Interfacing Hoare Logic and Type Systems for Foundational Proof-Carrying Code*

Nadeem Abdul Hamid and Zhong Shao

Department of Computer Science, Yale University
New Haven, CT 06520-8285, U.S.A.
{hamid,shao}@cs.yale.edu

Abstract. In this paper, we introduce a Foundational Proof-Carrying Code (FPCC) framework for constructing certified code packages from typed assembly language that will interface with a similarly certified runtime system. Our framework permits the typed assembly language to have a "foreign function" interface, in which stubs, initially provided when the program is being written, are eventually compiled and linked to code that may have been written in a language with a different type system, or even certified directly in the FPCC logic using a proof assistant. We have increased the potential scalability and flexibility of our FPCC system by providing a way to integrate programs compiled from different source type systems. In the process, we are explicitly manipulating the interface between Hoare logic and a syntactic type system.

1 Introduction

Proof-Carrying Code (PCC) [16, 17] is a framework for generating executable machine code along with a machine-checkable proof that the code satisfies a given safety policy. The initial PCC systems specified the safety policy using a logic extended with many (source) language-specific rules. While allowing implementation of a scalable system [18, 7], this approach to PCC suffers from too large of a trusted computing base (TCB). It is still difficult to trust that the components of this system – the verification-condition generator, the proof-checker, and even the logical axioms and typing rules – are free from error.

The development of another family of PCC implementations, known as Foundational Proof-Carrying Code (FPCC) [4, 3], was intended to reduce the TCB to a minimum by expressing and proving safety using only a foundational mathematical logic without additional language-specific axioms or typing rules. The trusted components in such a system are mostly reduced to a much simpler logic and the proof-checker for it.

Both these approaches to PCC have one feature in common, which is that they have focused on a single source language (*e.g.* Java or ML) and compile (type-correct) programs from that language into machine code with a safety proof. However, the runtime systems of these frameworks still include components that are not addressed in

* This research is based on work supported in part by DARPA OASIS grant F30602-99-1-0519, NSF grant CCR-9901011, NSF ITR grant CCR-0081590, and NSF grant CCR-0208618. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

the safety proof [3, 10] and that are written in a lower-level language (like C): memory management libraries, garbage collection, debuggers, marshallers, *etc.* The issue of producing a safety proof for code that is compiled and linked together from two or more different source languages was not addressed.

In this paper, we introduce an FPCC framework for constructing certified machine code packages from typed assembly language (TAL) that will interface with a similarly certified runtime system. Our framework permits the typed assembly language to have a "foreign function" interface in which stubs, initially provided when the program is being written, are eventually compiled and linked to code that may have been written in a language with a different type system, or even certified directly in the FPCC logic using a proof assistant. To our knowledge, this is the first account of combining such certification proofs from languages at different levels of abstraction. While type systems such as TAL facilitate reasoning about many programs, they are not sufficient for certifying the most low-level system libraries. Hoare logic-style reasoning, on the other hand, can handle low-level details very well but cannot account for embedded code pointers in data structures, a feature common to higher-order and object-oriented programming. We outline for the first time a way to allow both methods of verification to interact, gaining the advantages of both and circumventing their shortcomings.

Experience has shown that foundational proofs are much harder to construct than those in a logic extended with type-specific axioms. The earliest FPCC systems built proofs by constructing sophisticated semantic models of types in order to reason about safety at the machine level. That is, the final safety proof incorporated no concept of source level types – each type in the source language would be interpreted as a predicate on the machine state and the typing rules of the language would turn into lemmas which must prove properties about the interaction of these predicates. While it seems that this method of FPCC would already be amenable to achieving the goals outlined in the previous paragraph, the situation is complicated by the complexity of the semantic models [11, 5, 1] that were required to support a realistic type system. Nonetheless, the overall framework of this paper may work equally well with the semantic approach.

In this paper, we adopt the "syntactic" approach to FPCC, introduced in [13, 14] and further applied to a more realistic source type system by [9, 10]. In this framework, the machine level proofs do indeed incorporate and use the syntactic encoding of elements of the source type system to derive safety. Previous presentations of the syntactic approach involve a monolithic translation from type-correct source programs to a package of certified machine code. In this paper, we refine the approach by inserting a generic layer of reasoning above the machine code which can *(1)* be a target for the compilation of typed assembly languages, *(2)* certify low-level runtime system components using assertions as in Hoare logic, and *(3)* "glue" together these pieces by reasoning about the compatibility of the interfaces specified by the various types of source code.

A simple diagram of our framework is given in Figure 1. Source programs are written in a typed high-level language and then passed through a certifying compiler to produce machine code along with a proof of safety. The source level type system may provide a set of functionality that is accessed through a library interface. At the machine level, there is an actual library code implementation that should satisfy that interface. The non-trivial problem is how to design the framework such that not only will the two

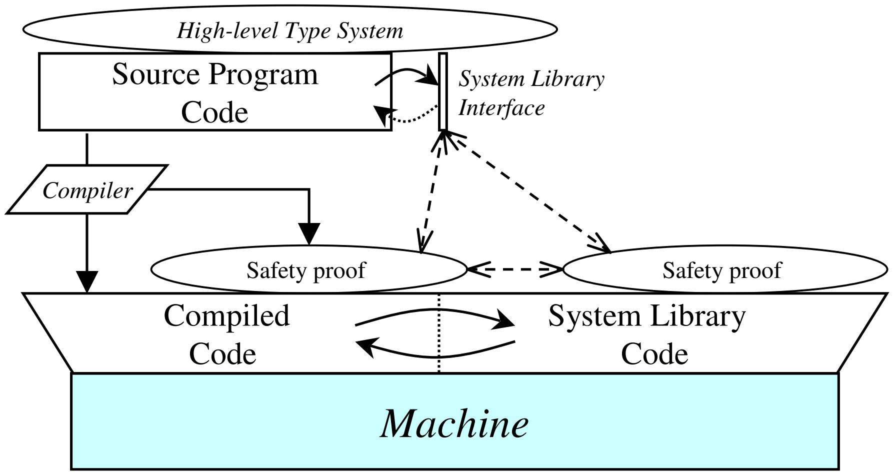

Fig. 1. FPCC certified runtime framework.

pieces of machine code link together to run, but that the safety proofs originating from two different sources are also able to "link" together, consistent with the high-level interface specification, to produce a unified safety proof for the entire set of code.

Notice that the interaction between program and library is two-way: either piece of code may make direct or indirect function calls and returns to the other. Ideally, we want to be able to certify the library code with no knowledge of the source language and type system that will be interacting with it. At the same time we would like to support first-class code pointers at all levels of the code. Methods for handling code pointers properly have been one of the main challenges of FPCC and are one of the differentiating factors between semantic and syntactic FPCC approaches. For the framework in this paper, we have factored out most of the code pointer reasoning that is needed when certifying library code so that the proofs thereof can be relatively straightforward.

In the following sections, after defining our machine and logic, we present the layer of reasoning which will serve as the common interface for code compiled from different sources. Then we present a typical typed assembly language, extended with library interfaces and external call facilities. We finally show how to compile this language to the target machine, expanding external function stubs, and linking in the runtime library, at the same time producing the proof of safety of the complete package. We conclude with a brief discussion of implementation in the Coq proof assistant and future and related work.

