Small proof witnesses for logical frameworks

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

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

Abstract. In this paper, 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 [14] in two main ways: 1) We consider the full fragment of LF including dependent types and higher-order terms and 2) We will incorporate caching of sub-proofs to generate even more 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 [20], an implementation of the logical framework LF [10], 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, 6, 3, 7]. 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 which 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.

In this paper, we show how to instrument the higher-order logic programming interpreter to generate and check small proof witness by extending and

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

generalizing ideas by Rahul and Necula[14]. 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 and has also been used by Wu *et. al* [27] for creating a foundational proof checker with small witnesses. However, all these approaches restrict 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*[27] 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 [14] left of saying "more experimental results are needed especially in the higher-order setting". Our work has been implemented and evaluated within the Twelf system [20] making it unnecessary to build separate proof checking engines. To obtain a practical scalable implementation, we use higher-order substitution tree indexing [21]. Furthermore, we improve on the size of proof witnesses by caching common sub-proofs[3] .

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 [18]. In this setting types are interpreted as clauses and goals and typing context represents the store of program clauses available. We will use types and formulas interchangeably. Types and programs are defined as follows:

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

[3] Eliminating common sub-proofs is an orthogonal problem to eliminating redundant implicit type information, as is proposed in [15].

P ranges over atomic formulas such as $a \cdot S$, where a is a type constant. We interpret the function arrow $A_1 \to A_2$ 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 [13] or Isabelle[17]. 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 [10]. For more information on how to encode formal systems in LF see [19]. 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:

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

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 type 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)
      <- Πx. prov (A x)
alle: prov (A T)
      <- prov (forall λx. A x).
```

```
impi: prov (imp A B)
      <- (prov A -> prov B).
impe: prov B
      <- 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`. From a logic programming view, it might be more intuitive to think of the clause $\texttt{H <- A}_1\texttt{ <- A}_2\texttt{ <- ... <- A}_n$ as $\texttt{H <- A}_1\texttt{, A}_2\texttt{, ..., A}_n$. 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 [12]):

Select $\Gamma \vdash G \Rightarrow c_i \cdot S$
  Given an atomic goal G and clauses Γ:
  Focus on a clauses $c_i : A_i$ from Γ to establish a proof $c \cdot S$ for G, by unifying the head of A_i with the current goal G.

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 $[a/x]M$ proof for $[a/x]G$ 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 and 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 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 y`) 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` λ`y.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 [15], 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 $\mathrm{O}(\sqrt{n})$, where n is the size of the query. However, extending this idea to full LF has been difficult [26]. Maybe more importantly, proofs in LF_i are still several times as big as the overall program they certify.

Our goal is to reduce the proof evidence to the non-deterministic choices we have made during the proof to produce small proof witnesses. 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 needed to generate a compact proof witness in form of a bit-string using the verifier's proof search procedure from the previous section. We assume that we already have the full proof term and we are merely interested in compressing the proof term to a small witness in form of a bit-string. This is not a restriction, since in certifying code systems the safety proof are typically generated by a compiler. The bit-string encodes the non-deterministic choices within the proof, namely picking the right clause $c_1{:}A$ from the program context Γ to establish a proof S for P, once the goal P is atomic, by unifying the head of A with the atomic goal P. Potentially, there is more than one clause whose head unifies with P, 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 what possibility was successful.

For the idea to work, generating and checking witnesses have to perform the same overall proof search. Moreover, the order in which different choices are considered must be the same. The only difference is that in proof search we would explore possibly multiple fruitless paths, backtracking 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 uses of backtracking. We modify the proof search steps presented earlier in the following way:

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 which leads to a proof $c_i \cdot S$ of G from Γ.
Focus on clauses $c_i : A_i$ from Γ to compress a proof $c \cdot S$ for G to W.

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

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. Therefore we have : Solve Goal $\Gamma \vdash W : G$, where W is a compact proof witness.

