Efficient Representation and Validation of Proofs

George C. Necula Peter Lee

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213–3891

{necula,petel}@cs.cmu.edu

Abstract

This paper presents a logical framework derived from the Edinburgh Logical Framework (LF) [5] that can be used to obtain compact representations of proofs and efficient proof checkers. These are essential ingredients of any application that manipulates proofs as first-class objects, such as a Proof-Carrying Code [11] system, in which proofs are used to allow the easy validation of properties of safety-critical or untrusted code.

Our framework, which we call LF_i, inherits from LF the capability to encode various logics in a natural way. In addition, the LF_i framework allows proof representations without the high degree of redundancy that is characteristic of LF representations. The missing parts of LF_i proof representations can be reconstructed during proof checking by an efficient reconstruction algorithm. We also describe an algorithm that can be used to strip the unnecessary parts of an LF representation of a proof. The experimental data that we gathered in the context of a Proof-Carrying Code system shows that the savings obtained from using LF_i instead of LF can make the difference between practically useless proofs of several megabytes and manageable proofs of tens of kilobytes.

1 Introduction

The problem of automated theorem proving has received considerable attention from the scientific community, in most cases directed at improving the capabilities and practicality of the proof search procedures. Much less consideration has been given to the generation and manipulation of the resulting proofs as first-class objects.

A theorem prover that generates an explicit proof object for each successfully proved predicate enables a distrustful user to verify the validity of the subject claim by checking the proof object. This effectively eliminates the need to trust the soundness of the theorem prover at the small expense of having to trust a much simpler proof checker. The generated proofs and the proof checker are also of great software engineering benefit as they can lead to the timely discovery of soundness bugs that are introduced during development or maintenance of the theorem prover.

Another direct use of explicit proof objects is in the context of *Proof-Carrying Code* (PCC) [11], which is a technique for safe execution of untrusted code. PCC requires the code producer to attach to the code a formal proof attesting that the code meets a safety specification that is published in advance by the code receiver. The distrustful code receiver can then easily verify the code with respect to the specification by checking the validity of the attached proof. The Proof-Carrying Code approach has been shown to be useful in safety critical systems which must be extended or upgraded with code that cannot be trusted, either because of its unknown origin or because of the high risks that are involved. Some of the published case studies discuss the uses of PCC in extensible operating systems [12], for safe interaction between components written in safe languages and native code [11] and for systems based on mobile-code agents [14].

The explicit proof object that accompanies the untrusted code as part of a proof-carrying code is the key element that enables the enforcement of a wide variety of safety policies, ranging from simple memory safety and type safety to resource-usage bounds, without the performance penalties incurred by the approaches based on interpretation or run-time checking [12].

In a Proof-Carrying Code system, as well as in

This research was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software," ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.
Submitted to LICS'98.

any application involving the explicit manipulation of proofs, it is of utmost importance that the proof representation is compact and the proof checking is efficient. In this paper we present a logical framework derived from the Edinburgh Logical Framework [5], along with associated proof representation and proof checking algorithms, that have the following desirable properties:

- The framework can be used to encode judgments and derivations from a wide variety of logics, including first-order and higher-order logics.
- The proof checker performs a directed, one-pass inspection of the proof object, without having to perform search. This leads to a simple implementation of the proof that is easy to trust and to install in existing extensible systems.
- A unique implementation of the proof checker can be easily configured for all of the logics that can be encoded in the framework. This allows a code receiver to use the same implementation of the checker for enforcing many unrelated safety policies.
- Even though the proof representation is detailed, it is also compact.

We chose the Edinburgh Logical Framework (LF) as the starting point in our quest for efficient proof manipulation algorithms because it alone scores very high on the first three of the four desirable properties listed above. In this respect, this paper can also be viewed as a testimony to the usefulness of LF for practical system applications, such as Proof-Carrying Code. However, LF is not completely appropriate for our purposes because the representation of proofs are unnecessarily large due to a high degree of redundancy. To address this issue, we have extended LF to handle implicit subterms, and we also extended the LF type checking algorithm to synthesize the missing subterms. In order to keep the reconstruction algorithm simple, and in particular to avoid higher-order unification and the need to postpone constraints, we impose syntactic restrictions on which proof subterms can be missing. Because of this restrictions we might be forced, in general, to retain small amounts of redundant information in the proof representations. However, our experience shows that this is only the case for the representation of level-two proofs (proofs of theorems *about* deductive systems), which are beyond the current scope of Proof-Carrying Code. By using the resulting framework, which we call LF_i, we observe, in our experiments with PCC and with a proof-generating theorem prover for first-order logic, reductions of more than two orders of magnitude in the size of the proofs and in the time required for proof checking, with the factors increasing with the size of the proofs.

This paper is organized as follows. In Section 2 we present the classical encoding of predicates and proofs in the Edinburgh Logical Framework. Then, in Section 3 we introduce the LF_i type system. Because the LF_i type system does not lend itself directly to deterministic type checking, we show, in Section 4, an LF_i reconstruction judgment that is shown to be sound with respect to LF_i typing. We continue with the representation algorithms, which convert LF representations to LF_i representations. We conclude with a few implementation details, in Section 6, and with a presentation of the experimental data supporting the claim that LF_i representation is more efficient than the LF representation. For reasons of space, we omit many interesting details that can be found in the expanded version of this paper [13].

2 The Edinburgh Logical Framework

The Edinburgh Logical Framework (also referred to as LF) has been introduced by Harper, Honsell and Plotkin [5] as a metalanguage for high-level specification of logics. LF provides natural support for the management of binding operators and of the hypothetical and schematic judgments through LF bound variables. This is a crucial factor for the succinct formalization of proofs.

