Small proof witnesses for LF

Susmit Sarkar[1]*, Brigitte Pientka[2], and Karl Crary[1]

[1] Carnegie Mellon University, Pittsburgh, USA
[2] McGill University, Montréal, Canada

Abstract. We instrument a higher-order logic programming search procedure to generate and check small proof witnesses for the Twelf system, an implementation of the logical framework LF. In particular, we extend and generalize ideas from Necula and Rahul [16] in two main ways: 1) We consider the full fragment of LF including dependent types and higher-order terms and 2) We study the use of caching of sub-proofs to further compact proof representations. Our experimental results demonstrate that many of the restrictions in previous work can be overcome and generating and checking small witnesses within Twelf provides valuable addition to its general safety infrastructure.

1 Introduction

Proof-carrying code applications establish trust by verifying compliance of the code with safety and security policies. A code producer verifies that the program is safe to run according to some predetermined safety policy, and supplies a binary executable together with its safety proof. Before executing the program, the code consumer then quickly checks the code's safety proof against the binary.

The Twelf system [22], an implementation of the logical framework LF [12], provides a general safety infrastructure to represent and execute safety policies via a higher-order logic program interpretation and has been employed in several proof-carrying code projects [4, 8, 3, 9]. Higher-order logic programming extends first order logic programming along two orthogonal dimensions: First, dynamic assumptions may be generated and used during proof search. Second, first-order terms are replaced with dependently typed λ-terms, thereby directly supporting encodings via higher-order abstract syntax.

One of the benefits of using Twelf is that the execution of a query will not only produce a yes or no answer, but produce a proof term as a certificate that can be checked independently. This increases the confidence in the overall correctness of the higher-order logic programming engine, and the certificate can be sent to the code consumer where compliance with the code is checked.

Unfortunately, the proof terms produced by Twelf are quite big in size. This creates problems in a proof carrying code setting where proof terms are sent across the network. We would like to produce small proof witnesses and check

* This work was supported by NSF ITR Grant 0121633:ITR/SY+SI:"Language Technology for Trustless Software Dissemination"

them. Our approach to this problem is to instrument the higher-order logic programming interpreter by extending and generalizing ideas by Rahul and Necula [16]. To obtain small proof witnesses, they propose to only record the non-deterministic choices during logic programming execution as a bit-string. We can check such a proof witness by guiding a deterministic logic programming interpreter using the bit-string and re-running the proof. This simple idea has been proven to be effective in many practical examples. We observe a minimum compression of a factor of 70 in proof size in our experiments, increasing up to a factor of almost 700 for larger proofs. This idea has also been used by Wu *et. al* [30] for creating a foundational proof checker with small witnesses.

Previous approaches restricted themselves to a fragment of LF excluding higher-order terms and dependent types thereby trading the expressive power of the logical framework LF against simplicity of implementation to generate and check proof witnesses. As a consequence, these systems do not support higher-order abstract syntax in practice, but each particular system now has to use encoding tricks to encode their variable binding constructs together with substitution operations. For example, Wu *et al.* [30] encode the explicit substitution calculus [1] together with the necessary proofs about substitutions for their foundational certified code implementation. As the technology of certified code evolves, we will move to more powerful and expressive safety policies and type systems and the use of higher-order abstract syntax will become crucial for achieving a simple, compact encoding of these systems.

In this paper, we describe the design of generating and checking of small proof witnesses for the full logical framework LF. This work continues where Necula and Rahul [16] left off saying “more experimental results are needed especially in the higher-order setting”. Our work has been implemented and evaluated within the Twelf system [22] making it unnecessary to build separate proof checking engines. To obtain a practical scalable implementation, we use higher-order substitution tree indexing [24]. Furthermore, we improve on the size of proof witnesses by caching common sub-proofs[1] .

This paper is structured as follows. We give background on higher-order logic programming in Twelf in Section 2. In Section 3, we present our approach to generating and checking small proof witnesses. In Section 4.1 we explain higher-order term indexing and in Section 4.2, we discuss caching techniques for factoring out common subproofs. We conclude with a discussion of some experimental results within Twelf and related work.

2 Higher-order logic programming

The theoretical foundation underlying higher-order logic programming within Twelf is the LF type theory, a dependently typed lambda calculus [20]. In this setting types are interpreted as clauses and goals and typing context represents

[1] Eliminating common sub-proofs is an orthogonal problem to eliminating redundant implicit type information, as is proposed in [17].

the store of program clauses available. We will use types and formulas interchangeably. Types and programs are defined as follows:

$$
\begin{array}{llll}
\text{Types } A & ::= P \mid A_1 \rightarrow A_2 \mid \Pi x : A_1.A_2 & \text{Programs } \Gamma & ::= \cdot \mid \Gamma, x : A \\
\text{Terms } M & ::= c \cdot S \mid x \cdot S \mid \lambda x.M & \text{Spines } S & ::= \mathsf{nil} \mid M; S
\end{array}
$$

We present terms and types using the spine notation [6]. We use meta-variables x to range over term level variables. There are constants at both the term level, denoted by c, and at type level, denoted by a. P ranges over atomic formulas such as $a \cdot S$, *i.e.* type constants applied to spines. We interpret the function arrow $A_1 \rightarrow A_2$ as implication and the Π-quantifier, denoting dependent function type, corresponds to the universal $\forall$-quantifier. Types, which are goals and clauses, are inhabited by corresponding proof terms M, and we assume that all proof terms are in normal form.

