A Syntactic Approach to Foundational Proof-Carrying Code*

Nadeem A. Hamid Zhong Shao Valery Trifonov Stefan Monnier Zhaozhong Ni
Department of Computer Science, Yale University
New Haven, CT 06520-8285, U.S.A.
{hamid-nadeem,shao,trifonov,monnier,ni-zhaozhong}@cs.yale.edu

Abstract

Proof-Carrying Code (PCC) is a general framework for verifying the safety properties of machine-language programs. PCC proofs are usually written in a logic extended with language-specific typing rules. In Foundational Proof-Carrying Code (FPCC), on the other hand, proofs are constructed and verified using strictly the foundations of mathematical logic, with no type-specific axioms. FPCC is more flexible and secure because it is not tied to any particular type system and it has a smaller trusted base.

Foundational proofs, however, are much harder to construct. Previous efforts on FPCC all required building sophisticated semantic models for types. In this paper, we present a syntactic approach to FPCC that avoids the difficulties of previous work. Under our new scheme, the foundational proof for a typed machine program simply consists of the typing derivation plus the formalized syntactic soundness proof for the underlying type system. We give a translation from a typed assembly language into FPCC and demonstrate the advantages of our new system via an implementation in the Coq proof assistant.

1. Introduction

Proof-Carrying Code (PCC), as pioneered by Necula and Lee [17, 15], allows a code producer to provide a machine-language program to a host along with a formal proof of its safety. The proof can be mechanically checked by the host and the producer need not be trusted because a valid proof is a dependable certificate of safety.

The proofs in Necula's PCC systems [16, 6] are written in a logic extended with many language-specific typing rules. They can guarantee safety only if there are no bugs in the verification-condition generator (VCgen), the typing rules, and the proof checker. The VCgen is fairly large, so establishing its full correctness is a daunting task. The typing rules are also error-prone: League *et al.* [11] recently discovered a serious bug in the Special J typing rules that undermines the integrity of the entire PCC-based system.

Foundational Proof-Carrying Code (FPCC) [4, 3] tackles these problems by constructing and verifying its proofs using strictly the foundations of mathematical logic, with no type-specific axioms. FPCC is more flexible and secure because it is not tied to any particular type system and has a smaller trusted base.

Foundational proofs, however, are much harder to construct. Previous efforts on FPCC [4, 8, 1, 5] required constructing sophisticated semantic models to reason about types. For example, to support contravariant recursive types, Appel and Felty [8] initially decided to model each type as a partial equivalence relation, but later found that building the actual foundational proofs would "require years of effort implementing machine-checked proofs of basic results in computability theory" [5, page 2]. Appel and McAllester [5] later proposed an indexed model which significantly simplified the proofs but still involves tedious reasoning of computation steps. More seriously, none of these approaches can be easily extended to support mutable fields and higher-order polymorphism. In fact, the only known solution to mutable fields was proposed only very recently by Ahmed *et al.* [2]—the proposal involves building a hierarchy of Gödel numberings and making extensive changes to semantic models used in existing FPCC systems [4, 5].

In this paper, we present a syntactic approach to FPCC that avoids all of these difficulties. Under our new scheme, the foundational proof for a typed machine program simply consists of the typing derivation plus the syntactic soundness proof (of the underlying type system). Here the typing derivation can be readily obtained from a type-checker while the syntactic soundness proof is known to be much

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

easier to construct than the semantic soundness proof [24]. Our paper makes the following new contributions:

- Foundational proofs are widely perceived as extremely hard and tedious to construct, partly because existing efforts [4, 8, 1, 5, 2, 21] on FPCC have all adopted the semantic approach (which requires building sophisticated models from first principles). We show that this perception is not true: with a syntactic approach, constructing foundational proofs is much simpler and more straightforward.
- As far as we know, our work is the first comprehensive study on how to use a syntactic approach to generate FPCC. The idea that attaching the soundness proof (for the underlying type system) can reduce the trusted base is not new [16, 3], however, none of the existing work has shown how to use the syntactic proof to build the foundational proof. In addition, we show in Sections 3 and 4 that naïvely combining existing typed assembly languages (TAL) [14, 13, 25] with their soundness proofs do not necessarily produce valid FPCC.
- The relationship between TAL [14] and PCC [17] has never been made precise even though the two are considered as related approaches for certifying low-level code. In Section 5 we show how to translate each well-typed program in a non-trivial TAL into FPCC. The translation is interesting because it not only shows the connection between the two but also gives new insights on how to turn the expressive invariants in PCC into rich typing constructs in TAL.
- We show that the syntactic approach to FPCC can support recursive types, mutable fields, and first-class code pointers without using complex constructions required by the semantic approaches.
- Finally, independent of our results on FPCC, the typed assembly language presented in Section 4 is interesting on its own. Here our main contribution is a simple technique for type-checking memory allocation and for maintaining invariants about the allocation state.

In the rest of this paper, we first give a formal definition of FPCC (following [3]) in Section 2 and present an overview of the requirements for constructing foundational proofs in Section 3. We then formally define our sample typed assembly language (called FTAL) in Section 4. In Sections 5 and 6 we give the detailed translation from FTAL programs into FPCC and show how to turn FTAL typing derivations and the (syntactic) soundness proof of FTAL into foundational proofs. Finally we compare our approach with the semantic approach, present other related work, and conclude.

2. Foundational Proof-Carrying Code

Unlike type-specialized PCC, foundational PCC avoids any commitment to a particular type system. The operational semantics of machine code as well as the concept of safety are defined in a suitably expressive logic. The code producer must provide both the executable code and a proof in the foundational logic that the code satisfies the safety condition. All required concepts and proofs must be explicitly defined based only on the foundations of mathematics.

2.1. The logic

To encode our safety policies and proofs, we use the calculus of inductive constructions (CiC) [22, 19]. CiC is an extension of the calculus of constructions (CC) [7], which is a higher-order typed lambda calculus. CC corresponds to Church's higher-order predicate logic via the Curry-Howard isomorphism [10]. The syntax of CC is:

$$A, B ::= \mathsf{Set} \mid \mathsf{Type} \mid X \mid \lambda X{:}A.\, B \mid A\, B \mid \Pi X{:}A.\, B$$

The λ term corresponds to the abstraction of the lambda calculus, and the Π term is a dependent product type. When the bound variable does not occur in the body, the product type is usually abbreviated as $A \to B$. In the terminology of pure type systems, Set and Type are the sorts.

CiC, as its name implies, extends the calculus of constructions with inductive definitions. An inductive definition can be written in a syntax similar to that of ML datatypes. For example, the following introduces an inductive specification of natural numbers:

$$\mathsf{Inductive\ Nat} : \mathsf{Set} := \mathsf{zero} : \mathsf{Nat} \mid \mathsf{succ} : \mathsf{Nat} \to \mathsf{Nat}$$

Inductive definitions may also be parameterized as in the following definition of polymorphic lists:

$$\begin{aligned}\mathsf{Inductive\ List}\ [t{:}\mathsf{Set}] : \mathsf{Set} := \mathsf{nil} &: \mathsf{List}\, t \\ \mid \mathsf{cons} &: t \to \mathsf{List}\, t \to \mathsf{List}\, t\end{aligned}$$

The logic also provides elimination constructs for inductive definitions, which combine case analysis with a fixpoint operation. Objects of an inductive type can thus be iterated over using these constructs. In order for the induction to be well-founded and for iterators to terminate, a few constraints are imposed on the shape of inductive definitions. Mutually inductive types are also supported.

CiC has been shown to be strongly normalizing [23], hence the corresponding logic is consistent. It is supported by the Coq proof assistant [22], which we use to implement a prototype system of the results presented in this paper.