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 in practice, 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 in fact 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. This can lead to savings of time and memory. On the other hand, paranoid consumers may need more assurance than trusting our implementation (including complex algorithms such as higher-order pattern unification). In such cases, we have an option of regenerating the proof term by instrumenting the search procedure above. This amounts to decompressing the proof witness $W$to an explicit proof term M and use a different trusted type-checker to verify the expanded proof witness.

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. That is, 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 requisite 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

A proof search procedure must have a way of retrieving all clauses of the logic program which may satisfy the current goal, since such a method will dictate how many choices we are returned at any step. Most first-order logic programming interpreter use term indexing strategies such as automata driven indexing [] to efficiently retrieve all clauses whose head unifies with the current goal. However, indexing strategies for higher-order terms are difficult, since in general retrieval and often also 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 mgu does not necessarily exist. The same holds for computing the most specific generalization of two terms.

We will adopt higher-order substitution trees [21, 22] to index higher-order logic programs. Substitution tree indexing has been successfully used in a first-order setting [9] and allows the sharing of common sub-expressions via substitutions. This is unlike other non-adaptive term indexing, which only allow sharing of common term prefixes. To extend substitution tree indexing to the higher-order setting, we will concentrate on the fragment of linear higher-order patterns[23], where all existential variables must occur only once and are applied to all distinct bound variables. Linear higher-order patterns refine the notion of higher-order patterns [11], where all existential variables must be applied to a some distinct bound variables. In [21, 22], we give a formal description for computing the most specific generalization of two linear higher-order patterns, for inserting terms in the index and for retrieving a set of terms from the index s.t.

the query is an instance of the term in the index, and show correctness. The construction of a substitution tree in the higher-order setting follows the overall algorithm described in [25]. Here we will illustrate higher-order substitution trees by an example.

To describe higher-order substitution trees, it is crucial to distinguish between bound variables, existential variables and "internal" existential variables. We will assume that any existential variable is applied to all bound variables in whose context it occurs in to facility simple and efficient insertion and retrieval algorithms. A higher-order substitution tree is a node with substitution ρ whose range is Σ and every child node has a substitution ρ_i with domain Σ. In other words, any internal existential variable in Σ will be defined at a later node in a path in the substitution tree and for every path from the top node ρ_0 with range Σ_0 to the leaf node ρ_n we have the range of the substitution which is the composition of all the substitutions along this path, is empty. In other words, there are no internal existential variables left after we compose all the substitutions ρ_n up to ρ_0. To illustrate, consider the following clauses describing part of a conversion of formulas into prenex normalform.

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

We see that the four given clauses share a lot of structure. For example clause `e1` and `e2` "almost" agree on the second argument. Similarly the clauses `e3` and `e4`. To insert these four clauses into a substitution tree, we need to first translate them into linear higher-order patterns. Although all the terms in these clauses fall into the pattern fragment, not all of them are linear patterns. For example, in clause `e1`, the existential variable B does not depend on the bound variable x, in ($\mathtt{forall}\ \lambda\mathtt{x.}(\mathtt{imp}\ (\mathtt{A\ x})\ \mathtt{B}$). Hence, B is not a linear higher-order pattern, since it is not applied to all bound variables in whose scope it occurs. In addition, the existential variable A occurs twice. Before inserting the clause heads into a substitution tree, 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:

$$\begin{array}{l}
\mathtt{e1} : \mathtt{eq}\ (\mathtt{imp}\ (\mathtt{exists}\ \lambda\mathtt{x.A\ x})\ \mathtt{B})\ (\mathtt{forall}\ \lambda\mathtt{x.}(\mathtt{imp}\ (\mathtt{A'\ x})\ (\mathtt{B'\ x}))). \\
\qquad \forall x.(A'\ x)\ \doteq\ (A\ x)\ \mathsf{and}\ B'\ x\ \doteq\ B \\[4pt]
\mathtt{e2} : \mathtt{eq}\ (\mathtt{imp}\ \mathtt{A}\ (\mathtt{forall}\ \lambda\mathtt{x.B\ x}))\quad (\mathtt{forall}\ \lambda\mathtt{x.}(\mathtt{imp}\ (\mathtt{A'\ x})\ (\mathtt{B'\ x}))). \\
\qquad \forall x.(A'\ x)\ \doteq\ A\ \mathsf{and}\ B'\ x\ \doteq\ (B\ x) \\[4pt]
\mathtt{e3} : \mathtt{eq}\ (\mathtt{and}\ \mathtt{A}\ (\mathtt{forall}\ \lambda\mathtt{x.B\ x}))\quad (\mathtt{forall}\ \lambda\mathtt{x.and}\ (\mathtt{A'\ x})\ (\mathtt{B'x})). \\
\qquad \forall x.(A'\ x)\ \doteq\ A\ \mathsf{and}\ B'\ x\ \doteq\ (B\ x)
\end{array}$$