2 A Machine and Logic for Certified Code

In this section, we present our machine on which programs will run and the logic that we use to reason about safety of the code being run. We use an idealized machine for purposes of presentation in this paper although implementation upon the IA-32 (Intel x86 architecture) is in progress. A "real" machine introduces many engineering details (*e.g.* fixed-size integers, addressing modes, memory model, variable length instructions and relative addressing) which we would rather avoid while presenting our central contributions along the subject of this paper.

$$
\begin{array}{lll}
\mathit{Word} \ni w, i, pc & ::= & 0 \mid 1 \mid \ldots \\
\mathit{Regt} \ni r & ::= & \mathsf{r0} \mid \mathsf{r1} \mid \ldots \mid \mathsf{r15} \\
\mathit{Cmd} \ni c & ::= & \texttt{add}\ r_d, r_s, r_t \mid \texttt{addi}\ r_d, r_s, i \mid \texttt{mov}\ r_d, r_s \mid \texttt{movi}\ r_d, i \\
& & \mid \texttt{bgt}\ r_s, r_t, w \mid \texttt{bgti}\ r_s, i, w \mid \texttt{ld}\ r_d, r_s(i) \mid \texttt{st}\ r_d(i), r_s \\
& & \mid \texttt{jd}\ w \mid \texttt{jmp}\ r \mid \texttt{illegal}
\end{array}
$$

$$
\begin{array}{l}
\mathbb{M} \in \mathit{Mem} = \mathit{Word} \to \mathit{Word} \\
\mathbb{R} \in \mathit{RFile} = \mathit{Regt} \to \mathit{Word} \\
\mathbb{S} \in \mathit{State} = \mathit{Mem} \times \mathit{RFile} \times \mathit{Word}
\end{array}
$$

Fig. 2. Machine state: memory, registers, and instructions (commands).

if $\mathsf{Dc}(\mathbb{M}(pc)) =$	then $\mathsf{Step}(\mathbb{M}, \mathbb{R}, pc) =$
$\texttt{add}\ r_d, r_s, r_t$	$(\mathbb{M}, \mathbb{R}\{r_d \mapsto \mathbb{R}(r_s) + \mathbb{R}(r_t)\}, pc+1)$
$\texttt{addi}\ r_d, r_s, i$	$(\mathbb{M}, \mathbb{R}\{r_d \mapsto \mathbb{R}(r_s) + i\}, pc+1)$
$\texttt{mov}\ r_d, r_s$	$(\mathbb{M}, \mathbb{R}\{r_d \mapsto r_s\}, pc+1)$
$\texttt{movi}\ r_d, i$	$(\mathbb{M}, \mathbb{R}\{r_d \mapsto i\}, pc+1)$
$\texttt{ld}\ r_d, r_s(i)$	$(\mathbb{M}, \mathbb{R}\{r_d \mapsto \mathbb{M}(\mathbb{R}(r_s) + i)\}, pc+1)$
$\texttt{st}\ r_d(i), r_s$	$(\mathbb{M}\{\mathbb{R}(r_d) + i \mapsto \mathbb{R}(r_s)\}, \mathbb{R}, pc+1)$
$\texttt{bgt}\ r_s, r_t, w$	$(\mathbb{M}, \mathbb{R}, pc+1)$ when $\mathbb{R}(r_s) \leq \mathbb{R}(r_t)$ *and* $(\mathbb{M}, \mathbb{R}, w)$ when $\mathbb{R}(r_s) > \mathbb{R}(r_t)$
$\texttt{bgti}\ r_s, i, w$	$(\mathbb{M}, \mathbb{R}, pc+1)$ when $\mathbb{R}(r_s) \leq i$ *and* $(\mathbb{M}, \mathbb{R}, w)$ when $\mathbb{R}(r_s) > i$
$\texttt{jd}\ w$	$(\mathbb{M}, \mathbb{R}, w)$
$\texttt{jmp}\ r$	$(\mathbb{M}, \mathbb{R}, \mathbb{R}(r))$
$\texttt{illegal}$	$(\mathbb{M}, \mathbb{R}, pc)$

Fig. 3. Machine semantics.

2.1 The Machine

The hardware components of our idealized machine are a memory, register file, and a special register containing the current program counter (*pc*). These are defined to be the machine state, as shown in Figure 2. We use a 16-register word-addressed machine with an unbounded memory of unlimited-size words. We also define a decoding function Dc which decodes integer words into a structured representation of instructions ("commands"), also shown in Figure 2. The machine is thus equipped with a Step function that describes the (deterministic) transition from one machine state to the next, depending on the instruction at the current *pc*.

The operational semantics of the machine is given in Figure 3. The instructions' effects are quite intuitive. The first half involve arithmetic and data movement in registers. The `ld` and `st` load and store data from/to memory. These are followed by the conditional and unconditional branch instructions. An `illegal` (non-decodable) instruction puts the machine in an infinite loop.

2.2 The Logic

In order to produce FPCC packages, we need a logic in which we can express (encode) the operational semantics of the machine as well as define the concept and criteria of safety. A code producer must then provide a code executable (initial machine state) along with a proof that the initial state and all future transitions therefrom satisfy the safety condition.

The foundational logic we use is the calculus of inductive constructions (CiC) [24, 20]. CiC is an extension of the calculus of constructions (CC) [8], which is a higher-order typed lambda calculus. Due to limited space we forgo a discussion of CiC here and refer the reader unfamiliar with the system to the cited references.

CiC has been shown to be strongly normalizing [25], hence the corresponding logic is consistent. It is supported by the Coq proof assistant [24], which we use to implement a prototype system of the results presented in this paper.

2.3 Defining Safety and Generating Proofs

The safety condition is a predicate expressing the fact that code will not "go wrong." We say that a machine state $\mathbb{S}$ is safe if every state it can ever reach satisfies the safety policy SP:

$$\mathsf{Safe}\,(\mathbb{S}\,,\mathsf{SP}) = \Pi n\!:\!\mathsf{Nat}.\,\mathsf{SP}\,(\mathsf{Step}^n\,(\mathbb{S}\,))$$

A typical safety policy may require such things as the program counter must point to a valid instruction address in the code area and that any writes (reads) to (from) memory must be from a properly accessible area of the data space. For the purposes of presentation in this paper, we will be using a very simple safety policy, requiring only that the machine is always at a valid instruction:

$$\mathsf{BasicSP}\,(\mathbb{M},\mathbb{R},pc) = \mathsf{Dc}\,(\mathbb{M}(pc)) \neq \mathtt{illegal} \wedge \mathsf{InCodeArea}(\mathbb{M},pc)$$

We can easily define access controls on memory reads and writes by including another predicate in the safety policy, $\mathsf{SafeRdWr}(\mathbb{M},\mathbb{R},pc)$. By reasoning over the number of steps of computation more complex safety policies including temporal constraints can potentially be expressed. However, we will not be dealing with such policies here.

The FPCC code producer has to provide an encoding[1] of the initial state $\mathbb{S}_0$ along with a proof A that this state satisfies the safety condition $\mathsf{BasicSP}$, specified by the code consumer. The final FPCC package is thus a pair:

$$F = (\mathbb{S}_0 : \mathit{State},\ A : \mathsf{Safe}\,(\mathbb{S}_0, \mathsf{BasicSP})).$$

3 A Language for Certified Machine Code (CAP)

We know now what type of proof we are looking for; the hard part is to generate that proof of safety. Previous approaches for FPCC [4, 2, 5, 13] have achieved this by constructing an induction hypothesis, also known as the global invariant, which can be proven (*e.g.* by induction) to hold for all states reachable from the initial state and is strong enough to imply the safety condition. The nature of the invariant has ranged from a semantic model of types at the machine level (Appel *et al.* [4, 2, 5, 23]) to a purely syntactic well-formedness property [13, 14] based on a type-correct source program in a typed assembly language.

What we have developed in this paper refines these previous approaches. We will still be presenting a typed assembly language in Section 4, in which most source programs are written. However, we introduce another layer between the source type system

[1] We must trust that our encoding of the machine and its operational semantics, and the definition of safety, are correct. Along with the logic itself and the proof-checker implementation thereof, these make up most of our software trusted computing base (TCB).

and the “raw” encoding of the target machine in the FPCC logic. This is a “type system” or “specification system” that is defined upon the machine encoding, allowing us to reason about its state using assertions that essentially capture Hoare logic-style reasoning. Such a layer allows more generality for reasoning than a fixed type system, yet at the same time is more structured than reasoning directly in the logic about the machine encoding.

