Efficient Representation and Validation of Logical Proofs

George C. Necula Peter Lee

October 1997

CMU-CS-97-172

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This report describes a framework for representing and validating formal proofs in various axiomatic systems. The framework is based on the Edinburgh Logical Framework (LF) but is optimized for minimizing the size of proofs and the complexity of proof validation, by removing redundant representation components. Several variants of representation algorithms are presented with the resulting representations being a factor of 15 smaller than similar LF representations. The validation algorithm is a reconstruction algorithm that runs about 7 times faster than LF typechecking. We present a full proof of correctness of the reconstruction algorithm and hints for the efficient implementation using explicit substitutions. We conclude with a quantitative analysis of the algorithms.

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.

Keywords: Proof Representation, Proof Checking, Logical Frameworks, Type Reconstruction, Predicate Logic, Explicit Substitutions, Unification, Occurs-Check Optimization, Proof-Carrying Code.

Contents

1 Introduction

The problem of theorem proving has received significant attention from the scientific community in the last 30 years, mostly as a critical component of program verification. The most often analyzed aspect of theorem proving has been the most challenging one, namely the efficiency and practicality of the proof search. Much less attention has been devoted to the problem of producing and manipulating the proofs of the proved theorems.

A typical theorem prover provides a counterexample for each failed theorem but it does not emit a proof of the theorem in case of success. This means that users of such theorem provers are obligated to take on faith their soundness. This is a significant problem, because powerful theorem provers are complex systems, with very complicated invariants that are easy to break during implementation or maintenance. Furthermore, theorem prover bugs might persist for a long time because the user does not usually have a simple criterion to distinguish between theorems that should succeed and those that should not. Because of these reasons we believe that it is useful to instrument a theorem prover to emit a proof for every theorem that it proves. The proof should be detailed enough so that it can be checked by a very simple and easy-to-trust proof checker.

Adding the proof-generating capability to a theorem prover not only makes it easier to verify and to maintain, but also enables its use as a front-end to a Proof-Carrying Code (PCC) [12] system for the safe execution of untrusted code. PCC is a protocol by which two software systems (say a *code producer* and a *code consumer*) can cooperate on the task of convincing the code consumer that a program supplied by the untrusted code producer is safe to run. The key element of the PCC technique is that the producer must supply together with the program a formal safety proof stating the the execution of the program does not violate the safety policy of the code consumer. Then the server can easily validate the safety proof before installing the untrusted code for execution. PCC has uses ranging from operating system kernel extensions to mobile code and safe inter-operation of components written in safe and unsafe languages [12, 13, 14]. In all of these situations, it is important to have compact representation of proofs because they are explicitly manipulated and possibly sent through communication networks. For the PCC technique to be practical it is also important to be able to validate the proofs quickly.

The purpose of this report is to present efficient algorithms for the representation and validation of proofs. These algorithms were developed as part of our PCC implementation effort, which is currently based on first-order logic. Even though the algorithms are optimized and validated experimentally for first-order logic, they are usable for other logics as well.

There are two main factors that have guided our choices for a proof representation algorithm: generality and the ability to handle higher-order representation. Each of these factors are motivated below.

The first impulse in designing an efficient proof representation and validation algorithms is to specialize them to a given logic. For example, we might define the representation and validation algorithm by cases, with one case for each inference rule in the logic. This approach has the major disadvantage that a new representation and validation algorithm has to be designed and implemented for each logic. We would prefer instead to use general algorithms that are parameterized by the particular logic of interest.

A simplistic view of a proof is as a tree, whose leaves represent uses of axioms and whose internal node represent uses of inference rules. If we label each node with the predicate proved by the subtree rooted at that node we can then check the validity of the proof by verifying that each node is a valid instance of an inference rule. Unfortunately, this is not quite enough for a large class of logics that have parametric and hypothetical judgments. For example, the implication introduction rule of first-order logic specifies that the left-hand side of the implication can be used as an assumption but only in the subtree that proves the right-hand side. This side-condition can be expressed naturally using a binding proof constructor that binds the assumption of the left-hand side in the proof of the right-hand side.

To achieve both the generality and the higher-order representation goals we use the Edinburgh Logical Framework (LF) [5] introduced by Harper, Honsell and Plotkin as a general framework for defining logics. LF is general in the sense that implementation of validation and representation is parameterized by the logic of interest. Section 3 presents the logical framework and how it can be used to validate proofs.

The major drawback of representing proofs in LF is that, in general, the representations are unnecessarily large and difficult to check because of many redundant components. To address this problem we introduce in Section 5 the framework LF_i, a variant of LF that can deal with representations where redundant components have been erased. We present in Sections 5 and 6 a validation algorithm for LF_i and then in Section 7 several algorithms that transform a LF representation to the LF_i implicit representation.