Now even more sharing becomes apparent. For example, the clauses `e1` and `e2` agree upon the last argument. Similarly, the clauses `e3` and `e4`. 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 [22, 21].

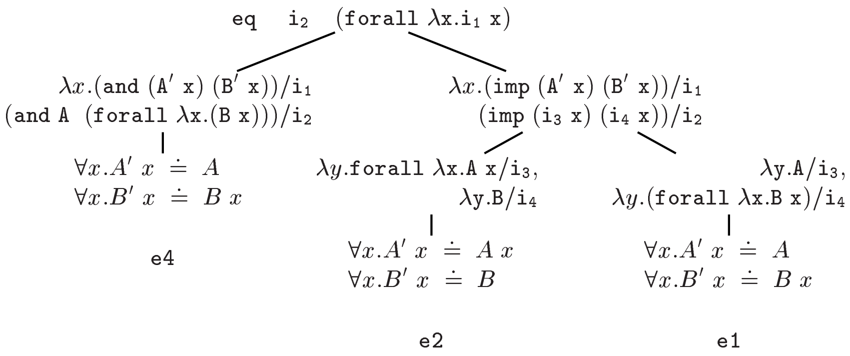

Fig. 1. Substitution tree

By composing the substitutions along a path, we will obtain a clause head. By composing the substitutions in the left-most branch, we obtain the clause head e3. In contrast to other indexing techniques such as discrimination tries, substitution trees allows the sharing of common sub-expressions instead of common term prefixes. As we can see in this example, this is especially useful 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, it is possible to use substitution tree indexing for dynamic clauses generated during proof search. However, it is not clear how useful this will be, since the process of creating the tree itself is time-consuming. It is also noted by Necula and Rahul [14] that indexing dynamic assumptions imposes a performance penalty. It is useful to pre-process the program, but the payoff with dynamic clauses is unclear. For dynamic clauses, we use only simple indexing on the type family of type.

4.2 Caching results

Since large proofs often have identical subproofs, there is a lot of potential for sharing subproofs. The problem is particularly acute in machine-generated proofs for certifying machine-code which tend to have repeated proofs of simple facts. This problem has been already pointed out by Necula and Lee in [16]

> "...it is very common for the proofs to have repeated sub-proofs that should be hoisted out and proved only once ..." [16]

In the context of generating and checking small proof witnesses, this leads to two problems. First, the proof witnesses become larger in size than necessary. This means that what has to be transmitted to the verifier is large in size. Secondly, the performance of witness checker may degrade, since it spends its time uselessly proving the same fact over and over again. Ideally we would like to cache intermediate results and re-use them later. In this section, we describe an extension to generating and checking proof witnesses with caching. The idea can be briefly summarized as follows: when generating compressed witnesses for a goal G from explicit proof terms, we store intermediate goals together with their compressed witness in a table, and re-use the result later on. For checking proof witnesses ....

Since caching everything is too overwhelming in most cases in practice, hence we support selective caching. The user declares certain predicates to be cached, and for those we will factor out common sub-proofs. We will modify the Select step in our previous search procedure to allow for caching by defining an auxiliary procedure $\mathsf{CallCheck}$. $\mathsf{CallCheck}(\Gamma \vdash G, \mathcal{T})$ checks whether a variant of the current subgoal exists or if the current subgoal is an instance of a previous 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 $\Gamma' \vdash G'$, then a pointer to the corresponding answer list is returned. If no such entry exists, $\Gamma \vdash G$ is added to the table $\mathcal{T}$ and a pointer to an empty answer list is returned. If no entry exists, then we will continue to focus on a clause c_i 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 the its answer list ptr.

