Foundational Proof Checkers with Small Witnesses*

Dinghao Wu Andrew W. Appel Aaron Stump

Princeton University
{dinghao,appel}@cs.princeton.edu

Washington University in St. Louis
stump@cs.wustl.edu

ABSTRACT

Proof checkers for proof-carrying code (and similar systems) can suffer from two problems: huge proof witnesses and untrustworthy proof rules. No previous design has addressed both of these problems simultaneously. We show the theory, design, and implementation of a proof-checker that permits small proof witnesses and machine-checkable proofs of the soundness of the system.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*correctness proofs, formal methods*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*logics of programs, mechanical verification*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*denotational semantics*

General Terms

Languages, Verification

Keywords

Proof Checker, Proof-Carrying Code

1. INTRODUCTION

In a proof-carrying code (PCC) system [10], or in other proof-carrying applications [3], an untrusted prover must convince a trusted checker of the validity of a theorem by sending a proof. Two of the potential problems with this approach are that the proofs might be too large, and that the checker might not be trustworthy. Each of these problems has been solved separately; in this paper we show how to solve them simultaneously.

*This material is based upon work supported by the National Science Foundation under Grant No. 0208601.

To appear in *Fifth ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2003).*

The general approach is to write a logic program that has a machine-checked semantic correctness proof; this technique can be used in other domains (besides "proof-carrying") to write logic programs with machine-checked guarantees of correctness.

1.1 Small proof witnesses

Necula has a series of results on reducing proof size [11, 12]. He represents logics, theorems, and proofs in the notation of the Edinburgh Logical Framework (LF) [8]. But the natural representation of an LF proof contains redundancy (common subexpressions) that can cause exponential blowup if the proofs are written in the usual textual representation. Necula's LF_i data structure [11] eliminated most of this redundancy, leading to reasonable-sized proof terms.

In the PCC framework, given a machine-language program, the proof is of a theorem that the program obeys some safety property. It's natural to compare the size of the representation of the proof witness to the size of the binary machine-language program. Necula's LF_i proof witnesses were about 4 times as big as the programs whose properties they proved.

Pfenning's Elf and Twelf systems [14, 15] are implementations of the Edinburgh Logical Framework. In these systems, proof-search engines can be represented as logic programs, much like (dependently typed, higher-order) Prolog. Elf and Twelf build proof witnesses automatically; if each logic-program clause is viewed as an inference rule, then the proof witness is a trace of the successful goals and subgoals executed by the logic program. That is, the proof witness is a tree whose nodes are the names of clauses and whose subtrees correspond to the subgoals of these clauses.

Necula's theorem provers were written in this style, originally in Elf and later in a logic-programming engine that he built himself. In later work, he moved the prover clauses into the trusted checker. In principle, proof witnesses for such a system can be just a single bit, meaning, "A proof exists: search and ye shall find it." However, to guarantee that proof-search time (in the trusted checker) would be small, Necula invented *oracle-based checking* [12]: the untrusted prover would record a sequence of bits that recorded which subgoals failed (and therefore, where backtracking was required). This bitstream serves as an "oracle" that the trusted checker can use to avoid backtracking. The oracle bitstream need not be trusted; if it is wrong, then the trusted checker will choose the wrong clauses to satisfy subgoals, and will fail to find a proof.

Using oracle-based checking, the proof witness (the oracle bitstream) is about 1/8 the size of the machine code.[1] The key idea is to run a simple Prolog engine in the trusted proof checker; the oracle is just an optimization to ensure that the checker doesn't run for too long.

1.2 Trustworthy checkers

Necula's oracle-based checker for PCC comprises approximately 26,000 lines of code:

23,000	Verification-condition generator, written in C
1,400	LF proof checker, written in C
800	Oracle-based Prolog interpreter, in C
700	Axioms for type system, written in LF
26,000	Total trusted lines of code

The largest component is the verification condition generator (VC-Gen), which traverses the machine-language program and extracts a formula in logic, the *verification condition*, which is true only if the program obeys a given safety policy.

This 26,000 lines forms the trusted code base (TCB) of the system: any bug in the TCB may cause an unsafe program to be accepted. The large VC-Gen component is a concern, but so are the axioms of the type system: if the type system is not sound, then unsafe programs will be accepted. League *et al.* [9] have shown that one of the SpecialJ typing rules is unsound.

The goal of our research [2] is to check proofs of program safety using a much smaller TCB. We do this by eliminating the VC-Gen component—we reason directly about machine code in higher-order logic, instead of the two-step process of extracting the verification condition and then proving it; and we write the rules of our type system as machine-checkable lemmas, instead of axioms. We have shown that the TCB for a proof-carrying code system can be reduced below 2700 lines, as follows [6]:

803	LF proof checker, written in C
135	Axioms & definitions of higher-order logic, in LF
160	Axioms & definitions for arithmetic, in LF
460	Specification of Sparc instruction encodings, in LF
1,005	Specification of Sparc instruction semantics, in LF
105	Specification of safety predicate, in LF
2,668	Total trusted lines of code

Unfortunately, in this prototype system the proof witnesses are huge: the DAG representation of a safety proof of a program might be 1000 times as large as the program. Proof size is approximately linear in the size of the program,[2] so this factor of 1000 will not grow substantially worse for larger programs. However, while this early prototype is useful in showing how small the TCB can be made, it is impractical for real applications because the proof witnesses are too big.

1.3 Synthesis

In this paper we will show that Necula's insight (run a resource-limited Prolog engine in the trusted checker) can be combined with our paranoia (don't trust the logic programming rules used by such a Prolog engine) to make a PCC checker with small witnesses and a small trusted base.

Our approach is as follows. We write a type-checking algorithm in a subset of Prolog with no backtracking, a very limited form of unification, and with efficiently indexed dynamic atomic clauses. We show that the operators of such a Prolog program can be given a semantics in higher-order logic, such that the soundness of each clause can be proved as a machine-checkable lemma. We show that this Prolog subset is adequate for writing efficient type-checkers for PCC and for other "proof-carrying" applications.

Our trusted checker is sent the Prolog clauses, with machine checkable soundness proofs; it checks these proofs before installing the clauses. Then it is sent a theorem to check (i.e., in a PCC application, the safety of a particular machine-language program) and a small proof witness. The Prolog program traverses the theorem and proof witness; this traversal succeeds only if the theorem is valid.

The TCB of our new checker is only 366 lines larger than our previous prototype. It mainly includes all the components of our previous system (2668 lines) plus a concise implementation of an interpreter (282 lines of C code) for our Prolog subset.

2. SEMANTIC PROOFS OF HORN CLAUSES

We will illustrate our approach using an example—a type checker for a very simple programming language. In this example we illustrate the following points, which are common to many proof-carrying applications:

- The specification of the theorem to be proved is quite simple *(in this case, that the program evaluates to an even number).*
- The proof technique involves the definition of a carefully designed set of predicates that allow a simple, syntax-directed decision procedure *(in this case, we define a syntax-directed type system for* evenness *and* oddness*).*
- The syntax-directed rules are provable, from the definitions of the operators, as machine-checkable lemmas in the underlying higher-order logic (this is what *foundational* means: the rules are provable from the foundations of logic).
- The syntax-directed rules require management of a symbol table, or *context,* that would lead to a quadratic algorithm if implemented naively; we want a linear-time prover, and we'll show how to make one.
- The language being typechecked in a proof-carrying code system (or in proof-carrying authentication) is the output of another program—the compiler (or a

[1]Unfortunately, this statistic is somewhat misleading. A "pure" PCC system would transmit two components from an untrusted code producer to a code consumer: a machine-language program and a proof witness. The SpecialJ proof-carrying Java system on which Necula measured oracle-based checking transmits three components: The machine code, the proof, and a Java "class file". The Java class file, as is usual in any Java system, contains descriptions of the types of all procedures (methods) in the program, including formal parameter and result types. However, the "1/8 size" figure does not include the Java class files.