The adequacy of representing logical derivation in LF is proved in [5]. In Section 8 we redo the proofs for the LF_i representations. This section is very technical and can be skipped on the first reading. The proof of correctness motivates the design of the reconstruction and validation algorithms and exposes opportunities for optimizations. A basic understanding of this section is required for writing a good implementation of the representation and validation algorithms and also for understanding of the optimizations presented in Section 9.

In Sections 9 and 10 we consider practical aspects related to the concrete implementation of the representation and validation algorithms. The implementation is based on explicit substitutions to minimize the time and space required for proof validation.

Finally, in Section 11 we show quantitative comparisons between the LF and LF_i sizes of representations and speed of validation algorithms. The performance results can be summarized by saying that the implicit representation is on the average 15 times smaller than the LF representation and validating it is on the average 7 times faster. Moreover these improvements seem to increase for larger problems, reaching factor of 44 for the size and 18 for the time, for the largest PCC experiments to date. Following these performance results we evaluate the various optimizations presented in Section 9, showing that they roughly halve the space and time requirements of LF_i proof validation.

2 A First-Order Predicate Logic $\mathcal{L}$

For illustration purposes we introduce in this section a fragment of the first-order predicate logic. In the rest of this paper we will refer to this logic as $\mathcal{L}$. The syntactic elements of $\mathcal{L}$ are grouped in two levels: expressions and predicates. The language of expressions includes integer variables, literals, addition and subtraction:

$$e \quad ::= \quad x \quad | \quad n \quad | \quad e_1 + e_2 \quad | \quad e_1 - e_2$$

The language of predicates is essentially a fragment of first-order predicate logic that includes the predicate **true**, conjunction, implication, universal quantification and expression equality:

$$P \quad ::= \quad \mathbf{true} \quad | \quad P_1 \wedge P_2 \quad | \quad P_1 \supset P_2 \quad | \quad \forall x.P \quad | \quad e_1 = e_2$$

This subset of first-order predicate logic is sufficient for our purposes. However the logic can be extended in two ways. One is to allow other boolean connectives such as negation, disjunction and existential quantification. The other opportunity for extension is to add new function symbols at the level of expressions and predicates. Such extensions of $\mathcal{L}$ can be expressed in the framework presented in the rest of this report by treating new expression and predicate constructors following the model of constructors already existent in $\mathcal{L}$.

2.1 The Proof System

The proof system of $\mathcal{L}$ is the collection of axioms and inference rules that define the valid derivations. Not surprisingly the proof system is composed of two orthogonal components. The first component contains the first-order predicate logic rules and the other gives an interpretation to integer arithmetic functions. We write $\triangleright P$ when the predicate P can be proved using the proof rules in $\mathcal{L}$.

Figure 1 shows the inference rules of first-order predicate logic from $\mathcal{L}$. In an extended logic, negation, disjunction and existential quantification rules must be added.

$$\frac{}{\triangleright\, \mathbf{true}}\,\mathbf{true\_i} \qquad \frac{\triangleright P_1 \qquad \triangleright P_2}{\triangleright P_1 \wedge P_2}\,\mathbf{and\_i} \qquad \frac{\triangleright P_1 \wedge P_2}{\triangleright P_1}\,\mathbf{and\_el} \qquad \frac{\triangleright P_1 \wedge P_2}{\triangleright P_2}\,\mathbf{and\_er}$$

$$\frac{\begin{array}{c}\overline{\triangleright P_1}^{\,u} \\ \vdots u \\ \triangleright P_2\end{array}}{\triangleright P_1 \supset P_2}\,\mathbf{impl\_i}^u \qquad \frac{\triangleright P_1 \supset P_2 \qquad \triangleright P_1}{\triangleright P_2}\,\mathbf{impl\_e} \qquad \frac{\begin{array}{c} v \\ \vdots v \\ \triangleright [v/x]P\end{array}}{\triangleright \forall x\,.\,P}\,\mathbf{all\_i}^v \qquad \frac{\triangleright \forall x\,.\,P}{\triangleright [e/x]P}\,\mathbf{all\_e}$$

Figure 1: Fragment of the first-order predicate logic proof rules.

The choice of axioms for dealing with integer arithmetic is a delicate one. We do not attempt here to have a complete logic but one that is suitable for illustrating the proof representation and validation framework presented in the rest of this report. Soundness of the logic is also not crucial for our purposes, but is a desirable feature. Figure 2 shows our choice of arithmetic axioms.

