664 of *LNCS*. Springer-Verlag, 1993.

[35] F. Pfenning. Automated theorem proving. `http://www-2.cs.cmu.edu/~fp/courses/atp/`, Apr. 2004.

[36] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proc. 1988 ACM Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, 1988.

[37] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proc. 17th Annual IEEE Symposium on Logic in Computer Science*, 2002.

[38] Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou. A type system for certified binaries. In *Proc. 29th ACM Symposium on Principles of Programming Languages*, pages 217–232. ACM Press, Jan. 2002.

[39] G. Tan. *A Compositional Logic for Control Flow and its Application in Foundational Proof-Carrying Code*. PhD thesis, Princeton University, 2005.

[40] The Coq Development Team. The Coq proof assistant reference manual. The Coq release v8.0, Oct. 2005.

[41] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[42] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proc. 26th ACM Symposium on Principles of Programming Languages*, pages 214–227. ACM Press, 1999.

[43] D. Yu, N. A. Hamid, and Z. Shao. Building certified libraries for PCC: Dynamic storage allocation. *Science of Computer Programming*, 50(1-3):101–127, Mar. 2004.

[44] D. Yu and Z. Shao. Verification of safety properties for concurrent assembly code. In *Proc. 2004 International Conference on Functional Programming*, Sept. 2004.

[45] Y. Yu. *Automated Proofs of Object Code For A Widely Used Microprocessor*. PhD thesis, University of Texas at Austin, 1992.

A. Implementation

We have implemented TM, CAP, XCAP, the CAP to XCAP translation, impredicative XCAP, Separation logic embedded in XCAP, and the destructive list-append function example in Coq proof assistant. Our implementation is available at [32].

For the XCAP extended propositions in Sec 3, we define it and its interpretation function as below. The encoding uses higher-order abstract syntax (HOAS) [36] to represent extended predicates.

```
Inductive PropX : Type
    := cptr: Word -> (State -> PropX) -> PropX
     | prop: Prop                      -> PropX
     | andx: PropX -> PropX            -> PropX
     | orx : PropX -> PropX            -> PropX
     | impx: PropX -> PropX            -> PropX
     | allx: forall A, (A -> PropX)    -> PropX
     | extx: forall A, (A -> PropX)    -> PropX.

Definition CdHpSpec := Map Label (State -> PropX).

Fixpoint Itp (P:PropX) (Si:CdHpSpec) {struct P} : Prop
  := match P with
     | cptr l a => lookup Si l a
     | prop p   => p
     | andx P Q => Itp P Si /\ Itp Q Si
     | orx  P Q => Itp P Si \/ Itp Q Si
     | impx P Q => Itp P Si -> Itp Q Si
     | allx A P => forall x, Itp (P x) Si
     | extx A P => exists x, Itp (P x) Si
     end.
```

For the XCAP with impredicative polymorphism defined in Sec 4, the HOAS encoding of extended propositions no longer works. The positivity requirement in Coq inductive definition limits the type *A* of the quantified terms to be of lower level than PropX, which can not be used for impredicative quantifications. We use de Bruijn notations [14] to encode them, but keep using HOAS for all other constructors.

```
Inductive PropX : list Type -> Type :=
  var : forall L A, A                 -> PropX (A :: L)
| lift: forall L A, PropX L           -> PropX (A :: L)
| cptr: forall L, Word -> (State -> PropX L) -> PropX L
| prop: forall L, Prop                      -> PropX L
| andx: forall L, PropX L -> PropX L        -> PropX L
| orx : forall L, PropX L -> PropX L        -> PropX L
| impx: forall L, PropX L -> PropX L        -> PropX L
| allx: forall L A, (A -> PropX L)          -> PropX L
| extx: forall L A, (A -> PropX L)          -> PropX L
| allv: forall L A, PropX (L ++ A :: nil)   -> PropX L
| extv: forall L A, PropX (L ++ A :: nil)   -> PropX L.
```