The LF type theory is a language with entities at three levels, namely objects, types and kinds, whose abstract syntax is shown below:

$$\begin{array}{llcl}
\text{Kinds} & K & ::= & \mathtt{Type} \mid \Pi x\!:\!A.K \\
\text{Types} & A & ::= & a \mid A\,M \mid \Pi x\!:\!A_1.A_2 \\
\text{Objects} & M & ::= & x \mid c \mid M_1 M_2 \mid \lambda x\!:\!A.M
\end{array}$$

The notation $A_1 \rightarrow A_2$ is sometimes used instead of $\Pi x\!:\!A_1.A_2$ if x is not free in A_2.

In order to define a logic in LF we define an LF signature Σ that contains declarations for a set of constants corresponding to the syntactic formula constructors and to the proof rules. A fragment of the signature that defines the first-order predicate logic is shown in Figure 1. The top section of the figure contains declarations of the type constructors ι and o corresponding respectively to individuals and predicates, and of the type family `pf` indexed by predicates. If P is the representation of a predicate, then "`pf` P" is the type of all valid proofs of P. The rest of Figure 1 contains the declarations of a few syntactic constructors followed by a few constants corresponding to proof rules of first-order logic. Note the use of higher-order features of LF for the succinct representation of the hypothetical and parametric

$$
\begin{array}{lcl}
\iota & : & \texttt{Type} \\
o & : & \texttt{Type} \\
\texttt{pf} & : & o \rightarrow \texttt{Type} \\
\\
\texttt{true} & : & o \\
\texttt{and} & : & o \rightarrow o \rightarrow o \\
\texttt{imp} & : & o \rightarrow o \rightarrow o \\
\texttt{all} & : & (\iota \rightarrow o) \rightarrow o \\
\\
\texttt{truei} & : & \texttt{pf true} \\
\texttt{andi} & : & \Pi P{:}o.\Pi R{:}o.\texttt{pf}\ P \rightarrow \texttt{pf}\ R \rightarrow \texttt{pf}\ (\texttt{and}\ P\ R) \\
\texttt{andel} & : & \Pi P{:}o.\Pi R{:}o.\texttt{pf}\ (\texttt{and}\ P\ \ R) \rightarrow \texttt{pf}\ P \\
\texttt{impi} & : & \Pi P{:}o.\Pi R{:}o.(\texttt{pf}\ P \rightarrow \texttt{pf}\ R) \rightarrow \texttt{pf}\ (\texttt{imp}\ P\ R) \\
\texttt{alli} & : & \Pi P{:}\iota \rightarrow o.(\Pi X{:}\iota.\texttt{pf}\ (P\ X)) \rightarrow \texttt{pf}\ (\texttt{all}\ \ P) \\
\texttt{alle} & : & \Pi P{:}\iota \rightarrow o.\Pi E{:}\iota.\texttt{pf}\ (\texttt{all}\ \ P) \rightarrow \texttt{pf}\ (P\ E)
\end{array}
$$

Figure 1: Fragment of the LF signature corresponding to the first-order predicate logic.

judgments that characterize the implication elimination (**impi**), the universal quantifier introduction (**alli**) and elimination (**alle**) rules.

We write $\ulcorner P \urcorner$ to denote the LF representation of the predicate P. As an example of an LF representation of a proof we show in Figure 2 the representation of a proof of the predicate $P \supset (P \wedge P)$, for some predicate P.

$$
\begin{array}{rcl}
M & = & \texttt{impi}\ulcorner P \urcorner (\texttt{and}\ulcorner P \urcorner \ulcorner P \urcorner) \\
 & & \qquad (\lambda x{:}\texttt{pf}\ulcorner P \urcorner.\texttt{andi}\ulcorner P \urcorner \ulcorner P \urcorner\ x\ x)
\end{array}
$$

Figure 2: The LF representation of the proof by implication introduction followed by conjunction introduction of the predicate $P \supset (P \wedge P)$.

An important benefit of using LF for representing proofs is that we can use LF type checking to verify that a proof is valid and that it proves a given predicate. The LF type-checking judgment is written as $\Gamma \overset{LF}{\vdash} M : A$, where Γ is a type environment for the free variables of M and A.[1] A definition of LF type checking, along with a proof of adequacy of using LF type checking for proof validation for first-order and higher-order logics, can be found in [5]. For example, the validation of the proof representation M shown in Figure 2 can be done by finding a derivation of the following judgment:

$$\cdot \overset{LF}{\vdash} M : \texttt{pf}\ (\texttt{imp}\ulcorner P \urcorner (\texttt{and}\ulcorner P \urcorner \ulcorner P \urcorner))$$

[1] The LF typing judgment and all the other typing judgments discussed in this paper depend also on a signature Σ, which is henceforth omitted in order to simplify the notation.

Owing to the simplicity of LF and of the LF type system, the implementation of the type checker is simple and easy to trust. Furthermore, because all of the dependencies on the particular object logic are segregated in the signature, the implementation of the type checker can be reused directly for proof checking in various first-order or higher-order logics. The only logic-dependent component of the proof-checker is the signature, which is usually easy to verify by visual inspection.

Unfortunately, the above-mentioned advantages of LF representation of proofs come at a high price. The typical LF representation of a proof is large, due to a significant amount of redundancy. This fact can already be seen in the proof representation shown in Figure 2, where there are six copies of P, as opposed to only three in the predicate to be proved. The effect of redundancy observed in practice increases non-linearly with the size of the proofs. Consider for example, the representation of the proof of the n-way conjunction $P \wedge \ldots \wedge P$. This representation contains about n^2 redundant copies of $\ulcorner P \urcorner$ and n occurrences of **andi**. The redundancy of representation is not only a space problem but also leads to inefficient proof checking, because all of the redundant copies have to be type checked and then checked for equivalence with the copies of P from the predicate.

The proof representation and checking framework presented in the remainder of this paper is based on the observation that it is possible to retain only the skeleton of an LF representation of a proof and use a modified LF type-checking algorithm to reconstruct on the fly the missing parts. The resulting *implicit LF* representation inherits the advantages of the LF representation (i.e., small and logic-independent implementation of the proof checker) without the disadvantages (i.e., large proof sizes and slow proof checking).