As an example of a derivation in $\mathcal{L}$ consider the proof of $P = \forall e_1.\forall e_2.\forall e_3.(e_1 = e_2 + e_3) \supset (e_1 - e_3 = e_2)$. The derivation $\mathcal{D}$ of the predicate P is shown in tree form in Figure 3. We will return to this derivation in the following sections to exemplify the various proof representations and the behavior of corresponding validation algorithms.

$$\frac{}{\triangleright E = E}\textbf{=id} \qquad \frac{\triangleright E_2 = E_1}{\triangleright E_1 = E_2}\textbf{=sym} \qquad \frac{\triangleright E_1 = E_2 \qquad \triangleright E_2 = E_3}{\triangleright E_1 = E_3}\textbf{=tr}$$

$$\frac{}{\triangleright E + 0 = E}\textbf{+id} \qquad \frac{}{\triangleright E_1 + E_2 = E_2 + E_1}\textbf{+com} \qquad \frac{}{\triangleright E - E = 0}\textbf{+inv}$$

$$\frac{\triangleright E_1 = E_2 \qquad \triangleright E_1' = E_2'}{\triangleright E_1 + E_1' = E_2 + E_2'}\textbf{+congr} \qquad \frac{\triangleright E_1 = E_2 \qquad \triangleright E_1' = E_2'}{\triangleright E_1 - E_1' = E_2 - E_2'}\textbf{-congr}$$

$$\frac{}{\triangleright (E_1 + E_2) + E_3 = E_1 + (E_2 + E_3)}\textbf{+assoc} \qquad \frac{}{\triangleright (E_1 + E_2) - E_3 = E_1 + (E_2 - E_3)}\textbf{+-assoc}$$

Figure 2: Arithmetic proof rules.

$$\mathcal{D} \;=\; \frac{\frac{\frac{\frac{\frac{\frac{\frac{}{\triangleright e_1 = e_2 + e_3}u \qquad \frac{}{\triangleright e_3 = e_3}\textbf{=id}}{\triangleright e_1 - e_3 = (e_2 + e_3) - e_3}\textbf{-congr} \qquad \genfrac{}{}{0pt}{}{\mathcal{D}_1^u}{\triangleright (e_2 + e_3) - e_3 = e_2}}{\triangleright e_1 - e_3 = e_2}\textbf{=tr}}{\triangleright e_1 = e_2 + e_3 \supset e_1 - e_3 = e_2}\textbf{impl\_i}^u}{\triangleright \forall e_3.\ e_1 = e_2 + e_3 \supset e_1 - e_3 = e_2}\textbf{all\_i}^{e_3}}{\triangleright \forall e_2.\forall e_3.\ e_1 = e_2 + e_3 \supset e_1 - e_3 = e_2}\textbf{all\_i}^{e_2}}{\triangleright \forall e_1.\forall e_2.\forall e_3.\ e_1 = e_2 + e_3 \supset e_1 - e_3 = e_2}\textbf{all\_i}^{e_1}$$

$$\mathcal{D}_1^u = \frac{\frac{}{\triangleright (e_2 + e_3) - e_3 = e_2 + (e_3 - e_3)}\textbf{+-assoc} \qquad \frac{\frac{\frac{}{\triangleright e_2 = e_2}\textbf{=id} \qquad \frac{}{\triangleright e_3 - e_3 = 0}\textbf{+inv}}{\triangleright e_2 + (e_3 - e_3) = e_2 + 0}\textbf{+congr} \qquad \frac{}{\triangleright e_2 + 0 = e_2}\textbf{+id}}{\triangleright e_2 + (e_3 - e_3) = e_2}\textbf{=tr}}{\triangleright (e_2 + e_3) - e_3 = e_2}\textbf{=tr}$$

Figure 3: The derivation $\mathcal{D}$ of $\forall e_1.\forall e_2.\forall e_3.(e_1 = e_2 + e_3) \supset (e_1 - e_3 = e_2)$ in $\mathcal{L}$. For typographical reasons the subderivation $\mathcal{D}_1^u$ is shown as a separate tree.

3 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, hypothetical and schematic judgments. For example it captures the convention that expressions that differ only in the names of bound variables are considered identical. Similarly, it allows direct expression of contexts and variable lookup as they arise in a hypothetical and parametrical judgment. The fact that these techniques are supported by the logical framework is a crucial factor for the succinct formalization of proofs.

The LF representation of a logic consists of two stages. The first stage is the representation of the abstract syntax of the logic under investigation. For example, we will show how to represent expressions and predicates of $\mathcal{L}$ in LF. The second stage is the representation of the semantics of $\mathcal{L}$. We do this by representing in LF the proof rules set $\mathcal{L}$. Then we show how actual proofs can be constructed from instances of proof rules.

