Oracle-Based Checking of Untrusted Software

George C. Necula
necula@cs.berkeley.edu

S. P. Rahul
sprahul@cs.berkeley.edu

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

Abstract

We present a variant of Proof-Carrying Code (PCC) in which the trusted inference rules are represented as a higher-order logic program, the proof checker is replaced by a nondeterministic higher-order logic interpreter and the proof by an oracle implemented as a stream of bits that resolve the nondeterministic interpretation choices. In this setting, Proof-Carrying Code allows the receiver of the code the luxury of using nondeterminism in constructing a simple yet powerful checking procedure.

This oracle-based variant of PCC is able to adapt quite naturally to situations when the property being checked is simple or there is a fairly directed search procedure for it. As an example, we demonstrate that if PCC is used to verify type safety of assembly language programs compiled from Java source programs, the oracles that are needed are on the average just 12% of the size of the code, which represents an improvement of a factor of 30 over previous syntactic representations of PCC proofs.

1 Introduction

Many software systems are being designed to be extensible and along with extensibility comes the need for mechanisms that protect the host system against misbehaving extensions. Cryptographic techniques can be used to verify the identity of the extension code [14] while a combination of run-time checking and static checking can be used to ensure that the extension has certain required semantic properties such as memory safety or bounded resource usage.

Static checking, while desirable, becomes theoretically impossible as soon as we attempt to check non-trivial semantic properties. Even more complications arise if the extension is written in a low-level language in order to reduce the amount of additional code required for translating the extension into an executable format. To address this difficulty, several techniques have been proposed, all having in common a requirement that the extension code contains additional information that allows a relatively simple checker on the host side to verify the properties of interest. Examples include the typing annotations in the Java byte code language [11] or in the Typed Assembly Language (TAL) system [16], for checking type safety of byte codes and assembly language. Proof-Carrying Code (PCC) [19] generalizes these approaches and requires the code to be accompanied by a detailed proof of compliance with the required property.

In the context of PCC it was demonstrated that a wide range of semantic properties can be verified of the untrusted extension code while using an extremely simple and easy-to-trust proof checker, even in situations when the extension code is optimized machine code [20]. For simple properties, such as type safety, it is also possible to generate the required proofs automatically while compiling a type-safe high-level language into optimized machine code [21].

One of the crucial issues for the practical applicability of Proof-Carrying Code and related techniques is the size of the proofs that must accompany the code, along with the overhead of checking them and the complexity of the trusted proof checker. In previous work, we started with what is probably the simplest possible proof checker, one that knows only how to do matching and thus must be given an excruciatingly detailed proof. It was not unusual in that implementation to see proofs that were 1000 times larger than the associated code, thus making the use of PCC impractical for all but the tiniest examples. Then we made the proof checker "smarter" so that it can infer by itself small but numerous fragments of the proofs. With this improvement proof sizes shrank to about 2.5 times the size of the code [22]. One important lesson learned then is that the proof-checking overhead is also reduced proportionally because the proof fragments that are reconstructed, unlike those present in the proof, do not need to be checked for validity; they are correct by construction.

What motivated us to look further into the issue of proof representation was the observation that PCC proof checking is not able to exploit domain-specific knowledge in order to reduce the size of the proofs. For example, PCC proofs of type safety are an order of magnitude larger than the size of the typing annotations that the Typed Assembly Language (TAL) system [16] uses. The overhead in TAL is smaller

This research was supported in part by the National Science Foundation Grant No. CCR-9875171, NSF Infrastructure Grant No. EIA-9802069, and gifts from AT&T. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

To appear in the Proceedings of the 28th ACM Symposium on Principles of Programming Languages, January 2001, London, pages 142–154.

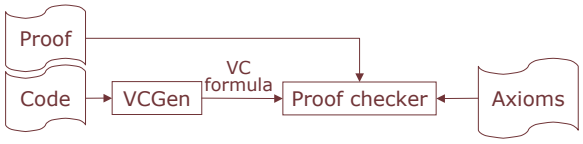

Figure 1: A previous implementation of PCC

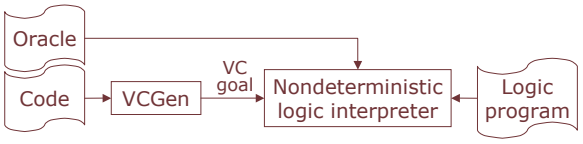

Figure 2: The oracle-based implementation of PCC.

because TAL is less general than PCC and targets a specific type-safety policy, for a specific language and type system. The TAL type checker can be viewed as a proof checker specialized and optimized for a specific logic.

We present in this paper a different implementation strategy for a proof checker that allows the size of PCC proofs to adapt automatically to the complexity of the property being checked. As a result, we do not have to pay a proof-size price for the generality of PCC in those instances when we check relatively simple properties. Using the new strategy, proofs of type safety average 12% of the size of the code, which is about 30 times smaller than before and 3 times smaller than the compression obtained by using general-purpose compression tools such as `gzip`. This significant reduction in the proof size enables PCC to handle even very large programs.

There are several components to our new strategy. First is a slightly different view on how the proofs in PCC can assist the verification on the host side. The proofs can act as an oracle guiding a nondeterministic checker for the property of interest. Every time the checker must make a choice between N possible ways to proceed, it consults the next $\lceil \log_2 N \rceil$ bits from the oracle. There are several important points that this new view of PCC exposes:

- The possibility of using nondeterminism simplifies the design of the checker and enables the code receiver to use a simple checker even for checking a complex property.
- This view of verification exposes a three-way tradeoff between the complexity of the checking problem, the complexity and "smartness" of the checker, and the oracle size. If the verification problem is highly directed, as is the case with typical type-checking problems, the number of nondeterministic choices is usually small, and thus the required oracles are small. If the checker is "smart" and can narrow down further the choices, the oracle becomes even smaller. At an extreme, the checker might explore by itself portions of the search space and require guidance from the oracle only in those situations when the search would be either too costly or not guaranteed to terminate (e.g., when the property being checked is undecidable).
- In the particular case of type-checking assembly language, the checking problem is so directed that in many situations there is only one applicable choice, meaning that no hand-holding from the oracle is needed. This explains the large difference between the size of the typing derivation and the size of the oracles in our experiments.
- This view of PCC makes direct use of the defining property of the complexity class NP. This suggests that one of the benefits of PCC is that it allows the checker to check the solutions of problems in NP in polynomial time (with help from the oracle). Furthermore, if the checker negotiates with the code producer a limit on the number of inference steps to be carried during verification, then oracles can help even in checking the solutions of semi-decidable problems as well.
- Since oracles are just streams of bits and no lookahead is necessary, they can be used in an online fashion, which is harder to do with syntactic representations of proofs. This translates in a smaller memory footprint for the checker, which is important in certain applications of PCC for embedded devices.

In the rest of this paper we describe a concrete implementation of PCC using oracles. In designing this implementation we struggled to preserve the useful property of the previous implementation of PCC, namely that it can be easily configured to check different safety policies without changing the implementation. This has great software-engineering advantages and contributes to the trustworthiness of a PCC infrastructure since code that changes rarely is less likely to have bugs.

A high-level view of a previous implementation of PCC is shown in Figure 1. The untrusted code is inspected by a verification condition generator (VCGen) that checks simple syntactic conditions (e.g. that direct jumps are within the code boundary) and constructs a formula (the verification condition) that is provable only if the code meets the safety policy. From the point of view of VCGen the safety policy is encoded as preconditions that must be established prior to performing a memory operation or a call to a runtime function. The bulk of the safety policy is encoded as axioms and inference rules that a proof checker uses as the basis for checking that the associated proof is a derivation of the verification condition.

Our choice for an easily-configurable nondeterministic PCC checker is shown in Figure 2. The VCGen is as before, except that its output is now viewed not as a logic formula but as a goal for a nondeterministic logic interpreter that uses the axioms and inference rules of the safety policy as a logic program and uses the oracle to resolve nondeterministic choices. In essence, we replaced the proof checker with a very simple nondeterministic theorem prover that can be configured easily using a logic program.

