Journal of Automated Reasoning **31:** 191–229, 2003.
© 2004 *Kluwer Academic Publishers. Printed in the Netherlands.*

A Syntactic Approach to Foundational Proof-Carrying Code ⋆

NADEEM A. HAMID, ZHONG SHAO, VALERY TRIFONOV,
STEFAN MONNIER, and ZHAOZHONG NI
Department of Computer Science, Yale University, New Haven, CT 06520-8285, U.S.A.
e-mail: {hamid,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; they certify safety but only if there is no bug in the typing rules. In foundational proof-carrying code (FPCC), on the other hand, proofs are constructed and verified by 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. Furthermore, none of them can be easily extended to support mutable fields and recursive types. In this article, 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 formalized syntactic soundness proof for the underlying type system. The former can be readily obtained from a type-checker, while the latter is known to be much easier to construct than the semantic soundness proofs. We give a translation from a typed assembly language into FPCC and demonstrate the advantages of our new system through an implementation in the Coq proof assistant.

Key words: foundational proof-carrying code, syntactic soundness proof, typed assembly language.

1. Introduction

Proof-carrying code (PCC), as pioneered by Necula and Lee [18, 16], 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 [17, 7] 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

⋆ An extended abstract of this paper appeared in the Seventeenth Annual IEEE Symposium on Logic in Computer Science (LICS'02), July 2002. 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.

proof checker. The VCgen is a fairly large program, so establishing its full correctness is a daunting task. The typing rules are also error-prone: League et al. [12] recently discovered a serious bug in the Special J typing rules that would undermine the integrity of the entire PCC-based system.

Foundational proof-carrying code (FPCC) [5, 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 [5, 9, 1, 6] all required constructing sophisticated semantic models to reason about types. For example, to support contravariant recursive types, Appel and Felty [9] 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" [6, page 2]. Appel and McAllester [6] later proposed an indexed model that significantly simplified the proofs but still involves tedious reasoning of computation steps with each type being defined as a complex set of indexed values. More serious, none of these approaches can be easily extended to support mutable fields. 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 [5, 6].

In this article, 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 consists simply of the typing derivation plus the formalized 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 easier to construct than the semantic soundness proof [26].⋆ Our article makes the following new contributions:

- Foundational proofs are widely perceived as extremely hard and tedious to construct, partly because existing efforts [5, 9, 1, 6, 2, 22] 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 a framework for foundational proofs is not necessarily an overly complex process. Our approach can support recursive types, mutable fields, and first-class code pointers in a straightforward way. With other recent results on certified binaries [21] and inductive definitions of quantified types [24], the syntactic approach offers a more scalable alternative for compiling high-level richly typed programs into FPCC.

⋆ That is, when done with "paper and pencil." We anticipate that a similar result will hold when it comes to formalizing the proofs.

- 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 [17, 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) [15, 14, 27] with their soundness proofs do not necessarily produce valid FPCC. To make the syntactic approach work, we need to ensure that a close correspondence can be established between the TAL and the underlying FPCC machine. This involves developing a type-system for TAL that not only is sound but also enforces the invariants needed for the FPCC safety proofs.
- The relationship between TAL [15] and PCC [18] 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 nontrivial TAL into FPCC. The translation is interesting because it not only shows the connection between the two but also gives new insights into the possibility of linking the expressive invariants in PCC to rich typing constructs in TAL.
- 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. Along with this, we have produced a fully formalized proof of soundness for the type system of the language.

In the rest of this artcile, 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. We thus have a prototype compilation framework, which starts with a typed assembly language program and compiles it into an FPCC package, consisting of an initial raw machine state and a proof of safety. In Sections 7–9, respectively, 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. Both the machine description and the proof must explicitly define, down to the foundations of mathematics, all required concepts and must prove any needed properties of these concepts.

2.1. THE LOGIC

To encode our safety policies and proofs, we use the calculus of inductive constructions (CiC) [23, 20]. CiC is an extension of the calculus of constructions (CC) [8], which is a higher-order typed lambda calculus. CC corresponds to higher-order predicate logic through the formulae-as-types principle (Curry–Howard correspondence [11]). 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 definition of natural numbers of kind schema Set with two constructors of the specified kinds.

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

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

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

The logic also provides elimination constructs for inductive definitions, which combine case analysis with a fix-point operation. Objects of an inductive type can thus be iterated over by 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; most important, the defined type can occur only positively in the arguments of its constructors. Mutually inductive types are also supported.

The calculus of inductive constructions has been shown to be strongly normalizing [25]; hence, the corresponding logic is consistent. It is supported by the Coq proof assistant [23], which we use to implement a prototype system of the results presented in this artcile.

In the remainder of this artcile, 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.

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, *Instr*. Informally, the instructions have the following effects:

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

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

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

Figure 1. Memory, registers, state, and instruction.

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

Figure 2. Machine semantics.

type *Word* $\rightarrow$ *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 and 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:

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

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

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

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 formulae-as-types correspondence, 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. [5, 6]. 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: $\Pi S{:}\mathit{State}.\,\mathit{Inv}\,(S) \rightarrow \mathit{Inv}\,(\mathsf{Step}\,(S))$

Progress: $\Pi S{:}\mathit{State}.\,\mathit{Inv}\,(S) \rightarrow \mathsf{SP}\,(S)$

These propositions intuitively state that our invariant holds for the initial state and for every subsequent state during the execution. The proposition 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 [26]: 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 syntactic approach to proving soundness of a type system involves proving progress (if $\vdash P$, then P is not stuck, i.e., there exists P' such that $P \longmapsto P'$) and preservation (if $\vdash P$ and $P \longmapsto P'$, then $\vdash P'$).

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 proofs of progress and preservation for the type system (encoded in the FPCC logic) 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 artcile, if the assembly language has "macro" instructions (e.g., `malloc` [15, 14] and `newarray` [27], 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. The reason is that *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, however, 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. [15]. The approach developed in this article can be applied to a TAL-like language extended with higher-order kinds and recursive types. For simplicity, we introduce here only a subset of such a language, which we call Featherweight Typed Assembly Language (FTAL). It does not include polymorphism or existential types, which can be easily added but would complicate the presentation. However, it does support recursive types, memory allocation, and mutable records (tuples).

The syntactic approach to FPCC as we present it here requires that for each machine state and each state transition, there be a corresponding FTAL program and transition. 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, as discussed previously. (See Section 4.6 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 (which we will call a *program* to distinguish from the machine state of Section 2.2) 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. We use $\{\}$ for an empty heap. The notation $H\{l \mapsto h\}$ rep-

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

Figure 3. Syntax of FTAL.

resents a heap that extends H with a label l mapped to h. Similar notation is used for heap types, register files, and register file types (except that for the latter two, we also use the same notation to indicate an update to the mapping). In the register file type (Γ) 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 ($?\tau$), which are used by the operational semantics to represent uninitialized tuple elements annotated with a type. The distinction between word values and small values in TAL is eliminated in FTAL by expanding the instruction set. Thus, for example, there are now two instructions for addition, one (add) taking a register and the other (addi) using an immediate value as the third operand.