The interpretation validity rules of extended propositions are defined as the following inductive definition.

```
Definition CdHpSpec := Map Label (State -> PropX nil).
Definition Env := list (PropX nil).

Inductive OK : Env -> CdHpSpec -> PropX nil -> Prop :=
| o_env   : forall E Si p, In p E ->          OK E Si p
| o_cptr_i: forall E Si l P,
            lookup Si l P ->   OK E Si (cptr nil l P)
| o_cptr_e: forall E Si l P q,
            OK E Si (cptr nil l P) ->
            (lookup Si l P -> OK E Si q) -> OK E Si q
| o_prop_i: forall E Si (p:Prop),
            p ->                 OK E Si (prop nil p)
| o_prop_e: forall E Si (p:Prop) q,
            OK E Si (prop nil p) ->
            (p -> OK E Si q) ->               OK E Si q
| ... .

Definition Itp P Si:= OK nil Si P.
```

B. Proof Structure of Theorem 4.1

In this section we give the proof structure of the soundness of *PropX* interpretation (Theorem 4.1). We follow the syntactic strong normalization proof methods in Pfenning [35]. We use structural induction in most of the proof. The full proof has been mechanized in Coq proof assistant and is available at [32].

The validity of extended propositions rules of form $\Gamma \vdash_{\Psi} \mathsf{P}$ (see Fig 8) are natural deduction rules. We classified them into the following two kind (and call them together as **normal natural validity rules**) as shown in Fig 10.

$\Gamma \vdash_{\Psi} \mathsf{P} \Uparrow$	Extended Proposition P has a normal deduction, and
$\Gamma \vdash_{\Psi} \mathsf{P} \downarrow$	Extended Proposition P is extracted from a hypothesis.

And define the **annotated natural deduction rules** by annotating each normal validity rules with a "+" symbol as $\Gamma \vdash^{+}_{\Psi} \mathsf{P} \Uparrow$ and $\Gamma \vdash^{+}_{\Psi} \mathsf{P} \downarrow$ and adding the following coercion rule.

$$\frac{\Gamma \vdash^{+}_{\Psi} \mathsf{P} \Uparrow}{\Gamma \vdash^{+}_{\Psi} \mathsf{P} \downarrow} \text{ (COER')}$$

We then define the **sequent style validity rules** of form $\Gamma \Longrightarrow_{\Psi} \mathsf{P}$ in Fig 11 and extend the sequent rules by annotating sequent judgments with a "+" as $\Gamma \Longrightarrow^{+}_{\Psi} \mathsf{P}$ and adding the cut rule.

$$\frac{\Gamma \Longrightarrow^{+}_{\Psi} \mathsf{P} \qquad \Gamma, \mathsf{P} \Longrightarrow^{+}_{\Psi} \mathsf{Q}}{\Gamma \Longrightarrow^{+}_{\Psi} \mathsf{Q}} \text{ (CUT)}$$

The strong normalization proof process is

$$\Gamma \vdash_{\Psi} \mathsf{P} \ \overset{B.3}{\longrightarrow} \ \Gamma \vdash^{+}_{\Psi} \mathsf{P} \Uparrow \ \overset{B.7}{\longrightarrow} \ \Gamma \Longrightarrow^{+}_{\Psi} \mathsf{P}$$
$$\overset{B.9}{\longrightarrow} \ \Gamma \Longrightarrow_{\Psi} \mathsf{P} \ \overset{B.4}{\longrightarrow} \ \Gamma \vdash_{\Psi} \mathsf{P} \Uparrow .$$

We first maps natural deduction derivations to that of a set of sequent validity rules with cut. Then we do cut-elimination in sequent rules, and map the new cut-free sequent derivation back