In the next section we introduce a running example and in that context we discuss the drawbacks of syntactic proof representations in the Edinburgh Logical Framework (LF). Then, in Section 3, we describe a higher-order nondeterministic logic interpreter, which we will use to verify that a proof term exists but without actually constructing it. While describing the interpreter we shall pay close attention to what are the nondeterministic choices and how can the interpreter use the oracle to resolve them. In Section 4 we move a step forward and we show how one can adapt well-known optimizations from logic programming to reduce the amount of

$$\frac{}{0 : \texttt{int}}\ \texttt{t0} \qquad \frac{E : \texttt{int}}{s\ E : \texttt{int}}\ \texttt{ts} \qquad \frac{\begin{array}{c}\overline{x : \tau_1}\ u \\ \vdots \\ E : \tau_2\end{array}}{\lambda x.E : \tau_1 \to \tau_2}\ \texttt{tlam}^{x,u}$$

$$\frac{E_1 : \tau_2 \to \tau \quad E_2 : \tau_2}{E_1\ E_2 : \tau}\ \texttt{tapp} \qquad \frac{E : T}{E : \forall t.T}\ \texttt{tgen}^{t} \qquad \frac{E : \forall t.T}{E : [T'/t]T}\ \texttt{tins}$$

Figure 3: The typing rules of implicitly-typed polymorphic λ-calculus (System F).

nondeterminism in the interpreter and thus to reduce the size of the required oracles. Finally, we discuss in Section 5 our practical experience with using this system to check type safety of Java programs compiled to machine code.

2 Example: Type Inference in System F

To illustrate the proof representation problem and then our solution, consider for the time being that the untrusted program is an expression of the implicitly-typed polymorphic λ-calculus (System F) [6, 7, 26] and the property to be checked is typeability. The syntax of expressions E and of types τ is shown below and the typing rules for the typing judgment $E : \tau$ are shown in Figure 3 in natural deduction style.

Expressions	E	::=	x	$\mid 0 \mid$	sE	$\mid E_1\ E_2 \mid$	$\lambda x.E$
Types	τ	::=	t	$\mid \texttt{int} \mid$	$\tau_1 \to \tau_2$	$\mid \forall t.\tau$	

Among types we include a base type of integers. The typing rules shown in Figure 3 use superscripts on the rule name to denote the required locality side-conditions. For example, in each use of the rule `tlam` the parameter x and the assumption "$x : \tau_1$" (named u) must be local to the derivation of the judgment "$E : \tau_2$". Similarly, the usual restriction on occurrences of the type variable in the type generalization rule is encoded as a locality condition for the `tgen` rule.

Typeability for System F is undecidable [27] and thus the code receiver must either use a conservative typeability algorithm, which rejects some valid programs, or must use some additional information to assist it during type reconstruction. An extreme case of such additional information is in the form of a complete typing derivation of the expression being checked. This would be the analog of a PCC system that uses full representation of proofs. As we shall see these proofs can be impractically large. Or, the required additional information can be in the form of type annotations placed in the expression to be checked in such a way as to reduce the typeability problem to a simpler type-checking problem. This approach would be the analog of Typed Assembly Language or the Java byte code verifier. What we want to explore here is the possibility of not sending any syntax at all (types or typing derivations) along with the expression but send instead hints to a nondeterministic type reconstructor. What remains to be discussed is how we actually build a type reconstructor that can be configured easily to work for other type systems or even as a checker for other kinds of properties.

In a previous implementation of PCC we used the Edinburgh Logical Framework (LF) [8] to encode proofs. LF

$$\frac{\Sigma(c) = A}{\Gamma \vdash c : A} \qquad \frac{\Gamma(x) = A}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash M : A \quad A \equiv_\beta B \quad \Gamma \vdash B : \texttt{Type}}{\Gamma \vdash M : B}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B} \qquad \frac{\Gamma \vdash M : \Pi x{:}A.B \quad \Gamma \vdash M' : A}{\Gamma \vdash M\ M' : [M'/x]B}$$

Figure 4: The LF typing rules for terms.

is based on the λ-calculus with dependent types and it has syntactic entities at three levels: terms (M), types (A) and kinds (K), as shown below:

M	::=	x	$\mid c \mid$	$M_1\ M_2$	$\mid \lambda x{:}A.M$			
A	::=	a	$\mid A\ M \mid$	$\Pi x{:}A_1.A_2$				
K	::=	`Type`	$\mid \Pi x{:}A.K$					

The special kind `Type` is the class of types, while the other kinds are used to describe type families dependent on terms. Typing in the LF system is with respect to a signature Σ consisting of a set of declarations "$a : K$" for type-family constants and "$c : A$" for term constants. We use the simpler notation $A \to B$ instead of $\Pi x{:}A.B$ when x does not occur free in B.

The typing judgment $\Gamma \vdash M : A$ (whose rules are shown in Figure 4) defines well-typed terms, and the typing judgment $\Gamma \vdash A : K$ defines well-formed types. In these judgments the type assignment Γ contains the types of the free variables of M, A and K.

A deductive system is represented in LF using the judgments-as-types and derivations-as-terms principle [8]. There are two major reasons why LF is a great proof representation language. One is that it allows higher-order representation techniques so that variables, binding and locality conditions in the logic or language being represented (the object language) are encoded as variables and bindings in LF (the meta language). This allows for very concise encodings even of complicated logics [2]. The other reason is that given a proper encoding of the syntactic constructors and of the derivation rules as constant declarations in a signature Σ, a necessary and sufficient condition for a derivation to be valid is that the term that encodes the derivation has the LF type that encodes the judgment being derived [2, 8].

As an example, the signature that describes the syntax and typing rules of System F is shown in Figure 6. There are three LF type constants being defined: `ty` for encoding System F types, `exp` for encoding System F expressions, and `of`, a type family indexed on expressions and types such that "`of` $E\ T$" is the type of encodings of System F typing derivations for the judgment $E : T$. Note how higher-order representation is used to encode the fact that abstraction at the level of types (`all`) and expressions (`lam`) are binding operators. In the context of this signature, the LF encoding of the expression:

$$(\lambda f.(f\ \lambda x.x)\ (f\ 0))\ \lambda y.y$$

is as an LF term E defined as follows:

$$E \stackrel{def}{=} \texttt{app}\ (\texttt{lam}\ (\lambda f{:}\texttt{exp}.\texttt{app}\ (\texttt{app}\ f\ (\texttt{lam}\ (\lambda x{:}\texttt{exp}.x)))\ (\texttt{app}\ f\ 0)))\ (\texttt{lam}\ (\lambda y{:}\texttt{exp}.y))$$

The reader can verify that E has indeed type `exp` according to the signature of Figure 6. It is possible to prove

```
(tapp (lam [F : exp] (app (app F (lam [X : exp] X)) (app F 0)))
      (lam ([X: exp] X))
      (all ([T : ty] arr T T)) int
      (tlam ([F : exp] (app (app F (lam [X : exp] X)) (app F 0)))
            (all ([T : ty] arr T T)) int
            ([F:exp][FT: of F (all ([T : ty] arr T T))]
            (tapp (app F (lam [X:exp] X)) (app F 0) int int
                  (tapp F (lam [X:exp] X) (arr int int) (arr int int)
                        (tins F ([T:ty] arr T T)
                              (arr int int) FT)
                        (tlam ([X:exp] X) int int ([X:exp][XT:of X int] XT)))
                  (tapp F 0 int int
                        (tins F ([T:ty] arr T T) int FT)
                        t0))))
      (tgen (lam [Y : exp] Y)
            ([T : ty] arr T T)
            ([T : ty] (tlam ([Y:exp] Y) T T ([Y : exp] [YT : of Y T] YT)))))
```

Figure 5: The LF term that represents the typing derivation for the System F expression $(\lambda f.(f\ \lambda x.x)\ (f\ 0))\ \lambda y.y$. Here the notation "`[X : exp] M`" denotes λX:exp.M. This term was obtained using the Elf proof assistant [24].