Our language is called CAP and it uses the same machine syntax as presented in Figure 2. The syntax of the additional assertion layer is given below:

$$P, Q, R \in \mathit{Pred} \quad = \mathit{State} \rightarrow \mathit{Prop}$$

$$\Phi \in \mathit{CdSpec} = \mathit{Word} \rightharpoonup (\mathit{Word} \times \mathit{Pred})$$

$$\begin{array}{l} \mathit{CmdList} \ni \mathbb{C} ::= \emptyset \mid c :: \mathbb{C} \\ \mathit{WordList} \ni \mathbb{W} ::= \emptyset \mid w :: \mathbb{W} \end{array}$$

The name CAP is derived from its being a “Certified Assembly Programming” language. An initial version was introduced in [27] and used to certify a dynamic storage allocation library. The version we have used for this paper introduces some minor improvements such as a unified data and code memory, assertions on the whole machine state, and support for user-specifiable safety policies (Section 3.3).

Assertions (P,Q,R) are predicates on the machine state and the code specification (Φ) is a partial function mapping memory addresses to a pair of an integer and a predicate. The integer gives the length of the command sequence at that address and the predicate is the precondition for the block of code. (The function of this is to allow us to specify the addresses of valid code areas of memory based on Φ.)

The operational semantics of the language has already been presented in Section 2.1. We now introduce CAP inference rules followed by some important safety theorems.

3.1 Inference Rules

CAP adds a layer of inference rules (“typing rules”) allowing us to prove specification judgments of the forms:

$$\begin{array}{rl} \Phi \vdash \{P\}\, \mathbb{C} & \textit{well-formed command sequence} \\ \vdash \mathbb{M} : \Phi & \textit{well-formed code specification} \\ \vdash (\mathbb{M}, \mathbb{R}, pc) & \textit{well-formed machine state} \end{array}$$

The inference rules for these judgments are shown in Figure 4. The rules for well-formed command sequences essentially require that if the given precondition P is satisfied in the current state, there must be some postcondition Q, which is the precondition of the remaining sequence of commands, that holds on the state after executing one step. The rules directly refer to the Step function of the machine; control flow instructions additionally use the code specification environment Φ in order to allow for the certification of mutually dependent code blocks.

We group as “pure” commands all those which do *not* involve control flow and do not change the memory (*i.e.* everything other than branches, jumps, and `st`). The `st` command requires an additional proof that the address being stored to is not in the code area (*i.e.* we do not permit self-modifying code). $\mathsf{curcmd}(\mathbb{S})$ is defined as:

$$\frac{\begin{array}{l} c \in \{\texttt{add}, \texttt{addi}, \texttt{mov}, \texttt{movi}, \texttt{ld}\} \\ \forall \mathbb{S}\,.(P(\mathbb{S}) \ \wedge\ \mathsf{curcmd}(\mathbb{S}) = c) \ \to\ Q(\mathsf{Step}(\mathbb{S}))) \qquad \Phi \vdash \{Q\}\,\mathbb{C} \end{array}}{\Phi \vdash \{P\}\ c :: \mathbb{C}} \ (\text{CAP-PURE})$$

$$\frac{\begin{array}{l} \forall \mathbb{S}\,.(P(\mathbb{S}) \wedge \mathsf{curcmd}(\mathbb{S}) = \texttt{st}\ r_d(i), r_s) \ \to\ \ Q(\mathsf{Step}(\mathbb{S})) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \neg\mathsf{InCodeArea}(\Phi, \mathbb{S}\,.\mathbb{R}(r_d)+i)) \\ \Phi \vdash \{Q\}\,\mathbb{C} \end{array}}{\Phi \vdash \{P\}\ \texttt{st}\ r_d(i), r_s :: \mathbb{C}} \ (\text{CAP-ST})$$

$$\frac{\begin{array}{l} \forall \mathbb{S}\,.(P(\mathbb{S}) \ \wedge\ \mathsf{curcmd}(\mathbb{S}) = \texttt{bgt}\ r_s, r_t, w) \\ \quad \to\ (\mathbb{S}\,.\mathbb{R}(r_s) \leq \mathbb{S}\,.\mathbb{R}(r_t) \to Q(\mathsf{Step}(\mathbb{S}))) \ \wedge\ (\mathbb{S}\,.\mathbb{R}(r_s) > \mathbb{S}\,.\mathbb{R}(r_t) \to Q'(\mathsf{Step}(\mathbb{S}))) \\ \Phi \vdash \{Q\}\,\mathbb{C} \qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{where}\ \Phi(w) = (n, Q') \end{array}}{\Phi \vdash \{P\}\ \texttt{bgt}\ r_s, r_t, w :: \mathbb{C}} \ (\text{CAP-BGT})$$

$$\frac{\forall \mathbb{S}\,.(P(\mathbb{S}) \ \wedge\ \mathsf{curcmd}(\mathbb{S}) = \texttt{jd}\,w) \to Q'(\mathsf{Step}(\mathbb{S})) \qquad \textit{where}\ \Phi(w) = (n, Q')}{\Phi \vdash \{P\}\ \texttt{jd}\ w :: \emptyset} \ (\text{CAP-JD})$$

$$\frac{\forall \mathbb{S}\,.(P(\mathbb{S}) \ \wedge\ \mathsf{curcmd}(\mathbb{S}) = \texttt{jmp}\,r) \to Q'(\mathsf{Step}(\mathbb{S}))) \quad \textit{where}\ \Phi(\mathbb{S}\,.\mathbb{R}(r)) = (n, Q')}{\Phi \vdash \{P\}\ \texttt{jmp}\ r :: \emptyset} \ (\text{CAP-JMP})$$

$$\frac{\begin{array}{l} \mathsf{Flatten}(\mathbb{W}, \mathbb{M}, f) \qquad \Phi \vdash \{P\}\,(\mathsf{Map}(\mathsf{Dc}, \mathbb{W})) \\ \quad \textit{for all}\ f\ \textit{where}\ \Phi(f) = (\mathsf{length}(\mathbb{W}), P) \end{array}}{\vdash \mathbb{M} : \Phi} \ (\text{CAP-CDSPEC})$$

$$\frac{\begin{array}{l} \vdash \mathbb{M} : \Phi \qquad \Phi \vdash \{P\}\,(\mathsf{Map}(\mathsf{Dc}, \mathbb{W})) \\ \mathsf{Flatten}(\mathbb{W}, \mathbb{M}, pc) \qquad \mathsf{InCodeArea}(\Phi, pc) \qquad P(\mathbb{M}, \mathbb{R}, pc) \end{array}}{\vdash (\mathbb{M}, \mathbb{R}, pc)} \ (\text{CAP-STATE})$$

Fig. 4. CAP inference rules.

$$\mathsf{curcmd}(\mathbb{M}, \mathbb{R}, pc) \ = \ \mathsf{Dc}(\mathbb{M}(pc))$$

The InCodeArea predicate in the rules uses the code addresses and sequence lengths in Φ to determine whether a given address lies within the code area. The (CAP-CDSPEC) rule ensures that the addresses and sequence lengths specified in Φ are consistent with the code actually in memory.

The Flatten predicate is defined as:

$$\begin{aligned} \mathsf{Flatten}(\emptyset, \mathbb{M}, f) &= \mathsf{True} \\ \mathsf{Flatten}(w :: \mathbb{W}, \mathbb{M}, f) &= \mathbb{M}(f) = w \ \wedge\ \mathsf{Flatten}(\mathbb{W}, \mathbb{M}, f+1) \end{aligned}$$

3.2 Safety Properties

The machine will execute continuously, even if an `illegal` instruction is encountered. Given a well-formed CAP state, however, we can prove that it satisfies our basic safety policy, and that executing the machine one step will result again in a good CAP state.