[2]Technically, proof size is roughly proportional to the size of the program multiplied by the average number of live variables on entry to a basic block; this is superlinear but much less than quadratic, for typical programs.

$$
\begin{array}{llll}
\textit{type} & \tau & ::= & \mathsf{even} \mid \mathsf{odd} \\
\textit{decl} & d & ::= & \cdot \mid \mathsf{let}\ x = e; d \\
\textit{expr} & e & ::= & x \mid n \mid e_1 + e_2 \\
\textit{prog} & p & ::= & (d; e)
\end{array}
$$

Figure 1: Syntax of even-odd system.

$$
\begin{array}{lll}
\mathit{Var} & \equiv & \mathit{Num} \\
\mathit{State} & \equiv & \mathit{Var} \to \mathit{Num} \to \mathit{Form} \\
\mathit{Decl} & \equiv & \mathit{State} \to \mathit{Form} \\
\mathit{Exp} & \equiv & \mathit{State} \to \mathit{Num} \to \mathit{Form} \\
\mathit{Program} & \equiv & \langle \mathit{Decl}, \mathit{Exp} \rangle \\
 & & \\
(d; e) & \equiv & \langle d, e \rangle \\
\cdot & \equiv & \lambda s.\ \mathrm{true} \\
\mathsf{let}\ x = e; d & \equiv & \lambda s.\ d\ s \wedge (\forall a. e\ s\ a \Rightarrow s\ x\ a) \\
x & \equiv & \lambda s. \lambda a.\ s\ x\ a \\
n & \equiv & \lambda s. \lambda a.\ a = n \\
e_1 + e_2 & \equiv & \lambda s. \lambda a.\ \exists a_1. \exists a_2. \\
 & & e_1\ s\ a_1\ \wedge\ e_2\ s\ a_2\ \wedge \\
 & & a = a_1\ \mathit{plus}\ a_2 \\
 & & \\
\mathit{safe} & \equiv & \lambda p.\ \forall s.\ \mathit{fst}(p)\ s \Rightarrow \\
 & & \exists a.\ \mathit{snd}(p)\ s\ a \wedge \mathrm{isEven}(a)
\end{array}
$$

Figure 2: Safety specification.

prover). Such languages don't need all of the syntactic sugar that human-readable languages have, and processing them is therefore easier.

2.1 Example: even-valued expressions

Consider a simple calculus for expressions with constants, variables, addition, and let-binding, as shown in Figure 1.

A program consists of a list of declarations and an expression. An expression is either a variable, a natural number, or the sum of two expressions. Here is an example:

$$\mathsf{let}\ x = 4\ ;\ \mathsf{let}\ y = x + 8\ ;\ x + y$$

There are two declarations followed by an expression; the program evaluates to 16.

2.2 Safety specification

In this simple example, we define that a "safe" program is one that evaluates to an even number. In order to define the safety theorem, we need to know what a program means and how to evaluate a program. The safety predicate, along with a conventional denotational semantics of the language in consideration, is shown in Figure 2.

All of these definitions are treated as axiomatic by our checker; that is, they are *trusted*. Variables are represented as numbers. An abstract *State* maps a variable to its content, i.e. a number. A program is a pair of a declaration and an expression; its semantics is the pair of semantics of the corresponding declaration and expression. Declaration *Decl* is a predicate on states. Expression *Exp* is a predicate on a state and a number; that is, given a state the expression evaluates to a number. The semantics of concrete expressions is straightforward from definitions.

Finally, the safety theorem is based on the semantics of language constructs.[3] Given a program p, it is "safe" if: for any states s, if the declaration of the program, i.e. $\mathit{fst}(p)$, holds on s, then there exists a number a such that the expression of the program, i.e. $\mathit{snd}(p)$, evaluates to a and a is even.

$$
\frac{\vdash_p p : \mathsf{even}}{\mathit{safe}(p)}\ \mathit{SafeTy}
\qquad
\frac{\cdot \vdash_d (d; e) : \tau}{\vdash_p (d; e) : \tau}\ \mathit{ProgTy}
$$

$$
\frac{\Gamma \vdash_e e_1 : \tau_1 \qquad \Gamma[x : \tau_1] \vdash_d (d; e) : \tau}{\Gamma \vdash_d (\mathsf{let}\ x = e_1; d; e) : \tau}\ \mathit{DeclConsTy}
$$

$$
\frac{\Gamma \vdash_e e : \tau}{\Gamma \vdash_d (\cdot; e) : \tau}\ \mathit{DeclNilTy}
\qquad
\frac{\Gamma(x) = \tau}{\Gamma \vdash_e x : \tau}\ \mathit{VarTy}
$$

$$
\frac{\Gamma \vdash_e e_1 : \tau_1 \qquad \Gamma \vdash_e e_2 : \tau_2 \qquad \tau_1 \boxplus \tau_2 = \tau}{\Gamma \vdash_e e_1 + e_2 : \tau}\ \mathit{PlusTy}
$$

$$
\frac{}{\mathsf{even} \boxplus \mathsf{even} = \mathsf{even}}\ \boxplus ee
\qquad
\frac{}{\mathsf{odd} \boxplus \mathsf{odd} = \mathsf{even}}\ \boxplus oo
$$

$$
\frac{}{\mathsf{even} \boxplus \mathsf{odd} = \mathsf{odd}}\ \boxplus eo
\qquad
\frac{}{\mathsf{odd} \boxplus \mathsf{even} = \mathsf{odd}}\ \boxplus oe
$$

Figure 3: Typing rules with static context.

2.3 Type checker

The typing rules appear in Figure 3. There are three kinds of typing judgements. The judgement for a program $\vdash_p$ checks that the program evaluates to a number whose type is τ. The declaration judgement $\vdash_d$ states that, assuming the environment built so far, and assuming the remaining declarations hold, the expression has a certain type. The expression judgement $\vdash_e$ asserts that an expression has certain type under typing context Γ.

These typing rules can be read as a Prolog-like logic program. Each rule is a clause of the logic program. The conclusion of a rule is the head of the clause, and each premise of the rule is a subgoal. The typing rules are designed such that the conclusions of these typing rules are disjoint. Therefore, when running the type checker (as a logic program) there is no need to backtrack; we say that such a type system is *syntax-directed.*

Furthermore, if we give denotational semantics expressed in higher-order logic to typing judgements such as $\vdash_p$, $\vdash_d$, and $\vdash_e$, each typing rule can be proved as a lemma in the system, thus its soundness is guaranteed with respect to the foundations of logic. The denotational semantics of typing judgements is given in Figure 4. Proofs of the typing rules are quite straightforward and thus omitted here. The denotational semantics of the type operators are part of the safety *proof,* not part of the safety specification. That is, they are *not* trusted. It is straightforward to prove the safety theorem from the conclusion of type checking rule *ProgTy* if we pass τ even when invoking the type checker, as shown in the *SafeTy* rule.