3 Implicit LF

The implicit LF representation of a proof is similar to the corresponding LF representation, with the exception that select parts of the representation are missing. For expository purposes, the positions in the representation where subterms are missing are marked with placeholders, written as $*$. To exemplify the use of placeholders we show in Figure 4 an LF_i representation of the proof shown in Figure 2.

The major difficulty in dealing with placeholders is in type checking. In particular, the type checking algorithm must be able to reconstruct on the fly the missing parts of a proof. That is why we refer to the LF_i type checking algorithm as the *reconstruction algorithm.*

The first step in establishing a reconstruction algorithm for LF_i is to define the LF_i type system, through

Objects :

$$\frac{\Sigma(c) = A}{\Gamma \overset{i}{\vdash} c : A} \quad \frac{\Gamma(x) = A}{\Gamma \overset{i}{\vdash} x : A} \quad \frac{\Gamma \overset{i}{\vdash} M : A \quad A \equiv_\beta B \quad \mathrm{PF}(A)}{\Gamma \overset{i}{\vdash} M : B}$$

$$\frac{\Gamma, x : A \overset{i}{\vdash} M : B}{\Gamma \overset{i}{\vdash} \lambda x{:}{*}.M : \Pi x{:}A.B} \quad \frac{\Gamma, x : A \overset{i}{\vdash} M : B}{\Gamma \overset{i}{\vdash} \lambda x{:}A.M : \Pi x{:}A.B}$$

$$\frac{\Gamma \overset{i}{\vdash} M : \Pi x{:}A.B \qquad \Gamma \overset{i}{\vdash} N : A \qquad \mathrm{PF}(A)}{\Gamma \overset{i}{\vdash} M\ N : [N/x]B}$$

$$\frac{\Gamma \overset{i}{\vdash} M : \Pi x{:}A.B \qquad \Gamma \overset{i}{\vdash} N : A \qquad \mathrm{PF}(A)}{\Gamma \overset{i}{\vdash} M\ * : [N/x]B}$$

Equivalence :

$$\frac{}{(\lambda x{:}A.M)N \equiv_\beta [N/x]M} \quad \frac{}{(\lambda x{:}{*}.M)N \equiv_\beta [N/x]M}$$

Figure 3: The LF_i type-system

$$\texttt{impi}\ *_1\ *_2\ (\lambda x{:}*_3.\texttt{andi}\ *_4\ *_5\ x\ x)$$

Figure 4: The LF_i representation of a proof of $P \supset P \wedge P$

a typing judgment $\Gamma \overset{i}{\vdash} M : A$, whose definition is shown in Figure 3. The LF_i typing rules are very similar to the LF typing rules, the only differences being the ability to type implicitly-typed abstractions and applications whose argument is implicit. The notation $\mathrm{PF}(A)$ means that A is placeholder-free (i.e., it does not contain placeholders). Similarly, we write $\mathrm{PF}(\Gamma)$ to denote that the the types contained in the type environment Γ do not contain placeholders. We have decided to restrict the LF_i typing to situations where the types involved do not contain placeholders. This does not seem to affect the representation of proofs, but it simplifies the various proofs of correctness of LF_i.

The LF_i typing rules cannot be used as the basis for an implementation of reconstruction, because the choice of the instantiation of placeholders is not completely determined by the context. Nevertheless, the LF_i typing system is an adequate basis for arguing the soundness of a reconstruction algorithm. This property is established through Theorem 3.1 that guarantees the existence of a well-typed placeholder-free object M' if the implicit object M can be typed in the LF_i typing discipline.

Theorem 3.1 (Soundness of LF_i typing.) *If $\Gamma \overset{i}{\vdash} M : A$ and $\mathrm{PF}(\Gamma)$, $\mathrm{PF}(A)$, then there exists M' such that $\Gamma \overset{LF}{\vdash} M' : A$.*

The proof of Theorem 3.1 is by induction on the structure of the LF_i typing derivation. This theorem is the keystone in the proof of adequacy of proof checking by using reconstruction algorithms that are sound with respect to LF_i.

4 The Reconstruction Algorithm

The reconstruction algorithm performs the same functions as the LF type-checking algorithm, except that it also finds well-typed instantiations for placeholders that occur in objects. To ease the understanding of the algorithm we first sketch its operation on the implicit proof object of Figure 4.

The reconstruction goal in this case is to verify that the implicit proof representation of Figure 4 has type

$$\texttt{pf}\ (\texttt{imp}\ \ulcorner P \urcorner\ (\texttt{and}\ \ulcorner P \urcorner\ \ulcorner P \urcorner))$$

Based on this goal type and on the declared type of the constant `impi` (the head of the top-level application), the algorithm collects the following constraints:

$$\begin{array}{rll} & *_1 & \equiv\ \ulcorner P \urcorner \\ & *_2 & \equiv\ \texttt{and}\ \ulcorner P \urcorner\ \ulcorner P \urcorner \\ \vdash & (\lambda x{:}*_3.\texttt{andi}\ *_4\ *_5\ x\ x) & \\ & & :\ \texttt{pf}\ \ulcorner P \urcorner \to \texttt{pf}\ (\texttt{and}\ \ulcorner P \urcorner\ \ulcorner P \urcorner) \end{array}$$

The first two constraints lead to substitutes for $*_1$ and $*_2$, which are guaranteed by construction to be well-typed representation of predicates. From the last constraint, using the rule for abstraction we obtain the following system of constraints:

$$\begin{array}{rcl} & *_3 & \equiv\ \texttt{pf}\ \ulcorner P \urcorner \\ x : \texttt{pf}\ \ulcorner P \urcorner \vdash & \texttt{andi}\ *_4\ *_5\ x\ x & :\ \texttt{pf}\ (\texttt{and}\ \ulcorner P \urcorner\ \ulcorner P \urcorner) \end{array}$$

The process continues until no more constraints are generated. Note that each system of constraints consists of typing constraints and unification constraints, which in turn are of the simple rigid-rigid or flex-rigid kinds that can be solved eagerly.