The LF type theory is a language with entities of three levels: objects, types and kinds. Types are used to qualify objects and similarly, kinds are used to qualify types. The abstract syntax of these entities is shown below:

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

Here `Type` is the base kind, a is a type constant and c is an object constant.

We represent our logic in LF by means of a signature Σ that assigns types to a set of constants describing the syntax of expressions and predicates, and the proof rules of our logic. Then we define a representation function that will map expressions, predicates and their proofs in our logic to LF objects constructed with constants declared in the signature Σ.

The main representation strategy in LF is that judgments (e.g., statements about the validity of predicates) are represented as LF types and judgment derivations (e.g. a proof of a predicate) are represented as objects whose type is the representation of the judgments they prove. Type checking in the LF type discipline can then be used to check the validity of logic proofs.

We start now to present the signature Σ corresponding to the logic $\mathcal{L}$.

3.1 Representing Abstract Syntax: Expressions and Predicates

First, we define in Figure 4 the LF types `exp` of expressions and `pred` of predicates. All of these are atomic LF types of base kind `Type`.

```
exp  : Type
pred : Type
```

Figure 4: the LF signature Σ (part 1). Base type constants.

Then for each expression and predicate constructor we define an LF constant as shown in Figure 5. One of the most interesting cases is the universal quantification. Care must be taken when dealing with universal quantification because of the presence of bound variables. For example, we must ensure that the representation captures the fact that the bound variable is local to the body of the quantification and that two expressions differing only in the name of the bound variables are equal. Moreover, when an expression is substituted for the bound variable we must ensure that no free variable of the substituted expression is captured.

One of the main reasons we chose LF as a proof representation language is that it provides mechanisms for dealing with bound variables. Note in Figure 5 how the universal quantification is represented as a higher-order construct by representing the bound logical variable by a bound LF variable. This effectively delegates all the tedious manipulations of bound variables to LF.

$$
\begin{array}{lcl}
0 & : & \texttt{exp} \\
+ & : & \texttt{exp} \to \texttt{exp} \to \texttt{exp} \\
- & : & \texttt{exp} \to \texttt{exp} \to \texttt{exp} \\
= & : & \texttt{exp} \to \texttt{exp} \to \texttt{pred} \\
<> & : & \texttt{exp} \to \texttt{exp} \to \texttt{pred} \\
\texttt{true} & : & \texttt{pred} \\
\texttt{and} & : & \texttt{pred} \to \texttt{pred} \to \texttt{pred} \\
\texttt{impl} & : & \texttt{pred} \to \texttt{pred} \to \texttt{pred} \\
\texttt{all} & : & (\texttt{exp} \to \texttt{pred}) \to \texttt{pred}
\end{array}
$$

Figure 5: The LF signature Σ (part 2). Expression and predicate constructors.

The LF representation function $\ulcorner \cdot \urcorner$ is defined inductively on the structure of expressions, types and predicates as shown in Figures 6 and 7.

$$
\begin{array}{rcl}
\ulcorner x \urcorner & = & x \\
\ulcorner e_1 + e_2 \urcorner & = & + \ulcorner e_1 \urcorner \ulcorner e_2 \urcorner \\
\ulcorner e_1 - e_2 \urcorner & = & - \ulcorner e_1 \urcorner \ulcorner e_2 \urcorner
\end{array}
$$

Figure 6: LF representation (part 1). Expressions.

$$
\begin{array}{rcl}
\ulcorner e_1 = e_2 \urcorner & = & = \ulcorner e_1 \urcorner \ulcorner e_2 \urcorner \\
\ulcorner e_1 \neq e_2 \urcorner & = & <> \ulcorner e_1 \urcorner \ulcorner e_2 \urcorner \\
\ulcorner \textbf{true} \urcorner & = & \texttt{true} \\
\ulcorner P \wedge R \urcorner & = & \texttt{and}\, \ulcorner P \urcorner \ulcorner R \urcorner \\
\ulcorner P \supset R \urcorner & = & \texttt{impl}\, \ulcorner P \urcorner \ulcorner R \urcorner \\
\ulcorner \forall x.P \urcorner & = & \texttt{all}\; (\lambda x\!:\!\texttt{exp}.\ulcorner P \urcorner)
\end{array}
$$

Figure 7: LF representation (part 2). Types and predicates.