Our checker will determine the validity of the safety predicate by determining whether a proof exists. It will not

[3]For our PCC application, there are only two language constructs for the machine code to be proved safe. The machine code is a sequence of integers encoding machine instructions; so we only need *cons* and *nil.*

$$
\begin{array}{lcl}
Ty & \equiv & Num \to Form \\
Env & \equiv & State \to Form \\
\textsf{even} & \equiv & \lambda x.\exists n.\ \text{isInt}(n) \wedge x = 2n \\
\textsf{odd} & \equiv & \lambda x.\exists n.\ \text{isInt}(n) \wedge x = 2n+1 \\
\\
\vdash_p p : \tau & \equiv & \forall s.\ fst(p)\ s \Rightarrow \exists a.\ snd(p)\ s\ a \wedge \tau\ a \\
\Gamma \vdash_d (d;e) : \tau & \equiv & \forall s.\ (d\ s \wedge \Gamma\ s) \Rightarrow \exists a.\ (e\ s\ a \wedge \tau\ a) \\
\Gamma \vdash_e e : \tau & \equiv & \forall s.\ \Gamma\ s \Rightarrow \exists a.\ (e\ s\ a \wedge \tau\ a) \\
\tau_1 \boxplus \tau_2 = \tau & \equiv & \forall n_1.\forall n_2.\ \tau_1\ n_1 \Rightarrow \tau_2\ n_2 \Rightarrow \tau\ (n_1 + n_2) \\
\Gamma[x:\tau] & \equiv & \lambda s.\ \Gamma\ s \wedge \exists a.\ s\ x\ a \wedge \tau\ a \\
\Gamma(x) = \tau & \equiv & \forall s.\ \Gamma\ s \Rightarrow \exists a.\ s\ x\ a \wedge \tau\ a
\end{array}
$$

Figure 4: Definitions of types and judgements.

construct such a proof as a data structure: instead, it will traverse a trace of such a proof, composing lemmas in a syntax-directed way. We call our set of lemmas a *type system*: our machine-checked safety proof of a program P consists of (1) a proof of soundness for the type system, and (2) the successful syntax-directed execution of the typing clauses as applied to P.

Efficiency and proof size problem. When type checking a program, we build a type environment, or *context*, from the declarations for variables that appear in the expression. The rules for traversing a list of declarations and building the corresponding type contexts are *DeclConsTy* and *DeclNilTy*. When a variable is encountered, we look up its type in the context. However, the typing rule *VarTy* does not specify a context lookup algorithm. Consider the following variable type-lookup rules.

$$\frac{}{\Gamma[x:\tau] \vdash x:\tau}\ VarTyHit$$

$$\frac{\Gamma \vdash x:\tau \qquad x \neq y}{\Gamma[y:\tau'] \vdash x:\tau}\ VarTyMiss$$

Suppose the context is simply organized as a list in these two rules; each element of the list is a pair: a variable and its type. Then each context lookup takes linear time, and type-checking a whole program will take quadratic time. Correspondingly, the size of the generated proof for a lookup operation is linear with respect to the size of the context, and thus the safety proof for a program has a quadratic blowup. In the next section, we give a more efficient algorithm that still has a provably sound semantic model, and generates concise proofs.

3. EFFECTIVE CONTEXT MANAGEMENT

As we have explained, we avoid sending large proofs to the trusted checker by sending a proof scheme with a soundness proof for the proof scheme. We want the proof scheme to "execute" efficiently, that is, in linear time with respect to the size of the program-safety-theorem being proved. And we want the proof schemes to be written in the "smallest possible" Prolog-like language: what set of language features are useful?

Here we will show an efficient proof scheme for contexts; because this scheme requires dynamic clauses in the Prolog subset, we have included a limited form of dynamic clauses in our language design.

$$\frac{\vdash_p p : \textsf{even}}{safe(p)}\ SafeTy \qquad \frac{d \vdash_d (d;e) : \tau}{\vdash_p (d;e) : \tau}\ ProgTy$$

$$\frac{\Gamma \vdash_e e_1 : \tau_1 \qquad bind(x,\tau_1,\Gamma)\ \to\ \Gamma \vdash_d (d;e) : \tau}{\Gamma \vdash_d (\textsf{let}\ x = e_1; d; e) : \tau}\ BindTy$$

$$\frac{\Gamma \vdash_e e : \tau}{\Gamma \vdash_d (\cdot;e) : \tau}\ BindNil \qquad \frac{bind(x,\tau_1,\Gamma)}{\Gamma \vdash_e x : \tau}\ VarTy$$

$$\frac{\Gamma \vdash_e e_1 : \tau_1 \qquad \Gamma \vdash_e e_2 : \tau_2 \qquad \tau_1 \boxplus \tau_2 = \tau}{\Gamma \vdash_e e_1 + e_2 : \tau}\ PlusTy$$

$$\frac{}{\textsf{even} \boxplus \textsf{even} = \textsf{even}}\ \boxplus ee \qquad \frac{}{\textsf{odd} \boxplus \textsf{odd} = \textsf{even}}\ \boxplus oo$$

$$\frac{}{\textsf{even} \boxplus \textsf{odd} = \textsf{odd}}\ \boxplus eo \qquad \frac{}{\textsf{odd} \boxplus \textsf{even} = \textsf{odd}}\ \boxplus oe$$

Figure 5: Typing rules with dynamic context.

3.1 Dynamic clauses and local assumptions

Many logic programming systems provide a facility for managing dynamic clauses at run time. In Prolog, users can *assert* a fact or clause into database or *retract* a clause dynamically. The assert/retract mechanism can be expensive if the dynamic clause in consideration is not atomic (i.e., has subgoals) because the dynamic clause has to be compiled and integrated into the program's decision trees. If the dynamic clause is atomic, with input-mode arguments that are integers or hashable, the assert/retract operation can be cheap: Prolog systems usually provide efficient support for asserting or retracting an atomic clause by using hash tables. That is, asserting, retracting, and querying indexable atomic clauses can be done in constant time per operation.

In the Logical Framework (LF) [8], or its implementation Twelf [13, 15], one can use local assumptions [16] to check dynamic clauses into database. Since these assumptions are local, their static scopes control their lifetimes; there is no need to provide an explicit retract mechanism. A clause of the form $\{x:\tau\}\ A\ x \to B\ x$ introduces a local assumption $A\ x$ into the context and then solves the goal $B\ x$ under this assumption.[4] When proof search on goal B has finished, assumption A is automatically retracted. That is, Twelf uses a dynamically well-scoped version of assert/retract. One can use Prolog assert/retract mechanism to simulate Twelf's local assumptions, however. We can give semantics to local assumptions and generate concise proofs so that clauses are guaranteed to be correct.

Local assumptions are particularly effective—efficient, secure (with a provably sound model), and concise—when we need to deal with big environments and generate proofs of lookups in these environments.

3.2 Typing rules

In this subsection, we present an efficient type checking algorithm using dynamic clauses. We give semantics in the next subsection. Figure 5 shows typing rules with a dynamic context management scheme.

The rule *ProgTy* calls a declaration checking rule and

[4] It is a dependent type on local parameter x.

passes declaration d to it. The declaration d appears twice in the premise. The declaration checking rules traverse one d, and the other d is used to pass the original declaration all the way to the expression checking rules.

The rule *BindTy* requires some explanation. It first checks that the expression e_1 has type τ_1, then asserts this fact as a dynamic clause (or local assumption) $\mathit{bind}(x, \tau_1, \Gamma)$ and continues type checking.

When type checking a variable expression, we try rule *VarTy* to match the previous checked-in local assumptions. The lookup operation takes constant time and the proof generated for it is concise. The $\boxplus$ rules remain the same as before.

In a conventional Prolog implementation that supports efficient assert/retract operations for atomic dynamic clauses like $\mathit{bind}(x, \tau_1, \Gamma)$, the type checking algorithm above is linear. Moreover, it is provably sound as shown in the next subsection.

3.3 Foundational semantics and proofs

The safety specification remains the same as presented in Figure 2. The definitions of types and typing judgements remain untouched except for $\vdash_d$ and the new constructor *bind*.

$$\begin{array}{lcl} \Gamma \vdash_d (d; e) : \tau & \equiv & \forall s.\ (\Gamma \sqsubseteq d \wedge \Gamma\ s) \Rightarrow \\ & & \quad \exists a.\ (e\ s\ a \wedge \tau\ a) \\ \mathit{bind}(x, \tau, \Gamma) & \equiv & \forall s.\ \Gamma\ s \Rightarrow \\ & & \quad \exists a.\ (s\ x\ a \wedge \tau\ a) \\ d_1 \sqsubseteq d_2 & \equiv & \forall s.\ d_1\ s \Rightarrow d_2\ s \end{array}$$

The semantics of dynamic clause $\mathit{bind}(x, \tau, \Gamma)$ is very similar to that of the static binding operator $\Gamma[x : \tau]$ and lookup operator $\Gamma(x) = \tau$. It serves both purposes. From these definitions it is straightforward to prove the typing rules as lemmas and the safety theorem can be proved from the successful type checking of a program from the goal $\vdash_p (d; e) : \mathsf{even}$. Here we give the proof for rule *BindTy*.