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 explicitly refer only to the first 31 "user" registers (r). After an alloc instruction, a corresponding bump must be executed, to adjust the allocation pointer, before alloc can be used again. To statically enforce this, we give the allocation pointer register a special *allocation status type*, ρ, rather than a normal type. The possible types for this register, fresh and used(n), reflect the two states of allocation. 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 (φ), and recursive types. The initialization flags indicate whether there is valid data at each position of the tuple (when a tuple is first allocated, all the flags are 0). Other than fold and unfold, 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 annotated with a register file type (essentially specifying the preconditions on the data expected in the registers when the code block begins executing). Code blocks always end with a jmp or jd instruction.

Operations on recursive types in FTAL are supported by the fold and unfold instructions. Dynamically, these are no different from a simple mov. Statically, however, their purpose is to “cast” the type of a word, by either “rolling up” or “unrolling” the recursive type. (See the relevant rules of the static semantics in Section 4.3.)

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 adjustment 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.

$(H, R, I) \longmapsto P$ 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 $\vec{\tau} = \tau_1, \ldots, \tau_n$, $R(\text{r31}) = l$, 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 $H(l) = \text{code}[\Gamma].I''$
bump $i; I'$	$(H, R\{\text{r31} \mapsto \|H\|\}, I')$
fold $r_d[\tau], r_s; I'$	$(H, R\{r_d \mapsto \text{fold } R(r_s) \text{ as } \tau\}, I')$
jd l	(H, R, I') where $H(l) = \text{code}[\Gamma].I'$
jmp r	(H, R, I') where $H(R(r)) = \text{code}[\Gamma].I'$
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 $R(r_s) = \text{fold } v \text{ as } \tau$

Figure 4. Operational semantics of FTAL.

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, 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 to 8.

The top-level well-formedness rules are shown in Figure 8. In order 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 $I \subseteq I'$

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.

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 specify only a subset of the registers in its domain.

Subtyping is used for two purposes: to allow a code block to be called when the current register file type is more detailed than needed, and to be able to type-check the initialization of an uninitialized tuple element as described below.

To type-check heap allocation and the load and store operations, we follow TAL by introducing initialization flags in the type of tuples. When a tuple is newly allocated on the heap, all the elements are flagged with $\mathsf{0}$. A store operation will set the flag of the appropriate element to $\mathsf{1}$. Thus, a load operation is well-formed only if the flagged type of the element being accessed is set to $\mathsf{1}$. Because the type system only approximately tracks the initialization of tuple elements, we use subtyping to allow initialized tuple elements to be treated as if they were not initialized.

The special allocation register is typed using a new judgment of allocation status.

$$\frac{l = |\Psi|}{\Psi \vdash l : \mathsf{fresh}} \ (\text{FRESH})$$

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

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

$$\frac{FTV(\tau) = \emptyset}{\vdash \tau}\ (\text{TYPE}) \qquad \frac{\vdash \tau_i \qquad (1 \leq i \leq n)}{\vdash \{\mathbf{0}:\tau_0, \ldots, \mathbf{n}:\tau_n\}}\ (\text{HTYPE})$$

$$\frac{\vdash \tau_i \qquad (0 \leq i \leq n)}{\vdash \{r_0:\tau_0, \ldots, r_n:\tau_n, \mathsf{r31}:\rho\}}\ (\text{RFTYPE})$$

$$\frac{\vdash \tau}{\vdash \tau \leq \tau}\ (\text{REFLEX}) \qquad \frac{\vdash \tau_1 \leq \tau_2 \qquad \vdash \tau_2 \leq \tau_3}{\vdash \tau_1 \leq \tau_3}\ (\text{TRANS})$$

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

$$\frac{\vdash \tau_i \qquad (0 \leq i \leq m) \qquad (m \geq n)}{\vdash \{r_0:\tau_0, \ldots, r_m:\tau_m, \mathsf{r31}:\rho\} \subseteq \{r_0:\tau_0, \ldots, r_n:\tau_n, \mathsf{r31}:\rho\}}\ (\text{WEAKEN})$$

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

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

$$\frac{}{\Psi \vdash i:\mathsf{int}}\ (\text{INT}) \qquad \frac{\Psi \vdash v:\tau[\mu\alpha.\tau/\alpha]}{\Psi \vdash \mathsf{fold}\ v\ \mathsf{as}\ \mu\alpha.\tau : \mu\alpha.\tau}\ (\text{FOLD}) \qquad \frac{\vdash \Psi(l) \leq \tau}{\Psi \vdash l:\tau}\ (\text{LABEL})$$

$$\frac{l = |\Psi|}{\Psi \vdash l:\mathsf{fresh}}\ (\text{FRESH}) \qquad \frac{l = |\Psi| - \mathbf{1} \qquad \Psi \vdash l:\langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle}{\Psi \vdash l:\mathsf{used}(n)}\ (\text{USED})$$

$$\frac{\Psi \vdash v:\tau}{\Psi \vdash v:\tau^{\varphi}}\ (\text{INIT}) \qquad \frac{\vdash \tau}{\Psi \vdash ?\tau:\tau^{0}}\ (\text{UNINIT})$$

Figure 6. Well-formedness of FTAL types, heap, and word values.

In the first typing rule, a label whose value is equivalent to the size of the heap type must necessarily be unallocated, that is, fresh. 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:

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

$$\frac{\Psi; \Gamma\{\mathsf{r31}:\mathsf{fresh}\} \vdash I}{\Psi; \Gamma\{\mathsf{r31}:\mathsf{used}(n)\} \vdash \mathsf{bump}\ n;\ I}\ (\text{BUMP})$$

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

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

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

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

$$\frac{\Psi; \Gamma\{\mathsf{r31} : \mathsf{fresh}\} \vdash I}{\Psi; \Gamma\{\mathsf{r31} : \mathsf{used}(n)\} \vdash \mathsf{bump}\ n; I} \ (\textsc{bump})$$

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

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

$$\frac{\Psi; \Gamma\{r_d : \tau\} \vdash I \quad \vdash \Psi(l) \leq \tau}{\Psi; \Gamma \vdash \mathsf{movl}\ r_d, l; I} \ (\textsc{movl})$$

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

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

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

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

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

Figure 7. Well-formedness of FTAL instruction sequences.

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

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

$$\frac{\vdash \Psi \quad |\Psi| = |H| \quad \Psi \vdash H(l) : \Psi(l)\ \mathsf{hval} \quad (\mathbf{0} \leq l < |H|)}{\vdash H : \Psi} \ (\text{HEAP})$$

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

Figure 8. Well-formedness of FTAL programs, heaps, and register files.

For an alloc instruction to be well-typed, the allocation register, r31, must be in the fresh status; 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 used(n). No further allocation can take place until a bump instruction is encountered, which resets the status to fresh, corresponding again to the update in the operational semantics.

4.4. EXAMPLES

In this section, we give a few examples of FTAL programs to demonstrate that such a language (eventually extended with polymorphism and existentials) provides features that make it suitable for compiling high-level languages such as Java, ML, or Safe C.