In the remainder of this paper, we will use more familiar mathematical notation to present the statement of propositions, rather than the strict definition of CiC syntax given in this section. For example, the application of two terms will be written as $A(B)$ and inductive definitions will be presented in BNF format. We will, however, retain the Π notation, which can generally be read as a universal quantifier.

$$\begin{aligned}
\overline{r} \in \mathit{Regnum} &= \{\, \overline{\mathsf{r0}},\ \overline{\mathsf{r1}},\ \ldots\ \overline{\mathsf{r31}} \,\} \\
w,\ pc \in \mathit{Word} &= \{\, 0,\ 1,\ \ldots \,\} \\
M \in \mathit{Mem} &= \mathit{Word} \to \mathit{Word} \\
\overline{R} \in \mathit{Regfile} &= \mathit{Regnum} \to \mathit{Word} \\
S \in \mathit{State} &= \mathit{Mem} \times \mathit{Regfile} \times \mathit{Word}
\end{aligned}$$

$$\begin{aligned}
\mathit{Instr} \ni\ \overline{\imath} ::= {} & \mathtt{add}\ \overline{r}_d, \overline{r}_s, \overline{r}_t \mid \mathtt{addi}\ \overline{r}_d, \overline{r}_s, w \\
& \mid \mathtt{movi}\ \overline{r}_d, w \mid \mathtt{bgt}\ \overline{r}_s, \overline{r}_t, w \mid \mathtt{jd}\ w \mid \mathtt{jmp}\ \overline{r} \\
& \mid \mathtt{ld}\ \overline{r}_d, \overline{r}_s(w) \mid \mathtt{st}\ \overline{r}_d(w), \overline{r}_s \mid \mathtt{illegal}
\end{aligned}$$

Figure 1. Machine state.

2.2. The machine

The machine is defined by a *machine state* and a step function describing the (deterministic) transition from one machine state to the next. Figure 1 defines the set of machine states. To simplify the presentation, we use an idealized 32-register word-addressed machine with an unbounded memory of words of unlimited size. A machine state is defined as a tuple of a memory, a register set, and a program counter. The figure shows also the instruction set. Informally, the instructions have the following effects:

$\mathtt{add}\ \overline{r}_d, \overline{r}_s, \overline{r}_t$	set $\overline{r}_d$ to the sum of the contents of $\overline{r}_s$ and $\overline{r}_t$;
$\mathtt{addi}\ \overline{r}_d, \overline{r}_s, w$	set $\overline{r}_d$ to the sum of w and the contents of $\overline{r}_s$;
$\mathtt{movi}\ \overline{r}_d, w$	move an immediate value w into $\overline{r}_d$;
$\mathtt{bgt}\ \overline{r}_s, \overline{r}_t, w$	branch to location w if $\overline{r}_s > \overline{r}_t$;
$\mathtt{jd}\ w$	unconditional jump to location w;
$\mathtt{jmp}\ \overline{r}$	indirect jump to the address in register $\overline{r}$;
$\mathtt{ld}\ \overline{r}_d, \overline{r}_s(w)$	load the contents of location $\overline{r}_s + w$ into $\overline{r}_d$;
$\mathtt{st}\ \overline{r}_d(w), \overline{r}_s$	store the contents of $\overline{r}_s$ into location $\overline{r}_d + w$;
$\mathtt{illegal}$	put the machine in an infinite loop.

Of course, these instructions are actually encoded as words (integers) in the machine state. We define *Instr* as an inductive type for reasons of convenience since its constructors are much easier to manipulate than encoded instruction words. Thus, the step function is decomposed into a decoding function and the specification of the machine's operational semantics. The decoding function Dc, of type *Word* $\to$ *Instr*, decodes a word into the appropriate element of *Instr* (non-decodable words will result in an `illegal` instruction); we will omit its exact definition since it is verbose but not interesting. The semantics of instructions is described by the function Step shown in Figure 2. This function is easily defined formally in CiC as an iterator on the *Instr* type.

2.3. The safety condition

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

if $\mathsf{Dc}(M(pc)) =$	then $\mathsf{Step}(M, \overline{R}, pc) =$
$\mathtt{add}\ \overline{r}_d, \overline{r}_s, \overline{r}_t$	$(M, \overline{R}\{\overline{r}_d \mapsto \overline{R}(\overline{r}_s) + \overline{R}(\overline{r}_t)\}, pc{+}1)$
$\mathtt{addi}\ \overline{r}_d, \overline{r}_s, w$	$(M, \overline{R}\{\overline{r}_d \mapsto \overline{R}(\overline{r}_s) + w\}, pc{+}1)$
$\mathtt{movi}\ \overline{r}_d, w$	$(M, \overline{R}\{\overline{r}_d \mapsto w\}, pc{+}1)$
$\mathtt{bgt}\ \overline{r}_s, \overline{r}_t, w$	$(M, \overline{R}, pc{+}1)$, when $\overline{R}(\overline{r}_s) \le \overline{R}(\overline{r}_t)$ $(M, \overline{R}, w)$, when $\overline{R}(\overline{r}_s) > \overline{R}(\overline{r}_t)$
$\mathtt{jd}\ w$	$(M, \overline{R}, w)$
$\mathtt{jmp}\ \overline{r}$	$(M, \overline{R}, \overline{R}(\overline{r}))$
$\mathtt{ld}\ \overline{r}_d, \overline{r}_s(w)$	$(M, \overline{R}\{\overline{r}_d \mapsto M(\overline{R}(\overline{r}_s){+}w)\}, pc{+}1)$
$\mathtt{st}\ \overline{r}_d(w), \overline{r}_s$	$(M\{\overline{R}(\overline{r}_d){+}w \mapsto \overline{R}(\overline{r}_s)\}, \overline{R}, pc{+}1)$
$\mathtt{illegal}$	$(M, \overline{R}, pc)$

Figure 2. Machine semantics.

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

For this presentation, we will define a very basic and simple safety policy which states that the machine is not stuck on an illegal instruction:

$$\mathsf{SP}\,(M, \overline{R}, pc) = (\mathsf{Dc}\,(M\,(pc)) \neq \mathtt{illegal})$$

In practice, the safety policy may also include more complex constraints, such as access control on memory regions.

An FPCC code producer must thus supply an initial state S_0 (which includes the machine code of the program), and a proof A that this state satisfies the safety condition. Via the Curry-Howard isomorphism, A can be represented by a term of type $\mathsf{Safe}\,(S_0)$. Thus, the FPCC package is a pair:

$$F = (S_0 : \mathit{State},\ A : \mathsf{Safe}\,(S_0)).$$

3. Generating Proofs

The actual proof of safety is organized following the approach used by Appel *et al.* [4, 5]. We construct an induction hypothesis *Inv*, also known as the global invariant, which holds for all states reachable from the initial state and is strong enough to imply safety. Then, to show that our initial state S_0 is safe, we provide proofs for the propositions:

Initial Condition: $\mathit{Inv}\,(S_0)$
Preservation: $\Pi S{:}\mathit{State}.\ \mathit{Inv}\,(S) \to \mathit{Inv}\,(\mathsf{Step}\,(S))$
Progress: $\Pi S{:}\mathit{State}.\ \mathit{Inv}\,(S) \to \mathsf{SP}\,(S)$

These propositions intuitively state that our invariant holds for the initial state, and for every subsequent state during the execution. The Progress establishes that whenever the invariant holds, the safety policy of the machine is also satisfied. Together, these imply that during the execution of the program the safety policy will never be violated. To prove the initial state is safe, first we use the Initial Condition and Preservation, and show by induction that

$$\Pi n{:}\mathsf{Nat}.\ \mathit{Inv}\,(\mathsf{Step}^n\,(S_0)).$$

Then $\mathsf{Safe}\,(S_0)$ follows directly by Progress.

Unlike Appel *et al.*, who construct the invariant by means of a semantic model of types at the machine level, our approach is based on the use of type soundness [24]: We define $\mathit{Inv}(S)$ to mean that S is "well-formed" syntactically. The well-formedness property must be preserved by the step function, and must imply safety; the proofs of these properties are encoded in the FPCC logic as proof terms for Preservation and Progress.

In the following sections we show how to derive the notion of well-formedness for a machine state by relating the state to a type-correct *program* in a typed assembly language. The type system of the language defines a set of inference rules for judgments of the form $\vdash P$, meaning that the program P is well-formed (type-correct). The dynamic semantics of the language specifies an evaluation relation $\longmapsto$ on programs; we use here the term "program" to denote not only code but a more general configuration fully representing a stage of the evaluation.

The central idea of our approach to FPCC is to find a typed assembly language and a translation relation $\Rightarrow$ between its programs and machine states, such that type-correct programs are mapped to well-formed states, and the evaluation relation is related to the step function—that is, if $P \Rightarrow S$ and $P \longmapsto P'$, then $P' \Rightarrow \mathsf{Step}(S)$. If these properties hold, we can define the invariant $\mathit{Inv}(S)$ as simply stating that there exists a type-correct program P such that $P \Rightarrow S$. Then the formal proofs of progress and preservation for the type system can be used to construct straightforward proofs of the corresponding propositions needed for the safety proof for S_0. Further details of the construction of proof terms are provided in Section 5.