LEMMA 1 (*BindTy*).

$$\frac{\Gamma \vdash_e e_1 : \tau_1 \qquad \mathit{bind}(x, \tau_1, \Gamma)\ \rightarrow\ \Gamma \vdash_d (d; e) : \tau}{\Gamma \vdash_d (\mathsf{let}\ x = e_1; d; e) : \tau}$$

Proof. By definition of $\vdash_d$, for all state s, we assume $\Gamma \sqsubseteq (\mathsf{let}\, x = e_1; d)$ and $\Gamma\ s$, then we prove $\exists a.\ (e\ s\ a \wedge \tau\ a)$. This can be obtained from $\Gamma \vdash_d (d; e) : \tau$. In order to use this fact, we need to prove the local assumption $\mathit{bind}(x, \tau_1, \Gamma)$, which can be proved from the premise $\Gamma \vdash_e e_1 : \tau_1$ and the assumption $\Gamma \sqsubseteq (\mathsf{let}\, x = e_1; d)$. □

The machine checkable proof for this rule can be found in Appendix A.

4. FLIT

For developing our semantic proofs of soundness we use Twelf, a sophisticated system with many useful features: in addition to an LF type checker, it contains a type reconstruction algorithm that permits users to omit many explicit parameters, a proof-search algorithm (which is like a higher-order Prolog interpreter), constraint regimes (e.g., linear programming over the exact rational numbers), mode analysis of parameters, a meta-theorem prover, a pretty-printer, a module system, a configuration system, an interactive Emacs mode, and more. We have found many of these features useful in proof development, but Twelf is certainly not a minimal proof checker, we would like to avoid the need to trust it. However, since Twelf does construct explicit proof objects internally, we can extract these objects to send to our minimal checker.

The previous section shows that efficient syntax-directed type-checking uses certain logic-programming constructs (dynamic clauses) but not others (backtracking), and that each Horn clause can be proved sound as a lemma in higher-order logic. This section describes a suitable logic programming interpreter implemented in Flit, our trusted LF proof checker. Other aspects of Flit are described in another paper [6]. To achieve a concise and efficient implementation, we impose several restrictions on the form of goals and programs. If these are violated, the interpreter will remain sound but may fail to be complete. This section discusses these restrictions and the implementation of the interpreter. We begin with a few basic definitions from logic programming.

4.1 Basic definitions

Flit's logic programming interpreter can solve goals with respect to logic programs containing dynamic clauses and simple arithmetic. Goals are atomic formulas (also called *atoms*) of first-order logic. Logic programs are conjunctions of clauses, where each clause is either a universally quantified atom or a universally quantified implicational formula formed from atoms using only conjunctions and implications. Implications other than the topmost one must be in the antecedents of other implications; such nested implications give rise to dynamic clauses. As usual, the *head* of a clause is the clause itself for atomic clauses and the consequent of the topmost implication for implicational clauses. The *body* of a clause is its antecedent if it is an implicational clause, and *TRUE* if it is an atomic clause. We assume the usual notion of unification of atoms, based on the usual notion of substitution for a finite set of variables. A substitution is *ground* if all terms in its range are ground. *Goals* and *subgoals* are just atoms. A *solution* for a goal G with respect to a logic program $\mathcal{P}$ is a substitution σ such that $(\mathcal{A} \cup \mathcal{P}) \vdash \sigma(G)$, where $\vdash$ is provability in first-order logic and $\mathcal{A}$ are axioms for addition, multiplication, and truncating division on 32-bit natural numbers. These arithmetic operations are represented as three-place relations, where the last place gives the output of the operation.[5]

For reasoning about our logic programming interpreter, we will make use of a standard natural deduction proof system for first-order logic. Then a *use* of a clause C in a proof is either an assumption of C, if C is atomic; or an application of modus ponens whose implicational argument is C. The subgoals produced while solving a goal G are just the atomic formulas contained in the proof of G from $\mathcal{A} \cup \mathcal{P}$.

4.2 Flit's logic programming language

Flit's logic programming interpreter is sound for arbitrary logic programs with dynamic clauses. It is complete for solving goals G with respect to logic programs $\mathcal{P}$, under certain

[5]Since the 32-bit natural numbers are not closed under addition or multiplication, goals such as `multiply` 2^{20} 2^{20} X are not satisfiable. However, our underlying higher-order logic has the complete theory of the integers; the 32-bit numbers are just a way of writing some of the constants and evaluating some of the expressions in that theory.

conditions. First, all dynamic clauses of $\mathcal{P}$ must be atomic. Second, suppose G is provable from $\mathcal{P}$. Then it must be the case that G has a proof satisfying the following conditions:

1. Bounded execution. The size of the proof is no more than some fixed constant *MAX_PROOF*. This assumption is used to avoid dynamic allocation of memory while trying to solve G. Although our logic program never constructs the proof itself, the number of Horn-clause matches it executes is proportional to the size of the proof; and since we have not implemented garbage collection, unreclaimed auxiliary memory used in constructing substitutions may grow proportionally to proof size.

2. Determinism. Every subgoal S in the proof is proved by a use of the first clause of $\mathcal{D} \cup \mathcal{P}$ whose head unifies with S, where $\mathcal{D}$ are the active assumptions (corresponding to active dynamic clauses). Under this condition of determinism, the interpreter need not backtrack to be complete. It is never necessary to try later clauses if a proof cannot be found from a use of the first unifying clause.

3. Bounded indexes. Let p be an arbitrary point, and let A be the set of assumptions active at that point. A corresponds to the set of dynamic clauses live at the corresponding point in execution of the logic program. We require that the first argument of every $a \in A$ is a natural number less than some fixed constant *MAX_INDEX* and distinct from all other such numbers in A. This allows simple, efficient indexing of dynamic clauses.

Prolog interpreters typically enter atomic dynamic clauses in hash table for efficient matching, using one of the predicate's arguments as the hash key. Our logic programs can be written with this very restricted form of clause indexing.

Example. The even-odd proof scheme of Figure 5 is a logic program that conforms to these restrictions. The proof scheme (1) executes in linear time and space, and it is (2) syntax-directed. Its dynamic clauses $bind(x, \tau, \Gamma)$ are all atomic. In our implementation of this proof scheme, we put the x argument of $bind(x, \tau, \Gamma)$ in the first position to conform to the (3) *bounded indexes* rule; and all the indices x are manifest constants that are small integers.

Our LTAL proof scheme used in the real PCC system also obeys these restrictions.

4.3 Constructing programs

A logic program is presented to Flit's interpreter as a set of LF terms, represented using an expression data structure (*expr*) [6]. These terms are built from the LF constants declared in the trusted computing base. They represent proofs of theorems derivable from the axioms represented by the constants. The type of each term represents the lemma that it proves. The set of these lemmas is essentially the logic program to be used. To massage the LF type into a form that is convenient for logic programming, however, the following transformations are applied in order:

Mark_logic_vars. The LF type is of the form

$$\Pi x_1 : \tau_1. \ldots \Pi x_m : \tau_m. \Pi a_1 : G_1. \ldots \Pi a_n : G_n. G$$

where $x_1, \ldots, x_m$ are the typed logic variables, $G_1, \ldots, G_n$ are the subgoals of the clause, and G is the head of the clause. Flit distinguishes Π-abstractions declaring logic variables and Π-abstractions stating subgoals by checking whether the bound variables, i.e. x_i and a_j ($0 < i \leq m$ and $0 < j \leq n$), occur free in their scopes. If a bound variable occurs free in its scope, it is a logic variable; otherwise, its type is a subgoal. The logic variables $x_1, \ldots, x_m$ are marked with a flag, for use in subsequent processing.