Other higher-order logic programming languages of a similar flavor are λ-Prolog [15] or Isabelle [19]. To illustrate the notation and explain the problem of small proof witnesses, we will first give an example of encoding the natural deduction calculus in the logical framework LF using higher-order logic programming following the methodology in Harper *et al.* [12]. For more information on how to encode formal systems in LF, see for example Pfenning [21]. Using this example, we will explain generating and checking of small proof witnesses.

2.1 Representing Logics

As a running example, we will consider a fragment of intuitionistic natural deduction calculus consisting of implications and universal quantifiers. Propositions can be then described as follows:

$$
\begin{array}{lll}
\text{Propositions} & A, B, C & := \ldots \mid A \supset B \mid \forall x.A \\
\text{Context} & \Gamma & := . \mid \Gamma, A
\end{array}
$$

Inference rules describing natural deduction are presented next.

$$
\frac{\Gamma \vdash [a/x]A \quad a \text{ is new}}{\Gamma \vdash \forall x.A}\,\mathsf{allI} \qquad \frac{\Gamma \vdash \forall x.A}{\Gamma \vdash [T/x]A}\,\mathsf{allE} \qquad \frac{}{\Gamma, A \vdash A}\,\mathsf{hyp}
$$

$$
\frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B}\,\mathsf{impI} \qquad \frac{\Gamma \vdash A \supset B \quad \Gamma \vdash A}{\Gamma \vdash B}\,\mathsf{impE}
$$

To represent this system in LF, we first need formation rules to construct terms for propositions. We intend that terms belonging to `prop` represent well-formed propositions and `i` represents individuals.

The connective for implication has a type that takes in two propositions and returns a proposition, hence the constructor `imp` has type `prop -> prop -> prop`. To represent the forall-quantifier, we will use higher-order abstract syntax. The crucial idea is to represent bound variables in the object language (logic) with bound variables in the meta-language (higher-order logic programming). Hence the type of `forall` is `(i -> prop) -> prop`.

Next we turn our attention to the inference rules. The judgment for provability within this logic is denoted by the type family `prov`. Each clause will correspond to an inference rule in the object logic. For convenience, we give the constructors descriptive names.

```
alli: prov (forall λx. A x)          impi: prov (imp A B)
      <- Πx. prov (A x)                    <- (prov A -> prov B).
alle: prov (A T)                     impe: prov B
      <- prov (forall λx. A x).            <- prov (imp A B)
                                           <- prov A.
```

A, B, C denote existential or logic variables which are instantiated during proof search. Throughout the example we reverse the arrow `A -> B` writing instead `B <- A`. This way, goals appear in the order in which they are processed during proof search. From a logic programming view, it might be more intuitive to think of the clause `H <- A`$_1$ `<- A`$_2$ `<- ... <- A`$_n$ as `H <- A`$_1$`, A`$_2$`, ..., A`$_n$. There are two key ideas which make the encoding of the logic calculus elegant and direct. First, we use and manipulate dynamic assumptions which higher-order logic programming provides, to eliminate the need to manage assumptions in a list explicitly. To illustrate, we consider the clause `impI`. To prove `prov (imp A B)`, we prove `prov B` assuming `prov A` In other words, the proof for `prov B` may use the dynamic assumption `prov A`. Second, we use higher-order abstract syntax to encode the bound variables in the universal quantifier. As a consequence substitution in the object language can be reduced to application and β-reduction in the meta-language (higher-order logic programming). Consider the rule for all-elimination. If we have a proof of $\forall x.A$, then we know that $[T/x]A$ is true for any term T. The substitution $[T/x]A$ in the object language is achieved via application in the meta-language `(A T)`.

2.2 Proof search in higher-order logic programming

Higher-order logic programming is similar to a Prolog interpreter in that it performs essentially a depth-first search over all the program clauses. The key challenges in moving to a higher order setting are twofold: First, we may have dynamic assumptions which may be used within a certain scope. Second, since we allow higher-order terms (i.e. terms may contain λ-abstraction), higher-order unification is used to unify clause heads with current goal.

In this section, we briefly describe the depth-first proof search procedure of the higher-order logic programming interpreter. Computation in logic programming is achieved through proof search. Given a goal (or query) G and a program Γ, we derive G by successive application of clauses of the program Γ. To solve a goal G from a set of clauses Γ, we decompose the compound goal G until it is atomic and then resolved it with a program clause. We have the following three possible actions (for a more detailed description see Miller *et al.* [14]):

Select $\Gamma \vdash G \Rightarrow c_i \cdot S$

Given an atomic goal G and clauses Γ:
Focus on a clause $c_i : A_i$ from Γ by unifying the head of A_i with the current goal G. Solve the subgoals of the clause, yielding a proof spine S. The proof term established for G is $c_i \cdot S$.