Syntax of types:

ty	:	Type
int	:	ty
arr	:	ty $\rightarrow$ ty $\rightarrow$ ty
all	:	(ty $\rightarrow$ ty) $\rightarrow$ ty

Syntax of expressions:

exp	:	Type
0	:	exp
s	:	exp $\rightarrow$ exp $\rightarrow$ exp
lam	:	(exp $\rightarrow$ exp) $\rightarrow$ exp
app	:	exp $\rightarrow$ exp $\rightarrow$ exp

The typing rules:

of	:	exp $\rightarrow$ ty $\rightarrow$ Type
t0	:	of 0 int
ts	:	ΠE:exp.of E int $\rightarrow$ of $(s\ E)$ int
tlam	:	ΠE:exp $\rightarrow$ exp.ΠT_1:ty.ΠT_2:ty. $(\Pi x$:exp.of $x\ T_1 \rightarrow$ of $(E\ x)\ T_2) \rightarrow$ of (lam E) (arr $T_1\ T_2$)
tapp	:	ΠE_1:exp.ΠE_2:exp.ΠT:ty.ΠT_2:ty. of E_1 (arr $T_2\ T$) $\rightarrow$ of $E_2\ T_2 \rightarrow$ of (app $E_1\ E_2$) T
tgen	:	ΠE:exp.ΠT:ty $\rightarrow$ ty. $(\Pi t$:ty.of $E\ (T\ t)) \rightarrow$ of E (all T)
tins	:	ΠE:exp.ΠT:ty $\rightarrow$ ty.$\Pi T'$:ty. of E (all T) $\rightarrow$ of $E\ (T\ T')$

Figure 6: The LF signature that encodes System F.

(e.g., with the techniques of [8]) for this encoding of System F that an expression E is typeable in System F if and only if there exist LF terms M and T such that $\emptyset \vdash M$: of $E\ T$. Our previous implementation of PCC exploits directly this theorem and expects the proof in the form of such a term M, and it uses an LF type checker to verify its correctness. The term M that is the LF encoding of a proof of typeability of our example expression E (of type int) is shown in Figure 5. This is a relatively large term (it requires 140 tokens, where a token can encode an occurrence of a variable, or a constant) but it requires only a simplistic proof checker whose most involved procedure is matching.

In [22] we show that with some modifications to the LF type checker we can omit large parts of the term. We will refer to this representation as LF_i (for implicit LF). With LF_i many redundant subterms of the above proof can be omitted, essentially keeping the skeleton consisting of the axiom names along with some higher-order terms since LF_i does not know how to perform higher-order unification. The resulting LF_i proof has only 44 tokens, which represents a factor of 3 reduction over the full proofs. The reduction is more pronounced as the proofs become bigger and it reaches factors of 1000 or more for some of the larger proofs.

Taking a close look at the typing rules for System F we observe that for any typing goal where the expression is known but the type is not there are three possible rules that could apply: `tgen`, `tins` or one of the others, determined by the structure of the expression. If the structure of the type is also known, then there are only two choices: `tins` and one of the other rules. Since there are 15 uses of typing rules and typing assumptions in the above typing derivation, a conservative calculation suggests that there are at most $\lceil \log_2 3 \rceil * 15 = 30$ bits of essential information in that tree. Everything else should be easily recovered from the typing rules and the expression being typed. If we assume that a syntactic proof representation token uses 16-bits then the 30 bits represent a reduction of a factor of 23 over the previous representation of proofs. Although this is only an approximate calculation, it comes close to the reduction factor of 30 that we observed experimentally.

This represents a significant savings over the previous representation, but more importantly we have a representa-

$$\frac{}{\Gamma \triangleright \top} \qquad \frac{\Gamma \triangleright G_1 \quad \Gamma \triangleright G_2}{\Gamma \triangleright G_1 \wedge G_2} \qquad \frac{\Gamma \vdash N : B}{\Gamma \triangleright B}$$

$$\frac{\Gamma, x : B \triangleright G}{\Gamma \triangleright \forall x : B.G} \qquad \frac{\Gamma \vdash M : A \quad \Gamma \triangleright [M/x]G}{\Gamma \triangleright \exists x : A.G}$$

$$\frac{\Gamma \vdash M : A \quad M \equiv_\beta M' \quad \Gamma \vdash M' : A}{\Gamma \triangleright M \doteq M'}$$

Figure 7: The meaning of goals.

tion that adapts easily to simpler checking problems. If we move to the simply-typed λ-calculus and eliminate the type abstraction and instantiation rules then the typing rules are syntax directed and there is absolutely no need of an oracle. And this is how it should be since there is a simple type inference algorithm for simply-typed lambda calculus.

What remains to be discussed is how we can actually build a relatively simple proof checker that can function with just the essential bits of information about the original derivation. Our choice is to implement the proof checker as a nondeterministic logic interpreter whose program consists of the derivation rules and whose initial goal is the judgment to be verified. Then we will show that by using standard optimizations for logic interpreters we can economize on the use of the oracle to the point where the oracle sizes are about a factor of 30 smaller than syntactic representations of proofs.

3 A Nondeterministic Higher-Order Logic Interpreter

There are several choices in the design of a nondeterministic logic interpreter. The simplest one is an interpreter for first-order Horn logic or maybe first-order hereditary Harrop formulas [9]. In such an interpreter the existential choices consist of what witnesses to chose for proving existentials. Such choices are postponed and solved by first-order unification. The disjunctive choices (sometimes called backchaining choices) consist of what clause to use at each step. These choices are tried in order using backtracking. In an oracle-based approach we would not need to implement support for backtracking since we are going to rely on the oracle to select the correct clause at each step. Since such an interpreter is nondeterministic complete [17], it means that for each satisfiable goal there is an oracle that guides the interpreter to success. Furthermore it is not possible for a malicious oracle to "fool" the interpreter into believing that it has solved a goal when actually it has not. Such a simple interpreter is a good choice when the safety policy can be encoded conveniently in a set of Horn clauses. In fact, this is almost the case for all of the experiments with PCC in the context of checking type safety of assembly language obtained by compiling Java programs, as described in Sections 3.2 and 5.

Notwithstanding the appeal and simplicity of Horn logic we want to show in this paper that the oracle-based approach can be adapted quite easily to more complex logics, including even higher-order logic or modal logics. In the interest of reusing the notation from before we shall assume that our logic programs are in fact expressed as LF signatures. Taking a look back at the signature of Figure 6 we see that the rules `t0`, `ts` and `tapp` are in fact Horn clauses with the Π-bound variables considered universally quantified, the type of the arguments being the subgoals and the head-type of the LF constant being the clause head. We also see that the other rules contain higher-order features that are not expressible in Horn logic. To handle these rules we need a higher-order logic interpreter, which we describe next.

To simplify the presentation we will only consider a fragment of LF, in which the syntax of types and kinds is restricted as shown below:

Level-0 types
$$A \quad ::= \quad a \;\mid\; A_1 \to A_2$$
Level-1 types (β-normal form)
$$B \quad ::= \quad a\ M_1\ \ldots\ M_n \;\mid\; B_1 \to B_2 \;\mid\; \Pi x{:}A.B$$
Level-1 kinds
$$K \quad ::= \quad \texttt{Type} \;\mid\; A \to K$$
Level-0 terms (β-normal form)
$$M \quad ::= \quad \lambda x{:}A.M \;\mid\; c\ M_1\ \ldots M_n \;\mid\; x\ M_1\ \ldots M_n$$

The level-0 types are the types of the simply-typed lambda calculus (no dependencies on terms). The only form of dependent types are the level-1 type families (for which we use the meta-variable B) and these are indexed by terms of level-0 only. Thus, the type constants occurring in level-0 types must be all of kind `Type` and the type constants occurring in level-1 types can be of kind $A_1 \to \cdots \to A_n \to$ `Type`. Note also that we restrict our attention to terms and types in β-normal form. In the rest of this paper we shall use the metavariable A only for level-0 types. We call *atomic* those types that are not functional.

Typically we use level-0 types to encode the syntax of an object logic (such as `ty` and `exp` in Figure 6) and level-1 type families to encode derivations (such as `of` in Figure 6). Note that there are no terms at level 1 since we will never reconstruct the derivation; we will only verify that one exists, which is sufficient for the purposes of Proof-Carrying Code.

This restriction to LF simplifies things considerably and at the same time it does not have a major effect on the expressiveness and hence the usability of PCC. What we disallow are level-2 and higher type families that can be indexed by level-1 objects. An example of a level-2 type family that is disallowed is `canon` declared as follows:

$$\texttt{canon} \quad : \quad \Pi E{:}\texttt{exp}.\Pi T{:}\texttt{ty}.\texttt{of}\ E\ T \to \texttt{Type}$$

and intended to encode proofs that a given typing derivation is in canonical form. In short, level-2 and higher families are used to encode deductions *about* proofs. Unfortunately, since LF types encode *open* concepts it is not possible to encode and check completely proofs by induction, which limits the usability of LF for checking level-2 proofs (see [2] for further details). Hence disallowing such families is not a great loss for Proof-Carrying Code.