Massage_type. The head of the clause is originally buried beneath Π-abstractions for the logic variables and the subgoals of the clause. For more efficient matching, we massage the type to get the following:

$$G \leftarrow G_n, \ldots, G_1, check(x_m), \ldots, check(x_1)$$

This puts the head at the top of the expr, and ensures that subgoals will be solved in the appropriate order, with G_n solved first. We put expressions $check(x_i)$ on this list after the G_i because if all the subgoals G_i succeed, we must then check that the logic variables have all been instantiated to ground terms; this is necessary to ensure that the proof is not dependent on some nonexistent element of an empty type.

Rename_vars. Finally, all the logic programming variables of the clause are replaced by distinct fresh variables.

For a clause to add an atomic dynamic clause G before checking a particular subgoal G', it should contain an expression of the form $\Pi a : G. G'$ in its list of subgoals.

4.4 Solving goals

After a logic program has been obtained in the way described in the previous section, Flit's logic programming interpreter can solve goals with respect to that program. The algorithm to do this is given in pseudocode in Figure 6. This `run_lp()` function aborts if no solution was found for the goal, and terminates normally otherwise. If a solution is found, a single global substitution is updated to hold the unifier. We use the standard simple unification algorithm, and unroll the unifier lazily while performing unification.

There are four cases in the code of Figure 6. The first is for solving a pair of subgoals, which is done in the obvious way. The second case of `run_lp()` is for dynamic clauses. We add the dynamic clause G, then solve the subgoal G', and then remove the dynamic clause. When adding the dynamic clause, the checker aborts if the first argument position is not a positive number or if some other active dynamic clause has this same number in its first argument position. As described in Section 4.2, the interpreter is not complete for programs where these conditions are violated. Otherwise the dynamic clause is added to a static array at the index given by the number in its first argument position. The third case implements the check described in Section 4.3 that makes sure logic variables are ultimately instantiated with ground terms. In fact, the check makes sure that each logic variable is instantiated with an acyclic ground term. This is sufficient to implement the occurs check, which is needed for soundness.

The default case of `run_lp()` is for atomic goals. This is the only other form of goals allowed by the conditions of Section 4.2. To solve an atomic goal, we try to find a clause whose head unifies with the goal. First, if the goal is an arithmetic goal such as $+\ 3\ 4\ X$ we perform the fixed-precision arithmetic. Next, if the first argument of the goal

```
bool run_lp(expr goal)  {
  int curr_expr_index = next_expr_index;
  case goal of
    (G,G') =>
         run_lp(G); run_lp(G');
    (G -> G') =>
         int i = add_dynamic_clause(G);
         run_lp(G');
         remove_dynamic_clause(i);
    check(t) =>
         ensure(ground(t));
    default =>
         expr clause = find_unifying_clause(goal);
         if (!clause) abort();
         else if (clause has subgoals S) {
            new_goal = apply_unifier_and_rename(S);
            clear_clause_vars();
            run_lp(new_goal);
         }
  esac
}
```

Figure 6: Logic program interpreter

is a positive number, we consult the dynamic clause table. Then we try static clauses of the program; we index on the predicate symbol of the goal.

If no matching clause is found, then `run_lp()` aborts. Otherwise, it either succeeds if the clause is atomic, or else applies the unifier to the clause's subgoals and tries to solve them by recursive call to `run_lp()`. When the unifier is applied, any uninstantiated logic variables are renamed to fresh variables. Bindings for the static clause variables are cleared after applying the unifier. Note that if a clause has more than one subgoal, the subgoals are already packaged up in order using commas (by "massage-type"), so the first case of `run_lp()` will be used in the recursive call to solve them in order.

5. PROOF WITNESSES

Our even-odd example is overly simplistic in that there is a syntax-directed decision procedure for the main safety theorem: for an expression E, if the formula $\mathit{safe}(E)$ is true, then the proof is easily found. In a real proof-carrying code application, the program E is in machine language; loops and recursion in the program, and quantified types in the type system, make type inference impossible.

Thus, in a PCC application, the input to the prover includes the program E and also an untrusted hint H. The hint provides loop invariants, type annotations, and other information which can be used by the prover. Because the hint is provided by the same adversary who provides the program, H cannot be assumed accurate, but it can still be useful in constructing the proof.

We will illustrate using the even-odd example. Let us provide a hint H which is a list of type annotations, $x_1 : \tau_1, x_2 : \tau_2, \ldots, x_n : \tau_n$. We will write a prover that uses this hint (even though for this simple language the hint is not necessary). The root goal is now $\vdash_p \; H \; E$ instead of $\mathit{safe}(E)$.

In addition to running the logic program on the root query $\vdash_p \; H \; E$, the checker verifies a (static) proof of the lemma,

$$\frac{\vdash_p \; H \; E}{\mathit{safe}(E)} \; .$$

We can't use this as a logic-programming rule, i.e. we can't use $\mathit{safe}(E)$ as our query, because then the logic program would have to "guess" H, which could require unbounded backtracking.

The hint H serves as a *proof witness* for E, in conjunction with the Prolog program (i.e. proof scheme) and its semantic soundness proof.

5.1 Layers of specification and proof

To handle proof-checking with hints, the checker software must process separately several layers of specification, semantics, proof, and logic-programming clauses. It is useful to think in terms of a *proof consumer* and an *adversary.*

	Axioms	stage 1
Trusted↑	Expression Operators	
Untrusted↓	Semantic Model	stage 2
	Hint Operators	
Proof scheme	Clauses	
Theorem to be proved	Expression	stage 3
Proof witness	Hint	

Stage 1. The proof consumer specifies the *Axioms* of a logic, and defines the kinds of theorems she wants to check—that is, the language of expressions for which she wants safety theorems—by defining *Expression Operators.* One of the expression operators must be a predicate called *safe*.

Stage 2. Then the adversary sends a proof scheme, that is, a logic program (the syntactic type checker in the even-odd example). This program manipulates goals expressed using the *Expression Operators* and the *Hint Operators.* All the hint operators must be defined in terms of the underlying logic—the adversary is not permitted to add uninterpreted operators to the logic. All the *Clauses* of the logic program must be proved as derived lemmas in the logic, from the definitions of the expression and hint operators, as Lemma 1 does.

The *Semantic Model*, sent by the adversary, is simply a set of supporting definitions and lemmas, defined in terms of the underlying logic, that can be useful in defining the hint operators and the clauses.

The adversary may define as many hint operators and clauses as he likes; however, there must be one operator called $\vdash_p$, and the semantic model must contain a lemma of the form,

$$\frac{\vdash_p \; H \; E}{\mathit{safe}(E)} \; .$$

The proof consumer uses the logical framework (LF) to check the wellformedness of all the definitions and the proofs of all the lemmas. Then she loads the *Clauses* into the subset-Prolog interpreter.