We give a precise description of the reconstruction algorithm in the form of five mutually recursive judgments, that are directly implementable. For this purpose we first introduce some notation. We use the letter u to range over members of a set of placeholder variables that are subject to unification (as opposed to the LF variables that are not). The symbol Δ is used to denote a type environment for placeholder variables only, while Γ is used for arbitrary type environments. In the case when an LF object M does not contain placeholder variables we write $\mathrm{PVF}(M)$. We extend this notation and we write $\mathrm{PVF}(\Gamma(\mathrm{FV}(M)))$ to mean that the types associated by Γ to the free variables of M do not contain placeholder variables. As in the LF_i typing judgments, types do not contain placeholders but they might contain placeholder variables.

The main operation on placeholder variables is substitution with LF objects. We write $\Psi(M)$ to denote the result of applying the substitution Ψ to the placeholder variables of M. We also write $Dom(\Psi)$ to refer to the set of placeholder variables on which Ψ is defined. Finally, we write $\Psi|_S$ to denote the substitution obtained from Ψ by restricting it to the set of placeholder variables S.

A list of constraints C consists of several typing constraints of the form $M : A$ and one unification constraint of the form $A \approx_a B$.

The definitions of the judgments that describe the reconstruction algorithm are shown in Figure 5. The notation for each judgment uses the double arrow to separate the inputs (on the left) and the outputs (on the right) of a procedure that implements the algorithmic interpretation of each judgment. In the rest of this section we briefly discuss each judgment.

Reconstruction: $\Gamma \overset{r}{\vdash} M : A \Rightarrow \Psi$**.** In this judgment the object M can contain placeholders but not placeholder variables, and the type A and the type environment cannot contain placeholders. This is the main judgment that checks that M is an implicit object of type A and also returns a substitution for placeholder variables of A.

If M is an abstraction then the implicit type of the bound variable is recovered from the dependent function type (rule **abs**). Note that in this case we require that the reconstruction of the abstraction body does not yield a substitution. This restriction simplifies the unification process as we do not have to worry about the substitution containing the bound variable.

If M is not an abstraction (rule **atom**) then we first collect a set of constraints C and compute B, the type of M, possibly containing a set of placeholder variables declared in Δ. Then we solve the constraints and return the resulting substitution Ψ.

The condition $Dom(\Delta) \subseteq Dom(\Psi)$ from the **atom** rule requires that all placeholder variables introduced during constraint collection have been instantiated during local constraint solving. In this respect, our algorithm is less powerful than that used by Elf [16], which can postpone unsolved constraints. This restriction does not seem to limit the power of our algorithm for reconstructing implicit proof representations while eliminating the need for the machinery for managing postponed constraints.

Note also that the reconstruction algorithm does not check explicitly that the returned substitution is well-typed because this is guaranteed to be true by the design of constraint solving.

Constraint Collection: $\Gamma \overset{r}{\vdash} M \Rightarrow (\Delta \; ; C \; ; B)$**.** This judgment computes, for objects M that are not abstractions, a set of constraints C and a type B, such that if C has a solution then M has type B. The environment Δ declares the placeholder variables that are introduced during constraint collection. These placeholder variables might appear in C and B.

The object M is first scanned to find the application head, whose type is read from signature (rule **cc_ct**) or from the type environment (rule **cc_var**)[2]. Then for each explicit application argument a typing constraint is added (rule **cc_app**) and for each implicit argument a new placeholder variable is introduced in the environment Δ (rule **cc_appi**). Note that typing constraints are only introduced for the explicit arguments because the implicit arguments are guaranteed to be reconstructed with objects of the appropriate type. The side conditions in the rule **cc_app** ensure that types never contain placeholders.

Constraint Solving: $\Gamma \overset{r}{\vdash} C \Rightarrow \Psi$**.** This judgment is described by one rule for each kind of constraint (rules **sc_typ** and **sc_unif**) and a reordering rule that allows an implementation to pick any desired order of solving the constraints. Our implementation does not employ the **sc_ord** rule, thus solving the constraints in last-in-first-out order.

Unification: $M \approx_a M' \Rightarrow \Psi$ **and** $M \approx M' \Rightarrow \Psi$**.** The objects or types being unified can contain placeholder variables but not placeholders. The result of unification is a substitution of objects for placeholder variables. In order to avoid the complications of higher-order unification we never unify a placeholder variable that is the head of an application. This restriction does not seem to have major effects on the usability of the reconstruction for first-order logic proofs. In order to express this restriction we distinguish between unification of an application head (atomic unification) and the other cases (normal unification). Only normal unification can produce a substitution for a placeholder variable. The costly "occurs check" of rule **nu_inst** can be avoided in many instances (see Section 6). Note that the rule **nu_abs** is sufficient for unifying abstractions because objects and types are kept in η-long normal form.

[2] The judgments described here only allow proofs in β-normal form, as they are most often produced by provers. The system can be easily extended to deal with the β-redices, which are useful for encoding proofs using unnamed lemmas.