A quick check of a compendium of logics encoded in LF [2] shows that our fragment covers first and higher-order logics and modal logics. It appears that the encodings of [2] for relevance logic and linear logic do not fall in this fragment because they use expression constructors that take proofs as arguments! We do not know at the time of this writing whether these logics can be encoded adequately in our fragment of LF.

Following the model of [23] we define the set of goals for our interpreter as follows:

$$G \quad ::= \quad B \;\mid\; M \doteq M' \;\mid\; \forall x : B.G \;\mid\; \exists x : A.G \;\mid\; \top \;\mid\; G_1 \wedge G_2 \;\mid$$

$$
\begin{array}{lcl}
\texttt{solve}(B_1 \to B_2) & = & \forall x : B_1.\texttt{solve}(B_2) \\
\texttt{solve}(\Pi x{:}A.B) & = & \forall x : A.\texttt{solve}(B) \\
\texttt{solve}(a\ M_1\ \ldots M_n) & = & \texttt{subgoals}(B, a\ M_1\ \ldots M_n) \\
 & & \text{where } B \text{ is the type of a level-1 constant or a level-1 quantified variable in scope here, as selected by the oracle.} \\
 & & \\
\texttt{subgoals}(B_1 \to B_2, B) & = & \texttt{subgoals}(B_2, B) \wedge \texttt{solve}(B_1) \\
\texttt{subgoals}(\Pi x{:}A.B', B) & = & \exists x : A.\texttt{subgoals}(B', B) \\
\texttt{subgoals}(a\ M'_1\ \ldots M'_n, a\ M_1\ \ldots M_n) & = & M_1 \doteq M'_1 \wedge \ldots \wedge M_n \doteq M'_n
\end{array}
$$

Figure 8: The definitions of the `solve` and `subgoals` functions.

The base goal is B for some level-1 type and expresses the problem of verifying that type B is inhabited. Again, we do not need to construct explicitly an inhabiting term. Another kind of goal is the unification goal $M \doteq M'$, which arises here only for level-0 terms with the same type. The goal $\forall x : B.G$ is used to introduce a logic parameter bound in G and similarly $\exists x : A.G$ introduces a logic variable (which is subject to unification). Note here that logic variables are all of level-0 non-dependent types. This is due to the previous restriction that level-1 dependent types are indexed only on level-0 terms.

Figure 7 defines formally the meaning of a goal using the judgment $\Gamma \triangleright G$ to denote that goal G has a solution involving logic parameters whose type is declared in the type assignment Γ.

In the case of our example, we will start the interpreter with the goal

$$\exists t : \texttt{ty.of}\ E\ t$$

where E is the encoding of the expression being verified, shown in Section 2. If we can establish that this goal is satisfied in the empty type assignment, that is, $\emptyset \triangleright \exists t : \texttt{ty.of}\ E\ t$ then we know that there exist LF terms M and T such that $\emptyset \vdash M : \texttt{of}\ E\ T$, and then by the adequacy of encoding of System F in LF we also know that E is the encoding of a typeable expression, which is what we need to verify.

We describe first a naive implementation of the logic interpreter for which it is easy to see where oracles come into play and it is also fairly easy to argue soundness and completeness. Then, in Section 4 we describe a few modifications for the purpose of using oracles economically.

The idealized interpreter works in two stages. In the first stage it eliminates all typing subgoals B by invoking the function `solve` with the type B. The definitions of the function `solve` and the helper function `subgoals` are shown in Figure 8.

The purpose of the `solve` function is to eliminate all of the typing goals and to transform the initial goal into one that contains only unification subgoals along with quantifiers. The function `solve` works recursively on the structure of the type B. For a functional type, whether dependent or not, `solve` continues recursively to verify that the result type is inhabited for any value of the argument type. When `solve` reaches an atomic type it uses the oracle to select a level-1 constant or a level-1 universally-quantified variable (i.e., parameter) in scope at that point, and then uses the helper function `subgoals` to verify that the atomic type is inhabited by a level-0 term constructed as an application whose head is the constant or parameter selected by the oracle. Theorem 1 establishes the sense in which the reduction performed by `solve` and `subgoals` is sound and complete.

Theorem 1 (Soundness and completeness of `solve`) *If Γ is a type assignment and B is a level-1 type then*

- $\Gamma \triangleright \texttt{solve}(B)$ *iff* $\Gamma \triangleright B$
- *If* $\Gamma \vdash M : B'$ *and* B *is an atomic type, then* $\Gamma \triangleright \texttt{subgoals}(B', B)$ *iff there exists* $m \geq 0$ *and* $N_1, \ldots, N_m$ *such that* $\Gamma \vdash M\ N_1 \ldots N_m : B$.

PROOF: Both facts can be proved by induction on the structure of the type B for `solve` and B' (B staying fixed) for `subgoals`. Along the same lines one can prove the termination of `solve`.

□

Before we go further, consider the operation of the `solve` procedure for the goal corresponding to our example. In this case the invocation is "`solve(of` E t)" which requires the consultation of the oracle. The are 6 level-1 constants that could be used at this point (there are no level-1 variables in scope) and thus 3 bits are fetched from the oracle. For our example, these bits will select the first choice, `tapp`. The next few invocations of `subgoal` bring us in the state shown in Figure 9.

$$
\begin{array}{l}
\texttt{solve}(\texttt{of}\ E\ t) = \\
\quad \texttt{subgoals}(\Pi E_1{:}\texttt{exp}.\Pi E_2{:}\texttt{exp}.\Pi T{:}\texttt{ty}.\Pi T_2{:}\texttt{ty}. \\
\qquad\qquad \texttt{of}\ E_1\ (\texttt{arr}\ T_2\ T) \to \texttt{of}\ E_2\ T_2 \to \\
\qquad\qquad \texttt{of}\ (\texttt{app}\ E_1\ E_2)\ T, \\
\qquad\quad t) = \\
\quad \exists E_1 : \texttt{exp}.\exists E_2 : \texttt{exp}.\exists T : \texttt{ty}.\exists T_2 : \texttt{ty}. \\
\qquad (\texttt{app}\ E_1\ E_2) \doteq E \quad \wedge \\
\qquad T \doteq t \quad \wedge \\
\qquad \texttt{solve}(\texttt{of}\ E_2\ T_2) \quad \wedge \\
\qquad \texttt{solve}(\texttt{of}\ E_1\ (\texttt{arr}\ T_2\ T))
\end{array}
$$

Figure 9: The first few steps of the `solve` procedure for our example goal.

One noteworthy fact about the `subgoals` procedure is the asymmetric treatment of the $B_1 \to B_2$ and $\Pi x : A.B$ cases. In the first case B_1 is passed further to `solve` in order to establish that it is inhabited, but we don't do that in the second case. This behavior is crucial in order to ensure good performance. In essence, this means that we never need to check terms of level-0 type; they are always well-typed by construction. And in practice there are many more subterms of level-0 in a proof than there are of level 1. If we were to check the level-0 terms then the verification would be as costly as type checking a full LF representation of the proof, and our experiments show that this would lead to several orders of magnitude increase in checking time [22]. As will

be made clear in Section 3.1, to ensure that our treatment of $\Pi x:A.B$ is sound we must first ensure that all base level-0 types a are inhabited. This is not generally true of level-1 types, which is a major motivating factor for our restriction of dependent type to level-0 dependencies.

3.1 Nondeterministic Higher-Order Unification

The `solve` procedure described in the previous section transforms the goal into one that contains only unification goals along with existential and universal quantifiers. A key observation at this point is that in all unification goals both terms have the same non-dependent level-0 type. Thus we have reduced the interpretation problem to that of higher-order unification in simply-typed λ-calculus. If we hadn't restricted type dependency in our fragment of LF then our unification problem would be in the context of dependently-typed λ-calculus, which leads to serious technical complications [5]. Fortunately, our unification problem is semi-decidable and Huet described a nondeterministic complete algorithm that can find the unifier if one exists [10].