Theorem 1 (Safety Policy and Preservation).
For some state $\mathbb{S}$*, if* $\vdash \mathbb{S}$ *then (1)* $\mathsf{BasicSP}(\mathbb{S})$ *and (2)* $\vdash \mathsf{Step}^n(\mathbb{S})$ *for all* n*.*

For the purposes of FPCC, we are interested in obtaining safety proofs in the context of our policy as described in Section 2.3. From Theorem 1 we can easily derive:

Theorem 2 (CAP Safety). *For any* $\mathbb{S}$*, if* $\vdash \mathbb{S}$ *then* $\mathsf{Safe}(\mathbb{S}, \mathsf{BasicSP})$.

Thus, to produce an FPCC package we just need to prove that the initial machine state is well-formed with respect to the CAP inference rules. This provides a structured method for constructing FPCC packages in our logic. However, programming and reasoning in CAP is still much too low-level for the practical programmer. We thus need to provide a method for compiling programs from a higher-level language and type system to CAP. The main purpose of programming directly in CAP will then be to "glue" code together from different source languages and to certify particularly low-level libraries such as memory management. In the next few sections, we present a "conventional" typed assembly language and show how to compile it to CAP.

3.3 Advanced safety policies

In the theorems above, and for the rest of this paper, we are only interested in proving safety according to our basic safety policy. For handling more general safety policies using CAP, we can extend our CAP inference rules by parameterizing them with a "global safety predicate" SP: $\Phi \vdash_{\mathsf{SP}} \{P\}\,\mathbb{C}$, $\vdash_{\mathsf{SP}} \mathbb{M} : \Phi$, and $\vdash_{\mathsf{SP}} (\mathbb{M}, \mathbb{R}, pc)$.

The inference rule for each command in this extended system requires an additional premise that the precondition for the command implies the global safety predicate. Then, using a generalized version of Theorem 1, we can establish that:

Theorem 3. *For any* $\mathbb{S}$ *and* SP, *if* $\vdash_{\mathsf{SP}} \mathbb{S}$ *then* $\mathsf{Safe}(\mathbb{S}, \lambda\mathbb{S}'{:}\mathit{State}.\ \mathsf{SP}(\mathbb{S}') \wedge \mathsf{BasicSP}(\mathbb{S}'))$.

Threading an arbitrary SP through the typing rules is a novel feature not found in the initial version of CAP [27]. In that case, there was no way to specify that an arbitrary safety policy beyond BasicSP (which essentially provides type safety) must hold at every step of execution.

4 Extensible Typed Assembly Language with Runtime System

In this section, we introduce an extensible typed assembly language (XTAL) based on that of Morrisett *et al.* [15]. After presenting the full syntax of XTAL, we give here only a brief overview of its static and dynamic semantics, due to space constraints of this paper. A more complete definition of the language can be found in the Coq implementation itself or the technical report [12].

4.1 Syntax

To simplify the presentation, we will use a much scaled down version of typed assembly language (see Figure 5)–its types involve only integers, pairs, and integer arrays. (We have extended our prototype implementation to include existential, recursive, and polymorphic code types.) The code type $\forall[\Gamma]$ describes a code pointer that expects a register file satisfying Γ. The register file type assigns types to the word values in each register and the heap type keeps track of the heap values in the data heap. We have separated the code and data heaps at this level of abstraction because the code heap will remain the same throughout the execution of a program.

Unlike many conventional TALs, our language supports "stub values" in its code heap. These are placeholders for code that will be linked in later from another source (outside the XTAL system). Primitive "macro" instructions that might be built into other TALs, such as array creation and access operations, can be provided as an external

(type)	τ	$::=$ int | array | $\tau_0 \times \tau_1$ | $\forall[\Gamma]$
(reg file type)	Γ	$::= \{r_0 : \tau_0, \ldots, r_n : \tau_n\}$
(heap type)	Ψ	$::= \{l_0 : \tau_0, \ldots, l_n : \tau_n\}$
(label)	l	$::= 0 \mid 1 \mid \ldots$
(register)	r	$::=$ r0 | r1 | ... | r7
(word val)	v	$::= l \mid i$
(code heap val)	$\overline{h}$	$::=$ code $[\Gamma].I$ | stub $[\Gamma].\emptyset$
(heap val)	h	$::= [i_0, \ldots, i_n] \mid \langle v_0, v_1 \rangle$
(instr)	ι	$::=$ add r_d, r_s, r_t | movi r_d, i | movl r_d, l | ld $r_d, r_s(i)$ | st $r_d(i), r_s$ | bgt r_s, r_t, l | bgti r_s, i, l | newpair $r_d[\tau_0, \tau_1]$
(instr seq)	I	$::= \iota; I$ | jd l | jmp r
(code heap)	$\mathcal{C}$	$::= \{l_0 \mapsto \overline{h}_0, \ldots, l_n \mapsto \overline{h}_n\}$
(data heap)	H	$::= \{l_0 \mapsto h_0, \ldots, l_n \mapsto h_n\}$
(reg file)	R	$::= \{r_0 \mapsto v_0, \ldots, r_n \mapsto v_n\}$
(program)	$\mathcal{P}$	$::= (\mathcal{C}, H, R, I)$

Fig. 5. XTAL syntax.

library with interface specified as XTAL types. We have also included a typical macro instruction for allocating pairs (newpair) in the language. When polymorphic types are added to the language, this macro instruction could potentially be provided through the external code interface; however, in general, providing built-in primitives can allow for a richer specification of the interface (see the typing rule for newpair below).

The abstract state of an XTAL program is composed of code and data heaps, a register file, and current instruction sequence. Labels are simply integers and the domains of the code and data heaps are to be disjoint. Besides the newpair operation, the arithmetic, memory access, and control flow instructions of XTAL correspond directly to those of the machine defined in 2.1. The movl instruction is constrained to refer only to code heap labels. Note that programs are written in continuation passing style; thus every code block ends with some form of jump to another location in the code heap.

4.2 Static and Dynamic Semantics

The dynamic (operational) semantics of the XTAL abstract machine is defined by a set of rules of the form $\mathcal{P} \mapsto \mathcal{P}'$. This evaluation relation is entirely standard (see [15, 14]) except that the case when jumping to a stub value in the code heap is not handled. The complete rules are omitted here.

For the static semantics, we define a set of judgments as illustrated in Figure 6. Only a few of the critical XTAL typing rules are presented here. The top-level typing rule for XTAL programs requires well-formedness of the code and data heaps, register file, and current instruction sequence, and that I is somewhere in the code heap:

$$\frac{\begin{array}{c}\vdash \mathcal{C} \qquad \vdash H : \Psi \qquad \mathcal{C}; \Psi \vdash R : \Gamma \qquad \mathcal{C}; \Gamma \vdash I \\ \exists l \in Dom(\mathcal{C}).\ \mathcal{C}(l) = \text{code}\,[\Gamma'].I' \ \ and\ \ I \subseteq_{tail} I'\end{array}}{\vdash (\mathcal{C}, H, R, I)} \ (\text{PROG})$$

Judgment	Meaning
$\vdash \Gamma_0 \subseteq \Gamma_1$	Γ_0 is a register file subtype of Γ_1
$\vdash (\mathcal{C}, H, R, I)$	$(\mathcal{C}, H, R, I)$ is a well-formed program
$\vdash \mathcal{C}$	$\mathcal{C}$ is a well-formed code heap
$\vdash H : \Psi$	H is a well-formed data heap of type Ψ
$\mathcal{C}; \Psi \vdash R : \Gamma$	R is a well-formed reg. file of type Γ
$\mathcal{C} \vdash \overline{h}$ cdval	$\overline{h}$ is a well-formed code heap value
$\Psi \vdash h : \tau$ hval	h is a well-formed data heap value of type τ
$\Psi; \vdash v : \tau$	v is a well-formed word value of type τ
$\mathcal{C}; \Gamma \vdash I$	I is a well-formed instruction sequence

Fig. 6. Static judgments.