This method imposes requirements on the design of the typed assembly language other than just having a sound type system. For the approach we follow in this paper, if the assembly language has "macro" instructions (*e.g.* `malloc` [14, 13] and `newarray` [25], which "expand" into sequences of several machine instructions), the well-formedness of the assembly program alone will be insufficient for the construction of the global invariant. This is because *Inv* must hold for all machine states reachable from S_0. For the intermediate states of the execution of a macro instruction there are no corresponding well-formed assembly programs. Hence, each one of the assembly instructions must correspond to exactly one machine instruction. Note that this exact correspondence of instructions is not necessary in general for the syntactic approach to work, but it facilitates the definition of the invariant and allows for a simpler presentation.

4. Featherweight Typed Assembly Language

The source language that we will be compiling to FPCC is a version of the typed assembly language (TAL) by Morrisett *et al.* [14]. The approach developed in this paper can be applied to a TAL-like language extended with higher-order kinds and recursive types. For simplicity, we only introduce here a subset of such a language, which we call Featherweight Typed Assembly Language (FTAL). It does not include polymorphism, existential types, and higher-order kinds. However, it does support recursive types, memory allocation, and mutable records (tuples).

(*type*)	τ	$::= \alpha \mid \mathsf{int} \mid \forall[].\Gamma \mid \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \mid \mu\alpha.\tau$
(*init flag*)	φ	$::= \mathsf{0} \mid \mathsf{1}$
(*heap ty*)	Ψ	$::= \{\mathbf{0}:\tau_0, \ldots, \mathbf{n}:\tau_n\}$
(*alloc pt ty*)	ρ	$::= \mathsf{fresh} \mid \mathsf{used}(n)$
(*regfile ty*)	Γ	$::= \{r_0:\tau_0, \ldots, r_n:\tau_n, \mathsf{r31}:\rho\}$
(*label*)	l	$::= \mathbf{0} \mid \mathbf{1} \mid \ldots$
(*user reg*)	r	$::= \mathsf{r0} \mid \mathsf{r1} \mid \ldots \mid \mathsf{r30}$
(*all reg*)	$\hat{r}$	$::= r \mid \mathsf{r31}$
(*word val*)	v	$::= l \mid i \mid ?\tau \mid \mathsf{fold}\ v\ \mathsf{as}\ \tau$
(*heap val*)	h	$::= \langle v_1, \ldots, v_n \rangle \mid \mathsf{code}[]\Gamma.I$
(*heap*)	H	$::= \{\mathbf{0} \mapsto h_0, \ldots, \mathbf{n} \mapsto h_n\}$
(*regfile*)	R	$::= \{\mathsf{r0} \mapsto v_0, \ldots, \mathsf{r31} \mapsto v_{31}\}$
(*instr*)	ι	$::= \mathsf{add}\ r_d, r_s, r_t \mid \mathsf{addi}\ r_d, r_s, i \mid \mathsf{alloc}\ r_d[\overline{\tau}] \mid \mathsf{bgt}\ r_s, r_t, l \mid \mathsf{bump}\ i \mid \mathsf{fold}\ r_d[\tau], r_s \mid \mathsf{ld}\ r_d, r_s(i) \mid \mathsf{mov}\ r_d, r_s \mid \mathsf{movi}\ r_d, i \mid \mathsf{movl}\ r_d, l \mid \mathsf{st}\ r_d(i), r_s \mid \mathsf{unfold}\ r_d, r_s$
(*instr seq*)	I	$::= \iota; I \mid \mathsf{jd}\ l \mid \mathsf{jmp}\ r$
(*program*)	P	$::= (H, R, I)$

Figure 3. Syntax of FTAL.

For most FTAL instructions it is easy to see there is a one-to-one mapping to the machine instructions of Section 2.2. However, having a malloc "macro instruction" in FTAL (as in TAL) will not work because it cannot be mapped to a single machine instruction and will not satisfy our requirements for generating FPCC proofs, since there would be no corresponding FTAL state between the expanded machine instructions. (See Section 4.5 for details on this issue.) Our approach is to make the memory allocation model explicit and split the malloc instruction into, in this case, two individual instructions.

4.1. Syntax

We present the syntax of FTAL in Figure 3. As in TAL, the abstract machine state consists of a heap H, a register file R, and a sequence of instructions I. The heap maps labels l to heap values h, and the register file maps registers $\hat{r}$ to word values v. The notation $H\{l \mapsto h\}$ represents a

$(H, R, I) \longmapsto P$ where	
if $I =$	then $P =$
add $r_d, r_s, r_t; I'$	$(H, R\{r_d \mapsto R(r_s) + R(r_t)\}, I')$
addi $r_d, r_s, i; I'$	$(H, R\{r_d \mapsto R(r_s) + i\}, I')$
alloc $r_d[\vec{\tau}]; I'$	$(H', R\{r_d \mapsto l\}, I')$ where $\vec{\tau} = \tau_1, \ldots, \tau_n$, $R(\mathsf{r31}) = l$, and $H' = H\{l \mapsto \langle ?\tau_1, \ldots, ?\tau_n \rangle\}$
bgt $r_s, r_t, l; I'$	(H, R, I') when $R(r_s) \leq R(r_t)$; and (H, R, I'') when $R(r_s) > R(r_t)$ where $H(l) = \mathsf{code}[]\Gamma.I''$
bump $i; I'$	$(H, R\{\mathsf{r31} \mapsto \lvert H \rvert\}, I')$
fold $r_d[\tau], r_s; I'$	$(H, R\{r_d \mapsto \mathsf{fold}\ R(r_s)\ \mathsf{as}\ \tau\}, I')$
jd l	(H, R, I') where $H(l) = \mathsf{code}[]\Gamma.I'$
jmp r	(H, R, I') where $H(R(r)) = \mathsf{code}[]\Gamma.I'$
ld $r_d, r_s(i); I'$	$(H, R\{r_d \mapsto v_i\}, I')$ where $0 \leq i < n$ $H(R(r_s)) = \langle v_0, \ldots, v_{n-1} \rangle$
mov $r_d, r_s; I'$	$(H, R\{r_d \mapsto R(r_s)\}, I')$
movi $r_d, i; I'$	$(H, R\{r_d \mapsto i\}, I')$
movl $r_d, l; I'$	$(H, R\{r_d \mapsto l\}, I')$
st $r_d(i), r_s; I'$	$(H\{l \mapsto h\}, R, I')$ where $0 \leq i < n$ $R(r_d) = l$, $H(l) = \langle v_0, \ldots, v_{n-1} \rangle$, and $h = \langle v_0, \ldots, v_{i-1}, R(r_s), v_{i+1}, \ldots, v_{n-1} \rangle$
unfold $r_d, r_s; I'$	$(H, R\{r_d \mapsto v\}, I')$ where $R(r_s) = \mathsf{fold}\ v\ \mathsf{as}\ \tau$

Figure 4. Operational semantics of FTAL.

heap which maps l to h, and on all other labels agrees with H. Similar notation is used for heap types, register files, and register file types. In (*regfile ty*), $n < 31$, and not all user registers need appear in the type. The notation $|H|$ and $|\Psi|$ is used to represent the number of labels in the heap and heap type, respectively. Only tuples and code blocks are stored in the heap and thus these are the heap values. Word values include labels (of heap values), integers, recursive data, and junk values which are used by the operational semantics to represent uninitialized tuple elements.

Our memory model is a simple linear unbounded heap with an allocation pointer pointing to the heap top, initially set to the bottom of the heap space. Memory allocation consists of copying the current allocation pointer to a register using alloc and then adjusting the allocation pointer with bump. In Section 5.2 we will see how these two instructions can be directly translated into one FPCC machine instruction each. One of the general registers, r31, is reserved as the allocation pointer register, tracking the amount of allocated memory. FTAL instructions will only explicitly refer to the first 31 "user" registers (r). To meaningfully implement linear allocation, we need an ordering on memory labels, so we define labels as natural numbers. To determine whether a label has been allocated, it is compared with $|H|$.

The types of FTAL are integers, code, tuple types annotated with initialization flags, and recursive types. Operations on recursive types in FTAL are supported by the fold and unfold instructions. The remaining instructions (add, addi, bgt, mov, movi, movl, ld, and st) are equivalent or similar to those in the original TAL. A code block is a sequence of instructions, with specified initial register types. Code blocks always end with a jmp or jd instruction.