Stage 3. Finally, the adversary sends an *Expression* and a *Hint.* The consumer needs to verify that the expression obeys her desired safety property—this was the point of the

Axioms

$$\frac{A \Rightarrow B \quad A}{B}\ \text{imp\_e} \qquad \frac{\forall x.A(x)}{A(B)}\ \forall\text{\_e} \qquad \textit{et cetera}$$

Expression Operators

$$\begin{array}{lcl}
Var & \equiv & Num \\
State & \equiv & Var \to Num \to Form \\
Decl & \equiv & State \to Form \\
Exp & \equiv & State \to Num \to Form \\
Prog & \equiv & \langle Decl, Exp\rangle \\
(d;e) & \equiv & \langle d,e\rangle \qquad \cdot \quad \equiv \quad \lambda s.\ \text{true} \\
\mathsf{let} & \equiv & \lambda x.\lambda e.\lambda d.\ \lambda s.\ d\ s \wedge (\forall a.e\ s\ a \Rightarrow s\ x\ a) \\
x & \equiv & \lambda s.\lambda a.\ s\ x\ a \\
n & \equiv & \lambda s.\lambda a.\ a = n \\
+ & \equiv & \lambda e_1.\lambda e_2.\ \lambda s.\lambda a.\ \exists a_1.\exists a_2. \\
 & & e_1\ s\ a_1\ \wedge\ e_2\ s\ a_2\ \wedge a = a_1\ plus\ a_2 \\
safe & \equiv & \lambda p.\ \forall s.\ fst(p)\ s \Rightarrow \\
 & & \quad \exists a.\ snd(p)\ s\ a \wedge \text{isEven}(a)
\end{array}$$

Semantic Model

$$\begin{array}{lcl}
Ty & \equiv & Num \to Form \\
Env & \equiv & State \to Form \\
\exists! & \equiv & \lambda F.\ \exists x.\ F\ x\ \wedge\ \forall y.\ F\ y\ \Rightarrow\ x = y \\
\text{upd} & \equiv & \lambda x.\lambda a.\lambda s.\ \lambda y.\lambda b.\ \text{if}\ (x = y)\ (a = b)\ (s\ y\ b) \\
\mathsf{even} & \equiv & \lambda x.\exists n.\ \text{isInt}(n) \wedge x = 2n \\
\mathsf{odd} & \equiv & \lambda x.\exists n.\ \text{isInt}(n) \wedge x = 2n+1 \\
\vdash_p & \equiv & \lambda h.\lambda p.\lambda\tau.\ \forall s.\ fst(p)\ s \Rightarrow \exists a.snd(p)\ s\ a \wedge \tau\ a \\
\vdash_d & \equiv & \lambda\Gamma.\lambda h.\lambda d.\lambda e.\lambda\tau.\ \forall s.\ (\Gamma \sqsubseteq d \wedge \Gamma\ s) \Rightarrow \\
 & & \quad \exists a.\ (e\ s\ a \wedge \tau\ a) \\
\vdash_e & \equiv & \lambda\Gamma.\lambda e.\lambda\tau.\ \forall s.\ \Gamma\ s \Rightarrow \exists a.\ (e\ s\ a \wedge \tau\ a) \\
\boxplus & \equiv & \lambda\tau_1.\lambda\tau_2.\lambda\tau.\ \forall n_1.\forall n_2. \\
 & & \tau_1\ n_1 \Rightarrow \tau_2\ n_2 \Rightarrow \tau\ (n_1 + n_2) \\
bind & \equiv & \lambda x.\lambda\tau.\lambda\Gamma.\ \forall s.\ \Gamma\ s \Rightarrow \exists a.\ (s\ x\ a \wedge \tau\ a) \\
\sqsubseteq & \equiv & \lambda d_1.\lambda d_2.\ \forall s.\ d_1\ s \Rightarrow d_2\ s
\end{array}$$

Hint Operators

$Ty \quad Env \quad \mathsf{even} \quad \mathsf{odd} \quad \mathsf{typeof} \quad \cdot$

Clauses

$$\begin{array}{l}
safe(p)\ \leftarrow\ \vdash_p p : \mathsf{even}. \\
\vdash_p (d;e) : \tau\ \leftarrow\ d \vdash_d (d;e) : \tau. \\
\Gamma \vdash_d (\mathsf{typeof}\ x : \tau_1; h) \parallel (\mathsf{let}\ x = e_1; d)\ ; e : \tau\ \leftarrow \\
\qquad \Gamma \vdash_e e_1 : \tau_1\ \leftarrow \\
\qquad (bind(x, \tau_1, \Gamma)\ \to\ \Gamma \vdash_d (h \parallel d\ ; e) : \tau). \\
\Gamma \vdash_d (\cdot \| \cdot ; e) : \tau\ \leftarrow\ \Gamma \vdash_e e : \tau. \\
\Gamma \vdash_e x : \tau\ \leftarrow\ bind(x, \tau_1, \Gamma). \\
\Gamma \vdash_e e_1 + e_2 : \tau\ \leftarrow\ \Gamma \vdash_e e_1 : \tau_1\ \leftarrow \\
\qquad \Gamma \vdash_e e_2 : \tau_2\ \leftarrow\ \tau_1 \boxplus \tau_2 = \tau. \\
\mathsf{even} \boxplus \mathsf{even} = \mathsf{even}. \qquad \mathsf{even} \boxplus \mathsf{odd} = \mathsf{odd}. \\
\mathsf{odd} \boxplus \mathsf{odd} = \mathsf{even}. \qquad \mathsf{odd} \boxplus \mathsf{even} = \mathsf{odd}.
\end{array}$$

Expression

$\mathsf{let}\ x = 4\ ;\ \mathsf{let}\ y = x + 8\ ;\ x + y$

Hint

$\mathsf{typeof}\ x\ \mathsf{even}\ (\mathsf{typeof}\ y\ \mathsf{even}\ \cdot)$

Figure 7: Proof scheme for even-odd system. Not shown are the proofs (in higher-order logic) of all the clauses.

whole exercise!—and she will do it using the adversary's proof scheme. Since the proof scheme was proved sound (and she has checked the proof), then if the logic program completes successfully, then $safe(E)$ must be valid.

For the even-odd system, the implementation of these stages is shown in Figure 7; sample source code written in Twelf is in Appendix A.

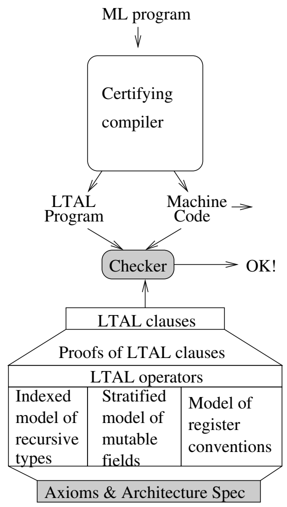

Figure 8: Foundational PCC Framework. Trusted components are shaded.

What is a proof witness? Stage 1 (loading axioms and safety predicate) needs to be done only once per safety policy. In a PCC application, stage 2 (loading the proof scheme) would need to be done when there are substantial modifications to the the untrusted compiler. Stage 3 is repeated for each compiled program sent from the compiler to the consumer. Clearly, any work done in stages 1 and 2 can be amortized over many executions of stage 3. Although the foundational proof derives from information transmitted in stages 2 and 3, in measuring the effective size of proof witnesses we can consider just the *Hint* sent in stage 3.

6. APPLICATION: FOUNDATIONAL PCC

The even-odd type system is just a toy example to demonstrate some of the principles. Our real applications are in proof-carrying code and distributed authorization. Our checking system scales up to these examples quite well, as we will explain.

In our application to foundational PCC, the hint H is an expression in a calculus called Low-level Typed Assembly Language (LTAL) [7], and the expression E is a machine-language program, that is, a sequence of 32-bit natural numbers.

Figure 8 shows the major components of our foundational proof-carrying code framework. The *LTAL clauses* are a set of clauses in our restricted Prolog subset. *Axioms & Architecture Spec* are preloaded into our *Checker* and must be trusted as axioms and trusted definitions.[6] Between these two components are proofs, based on the axioms, of all the *LTAL clauses.*

A source program is compiled into a machine-code program and an LTAL expression. The compiler is not trusted, because it is a large program that may have bugs. The trusted checker receives the LTAL clauses, along with their soundness proofs in higher-order logic; checks the soundness proofs; and then runs the LTAL checker, which is a syntax-directed computation in our subset Prolog.