Our first example is the calculation of a Fibonacci number in Figure 9. The C-like program at the top of the figure can be compiled to the FTAL code below it. The code segments fib, fib_loop and fib_return form a function, written in continuation passing style (CPS), which calculates the Fibonacci number with index given in r1, and then passes control to the continuation function given in r30. The main block calls fib to calculate F_{10} and passes the address of the halt block as its continuation. fib initializes the loop variables and then jumps into the loop code segment fib_loop, which jumps to fib_return when the calculation is done.

The second example, in Figure 10, demonstrates how to use recursive types and memory allocation to handle classes and objects. Class c has no data fields and only one method f, which takes an object of class c and invokes its method f. In the main program, an object of class c is created and its method f is called with the object itself as argument. The program will end up in an infinite recursive call to c.f. In FTAL, an object of class c is represented as a recursive tuple type whose only element is a code block with an only argument of the object type c. The code

```
void fib (n:int) {                    // "Safe C" code
  int a=1, b=1;
  for (int i=2; i++; i<=n)
    { int c = a + b; a = b; b = c; }
  return a;
}
int main () {
  return fib(10)
}

P = (H, {}, I)                             // FTAL code

H = fib: code[{ r1:int, r30:∀[{r1:int}] }].
      mov r3, r1;
      movi r1, 1;
      movi r2, 1;
      movi r4, 2;
      jd fib_loop
    fib_loop: code[{ r1:int, r2:int, r3:int, r4:int,
                     r30:∀[{r1:int}] }].
      bgt r4, r3, fib_return;
      add r5, r1, r2;
      mov r1, r2;
      mov r2, r5;
      addi r4, r4, 1;
      jd fib_loop
    fib_return: code[{ r1:int, r30:∀[{r1:int}] }].
      jmp r30
    halt: code[{r1:int}].
      jd halt
    main: code[{}].
      I

I =   movi r1, 10;
      movl r30, halt;
      jd fib
```

Figure 9. FTAL example: Fibonacci numbers.

block at label c_f uses the unfold and ld instructions to extract the argument object's own method f, and then jumps to it. The constructor for c, inlined in the main code block, uses the alloc and bump instructions to allocate heap space for a tuple, then initializes its method f with the label c_f, and folds the tuple into an object using the fold instruction. Similarly to c_f, the main code block then extracts method f from the newly created object and jumps to it.

```
class c {                               // "Safe C++" code
  void f (c x) { x.f(x) }
}
void main () {
  c x = new c;
  x.f(x)
}

P =  (H, {}, I)                             // FTAL code

c = μα.<∀[{r1:α}]>
H = c_f: code[{r1:c}].
       unfold r2, r1;
       ld r2, r2(0);
       jmp r2
     main: code[{}].
       I

I =    alloc r1 [∀[{r1:c}]];
       bump 1;
       movl r2, c_f;
       st r1(0), r2;
       fold r1[c], r1;
       unfold r2, r1;
       ld r2, r2(0);
       jmp r2
```

Figure 10. FTAL example: mini-object.

4.5. SOUNDNESS

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, 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. The most important of these lemmas are given below. Their encoding in Coq is described in Section 6.

LEMMA 1 (Register File Update). (1) *If* $\Psi \vdash R : \Gamma$ *and* $\Psi \vdash v : \tau$ *then* $\Psi \vdash R\{r \mapsto v\} : \Gamma\{r : \tau\}$.

(2) *If* $\Psi \vdash R : \Gamma$ *and* $\Psi \vdash l : \rho$ *then* $\Psi \vdash R\{\mathsf{r31} \mapsto l\} : \Gamma\{\mathsf{r31} : \rho\}$.

LEMMA 2 (Canonical Word Forms). *If* $\vdash H : \Psi$ *and* $\Psi \vdash v : \tau$, *then*

(1) *if* $\tau = \mathsf{int}$ *then* $v = i$;
(2) *if* $\tau = \forall[\Gamma]$ *then* $v = l$ *and* $H(l) = \mathsf{code}[\Gamma].I$;
(3) *if* $\tau = \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle$ *then* $v = l$;
(4) *if* $\tau = \mu\alpha.\tau'$ *then* $v = \mathsf{fold}\ v'\ \mathsf{as}\ \tau$.

LEMMA 3 (Canonical Register Word Forms). *If* $\Psi \vdash R : \Gamma$ *and* $\Gamma(r) = \tau$, *then*

(1) $R(r) = v$;
(2) *if* $\tau = \mathsf{int}$ *then* $R(r) = i$;
(3) *if* $\tau = \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle$ *then* $R(r) = l$.

LEMMA 4 (Canonical Heap Forms). *If* $\Psi \vdash h : \tau\ \mathsf{hval}$, *then*

(1) *if* $\tau = \forall[\Gamma]$ *then* $h = \mathsf{code}[\Gamma].I$ *and* $\Psi; \Gamma \vdash I$;
(2) *if* $\tau = \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle$ *then* $h = \langle v_1, \ldots, v_n \rangle$ *and* $\Psi \vdash v_i : \tau_i{}^{\varphi_i}$.

LEMMA 5 (Register File Weakening). *If* $\vdash \Gamma_1 \subseteq \Gamma_2$ *and* $\Psi \vdash R : \Gamma_1$, *then* $\Psi \vdash R : \Gamma_2$.

LEMMA 6 (Heap Extension). *If* $\vdash H : \Psi$, $l = |H|$ (*thus,* $l \notin Dom(H)$), *and* $\vdash \tau$, *then*

(1) $\vdash \Psi\{l : \tau\}$;
(2) *if* $\Psi \vdash v : \tau'$ *then* $\Psi\{l : \tau\} \vdash v : \tau'$;
(3) *if* $\Psi \vdash v : \tau^{\varphi}$ *then* $\Psi\{l : \tau\} \vdash v : \tau^{\varphi}$;
(4) *if* $\Psi; \Gamma \vdash I$ *then* $\Psi\{l : \tau\}; \Gamma \vdash I$;
(5) *if* $\Psi \vdash R : \Gamma\{\mathsf{r31} : \mathsf{fresh}\}$ *then* $\Psi\{l : \tau\} \vdash R : \Gamma\{\mathsf{r31} : \mathsf{used}(n)\}$;
(6) *if* $\Psi \vdash h : \tau'\ \mathsf{hval}$ *then* $\Psi\{l : \tau\} \vdash h : \tau'\ \mathsf{hval}$;
(7) *if* $\Psi\{l : \tau\} \vdash h : \tau\ \mathsf{hval}$ *then* $\vdash H\{l \mapsto h\} : \Psi\{l : \tau\}$.