If a table entry for $\Gamma \vdash G$ already exists, there are two possible situations: 1) If the answer list contains an answer substitution θ_k which leads to a proof $c_i \cdot S$, then we will just re-use the answer substitution θ_k. 2) If the answer list does not contain an answer which would lead to a 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 witness generation using caching can be summarized as follows:

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

Given an atomic goal G, clauses Γ, and a table $\mathcal{T}$:
if CallCheck($\Gamma \vdash G$, $\mathcal{T}$)
then return a pointer ptr to the answer list
    if answer($c_i \cdot S$, ptr)
        then return θ_j where $(c_i \cdot S, \theta_j)$ is j-th answer in the answer list and
            continue to solve any open subgoals under θ_j
        otherwise
            Let k'' be the length of the answer list ptr and $c_i : A_i$ be the
            i-th clause which leads to a proof $c_i \cdot S$ of G from Γ; then $j = k'' + i$
            Focus on clauses $c_i : A_i$ from Γ to compress a proof $c \cdot S$ for G to W and
            add the answer substitution together with $c_i \cdot S$ to the answer list ptr
otherwise
    Let $c_j : A_j$ be the j-th clause which leads to a proof $c_i \cdot S$ of G from Γ;
    Focus on clauses $c_j : A_j$ from Γ to compress a proof $c \cdot S$ for G to W.
    Add the answer substitution together with $c_i \cdot S$ to the answer list ptr

The generation and checking of witnesses will following similar algorithms, so both have identical caches. The number of choices is also the same in both the prover and verifier. In the following, we will highlight some of the important invariants underlying our implementation. Typically intermediate goal G may contain existential variables which are realized via references and destructive updates in an implementation. This achieves that instantiations of existential variables are immediate.On the other hand the state of the existential variables may change during proof search. Hence when tabling a given subgoal, we abstract over all the existential variables and store an abstract version of the subgoal to avoid the pollution of the table entries. A similar problem arises when adding answer substitutions, since answers may not necessarily be ground. In general, the invariant about table entries and answer substitutions are:

Table entry	Answer substitution
$\Delta; \Gamma \vdash G$	$\Delta' \vdash \theta : \Delta$

Δ refers to a context describing existential variables, Γ describes the context for the bound variables and dynamic assumptions and G describes the goal we are trying to prove. The design supports naturally substitution factoring based on explicit substitutions[24]. With substitution factoring the access cost is proportional to the size of the answer substitution rather than the size of the answer itself. It guarantees that we only store the answer substitutions, and create a mechanism of returning answers to active subgoals that takes time linear in the size of the answer substitution θ rather than the size of the solved query $[\theta]G$. In other words, substitution factoring ensures that answer tables contain no information that also exists in their associated call table. Operationally, this means that the constant symbols in the subgoal need not be examined again when we insert answer substitutions. For this setup to work cleanly in the higher-order setting, it is crucial that we distinguish between existential variables in Δ and bound variables and assumptions in Γ.

To allow easy comparison of goals G with dynamic assumptions Γ modulo renaming of existential variables and bound variables, we represent terms internally using explicit substitutions [1] and de Bruijn indices. The basic underlying idea is to use a nameless representation of variables based on de Bruijn indices. The main advantage is that equality up to renaming of bound variables is reduced to syntactic equality checks, if all objects are in $\beta\eta$-normalform.