[6]A trusted definition is one that is used in the statement of the theorem to be proved; an untrusted definition is used only in the proof.

programs	P	$::=$	$(M, \vec{B}, l_s)$
basic blocks	B	$::=$	$f[\vec{\alpha}](v_1 : \tau_1, \ldots, v_n : \tau_n)S$
maps	M	$::=$	(L, R, T)
label map	L	$::=$	$\{l_1 \mapsto a_1, \ldots, l_n \mapsto a_n\}$
register map	R	$::=$	$\{v_1 \mapsto r_1, \ldots, v_n \mapsto r_n\}$
type abbrev. map	T	$::=$	$\{t_1 \mapsto \tau_1, \ldots, t_n \mapsto \tau_n\}$
instr. sequence	S	$::=$	$\iota; S \mid \mathsf{branch} \mid \mathsf{jmp}$
instructions	ι	$::=$	$\mathsf{add}\ v_d, v_1, v_2 \mid \cdots$ (53 more)
types	τ	$::=$	$\alpha \mid \top \mid \bot \mid \mathrm{int}_{32} \mid \exists \alpha.\tau \mid \cdots$

Figure 9: LTAL Syntax

Chen *et al.* [7] describe the LTAL and the compiler that produces it; Tan *et al.* [19] describe the semantic model of the LTAL. In this paper we focus on the aspects of the LTAL calculus that enable it to be type-checked by our tiny trusted checker.

Because a source-language programmer never sees the LTAL program, we can design the LTAL calculus to be checkable in our very restricted language. To use the checker's limited support for dynamic clauses, we have arranged the LTAL so that: All identifiers in LTAL are small integers. No variables have the same identifier. Program labels, local variables, and type abbreviations are represented by disjoint sets of integers. To make the LTAL type system entirely syntax-directed, we use explicit coercions to guide the typing rules, instead of relying on subtyping which would require a search.

We use the simple and limited arithmetic provided by the checker: addition, multiplication, and truncating division on 32-bit natural numbers. Other operators are synthesized, such as $A > B$ by `div` B A 0, using truncating division.

6.1 Syntax of LTAL

The syntax is illustrated in Figure 9. An LTAL program consists of various maps (including type abbreviation declarations, label map, and register map), a set of function declarations, and a start label. Function declaration $f[\vec{\alpha}](v_1 : \tau_1, \ldots, v_n : \tau_n)S$ defines a function (basic block) with label f, type parameters $\vec{\alpha}$, formal parameters $v_1 : \tau_1, \ldots, v_n : \tau_n$, and function body S which is a sequence of LTAL instructions. The function label f is assigned a code pointer type $\mathbf{codeptr}[\vec{\alpha}](v_1 : \tau_1, \ldots, v_n : \tau_n)$.

The label environment L is a map from program labels to their addresses. The register-allocation environment R maps variables to temporaries (registers or spill locations). The type abbreviation environment T maps type abbreviations to their expansions. Type abbreviations are used to gain concise type expressions and the type checker opens a type abbreviation when needed.

6.2 Typing rules

The LTAL has hundreds of clauses. Here we will show just one: a rule for Sparc `add` instruction, shown in Figure 10.

The first and second premises state that both x and y have type int_{32}, 32-bit integer type. Environment LRT is label, register allocation, and type abbreviation maps. Address ℓ is the location of current instruction $v \leftarrow x + y$; address ℓ' is the location of the next instruction. Premise (3) specifies that this instruction is 4 bytes long.

$$\frac{\begin{array}{llll} (1) & LRT; \rho; \Phi \vdash_v x : \mathrm{int}_{32} & (2) & LRT; \rho; \Phi \vdash_v y : \mathrm{int}_{32} \\ (3) & \ell' = \ell + 4 & & \\ (4) & R(v) = t_v & (5) & R(x) = t_x \\ (6) & \mathrm{realreg}(t_v) = r_v & (7) & \mathrm{realreg}(t_x) = r_x \\ (8) & y_m = \mathrm{match\_reg\_or\_imm}(y) & & \\ (9) & \Phi' = \{v : \mathrm{int}_{32}\} \cap (\Phi \backslash v) & & \\ (10) & \mathrm{decode\_list}\ \ell\ \ell' P\ P'\ \mathrm{i\_ADD}(r_x, y_m, r_v) & & \end{array}}{LRT; \Gamma \vdash_\iota (\ell; \rho; \hbar; \Phi; P)\{v \leftarrow x + y\}(\ell'; \rho; \hbar; \Phi'; P')}$$

Figure 10: Typing rule for add instruction.

Premises (4), (5), (6), and (7) map variables to temporaries and temporaries to registers. They use the R component of the LRT environment; R is a context managed with dynamic clauses.

Premise (8) matches a particular Sparc addressing mode; premise (9) relates the value typing contexts Φ, Φ' before and after execution of the current instruction. The Φ context is small (it just maps currently live local variables) and is represented as a list, not with dynamic atomic clauses.

The *decode_list* relation in premise (10) maps an instruction encoding (i.e., an integer) to its semantics. Specifically, it says that the instruction word at the beginning of machine code P with length $\ell' - \ell$ is an add instruction $\mathrm{i\_ADD}(r_x, y_m, r_v)$. Machine code P is a sequence of integers (instruction words); the pair (P, P') is a conventional Prolog difference-list.

The conclusion is like a Hoare-logic judgement. In environment LRT, the instruction $v \leftarrow x + y$ is at location ℓ; the length of the instruction is $\ell' - \ell$; this instruction does not affect type contexts ρ or heap allocation environment $\hbar$; value context Φ becomes Φ' after execution; the machine code at location ℓ' is P'.

For a real-life program, the generated maps L, R, and T can be very large: the sizes of L and R are approximately linear in the size of the program, and we intend to be able to type-check programs with millions of instructions. In this typing rule, premises (4) and (5) look up the temporaries of variables v and x in map R; premise (8) looks up the temporary of y if it is not an immediate. Therefore, an efficient environment management scheme is necessary.

Such typing rules, though bigger and more complicated than the rules we presented for the even-odd system, can be executed by our simple subset Prolog interpreter.

7. EXPERIMENTAL RESULTS

We have measured our trusted checker on the even-odd microbenchmark and on some small but nontrivial LTAL benchmarks. Gross statistics about these proof schemes are as follows:

	EvenOdd	*LTAL*	
Core Axioms	341	341	lines of LF
App-specific	10	1522	lines of LF
Expr. Ops.	40	2	lines of LF
Sem. Model	218	~100,000	lines of LF
Hint Ops.	10	500	lines of LF
Clauses	12	3,500	lines of LF
Expression	$\sim 7N$	$\sim 2N$	tokens
Hint	$\sim 4N$	$\sim 30N$	tokens

Lines of LF does not include blank lines and comments. Expression sizes for EvenOdd are measured with N as the number of declarations, each declaration of the form let $x_i = x_j + x_k$; which is 7 tokens per declaration. Expression sizes for LTAL are measured with N as the number of machine instructions (32-bit integers) in the program to be proved safe, with two tokens per integer, for example:

```
2551193600 next_word 2181292040 next_word 2214748172 end
```

From this it should be clear why LTAL has only two *Expression Operators*; everything shown in Figure 9 is actually *Hint Operators*.

The logic program is the set of LTAL typing rules. There are several hundred LTAL *clauses* or typing rules, some of which take dozens of lines to write down, such as the one we showed in Section 6.2 for Sparc add instruction. The LTAL *semantic model*, which provides proofs of all these clauses, is rather intricate and is the subject of several other papers [4, 5, 1, 19].

Since the clauses are written in a subset of Prolog, we can execute them in a standard Prolog system. For each benchmark, we compare execution time in the (highly optimized) SICStus Prolog compiler with execution time in the Flit interpreter.

Input size	SICS	Twelf	Flit
EvenOdd			
$N = 100$	0.002	0.99	0.01
$N = 1000$	0.030	> 3600	0.05
$N = 10000$	1.460		0.26
LTAL			
$N = 32$	0.005	1.21	0.43
$N = 870$	0.183	1018	1.32
$N = 1816$	0.432	> 3600	2.19

All times are in seconds on a 2.2 GHz Pentium 4. Twelf is not designed for performance, but its advanced features make it a convenient tool for us to develop machine-checkable proofs in LF. Flit is faster than SICStus Prolog for large EvenOdd examples; EvenOdd is unrealistic because the prolog program has only a few simple clauses. Parsing the expression and hint contributes a significant portion of execution time for EvenOdd examples in SICStus Prolog. Checking LTAL, Flit is about five times slower than SICStus; this performance may be acceptable in the intended application.