Judgment	Meaning
$\vdash \tau$	τ is a well-formed type
$\vdash \Psi$	Ψ is a well-formed heap type
$\vdash \Gamma$	Γ is a well-formed regfile type
$\vdash \tau_1 \leq \tau_2$	τ_1 is a subtype of τ_2
$\vdash \Gamma_1 \subseteq \Gamma_2$	Γ_1 is a regfile subtype of Γ_2
$\vdash P$	P is a well-formed program
$\vdash H : \Psi$	H is a well-formed heap of type Ψ
$\Psi \vdash R : \Gamma$	R is a well-formed regfile of type Γ
$\Psi \vdash l : \rho$	l is a label of allocation status ρ
$\Psi \vdash h : \tau$ hval	h is a well-formed heap value of type τ
$\Psi \vdash v : \tau$	v is a well-formed word value of type τ
$\Psi \vdash v : \tau^{\varphi}$	v is a well-formed word value of type τ^{φ}
$\Psi; \Gamma \vdash I$	I is a well-formed instruction sequence

Figure 5. Static judgments.

4.2. Dynamic semantics

The operational semantics of FTAL is presented in Figure 4. Most of the instructions have an intuitively clear meaning. The ld and st instructions load from and store to a tuple in the heap using the specified index. The instruction bgt r_s, r_t, l tests whether the value in r_s is larger than that in r_t, and, if so, transfers control to the code block at l.

In order to allocate a tuple in the heap, first the alloc instruction is used to copy the current heap allocation pointer to r_d and allocate the desired size in the heap. Before the next allocation, the allocation pointer needs to be adjusted. This is achieved using the bump instruction, which sets the allocation pointer to the next unused region of the heap, as described earlier. (The i argument is not used by the operational semantics.) Since we assume a linear allocation method, unused regions of the heap are simply all those beyond the currently allocated data.

The fold instruction annotates the value of r_s with the recursive type and moves it into r_d, while unfold extracts the value from the recursive package in r_s into r_d. Note that the fold and unfold instructions of FTAL (as well as TAL) are not no-ops but copy a value from one register to another.

4.3. Static semantics

The primary judgment of the static semantics is that of the well-formedness of a program. That in turn depends on judgments of the well-formedness of the heap, heap type,

$\boxed{\vdash P \qquad \vdash H:\Psi \qquad \Psi \vdash R:\Gamma}$

$$\frac{\vdash H:\Psi \qquad \Psi \vdash R:\Gamma \qquad \Psi;\Gamma \vdash I \qquad \exists l \in Dom(H).H(l) = \mathsf{code}[]\Gamma'.I' \textit{ and } I \subseteq I'}{\vdash (H,R,I)} \text{ (PROG)}$$

$$\frac{\vdash \Psi \quad |\Psi| = |H| \quad \Psi \vdash H(l):\Psi(l)\ \mathsf{hval} \quad (\mathbf{0} \le l < |H|)}{\vdash H:\Psi} \text{ (HEAP)} \qquad \frac{\Psi \vdash R(r_i):\tau_i \quad (0 \le i \le n) \quad \Psi \vdash R(\mathsf{r31}):\rho \quad \forall r \in Dom(R) - \{\mathsf{r31}\}.\textit{if } R(r) = l \textit{ then } l < |\Psi|}{\Psi \vdash R:\{r_0:\tau_0,\ldots,r_n:\tau_n,\mathsf{r31}:\rho\}} \text{ (REG)}$$

$\boxed{\vdash \tau \qquad \vdash \Psi \qquad \vdash \Gamma \qquad \vdash \tau_1 \le \tau_2 \qquad \vdash \Gamma_1 \subseteq \Gamma_2}$

$$\frac{FTV(\tau) = \emptyset}{\vdash \tau} \text{ (TYPE)} \qquad \frac{\vdash \tau_i \qquad (1 \le i \le n)}{\vdash \{\mathbf{0}:\tau_0,\ldots,\mathbf{n}:\tau_n\}} \text{ (HTYPE)} \qquad \frac{\vdash \tau_i \qquad (1 \le i \le n) \qquad n < 31}{\vdash \{r_0:\tau_0,\ldots,r_n:\tau_n,\mathsf{r31}:\rho\}} \text{ (RFTYPE)}$$

$$\frac{\vdash \tau}{\vdash \tau \le \tau} \text{ (REFLEX)} \qquad \frac{\vdash \tau_i \qquad (1 \le i \le n)}{\vdash \langle \tau_1^{\varphi_1},\ldots,\tau_{i-1}^{\varphi_{i-1}},\tau_i^{\mathsf{1}},\tau_{i+1}^{\varphi_{i+1}},\ldots,\tau_n^{\varphi_n}\rangle \le \langle \tau_1^{\varphi_1},\ldots,\tau_{i-1}^{\varphi_{i-1}},\tau_i^{\mathsf{0}},\tau_{i+1}^{\varphi_{i+1}},\ldots,\tau_n^{\varphi_n}\rangle} \text{ (0-1)}$$

$$\frac{\vdash \tau_1 \le \tau_2 \qquad \vdash \tau_2 \le \tau_3}{\vdash \tau_1 \le \tau_3} \text{ (TRANS)} \qquad \frac{\vdash \tau_i \qquad (m \ge n) \qquad (0 \le i \le m)}{\vdash \{r_0:\tau_0,\ldots,r_m:\tau_m,\mathsf{r31}:\rho\} \subseteq \{r_0:\tau_0,\ldots,r_n:\tau_n,\mathsf{r31}:\rho\}} \text{ (WEAKEN)}$$

$\boxed{\Psi \vdash h:\tau\ \mathsf{hval} \qquad \Psi \vdash v:\tau \qquad \Psi \vdash l:\rho \qquad \Psi \vdash v:\tau^{\varphi}}$

$$\frac{\Psi \vdash v_i:\tau_i^{\varphi_i} \qquad (1 \le i \le n)}{\Psi \vdash \langle v_1,\ldots,v_n\rangle:\langle \tau_1^{\varphi_1},\ldots,\tau_n^{\varphi_n}\rangle\ \mathsf{hval}} \text{ (TUPLE)} \qquad \frac{\vdash \Gamma \qquad \Psi;\Gamma \vdash I}{\Psi \vdash \mathsf{code}[]\Gamma.I:\forall[].\Gamma\ \mathsf{hval}} \text{ (CODE)} \qquad \frac{}{\Psi \vdash i:\mathsf{int}} \text{ (INT)}$$

$$\frac{\Psi \vdash v:\tau[\mu\alpha.\tau/\alpha]}{\Psi \vdash \mathsf{fold}\ v\ \mathsf{as}\ \mu\alpha.\tau:\mu\alpha.\tau} \text{ (FOLD)} \qquad \frac{\vdash \Psi(l) \le \tau}{\Psi \vdash l:\tau} \text{ (LABEL)} \qquad \frac{l = |\Psi|}{\Psi \vdash l:\mathsf{fresh}} \text{ (FRESH)}$$

$$\frac{l = |\Psi| - \mathbf{1} \qquad \Psi \vdash l:\langle \tau_1^{\varphi_1},\ldots,\tau_n^{\varphi_n}\rangle}{\Psi \vdash l:\mathsf{used}(n)} \text{ (USED)} \qquad \frac{\Psi \vdash v:\tau}{\Psi \vdash v:\tau^{\varphi}} \text{ (INIT)} \qquad \frac{\vdash \tau}{\Psi \vdash ?\tau:\tau^{\mathsf{0}}} \text{ (UNINIT)}$$

Figure 6. Static semantics of FTAL.

register file, register file type, and instruction sequence. The various typing judgments are summarized in Figure 5.

The complete rules of the FTAL static semantics are given in Figures 6 and 7. To have a well-formed program, the heap and register file must be well-formed in some appropriate environments, as must be the current instruction sequence. Additionally, the current instruction sequence must be present in the heap. The notation $I \subseteq I'$ means that I is a suffix of I'. For a heap to be well-formed the domain of the heap type must be the same as that of the heap, and each heap value must be well-formed. However, the type of a well-formed register file need only specify a subset of the registers in its domain. Subtyping is used for two purposes: one to allow a code block to be called when the current register file type is more detailed than needed, and the other to be able to type-check the initialization of an uninitialized tuple element as described below.