$\boxed{\Gamma \vdash_\Psi \mathtt{P} \Uparrow \qquad \Gamma \vdash_\Psi \mathtt{P} \downarrow}$ **(*Validity of Extended Propositions*)** *(The following rules omits the Ψ in judgments $\Gamma \vdash_\Psi P \Uparrow$ and $\Gamma \vdash_\Psi P \downarrow$.)*

$$\frac{\Gamma \vdash \mathtt{P} \downarrow}{\Gamma \vdash \mathtt{P} \Uparrow}\ (\text{COER}) \qquad \frac{\mathtt{P} \in \Gamma}{\Gamma \vdash \mathtt{P} \downarrow}\ (\text{ENV}) \qquad \frac{p}{\Gamma \vdash \langle p \rangle \Uparrow}\ (\langle\rangle\text{-I}) \qquad \frac{\Gamma \vdash \langle p \rangle \downarrow \qquad p \supset (\Gamma \vdash \mathtt{Q} \Uparrow)}{\Gamma \vdash \mathtt{Q} \Uparrow}\ (\langle\rangle\text{-E}) \qquad \frac{\Psi(\mathtt{f}) = \mathtt{a}}{\Gamma \vdash \mathsf{cptr}(\mathtt{f},\mathtt{a}) \Uparrow}\ (\text{CP-I})$$

$$\frac{\Gamma \vdash \mathsf{cptr}(\mathtt{f},\mathtt{a}) \downarrow \qquad (\Psi(\mathtt{f}) = \mathtt{a}) \supset (\Gamma \vdash \mathtt{Q} \Uparrow)}{\Gamma \vdash \mathtt{Q} \Uparrow}\ (\text{CP-E}) \qquad \frac{\Gamma \vdash \mathtt{P} \Uparrow \qquad \Gamma \vdash \mathtt{Q} \Uparrow}{\Gamma \vdash \mathtt{P} \wedge\!\!\wedge \mathtt{Q} \Uparrow}\ (\wedge\!\!\wedge\text{-I}) \qquad \frac{\Gamma \vdash \mathtt{P} \wedge\!\!\wedge \mathtt{Q} \downarrow}{\Gamma \vdash \mathtt{P} \downarrow}\ (\wedge\!\!\wedge\text{-E1}) \qquad \frac{\Gamma \vdash \mathtt{P} \wedge\!\!\wedge \mathtt{Q} \downarrow}{\Gamma \vdash \mathtt{Q} \downarrow}\ (\wedge\!\!\wedge\text{-E2})$$

$$\frac{\Gamma \vdash \mathtt{P} \Uparrow}{\Gamma \vdash \mathtt{P} \vee\!\!\vee \mathtt{Q} \Uparrow}\ (\vee\!\!\vee\text{-I1}) \qquad \frac{\Gamma \vdash \mathtt{Q} \Uparrow}{\Gamma \vdash \mathtt{P} \vee\!\!\vee \mathtt{Q} \Uparrow}\ (\vee\!\!\vee\text{-I2}) \qquad \frac{\Gamma \vdash \mathtt{P} \vee\!\!\vee \mathtt{Q} \downarrow \qquad \Gamma, \mathtt{P} \vdash \mathtt{R} \Uparrow \qquad \Gamma, \mathtt{Q} \vdash \mathtt{R} \Uparrow}{\Gamma \vdash \mathtt{R} \Uparrow}\ (\vee\!\!\vee\text{-E}) \qquad \frac{\Gamma, \mathtt{P} \vdash \mathtt{Q} \Uparrow}{\Gamma \vdash \mathtt{P} \twoheadrightarrow \mathtt{Q} \Uparrow}\ (\twoheadrightarrow\text{-I})$$