We describe here a few details of Huet's algorithm to expose the points where the oracle can assist in the verification. In the first phase the unification goals are simplified using the rules shown below, along with their symmetric counterparts:

$$\begin{array}{ll}
\lambda x:A.M \doteq N & \rightarrow \forall x:A.M \doteq N\ x \\
(\lambda x:A.M)\ M_1\ \ldots M_n \doteq N & \rightarrow ([M_1/x]M)\ M_2 \ldots M_n \doteq N \\
r\ M_1\ \ldots M_n \doteq r\ M_1'\ \ldots M_n' & \rightarrow M_1 \doteq M_1' \wedge \ldots \wedge M_n \doteq M_n' \\
M \doteq N & \rightarrow \forall x:A.M\ x \doteq N\ x \\
\multicolumn{2}{r}{\text{whenever } M \text{ and } N \text{ have type } A \rightarrow A'}
\end{array}$$

Here r stands for a constant or a universally quantified variable. An application with such a head is called *rigid*. An application whose head is an existentially quantified variable is called *flexible*. Note that the simplification phase does not require assistance from the oracle. At the end of this phase all unification goals are of the form:

$$h\ M_1 \ldots M_m \doteq h'\ N_1 \ldots N_n$$

where h and h' are either constants or variables such that not both are rigid, and the type of the terms is a type constant. Huet observes [10] that if all unification goals contain only flexible terms then there exists a canonical unifier, as long as each atomic type is inhabited. This means in our case that each level-0 constant type a must be populated, a condition that holds for the logic shown in Figure 6 and is easy to establish for any logic.

The remaining case is that of a flexible-rigid unification goal at an atomic type a:

$$\begin{array}{l}
x\ M_1 \ldots M_m \doteq r\ N_1 \ldots N_n \\
x : A_1 \rightarrow \cdots \rightarrow A_m \rightarrow a \\
r : A_1' \rightarrow \cdots \rightarrow A_n' \rightarrow a
\end{array}$$

Since the head of the rigid term cannot change through substitution we must look for a substitution for the flexible head that will make it equal to the rigid head. There are essentially two kinds of substitutions that we must consider. One possibility is called *imitation* and substitutes x as follows:

$$\begin{array}{l}
x \doteq \lambda y_1:A_1. \ldots \lambda y_m:A_m.r\ (x_1\ y_1 \ldots y_m) \ldots (x_n\ y_1 \ldots y_m) \\
x_i : A_1 \rightarrow \cdots \rightarrow A_m \rightarrow A_i'
\end{array}$$

where x_i (for $i = 1..n$) are new unification variables of the types shown above.

The other possibility is a *projection* which attempts to project one of the arguments of x in head position. Of the m possible projections, we need to consider only those for which $A_k = A_1'' \rightarrow \cdots A_p'' \rightarrow a$ (for $k \in 1..m$). For such a projection on position k the substitution for x is:

$$\begin{array}{l}
x \doteq \lambda y_1:A_1. \ldots \lambda y_m:A_m.y_k\ (x_1\ y_1 \ldots y_m) \ldots (x_p\ y_1 \ldots y_m) \\
x_i : A_1 \rightarrow \cdots \rightarrow A_m \rightarrow A_i''
\end{array}$$

where x_i (for $i = 1..p$) are new unification variables of the types shown above.

Huet proves [10] that if there is a unifier then exactly one of the $m+1$ choices (1 imitation and m projections) will lead to it. Thus this is a perfect place to use the oracle to select among the choices.

It is important to note that in practice the number of projections is very small (and thus few bits from the oracle are needed) because of the restriction on types. Consider again our example signature shown in Figure 6. The most complicated type that the flexible head could have is `exp` $\rightarrow$ `exp` or `ty` $\rightarrow$ `ty` (these are the only higher-order types assigned to Π variables). For these types there is exactly one projection possible.

By inspection of the unification rules we observe that the unification goals remain homogeneous (both sides have the same level-0 type) and that the substitution found for unification variables matches the type of the variable. However, we must also check that existential variables are not instantiated with terms that contain universal variables whose scope does not include that of the existential variable. Thus, we must not allow the imitation rule when the head r is a universal variable whose scope is nested in that of x.

So far we have described an interpreter that first reduces typing goals to unification goals, then simplifies the rigid-rigid unification goals, simplifies the rigid-flexible goals using the assistance of the oracle and stops with a set of flexible-flexible goals that are known to be unifiable. The use of the oracle when reducing the typing goals to unification goals could in principle be avoided by a `solve` procedure that uses backtracking and tries all the choices. However, the use of the oracle in driving the higher-order unification phase is strictly necessary since the algorithm itself can diverge if there is no unifier.

Unfortunately, the number of steps that the unification algorithm might make without needing to consult the oracle is unbounded. This means that a malicious oracle can place our checker into an infinite loop. To see this consider the unification goal

$$x \doteq s\ x$$

where x is an existential variable of type `exp` and s is the LF constant that encodes the successor function. This is a flexible-rigid pair and only imitation applies, so the oracle is not consulted. Instead, a new unification variable x' of type `exp` is introduced and x is replaced by $s\ x'$. The goal now becomes $s\ x' \doteq s\ (s\ x')$ and after one simplification step we return to where we started.[1] There are several ways to solve this problem. One is to perform an occurs check along rigid paths [10], which can make the interpreter incomplete. Or

[1] This behavior could appear even in the first-order case but only for a malformed logic program: one that leads the interpreter into an infinite loop such that at each step there is exactly one applicable clause.

we can place a bound on the number of unification steps that are performed without consulting the oracle. Such a bound can be specified by the oracle itself instead of hardwiring it into the checker.

3.2 A Useful Special Case

We showed how an oracle can guide an interpreter for a powerful higher-order logic programming language. The intention is that higher-order programming offers the maximum of flexibility to the safety policy designer, who in this case will be writing the logic programs. However, it has been observed in the practice of higher-order logic programming that almost 95% of the unification goals are either simple assignment or first-order unification [13]. For this reason it makes sense to explore a weaker and simpler special case of the techniques described before where we restrict unification to be first-order.

In most of our experiments to date with Proof-Carrying Code we use a logic whose syntax contains first-order expressions E and formulas F as follows:

$$\begin{array}{rcl} E & ::= & x \mid c\ E_1 \ldots E_n \\ F & ::= & \top \mid F_1 \wedge F_2 \mid \forall x.F \mid E \mid E \Rightarrow F \end{array}$$

This logic is sufficient to encode the output of a verification condition generator whenever the basic verification conditions (e.g., memory safety) can be encoded as atomic predicates E. We will show in Section 5 how some problems where higher-order features seem to arise naturally (e.g., type checking object-oriented programs) can still be modeled within such a poor logic.

This logic can be encoded in LF as a signature whose higher-order features are restricted to the encodings of implication and universal quantification, as follows:

$$\begin{array}{lcl} \iota & : & \texttt{Type} \\ \texttt{o} & : & \texttt{Type} \\ \texttt{pf} & : & \texttt{o} \rightarrow \texttt{Type} \\ \texttt{eqsym} & : & \Pi E_1{:}\iota.\Pi E_2{:}\iota.\texttt{pf}\ (=\ E_2\ E_1) \rightarrow \texttt{pf}\ (=\ E_1\ E_2) \\ \texttt{andi} & : & \Pi P_1{:}\texttt{o}.\Pi P_2{:}\texttt{o}.\texttt{pf}\ P_1 \rightarrow \texttt{pf}\ P_2 \rightarrow \texttt{pf}\ (\texttt{and}\ P_1\ P_2) \\ \texttt{alli} & : & \Pi P{:}\iota \rightarrow \texttt{o}.(\Pi x{:}\iota.\texttt{pf}\ (P\ x)) \rightarrow \texttt{pf}\ (\texttt{all}\ P) \\ \texttt{impi} & : & \Pi P_1{:}\texttt{o}.\Pi P_2{:}\texttt{o}.(\texttt{pf}\ P_1 \rightarrow \texttt{pf}\ P_2) \rightarrow \texttt{pf}\ (\texttt{imp}\ P_1\ P_2) \end{array}$$

Here we have the level-0 types ι and `o` for encoding syntax of terms and formulas respectively, and the level-1 type family `pf` for encoding proofs. Our signatures typically contain a number of other constants whose declarations follow the model of `eqsym` above (the encoding of the inference rule for symmetry of equality). These are simple Horn clauses and do not introduce any functional existential variables and thus unification stays first order.