The special allocation register is typed using a new judgment of allocation status, defined by the two rules (FRESH) and (USED). In the first typing rule, a label whose value is equivalent to the size of the heap type must necessarily be unallocated. When allocation takes place, then the allocation register temporarily points to the newly allocated memory, and thus will have allocation status $\mathsf{used}(n)$ where n is the length of the allocated tuple. The assignment of allocation status interacts with the two novel FTAL instructions, alloc and bump, as shown in their typing rules in Figure 7. For an alloc instruction to be well-typed, the allocation register, r31, must be in the fresh status, since otherwise, as can be seen from the operational semantics, the previously allocated data will be overwritten. After the alloc instruction, the remainder of the instruction sequence is checked with the status of r31 changed to $\mathsf{used}(n)$. No further allocation can take place until a bump instruction is encountered, which resets the status to fresh.

4.4. Soundness

In order to produce the necessary FPCC proofs as described in Section 3, we must encode the complete semantics of FTAL in CiC along with its proof of soundness,

$\boxed{\Psi;\Gamma\vdash I}$

$$\frac{\Gamma(r_s)=\mathsf{int}\qquad\Gamma(r_t)=\mathsf{int}\qquad\Psi;\Gamma\{r_d:\mathsf{int}\}\vdash I}{\Psi;\Gamma\vdash \mathsf{add}\ r_d,r_s,r_t;I}\ (\textsc{add})\qquad \frac{\Gamma(r_s)=\mathsf{int}\qquad\Psi;\Gamma\{r_d:\mathsf{int}\}\vdash I}{\Psi;\Gamma\vdash \mathsf{addi}\ r_d,r_s,i;I}\ (\textsc{addi})$$

$$\frac{\vdash\tau_i\quad\Psi;\Gamma\{r_d:\langle\tau_1^{\mathsf{0}},\ldots,\tau_n^{\mathsf{0}}\rangle\}\{\mathsf{r31}:\mathsf{used}(n)\}\vdash I}{\Psi;\Gamma\{\mathsf{r31}:\mathsf{fresh}\}\vdash \mathsf{alloc}\ r_d[\tau_1,\ldots,\tau_n];I}\ (\textsc{alloc})\qquad \frac{\Psi;\Gamma\{\mathsf{r31}:\mathsf{fresh}\}\vdash I}{\Psi;\Gamma\{\mathsf{r31}:\mathsf{used}(n)\}\vdash \mathsf{bump}\ n;I}\ (\textsc{bump})$$

$$\frac{\Gamma(r_s)=\mathsf{int}\qquad\Gamma(r_t)=\mathsf{int}\qquad\Psi(l)=\forall[].\Gamma'\qquad\vdash\Gamma\subseteq\Gamma'\qquad\Psi;\Gamma\vdash I}{\Psi;\Gamma\vdash \mathsf{bgt}\ r_s,r_t,l;I}\ (\textsc{bgt})$$

$$\frac{\Psi;\Gamma\{r_d:\Gamma(r_s)\}\vdash I}{\Psi;\Gamma\vdash \mathsf{mov}\ r_d,r_s;I}\ (\textsc{mov})\qquad \frac{\Psi;\Gamma\{r_d:\mathsf{int}\}\vdash I}{\Psi;\Gamma\vdash \mathsf{movi}\ r_d,i;I}\ (\textsc{movi})\qquad \frac{\Psi;\Gamma\{r_d:\tau\}\vdash I\quad\vdash\Psi(l)\leq\tau}{\Psi;\Gamma\vdash \mathsf{movl}\ r_d,l;I}\ (\textsc{movl})$$

$$\frac{\Gamma(r_s)=\langle\tau_0^{\varphi_0},\ldots,\tau_{i-1}^{\varphi_{i-1}},\tau_i^{\mathsf{1}},\tau_{i+1}^{\varphi_{i+1}},\ldots,\tau_{n-1}^{\varphi_{n-1}}\rangle\qquad\Psi;\Gamma\{r_d:\tau_i\}\vdash I\qquad(0\leq i<n)}{\Psi;\Gamma\vdash \mathsf{ld}\ r_d,r_s(i);I}\ (\textsc{ld})$$

$$\frac{\Gamma(r_s)=\tau_i\qquad\Gamma(r_d)=\langle\tau_0^{\varphi_0},\ldots,\tau_{n-1}^{\varphi_{n-1}}\rangle\qquad\Psi;\Gamma\{r_d:\langle\tau_0^{\varphi_0},\ldots,\tau_{i-1}^{\varphi_{i-1}},\tau_i^{\mathsf{1}},\tau_{i+1}^{\varphi_{i+1}},\ldots,\tau_{n-1}^{\varphi_{n-1}}\rangle\}\vdash I\qquad(0\leq i<n)}{\Psi;\Gamma\vdash \mathsf{st}\ r_d(i),r_s;I}\ (\textsc{st})$$

$$\frac{\Gamma(r_s)=\tau[\mu\alpha.\tau/\alpha]\qquad\Psi;\Gamma\{r_d:\mu\alpha.\tau\}\vdash I}{\Psi;\Gamma\vdash \mathsf{fold}\ r_d[\mu\alpha.\tau],r_s;I}\ (\textsc{fold-i})\qquad \frac{\Gamma(r_s)=\mu\alpha.\tau\qquad\Psi;\Gamma\{r_d:\tau[\mu\alpha.\tau/\alpha]\}\vdash I}{\Psi;\Gamma\vdash \mathsf{unfold}\ r_d,r_s;I}\ (\textsc{unfold})$$

$$\frac{\Psi(l)=\forall[].\Gamma'\qquad\vdash\Gamma\subseteq\Gamma'}{\Psi;\Gamma\vdash \mathsf{jd}\ l}\ (\textsc{jd})\qquad \frac{\Gamma(r)=\forall[].\Gamma'\qquad\vdash\Gamma\subseteq\Gamma'}{\Psi;\Gamma\vdash \mathsf{jmp}\ r}\ (\textsc{jmp})$$

Figure 7. Well-formedness of FTAL instruction sequences.

which will be used in defining and proving the FPCC propositions. The critical theorems for the soundness of FTAL are the usual progress and preservation lemmas:

Theorem 1 (Progress)
If $\vdash P$, then there exists P' such that $P \longmapsto P'$.

Theorem 2 (Preservation)
If $\vdash P$ and $P \longmapsto P'$, then $\vdash P'$.

As usual, several intermediate lemmas are used to prove these two theorems, all of which can be formally encoded and proved in the Coq proof assistant. (See the companion technical report [9] for details.)

Now that we have an assembly language with a sound type system, we are ready to show how to generate proof-carrying code from a well-typed FTAL program.

4.5. Designing TAL for FPCC

We have designed a novel FTAL language for our presentation in this paper which corresponds closely to the underlying machine defined in Section 2.2. As will become clear in the next section, every well-formed FTAL state can be mapped to a safe machine state, and this property is used to produce a safety proof for the machine state. For safety policies which need to enforce complex constraints on every machine state or step, such a one-to-one mapping can be very important. In general, however, this strict correspondence is not necessary for the syntactic approach to work. For example, if we wished to retain "macro" instructions in FTAL, our FPCC Preservation might be modified to

$$\Pi S{:}\mathit{State}.\,\mathit{Inv}\,(S)\rightarrow\exists n{:}\mathsf{Nat}.\,\mathit{Inv}\,(\mathsf{Step}^{(n+1)}\,(S))$$

stating that starting from a state satisfying the global invariant, the machine will eventually (after one or more steps) reach another state satisfying the invariant.

Also, when introducing polymorphism or existentials into the FTAL language, there will be certain FTAL operations (*e.g.* type application) which do not correspond to any run-time machine instructions at all. In this case, the FTAL operation would correspond to a "cast" in the FPCC proof for the machine state.

Another reason why naïvely using existing typed assembly languages will not necessarily help in producing FPCC is that the type system must be designed to enforce appropriate invariants. There are requirements in the typing rules of FTAL which are not critical for FTAL soundness but are necessary when translating FTAL to FPCC as described in the next section. An example of this is the requirement in the (REG) rule (Figure 6) that all labels in registers be within

the domain of the heap (including those registers that are not specified in the type of the register file and hence not accessible by well-formed code anyway). This condition is crucial in proving the properties discussed in Section 5.3.

5. Translating FTAL to FPCC

As outlined in Section 2.3, an FPCC package provides an initial state, S_0, and a proof that the state satisfies the safety policy. In the next few subsections, we show how to translate an FTAL program into a machine state and how to use the FTAL type system to generate proofs of the FPCC Preservation and Progress propositions, which imply safety.