Augment $\Gamma \vdash G_1 \rightarrow G_2 \Rightarrow \lambda u.M$ if $\Gamma, u{:}G_1 \vdash G_2 \Rightarrow M$

Augment the clauses in Γ with the dynamic assumption $u{:}G_1$ and establish a proof M for the goal G_2 from the extended program $\Gamma, u{:}G_1$.

Universal $\Gamma \vdash \Pi x.G \Rightarrow \lambda x.M$ if $\Gamma \vdash [a/x]G \Rightarrow [a/x]M$ where a is a new parameter

Given a universally quantified goal $\Pi x.G$, we generate a new parameter a, and establish a proof $[a/x]M$ for the goal $[a/x]G$ in the program context Γ.

Once the goal is atomic, we need to select a clause from the program context Γ to establish a proof for G. In a logic programming interpreter, we consider all the clauses in Γ in order. First, we will consider the dynamic assumptions, and then we will try the static program clauses one after the other. Let us assume, we picked a clause A from the program context Γ. We now need to establish a proof for G, by unifying the head of the clause A with G and solving the subgoals of A. We will illustrate proof search by considering the following example:

```
prov (forall λy. (imp (forall λ x. p x) (p y)))
```

which corresponds to $(\forall y.(\forall x.p(x)) \supset p(y))$. where `p` is a defined predicate. To prove the query, we will start by unifying the head of the clause (`allI`) with the query, which results in subgoal:

$$\Pi a.\texttt{prov}(\texttt{imp}\ (\texttt{forall}\ \lambda \texttt{y.p y})\ (\texttt{pa}))$$

In the Universal step, we introduce a new parameter a yielding the subgoal:

$$\texttt{prov}(\texttt{imp}\ (\texttt{forall}\ \lambda \texttt{y.p y})\ (\texttt{pa})).$$

To prove this subgoal we will again inspect our clauses. Three of them will be applicable, namely `allE`, `impI`, and `impE`. This time we will pick the second clause `impI`. Hence we will introduce the dynamic assumption `u:prov(forall λ y.p y)` and show `prov (p a)` using the dynamic assumption `u`. In the third step, again two clauses are applicable, `allE`, and `impE`. Using the first one, `allE`, we need to show that we can prove `prov(forall λy.P y)`. There are four possible clauses whose clause head will unify: the dynamic clause `u` and the three program clauses `alli`, `alle`, and `impe`. Using the dynamic assumption `u`, we can finish the proof. Twelf's higher-order logic programming engine will generate the following proof term in explicit form:

```
(alli (λx. ((forall λy. p y) imp p x))
   λa. (impi (forall λy.p y) (p a)
          λu. (alle (λy.p y) a u))).
```

The final proof term not only tracks the rules which have been used in every step of the proof, but also tracks the instantiations for the logic variables in each steps. In the proof term above we show the instantiations in gray.

As shown in Necula [17], the instantiations of existential variables need not be recorded in the explicit proof terms but can be reconstructed as long as we only concentrate on a fragment of LF, called LF_i. This can lead to substantial savings in proof checking and proof size. Proofs are roughly $O(\sqrt{n})$, where n is the size of the query. However, extending this idea to full LF has been difficult [29]. Maybe more importantly, proofs in LF_i are still several times as big as the overall program they certify.

Our goal is to produce smaller proof witnesses by reducing the proof evidence to the choices we make while constructing the proof. In the previous example, it suffices to know that in the first step, three possible rules apply, namely `alli`, `alle`, and `impe` and we want to follow the first possibility. In the second step, again three possible rules apply, namely `alle`, `impi`, and `impe`, and we want to follow the second possibility. In the final step, we have four potential candidates, the dynamic assumption `u:forall` (λ`y.p y`), and the rules `allI`, `allE`, and `impE`. Hence it would suffice to store only a list of the choices made in the proof. In this example, the choices can be characterized by the following sequence: 1/3, 2/3, 1/2, 1/4, keeping in mind that dynamic assumptions are tried first by proof search procedures. This sequence will constitute our compact proof witness and is all that needs to be generated and sent to the verifier. In the remainder of the paper, we show how to incorporate this technique into Twelf.

3 Generating and checking small proof witnesses

3.1 Proof compression

In this section, we describe the modifications to the proof search procedure needed to generate a compact proof witness in the form of a bit-string. We assume that we already have the full proof term, which in certifying code systems is typically generated by a compiler. The bit-string encodes the non-deterministic choices within the proof, namely picking the right clause $c_i{:}A$ from the program context Γ to establish a proof P for G, once the goal G is atomic, by unifying the head of A with the atomic goal G. Potentially, there is more than one clause whose head unifies with G, and hence a proof search procedure would need to try all the possible choices in order. The proof witness just needs to keep track of which possibility was successful.