$$\frac{\Gamma \vdash \mathtt{P} \twoheadrightarrow \mathtt{Q} \downarrow \qquad \Gamma \vdash \mathtt{P} \Uparrow}{\Gamma \vdash \mathtt{Q} \downarrow}\ (\twoheadrightarrow\text{-E}) \qquad \frac{\Gamma \vdash \mathtt{P}[B/x] \Uparrow \quad \forall B{:}A}{\Gamma \vdash \forall\!\!\!\forall x{:}A.\mathtt{P} \Uparrow}\ (\forall\!\!\!\forall\text{-I1}) \qquad \frac{\Gamma \vdash \forall\!\!\!\forall x{:}A.\mathtt{P} \downarrow \quad B{:}A}{\Gamma \vdash \mathtt{P}[B/x] \downarrow}\ (\forall\!\!\!\forall\text{-E1}) \qquad \frac{B{:}A \quad \Gamma \vdash \mathtt{P}[B/x] \Uparrow}{\Gamma \vdash \exists\!\!\!\exists x{:}A.\mathtt{P} \Uparrow}\ (\exists\!\!\!\exists\text{-I1})$$

$$\frac{\Gamma \vdash \exists\!\!\!\exists x{:}A.\mathtt{P} \downarrow \qquad \Gamma, \mathtt{P}[B/x] \vdash \mathtt{Q} \Uparrow \quad \forall B{:}A}{\Gamma \vdash \mathtt{Q} \Uparrow}\ (\exists\!\!\!\exists\text{-E1}) \qquad \frac{\Gamma \vdash \mathtt{P}[\mathtt{a}/\alpha] \Uparrow \quad \forall \mathtt{a}{:}A \to \mathit{PropX}}{\Gamma \vdash \forall\!\!\!\forall \alpha{:}A \to \mathit{PropX}.\mathtt{P} \Uparrow}\ (\forall\!\!\!\forall\text{-I2}) \qquad \frac{\mathtt{a}{:}A \to \mathit{PropX} \quad \Gamma \vdash \mathtt{P}[\mathtt{a}/\alpha] \Uparrow}{\Gamma \vdash \exists\!\!\!\exists \alpha{:}A \to \mathit{PropX}.\mathtt{P} \Uparrow}\ (\exists\!\!\!\exists\text{-I2})$$

Figure 10. Normal natural deduction validity rules

$\boxed{\Gamma \Longrightarrow_\Psi \mathtt{P}}$ **(*Validity of Extended Propositions*)** *(The following rules omits the Ψ in judgment $\Gamma \Longrightarrow_\Psi P$.)*

$$\frac{p}{\Gamma \Longrightarrow \langle p \rangle}\ (\langle\rangle\text{-R}) \qquad \frac{p \supset (\Gamma, \langle p \rangle \Longrightarrow \mathtt{Q})}{\Gamma, \langle p \rangle \Longrightarrow \mathtt{Q}}\ (\langle\rangle\text{-L}) \qquad \frac{\Psi(\mathtt{f}) = \mathtt{a}}{\Gamma \Longrightarrow \mathsf{cptr}(\mathtt{f},\mathtt{a})}\ (\text{CP-R}) \qquad \frac{(\Psi(\mathtt{f}) = \mathtt{a}) \supset (\Gamma, \mathsf{cptr}(\mathtt{f},\mathtt{a}) \Longrightarrow \mathtt{Q})}{\Gamma, \mathsf{cptr}(\mathtt{f},\mathtt{a}) \Longrightarrow \mathtt{Q}}\ (\text{CP-L})$$

$$\frac{\Gamma \Longrightarrow \mathtt{P} \quad \Gamma \Longrightarrow \mathtt{Q}}{\Gamma \Longrightarrow \mathtt{P} \wedge\!\!\wedge \mathtt{Q}}\ (\wedge\!\!\wedge\text{-R}) \qquad \frac{\Gamma, \mathtt{P} \wedge\!\!\wedge \mathtt{Q}, \mathtt{P} \Longrightarrow \mathtt{R}}{\Gamma, \mathtt{P} \wedge\!\!\wedge \mathtt{Q} \Longrightarrow \mathtt{R}}\ (\wedge\!\!\wedge\text{-L1}) \qquad \frac{\Gamma, \mathtt{P} \wedge\!\!\wedge \mathtt{Q}, \mathtt{Q} \Longrightarrow \mathtt{R}}{\Gamma, \mathtt{P} \wedge\!\!\wedge \mathtt{Q} \Longrightarrow \mathtt{R}}\ (\wedge\!\!\wedge\text{-L2}) \qquad \frac{\Gamma \Longrightarrow \mathtt{P}}{\Gamma \Longrightarrow \mathtt{P} \vee\!\!\vee \mathtt{Q}}\ (\vee\!\!\vee\text{-R1}) \qquad \frac{\Gamma \Longrightarrow \mathtt{Q}}{\Gamma \Longrightarrow \mathtt{P} \vee\!\!\vee \mathtt{Q}}\ (\vee\!\!\vee\text{-R2})$$