Main reconstruction:

$$\frac{\Gamma, x : A \overset{r}{\vdash} M : B \Rightarrow \cdot}{\Gamma \overset{r}{\vdash} \lambda x.M : \Pi x{:}A.B \Rightarrow \cdot}\textbf{abs}$$

$$\frac{\Gamma \overset{r}{\vdash} M \Rightarrow (\Delta\ ;\ C\ ;\ B) \qquad \Gamma, \Delta \overset{r}{\vdash} C, A \approx_a B \Rightarrow \Psi \qquad Dom(\Delta) \subseteq Dom(\Psi)}{\Gamma \overset{r}{\vdash} M : A \Rightarrow \Psi\big|_{Dom(\Gamma)}}\textbf{atom} \qquad M \text{ is not an abstraction}$$

Collecting constraints:

$$\frac{}{\Gamma \overset{r}{\vdash} c \Rightarrow (\cdot\ ;\ \cdot\ ;\ \Sigma(c))}\textbf{cc\_ct} \qquad \frac{}{\Gamma \overset{r}{\vdash} x \Rightarrow (\cdot\ ;\ \cdot\ ;\ \Gamma(x))}\textbf{cc\_var}$$

$$\frac{\Gamma \overset{r}{\vdash} M \Rightarrow (\Delta\ ;\ C\ ;\ \Pi x{:}A.B) \qquad x \in \mathrm{FV}(B) \supset (\mathrm{PF}(N) \textit{ and } \mathrm{PVF}(A) \textit{ and } \mathrm{PVF}(\Gamma(\mathrm{FV}(N))))}{\Gamma \overset{r}{\vdash} M\ N \Rightarrow (\Delta\ ;\ C, N : A\ ;\ [N/x]B)}\textbf{cc\_app}$$

$$\frac{\Gamma \overset{r}{\vdash} M \Rightarrow (\Delta\ ;\ C\ ;\ \Pi x{:}A.B)}{\Gamma \overset{r}{\vdash} M * \Rightarrow (\Delta, u : A\ ;\ C\ ;\ [u/x]B)}\textbf{cc\_appi} \qquad u \text{ is a new placeholder variable}$$

Solving constraints:

$$\frac{}{\Gamma \overset{r}{\vdash} \cdot \Rightarrow \cdot}\textbf{sc\_0} \qquad \frac{\Gamma \overset{r}{\vdash} C_1, C_2, C_3 \Rightarrow \Psi}{\Gamma \overset{r}{\vdash} C_2, C_1, C_3 \Rightarrow \Psi}\textbf{sc\_ord}$$

$$\frac{\Gamma \overset{r}{\vdash} M : A \Rightarrow \Psi \qquad \Psi(\Gamma) \overset{r}{\vdash} \Psi(C) \Rightarrow \Psi'}{\Gamma \overset{r}{\vdash} C, M : A \Rightarrow \Psi' \circ \Psi}\textbf{sc\_typ} \qquad \frac{A \approx_a B \Rightarrow \Psi \qquad \Psi(\Gamma) \overset{r}{\vdash} \Psi(C) \Rightarrow \Psi'}{\Gamma \overset{r}{\vdash} C, A \approx_a B \Rightarrow \Psi' \circ \Psi}\textbf{sc\_unif}$$

Atomic Unification:

$$\frac{}{c \approx_a c \Rightarrow \cdot}\textbf{au\_ct} \qquad \frac{}{x \approx_a x \Rightarrow \cdot}\textbf{au\_var}$$

$$\frac{M \approx_a M' \Rightarrow \Psi \qquad \Psi(N) \approx \Psi(N') \Rightarrow \Psi'}{M\ N \approx_a M'\ N' \Rightarrow \Psi' \circ \Psi}\textbf{au\_app} \qquad \frac{[N_n/x_n] \ldots [N_1/x_1]M \approx_a M' \Rightarrow \Psi}{(\lambda x_1. \ldots \lambda x_n.M)\ N_1 \ldots N_n \approx_a M' \Rightarrow \Psi}\textbf{au\_beta}$$

Normal Unification:

$$\frac{M \approx M' \Rightarrow \cdot}{\lambda x.M \approx \lambda x.M' \Rightarrow \cdot}\textbf{nu\_abs} \qquad \frac{M \approx_a M' \Rightarrow \Psi}{M \approx M' \Rightarrow \Psi}\textbf{nu\_atom} \qquad \frac{u \notin \mathrm{FV}(M)}{u \approx M \Rightarrow u \leftarrow M}\textbf{nu\_inst}$$

Figure 5: The judgments that constitute the LF_i reconstruction algorithm

This concludes the presentation of the reconstruction algorithm. It is interesting to note that if the object to be reconstructed does not contain placeholders (it is an LF object) then the reconstruction algorithm does the same amount of work as the LF type checking. For a more detailed discussion, including the complete proofs of soundness, we refer the interested reader to [13]. We only mention here the main soundness result (Theorem 4.1) which states that the existence of a derivation of the reconstruction judgment (or equivalently, a successful run of the reconstruction algorithm) guarantees the existence of an LF_i typing, which in turn (Theorem 3.1) guarantees the existence of an LF typing. This sequence of theorems allows us to use the reconstruction algorithm for checking compact representation of proofs, while preserving the adequacy results known for the Edinburgh Logical Framework.

Theorem 4.1 (Soundness of proof reconstruction.) *If M and A are an LF_i object and an LF_i type respectively such that* $\mathrm{PVF}(M)$, $\mathrm{PF}(A)$, $\cdot \overset{i}{\vdash} A : \texttt{Type}$ *and* $\cdot \overset{r}{\vdash} M : A \Rightarrow \cdot$ *then* $\cdot \overset{i}{\vdash} M : A$.

5 The Representation Algorithm

The reconstruction judgment presented in Section 4 does not accept all implicit forms of a well-typed LF object. It is therefore useful to define an algorithm that can be used to erase from an LF representation of a proof as many redundant subterms as possible, while ensuring that the resulting implicit object is accepted by the reconstruction. In a Proof-Carrying Code environment, such a representation algorithm is applied to the proof before it is sent to the receiving system in order to reduce the burden on the communication network.

We have designed and implemented three representation algorithms with varying degrees of complexity and effectiveness. For space reasons, we present only one that we have determined experimentally to be a good choice in practice.

Our representation algorithm of choice does not attempt to find the maximum number of redundant subterms in a proof representation, but instead it statically defines statically *representation recipes* for each constant of functional type defined in the signature. These representation recipes specify which arguments of an application of the constant can be safely eliminated. For example, the representation recipe for the constant **andi** of Figure 1 says that the first two arguments can be left implicit. To alleviate the inherent loss of effectiveness due to a context-insensitive representation algorithm, we actually use two separate representation recipes for each constant and we dynamically pick the appropriate one based on the context.