In our approach, generating and checking witnesses essentially perform the same overall proof search. The only difference is that in proof search we would likely explore multiple fruitless paths and backtrack until we find the right path. When generating and checking witnesses, we will consult the proof term or witness respectively to know which choice to consider, and thus eliminate backtracking. We modify the proof search steps presented earlier as follows:

Select $\Gamma \vdash c_i \cdot S : G \Rightarrow \underbrace{0 \ldots 0}_{1 \ldots (i-1)} 1W$

Given an atomic goal G and clauses Γ:
Let $c_i : A_i$ be the $i-$th clause from Γ whose head unifies with goal G.
Focus on clause $c_i : A_i$ from Γ. Use the proof spine S to guide the solving of subgoals, yielding witness W.

Augment $\Gamma \vdash \lambda u.M : G_1 \to G_2 \Rightarrow W$ if $\Gamma, u{:}G_1 \vdash M : G_2 \Rightarrow W$

Augment the clauses in Γ with the dynamic assumption $u{:}G_1$ and compress a proof M for G_2 within the extended program $\Gamma, u{:}G_1$ to obtain the witness W.

Universal $\Gamma \vdash \lambda x.M : \Pi x.G \Rightarrow W$ if $\Gamma \vdash [a/x]M : [a/x]G \Rightarrow W$

Given a universally quantified goal $\Pi x.G$, we generate a new parameter a, and compress a proof $[a/x]M$ for $[a/x]G$ in the program context Γ to W.

Note that the Select step is deterministic as the proof term determines which choice will be successful. It should be intuitively clear that we do not necessarily have to pass in the full proof term, but could directly produce a proof witness in form of a bit-string if our proof search is powerful enough that it will eventually find a proof.

3.2 Checking small proof witnesses

In this section, we modify the previous search procedure, in such a way that it is not parameterized by the proof term M, but rather by the compact proof witness W encoded as a bit-string. We are given goal G in a program context Γ, together with a proof witness W. The procedure is the dual of the compression case, and we show the important **Select** case.

Select $\Gamma \vdash \underbrace{0 \ldots 0}_{1 \ldots (i-1)} 1W : G$

Given an atomic goal G and clauses Γ:
Let k be the number of clauses whose head unifies with the current goal G, then inspect up to k bits, and find the i-th bit which is one.
Focus on clauses $c_i : A_i$ from Γ to establish a proof for the atomic goal G from Γ using remaining proof witness W.

In Select step, we first generate the k possible candidates whose head will unify with the current goal G. If k is greater than 1, we will examine up to k bits from the witness to see which choice to take. If a bit 1 occurs at position i of these k bits, we will pick the i-th candidate. For this idea to work, it is crucial that the order of choices during witness checking is same as during witness generation.

In order to check the proof witnesses, we re-run the prover guided with the advice encoded in the bit-string. The witness checker is then a deterministic search procedure. No backtracking is necessary, since all the non-deterministic choices are resolved. Note that the proof term does not need to be reconstructed.

3.3 Bit-string encodings for proof witnesses

The choices as described above are choice sequences of the form $i_1/k_1, i_2/k_2, \ldots$, where at the j^{th} stage we have k_j choices, and we want to pick i_j $(1 \leq i_j \leq k_j)$. With the tight coupling of the witness generation and checking phases, both phases agree on the number of choices (k_j) as well as the ordering of those choices,i.e. both producer and checker agree on which choice is to be considered the i_j-th one.

We can now see that the separator between choices is unnecessary. We can decide on a encoding scheme, and pull only the required number of bits from the oracle. The witness checker will always know how many bits to extract.

We have experimented with two simple encoding schemes, though more complex coding schemes can be imagined. The original proposal by Necula and Rahul proposed what we call the binary scheme, in that the number would be encoded in binary. If k choices apply, this will require $\lceil \log k \rceil$ bits. We discover that a scheme we call unary encoding works better in practice. In this scheme, the choice number i is encoded as $000\ldots$(i-1 zeros)1. This takes i bits.

The binary scheme will work better when we habitually have a large number of choices, and we take one of the later choices in the ordering considered by the producer/checker. The unary scheme will work better precisely in the other cases. In all our examples, we have observed that only a few choices typically apply. Further, logic programmers usually write their programs so that the more common choices are tried first. With these observations, unary encodings should outperform binary encodings, as indeed they do in experimental studies. This is a configurable option in our engine, and can be set depending on the particular proof or logic.

4 Optimizations

4.1 Higher-order term indexing

In the **Select** step of our algorithms, we need to retrieve clauses which may unify with the goal. To avoid redundant computations most first-order logic programming interpreter use efficient term indexing strategies such as automata driven indexing [28]. Indexing strategies for higher-order terms are more difficult, since in general retrieval and insertion operations rely on computing the most general unifier or the most specific generalization. However, in the higher-order case, unification is in general undecidable and the most general unifier does not necessarily exist. The same holds for computing the most specific generalization of two terms.