$$\frac{\Gamma, \mathtt{P} \vee\!\!\vee \mathtt{Q}, \mathtt{P} \Longrightarrow \mathtt{R} \qquad \Gamma, \mathtt{P} \vee\!\!\vee \mathtt{Q}, \mathtt{Q} \Longrightarrow \mathtt{R}}{\Gamma, \mathtt{P} \vee\!\!\vee \mathtt{Q} \Longrightarrow \mathtt{R}}\ (\vee\!\!\vee\text{-L}) \qquad \frac{\Gamma, \mathtt{P} \Longrightarrow \mathtt{Q}}{\Gamma \Longrightarrow \mathtt{P} \twoheadrightarrow \mathtt{Q}}\ (\twoheadrightarrow\text{-R}) \qquad \frac{\Gamma, \mathtt{P} \twoheadrightarrow \mathtt{Q} \Longrightarrow \mathtt{P} \qquad \Gamma, \mathtt{P} \twoheadrightarrow \mathtt{Q}, \mathtt{P} \Longrightarrow \mathtt{R}}{\Gamma, \mathtt{P} \twoheadrightarrow \mathtt{Q} \Longrightarrow \mathtt{R}}\ (\twoheadrightarrow\text{-L})$$

$$\frac{\mathtt{P} \in \Gamma}{\Gamma \Longrightarrow \mathtt{P}}\ (\text{INIT}) \qquad \frac{\Gamma \Longrightarrow \mathtt{P}[B/x] \qquad \forall B{:}A}{\Gamma \Longrightarrow \forall\!\!\!\forall x{:}A.\mathtt{P}}\ (\forall\!\!\!\forall\text{-R1}) \qquad \frac{\Gamma, \forall\!\!\!\forall x{:}A.\mathtt{P}, \mathtt{P}[B/x] \Longrightarrow \mathtt{Q} \qquad B{:}A}{\Gamma, \forall\!\!\!\forall x{:}A.\mathtt{P} \Longrightarrow \mathtt{Q}}\ (\forall\!\!\!\forall\text{-L1}) \qquad \frac{B{:}A \qquad \Gamma \Longrightarrow \mathtt{P}[B/x]}{\Gamma \Longrightarrow \exists\!\!\!\exists x{:}A.\mathtt{P}}\ (\exists\!\!\!\exists\text{-R1})$$

$$\frac{\Gamma, \exists\!\!\!\exists x{:}A.\mathtt{P}, \mathtt{P}[B/x] \Longrightarrow \mathtt{Q} \qquad \forall B{:}A}{\Gamma, \exists\!\!\!\exists x{:}A.\mathtt{P} \Longrightarrow \mathtt{Q}}\ (\exists\!\!\!\exists\text{-L1}) \qquad \frac{\Gamma \Longrightarrow \mathtt{P}[\mathtt{a}/\alpha] \qquad \forall \mathtt{a}{:}A \to \mathit{PropX}}{\Gamma \Longrightarrow \forall\!\!\!\forall \alpha{:}A \to \mathit{PropX}.\mathtt{P}}\ (\forall\!\!\!\forall\text{-R2}) \qquad \frac{\mathtt{a}{:}A \to \mathit{PropX} \qquad \Gamma \Longrightarrow \mathtt{P}[\mathtt{a}/\alpha]}{\Gamma \Longrightarrow \exists\!\!\!\exists \alpha{:}A \to \mathit{PropX}.\mathtt{P}}\ (\exists\!\!\!\exists\text{-R2})$$

Figure 11. Sequent style validity rules

to a normal natural deduction derivation. Soundness of XCAP interpretation can then be proved since the last rule in a normal natural deduction derivation must be one of the introduction rules.