3.2 Representing Semantics: Proofs

Up to this point we have defined the representation of expressions, types and predicates in LF. Our ultimate goal is to be able to represent proofs of predicates or equivalently derivations of the validity of predicates. We follow the same pattern as for syntactic constructs and we introduce a type of proofs `pf` and then define each proof rule as an LF constant of this type. Things are actually more involved due to the fact that we want the type of a proof to determine the predicate that is being proved. In this way we verify by type checking not only that a proof is valid but also

$$\begin{array}{lcl}
\texttt{true_i} & : & \texttt{pf true} \\
\texttt{and_i} & : & \Pi p\!:\!\texttt{pred}.\Pi r\!:\!\texttt{pred}.\texttt{pf}\ p \rightarrow \texttt{pf}\ r \rightarrow \texttt{pf}\ (\texttt{and}\ p\ \ r) \\
\texttt{and_el} & : & \Pi p\!:\!\texttt{pred}.\Pi r\!:\!\texttt{pred}.\texttt{pf}\ (\texttt{and}\ p\ \ r) \rightarrow \texttt{pf}\ p \\
\texttt{and_er} & : & \Pi p\!:\!\texttt{pred}.\Pi r\!:\!\texttt{pred}.\texttt{pf}\ (\texttt{and}\ p\ \ r) \rightarrow \texttt{pf}\ r \\
\texttt{impl_i} & : & \Pi p\!:\!\texttt{pred}.\Pi r\!:\!\texttt{pred}.(\texttt{pf}\ p \rightarrow \texttt{pf}\ r) \rightarrow \texttt{pf}\ (\texttt{impl}\ p\ \ r) \\
\texttt{impl_e} & : & \Pi p\!:\!\texttt{pred}.\Pi r\!:\!\texttt{pred}.\texttt{pf}\ (\texttt{impl}\ p\ \ r) \rightarrow \texttt{pf}\ p \rightarrow \texttt{pf}\ r \\
\texttt{all_i} & : & \Pi p\!:\!\texttt{exp} \rightarrow \texttt{pred}.(\Pi v\!:\!\texttt{exp}.\texttt{pf}\ (p\ v)) \rightarrow \texttt{pf}\ (\texttt{all}\ \ p) \\
\texttt{all_e} & : & \Pi p\!:\!\texttt{exp} \rightarrow \texttt{pred}.\Pi e\!:\!\texttt{exp}.\texttt{pf}\ (\texttt{all}\ \ p) \rightarrow \texttt{pf}\ (p\ e)
\end{array}$$

Figure 8: The LF signature Σ (part 3). First-order logic proof constants (see Figure 1).

$$\begin{array}{lcl}
= \texttt{id} & : & \Pi e\!:\!\texttt{exp}.\texttt{pf}\ (=\ e\ e) \\
= \texttt{sym} & : & \Pi e_1\!:\!\texttt{exp}.\Pi e_2\!:\!\texttt{exp}.\texttt{pf}\ (=\ e_2\ e_1) \rightarrow \texttt{pf}\ (=\ e_1\ e_2) \\
= \texttt{tr} & : & \Pi e_1\!:\!\texttt{exp}.\Pi e_2\!:\!\texttt{exp}.\Pi e_3\!:\!\texttt{exp}.\texttt{pf}\ (=\ e_1\ e_2) \rightarrow \texttt{pf}\ (=\ e_2\ e_3) \rightarrow \texttt{pf}\ (=\ e_1\ e_3) \\
+\texttt{id} & : & \Pi e\!:\!\texttt{exp}.\texttt{pf}\ (=\ (+\ e\ 0)\ e) \\
+\texttt{com} & : & \Pi e_1\!:\!\texttt{exp}.\Pi e_2\!:\!\texttt{exp}.\texttt{pf}\ (=\ (+\ e_1\ e_2)\ (+\ e_2\ e_1)) \\
+\texttt{inv} & : & \Pi e\!:\!\texttt{exp}.\texttt{pf}\ (=\ (-\ e\ e)\ 0) \\
+\texttt{congr} & : & \Pi e_1\!:\!\texttt{exp}.\Pi e_2\!:\!\texttt{exp}.\Pi e_1'\!:\!\texttt{exp}.\Pi e_2'\!:\!\texttt{exp}. \\
 & & \texttt{pf}\ (=\ e_1\ e_2) \rightarrow \texttt{pf}\ (=\ e_1'\ e_2') \rightarrow \texttt{pf}\ (=\ (+\ e_1\ e_1')\ (+\ e_2\ e_2')) \\
-\texttt{congr} & : & \Pi e_1\!:\!\texttt{exp}.\Pi e_2\!:\!\texttt{exp}.\Pi e_1'\!:\!\texttt{exp}.\Pi e_2'\!:\!\texttt{exp}. \\
 & & \texttt{pf}\ (=\ e_1\ e_2) \rightarrow \texttt{pf}\ (=\ e_1'\ e_2') \rightarrow \texttt{pf}\ (=\ (-\ e_1\ e_1')\ (-\ e_2\ e_2')) \\
+\texttt{assoc} & : & \Pi e_1\!:\!\texttt{exp}.\Pi e_2\!:\!\texttt{exp}.\Pi e_3\!:\!\texttt{exp}.\texttt{pf}\ (=\ (+\ (+\ e_1\ e_2)\ e_3)\ (+\ e_1\ (+\ e_2\ e_3))) \\
+-\texttt{assoc} & : & \Pi e_1\!:\!\texttt{exp}.\Pi e_2\!:\!\texttt{exp}.\Pi e_3\!:\!\texttt{exp}.\texttt{pf}\ (=\ (-\ (+\ e_1\ e_2)\ e_3)\ (+\ e_1\ (-\ e_2\ e_3)))
\end{array}$$