LEMMA 7 (Heap Update). *If* $\vdash H : \Psi$ *and* $\vdash \tau \leq \Psi(l)$, *then*

(1) $\vdash \Psi\{l : \tau\}$;
(2) *if* $\Psi \vdash v : \tau'$ *then* $\Psi\{l : \tau\} \vdash v : \tau'$;
(3) *if* $\Psi \vdash v : \tau^{\varphi}$ *then* $\Psi\{l : \tau\} \vdash v : \tau^{\varphi}$;
(4) *if* $\Psi; \Gamma \vdash I$ *then* $\Psi\{l : \tau\}; \Gamma \vdash I$;
(5) *if* $\Psi \vdash R : \Gamma$ *then* $\Psi\{l : \tau\} \vdash R : \Gamma$;
(6) *if* $\Psi \vdash h : \tau'\ \mathsf{hval}$ *then* $\Psi\{l : \tau\} \vdash h : \tau'\ \mathsf{hval}$;
(7) *if* $\Psi\{l : \tau\} \vdash h : \tau\ \mathsf{hval}$ *then* $\vdash H\{l \mapsto h\} : \Psi\{l : \tau\}$.

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.

4.6. DESIGNING TAL FOR FPCC

For our presentation in this article, we have designed a novel FTAL language, that 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 that 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 the FTAL language, 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) that 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 that 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 8) 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 that 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. The layout must ensure that there are no overlaps between the images of tuples and code sequences in the memory. Our choice of the FTAL instruction set allows us to translate every FTAL instruction into one machine instruction word.

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 relations are as follows.

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 (either a tuple or a code block), 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* that, 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.) For our current purpose, we define

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

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 of the *Layout* function are useful later in proving Preservation and Progress.

The translation relations are defined by a set of inference rules, given in Figure 11. The rules are straightforward and operate purely on the syntax of FTAL

WORD VALUES

$$L \vdash l \Rightarrow_{\mathsf{w}} L(l) \qquad L \vdash i \Rightarrow_{\mathsf{w}} i \qquad \frac{\text{for any } w}{L \vdash ?\tau \Rightarrow_{\mathsf{w}} w} \qquad \frac{L \vdash v \Rightarrow_{\mathsf{w}} w}{L \vdash \mathsf{fold}\ v\ \mathsf{as}\ \tau \Rightarrow_{\mathsf{w}} w}$$

INSTRUCTIONS

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

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

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

HEAP, REGISTER FILE, PROGRAM

$$\frac{L \vdash H(l) \Rightarrow_{\mathsf{h}} M[L(l)..L(l{+}1){-}1] \quad \mathbf{0} \le l < |H|}{L \vdash H \Rightarrow M} \qquad \frac{L \vdash R(\hat{r}_i) \Rightarrow_{\mathsf{w}} \overline{R}(\overline{r}_i) \quad 0 \le i \le 31}{L \vdash R \Rightarrow \overline{R}}$$

$$\frac{\begin{array}{ll} \mathit{Layout}(H) \vdash H \Rightarrow M & \mathit{Layout}(H) \vdash I \Rightarrow_{\mathsf{s}} M[pc..pc + |I| - 1], \\ \mathit{Layout}(H) \vdash R \Rightarrow \overline{R} & \text{where } \exists l \in \mathrm{dom}\, H.(H(l) = \mathsf{code}\ [\Gamma].I',\ I \subseteq I', \text{ and} \\ & \qquad pc = \mathit{Layout}(H)(l) + |I'| - |I|) \end{array}}{(H, R, I) \Rightarrow (M, \overline{R}, pc)}$$

Figure 11. Relating FTAL programs to machine states.

programs. Note that FTAL type annotations are discarded in the translation (for example, in the fold instruction) and the label word values are mapped to memory words by 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 ($\hat{r}$ is defined in Figure 3 and $\overline{r}$ in Figure 1). 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. Since in a well-typed FTAL program the current instruction sequence must also be present in the heap, we can always translate it to a known program counter. 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 11 is 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 that 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: $Inv\,(S_0)$

Preservation: $\Pi S : State.\, Inv\,(S) \rightarrow Inv\,(\mathsf{Step}\,(S))$

Progress: $\Pi S : State.\, Inv\,(S) \rightarrow \mathsf{SP}\,(S)$

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 the progress and preservation lemmas 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 also in terms of their typing derivations.

$$Inv(S) = \exists P : program.\ ((\vdash P)\ \wedge\ (P \Rightarrow S))$$

Hence, the invariant holds on a state if there exists an FTAL program that is well-typed and translates to the state.

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 that 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 12.

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.

$$\begin{tikzcd}
\vdash P \arrow[rr, Rightarrow, "(\textit{translate})"] \arrow[d, "(\textit{evaluate})"] & & S \arrow[d, "(\mathsf{Step})"] \\
\vdash P' \arrow[rr, Rightarrow, dashed, "\textit{translate}\ ?"] & & \mathsf{Step}\,(S)
\end{tikzcd}$$

Figure 12. Relationship between FTAL evaluation and machine semantics.

* "Inversion" is simply a process of backwards reasoning – given a premise, one infers what judgment rule must have been used to prove it and then adds the assumptions of that rule to one's list of available facts. For example, if we know an FTAL program is well-typed, then by inversion the only possible rule that would allow one to prove such a judgment is (PROG) on page 205; hence, we can infer, for instance, that there must also exist a proof that the heap component of the program is well-typed.

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 12, 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 (Figure 2).

We now need to prove that P' is related to 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 that 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 (as described in Sections 2.3 and 3) using the proofs of Preservation and Progress developed here.

6. Implementation

An implementation of the syntactic approach presented in this article consists of an FTAL compiler that generates FPCC packages. An FPCC package consists 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 that has been generated by compiling an FTAL program. Thus, these properties need be proven only once and can then be reused for all FPCC packages produced by this compiler.

In the following sections, we first describe our Coq representation of the machine and the encoding of FTAL syntax and semantics and soundness theorems. Next we discuss implementation of the formal proofs of FPCC Preservation and

Progress, which were done interactively by using the Coq proof assistant. Then, we describe a compiler that parses an FTAL program, performs type-checking, and automatically produces the Coq term representing the typing derivation. This typing derivation is then used to construct the proof of the Initial Condition property.

Coq is a proof assistant tool for the calculus of inductive constructions (Section 2.1). It provides an interactive interface for constructing formal proofs in the logic. The Coq syntax for λ-abstraction, $\lambda X : A. B$, is `[X:A]B`. The syntax for dependent products, $\Pi X : A. B$, is `(X:A)B` and Coq allows for the normal arrow abbreviation of this when the bound variable does not occur in the body, e.g., `A->B`. Coq syntax for inductive definitions is exactly that described in Section 2.1. Coq uses the sort `Prop` for logical propositions and the sort `Set` for the type of specifications (Booleans, natural numbers, lists, programs, etc.).