Heap and register file typing depends on the well-formedness of the elements in each. Stub values are simply assumed to have the specified code type. From the instruction typing rules, we show below the rules for newpair, jd, and jmp. The newpair instruction expects initialization values for the newly allocated space in registers r0 and r1 and a pointer to the new pair is put in r_d.

$$\frac{\mathcal{C}; \Gamma \vdash I}{\mathcal{C} \vdash \mathsf{code}\ [\Gamma].I\ \mathsf{cdval}}\ (\text{CODE}) \qquad \frac{}{\mathcal{C} \vdash \mathsf{stub}\ [\Gamma].\emptyset\ \mathsf{cdval}}\ (\text{STUB})$$

$$\frac{\Gamma(\mathsf{r0}) = \tau_0 \qquad \Gamma(\mathsf{r1}) = \tau_1 \qquad \mathcal{C}; \Gamma\{r_d : \tau_0 \times \tau_1\} \vdash I}{\mathcal{C}; \Gamma \vdash \mathsf{newpair}\ r_d[\tau_0, \tau_1]; I}\ (\text{IS-NEWPAIR})$$

$$\frac{\mathsf{typeof}(\mathcal{C}(l)) = \forall[\Gamma'] \quad \vdash \Gamma \subseteq \Gamma'}{\mathcal{C}; \Gamma \vdash \mathsf{jd}\ l}\ (\text{IS-JD}) \qquad \frac{\Gamma(r) = \forall[\Gamma'] \quad \vdash \Gamma \subseteq \Gamma'}{\mathcal{C}; \Gamma \vdash \mathsf{jmp}\ r}\ (\text{IS-JMP})$$

Although the details of the type system are certainly important, the key thing to be understood here is just that we are able to encode the syntactic judgment forms of XTAL in our logic and prove soundness in Wright-Felleisen style [26]. We will then refer to these judgments in CAP assertions during the process of proving machine code safety.

4.3 External Code Stub Interfaces

XTAL can pass around pointers to arrays in its data heap but has no built-in operations for allocating, accessing, or modifying arrays. We provide these through code stubs:

$$\begin{aligned}
\mathsf{newarray} &\mapsto \mathsf{stub}\ [\{\ \mathsf{r0}:\mathsf{int},\ \mathsf{r1}:\mathsf{int},\ \mathsf{r7}:(\forall[\{\mathsf{r0}:\mathsf{array}\}])\ \}].\emptyset \\
\mathsf{arrayget} &\mapsto \mathsf{stub}\ [\{\ \mathsf{r0}:\mathsf{array},\ \mathsf{r1}:\mathsf{int},\ \mathsf{r7}:(\forall[\{\mathsf{r0}:\mathsf{int}\}])\ \}].\emptyset \\
\mathsf{arrayset} &\mapsto \mathsf{stub}\ [\{\ \mathsf{r0}:\mathsf{array},\ \mathsf{r1}:\mathsf{int},\ \mathsf{r2}:\mathsf{int},\ \mathsf{r7}:(\forall[\{\mathsf{r0}:\mathsf{array}\}])\ \}].\emptyset
\end{aligned}$$

newarray expects a length and initial value as arguments, allocates and initializes a new array accordingly, and then jumps to the code pointer in r7. The accessor operations similarly expect an array and index arguments and will return to the continuation pointer in r7 when they have performed the operation. As is usually the case when dealing with external libraries, the interfaces (code types) defined above do not provide a complete specification of the operations (such as bounds-checking issues). Section 5.3 discusses how we deal with this in the context of the safety of XTAL programs and the final executable machine code.

4.4 Soundness

As usual, we need to show that our XTAL type system is sound with respect to the operational semantics of the abstract machine. This can be done using the standard progress and preservation lemmas. However, in the presence of code stubs, the complete semantics of a program is undefined, so at this level of abstraction we can only assume that those typing rules are sound. In the next section, when compiling XTAL programs to the real machine and linking in code for these libraries and stubs, we will need to prove at that point that the linked code is sound with respect to the XTAL typing rules. Let us define the state when the current XTAL program is jumping to external code:

Definition 1 (External call state). *We define the current instruction of a program,* $(\mathcal{C}, H, R, I)$*, to be an* external call *if* $I \in \{\mathsf{jd}\ l,\ \mathsf{jmp}\ r,\ \mathsf{bgt}\ldots,\ \mathsf{bgti}\ldots\}$ *and* $\mathcal{C}(l) = \mathsf{stub}\ [\Gamma].\emptyset$ *or* $\mathcal{C}(R(r)) = \mathsf{stub}\ [\Gamma].\emptyset$*, as appropriate.*

Theorem 4 (XTAL Progress). *If* $\vdash \mathcal{P}$ *and the current instruction of* $\mathcal{P}$ *is not an external call then there exists* $\mathcal{P}'$ *such that* $\mathcal{P} \mapsto \mathcal{P}'$*.*

Theorem 5 (XTAL Preservation). *If* $\vdash \mathcal{P}$ *and* $\mathcal{P} \mapsto \mathcal{P}'$ *then* $\vdash \mathcal{P}'$*.*

These theorems are proven by induction on the well-formed instruction premise $(\mathcal{C}; \Gamma \vdash I)$ of the top level typing rule $(\vdash \mathcal{P})$. Of course the proof of these must be done entirely in the FPCC logic in which the XTAL language is encoded.

In our previous work [13, 14], we demonstrated how to get from these proofs of soundness directly to the FPCC safety proof. However, now we have an extra level to go through (the CAP system) in which we will also be linking external code to XTAL programs, and we must ensure safety of the complete package at the end.

5 Compilation and Linking

In this section we first define how abstract XTAL programs will be translated to, and laid out in, the real machine state (the runtime memory layout). We also define the necessary library routines as CAP code (the runtime system). Then, after compiling and linking an XTAL program to CAP, we must show how to maintain the well-formedness of that CAP state so that we can apply Theorem 2 to obtain the final FPCC proof of safety.

5.1 The Runtime System

In our simple runtime system, memory is divided into three sections–a static data area (used for global constants and library data structures), a read-only code area (which might be further divided into subareas for external $(\mathcal{E})$ and program code), and the dynamic heap area, which can grow indefinitely in our idealized machine. We use a data allocation framework where a heap limit, stored in a fixed allocation pointer register,[2] designates a finite portion of the dynamic heap area as having been allocated for use. (Our safety policy could use this to specify "readable" and "writeable" memory.)

[2] XTAL source programs use fewer registers than the actual machine provides.

5.2 Translating XTAL Programs to CAP

We now outline how to construct (compile) an initial CAP machine state from an XTAL program. Given an initial XTAL program, we need the following (partial) functions or mappings to produce the CAP state:

- $\mathcal{A}_C$: *label* $\rightharpoonup$ *Word* – a layout mapping from XTAL code heap labels to CAP machine addresses.
- $\mathcal{A}_D$: *label* $\rightharpoonup$ *Word* – a layout mapping from XTAL data heap labels to CAP machine addresses. Both the domain and range of the two layout functions should be disjoint. We use $\mathcal{A}$ without any subscript to indicate the union of the two: $\mathcal{A} = \mathcal{A}_C \cup \mathcal{A}_D$.
- $\mathcal{E}$: *Word* $\rightharpoonup$ *CmdList* $\times$ *Pred* – the external (from XTAL's point of view) code blocks and their CAP preconditions for well-formedness. Proving that these blocks are well-formed according to the preconditions will be a proof obligation when verifying the safety of the complete CAP state. The range of $\mathcal{A}_C$ may overlap with the domain of $\mathcal{E}$ – these addresses are the implementation of XTAL code stubs.