Storing large sets of intermediate goals together with their proofs can only be practical if the table operations are efficient and exploit sharing common structure and common operations. We use higher-order substitution tree indexing from the previous section to store intermediate goals G together with their dynamic assumptions Γ and the corresponding proof $c_i \cdot S$[4]. This requires however that goals are in linear form, i.e. all existential variables are fully applied. Therefore, we linearize goals $\Gamma \vdash G$ during abstraction and factor out any non-linear sub-expressions. This motivates the final table design:

Table entry	Residual Equ.	Answer substitution
$\Delta; \Gamma \vdash G$	$\Delta; \Gamma \vdash R$	$\Delta' \vdash \theta : \Delta$

where G is a linear higher-order pattern, Γ denotes the bound variables and dynamic assumptions and Δ describes the existential variables occurring in G and Γ. This linearization step can be done together with abstraction and

[4] We only store a skeleton of the proof term.

standardization over the existential variables in goal, hence only one pass through the term is required.

5 Experimental Results

In this section, we give an experimental evaluation of generating and checking compact proof witnesses. In particular, we will compare the checking times and size between explicit proof terms and compact proof witnesses. Moreover, we will discuss the trade-offs of caching subproofs.

5.1 Sequent Calculus

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.

Example	Proof Search time (s)	Proof Compression time (s)	Witness Verification time (s)
$(A \supset B) \wedge (A \supset C) \Rightarrow A \supset B \wedge C$	0.470	0.030	0.000
$(A \supset B) \vee (A \supset C) \Rightarrow A \supset B \vee C$	0.380	0.010	0.000
$A \vee C \wedge (B \supset C) \Rightarrow (A \supset B) \supset C$	1.700	0.010	0.010
$(A \supset C) \wedge (B \supset C) \Rightarrow A \vee B \supset C$	2.180	0.020	0.000
$A \wedge B \vee C \Rightarrow (A \wedge B) \vee (A \wedge C)$	0.190	0.020	0.000
$(A \wedge B) \vee (A \wedge C) \Rightarrow A \wedge B \vee C$	0.760	0.010	0.000
$A \vee (B \wedge C) \Rightarrow A \vee B \wedge A \vee C$	0.210	0.020	0.000
$A \vee B \wedge A \vee C \Rightarrow A \vee (B \wedge C)$	0.790	0.020	0.000
$\Rightarrow (A \supset B) \wedge (A \supset C) \supset A \supset B \supset C$	0.390	0.020	0.000
$\Rightarrow (A \supset C) \wedge (B \supset C) \supset A \vee B \supset C$	2.430	0.010	0.010
$\Rightarrow (A \supset B) \vee (A \supset C) \supset A \supset B \vee C$	0.480	0.010	0.000
$\Rightarrow A \wedge (B \supset \perp) \supset (A \supset B) \supset \perp$	0.370	0.020	0.000

Table 1. Sequent Calculus: Times with Caching of User-Selected Predicates

We included the time to find a proof to contrast it against the proof checking times. In these examples, the proof search procedure uses the table in a significant manner. Tabling is used to detect loops in the goals considered by the sequent calculus. The proof compression and the verification procedures provide significant time speedups, since in these procedures, we already know the proof. This is exhibited in Table 1.

Next, we turn our attention to questions of proof size. Table 2 compares the size of our proof witnesses to the size of the original proof. The original proof is measured both by number of bytes as well as the number of tokens. These figures are assuming caching is done of predicates the user has selected, and the encoding of the oracle is in terms of the unary encoding scheme described earlier.

Example	Proof Size (bytes)	Proof Size (number of tokens)	Witness Size (bytes)
$(A \supset B) \wedge (A \supset C) \Rightarrow A \supset B \wedge C$	361	43	5
$(A \supset B) \vee (A \supset C) \Rightarrow A \supset B \vee C$	292	45	5
$A \vee C \wedge (B \supset C) \Rightarrow (A \supset B) \supset C$	570	50	6
$(A \supset C) \wedge (B \supset C) \Rightarrow A \vee B \supset C$	561	56	6
$A \wedge B \vee C \Rightarrow (A \wedge B) \vee (A \wedge C)$	280	37	6
$(A \wedge B) \vee (A \wedge C) \Rightarrow A \wedge B \vee C$	437	66	7
$A \vee (B \wedge C) \Rightarrow A \vee B \wedge A \vee C$	331	50	6
$A \vee B \wedge A \vee C \Rightarrow A \vee (B \wedge C)$	428	47	6
$\Rightarrow (A \supset B) \wedge (A \supset C) \supset A \supset B \supset C$	454	44	5
$\Rightarrow (A \supset C) \wedge (B \supset C) \supset A \vee B \supset C$	792	57	6
$\Rightarrow (A \supset B) \vee (A \supset C) \supset A \supset B \vee C$	440	46	5
$\Rightarrow A \wedge (B \supset \bot) \supset (A \supset B) \supset \bot$	513	38	5