We will adopt higher-order substitution trees [24, 26] as our indexing mechanism. Substitution tree indexing has been successfully used in a first-order setting [11] and allows the sharing of common sub-expressions via substitutions. This is unlike other non-adaptive term indexing methods which only allow sharing of common term prefixes. To extend substitution tree indexing to the higher-order setting, we use linear higher-order patterns [27]. Higher-order patterns [13] are terms where all existential variables must be applied to distinct bound variables. Linear higher-order patterns further restrict existential variables to occur only once and to be applied to all distinct bound variables.

The construction of a substitution tree in the higher-order setting follows the overall algorithm described in Ramakrishnan *et al* [28]. We will illustrate higher-order substitution trees by an example. Assume we have the following clauses which allow quantifier manipulation for first-order logic:

$$\begin{array}{ll}
\mathsf{e1}: \mathtt{eq}\ (\mathtt{imp}\ (\mathtt{exists}\ \lambda\mathtt{x.A\ x})\ \mathtt{B}) & (\mathtt{forall}\ \lambda\mathtt{x.}(\mathtt{imp}\ (\mathtt{A\ x})\ \mathtt{B})). \\
\mathsf{e2}: \mathtt{eq}\ (\mathtt{imp}\ \mathtt{A}\ (\mathtt{forall}\ \lambda\mathtt{x.B\ x})) & (\mathtt{forall}\ \lambda\mathtt{x.}(\mathtt{imp}\ \mathtt{A}\ (\mathtt{B\ x}))). \\
\mathsf{e3}: \mathtt{eq}\ (\mathtt{and}\ \mathtt{A}\ (\mathtt{forall}\ \lambda\mathtt{x.B\ x})) & (\mathtt{forall}\ \lambda\mathtt{x.}(\mathtt{and}\ \mathtt{A}\ (\mathtt{B\ x}))).
\end{array}$$

Although all the terms in these clauses fall into the pattern fragment, not all of them are linear patterns. We linearize them by eliminating any duplicate occurrences of existential variables, and replacing any existential variable which is not fully applied with one which is. The linearized program is given next:

$$\begin{array}{l}
\mathsf{e1}: \mathtt{eq}\ (\mathtt{imp}\ (\mathtt{exists}\ \lambda\mathtt{x.A\ x})\ \mathtt{B})\ (\mathtt{forall}\ \lambda\mathtt{x.}(\mathtt{imp}\ (\mathtt{A'\ x})\ (\mathtt{B'\ x}))). \\
\qquad \forall x.(A'\ x) \doteq (A\ x) \text{ and } B'\ x \doteq B
\end{array}$$

$$\begin{array}{l}
\mathsf{e2}: \mathtt{eq}\ (\mathtt{imp}\ \mathtt{A}\ (\mathtt{forall}\ \lambda\mathtt{x.B\ x}))\quad (\mathtt{forall}\ \lambda\mathtt{x.}(\mathtt{imp}\ (\mathtt{A'\ x})\ (\mathtt{B'\ x}))). \\
\qquad \forall x.(A'\ x) \doteq A \text{ and } B'\ x \doteq (B\ x)
\end{array}$$

$$\begin{array}{l}
\mathsf{e3}: \mathtt{eq}\ (\mathtt{and}\ \mathtt{A}\ (\mathtt{forall}\ \lambda\mathtt{x.B\ x}))\quad (\mathtt{forall}\ \lambda\mathtt{x.and}\ (\mathtt{A'\ x})\ (\mathtt{B'x})). \\
\qquad \forall x.(A'\ x) \doteq A \text{ and } B'\ x \doteq (B\ x)
\end{array}$$

We now compute the most specific generalization between these clauses, and can build up a substitution tree. The algorithm for computing the most specific generalization is given in [26, 24].

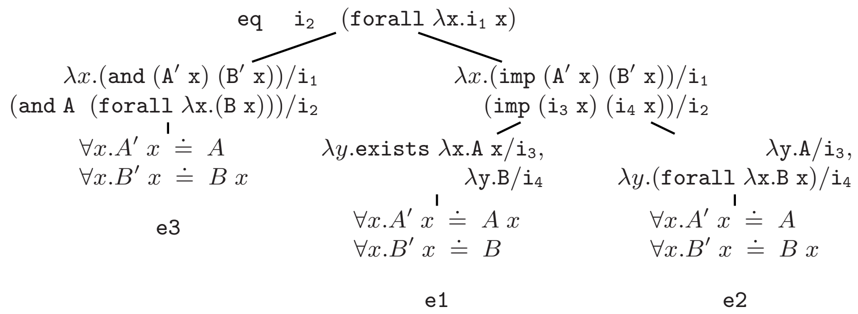

Fig. 1. Substitution tree

By composing the substitutions along a path, we will obtain a clause head. By composing the substitutions in the right-most branch, we obtain the clause head e2. In contrast to other indexing techniques such as discrimination tries, substitution trees allows the sharing of common sub-expressions instead of common term prefixes. This is often very useful, as we can see in this example, since the most sharing is done in the second argument.