With these elements, the translation from XTAL programs to CAP is quite straightforward. As in [13], we can describe the translation by a set of relations and associated inference rules. Because of limited space, we only show here the top-level rule:

$$\frac{\begin{array}{l}\mathcal{A} \vdash (\mathcal{C}, H) \Rightarrow \mathbb{M} \qquad \mathcal{A} \vdash R \Rightarrow \mathbb{R} \qquad \mathcal{A}_C \vdash I \Rightarrow \mathbb{C} \qquad \mathsf{Flatten}(\mathbb{C}, \mathbb{M}, pc) \\ \exists l.\, \mathcal{C}(l) = \mathsf{code}\,[\Gamma].I' \;\wedge\; I \subseteq_{tail} I' \;\wedge\; pc = \mathcal{A}_C(l) + |I'| - |I| \\ \forall w \in Dom(\mathcal{E}).\, \mathsf{Flatten}(\mathsf{Fst}(\mathcal{E}(w)), \mathbb{M}, w)\end{array}}{\mathcal{E}; \mathcal{A} \vdash (\mathcal{C}, H, R, I) \Rightarrow (\mathbb{M}, \mathbb{R}, pc)} \;(\text{TR-PROG})$$

Register files and word values translate fairly directly between XTAL and the machine. XTAL labels are translated to machine addresses using the $\mathcal{A}$ functions. Every heap value in the code and data heaps must correspond to an appropriately translated sequence of words in memory. All XTAL instructions translate directly to a single machine command except newpair which translates to a series of commands that adjust the allocation pointer to make space for a new pair and then copy the initial values from r0 and r1 into the new space. We ignore the stubs in the XTAL code heap translation because they are handled in the top-level translation rule shown above (when $\mathcal{E}$ is Flatten'ed).

5.3 Generating the CAP Proofs

In this section we proceed in a top-down manner by first stating the main theorem we wish to establish. The theorem says that for a given runtime system, any well-typed XTAL program that compiles and links to the runtime will result in an initial machine state that is well-formed according to the CAP typing rules. Applying Theorem 2, we would then be able to produce an FPCC package certifying the safety of the initial machine state.

Theorem 6 (XTAL-CAP Safety Theorem). *For some specified external code environment $\mathcal{E}$, and for all $\mathcal{P}$ and $\mathcal{A}$, if $\vdash \mathcal{P}$ (in XTAL) and $\mathcal{E}; \mathcal{A} \vdash \mathcal{P} \Rightarrow \mathbb{S}$, then $\vdash \mathbb{S}$ (in CAP).*

To prove that the CAP state is well-formed (using the (CAP-STATE) rule, Figure 4), we need a code heap specification, Φ, and a top-level precondition, P, for the current program counter. The code specification is generated as follows: $\Phi = \mathsf{CpGen}(\mathcal{E}, \mathcal{A}_C, \mathcal{C})$, where

$$\mathsf{CpGen}(\mathcal{E}, \mathcal{A}_C, \mathcal{C})(w) = \begin{cases} \mathsf{CpInv}(\mathcal{A}_C, \mathcal{C}, \Gamma) & \textit{if } w \notin Dom(\mathcal{E}) \textit{ and } \exists l. \mathcal{A}_C(l) = w \ \wedge \ \mathcal{C}(l) = (\mathsf{code}\ [\Gamma].I) \\ \mathsf{Snd}(\mathcal{E}(w)) & \textit{if } w \in Dom(\mathcal{E}) \end{cases}$$

That is, for external code blocks, the precondition comes directly from $\mathcal{E}$, while for code blocks that have been compiled from XTAL, the CAP preconditions are constructed by the following definition:

$$\mathsf{CpInv}(\mathcal{A}_C, \mathcal{C}, \Gamma) = \lambda \mathbb{S}.\, \exists \mathcal{A}_D, \Psi, H, R. (\vdash \mathcal{C}) \ \wedge \ (\vdash H : \Psi) \ \wedge \ (\mathcal{C}; \Psi \vdash R : \Gamma) \\ \wedge (\mathcal{A} \vdash (\mathcal{C}, H) \Rightarrow \mathbb{S}.\mathbb{M}) \ \wedge \ (\mathcal{A} \vdash R \Rightarrow \mathbb{S}.\mathbb{R})$$

For any given program, the code heap and layout ($\mathcal{C}$ and $\mathcal{A}_C$) must be unchanged, therefore they are global parameters of these predicate generators. CpInv captures the fact that at a particular machine state there is a well-typed XTAL memory and register file that syntactically corresponds to it. We only need to specify the register file type as an argument to CpInv because the typing rules for the well-formed register file and heap will imply all the necessary restrictions on the data heap structure. One of the main insights of this work is the definition of CpInv, which allows us to both establish a syntactic invariant on CAP machine states as well as define the interface between XTAL and library code at the CAP level. CpInv is based on a similar idea as the global invariant defined in [13] but instead of a generic, monolithic safety proof using the syntactic encoding of the type system, CpInv makes clear what the program-specific preconditions are for each command (instruction) and allows for easy manipulation and reasoning thereupon, as well as interaction with other type system-based invariants.

Returning to the proof of Theorem 6, if we define the top-level precondition of the (CAP-STATE) rule to be $\mathsf{CpInv}(\mathcal{A}_C, \mathcal{C}, \Gamma)$, then it is trivially satisfied on the initial state $\mathbb{S}$ by the premises of the theorem. We now have to show well-formedness of the code at the current program counter, $\Phi \vdash \{P\}\, \mathbb{C}$, and, in fact, proofs of the same judgment form must be provided for each of the code blocks in the heap, according to the (CAP-CDSPEC) rule. The correctness of the CAP code memory is shown by the theorem:

Theorem 7 (XTAL-CAP Code Heap Safety). *For a specified $\mathcal{E}$, and for any XTAL program state $(\mathcal{C}, H, R, I)$, register file type Γ, layout functions $\mathcal{A}$, and machine state $(\mathbb{M}, \mathbb{R}, pc)$, such that $\vdash (\mathcal{C}, H, R, I)$ and $\mathcal{E}; \mathcal{A} \vdash (\mathcal{C}, H, R, I) \Rightarrow (\mathbb{M}, \mathbb{R}, pc)$, if $\Phi = \mathsf{CpGen}(\mathcal{E}, \mathcal{A}_C, \mathcal{C})$, then $\vdash \mathbb{M} : \Phi$.*

This depends in turn on the proof that each well-typed XTAL instruction sequence translated to machine commands will be well-formed in CAP under CpInv:

Theorem 8 (XTAL-CAP Instruction Safety). *For a specified $\mathcal{E}$, and for all $\mathcal{A}_C$, $\mathcal{C}$, I, Γ, and $\mathbb{C}$ (where $\Phi = \mathsf{CpGen}(\mathcal{E}, \mathcal{A}_C, \mathcal{C})$), if $\mathcal{C}; \Gamma \vdash I$ and $\mathcal{A}_C \vdash I \Rightarrow \mathbb{C}$, then $\Phi \vdash \{\mathsf{CpInv}(\mathcal{A}_C, \mathcal{C}, \Gamma)\}\, \mathbb{C}$.*

Due to space constraints, we omit details of the proof of this theorem except to mention that it is proved by induction on I. In cases where the current instruction directly maps to a machine command (*i.e.*, other than newpair), the postcondition (Q in the CAP rules) is generated by applying CpInv to the updated XTAL register file type. We use the XTAL safety theorems (4 and 5) here to show that Q holds after one step of execution. In the case of the expanded commands of newpair, we must construct the intermediate postconditions by hand and then show that CpInv is re-established on the state after the sequence of expanded commands has been completed. In the case when jumping to external code, we use the result of Proof Obligation 10 below.