In this logic the initial invocation of the interpreter would be with the goal `pf` F, where F is the encoding of the verification condition. By examining the `solve` function and our logic program we observe two things:

- The `solve` function should not need assistance in proving goals whose top level symbol is a conjunction, implication or universal quantification, since in each case there is only one possible choice. Section 4 discusses one way in which the interpreter can quickly observe this property.
- The only flexible unification term that is introduced arises in the case of `alli`.

In addition to following the interpretation scheme described before, there are two other possibilities for handling the `alli` rule. First, this case falls in a subset of higher-order unification problems that Miller has identified and proved that are decidable [15]. Thus, we can extend our interpreter to handle the special cases identified by Miller. Or, since this is such a unique situation in our logic, we can hardcode in the interpreter the solution for the `alli` case: create a new logical parameter and proceed with the goal obtained by substituting it in the body of the universal quantification. The advantage is that we can avoid all complications and implementation cost due to higher-order unification. We use this simpler approach in the experiments.

4 Reducing the Amount of Nondeterminism

The non-deterministic interpreter needs to make two kinds of choices: a choice of clause when `solve` is presented an atomic type $a\ M_1 \ldots M_n$, and a choice of imitation or projection during the simplification of a flexible-rigid unification constraint. In this section we will explore ways to narrow the former kind of choices. The function `solve` must choose a constant or a variable and pass its type B on to the `subgoals` procedure. It is clear from the definition of `subgoals` that B must end in an atomic type of the form $a\ M'_1 \ldots M'_n$ or else `subgoals` would eventually fail. This restricts the choice of constants and level-1 variables that are usable. For example, in the top-level goal "`of` $E\ t$" from Figure 9, only 3 possible clauses are applicable: `tapp`, `tgen` and `tins`. To discover this we use the technique of automata-driven term indexing(ATI) [25]. Some implementations of Prolog use this technique to avoid backtracking; in our setting, we are using it to make oracles smaller.

To determine what are the choices for `solve`(`of` $E\ t$), we have to consider all the clause heads in the logic program. Let us first take the clause heads for the logic program shown in Figure 6 and replace all occurrences of variables by `_`. We obtain the following patterns:

```
1 of 0 int
2 of (s _) int
3 of (lam _) (arr _ _)
4 of (app _ _) _
5 of _ (all _)
6 of _ (_ _)
```

ATI works by constructing a decision tree based on patterns corresponding to the clause heads. Figure 11 shows the tree corresponding to the patterns shown above. There are two types of nodes: symbol nodes (shown with rounded corners) and choice nodes (shown with square corners). A symbol node corresponds to a symbol shown on the incoming edge to that node and it has a number of children equal to that symbol's arity. Each child of a symbol node corresponds to an argument of that symbol. Each choice node (with the exception of the root) is a child of a symbol node and it represents the possible choices of function symbols that can appear at the head of the corresponding argument. The descendants of a choice node are all symbol nodes. A choice node without descendants corresponds to a unification variable (i.e. `_` in the above list).

All nodes in the decision tree can be labeled with a subset of clause names (we use here the names 1–6). A clause name appears in a node if the path from the root of the tree to that node corresponds exactly to a path from the root to a leaf in the syntax tree of the pattern corresponding to the

```
choose(c E_1 ... E_n, N) =
  if N has no outgoing edge labeled c then
    return Labels(N)
  else
    let S = the successor reachable by edge labeled c
    let N_1, ..., N_n = S's children
    return Labels(N) ∪ Labels(S) ∪ (⋂_{i=1..n} choose(E_i, N_i))
choose(x E_1 ... E_n, N) =
  choose(y, N), y is a fresh variable
choose(x, N) =
  foreach S_i = successor of N, i = 1..m
    let N_1 ... N_n = S_i's children
    let L_i = Labels(S_i) ∪ (⋂_{i=1..n} choose(x, N_i))
  return Labels(N) ∪ (⋃_{i=1..m} L_i)
choose(λx : A.M, N) =
  if N has no outgoing edge labeled λ_c then
    return Labels(N)
  else
    let S = the successor reachable by edge labeled λ_c
    let N_1 = the successor of S
    return Labels(N) ∪ Labels(S) ∪ choose([c/x]M, N_1)
```

Figure 10: The automaton lookup routine.

clause head. Possible leaves are nullary constants or _. To handle higher-order variables, we replace in all patterns an application of _ to some arguments by a single _. If the pattern contains a subterm of the form $\lambda x : A.M$ then we create a fresh new constant c of type A and we represent the term $\lambda_c[c/x]M$, where λ_c acts as a unary constructor annotated with the name of the constant c. (An alternative could use deBruijn indices.)

Figure 10 shows the implementation of the `choose` function, which is given a term M and a starting choice node N in the tree and returns the maximal set of clause names that could possibly match M. The function `choose` traverses the tree based on the structure of M. If M is an application whose head is a constant, then the search continues with the symbol node reachable by an edge labeled with that constant. If no such node exists then it means that all patterns contained unification variables in this position and we simply return all the labels of N. Otherwise, if the symbol node has arity n we have to explore recursively the n subtrees corresponding to all argument positions. The returned value consists of all the labels of N and S (to account for all patters that had unification variables in the current position) along with the intersection of the sets of clauses that match all of the arguments. The other cases are similar.

Part of the decision tree is precomputed from the LF signature, but while the interpreter is solving goals of the form $\forall x : B.G$ we must extend the decision tree to include the pattern corresponding to the head of B. Note that in implementations for first-order Horn logic, which is the case discussed in [25], a static tree suffices.

Returning to our example from Section 2, `choose(of` E $t, r)$ where r is the root of the decision tree will compute `choose`$(E, n_1) \cap$ `choose`(t, n_2). Here t is an existential variable and the second invocation of `choose` returns the set $\{1, 2, 3, 4, 5, 6\}$ (this is the set of all labels present in the subtree rooted at n_2). The first invocation of `choose` will examine the structure of E and proceed to

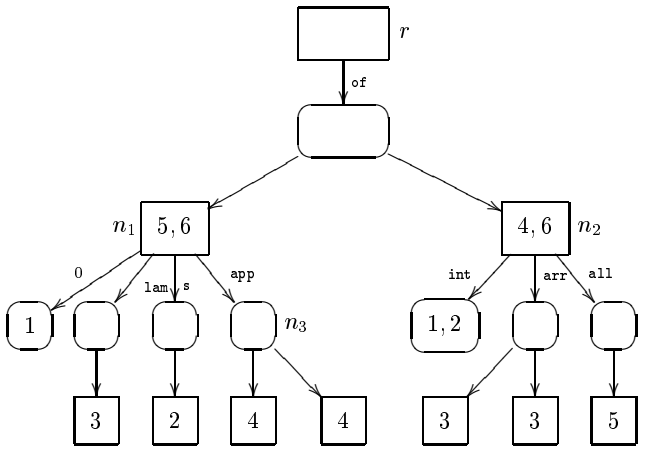

Figure 11: The automaton used to select choices for our logic program.

node n_3, since E is of the form (`app` ...). The result of the recursive call on n_3 will be the set $\{4\}$, to which we add the labels of n_1 (corresponding to the patterns 5 and 6 that have _ in this position). The result on n_1 and also from r is thus $\{4, 5, 6\}$. Among them is the constant `tapp` which is the one that will be indicated by the oracle. In this case the ATI was able to reduce the set of applicable choices to one half. For larger examples it is not uncommon for the number of possible clauses to reach 1000, the vast majority of which are dynamically created when solving goals of the form $\forall x : B.G$. Even in such situations that ATI is able to select to select less than 20 usable clauses, thus greatly improving the use of the oracle.

After the oracle selects the clause 4, the `subgoals` procedure is invoked to generate unification constraints. According to our algorithm we postpone all unification goals and thus the next goal to be solved is "`of` E_2 T_2". Since now E_2 and T_2 are both uninstantiated `choose` will not be able to do better than to say that any of the six patterns could match. However, if we performed the unification goal (`app` E_1 E_2) $\doteq E$ eagerly we would find an instantiation for E_2, which would allow `choose` to narrow the set of possible patterns to $\{2, 4, 5\}$, resulting in a better use of the oracle.

Thus, solving eagerly some unification goals can reduce our reliance on the oracles. However, we do not want to solve in this phase unification goals that require the use of oracles but only those that could be performed deterministically. Miller has observed that the majority of the unification goals fall in a special case that can be performed deterministically [15]. We can safely perform these eagerly and postpone all the others. This includes all instances of first-order unification with the expectation that in the process of carrying out these unifications, many variables will get instantiated which in turn will increase the effectiveness of the `choose` procedure.