Table 2. Sequent Calculus: Sizes with Caching of User-Selected Predicates

5.2 Refinement Types as an advanced type system

Our next example is an advanced type system for a high-level call-by-value functional language. The language has functions, a fixpoint construct, booleans and bitstrings. The type system for the language has refinement types, as described in Davies and Pfenning [8]. In particular, the type of bitstrings is refined by zero and strictly positive number representations.

In Table 3 we demonstrate the time improvements that can be had if we know the proof already. We achieve a speedup of between 2 and 6 times. This figure is achieved if we are caching subgoals to get maximum compressions. As we see later, even more gains can be achieved by turning off caching. Turning the cache off however would overstate the gains, since proof search has to use tabling to find the proof. This gain comes about since we do not have to explore unproductive branches of the proof tree.

We also compare proof size to the size of the compact witness we produce in Table 4. We notice that the compact oracle is about 1 % of the size of the proof term.

Example	Proof Search Time (s) (1)	Proof Compression Time (s) (2)	Witness Verification Time (s)	Speedup (1 / 2)
mult-pos-nat	5.810	1.180	1.100	4.92
mult-3-types	8.420	1.600	1.490	5.26
mult	0.390	0.140	0.130	2.78
square	0.420	0.160	0.130	2.62
square-pos-nat	12.550	2.310	1.850	5.43
square-pos-pos	11.880	1.920	2.130	6.18

Table 3. Refinement Type System : Proof Compression Times with Caching

Example	Proof Term Size (Bytes) (1)	Proof Term Size (Tokens)	Witness Size (Bytes) (2)	Proof Size percentage (2 / 1)
mult-pos-nat	15654	1159	169	1.07 %
mult-3-types	18537	1383	211	1.13 %
mult	6074	509	47	0.77 %
square	7060	546	50	0.70 %
square-pos-nat	25303	1587	242	0.95 %
square-pos-pos	24957	1560	237	0.94 %

Table 4. Refinement Type System : Size of Witness with Caching

Next we investigate the practicality of caching subgoals. Caching is a fairly expensive operation, in terms of both time to store and lookup, as well as the extra memory required to maintain the table. In Table 5 we investigate this tradeoff. We find that using caching gives us a speed hit of between 3 and 16 times. The gains from this is that the size of the oracle is smaller for the cached version, for every experiment in this set. The gain is small but significant.

Example	Compression Time			Witness Size			Table Size
	Cached (s) (1)	Uncached (s) (2)	Slowdown (1 / 2)	Cached (bytes) (3)	Uncached (bytes) (4)	Saving (4 - 3) / 4	
mult-pos-nat	1.180	0.110	10.72	169	171	1.16 %	579
mult-3-types	1.600	0.410	3.90	211	216	2.31 %	691
mult	0.140	0.050	2.80	47	67	29.85 %	164
square	0.160	0.040	4.0	50	71	29.57 %	179
square-pos-nat	2.310	0.160	14.43	242	247	2.02 %	794
square-pos-pos	1.920	0.160	12.0	237	243	2.46 %	775

Table 5. Refinement Type System : Caching during proof compression

5.3 Foundational Proof Carrying Code

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 [5].

In Table 6 we show the gains to be achieved in terms of time performance. In the proof carrying code scenario, asking the consumer to verify our compact proof witness as opposed to the proof term 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, as we show in Table 7.