Finally, establishing the theorems above depends on satisfying some proof obligations with respect to the external library code and its interfaces as specified at the XTAL level. First, we must show that the external library code is well-formed according to its supplied preconditions:

Proof Obligation 9 (External Code Safety) *For a given* $\mathcal{E}$*, if* $\Phi = \mathsf{CpGen}(\mathcal{E}, \mathcal{A}_C, \mathcal{C})$ *for any* $\mathcal{A}_C$ *and* $\mathcal{C}$*, then* $\Phi \vdash \{\mathsf{Snd}(\mathcal{E}(w))\}\,\mathsf{Fst}(\mathcal{E}(w))$*, for all* $w \in Dom(\mathcal{E})$*.*

For now, we assume that the proofs of this lemma are constructed "by hand" using the rules for well-formedness of CAP commands.

Secondly, when linking the external code with a particular XTAL program, where certain labels of the XTAL code heap are mapped to external code addresses, we have to show that the typing environment that would hold at any XTAL program that is jumping to that label implies the actual precondition of that external code:

Proof Obligation 10 (Interface Correctness) *For a given* $\mathcal{E}$*,* $\mathcal{A}_C$*, and* $\mathcal{C}$*, and for all* l *such that* $\mathcal{C}(l) = \mathsf{stub}\,[\Gamma].\emptyset$ *and* $\mathcal{A}_C(l) = w$*, if* $\mathsf{CpInv}(\mathcal{A}_C, \mathcal{C}, \Gamma)(\mathbb{S})$ *then* $\mathsf{Snd}(\mathcal{E}(w))(\mathbb{S})$*.*

These properties must be proved for each instantiation of the runtime system $\mathcal{E}$. With them, the proofs of Theorems 8, 7, and, finally, 6 can be completed.

5.4 arrayget Example

As a concrete example of the process discussed in the foregoing subsection, let us consider arrayget. The XTAL type interface is defined in Section 4.3. An implementation of this function could be:

$$\mathbb{C}_{aget} = [\texttt{ld}\ \mathsf{r8}, \mathsf{r0}(0);\ \texttt{addi}\ \mathsf{r1}, \mathsf{r1}, 1;\ \texttt{bgt}\ \mathsf{r1}, \mathsf{r8}, \mathsf{bnderr};\ \texttt{add}\ \mathsf{r0}, \mathsf{r0}, \mathsf{r1};\ \texttt{ld}\ \mathsf{r0}, \mathsf{r0}(0);\ \texttt{jmp}\ \mathsf{r7}]$$

The runtime representation of an array in memory is a length field followed by the actual array of data. We assume that there is some exception handling routine for out-of-bounds accesses with a trivial precondition defined by $\mathcal{E}(\mathsf{bnderr}) = (\mathbb{C}_{bnderr}, Q_{bnderr})$.

Before describing the CAP assertions for the safety of $\mathbb{C}_{aget}$, notice that the code returns indirectly to an XTAL function pointer. Similarly, the arrayget address can be passed around in XTAL programs as a first-class code pointer. While the syntactic type system handles these code pointers quite easily using the relevant XTAL types, dealing with code pointers in a Hoare logic-based setup like CAP requires a little bit of machinery.

We can thus proceed to directly define the precondition of $\mathbb{C}_{aget}$ as,

$$Q_{aget} = \mathsf{CpInv}(\mathcal{A}_C, \mathcal{C}, \{\ \mathsf{r0}:\mathsf{array},\ \mathsf{r1}:\mathsf{int},\ \mathsf{r7}:(\forall[\{\mathsf{r0}:\mathsf{int}\}])\ \})$$

for some $\mathcal{A}_C$ and $\mathcal{C}$. Then we certify the library code in CAP by providing a derivation of $(\Phi \vdash \{Q_{aget}\}\,\mathbb{C}_{aget})$. We do this by applying the appropriate rules from Figure 4 to track the changes that are made to the state with each command. When we reach the final jump to r7, we can then show that $\mathsf{CpInv}(\mathcal{A}_C, \mathcal{C}, \{\mathsf{r0}:\mathsf{int}\})$ holds, which must be the precondition specified for the return code pointer by $\Phi(\mathbb{S}.\mathbb{R}(\mathsf{r7}))$ (see the definition of Φ in the beginning of Section 5.3). The problem with this method of certifying arrayget, however, is that we have explicitly included details about the source language type system in its preconditions. In order to make the proof more generic, while at the same time be able to leverage the syntactic type system for certifying code pointers, we follow a similar approach as in [27]: First, we define generic predicates for the pre- and postconditions, abstracting over an arbitrary external predicate, P_{aget}. The actual requirements of the arrayget code are minimal (for example, that the memory area of the array is readable according to the safety policy). The post-condition predicate relates the state of the machine upon exiting the code block to the initial entry state:

$$\begin{aligned}
\mathsf{Pre} &= \lambda P_{aget}.\lambda\mathbb{S}.\ P_{aget}(\mathbb{S}) \wedge \mathsf{SafeToRead}(\mathbb{S}.\mathbb{M},\ \mathbb{S}.\mathbb{R}(\mathsf{r0}),\ \mathbb{S}.\mathbb{R}(\mathsf{r1})+1)\\
\mathsf{Post} &= \lambda(\mathbb{M},\mathbb{R},pc).\ \lambda(\mathbb{M}',\mathbb{R}',pc').\ \mathbb{M}' = \mathbb{M} \wedge pc' = \mathbb{S}.\mathbb{R}(\mathsf{r7})\\
&\qquad\qquad \wedge \mathbb{R}'(\mathsf{r0}) = \mathbb{M}(\mathbb{R}(\mathsf{r0})+\mathbb{R}(\mathsf{r1})+1) \wedge \ldots
\end{aligned}$$

Now we certify the arrayget code block, quantifying over all P_{aget} and complete code specifications Φ, but imposing some appropriate restrictions on them:

$$\begin{aligned}
\forall\Phi, P_{aget}.\ & \Phi(\mathsf{bnderr}) = Q_{bnderr} \wedge (\forall\mathbb{S},\mathbb{S}'.\mathsf{Pre}(P_{aget})(\mathbb{S}) \wedge \mathsf{Post}(\mathbb{S})(\mathbb{S}') \to \Phi(\mathbb{S}.\mathbb{R}(\mathsf{r7}))(\mathbb{S}'))\\
& \to \Phi \vdash \{\mathsf{Pre}(P_{aget})\}\,\mathbb{C}_{aget}
\end{aligned}$$

Thus, under the assumption that the Pre predicate holds, we can again apply the inference rules for CAP commands to show the well-formedness of the $\mathbb{C}_{aget}$ code. When we reach the final jump, we show that the Post predicate holds and then use that fact with the premise of the formula above to show that it is safe to jump to the return code pointer.

The arrayget code can thus be certified independent of any type system, by introducing the quantified P_{aget} predicate. Now, when we want to use this as an external function for XTAL programs, we instantiate P_{aget} with Q_{aget} above. We have to prove the premise of the formula above, $(\forall\mathbb{S},\mathbb{S}'.\mathsf{Pre}(Q_{aget})(\mathbb{S}) \wedge \mathsf{Post}(\mathbb{S})(\mathbb{S}') \to \Phi(\mathbb{S}.\mathbb{R}(\mathsf{r7}))(\mathbb{S}'))$. Proving this is not difficult, because we use properties of the XTAL type system to show that from a state satisfying the precondition–*i.e.* there is a well-formed XTAL program whose register file satisfies the arrayget type interface– the changes described by the Post predicate will result in a state to which there does correspond another well-formed XTAL program, one where the register r0 is updated with the appropriate element of the array. Then we can let $\mathcal{E}(\mathsf{arrayget}) = (\mathbb{C}_{aget}, \mathsf{Pre}(Q_{aget}))$ and we have satisfied Proof Obligation 9. Proof Obligation 10 follows almost directly given our definition of Q_{aget}.