5 Experimental Results

The experiments discussed in this section are in the context of a PCC system that checks Intel x86 binaries for compliance with the Java type-safety policy. Before VCGen

Program	Description	LOCs (Java)	Code Size (bytes)	LF_i Size (bytes)	LF_i Checking Time (ms)	Oracle Size (bytes)	Logic Interpretation Time (ms)
gnu-getopt	Command-line parser	1588	12644	49804	82.27	1936	223.28
linpack	Linear algebra routines	1050	17408	65008	117.83	2360	319.69
jal	SGI's Java Algorithm Library	3812	27080	53328	84.98	1698	314.11
nbody†	N-body simulation	3700	44064	187026	373.51	7259	814.39
lexgen	Lexical analyzer generator	7656	109196	413538	655.15	15726	1948.39
ocaml†	Ocaml byte code interpreter	9400	112060	415218	641.59	13607	1837.93
raja	Raytracer	8633	126364	371276	747.63	11854	2030.95
kopi†	Java development tools	71200	760548	3380054	5321.07	96378	14693.16
hotjava	Web browser	229853	2747548	10813894	19757.00	354034	53491.90

Figure 12: Description of our test cases, along with the size of the Java source code, machine code size, the size and time to check the LF_i representation of proofs, the size of the oracles and the logic interpretation time using oracles. † indicates that source code was not available and its size is estimated based on the size of the generated machine code.

inspects the code of methods, it first scans the object file for a class descriptor whose syntax is very close to that of the JVM class descriptors [11]. The class descriptor contains information about what Java classes are contained in the object file, what fields and methods they have and with what types, and what superclasses and superinterfaces they have. Essentially in this stage VCGen performs the same checks on the class hierarchy that a JVM byte code verifier performs. The difference from a byte code verifier is that the implementation of the methods is as native code and PCC uses the oracles to check type safety.

Using the information in the class descriptor VCGen dynamically extends the LF signature describing the logic with new level-0 constants for each class and interface name, for each method name, and for each static field name. Consider for example the following Java class:

```
class Complex {
      double imm, real;
      void zero() { imm = 0.0; real = 0.0; }
}
```

After processing the class descriptor VCGen extends the signature with the following constants:

```
_Complex  :  javaclass
if187     :  instfld _Complex 8 double
if188     :  instfld _Complex 16 double
vm325     :  vmethod _Complex 44 _Complex_zero
sp132     :  supercls _Complex _javaLangObject
```

These constants state, in order, that there is a class whose internal name is `_Complex`, that at offset 8 and 16 in an instance of the class `Complex` there are doubles stored, that at offset 44 in the virtual table of class `Complex` there is a pointer to a function that implements the same specification as that of method `Complex.zero`, and that the superclass of `Complex` is `java.lang.Object`. In addition to these constants the signature contains about 140 other static constants that define the Java type safety policy. One example, used to infer that it is safe to write a field instance of a non-null object is shown below:

$$\begin{aligned}\texttt{rdifld} \;:\; &\Pi E{:}\iota.\Pi E'{:}\iota.\Pi I{:}\iota.\Pi C{:}\texttt{javaclass}.\Pi T{:}\texttt{ty}.\\ &\quad \texttt{of}\ E\ (\texttt{instof}\ C) \rightarrow\\ &\quad \texttt{nonnull}\ E \rightarrow\\ &\quad \texttt{instfld}\ C\ I\ T \rightarrow\\ &\quad \texttt{of}\ E'\ T \rightarrow\\ &\quad \texttt{safewr}\ (\texttt{add}\ E\ I)\ E'\end{aligned}$$

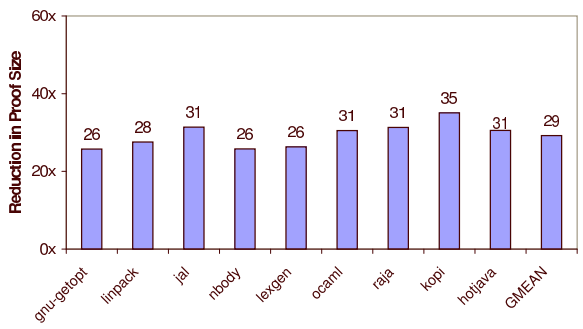

Figure 13: Ratio between LF_i proof size and oracle size.

The constant `of` is used to encode a typing judgment, `instof` is a type constructor from an expression of type `javaclass`, and `safewr` is a formula constructor that VCGen uses to request the checking of an attempted memory write at a given address and with a given value.

Our theorem prover still generates proofs in the LF_i representation. We wrote routines to convert the LF signature to a logic program and an associated ATI tree, and to convert the LF-based proofs into oracles. For the experiments in this section we used a logic interpreter with first-order unification and a special rule for handling the universal quantification introduction, as described in Section 3.2. The size of our new implementation is about the same as the original LF-based implementation, thus keeping the size of the trusted component of the PCC system roughly the same.

We carried out experiments on a set of nearly 300 Java packages of varying sizes. Some of the larger ones are described in Figure 12. The running times were averaged over a sufficient number of iterations of the checker on a 400MHz Pentium II machine with 128MB RAM.

Based on the data shown in Figure 12 we computed, for each example, how much smaller are the oracles compared to LF_i proofs (shown in Figure 13), what percentage of the code size are the oracles (shown in Figure 14) and how much slower is the logic interpreter compared to the LF_i checker (shown in Figure 15). These figures also show the geometric means for the corresponding ratios over these examples.

We observe that oracles are on average nearly 30 times smaller than LF_i proofs, and about 12% of the size of the machine code. While the binary representation of oracles

is straightforward, for LF_i it is more complicated. In particular, one has to decide how to represent various syntactic entities. For the purpose of computing the size of LF_i proofs, we streamline LF_i terms as 16-bit tokens, each containing tag and data bits. The data can be the deBruijn index [3] for a variable, the index into the signature for constants, or the number of elements in an application or abstraction.

We also compared the performance of our technique with that obtained by using a popular off-the-shelf compression tool, namely `gzip`. We do more than 3 times better than `gzip` with maximum compression enabled, without incurring the decompression time or the addition of about 8000 lines of code to the server side of the PCC system. That is not to say that oracles themselves could not benefit from further compression. There could be opportunities for Lempel-Ziv compression in oracles, in those situations when sequences of deduction rules are repeated. Instead, we are looking at the possibility of compressing these sequences at a semantic level, by discovering lemmas whose proof can be factored out.

It is also interesting to note that logic interpretation is about 3 times slower than LF_i type checking. We investigated the cause of this increase and discovered that it was due to the overhead of ATI. Indeed, if we disable the ATI module, the checking times are better than the LF_i type checking times. Moreover, we found that the cost of maintaining the (dynamic) automaton accounts for most of this overhead; the process of using it is fairly inexpensive. A close examination of the cases where our new checking times are especially bad revealed that they were trivial proofs with a large number of assumptions in scope. As explained in Section 4, we have to modify the automaton every time we enter or leave the scope of an assumption. It might be possible to optimize the verification condition for these cases and thereby reduce the checking time.

There are some simple optimizations that one can do to reduce the checking time. For example, the results shown here are obtained by using an ATI truncated to depth 3. This saves time for the maintenance of the ATI but also loses precision thus leading to larger oracles. If we don't limit the size of the ATI we can save about 8% in the size of the oracles at the cost of increasing the checking time by 24%.

One interesting observation is that while LF_i checking is faster than oracle checking, it also uses a lot more memory. While oracles can be consumed a few bits at a time, the LF_i syntactic representation of a proof must be entirely brought in memory for checking. While we have not measured precisely the memory usage we encountered examples whose oracles can be checked using less than 1Mbyte of memory while the checking of the corresponding LF_i terms could not be performed even with 1Gbyte of virtual memory.