We have chosen to index only the static set of program clauses. In theory, substitution tree indexing can be used for dynamic clauses generated during proof search also. However, it is not clear how useful this will be, since creating the tree itself is time-consuming. It is also noted by Necula and Rahul [16] that indexing dynamic assumptions imposes a performance penalty.

4.2 Caching results

Since large proofs often have identical subproofs, there is often potential for sharing subproofs, particularly in machine-generated proofs which tend to have repeated proofs of simple facts. This was also pointed out by Necula and Lee [18]: "... it is very common for the proofs to have repeated sub-proofs that should be hoisted out and proved only once ...".

When generating and checking small proof witnesses, this leads to two problems. First, the proof witnesses become larger, thus increasing transmission costs. Second, the performance of the witness checker may degrade, since it spends its time uselessly verifying the same fact over and over again. Ideally we would like to cache intermediate results and re-use them later.

We use ideas from and also infrastructure developed for tabled higher order logic programming [26, 23]. Since caching everything may be too costly in practice, we support selective caching. The user can declare certain predicates to be cached. We modify the Select step in our previous search procedure to allow for caching. Assume our subgoal is $\Gamma \vdash G$. We check whether the current subgoal is an instance of a previous table entry. If there exists a table entry $\Gamma' \vdash G'$ s.t. $\Gamma \vdash G$ is a variant (or instance) of the already existing entry, then a pointer to the corresponding answer list is returned. If no such entry exists, $\Gamma \vdash G$ is added to the table and a pointer to an empty answer list is returned. In this case, we will continue to focus on a clause c_i as usual to solve the goal $\Gamma \vdash G$. When we are done, we will add the answer substitution for the existential variables in $\Gamma \vdash G$ together with its proof term $c_i \cdot S$ to its answer list.

If a table entry for the goal already exists, there are two possible situations:

1. If the answer list contains an answer substitution θ_k that leads to the proof $c_i \cdot S$ we are compressing, then we will just re-use the answer substitution θ_k.
2. If the answer list does not contain an answer that would lead to the proof $c_i \cdot S$, then we need to use a program clause c_j to focus on and solve the goal $\Gamma \vdash G$.

The generation and checking of witnesses will follow similar algorithms, so both have identical caches and consider the same number of choices.

5 Experimental Results

In this section, we give an experimental evaluation of generating and checking compact proof witnesses. In particular, the results discuss the trade off between witness size and the time it takes to construct or check witnesses. Thus, we will consider three cases: asking the checker to perform proof search, proof checking of explicit proofs, and our approach of small proof witnesses. The first two represent two extreme cases, the first one with zero witness size but large proof search time, and the second one with large witness size but fast checking times. We will also discuss the trade-offs of caching subproofs. Finally, we will compare different encoding schemes for describing the non-deterministic choices and see how this affects the size of the proof witness.

Our experiments are run on a Pentium 4 machine running at 2GHz with 1 GB of memory size. The machine runs Twelf compiled by SML of New Jersey version 110.0.7, and runs it on the Redhat Linux 7.1 operating system, with no programs running on the background. We present a representative selection of results from an extensive suite of experiments we have run.

5.1 Time and size trade-offs

Our first example suite is an implementation of a sequent calculus for intuitionistic propositional logic where invertible rules are chained together thereby eliminating some non-determinism in the overall proof search.

Sequent Calculus: Times with Caching of User-Selected Predicates

Example	PST	PCT	WV	(PST/WV)	PS	PST	WS	(PS/WS)
$(A \supset B) \wedge (A \supset C) \Rightarrow A \supset B \wedge C$	0.47	< 0.01	< 0.01	∞	361	43	5	72.2
$A \vee C \wedge (B \supset C) \Rightarrow (A \supset B) \supset C$	1.70	< 0.01	0.01	170	570	50	6	95.0
$(A \supset C) \wedge (B \supset C) \Rightarrow A \vee B \supset C$	2.18	< 0.01	< 0.01	∞	561	56	6	93.5
$\Rightarrow (A \supset C) \wedge (B \supset C) \supset A \vee B \supset C$	2.43	< 0.01	0.01	243	792	57	6	132.0

Key: PST =Proof Search time (s) PS =Proof Size in bytes
PCT=Proof Checking time (s) PST=Proof Size in Number of tokens
WV =Witness Verification time (s) WS =Witness Size in bytes

We study the time to find a proof and contrast it against the proof checking times. We use the tabled higher-order logic programming engine [25, 26] to find proofs for the propositional logic. The proof compression and the verification procedures provide significant time speedups, since in these procedures, we already know the proof. Next, we turn our attention to questions of proof size. The original proof is measured both by number of bytes as well as the number of tokens, and the compact proof witness is produced using the unary encoding described earlier.

Our second example is an advanced type system for a high-level call-by-value functional language using refinement types [10]. The language has functions, a fix-point construct, booleans and bit-strings. In particular, the type of bit-strings is refined by zero and strictly positive number representations.