6.1. ENCODING MACHINE SEMANTICS

The Coq encoding of the machine to which FTAL programs are translated is very similar to the presentation in Section 2.2. For example, having defined the registers as an inductive set with 32 constructors, we then define the memory and register file as being functions and the state as a triple of memory, register file, and program counter.

```
Definition Word := nat.
Inductive _Reg : Set := _r0 : _Reg | _r1 : _Reg | ...
Definition Mem  := Word -> Word.
Definition _RegFile := _Reg -> Word.
Definition State := (Mem * (_RegFile * Word)).
```

The instruction set is then defined as an inductive definition with appropriate constructors.

```
Inductive _Instr : Set
 := _add  : _Reg -> _Reg -> _Reg -> _Instr
  | _addi : _Reg -> _Reg -> Word -> _Instr
  | _movi : _Reg -> Word -> _Instr
  | _bgt  : _Reg -> _Reg -> Word -> _Instr
  | _jd   : Word -> _Instr
  | _jmp  : _Reg -> _Instr
  | _ld   : _Reg -> _Reg -> Word -> _Instr
  | _st   : _Reg -> Word -> _Reg -> _Instr
  | _ill  : _Instr.
```

We next decide on how to encode the instructions above as natural numbers and write a Coq function that uses the appropriate arithmetic operations to decode a natural number into an `_Instr`.

```
   Definition Dc : Word -> _Instr := ...
```

We are now ready to encode the semantics of the machine as given in Section 2.2. For updating the register file and memory, we define auxiliary functions, as in the code below (`beq_reg` compares equality of two register names and returns a Boolean).

```
Definition updateregfile
 : _RegFile -> _Reg -> Word -> _RegFile
 := [R:_RegFile; rd:_Reg; v:Word]
    ([r:_Reg] if (beq_reg r rd) then v else (R r)).

Definition Step : State -> State
 := [St:State] Cases St of (M, (R, pc)) =>
    Cases (Dc (M pc)) of
        (_add rd rs rs')
           => (M, ((updateregfile R rd
                            (plus (R rs) (R rs'))),
               (S pc)))
      | (_jd l)
           => (M, (R, l))
      | ...
      | _ill => St
    end
  end.
```

Finally, we can state the safety policy we wish to enforce and define what a safe machine state is. The `MultiStep` function simply applies the `Step` function to the given state `n` times.

```
Definition SP [S:State]
 := (let (M,T')=S in
               (let (R,PC)=T' in
                  ~(Dc (M PC))=_ill)).

Definition Safe [S:State]
 := (n:nat)(SP (MultiStep n S)).
```

6.2. ENCODING FTAL SYNTAX

Encoding the FTAL language is a more involved process. We start by defining each syntactic category as an inductive type. For example, the FTAL types are encoded as follows.

```
Definition initflag := bool.

Inductive Omega : Set
 := intty : Omega
  | codety : (Map Reg Omega) -> APTy -> Omega
  | tupty  : (list Omega) -> (list initflag) -> Omega
  | recty  : (OmegaL (S O)) -> Omega.
```

The `list` in the tuple type constructor is the usual definition of a list, found in the Coq library. Hence, the tuple type constructor takes as arguments a list of types and a list of initialization flags, encoded as Booleans. (Alternatively, we could have used a list of pairs instead of two lists, but in practice this encoding seemed easier to work with.) `Map` is defined as a list of pairs. The type of a register file (used by `codety`) is a map from registers (definition presented below) to types. We also define a "well-formed `Map`," used later, as being a list of pairs in which the first element of every pair in the list is distinct from all others.

A well-formed type in the FTAL language will never have free type variables, but variables may appear in a recursive type. Hence, we represent the type under the recursive type constructor by a "lifted" version of `Omega` that uses deBruijn indices to represent variables. The parameter of the `OmegaL` type below tracks the number of free type variables in the term to ensure the correctness of our substitution and unfolding functions for recursive types.

```
Inductive OmegaL : nat -> Set
 := inttyL  : (OmegaL O)
 | codetyL : (i:nat) (Map Reg (OmegaL i)) ->
                     APTy -> (OmegaL i)
 | tuptyL  : (i:nat) (list (OmegaL i)) ->
                     (list initflag) -> (OmegaL i)
 | rectyL  : (i:nat) (OmegaL (S i)) -> (OmegaL i)
 | varL    : (i:nat) (OmegaL (S i))
 | liftL   : (i:nat) (OmegaL i) -> (OmegaL (S i)).
```

Registers are defined as in the machine above. Unlike the presentation in the preceding sections, we carry the special allocation pointer separately from the rest of of the register file; hence, there are only 31 registers defined for FTAL. The r31 register, or `AP` below, is simply a label (which is defined to be a natural number). The special allocation pointer types are encoded as an inductive definition, and the types of register files and heaps are maps from registers or labels, respectively, to `Omega` (the heap type also requires that the map be well-formed, as defined above).

```
Inductive Reg : Set := r0 : Reg | r1 : Reg | ...
Definition label := nat.
Definition AP := label. (* alloc. ptr. (r31) *)
Inductive APTy : Set
 := fresh : APTy
 | used  : nat -> APTy.
Definition RegFileTy := (Map Reg Omega).
Definition HeapTy := (WFMap label Omega).
```

The remainder of the definitions for FTAL syntax are fairly intuitive and match closely the presentation in Figure 3, except that r31 and its type are carried separately as `AP` and `APTy`.

```
Inductive Instr : Set
 := add    : Reg -> Reg -> Reg -> Instr
  | addi   : Reg -> Reg -> int -> Instr
  | alloc  : Reg -> (list Omega) -> Instr
  | bgt    : Reg -> Reg -> label -> Instr
  | bump   : int -> Instr
  | fold   : Reg -> Omega -> Reg -> Instr
  | ld     : Reg -> Reg -> int -> Instr
  | mov    : Reg -> Reg -> Instr
  | movi   : Reg -> int -> Instr
  | movl   : Reg -> label -> Instr
  | st     : Reg -> int -> Reg -> Instr
  | unfold : Reg -> Reg -> Instr.

Inductive InstrSeq : Set
 := iseq : Instr -> InstrSeq -> InstrSeq
  | jd   : label -> InstrSeq
  | jmp  : Reg -> InstrSeq.

Inductive WordVal : Set
 := wl      : label -> WordVal
  | wi      : int -> WordVal
  | wuninit : Omega -> WordVal
  | wfold   : WordVal -> Omega -> WordVal.

Inductive HeapVal : Set
 := tuple : (list WordVal) -> HeapVal
  | code  : RegFileTy -> APTy -> InstrSeq -> HeapVal.

Definition Heap    := (WFMap label HeapVal).
Definition RegFile := (Map Reg WordVal).

Definition Program := (Heap * (RegFile * (AP * InstrSeq))).
```

6.3. ENCODING FTAL SEMANTICS AND SOUNDNESS

Each judgment form of the dynamic and static semantics can be viewed as a relation and is also encoded as an inductive definition. For every evaluation or typing rule, there is an associated constructor of the appropriate inductive definition. (This allows us to use Coq's inductive elimination constructs to perform inversion and induction on typing derivations.) We show the encoding of several evaluation rules in Figure 13.