5.1. From FTAL to machine state

FTAL programs are compiled to machine code by (1) defining a layout for the memory which maps heap values of the program to memory addresses, (2) translating FTAL instructions to machine instructions, and (3) choosing the appropriate program counter and register values. We will express the correspondence between an FTAL program and a machine state by a family of translation relations upon the various syntactic categories. The forms of these are:

Relation	Correspondence
$(H, R, I) \Rightarrow (M, \overline{R}, pc)$	FTAL program to machine state
$L \vdash H \Rightarrow M$	FTAL heap to memory
$L \vdash R \Rightarrow \overline{R}$	register files
$L \vdash I \Rightarrow_{\mathsf{s}} M[i..j]$	sequence of instructions to memory layout
$L \vdash \iota \Rightarrow_{\mathsf{i}} w$	instruction translation
$L \vdash h \Rightarrow_{\mathsf{h}} M[i..j]$	heap value to memory layout
$L \vdash v \Rightarrow_{\mathsf{w}} w$	word value to machine word

Recall that the machine memory is modeled as a function, *Word* $\rightarrow$ *Word*, so $M(w)$ denotes the memory word at address w. The judgments $L \vdash I \Rightarrow_{\mathsf{s}} M[i..j]$ and $L \vdash h \Rightarrow_{\mathsf{h}} M[i..j]$ state that a sequence of instructions and a heap value, respectively, translate to a series of consecutive words in memory M from address i to address j.

An important step in the translation is flattening the FTAL heap into the machine memory. To achieve this, we define a *Layout* function of type *Heap* $\rightarrow$ *Label* $\rightarrow$ *Word* which, given an FTAL heap, returns a mapping from labels to memory addresses. (In the relations above, L is this *Layout* function applied to the heap.) Thus, we have:

$$\begin{aligned} \mathit{Layout}\,(\{\})\,(l') &= 0 \\ \mathit{Layout}\,(H\{l \mapsto h\})\,(l') &= \begin{cases} w + \mathit{size}\,(h), & \text{if } l < l' \\ w, & \text{otherwise,} \end{cases} \\ & \quad \text{where } w = \mathit{Layout}\,(H)\,(l') \end{aligned}$$

where $\mathit{size}\,(h)$ is the size of the heap value h (n for an n-tuple, and the length of the instruction sequence for a code block). This *Layout* function maps labels to addresses starting at 0 and forces the translation $\Rightarrow$ to lay out FTAL heap values compactly, consecutively, and with no overlapping (due to the implicit constraint that the labels in the heap appear in descending order). Additionally, the first unused label (whose value equals the size of the heap) is mapped to the first unused address. These properties are useful later on in proving Preservation and Progress.

The translation relations are defined by a set of inference rules, given in Figure 8. The rules are straightforward and operate purely on the syntax of FTAL programs. Note that FTAL type annotations are discarded in the translation (for example, in the fold instruction), and label word values are mapped to memory words using the layout function. Each FTAL heap value corresponds to a sequence of words in memory. A heap translates to a memory if every heap value in the heap translates to the appropriate sequence of memory words. Registers translate directly between FTAL and the machine. An FTAL program corresponds to a machine state if the translation relation holds on the heap and register file, and if the current instruction sequence is at some location in the memory. Our choice of the FTAL instruction set allows us to translate every FTAL instruction into one machine instruction word. Notice that the FTAL alloc and bump instructions correspond to machine move and addition instructions, respectively, using the register reserved for allocation, $\overline{\mathsf{r31}}$. (It is for this purpose that bump has an i argument.)

The translation relation as presented in Figure 8 is also not deterministic with respect to the unused and uninitialized parts of the memory and to the positioning of the program counter. However, it is straightforward on the basis of its definition to develop a deterministic function which translates an FTAL program into a machine state for which the translation relation described above holds. In the next section, we will show how this initial translation is used to provide the Initial Condition FPCC proof.

5.2. The global invariant

As discussed in Section 3, in addition to translating the FTAL program to an initial machine state S_0, we must define the invariant *Inv*, which holds during the execution of a machine program, and provide proofs of:

Initial Condition:	$\mathit{Inv}\,(S_0)$
Preservation:	$\Pi S : \mathit{State}.\, \mathit{Inv}\,(S) \rightarrow \mathit{Inv}\,(\mathsf{Step}\,(S))$
Progress:	$\Pi S : \mathit{State}.\, \mathit{Inv}\,(S) \rightarrow \mathsf{SP}\,(S)$

The invariant simply has to ensure that the machine state at each step corresponds to a well-typed FTAL program, which will allow us to use the formalized versions of the proofs of progress and preservation for FTAL to generate formal proofs of the corresponding properties of the invariant. Since the definition of *Inv* requires us to state that an FTAL program is well-typed, it must be expressed not just in terms of FTAL programs, but of their typing derivations:

$$\mathit{Inv}(S) = \exists P : \mathit{program}.\, \exists D : (\vdash P).\, P \Rightarrow S$$

WORD VALUES

$$L \vdash l \Rightarrow_{\mathsf{w}} L(l) \qquad L \vdash i \Rightarrow_{\mathsf{w}} i$$

$$\frac{\textit{for any } w}{L \vdash ?\tau \Rightarrow_{\mathsf{w}} w} \qquad \frac{L \vdash v \Rightarrow_{\mathsf{w}} w}{L \vdash \mathsf{fold}\ v\ \mathsf{as}\ \tau \Rightarrow_{\mathsf{w}} w}$$

INSTRUCTIONS

$$\begin{array}{ll}
L \vdash \mathsf{add}\ r_d, r_s, r_t & \Rightarrow_{\mathsf{i}} \mathtt{add}\ \overline{r}_d, \overline{r}_s, \overline{r}_t \\
L \vdash \mathsf{addi}\ r_d, r_s, i & \Rightarrow_{\mathsf{i}} \mathtt{addi}\ \overline{r}_d, \overline{r}_s, i \\
L \vdash \mathsf{alloc}\ r_d[\vec{\tau}] & \Rightarrow_{\mathsf{i}} \mathtt{addi}\ \overline{r}_d, \overline{\mathsf{r31}}, 0 \\
L \vdash \mathsf{bump}\ i & \Rightarrow_{\mathsf{i}} \mathtt{addi}\ \overline{\mathsf{r31}}, \overline{\mathsf{r31}}, i \\
L \vdash \mathsf{fold}\ r_d[\tau], r_s & \Rightarrow_{\mathsf{i}} \mathtt{addi}\ \overline{r}_d, \overline{r}_s, 0 \\
L \vdash \mathsf{unfold}\ r_d, r_s & \Rightarrow_{\mathsf{i}} \mathtt{addi}\ \overline{r}_d, \overline{r}_s, 0 \\
L \vdash \mathsf{ld}\ r_d, r_s(i) & \Rightarrow_{\mathsf{i}} \mathtt{ld}\ \overline{r}_d, \overline{r}_s(i) \\
L \vdash \mathsf{st}\ r_d(i), r_s & \Rightarrow_{\mathsf{i}} \mathtt{st}\ \overline{r}_d(i), \overline{r}_s \\
L \vdash \mathsf{mov}\ r_d, r_s & \Rightarrow_{\mathsf{i}} \mathtt{addi}\ \overline{r}_d, \overline{r}_s, 0 \\
L \vdash \mathsf{movi}\ r_d, i & \Rightarrow_{\mathsf{i}} \mathtt{movi}\ \overline{r}_d, i \\
L \vdash \mathsf{movl}\ r_d, l' & \Rightarrow_{\mathsf{i}} \mathtt{movi}\ \overline{r}_d, L(l') \\
L \vdash \mathsf{bgt}\ r_s, r_t, l & \Rightarrow_{\mathsf{i}} \mathtt{bgt}\ \overline{r}_s, \overline{r}_t, L(l)
\end{array}$$

INSTRUCTION SEQUENCES

$$\frac{L \vdash \iota \Rightarrow_{\mathsf{i}} \mathsf{Dc}(M(i)) \quad L \vdash I \Rightarrow_{\mathsf{s}} M[(i+1)..j]}{L \vdash \iota; I \Rightarrow_{\mathsf{s}} M[i..j]}$$

$$\frac{\mathsf{Dc}(M(i)) = \mathtt{jd}\ (L(l'))}{L \vdash \mathsf{jd}\ l' \Rightarrow_{\mathsf{s}} M[i..i]} \qquad \frac{\mathsf{Dc}(M(i)) = \mathtt{jmp}\ \overline{r}}{L \vdash \mathsf{jmp}\ r \Rightarrow_{\mathsf{s}} M[i..i]}$$