Theorem B.1 (Soundness of Normal Deductions)

1. If $\Gamma \vdash_\Psi \mathtt{P} \Uparrow$ then $\Gamma \vdash_\Psi \mathtt{P}$, and
2. if $\Gamma \vdash_\Psi \mathtt{P} \downarrow$ then $\Gamma \vdash_\Psi \mathtt{P}$.

Theorem B.2 (Soundness of Annotated Deductions)

1. If $\Gamma \vdash^+_\Psi \mathtt{P} \Uparrow$ then $\Gamma \vdash_\Psi \mathtt{P}$, and
2. if $\Gamma \vdash^+_\Psi \mathtt{P} \downarrow$ then $\Gamma \vdash_\Psi \mathtt{P}$.

Theorem B.3 (Completeness of Annotated Deductions)

1. If $\Gamma \vdash_\Psi \mathtt{P}$ then $\Gamma \vdash^+_\Psi \mathtt{P} \Uparrow$, and
2. if $\Gamma \vdash_\Psi \mathtt{P}$ then $\Gamma \vdash^+_\Psi \mathtt{P} \downarrow$.

Theorem B.4 (Soundness of Sequent Calculus)
If $\Gamma \Longrightarrow_\Psi \mathtt{P}$ then $\Gamma \vdash_\Psi \mathtt{P} \Uparrow$.

Theorem B.5 (Completeness of Sequent Derivations)

1. If $\Gamma \vdash_\Psi \mathtt{P} \Uparrow$ then $\Gamma \Longrightarrow_\Psi \mathtt{P}$, and
2. if $\Gamma \vdash_\Psi \mathtt{P} \downarrow$ and $\Gamma, \mathtt{P} \Longrightarrow_\Psi \mathtt{Q}$ then $\Gamma \Longrightarrow_\Psi \mathtt{Q}$.

Theorem B.6 (Soundness of Sequent Calculus with Cut)
If $\Gamma \Longrightarrow^+_\Psi \mathtt{P}$ then $\Gamma \vdash^+_\Psi \mathtt{P} \Uparrow$.

Theorem B.7 (Completeness of Sequent Calculus with Cut)

1. If $\Gamma \vdash^+_\Psi \mathtt{P} \Uparrow$ then $\Gamma \Longrightarrow^+_\Psi \mathtt{P}$, and
2. if $\Gamma \vdash^+_\Psi \mathtt{P} \downarrow$ and $\Gamma, \mathtt{P} \Longrightarrow^+_\Psi \mathtt{Q}$ then $\Gamma \Longrightarrow^+_\Psi \mathtt{Q}$.

Theorem B.8 (Admissibility of Cut)
If $\Gamma \Longrightarrow_\Psi \mathtt{P}$ and $\Gamma, \mathtt{P} \Longrightarrow_\Psi \mathtt{Q}$ then $\Gamma \Longrightarrow_\Psi \mathtt{Q}$.

Theorem B.9 (Cut Elimination)
If $\Gamma \Longrightarrow^+_\Psi \mathtt{P}$ then $\Gamma \Longrightarrow_\Psi \mathtt{P}$.

Theorem B.10 (Normalization for Natural Deduction)
If $\Gamma \vdash_\Psi \mathtt{P}$ then $\Gamma \vdash_\Psi \mathtt{P} \Uparrow$.

A special form of the above theorem is "if $[\![\mathtt{P}]\!]_\Psi$ then $\cdot \vdash_\Psi \mathtt{P} \Uparrow$". The soundness of *PropX* interpretation (Theorem 4.1) can be proved using this theorem.