The `reglookup` and `regupdext` are to be read as propositions stating that looking up the value of a given register in a register file (which is defined to be a `Map`) yields the given word value and that updating or extending the mapping of a register in a register file results in a new register file, respectively. For the heap (and

```
Inductive Eval : Program -> Program -> Prop
:= ev_add
   : (H:Heap; R,R':RegFile; r31:AP; I':InstrSeq)
     (rd,rs,rs':Reg; rsval,rsval':int)
     (reglookup R rs (wi rsval)) ->
     (reglookup R rs' (wi rsval')) ->
     (regupdext R rd (wi (plus rsval rsval')) R') ->
     (Eval (H,(R,(r31,(iseq (add rd rs rs') I'))))
          (H,(R',(r31,I'))))
|  ev_alloc
   : (H,H':Heap; R,R':RegFile; r31:AP; I':InstrSeq)
     (rd:Reg; V:(list Omega))
     (regupdext R rd (wl r31) R') ->
     (hextend H r31 (tuple (makeUninitTup V)) H') ->
     (Eval (H, (R, (r31, (iseq (alloc rd V) I'))))
          (H', (R', (r31, I'))))
|  ev_bump
   : (H:Heap; R:RegFile; r31:AP; I':InstrSeq)
     (i:int; l:nat)
     (hsize H l) ->
     (Eval (H, (R, (r31, (iseq (bump i) I'))))
          (H, (R, (l, I'))))
|  ev_jd
   : (H:Heap; R:RegFile; r31:AP)
     (l:label; G:RegFileTy; T:APTy; I':InstrSeq)
     (hlookup H l (code G T I')) ->
     (Eval (H, (R, (r31, (jd l))))
          (H, (R, (r31, I'))))
|  ev_movl
   : (H:Heap; R,R':RegFile; r31:AP; I':InstrSeq)
     (rd:Reg; l:label)
     (regupdext R rd (wl l) R') ->
     (Eval (H, (R, (r31, (iseq (movl rd l) I'))))
          (H, (R',(r31, I'))))
|  ev_store
   : (H,H':Heap; R:RegFile; r31:AP; I':InstrSeq)
     (rd,rs:Reg; i:int; l:label;
      V,V':(list WordVal); w:WordVal)
     (reglookup R rd (wl l)) ->
     (reglookup R rs w) ->
     (hlookup H l (tuple V)) ->
     (updatetuple V i w V') ->
     (hupdate H l (tuple V') H') ->
     (Eval (H, (R, (r31, (iseq (st rd i rs) I'))))
          (H',(R, (r31, I'))))
|  ...
```

Figure 13. Coq encoding of FTAL dynamic semantics.

similarly heap type, which like heap is defined as a well-formed `Map`) the `hextend` proposition requires that the label being added to the domain of the heap is not already being mapped in the heap. The `hupdate` proposition holds true only when the label is in fact present in the heap mapping. These propositions are defined inductively as relations on `Map`s.

The encodings of the main static judgments are given in Figures 14 and 15.

To formally prove the soundness of FTAL as encoded above, we proceed by first proving the same lemmas that are listed in Section 4.5. The statements of these lemmas in Coq, while slightly verbose, are essentially the same as those listed in that section. We generate the proofs of these lemmas interactively using Coq proof "tactics." The tactics of the proof assistant correspond much to the steps that would be used in a hand proof, for example, induction, inversion, rewriting, and application of rules (constructors). We present the statements of a few of these lemmas in Coq below (Register File Update, the second case of Canonical Word Forms, and several cases of the Heap Extension lemma).

```
Lemma regfile_update
 : (HT:HeapTy; R,R':RegFile; G,G':(Map Reg Omega))
   (rd:Reg; v:WordVal; t:Omega)
   (WFRegFile HT R G) ->
   (WFWordVal HT v t) ->
   (regupdext R rd v R') ->
   (regupdext G rd t G') ->
   (WFRegFile HT R' G').

Lemma can_word_forms_code
 : (H:Heap; HT:HeapTy; v:WordVal; G:RegFileTy; T:APTy)
   (WFHeap H HT) ->
   (WFWordVal HT v (codety G T)) ->
   (EX l | v=(wl l) /\ (EX I | (hlookup H l (code G T I)))).

Lemma heap_ext_2
 : (H,H':Heap; HT,HT':HeapTy; t:Omega; l:label)
   (v:WordVal; t':Omega)
   (WFHeap H HT) ->
   (hsize H l) ->
   (htextend HT l t HT') ->
   (WFWordVal HT  v t') ->
   (WFWordVal HT' v t').

Lemma heap_ext_4
 : (I:InstrSeq)
   (H,H':Heap; HT,HT':HeapTy; t:Omega; l:label)
   (R:RegFileTy; A:APTy)
   (WFHeap H HT) ->
   (hsize H l) ->
   (htextend HT l t HT') ->
   (WFInstrSeq HT  R A I) ->
   (WFInstrSeq HT' R A I).

Lemma heap_ext_7
 : (H,H':Heap; HT,HT':HeapTy; t:Omega; l:label)
   (h:HeapVal)
   (WFHeap H HT) ->
   (hsize H l) ->
```

```
Inductive RegFileSubtype            (* register file subtyping: G <= G' *)
  : RegFileTy -> RegFileTy -> Prop
  := weaken
     : (G,G':RegFileTy)
       ((r:Reg; t:Omega) (reglookup G' r t) -> (reglookup G r t)) ->
       (RegFileSubtype G G').

Inductive WFWordVal         (* well-formed word values: HT |- w : t wval *)
  : HeapTy -> WordVal -> Omega -> Prop
  := int_wval : (HT:HeapTy; i:int)(WFWordVal HT (wi i) intty)
  |  label_wval
     : (HT:HeapTy; l:label; t,t':Omega)
       (htlookup HT l t') ->
       (Subtype t' t) ->
       (WFWordVal HT (wl l) t)
  |  fold_word_wval
     : (HT:HeapTy; w:WordVal;  t:OmegaR; t':Omega)
       (RUnlift (RUnfold t))=t' ->
       (WFWordVal HT w t') ->
       (WFWordVal HT (wfold w (recty t)) (recty t)).

Inductive WFInstrSeq    (* well-formed instruction sequences: HT; G |- I *)
 : HeapTy -> RegFileTy -> APTy -> InstrSeq -> Prop
 := s_add
    : (HT:HeapTy; G,G':RegFileTy; T:APTy; I:InstrSeq)
      (rd,rs,rs':Reg)
      (reglookup G rs intty) ->
      (reglookup G rs' intty) ->
      (regupdext G rd intty G') ->
      (WFInstrSeq HT G' T I) ->
      (WFInstrSeq HT G T (iseq (add rd rs rs') I))
 |  s_alloc
    : (HT:HeapTy; G,G':RegFileTy; I:InstrSeq)
      (rd:Reg; n:nat; V:(list Omega))
      n=(length V) ->
      (regupdext G rd (tupty V (makeUninitTupty V)) G')->
      (WFInstrSeq HT G' (used n) I) ->
      (WFInstrSeq HT G fresh (iseq (alloc rd V) I))
  |  s_jd
     : (HT:HeapTy; G,G':RegFileTy; T:APTy)
       (l:label)
       (htlookup HT l (codety G' T)) ->
       (RegFileSubtype G G') ->
       (WFInstrSeq HT G T (jd l))
 |  s_st
    : (HT:HeapTy; G,G':RegFileTy; T:APTy; I:InstrSeq)
      (rd,rs:Reg; i:int;
      V,V':(list initflag); Ts:(list Omega); t:Omega)
      (reglookup G rd (tupty Ts V)) ->
      (reglookup G rs t) ->
      (ListNth ? Ts i t) ->
      (updatetupty V i V') ->
      (regupdate G rd (tupty Ts V') G') ->
      (WFInstrSeq HT G' T I) ->
      (WFInstrSeq HT G T (iseq (st rd i rs) I))
 |  ...
```