Example	Proof Search time (s) (1)	Proof Compression time (s)	Witness Verification time (s) (2)	Speedup (1 / 2)
clos	12.260	1.940	0.470	26.08
mid	10.290	1.800	0.450	22.86
inc	11.550	1.880	0.470	24.57
lint	12.840	2.530	0.700	18.34

Table 6. FPCC: Times without Caching

Example	Proof Size (bytes) (1)	Proof Size (number of tokens)	Witness Size (bytes)(2)	Gain (1 / 2)
clos	201910	16502	638	316.47
mid	398589	34250	528	754.90
inc	410600	35724	579	709.15
lint	441965	38416	703	628.68

Table 7. FPCC: Sizes without Caching

6 Related Work

The idea of compact proof witnesses which encode the non-deterministic choice in a logic programming interpreter was first proposed by [14] for a fragment of the logical framework LF, called LF_i, which excludes the use of higher-order terms and dependent types in practice. Their main goal was to design a practical method for current proof-carrying code applications which would reduce the size of proofs sent to a consumer. To achieve this goal, their approach has been very pragmatic. For example they restrict themselves in practice to first-order terms and and significantly limit the use of dependent types such that LF_i. 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 of which unsound candidates need to be weeded out by calling full 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 they use Huet's unification algorithm, which is non-deterministic itself, their proof witnesses need to record the choices made within higher-order unification. To avoid these problems in practice, their realization and their experimental evaluation does not consider terms defined via λ-abstraction. Our work extends and continues where Necula and Rahul left of by saying "more experimental results are needed especially in the higher-order setting". By using linear higher-order patterns and higher-order substitution tree indexing, the non-determinism based on Huet's unification algorithm is removed. Moreover, our implementation scales to dependent types thereby removing any restrictions imposed by Necula and Rahul. In addition, we designed an extension where sub-proofs can be cached and re-used resulting in potentially even smaller proof witnesses.

The idea of using oracles was also explored in [27]. Their primary concern was to achieve a minimal trusted proof checker. Their checker follows the path explored by Necula and Rahul and ignores higher-order terms. The main difference between the two approaches 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 or safety rules and size of proofs.

As Necula and Rahul's work, their system is not able to support higher-order abstract syntax, which means that that any variable binding constructs must be explicitly encoded. Wu *et al*[27] 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 an valuable addition to the general safety infrastructure already provided, which not only supports specification and execution of safety policies, but also 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, this will provide a comprehensive guide for future implementations of proof checkers which need not be restricted to first-order Prolog-like systems.

Finally, small proof witnesses within the logical framework LF are interesting independently of the proof-carrying code application. They serve to increase the confidence in the logic programming and theorem proving engines. As the size of the proof term can be at least 4-times as big as the query we are trying to prove, constructing and possibly storing full proof terms during logic programming execution is expensive. As an example consider tabled higher-order logic programming, where we memoize subgoals together with their answers and proofs to re-use the results later. Constructing and storing the full proof term however would impose a substantial performance penalty. Hence, small witnesses provide a cheap simple alternative and are in fact crucial to obtain a practical tabled logic programming interpreter.

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. Juan Chen, Dinghao Wu, Andrew W. Appel, and Hai Fang. A provably sound TAL for back-end optimization. pages 208–219, June 2003.
6. 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.
7. 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.
8. 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.
9. Peter Graf. *Term Indexing*. LNAI. Springer-Verlag, 1995.
10. 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.
11. 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.
12. 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.
13. 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.
14. 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.
15. 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.
16. 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.
17. Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.
18. 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.
19. Frank Pfenning. Computation and deduction, 1997.
20. Frank Pfenning and Carsten Schürmann. System description: Twelf — a metalogical 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.

21. 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.
22. Brigitte Pientka. *Tabled higher-order logic programming*. PhD thesis, Department of Computer Sciences, Carnegie Mellon University, December 2003. CMU-CS-03-185.
23. 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, July 2003.
24. I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. Warren. Efficient access mechanisms for tabled logic programs. *Journal of Logic Programming*, 38(1):31–54, Jan 1999.
25. 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.
26. Jason Reed. Redundancy Elimination for LF. In Carsten Schuermann, editor, *Fourth Workshop on Logical Frameworks and Meta-languages — LFM'04*, Cork, Ireland, 5 July 2004.
27. 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.