Refinement Type System : Proof Compression Times with Caching

Example	PST	PCT	WV	(PST / WV)	PS	PSN	WS	(PS / WS)
mult-pos-nat	5.81	0.05	1.10	5.3	15,654	1,159	169	92.6
mult	0.39	0.02	0.13	3.0	6,074	509	47	129.2
square-pos-nat	12.55	0.06	1.85	6.8	25,303	1,587	242	104.6

The experimental results demonstrates that proof checking yields a speedup between three and six times. This figure is achieved if we are caching subgoals to get maximum compressions. As we see later, even more time gains can be achieved by turning off caching, since we do not explore unproductive branches in the proof tree. Finally, we notice that the compact oracle is about 1% of the size of the proof term.

Our last example suite is an implementation from the Foundational Proof Carrying Code project at Princeton [2]. This is a large program that type checks SPARC object code with the help of annotations produced by a compiler. The type system used is a low-level type system known as LTAL [7].

FPCC: Times without Caching

Example	PST	PCT	WV	(PST / WV)	PS	PSN	WS	(PS / WS)
clos	12.26	2.505	0.47	26.1	201,910	16,502	638	316.5
mid	10.29	2.246	0.45	22.9	398,589	34,250	528	754.9
inc	11.55	2.310	0.47	24.6	410,600	35,724	579	709.2
lint	12.84	2.591	0.70	18.3	441,965	38,416	703	628.7

In the proof carrying code scenario, asking the consumer to verify our compact proof witness as opposed to doing proof search gives a speedup of about 20 times. Also important is the size of the proof that must be sent to the consumer. Our proof witnesses are between 300 and 700 times smaller than the corresponding proof terms. Finally, we notice that as proof sizes become bigger, our mechanisms perform better at compressing proofs. The space savings go from a factor of about 70 in the smallest examples all the way to about 700 times for our largest examples. The gains in time are from a factor of about 5 to a factor of about 25 for the larger examples.

5.2 Caching: time vs space

Next we investigate the practicality of caching subgoals. Caching is a fairly expensive operation, in terms of both time for stores and lookups and the memory required to maintain the table and we investigate this trade-off next. As our result show, caching results in a speed penalty of between three and fifteen times. The gain from this is that the size of the oracle is smaller for the cached version in every experiment. Disappointingly, the gain is usually small.

Refinement Type System: Caching during proof compression

Example	Compression Time			Witness Size			Table
	Cached (sec) (A)	Uncached (sec) (B)	Slowdown (A / B)	Cached (bytes) (C)	Uncached (bytes) (D)	Saving (D - C)/D	Size
mult-pos-nat	1.18	0.11	10.7	169	171	1.2 %	579
mult	0.14	0.05	2.8	47	67	29.9 %	164
square-pos-nat	2.31	0.16	14.4	242	247	2.0 %	794

5.3 Encoding Schemes

Finally, we study the issue of unary versus binary encodings of the choices. A representative study with examples from multiple example suites is given next. We notice that binary encodings always increase the size of the oracle, by between 7% and 115%. As we discussed before, logic programmers usually write their programs so that the first few clauses are the ones that are used more commonly, in which case unary encodings are better.

Unary versus Binary Encodings: no Caching

Example	WSU	WSB	(WSB - WSU/ WSU)
clos	638	715	12.0 %
mid	528	652	23.5 %
lint	703	754	7.3 %
mult-pos-nat	171	338	97.7 %
mult	67	144	114.9 %

Key WSU = Witness Size (Unary Encoded)
WSB = Witness Size (Binary Encoded)

6 Related Work

The idea of compact proof witnesses that encode the non-deterministic choice in a logic programming interpreter was first proposed by Necula and Rahul [16] for a fragment of the logical framework LF, that excludes the use of higher-order terms and significantly limits the use of dependent types in practice. Their main goal was to design a practical method for current proof-carrying code applications to reduce the size of proofs sent to a consumer. To achieve an efficient implementation, they propose the use of automata-driven indexing, where any higher-order features are ignored. Their indexing algorithm will generate a set of potential candidates from which unsound candidates need to be weeded out by calling higher-order unification based on Huet's algorithm. This is clearly wasteful and expensive in the general higher-order case, since we will traverse higher-order terms at least twice. Moreover, since Huet's unification algorithm is non-deterministic itself, their proof witnesses also record the choices made during unification. To avoid these problems in practice, their realization and their experimental evaluation does not consider terms defined via λ-abstraction.

The idea of using oracles was also explored in Wu *et al* [30]. The main difference between their and the previous approach is that the proof rules are proven correct independently thereby minimizing the trusted computing base. Trust is not our concern here, rather we aim at extending the safety infrastructure already provided by Twelf with capabilities of generating and checking small proof witnesses. This step, we believe, will provide the developers of safety policies in Twelf with new insights about the relationship of safety rules and size of proofs.