6 Related Work and Conclusions

We described in this paper a variation on the Proof-Carrying Code idea that allows the size of proofs accompanying the code to adapt to the complexity of the property being checked. In a sense, our quest has been that of researching how much entropy there is in a typical proof used with PCC, and to find ways to exploit that automatically. Our results show that when PCC is used to verify relatively simple properties such as type safety, the essential information contained in a proof is significantly smaller than the entire proof

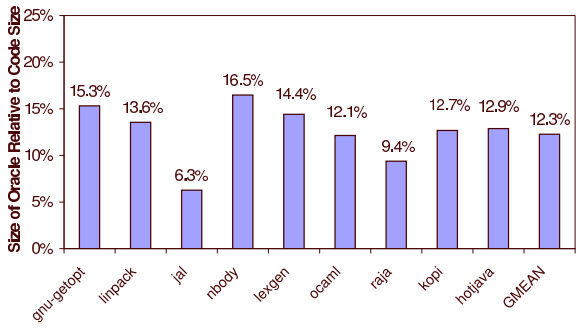

Figure 14: Ratio between oracle size and machine code.

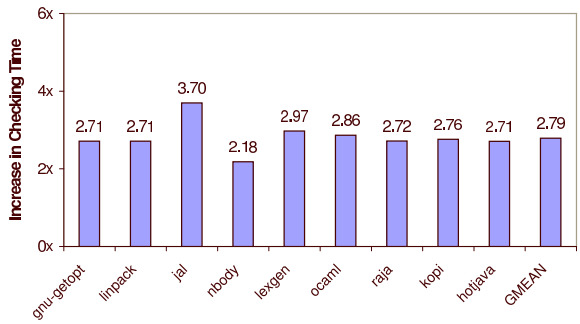

Figure 15: Ratio between logic-interpretation time and LF_i type-checking time.

or even than the skeleton of the proof. If we manage to represent just the essence of the proof then Proof-Carrying Code becomes competitive with more special-purpose techniques (such as Typed Assembly Language [16]) while remaining more general. Furthermore, the savings in the proof size due to the use of oracles allowed us to scale PCC to checking the type safety of programs up to several hundreds of thousands of lines of code.

The starting point in our research has been the observation that PCC allows the checking party the luxury of a nondeterministic checker. From this point of view the proof needs to be a sequence of bits that guide the checker to a solution. When concretizing this idea we have adapted a number of known techniques from logic programming and type theory, which allowed us to modularize the system and to separate the generic nondeterministic search engine from the details of a specific safety policy.

Our efforts in reducing the proof size are related to similar efforts whose motivation was to simplify the user interface of proof assistants such as LEGO [12], Coq [4] and Elf [24]. In those cases at least some skeleton of the proof is retained (after all, one of the objective of such tools is to present the user with a proof) while in our work we want to avoid altogether sending syntactic elements of proof.

The higher-order logic interpreter that we described in Section 3 is close to the type reconstruction algorithm in the Elf system [24, 23, 5], with two differences. We restrict our attention to a level-1 subset of LF and we never attempt to reconstruct level-1 terms (proofs), just to verify their existence. These restrictions avoid a series of technical

difficulties due to unification in the context of a higher-order language with dependent types [5]. The interpreter is also similar to a λ-Prolog [18] interpreter if one first encodes the connectives and proof rules of higher-order hereditary Harrop formulas as an LF signature. In fact, one could easily plug-in as PCC proof checkers the Elf or the λ-Prolog engines modified to use the oracle instead of search and backtracking.

The nondeterministic unification for higher-order logic of Section 3 follows closely Huet's algorithm [10] with additional observations for how an untrusted oracle can be used to guide the search.

Our work also relates to work on logic program optimizations. Currently we use the technique of automata-driven term indexing [25] with minor modifications to adapt it to a higher-order setting. However, any optimizations aimed at reducing the backtracking in a logic interpreter is directly applicable in our setting to reduce the size of the oracle, since an interpreter with fewer choices will require smaller oracles.

Formal arguments are useful to convince oneself, thus increasing the reliability of results, or to convince somebody else, thus breaking trust barriers, that a certain fact holds. The techniques of Proof-Carrying Code and Proof-Carrying Authentication [1] attempt to make a similar use of formal arguments in distributed software systems. Reliability is increased since we can better separate the trusted core of our system (e.g., the proof checker) from the bulk of it (e.g., the theorem prover, the compiler), with the language of proofs acting as an interface. And trust can be gained by presenting proofs to a party that agrees on the formalism and the language of proofs. This paper suggests that the language of proofs can be as simple as a sequence of bits used to guide a nondeterministic checking procedure to success. Although we still need more experience with this idea, especially in a higher-order setting, our empirical results suggest that this is a promising direction for further study.

7 Acknowledgments

We are indebted to Trevor Jim and Alex Aiken for their suggestions for improving a previous version of this paper. We also thank Mark Plesko, Guy Bialostoki, Michael Donohue and Andrew McCreight at Cedilla Systems for rushing to fix bugs in their certifying Java compiler thus allowing us to experiment with some large examples.

References

[1] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *5th ACM Conference on Computer and Communications Security*, pages 52–62, Singapore, Nov. 1999. ACM Press.

[2] A. Avron, F. A. Honsell, I. A. Mason, and R. Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9(3):309–354, 1992. A preliminary version appeared as University of Edinburgh Report ECS-LFCS-87-31.

[3] N. DeBruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Mat.*, 34:381–392, 1972.

[4] G. Dowek, A. Felty, H. Herbelin, G. P. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq proof assistant user's guide. Version 5.8. Technical report, INRIA – Rocquencourt, May 1993.

[5] C. Elliott. Higher-order unification with dependent types. In N. Dershowitz, editor, *Rewriting Techniques and Applications*, pages 121–136, Chapel Hill, North Carolina, Apr. 1989. Springer-Verlag LNCS 355.

[6] J.-Y. Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings 2nd Scandinavian Logic Symp., Oslo, Norway, 18–20 June 1970*, volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 63–92. North-Holland, Amsterdam, 1971.

[7] Girard, J.-Y. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Thèse de doctorat d'état, Université Paris VII, June 1972.

[8] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, Jan. 1993.

[9] R. Harrop. Concerning formulas of the types $A \rightarrow B \vee C$, $A \rightarrow (Ex)B(x)$ in intuitionistic formal systems. *Journal of Symbolic Logic*, pages 27–32, 1960.

[10] G. Huet. A unification algorithm for typed lambda calculus. *Theoretical Computer Science*, 1(1):27–57, 1973.

[11] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, Jan. 1997.

[12] Z. Luo and R. Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.

[13] S. Michaylov and F. Pfenning. An empirical study of the runtime behavior of higher-order logic programs. In D. Miller, editor, *Proceedings of the Workshop on the λProlog Programming Language*, pages 257–271, July 1992. Available as Technical Report MS-CIS-92-86.

[14] Microsoft Corporation. Proposal for authenticating code via the Internet. http://www.microsoft.com-/security/tech/authcode/authcode-f.htm, Apr. 1996.

[15] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, Sept. 1991.

[16] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[17] G. Nadathur. A proof procedure for the logic of hereditary Harrop formulas. *Journal of Automated Reasoning*, 11(1):115–145, Aug. 1993.

[18] G. Nadathur and D. Miller. Higher-order logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, chapter 8. Oxford University Press, 1998.

[19] G. C. Necula. Proof-carrying code. In *The 24th Annual ACM Symposium on Principles of Programming Languages*, pages 106–119. ACM, Jan. 1997.

[20] G. C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, Sept. 1998. Also available as CMU-CS-98-154.

[21] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 333–344, June 1998.

[22] G. C. Necula and P. Lee. Efficient representation and validation of proofs. In *Thirteenth Annual Symposium on Logic in Computer Science*, pages 93–104, Indianapolis, June 1998. IEEE Computer Society Press.

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

[24] F. Pfenning. Elf: A meta-language for deductive systems (system description). In A. Bundy, editor, *12th International Conference on Automated Deduction*, LNAI 814, pages 811–815, Nancy, France, June 26–July 1, 1994. Springer-Verlag.

[25] R. Ramesh, I. V. Ramakrishnan, and D. S. Warren. Automata-driven indexing of Prolog clauses. *Journal of Logic Programming*, 23(2):151–202, May 1995.

[26] J. C. Reynolds. Towards a theory of type structures. In *Programming Symposium (Colloque sur la Programmation, Paris)*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, Berlin, Heidelberg, and New York, 1974.

[27] J. B. Wells. Typability and type-checking in the second-order λ-calculus are equivalent and undecidable. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 176–185, Paris, France, 4–7 July 1994. IEEE Computer Society Press.