A representation recipe for an n-ary constant c is composed of a sequence containing n recipe characters, with each character corresponding to one argument position in an application of c. We denote the two recipes corresponding to the constant c by $R^c(c)$ (the "checking" recipe) and $R^i(c)$ (the "inference recipe"). The algorithm described here uses three recipe characters, $*$ to denote an implicit argument, and e_c and e_i to denote explicit arguments for which the checking and inference recipes, respectively, must be used recursively. To refer to both the checking and inference modes at the same time we shall use the letter m to stand for either c or i. There are two components of the representation algorithm. We describe first the actual representation algorithm that applies the representation recipes, and then we show how we compute the recipes.

The main representation function is described in Figure 6 by the judgments $M \stackrel{m}{\to} M'$ (compute M', which is the representation of M in mode m, with $m \in \{c, i\}$) and $M \stackrel{m}{\to} M' + R$ (compute the representation of the application head M and the recipe for the remaining arguments R). In order to represent an application of a constant c we first get the appropriate recipe for c (depending on the kind of representation). Then we drop the arguments corresponding to $*$ recipe characters and we represent recursively the arguments corresponding to e_m characters, using the mode specified by m. Abstractions are always implicitly typed and the application of a variable is represented with all the arguments explicit.

The reader can verify that applying the $\stackrel{c}{\to}$ representation function with the recipes shown in Figure 7 to the LF proof representation of Figure 2 we obtain the implicit representation of Figure 4.

The core of the representation algorithm is the computation of the representation recipes for each constant based on its type as declared by the signature Σ. Informally, the representation recipes for c predict the behavior of the reconstruction algorithm when analyzing an application whose head is c, a situation that is

$$\frac{M \stackrel{m}{\to} M'}{\lambda x\!:\!A.M \stackrel{m}{\to} \lambda x\!:\!*.M'} \qquad \frac{M \stackrel{m}{\to} M' + \cdot}{M \stackrel{m}{\to} M'}$$

$$\frac{}{c \stackrel{m}{\to} c + R^m(c)} \qquad \frac{}{x \stackrel{m}{\to} x + e_c \ldots e_c}$$

$$\frac{M_1 \stackrel{m}{\to} M_1' + *R}{M_1\ M_2 \stackrel{m}{\to} M_1' * + R} \qquad \frac{M_1 \stackrel{m}{\to} M_1' + e_m R \qquad M_2 \stackrel{m}{\to} M_2'}{M_1\ M_2 \stackrel{m}{\to} M_1'\ M_2' + R}$$

Figure 6: The representation algorithm.

described by the judgment below:

$$\Gamma \stackrel{r}{\vdash} cM_1 \ldots M_n : A \Rightarrow \Psi$$

Two situations are distinguished. The "checking" situation is when A does not contain placeholder variables, and can therefore be used to recover some implicit arguments of c. This situation occurs for example at the top level because we know the predicate that the proof is supposed to prove.

The other situation is when the type of the application might contain placeholder variables, in which case we conservatively assume that no implicit arguments can be recovered from it. The latter situation is referred to as "inference". In each situation we can predict, based on the type of the constant c and because the typing constraints are solved from right to left, which arguments can be implicit and also, for each explicit argument, whether a checking or an inference reconstruction will occur.

In order to describe precisely the computation of the representation recipes we first define the function $\mathit{Unif}(M)$ which computes the set of free variables of M that are guaranteed to be instantiated during a successful unification with another object M'. The definition of *Unif* follows that of the unification. The complement of $\mathit{Unif}(M)$ is denoted by $\mathit{NUnif}(M)$, which normally consists of those variables that occur in functor position in M. A few notable cases in the definition of *Unif* are shown below:

$$\frac{}{\mathit{Unif}(\lambda x\!:\!A.M) = \emptyset} \qquad \frac{}{\mathit{Unif}(x) = \{x\}}$$

$$\frac{\text{The head of } M_1 \text{ is a constant}}{\mathit{Unif}(M_1\ M_2) = \mathit{Unif}(M_1) \cup \mathit{Unif}(M_2)}$$

$$\frac{\text{The head of } M_1 \text{ is a variable}}{\mathit{Unif}(M_1\ M_2) = \emptyset}$$

Consider an application of the constant c whose type is $\Sigma(c) = \Pi x_1 : A_1 \ldots . \Pi x_n : A_n . A_{n+1}$ such that A_{n+1} is not a type abstraction. Due to the right-to-left order of solving constraints, the first operation is to unify

A_{n+1} with a given type B. For this unification to succeed it is necessary that the arguments corresponding to variables in $NUnif(A_{n+1})$ are explicit. If we are in a checking situation (i.e., B does not contain placeholder variables) then the unification finds instantiations for the arguments corresponding to variables in $Unif(A_{n+1})$. If we are in an inference situation then the unification is not guaranteed to find any instantiation. Next, the typing constraints for the arguments of c are processed, in reverse order. Let I_k^m, with $m \in \{c, i\}$ and $k \in 1, \ldots, n+1$, be the set of the variables among $\{x_1, \ldots, x_n\}$ that are guaranteed to be instantiated after processing the constraint corresponding to A_k. Similarly, let E_k^m be the set of those variables that are required to be explicit so that unification does not fail when processing the constraints corresponding to $A_k, \ldots, A_{n+1}$. These sets are computed starting with $k = n+1$ as follows:

$$
\begin{array}{lcl}
I_{n+1}^c = Unif(A_{n+1}) & I_{n+1}^i = \emptyset & E_{n+1}^m = NUnif(A_{n+1}) \\
I_k^m & = & I_{k+1}^m \cup (Unif(A_k) \setminus E_{k+1}^m) \\
E_k^m & = & E_{k+1}^m \cup (NUnif(A_k) \setminus I_{k+1}^m)
\end{array}
$$