As in Necula and Rahul's work, Wu *et al.*'s system does not support higher-order abstract syntax, which drastically limits its usefulness. Wu *et al.*[30] encode the explicit substitution calculus [1] together with the necessary proofs about substitutions for their foundational implementation of LTAL. Although the overhead in this setting is still manageable, it is not general enough to handle richer safety polices.

7 Conclusion

In this paper, we extended the logical framework LF with small proof witnesses. Witness generation and checking within the logical framework LF constitutes a valuable addition to the general safety infrastructure already provided. This can provide insights into the relationship between safety policies and small safety proofs and allows for experiments with different kinds of encoding schemes. Given the potential of proof-carrying code methods and their new applications to proof-carrying authorization [3, 5], this will provide a comprehensive guide for future implementations of proof checkers which need not be restricted to first-order Prolog-like systems.

References

1. Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. Explicit substitutions. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 31–46. ACM, 1990.
2. Andrew Appel. Foundational proof-carrying code. In J. Halpern, editor, *Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS'01)*, pages 247–256. IEEE Computer Society Press, June 2001. Invited Talk.
3. Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *ACM Conference on Computer and Communications Security*, pages 52–62, 1999.
4. W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*, pages 243–253, Jan. 2000.
5. Lujo Bauer. *Access Control for the Web via Proof-carrying Authorization*. PhD thesis, Princeton University, November 2003.
6. Iliano Cervesato and Frank Pfenning. A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688, 2003.
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 ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 208–219. ACM Press, June 2003.
8. Karl Crary. Toward a foundational typed assembly language. In *30th ACM Symposiumn on Principles of Programming Languages (POPL)*, pages 198–212, New Orleans, Louisisana, January 2003. ACM-Press.
9. Karl Crary and Susmit Sarkar. Foundational certified code in a metalogical framework. In *19th International Conference on Automated Deduction*, Miami, Florida, USA, 2003. Extended version published as CMU technical report CMU-CS-03-108.
10. Rowan Davies and Frank Pfenning. Intersection types and computational effects. In P. Wadler, editor, *Proceedings of the Fifth International Conference on Functional Programming*, pages 198–208. ACM Press, 2000.
11. Peter Graf. *Term Indexing*, volume 1053 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1995.
12. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.
13. Dale Miller. Unification of simply typed lambda-terms as logic programming. In *Eighth International Logic Programming Conference*, pages 255–269, Paris, France, June 1991. MIT Press.
14. Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

15. Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus – a compiler and abstract machine based implementation of Lambda Prolog. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 287–291, Trento, Italy, July 1999. Springer-Verlag LNCS.
16. G. Necula and S. Rahul. Oracle-based checking of untrusted software. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 142–154, 2001.
17. George C. Necula and Peter Lee. Efficient representation and validation of logical proofs. In Vaughan Pratt, editor, *Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS'98)*, pages 93–104, Indianapolis, Indiana, June 1998. IEEE Computer Society Press.
18. George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In Giovanni Vigna, editor, *Mobile Agents and Security*, pages 61–91. Springer-Verlag LNCS 1419, August 1998.
19. Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.
20. 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.
21. Frank Pfenning. Computation and deduction, 1997.
22. 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 Lecture Notes in Artificial Intelligence (LNAI) 1632.
23. Brigitte Pientka. A proof-theoretic foundation for tabled higher-order logic programming. In P. Stuckey, editor, *18th International Conference on Logic Programming, Copenhagen, Denmark*, Lecture Notes in Computer Science (LNCS), 2401, pages 271 –286. Springer-Verlag, 2002.
24. Brigitte Pientka. Higher-order substitution tree indexing. In C. Palamidessi, editor, *19th International Conference on Logic Programming, Mumbai, India*, Lecture Notes in Computer Science (LNCS 2916), pages 377–391. Springer-Verlag, 2003.
25. Brigitte Pientka. Tabling for higher-order logic programming. In R. Nieuwenhuis, editor, *20th International Conference on Automated Deduction, Talinn, Estonia*, Lecture Notes in Computer Science (LNCS) (to appear). Springer-Verlag, 2005.
26. Brigitte Pientka. *Tabled higher-order logic programming*. PhD thesis, Dept. of Computer Sciences, Carnegie Mellon University, Dec 2003. CMU-CS-03-185.
27. Brigitte Pientka and Frank Pfennning. Optimizing higher-order pattern unification. In F. Baader, editor, *19th International Conference on Automated Deduction, Miami, USA*, Lecture Notes in Artificial Intelligence (LNAI) 2741, pages 473–487. Springer-Verlag, July 2003.
28. I. V. Ramakrishnan, R. Sekar, and A. Voronkov. Term indexing. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, pages 1853–1962. Elsevier Science Publishers B.V., 2001.
29. Jason Reed. Redundancy Elimination for LF. In Carsten Schuermann, editor, *Proceedings of the Fourth Workshop on Logical Frameworks and Meta-languages — LFM'04*, Cork, Ireland, 5 July 2004.
30. Dinghao Wu, Andrew W. Appel, and Aaron Stump. Foundational proof checkers with small witnesses. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 264–274. ACM Press, 2003.