HEAP VALUES

$$\frac{L \vdash v_i \Rightarrow_{\mathsf{w}} M(j+i) \quad \textit{for } 0 \le i \le n}{L \vdash \langle v_o, \ldots, v_n \rangle \Rightarrow_{\mathsf{h}} M[j..(j+n)]}$$

$$\frac{L \vdash I \Rightarrow_{\mathsf{s}} M[i..j]}{L \vdash \mathsf{code}\ []\Gamma.I \Rightarrow_{\mathsf{h}} M[i..j]}$$

HEAP, REGISTER FILE, PROGRAM

$$\frac{L \vdash H(l) \Rightarrow_{\mathsf{h}} M[L(l)..L(l{+}1){-}1] \quad \textit{for } \mathbf{0} \le l < |H|}{L \vdash H \Rightarrow M}$$

$$\frac{L \vdash R(\hat{r}) \Rightarrow_{\mathsf{w}} \overline{R}(\overline{r})}{L \vdash R \Rightarrow \overline{R}}$$

$$\frac{\begin{array}{c}
\textit{Layout}(H) \vdash H \Rightarrow M \\
\textit{Layout}(H) \vdash R \Rightarrow \overline{R} \\
\textit{Layout}(H) \vdash I \Rightarrow_{\mathsf{s}} M[pc..pc + |I| - 1], \\
\textit{where} \quad \exists l \in Dom(H).(H(l) = \mathsf{code}\ []\Gamma.I', I \subseteq I',\ \textit{and} \\
pc = \textit{Layout}(H)(l) + |I'| - |I|)
\end{array}}{(H, R, I) \Rightarrow (M, \overline{R}, pc)}$$

Figure 8. Relating FTAL programs to machine states.

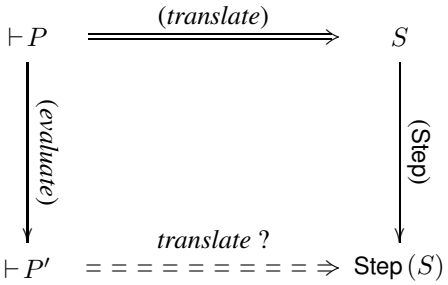

Figure 9. Relationship between FTAL evaluation and machine semantics.

where the type annotation $\vdash P$ in the quantification on D introduces D as a proof term for the judgment $\vdash P$.

The proof of the initial condition can now be obtained directly in the process of translating an initial well-formed FTAL program to machine state as described in Section 5.1. It remains, therefore, to prove the two lemmas.

5.3. The Preservation and Progress properties

Progress in our case is easy to prove: since the invariant states that there exists a well-typed FTAL program which translates to the current state, it is obvious by examination of the translation rules that such an FTAL program will never translate to a state in which the program counter points to an illegal instruction. The remaining proof term, for Preservation, is thus the most involved of the generated FPCC proofs. It is obtained in the following way:

Given a program P and a typing derivation for $\vdash P$, we know by FTAL progress that there exists a program P' such that $P \longmapsto P'$. Furthermore, by FTAL preservation, we know that $\vdash P'$. Now, the premise of our FPCC Preservation theorem provides us with a machine state S such that $P \Rightarrow S$, and we need to show that there exists another well-typed program that translates to $\mathsf{Step}\,(S)$. The semantics of FTAL has been set up so that this well-typed program is exactly P'. It remains now for us to prove that indeed $P' \Rightarrow \mathsf{Step}\,(S)$, as diagrammed in Figure 9.

Essentially, we need to show that the FTAL evaluation relation corresponds to the machine's step function. This is proved by induction on the typing derivation of $\vdash P$. For each possible case, we use inversion on the structure of P, the FTAL evaluation relation, the translation relation, and the machine Step function to gain the necessary information about the structure of P', S, and $\mathsf{Step}\,(S)$. Many of the cases of this proof are fairly straightforward.

Let us briefly consider one of the interesting cases of the Preservation proof, which is when the current instruction is alloc. Corresponding to the diagram in Figure 9, we have the following setup:

$$
\begin{aligned}
P &= (H, R, \mathsf{alloc}\ r_d[\tau_1, \ldots, \tau_n]; I) \\
P' &= (H', R', I) \\
S &= (M, \overline{R}, pc) \\
\mathsf{Step}\,(S) &= (M, \overline{R}', (pc + 1))
\end{aligned}
$$

where H', R', and $\overline{R}'$ can be determined by the operational semantics of FTAL and the definition of the Step function.

We now need to prove that P' is related to $\mathsf{Step}\,(S)$ by the translation. First, we know by the properties of the layout function that applying it to an extended heap maintains the mapping of all the existing labels in the old heap. Now, the FTAL heap is updated after evaluation but the memory stays the same after the step. However, since the update to the heap is only with uninitialized values which can be translated to any word, the translation will still hold on the unchanged memory. Thus, we can show that the updated heap translates to the unaltered memory. Then, relating the two updated register files is not difficult, nor is showing that the residual instruction sequence corresponds to the next program counter value. Well-formedness of P (*i.e.* $\vdash P$) is used in various steps of this proof, for instance, to reason that any labels in the registers are within the domain of the heap, hence the layout function on the updated heap, H', preserves the mappings of existing labels.

This completes the translation, or compilation, of a well-typed FTAL program to an FPCC code package. The FTAL program can be shown to correspond to an initial machine state and that state can be shown safe using the proofs of Preservation and Progress developed here.

6. Implementation

An implementation of the syntactic approach presented in this paper consists of an FTAL compiler which generates FPCC packages, made up of two parts: the initial machine state and the proof of safety. The proof of safety can be further divided into two pieces: one is the proof of the Preservation and Progress theorems and the other is the proof that the initial machine state satisfies the Initial Condition property. Note that the proofs of Preservation and Progress (which are built semi-automatically) do not change for any machine state which has been generated by compiling an FTAL program. Thus, these properties need only be proven once and can then be reused.

In order to generate the Initial Condition, we use a compiler that takes an FTAL program and compiles it to a machine state, producing the necessary proofs in the process. The structure of this compiler is fairly straightforward: After parsing an FTAL source file, type-checking is performed. The algorithm for type-checking follows closely the structure of the inductively defined static semantics in Coq. (Similarly, the compiler structures for FTAL abstract syntax mirror the Coq encoding.) Thus, the type-checker, as it analyzes the FTAL program, simultaneously builds a Coq term for the proof of well-formedness of the program.

Once type-checking is successfully completed, the compiler then translates the FTAL program into a machine state. Again, this is done in such a manner that a Coq term representing the machine state and the proof of the relation between the FTAL program and the machine state can be generated. Along with the typing derivation term produced above, we can now construct a proof that the global invariant holds on the initial machine state. This can be composed with the Preservation and Progress properties to produce a complete proof of the safety of the machine state as specified by our safety policy. More details on the Coq encoding of FTAL and its soundness proofs can be found in the companion technical report [9].

We thus have a complete system which starts with a typed assembly language program and compiles it into a FPCC package. Although our current implementation is not as realistic as [6, 4], the advantages of the syntactic FPCC approach are still clear. We compare the syntactic and semantic approaches to FPCC in detail in Section 7.

With respect to PCC implementations in general, the two most practical considerations are the extent of the trusted computing base (TCB) and the size of the proofs that are shipped with code. As for the former, the TCB of our syntactic FPCC implementation would consist of the following: (1) a parser, which converts the state of the raw machine into the encoding in the logic; (2) the encoding of the machine step function in the logic, which must accurately capture the semantics of the real machine (that is, it must be adequate); and (3) the proof-checker of the logic. The first two will necessarily exist in any PCC system. For syntactic FPCC, the proof-checker is smaller and more reliable than that of existing PCC systems because the logic used is much simpler. In addition, the VCgen is completely eliminated from the system.

Regarding the proofs that are shipped with syntactic FPCC packages, note that a large portion of the safety proof is static—the Progress and Preservation theorems hold regardless of the particular FTAL program from which the machine state was compiled. Hence, this part of the proof does not need to be re-supplied (or even re-checked) with every individual FPCC package. Furthermore, the remaining portion of the proof simply consists of the initial FTAL program and its typing derivation. The typing derivation can be easily and quickly generated by either the code producer or consumer. Thus, if proof size is especially critical, the only additional information that needs to be supplied with the initial machine state is the FTAL program itself.