Figure 9: The LF signature Σ (part 4). Application-specific proof constants (see Figure 2).

that it proves the desired predicate. This is possible to express in the LF type discipline by using type families indexed by terms.

Thus pf is actually a type family indexed by LF representation of predicates:

$$\texttt{pf} \quad : \quad \texttt{pred} \rightarrow \texttt{Type}$$

Following the model of expressions and predicates we add to the signature Σ a constant for each proof rule in $\mathcal{L}$. The constants corresponding to the proof rules used by our example are shown in Figure 8 (first-order logic proof rules) and Figure 9 (arithmetic proof rules).

We then extend the representation function $\ulcorner\cdot\urcorner$ to derivations. When doing so care must be taken with hypothetical and schematic judgments, such as the implication introduction and the universal quantification introduction rules. We show in Figure 11 the representation of the introduction rules for conjunction, implication and universal quantification. The representation of the conjunction introduction is typical for all other rules not shown here, including the arithmetic proof rules.

The implication introduction rule introduces the hypothesis labelled u for the purpose of deriving P_2. Checking an instance of this rule schema involves verifying that it discharges properly the hypothesis u. Equivalently, the derivation $\mathcal{D}_u$ must be hypothetical in u. This is expressed naturally in LF by representing the hypothesis as a variable bound in the derivation $\mathcal{D}^u$. Finally, the LF representation of our logic contains also the representation of the application-specific proof rules. Their representation is straightforward because they do not involve hypothetical judgments. As an example we show below the LF representation of the symmetry rule for equality:

$$\ulcorner \dfrac{\begin{array}{c}\mathcal{D}\\ \rhd e_2 = e_1\end{array}}{\rhd e_1 = e_2} \urcorner = \texttt{=\_sym}\ \ulcorner e_1 \urcorner\, \ulcorner e_2 \urcorner\, \ulcorner \mathcal{D} \urcorner$$

Figure 10: LF representation (part 4). Fragment of the application-specific rule representation (see Figure 2).

$$\ulcorner \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ \rhd P_1 & \rhd P_2\end{array}}{\rhd P_1 \wedge P_2} \urcorner = \texttt{and\_i}\ \ulcorner P_1 \urcorner\, \ulcorner P_2 \urcorner\, \ulcorner \mathcal{D}_1 \urcorner\, \ulcorner \mathcal{D}_2 \urcorner$$

$$\ulcorner \dfrac{\begin{array}{c}\overline{\rhd P_1}^{\,u}\\ \vdots\ \mathcal{D}_u\\ \rhd P_2\end{array}}{\rhd P_1 \supset P_2}u \urcorner = \texttt{impl\_i}\ \ulcorner P_1 \urcorner\, \ulcorner P_2 \urcorner\ (\lambda u{:}\texttt{pf}\ \ulcorner P_1 \urcorner . \ulcorner \mathcal{D}^u \urcorner)$$

$$\ulcorner \dfrac{\begin{array}{c}\mathcal{D}_v\\ \rhd [v/x]P_x\end{array}}{\rhd \forall x.P_x} \urcorner = \texttt{all\_i}\ (\lambda x{:}\texttt{exp}.\ulcorner P_x \urcorner)\ (\lambda v{:}\texttt{exp}.\ulcorner \mathcal{D}^v \urcorner)$$

Figure 11: LF representation (part 3). Fragment of the first-order logic rule representation (see Figure 1).

As an example of a proof representation in LF we show in Figure 12 the representation of the proof $\mathcal{D}$ from Figure 3 of the predicate $\forall e_1.\forall e_2.\forall e_3.(e_1 = e_2 + e_3) \supset (e_1 - e_3 = e_2)$. The LF representation of the proof is computed as $\ulcorner \mathcal{D} \urcorner$ from the representation in tree form. Compare this LF representation to the proof in tree form as shown in Figure 3. The only difference is the syntax used to express the proof and it seems obvious how one could reconstruct the tree form of the proof from the LF representation.