Figure 14. Coq encoding of FTAL static semantics: main definitions (1 of 2).

```
Inductive WFHeapVal         (* well-formed heap values: HT |- h : t hval *)
 : HeapTy -> HeapVal -> Omega -> Prop
 := tuple_wf
    : (HT:HeapTy; wl:(list WordVal); tl:(list Omega); il:(list initflag))
      (WFWordValinitList HT wl tl il) ->
      (WFHeapVal HT (tuple wl) (tupty tl il))
 |  code_wf
    : (HT:HeapTy; G:RegFileTy; I:InstrSeq; T:APTy)
      (WFInstrSeq HT G T I) ->
      (WFHeapVal HT (code G T I) (codety G T)).

Inductive WFHeap                                   (* well-formed heap *)
 : Heap -> HeapTy -> Prop
 := heap_wf
    : (H:Heap; HT:HeapTy)
      (EX s | (hsize H s) /\
              (htsize HT s) /\
              ((n:label; h:HeapVal) (hlookup H n h) -> (lt n s)) /\
              ((n:label; t:Omega) (htlookup HT n t) -> (lt n s)) /\
              ((n:label) (lt n s) -> (EX h | (hlookup H n h))) /\
              ((n:label) (lt n s) -> (EX t | (htlookup HT n t))) /\
              ((n:label; h:HeapVal; t:Omega)
                (hlookup H n h)->(htlookup HT n t)->(WFHeapVal HT h t)) /\
              (OrdHeap H)
      ) ->
      (WFHeap H HT).

Inductive WFRegFile                       (* well-formed register file *)
  : HeapTy -> RegFile -> RegFileTy -> Prop
  := regfile_wf
     : (HT:HeapTy; R:RegFile; G:RegFileTy)
       ((r:Reg; t:Omega)
          (reglookup G r t) ->
             (EX w | (reglookup R r w) /\ (WFWordVal HT w t))) ->
       ((r:Reg; v:WordVal; l:label; n:nat)
          (reglookup R r v) ->
          (stripWV v)=(wl l) ->
          (htsize HT n) ->
          (lt l n)) ->
       (WFRegFile HT R G).

Inductive WFProgram                             (* well-formed program *)
 : Program -> Prop
 := program_wf
    : (H:Heap; HT:HeapTy; R:RegFile; G:RegFileTy;
       l:AP; t:APTy; I:InstrSeq)
      (WFHeap H HT) ->
      (WFRegFile HT R G) ->
      (WFap HT l t) ->
      (WFInstrSeq HT G t I) ->
      (EX l | (EX G' | (EX T' | (EX I' | (EX n |
         (hlookup H l (code G' T' I')) /\
         (ISubDepth I I' n)))))) ->
      (WFProgram (H, (R, (l, I)))).
```

Figure 15. Coq encoding of FTAL static semantics: main definitions (2 of 2).

```
(htextend HT l t HT') ->
(hextend H l h H') ->
(WFHeapVal HT' h t) ->
(WFHeap H' HT').
```

The main theorems for the soundness of FTAL, preservation and progress, follow from the various lemmas:

```
   Theorem ftal_preserv
 : (P,P':Program) (WFProgram P) -> (Eval P P') -> (WFProgram P').

Theorem ftal_progress
    : (P:Program) (WFProgram P) -> (EX P' | (Eval P P')).
```

We have now completely formalized the syntactic soundness proof of FTAL. In the next section, we discuss the encoding of the translation relations between FTAL and the machine, and how FTAL soundness is used to produce the proofs of the FPCC Preservation and Progress theorems.

6.4. ENCODING FPCC PRESERVATION AND PROGRESS

The translation relations (not shown here) are represented as a set of inductive definitions that follow precisely the presentation in Figure 5.1, for example,

```
Inductive TrProgram
  : Program -> State -> Prop := ....
```

The global invariant for FPCC can be defined in terms of the translation between a well-formed FTAL program and the machine state:

```
Definition Inv [S:State]
  := (EX P:Program | (WFProgram P) /\ (TrProgram P S)).
```

Now we proceed to prove the FPCC Progress theorem:

```
Theorem Progress : (S:State) (Inv S) -> (SP S).
```

As mentioned in Section 5.3, the Progress theorem is straightforward. Using several Coq “Inversion” tactics, we determine that there exists a well-formed instruction sequence that translates to the program counter of the state. Then we perform case analysis on the well-formed instruction sequence judgment and show that in every possible case, the program counter of the state must be pointing to a non-`illegal` instruction.

Next is the FPCC Preservation theorem, which is more involved to prove but which follows the discussion in Section 5.3.

```
Theorem Preservation : (S:State) (Inv S) -> (Inv (Step S)).
```

With these two theorems, we can now prove that a machine state will be safe if the FPCC Initial Condition property is satisfied.

```
Theorem Safety : (S:State) (Inv S) -> (Safe S).
```

6.5. GENERATING THE INITIAL CONDITION

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 representing the proof of well-formedness of the program. In particular, if `P:Program`, then the type-checking phase produces a term, `D:(WFProgram P)`.

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. That is, for some `S:State`, a term, `T:(TrProgram P S)`, is constructed. Along with the typing derivation term of `P` produced above, we can now construct a proof that the global invariant holds on `S`. This can then be composed with the `Safety` theorem of the preceding section to produce a complete proof of the safety of the machine state `S`, as specified by our safety policy.

6.6. THE COMPLETE SYSTEM

We now have a complete system that starts with a typed assembly language program and compiles it into an FPCC package, consisting of an initial machine state and a proof of safety. Although our current implementation is not so realistic as [7, 5], 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 (and FPCC in general), 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.

⋆ That is, taking into consideration the type-related axioms that need to be added to the base logic of those systems. Furthermore, although our prototype uses the Coq proof assistant, which integrates the proving and checking processes, it would not be at all difficult to separate the proof-checker out into a very small, simple program, as has been done in previous FPCC approaches.

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 resupplied (or even rechecked) 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 the 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 [5, 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 (of the real machine), 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 (or FTAL) program is formalized and directly encoded in the logic. The FPCC safety proof is generated based on the similarly formalized soundness proof for TAL. Unlike the original PCC systems, however, the typing rules are not part of the trusted base of our system – they must be encoded and their soundness proved by using only the foundations of the logic.

The difference discussed in the preceding paragraph is clearly exhibited in the nature of the invariants that are generated for the two approaches. In the semantic approach, the global invariant used to prove safety, although it may be derived from the type system of a typed assembly language, actually states properties directly about the machine state – the contents of registers, memory addresses, and so on. This approach is very general, and the invariant can be used to express arbitrary properties about the machine. On the other hand, the invariant used for the syntactic approach does not prove properties directly about the machine state. Instead, it simply requires that there exist a correspondence between the machine state and an assembly program. The safety and soundness of the assembly language is used to ensure safety of the machine code.