Of course, execution in SICStus loses the benefits of the tiny trusted base: in that mode we don't mechanically connect the soundness proof for the LTAL clauses to the actual SICStus execution, and the SICStus Prolog compiler and interpreter also become part of the trusted base.

The Flit software currently comprises about 1169 lines of C code: the 803 lines described in Section 1.2 for parsing axioms, loading proof graphs, and LF checking the proofs have grown to 852 lines; our new logic-program interpreter is about 282 lines, and there are about 35 lines to manage the stages described in Section 5.1.

Necula's oracle-based Prolog interpreter [12] is about 800 lines of C code. It should be straightforward to use our style of LF proof-checking of Prolog clauses, but use oracle-based execution instead of our interpreter. Then, instead of an 1169-line C program, we would have a 1700-line program. In such a system, the proof witnesses would be just as tiny as Necula's, and the trusted base would be somewhat larger than that of the system we have described in this paper.

Our interpreter has no garbage collector. Checking the $N = 1816$ LTAL example consumes approximately 4 million heap nodes. To scale Flit to significantly larger inputs, garbage collection would be necessary; our trial implementation of an allocator with two-space copying collector is 70 lines of C code.

8. CONCLUSION

To make a trustworthy proof-checker with small witnesses, one should define a language for proof-schemes, with a way to represent and check soundness theorems for the proof schemes; then one should implement an interpreter to execute the proof scheme on the theorem and the witness.

Pollack explained much of this in "How to believe a machine-checked proof" [17]:

> ... I suggest that the "programming language" for the checking program be a logical framework [such as] the Edinburgh Logical Framework .... we [could] program a checker in the internal language of the framework .... The question then arises: where will we find a believable implementation of a logical framework?

We ask you to believe very little. Our implementation is based on LF, higher-order logic, and a small subset of pure Prolog, all of which are well understood; and our implementation is about as small as possible—that is, to trust our system there are less than 1200 lines of code that you have to understand.

9. REFERENCES

[1] Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. A stratified semantics of general references embeddable in higher-order logic. In *In Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS 2002)*, July 2002.

[2] Andrew W. Appel. Foundational proof-carrying code. In *Symposium on Logic in Computer Science (LICS '01)*, pages 247–258. IEEE, 2001.

[3] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *6th ACM Conference on Computer and Communications Security*. ACM Press, November 1999.

[4] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *POPL '00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 243–253, New York, January 2000. ACM Press.

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

[6] Andrew W. Appel, Neophytos Michael, Aaron Stump, and Roberto Virga. A trustworthy proof checker. In Iliano Cervesato, editor, *Foundations of Computer Security workshop*, pages 37–48. DIKU, July 2002.

[7] Juan Chen, Dinghao Wu, Andrew W. Appel, and Hai Fang. A provably sound TAL for back-end optimization. In *PLDI '03: Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, June 2003. ACM Press.

[8] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[9] Christopher League, Zhong Shao, and Valery Trifonov. Precision in practice: A type-preserving Java compiler. In *12th International Conference on Compiler Construction (CC'03)*, page to appear, April 2003.

[10] George Necula. Proof-carrying code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, New York, January 1997. ACM Press.

[11] George C. Necula and Peter Lee. Efficient representation and validation of proofs. In *In Proceedings of the 13th Annual Symposium on Logic in Computer Science*, 1998.

[12] George C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 142–154. ACM Press, January 2001.

[13] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[14] Frank Pfenning. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815, Nancy, France, June 1994. Springer-Verlag LNAI 814.

[15] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag. LNAI 1632.

[16] Frank Pfenning and Carsten Schürmann. *Twelf User's Guide (Version 1.4)*. Carnegie-Mellon Univ., 2002.

[17] Robert Pollack. How to believe a machine-checked proof. In G. Sambin and J. Smith, editors, *Twenty Five Years of Constructive Type Theory*. Oxford University Press, 1998.

[18] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Boston, 1986.

[19] Gang Tan, Kedar Swadi, Dinghao Wu, and Andrew W. Appel. Construction of a semantic model for a typed assembly language. March 2003.

APPENDIX

A. MACHINE CHECKABLE PROOFS

To illustrate the format of the machine-checked soundness proofs of the type-checking clauses, here we will show the proofs related to the rule *BindTy*. Note this is the version with hints we described in Section 5; the rule without hints is quite similar.

Since the proof is written in LF, we begin with a brief introduction to LF. LF is based on the λ-calculus with dependent types, and it has syntactic entities at three levels: objects, types, and kinds. Types classify objects and kinds classify families of types. A deductive system is represented in LF using the judgements-as-types and derivations-as-terms principle [8]: judgements (theorems) are represented as types, and derivations (proofs) are represented as terms whose type is the representation of the judgement (theorem) that they prove. In this way proof checking of the object logic is reduced to type checking of the LF terms.

In general, a definition in Twelf (an implementation of LF with many extended features) has the form: *name* : τ = *exp*. including the dot. The type τ encodes the theorem to be proved, and *exp* is a term of type τ. By judgements-as-types and derivations-as-terms principle, term *exp* is a proof of the theorem that τ encodes. And the *name* stands for the whole term *exp* with type τ, i.e. the theorem and the proof. LF and Twelf also permit introducing constructors with the form *name* : τ. In our case, we have:

```
check_decl_cons:
    |-d (typeof V Tv HINT) (let V Ev D) Gamma E T <-
    |-e Gamma Ev Tv <-
    (bind V Tv Gamma
       -> |-d HINT D Gamma E T) =
 [p1: bind V Tv Gamma -> |-d HINT D Gamma E T]
 [p2: |-e Gamma Ev Tv]
 |-d_i [s]
       [p3: pf (sub_env @ Gamma @ (let V Ev D))]
       [p4: pf (Gamma @ s)]
 cut (bind_i [s_v]
             [p7: pf (Gamma @ s_v)]
      |-e_1 p2 p7 [a_v]
             [p5: pf (Ev @ s_v @ a_v)]
             [p6: pf (Tv @ a_v)]
      cut (let_e1 (sub_env_e p3 p7) p5)
             [p8: pf (s_v @ c V @ a_v)]
      exists_i a_v
      (and_i p8 p6))
    [p10: bind V Tv Gamma]
 cut (sub_env_i [s']
                [p12: pf (Gamma @ s')]
      let_e2 (sub_env_e p3 p12))
    [p20: pf (sub_env @ Gamma @ D)]
 |-d_e (p1 p10) p20 p4.
```

The notation "`[x:t]A`" denotes $\lambda\mathtt{x}:\mathtt{t}.\mathtt{A}$. In the proof above we first introduce two λ-bindings; that is, we assume that the two premises of the typing rule hold. Then we use the *|-d* introduction rule *|-d_i* to get a proof of

`|-d (typeof V Tv HINT) (let V Ev D) Gamma E T`,

i.e. the conclusion.

The rule *|-d_i* introduces three λ-bindings: *s*, *p3*, and *p4*. Note that the type of *s* is omitted and Twelf will reconstruct it to a *State* type. Lemma *cut* is as follows:

```
cut: pf A -> (pf A -> pf B) -> pf B =
  [p1:pf A][p2:pf A -> pf B] imp_e (imp_i p2) p1.
```

The *imp_i* and *imp_e* (*modus ponens*) are introduction and elimination lemmas for implication. In general, the lemma *cut* means if we have a proof of A, and a function which maps a proof of A to a proof of B, then we can get a proof of B. This is similar to *imp_e* or *modus ponens*, but *cut* uses LF function type -> instead of object implication. When using *cut*, we first prove some formula A, then bind this proof (give it a name so that we can refer to it later) and continue to prove the goal (B in this case). The @ is the object logic level term application.