In the next section we show a simple algorithm that can be used to validate proof representations. The algorithm is parameterized by the signature Σ and therefore can be reused for checking validity of derivations in other logics just by changing the LF representation signature.

4 Type Checking in the LF Type System

One of the advantages of using LF for proof representation is that proof validity can be determined by a simple type-checking algorithm. That is, to check that the LF object M is the representation of a valid proof of the predicate P we use the LF typing rules (to be presented below) to verify that M has type $\texttt{pf}\ \ulcorner P \urcorner$ in the context of the signature Σ defining the valid proof rules.

In order to define the typing rules we must formalize the syntax of signatures and we also introduce typing contexts that assign types to free variables:

$$\begin{array}{lcllll}\Sigma & ::= & \cdot & \mid\ \Sigma, a : K & \mid\ \Sigma, c : A & \text{Signatures}\\ \Gamma & ::= & \cdot & \mid\ \Gamma, x : A & & \text{Contexts}\end{array}$$

We define the typing judgment for LF objects in terms of three additional judgments as shown below:

$$\begin{array}{ll} \Gamma \vdash^{LF} M : A & M \text{ is a valid object of type } A \\ \Gamma \vdash^{LF} A : K & A \text{ is a valid type of kind } K \\ A \equiv_\beta B & A \text{ is } \beta\text{-equivalent to } B \\ M \equiv_\beta N & M \text{ is } \beta\text{-equivalent to } N \end{array}$$

The complete LF type-system defines also a typing judgment for kinds. We omit this judgment here because, for the purpose of checking the validity of proofs, kinds can be assumed to be well-typed. In particular the only kinds relevant to checking proofs are the base kind and the kinds associated with the type-constants in the signature Σ, all of which can be trusted to be well-typed. For the signature corresponding to the logic $\mathcal{L}$ these kinds are `Type` and `pred` $\to$ `Type`, which are obviously well-typed.

We show in Table 1 the definition of the judgments introduced above. For the β-equivalence judgment we omit the rules that define it to be an equivalence and a congruence.

Types :

$$\frac{\Sigma(a) = K}{\Gamma \vdash^{LF} a : K} \qquad \frac{\Gamma \vdash^{LF} A : \Pi x{:}B.K \qquad \Gamma \vdash^{LF} M : B}{\Gamma \vdash^{LF} A\ M : [M/x]K} \qquad \frac{\Gamma \vdash^{LF} A : \mathtt{Type} \qquad \Gamma, x : A \vdash^{LF} B : \mathtt{Type}}{\Gamma \vdash^{LF} \Pi x{:}A.B : \mathtt{Type}}$$

Objects :

$$\frac{\Sigma(c) = A}{\Gamma \vdash^{LF} c : A} \qquad \frac{\Gamma(x) = A}{\Gamma \vdash^{LF} x : A} \qquad \frac{\Gamma, x : A \vdash^{LF} M : B}{\Gamma \vdash^{LF} \lambda x{:}A.M : \Pi x{:}A.B}$$

$$\frac{\Gamma \vdash^{LF} M : \Pi x{:}A.B \qquad \Gamma \vdash^{LF} N : A}{\Gamma \vdash^{LF} MN : [N/x]B} \qquad \frac{\Gamma \vdash^{LF} M : A \qquad A \equiv_\beta B}{\Gamma \vdash^{LF} M : B}$$

Equivalence :

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

Table 1: Type checking in the LF type discipline

The following theorems can be proven using techniques which are similar to those used in [5, 13]. These theorems establish that LF type-checking is an adequate procedure for checking the validity of LF representations of proofs in $\mathcal{L}$. As mentioned before, the adequacy of LF type-checking holds even if $\mathcal{L}$ is extended with additional boolean connectives and function symbols. The adequacy holds even for higher-order logics [5].

Theorem 4.1 (Adequacy of Expression Representation.)

1. *If e has free variables among $x_1, \ldots, x_n$, then* $x_1 : \mathtt{exp}, \ldots, x_n : \mathtt{exp} \vdash^{LF} \ulcorner e \urcorner : \mathtt{exp}$.

2. *If M is an LF object such that* $x_1 : \mathtt{exp}, \ldots, x_n : \mathtt{exp} \vdash^{LF} M : \mathtt{exp}$, *then exists an expression e with free variables among $x_1, \ldots, x_n$ such that $\ulcorner e \urcorner \equiv_\beta M$.*

Theorem 4.2 (Adequacy of Predicate Representation.)

1. *If P has free variables among $x_1, \ldots, x_n$, then* $x_1 : \mathtt{exp}, \ldots, x_n : \mathtt{exp} \vdash^{LF} \ulcorner P \urcorner : \mathtt{pred}$.