The most obvious feature of the syntactic approach to FPCC is the resulting simplicity of the overall system. The complexities evident in [3, 6, 1, 2] do not arise in our system. For example, in order to support contravariant recursive types, an “indexed” semantic model is necessary [6], which complicates the definition of types. 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 [2, 3]. These, and various other difficulties in the semantic approach, result from attempting to give a meaning to types.

The reason our approach does not suffer from the same complexity is that it needs to give a meaning to types only 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 needs to do it only 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 article, 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 by using an intuitive sequence of steps. The fact that these proofs are generated interactively (i.e., manually) is not an issue because it needs to be done only once. Hence, in terms of implementation effort, a syntactic approach does not seem to be any harder, at the worst, than a semantic one.

Moreover, 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 article, 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 thus 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 are specified by the safety policy (i.e., in the definition of $\mathsf{SP}(S)$) will need to be reflected in the type system. For example, supporting fine-grained access control could be achieved by using a more refined type system such as that in [10]. Designing a type system that can be used with the syntactic approach to FPCC to prove complex safety invariants is future work.

8. Related Work

The original PCC system was designed by Necula and Lee [18, 16, 17], 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 [19, 7]. As also mentioned, 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. [15]. 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 that 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 [5, 3]. Work on the semantic approach to FPCC has been carried out by Appel, Felty, and others [5, 6, 1, 13, 4].

In a recent paper, Shao et al. [21] showed how to incorporate a logic such as CiC into a typed intermediate language. Together with the work described in this article, we can now build an end-to-end compiler that compiles high-level richly typed programs into FPCC.

We note that the syntactic approach to proving type soundness, an idea that we take advantage of in this article, was introduced by Wright and Felleisen [26].

9. Conclusion

This article 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, extending and scaling our compilation framework to construct foundational proofs for realistic languages are likely to be easier than previously published approaches.

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 thank Peter Lee and anonymous referees for discussions and comments on earlier versions of this article.

References

1. Ahmed, A. J.: Mutable fields in a semantic model of types, Talk presented at 2000 PCC Workshop, June 2000.
2. Ahmed, A. J., Appel, A. W. and Virga, R.: A stratified semantics of general references embeddable in higher-order logic, in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, June 2002, pp. 75–86.
3. Appel, A. W.: Foundational proof-carrying code, in *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, June 2001, pp. 247–258.
4. Appel, A. W. and Felten, E. W.: Models for security policies in proof-carrying code, Technical Report CS-TR-636-01, Department of Computer Science, Princeton University, Mar. 2001.
5. Appel, A. W. and Felty, A. P.: A semantic model of types and machine instructions for proof-carrying code, in *Proceedings 27th ACM Symposium on Principles of Programming Languages*, ACM Press, 2000, pp. 243–253.
6. Appel, A. W. and McAllester, D.: An indexed model of recursive types for foundational proof-carrying code, *ACM Trans. on Programming Languages and Systems* **23**(5) (Sept. 2001), 657–683.
7. Colby, C., Lee, P., Necula, G., Blau, F., Plesko, M. and Cline, K.: A certifying compiler for Java, in *Proceedings 2000 ACM Conference on Programming Language Design and Implementation*, ACM Press, New York, 2000, pp. 95–107.
8. Coquand, T. and Huet, G.: The calculus of constructions, *Inform. and Comput.* **76** (1988), 95–120.
9. Felty, A.: Semantic models of types and machine instructions for proof-carrying code, Talk presented at 2000 PCC Workshop, June 2000.
10. Grossman, D., Morrisett, G. and Zdancewic, S.: Syntactic type abstraction, *ACM Trans. on Programming Languages and Systems* **22**(6) (Nov. 2000), 1037–1080.
11. Howard, W. A.: The formulae-as-types notion of constructions, in *To H. B. Curry: Essays on Computational Logic, Lambda Calculus and Formalism*, Academic Press, 1980.
12. League, C., Shao, Z. and Trifonov, V.: Precision in practice: A type-preserving Java compiler, in *Proceedings 12th International Conference on Compiler Construction*, Lecture Notes in Comput. Sci. 2622, Springer-Verlag, Heidelberg, 2003, pp. 106–120.
13. Michael, N. and Appel, A.: Machine instruction syntax and semantics in higher order logic, in *Proceedings 17th International Conference on Automated Deduction*, Springer-Verlag, June 2000, pp. 7–24.
14. Morrisett, G., Crary, K., Glew, N. and Walker, D.: Stack-based typed assembly language, in X. Leroy and A. Ohori (eds), *Proceedings 1998 International Workshop on Types in Compilation*, Kyoto, Japan, Lecture Notes in Comput. Sci. 1473, Springer-Verlag, March 1998, pp. 28–52.
15. Morrisett, G., Walker, D., Crary, K. and Glew, N.: From System F to typed assembly language, in *Proceedings 25th ACM Symposium on Principles of Programming Languages*, ACM Press, Jan. 1998, pp. 85–97.
16. Necula, G.: Proof-carrying code, in *Proceedings 24th ACM Symposium on Principles of Programming Languages*, ACM Press, New York, Jan. 1997, pp. 106–119.
17. Necula, G.: Compiling with proofs, PhD thesis, School of Computer Science, Carnegie Mellon University, Sept. 1998.
18. Necula, G. and Lee, P.: Safe kernel extensions without run-time checking, in *Proceedings 2nd USENIX Symp. on Operating System Design and Impl.*, 1996, pp. 229–243.

19. Necula, G. and Lee, P.: The design and implementation of a certifying compiler, in *Proceedings 1998 ACM Conference on Programming Language Design and Implementation*, New York, 1998, pp. 333–344.
20. Paulin-Mohring, C.: Inductive definitions in the system Coq – rules and properties, in M. Bezem and J. Groote (eds), *Proceedings TLCA*, Lecture Notes in Comput. Sci. 664, Springer-Verlag, 1993.
21. Shao, Z., Saha, B., Trifonov, V. and Papaspyrou, N.: A type system for certified binaries, in *Proceedings 29th ACM Symposium on Principles of Programming Languages*, ACM Press, Jan. 2002, pp. 217–232.
22. Swadi, K. N. and Appel, A. W.: Typed machine language and its semantics, Preliminary version available at `www.cs.princeton.edu/~appel/papers/tml.pdf`, July 2001.
23. The Coq Development Team: The Coq proof assistant reference manual. The Coq release v7.1, Oct. 2001.
24. Trifonov, V., Saha, B. and Shao, Z.: Fully reflexive intensional type analysis, in *Proceedings 2000 ACM International Conference on Functional Programming*, ACM Press, Sept. 2000, pp. 82–93.
25. Werner, B.: Une théorie des constructions inductives, PhD thesis, L'Université Paris 7, Paris, France, 1994.
26. Wright, A. K. and Felleisen, M.: A syntactic approach to type soundness, *Inform. and Comput.* **115**(1) (1994), 38–94.
27. Xi, H. and Harper, R.: A dependently typed assembly language, in *Proceedings 2001 ACM International Conference on Functional Programming*, ACM Press, Sept. 2001, pp. 169–180.