With these definitions we can define the representation recipe as $R^m(c) = r_1^m \ldots r_n^m$, where the recipe characters are defined as follows:

$$
r_k^m = \begin{cases} * & \text{if } x_k \in I_{k+1}^m \\ e_c & \text{if } \mathrm{FV}(A_k) \subseteq I_{k+1}^m \cup E_1^m \\ e_i & \text{otherwise} \end{cases}
$$

Note that an argument corresponding to x_k can be implicit if it can be inferred from the constraints corresponding to $x_{k+1}, \ldots x_n$, and if x_k does not appear in a functor position in any of the $A_{k+1}, \ldots, A_{n+1}$. (This is because $I_k^m \cap E_k^m = \emptyset$.) Otherwise, the argument must be explicit and is represented recursively in checking mode only if all the variables occurring in its type are either required to be explicit (E_1^m) or can be inferred from previous constraints (I_{k+1}^m).

The reader can verify that this algorithm for computing representation recipes generates the results shown in Figure 7. The interested reader can find in [13] two more representation algorithms, one simpler and one more complex. All these representation algorithms yield implicit objects that are suitable for the reconstruction algorithm discussed in Section 4. We do not show here proofs of correctness of the representation algorithm. In any case, the correctness of the representation algorithm is not as critical as the one for the reconstruction, because each result of the representation algorithm can be tested using the reconstruction.

Constant	R^c	R^i
`and`	$e_c e_c$	$e_c e_c$
`andi`	$* * e_c e_c$	$* * e_i e_i$
`andel`	$* * e_i$	$* * e_i$
`impi`	$* * e_c$	$e_c * e_i$
`alle`	$e_c e_c e_c$	$e_c e_c e_c$

Figure 7: A fragment of the representation recipes for first-order logic.

6 Implementation Details

In this section we discuss some aspects of our actual implementation of the proof representation and proof checking algorithms, specifically the binary encoding of LF_i objects and an "occurs-check" optimization that leads to significant improvements in the reconstruction time. A more detailed presentation of the implementation can be found in [13].

Our implementation of LF_i reconstruction is based on explicit substitutions [1] and deBruijn indices [2]. This choice simplifies considerably the implementation of substitution and unification.

[*The rest of this section is missing from the extended abstract due to space constraints.*]

7 Experimental Results

The representation algorithm presented in Section 5 and the reconstruction algorithm shown in Section 4 have been implemented as part of our Proof-Carrying Code system. In this system, we use a first-order theorem prover to prove the memory safety and type safety of various mobile agents written in assembly language. The proofs emitted by the prover are converted to implicit form using the representation algorithm and then sent to the code receiver, where they are checked using the reconstruction algorithm.

In this section we present the results of our experimental comparisons between the LF and the LF_i representation of proofs, in terms of proof size and proof checking time. For this purpose we use the length of the external representation of proofs as the proof size. For both the LF and the LF_i representations we measure the time required for proof reconstruction, as the average of 100 runs of the reconstruction algorithm on a Pentium 300MHz machine. We consider the reconstruction time for the LF representation of the proof to be indicative of the LF type-checking time.

We show in Figure 8 a representative sample of the experimental data that we collected. We note that the savings in proof size and proof-checking time due the LF_i representation increase nonlinearly with the size of

Experiment	Proof size		Checking time	
	LF	LFI	LF	LFI
unpack	$> 10 \cdot 10^6$	23728	8256	42
simplex	$> 2 \cdot 10^6$	23888	1656	42
sharpen	183444	4816	136	7
qsort	92412	3098	74	6
kmp	77246	2092	60	3
bcopy	12466	796	11	1

Figure 8: The effect of LF_i representation on the proof size and proof-checking time. The proof sizes are measured in bytes on the external representation, and the checking time is measured in milliseconds on a Pentium 300MHz.

the proof. The space savings are so significant that, for the larger proofs, they make the difference between practically-useless proofs of several megabytes and manageable proofs of a few tens of kilobytes. A similar reduction of two orders of magnitude can be observed for proof-checking time.

In order to better understand the dependency of the LF_i proof size and checking time on the corresponding LF values, we have fitted the polynomial $y = ax^b$ to the experimental data, using the least-squares method. The fitting results (having a correlation of about 97%) are shown below:

$$\begin{aligned} \mathit{LFIsize} &= 0.85 \times \mathit{LFsize}^{0.62} \\ \mathit{LFItime} &= 0.17 \times \mathit{LFtime}^{0.65} \end{aligned}$$

These results are close to the expected $\mathcal{O}(\sqrt{n})$ improvement suggested by the n-way example discussed in Section 2. In practice, the exponent is larger than 0.5 due to the representation cost of the proof rules that appear in a proof and cannot be eliminated.

The measured effect of the "occurs-check" optimization described in Section 6 is to reduce the overall reconstruction time for implicit proofs by 40% on the average. If initially the cost of the occurs-check was 44% of the whole reconstruction time, the optimization reduces it to only 2%, thus eliminating over 90% of the cost of occurs checks. These results are consistent with those presented in [7]. As expected, the occurs check optimization does not have any effect when the reconstruction is applied to LF_i proofs without placeholders.

8 Related Work

The problem of redundancy in the representation of proofs has been addressed before for the purpose of simplifying the user interface of theorem provers and proof assistants. Miller [10] suggests an extreme approach where the proof object records only the substitutions for the quantifiers, relying on the decidability of the tautology of the resulting matrix. This leads to very compact proof representations at the expense of an NP-complete tautology checking problem. Furthermore, in the presence of interpreted function symbols and arithmetic (as is always the case in the PCC proofs) the tautology checking problem can easily become undecidable.