In summary, we have shown how to certify runtime library code independent of a source language. In order to handle code pointers, we simply assume their safety as a premise; then, when using the library with a particular source language type system, we instantiate with a syntactic well-formedness predicate in the form of CpInv and use the facilities of the type system for checking code pointers to prove the safety of indirect jumps.

6 Implementation and Future Work

We have a prototype implementation of the system presented in this paper, developed using the Coq proof assistant. Due to space constraints, we have left out its details here. As mentioned earlier in the paper, our eventual goal is to build an FPCC system for real IA-32 (Intel x86) machines. We have already applied the CAP type system to that architecture and will now need to develop a more realistic version of XTAL. Additionally, our experience with the Coq proof assistant leads us to believe that there should be more development on enhancing the automation of the proof tactics, because many parts of the proofs needed for this paper are not hard or complex, but tedious to do given the rather simplistic tactics supplied with the base Coq system.

In this paper, we have implicitly assumed that the CAP machine code is generated from one of two sources: *(a)* XTAL source code, or *(b)* code written directly in CAP. However, more generally, our intention is to support code from multiple source type systems. In this case, the definition of CpGen (Section 5.3) would utilitize code precondition invariant generators (CpInv) from the multiple type systems. The general form of each CpInv would be the same, although, of course, the particular typing environments and judgments would be different for each system. Then we would have a series of theorems like those in Section 5.3, specialized for each CpInv. Proof Obligation 10 would also be generalized as necessary, requiring proofs that the interfaces between the various type systems are compatible. Of course there will be some amount of engineering required to get such a system up and running, but we believe that there is true potential for building a realistic, scalable FPCC framework along these lines.

7 Related Work and Conclusion

In the context of the original PCC systems cited in the Introduction, there has been recent work to improve their flexibility and reliability by removing type-system specific components from the framework [19]. These systems have the advantage of working, production-quality implementations but it is still unclear whether they can approach the trustworthiness goals of FPCC.

We also mentioned the first approaches to generating FPCC, which utilized semantic models of the source type system, and their resulting complexities. Attempting to address and hide the complexity of the semantic soundness proofs, Juan Chen *et al.* [6] have developed LTAL, a low-level typed assembly language which is used to compile core ML to FPCC. LTAL is based in turn upon an abstraction layer, TML (typed machine language) [22], which is an even lower-level intermediate language. Complex parts of the semantic proofs, such as the indexed model of recursive types and stratified model of mutable fields, are hidden in the soundness proof of TML and as long as a typed assembly language can be compiled to TML, one need not worry about the semantic models. All the same, LTAL and TML are only assembly language type systems, albeit at a much lower level that XTAL. They do not provide CAP's generality of reasoning nor can their type systems be used to certify their own runtime system components. It should be clearly noted that the ideas presented in this paper are not restricted to use with a syntactic FPCC approach, as we have pursued. Integrating LTAL or TML with the CAP framework of this paper to certify their runtime system components seems feasible as well.

Along the syntactic approach to FPCC, Crary [9, 10] applied our methods [13, 14] to a realistic typed assembly language initially targeted to the Intel x86. He even went on to specify invariants about the garbage collector interface, but beyond the interface the implementation is still uncertified. In his work he uses the metalogical framework of Twelf [21] instead of the CiC-based Coq that we have been using.

In conclusion, there is much ongoing development of PCC technology for producing certified machine code from high-level source languages. Concurrently, there is exciting work on certifying garbage collectors and other low-level system libraries. However, integrating the high and low-level proofs of safety has not yet received much attention. The ideas presented in this paper represent a viable approach to dealing with the issue of interfacing and integrating safety proofs of machine code from multiple sources in a fully certified framework.

Acknowledgments

We would like to thank the anonymous referees for their comments on an earlier version of this paper.

References

1. A. Ahmed, A. W. Appel, , and R. Virga. A stratified semantics of general references embeddable in higher-order logic. In *Proc. 17th Annual IEEE Symposium on Logic in Computer Science*, pages 75–86, June 2002.
2. A. J. Ahmed. Mutable fields in a semantic model of types. Talk presented at 2000 PCC Workshop, June 2000.
3. A. W. Appel. Foundational proof-carrying code. In *Proc. 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–258, June 2001.
4. A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proc. 27th ACM Symp. on Principles of Prog. Lang.*, pages 243–253. ACM Press, 2000.
5. A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. on Programming Languages and Systems*, 23(5):657–683, Sept. 2001.
6. J. Chen, D. Wu, A. W. Appel, and H. Fang. A provably sound tal for back-end optimization. In *Proc. 2003 ACM Conf. on Prog. Lang. Design and Impl.*, pages 208–219, New York, 2003. ACM Press.
7. C. Colby, P. Lee, G. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *Proc. 2000 ACM Conf. on Prog. Lang. Design and Impl.*, pages 95–107, New York, 2000. ACM Press.
8. T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.
9. K. Crary. Towards a foundational typed assembly language. In *Proc. 30th ACM Symp. on Principles of Prog. Lang.*, pages 198 – 212. ACM Press, Jan. 2003.

10. K. Crary and S. Sarkar. A metalogical approach to foundational certified code. Technical Report CMU-CS-03-108, Carnegie Mellon University, Jan. 2003.
11. A. Felty. Semantic models of types and machine instructions for proof-carrying code. Talk presented at 2000 PCC Workshop, June 2000.
12. N. A. Hamid and Z. Shao. Coq code for interfacing hoare logic and type systems for fpcc. Available at `flint.cs.yale.edu/flint/publications`, May 2004.
13. 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, June 2002.
14. N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof carrying-code. *Journal of Automated Reasoning (Special issue on Proof-Carrying Code)*, 31(3-4):191–229, Dec. 2003.
15. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *Proc. 25th ACM Symp. on Principles of Prog. Lang.*, pages 85–97. ACM Press, Jan. 1998.
16. G. Necula. Proof-carrying code. In *Proc. 24th ACM Symp. on Principles of Prog. Lang.*, pages 106–119, New York, Jan. 1997. ACM Press.
17. G. Necula. *Compiling with Proofs*. PhD thesis, School of Computer Science, Carnegie Mellon Univ., Sept. 1998.
18. G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proc. 2nd USENIX Symp. on Operating System Design and Impl.*, pages 229–243, 1996.
19. G. C. Necula and R. R. Schneck. A sound framework for untrustred verification-condition generators. In *Proc. 18th Annual IEEE Symposium on Logic in Computer Science*, pages 248–260, June 2003.
20. 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.
21. F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In *Proc. 16th International Conference on Automated Deduction*, volume 1632 of *LNCS*, pages 202–206. Springer-Verlag, July 1999.
22. K. N. Swadi and A. W. Appel. Typed machine language and its semantics. Unpublished manuscript available at `www.cs.princeton.edu/~appel/papers`, July 2001.
23. G. Tan, A. W. Appel, K. N. Swadi, and D. Wu. Construction of a semantic model for a typed assembly language. In *5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '04)*, page (to appear), Jan. 2004.
24. The Coq Development Team. The Coq proof assistant reference manual. The Coq release v7.1, Oct. 2001.
25. B. Werner. *Une Théorie des Constructions Inductives*. PhD thesis, A L'Université Paris 7, Paris, France, 1994.
26. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.
27. D. Yu, N. A. Hamid, and Z. Shao. Building certified libraries for PCC: Dynamic storage allocation. In *Proc. 2003 European Symposium on Programming (ESOP'03)*, April 2003.