7. Syntactic vs. Semantic FPCC

We have found that the choice between the syntactic and semantic approaches to generating FPCC involves some

trade-offs, which we briefly outline in this section.

In previous work on FPCC [4, 3], type judgments were assigned a meaning (a semantic truth value). In other words, each type of the typed assembly language is viewed as a predicate to be applied to memory, a value, and perhaps more arguments. The TAL typing rules then become lemmas to be proved in this semantic model. In contrast, the syntactic approach does not attempt to give any meaning to types or typing rules. The entire typing derivation of a TAL program is formalized and directly encoded in the logic. The FPCC safety proof is generated based on the similarly formalized soundness proof. Note, however, that unlike the original PCC systems, the typing rules are not part of the trusted base—they must be encoded and their soundness proved using only on the foundations of the logic.

The most obvious feature of the syntactic approach to FPCC is the resulting simplicity of the overall system. The complexities evident in [3, 5, 1, 2] do not arise in our system. For example, in order to support contravariant recursive types, an "indexed" semantic model is necessary, which complicates the definition of types and requires tedious reasoning about steps of computation. A more serious limitation of current semantic approaches to FPCC is the difficulty to model mutable record fields. This is a consequence of circularity in the definition of a "type" as a predicate on a state that is a pair of memory and a set of allocated addresses [3]. A third issue which has yet to be addressed by that model is supporting a type system with higher-order kinds. These, and various other difficulties in the semantic approach, result from attempting to give a meaning to types.

The reason why our approach does not suffer from the same complexity is that it only needs to give a meaning to types one step at a time. For example, in a semantic approach, when trying to show that two mutually-recursive functions f and g satisfy the predicates for their function types, we have the problem that the proof for f needs the proof for g and vice-versa. Resolving this circularity requires a coinduction principle or forces the use of an "indexed" semantic model. On the other hand, a syntactic approach will simply provide a typing rule for mutually-recursive functions. Of course, the soundness proof still needs to show that the typing rule is meaningful, but it only needs to do it one step at a time, in which case the circularity is gone: we do not need to assume anything about g in order to show that the first instruction of f can be executed safely. Only when we reach the call to g need we pay attention to it, but at that point we do not need to assume anything about f any more. Another way to look at it is that the "indexing" is done implicitly, for free, when we combine the progress and preservation lemmas to get the actual safety proof.

Despite the overall simplicity of the approach to FPCC given in this paper, it is not without potential technical intricacies. One of the most critical of these is the encoding of the syntactic typing rules and the soundness proof. In our prototype Coq "implementation" we have indeed been able to completely formalize and encode the static and operational semantics of FTAL, as well as prove the progress and preservation theorems. Although the encoding is not entirely trivial, it was achieved with reasonable effort. (In particular, the current implementation of the proofs of FTAL soundness and the FPCC Preservation and Progress theorems was completed within several months by a single graduate student with no previous experience in Coq or CiC.) The ability in CiC to perform eliminations on inductive definitions means that most proofs are quite straightforward and are proven using an intuitive sequence of steps. The fact that these proofs are generated interactively (*i.e.* manually) is not an issue because it only needs to be done once.

Finally, our approach relies on the availability of a typed assembly language that is similar to the machine for which proofs will be generated. It is also necessary that the type system capture all the invariants needed to prove soundness of the machine code. In this paper, since we took the interesting step of splitting the conventional malloc instruction of TAL into two separate instructions (alloc and bump), each of which is directly translated into a single machine instruction, we needed to refine the type system so that the information about the allocation state is correctly maintained in the invariant during translation. In general, whatever criteria is specified by the safety policy (*i.e.,* in the definition of $\mathsf{SP}(S)$) will need to be reflected in the type system.

8. Related Work and Conclusion

The original PCC system was designed by Necula and Lee [17, 15, 16], as discussed in our introduction. In addition to the general framework laid out in their work, implementation effort on building a certifying compiler has also been carried out [18, 6]. As also mentioned previously, however, these existing certifying compilers and clients are very language-specific and incorporate "built-in" understanding of a particular type-system into the logic.

Our source language, FTAL, is derived from the typed assembly language framework designed by Morrisett *et al.* [14]. Although, in contrast with PCC, typed assembly language does not deal with code at the lowest level of the machine, it is a critical tool which makes automatic generation of PCC proofs possible—following either the syntactic or the semantic approach.

Appel and Felty were the first to propose the notion of *foundational* PCC [4, 3]. Work on the semantic approach to FPCC has been carried out by Appel, Felty, and others [4, 5, 1, 12].

In a recent paper, Shao *et al.* [20] showed how to incorporate a logic such as CiC into a typed intermediate language. Together with the work described in this paper, we

can now build an end-to-end compiler that compiles high-level richly typed programs into FPCC.

Lastly, the syntactic approach to proving type soundness, an idea which we take advantage of in this paper, was introduced by Wright and Felleisen [24].

This paper presents an approach for producing foundational proof-carrying code based on syntactic soundness proofs. Starting with a type system for a typed assembly language, we formally encode its soundness proof and show a precise correspondence between TAL and the language of the actual machine. We use this (syntactic) correspondence, along with the proof that the type system enforces the invariants or constraints of the safety policy, to generate a package consisting of machine code and its proof of safety. By avoiding semantic modeling of types as in previous approaches, our framework for constructing foundational proofs is much simpler and more straightforward.

Acknowledgments

We are grateful to the members of the Coq mailing list who provided us with help and suggestions in the course of our encoding FTAL and the translation to FPCC, especially Yves Bertot, Pierre Casteran, Jean Goubault-Larrecq, Christine Paulin-Mohring, and Clement Renard. We also want to thank Peter Lee and anonymous referees for discussions and comments on an earlier version of this paper.

References

[1] A. J. Ahmed. Mutable fields in a semantic model of types. Talk presented at 2000 PCC Workshop, June 2000.

[2] A. J. Ahmed, A. W. Appel, and R. Virga. A stratified semantics of general references embeddable in higher-order logic (extended abstract). In *Proc. 17th IEEE Annual Symposium on Logic in Computer Science*, page (to appear), July 2002.

[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. Technical Report CS-TR-629-00, Princeton University, Dept. of Computer Science, Nov. 2000. To appear in TOPLAS.

[6] 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.

[7] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[8] A. Felty. Semantic models of types and machine instructions for proof-carrying code. Talk presented at 2000 PCC Workshop, June 2000.

[9] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof carrying-code. Technical Report YALEU/DCS/TR-1224, Dept. of Computer Science, Yale University, New Haven, CT, Jan. 2002.

[10] W. A. Howard. The formulae-as-types notion of constructions. In *To H.B.Curry: Essays on Computational Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[11] C. League, Z. Shao, and V. Trifonov. Precision in practice: A type-preserving Java compiler. Technical Report YALEU/DCS/TR-1223, Dept. of Computer Science, Yale University, New Haven, CT, Jan. 2002.

[12] N. Michael and A. Appel. Machine instruction syntax and semantics in higher order logic. In *Proc. 17th International Conference on Automated Deduction*, pages 7–24. Springer-Verlag, June 2000.

[13] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. In X. Leroy and A. Ohori, editors, *Proc. 1998 International Workshop on Types in Compilation: LNCS Vol 1473*, pages 28–52, Kyoto, Japan, March 1998. Springer-Verlag.

[14] 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.

[15] G. Necula. Proof-carrying code. In *Proc. 24th ACM Symp. on Principles of Prog. Lang.*, pages 106–119, New York, Jan. 1997. ACM Press.

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

[17] 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.

[18] G. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proc. 1998 ACM Conf. on Prog. Lang. Design and Impl.*, pages 333–344, New York, 1998.

[19] 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.

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

[21] K. N. Swadi and A. W. Appel. Typed machine language and its semantics. Preliminary version available at `www.cs.princeton.edu/~appel/papers/tml.pdf`, July 2001.

[22] The Coq Development Team. The Coq proof assistant reference manual. The Coq release v7.1, Oct. 2001.

[23] B. Werner. *Une Théorie des Constructions Inductives*. PhD thesis, A L'Université Paris 7, Paris, France, 1994.

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

[25] H. Xi and R. Harper. A dependently typed assembly language. In *Proc. 2001 ACM SIGPLAN Int'l Conf. on Functional Prog.*, pages 169–180. ACM Press, Sept. 2001.