The argument synthesis and term reconstruction algorithms implemented in the LEGO [6, 18] and Coq [3] proof assistants are less effective than our algorithm, in the sense that fewer proof subterms can be omitted from the proof representation, and therefore more redundancy has to be tolerated. This is because they address the more difficult problem of representing a proof so that the predicate that it proves can be recovered from it. Our algorithm is able to synthesize more proof subterms by using information both from the context and from the predicate that the proof is supposed to prove.

The implementation of Elf [17], a logic programming language based on LF, contains a reconstruction algorithm that is similar to the one presented here in the sense that missing application arguments can be recovered both from the type of the application (inherited arguments) and from the other arguments of the same application (synthesized arguments). In fact, the Elf reconstruction algorithm is more powerful than ours because it does not impose any restriction on which terms can be missing from the proof. To achieve this flexibility, Elf type reconstruction uses an incomplete algorithm based on constraint solving [16, 15]. Our proof checking algorithm can be characterized as a special and more efficient case of the Elf's reconstruction algorithm, where enough of the proof structure is present so that: (1) there is not need for search, (2) higher-order unification is reduced to a simple extension of first-order unification that respects bound variables, and (3) all constraints that are generated have the simple rigid-rigid or flex-rigid form that can be solved eagerly.

The design of the language L_λ [9] also relies on syntactic restrictions for the purpose of eliminating the need for higher-order unification during type checking. However, the restrictions of L_λ are too strict for our purposes because they prevent the free use of higher-order abstract syntax in the representation of predicates and proofs, requiring instead costly [7, 8] explicit implementations of substitution. In our framework we can still make use of all the representation techniques of the full LF language, and thus gain leverage from substitution in the meta-language, because the restrictions are imposed only on which subterms might be missing from the representation.

Finally, a distinguishing feature of our work from the related work discussed above are the representation algorithms, which take a regular LF representation of a proof and eliminate as many subterms from it as possible, while keeping the resulting proof within the reconstruction capability of the proof checker. Our experimental data supports our claim that the LF_i representation for first-order logic proofs has substantial practical benefits.

9 Conclusion

In this paper we describe the logical framework LF_i that extends the Edinburgh Logical Framework (LF) with the notion of implicit objects, which are incomplete LF objects. Of these implicit objects, the most interesting are those whose missing parts can be easily reconstructed by a simple extension of the LF type checking algorithm, without requiring search, higher-order unification or complicated constraint solving. We define such a subset in a constructive manner by presenting a simple reconstruction algorithm that is syntax directed and requires only first-order unification.

The subset of LF_i that is accepted by the reconstruction algorithm is particularly suited for the compact representation for logical proofs. In addition to the reduced size of proofs, LF_i inherits the advantages of using LF for the formalization of proofs: natural support for hypothetical and parametric judgments and a simple and logic-independent proof checker.

Along with the reconstruction algorithm, we also show a representation algorithm that selectively erases subterms of an LF representation of a proof, operation that sometimes results in two-orders-of-magnitude reductions in proof size and proof-checking time.

To fully comprehend the practical advantages of the LF_i representation of proofs, it is useful to compare it with an alternative special-purpose representation. Consider, for example, a representation of proofs as trees whose internal nodes correspond to uses of proof rules. For each use of a proof rule, we retain in the proof tree a minimal amount of information, so as to minimize the size of the proof representation. The proof checker is implemented as a large case statement, with each case being manually specialized to one proof rule. Such a special-purpose representation is efficient both in terms of proof sizes and proof checking time, but requires a large and difficult to maintain implementation. For example, the implementation must be changed for for every modification to the logic and the proof rules.

We have experimented with such a special-purpose representation for first-order logic proofs and observed that they contain the same amount of information as the corresponding LF_i proofs. Similarly, the process of checking the special-purpose representation of proofs involves the same amount of work (in terms of the number and kind of pattern matching and unification operations) as LF_i reconstruction. In conclusion, by using the LF_i framework, we achieve the efficiency of a special-purpose representation with a general and easier to maintain implementation.

The efficiency gains due to LF_i, added to the simplicity and generality of LF, make LF_i an ideal representation framework for many practical applications where manipulating proofs as first-class objects is required. We have successfully tested the framework as a backend for a theorem prover that emits explicit proofs and as a medium for communicating program-correctness proofs among computer systems, as part of Proof-Carrying Code.

Acknowledgements

We would like to thank Frank Pfenning for introducing us to LF and for many helpful comments on earlier versions of this paper.

References

[1] Abadi, M., Cardelli, L., Curien, P.-L., and Lévy, J.-J. Explicit substitutions. *Journal of Functional Programming 1*, 4 (Oct. 1991), 375–416.

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

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

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

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

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

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

[8] Michaylov, S., and Pfenning, F. Higher-order logic programming as constraint logic programming. In *PPCP'93: First Workshop on Principles and Practice of Constraint Programming* (Newport, Rhode Island, Apr. 1993), Brown University, pp. 221–229.

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

[10] Miller, D. A. A compact representation of proofs. *Studia Logica 46*, 4 (1987), 347–370.

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

[12] Necula, G. C., and Lee, P. Safe kernel extensions without run-time checking. In *Second Symposium on Operating Systems Design and Implementations* (Oct. 1996), Usenix, pp. 229–243.

[13] Necula, G. C., and Lee, P. Efficient representation and validation of logical proofs. Technical Report CMU-CS-97-172, Computer Science Department, Carnegie Mellon University, Oct. 1997.

[14] Necula, G. C., and Lee, P. Safe, untrusted agents using proof-carrying code. In *Special Issue on Mobile Agent Security* (Jan. 1998), G. Vigna, Ed., Springer-Verlag LNCS. To appear.

[15] Pfenning, F. Logic programming in the LF logical framework. In *Logical Frameworks* (1991), G. Huet and G. Plotkin, Eds., Cambridge University Press, pp. 149–181.

[16] Pfenning, F. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science* (Amsterdam, The Netherlands, July 1991), pp. 74–85.

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

[18] Pollack, R. Implicit syntax. Informal Proceedings of First Workshop on Logical Frameworks, Antibes, May 1990.