2. *If M is an LF object such that $x_1 : \mathtt{exp}, \ldots, x_n : \mathtt{exp} \stackrel{LF}{\vdash} M : \mathtt{pred}$, then there exists a predicate P with free variables among $x_1, \ldots, x_n$ such that $\ulcorner P \urcorner \equiv_\beta M$.*

Theorem 4.3 (Adequacy of Derivation Representation.)

1. *If $\mathcal{D} :: \ \triangleright \mathcal{P}$ is a derivation of P with parameters v_i $(i = 1, \ldots, n)$ and from hypotheses $u_j :: \ \triangleright P_j$ $(j = 1, \ldots, m)$ then $v_i : \mathtt{exp}, u_j : \mathtt{pf} \ulcorner P_j \urcorner \stackrel{LF}{\vdash} \ulcorner \mathcal{D} \urcorner : \mathtt{pf} \ulcorner P \urcorner$.*

2. *If M is an LF object such that $v_i : \mathtt{exp}, u_j : \mathtt{pf} \ulcorner P_j \urcorner \stackrel{LF}{\vdash} M : \mathtt{pf} \ulcorner P \urcorner$, then there exists a derivation $\mathcal{D} :: \ \triangleright \mathcal{P}$ of P with parameters v_i $(i = 1, \ldots, n)$ and from hypotheses $u_j :: \ \triangleright P_j$ $(j = 1, \ldots, m)$ such that $\ulcorner \mathcal{D} \urcorner \equiv_\beta M$.*

The reason for the asymmetry in the statement of the adequacy theorems above is that while the representation of an expression, predicate or derivation is always a well-typed LF object (point 1 in the theorems above) not every well-typed LF object is the representation of an expression, predicate or derivation even though it has the appropriate type. The reason is that the LF representation function only produces objects that do not contain β-redices (canonical objects). In fact the adequacy proofs in [5, 13] introduce first the notion of canonical LF objects and then prove that $\ulcorner \cdot \urcorner$ is a bijective function. Those results can be used to obtain proofs of the above adequacy theorems by introducing a normalization judgment that can be shown to preserve β-equivalence.

```
all_i (λe1 : exp.
        all (λe2 : exp.
              all (λe3 : exp.
                   => (= e1(+ e2 e3))
                      (= (- e1 e3) e2))))
      (λe1 : exp
        all_i (λe2 : exp.
               all (λe3 : exp.
                    => (= e1(+ e2 e3))
                       (= (- e1 e3) e2)))
              (λe2 : exp
                all_i (λe3 : exp.
                       => (= e1(+ e2 e3))
                          (= (- e1 e3) e2))
                      (λe3 : exp
                        (impl_i (= e1(+ e2 e3))
                                (= (- e1 e3) e2)
                                   (λu : pf (= e1(+ e2 e3)).
                                     (=tr (- e1 e3)
                                          (- (+ e2 e3) e3)
                                          e2
                                          (-congr e1
                                                  (+ e2 e3)
                                                  e3
                                                  e3
                                                  u
                                                  (=id e3))
                                              (=tr (- (+ e2 e3) e3)
                                                   (+ e2 (- e3 e3))
                                                   e2
                                                   (+-assoc e2
                                                            e3
                                                            e3)
                                                   (=tr (+ e2 (- e3 e3))
                                                        (+ e2 0)
                                                        e2
                                                        (+congr e2
                                                                e2
                                                                (- e3 e3)
                                                                0
                                                                (=id e2)
                                                                (+inv e3))
                                                  (+id e2)))))))))
```

Figure 12: LF representation of the proof shown in Figure 3.

5 The Implicit LF Representation

The LF representation and type-checking algorithm presented in the previous section are adequate for the representation and validation of proofs. However the proof representations are unnecessarily large. This is apparent in the proof representation of Figure 12, where, for example, the expression e_2 occurs 23 times, and it becomes even more pronounced in larger examples. The redundancy in the LF representation is no larger than the redundancy in the original proof of Figure 3, but nevertheless we would prefer to manipulate smaller proofs.

The size of proofs, in general, is an important factor in any application that manipulates proofs explicitly, but the redundancy of representation in particular, has important consequences for the efficiency of proof checking. Consider an instance of a Proof-Carrying Code where the code consumer desires to check that the untrusted safety proof (for example, the one from Figure 12) enclosed with the code proves a certain predicate. In such a situation every subterm of the proof representation must be type-checked. This means that each of the 23 occurrences of the LF term e_2 must be type-checked separately. Moreover, following each of the type-checking operations the term must be compared with the instance of itself contained in the predicate to be proven, to ensure that every subderivation proves the desired predicate. Therefore the redundancy in the representation increases the amount of required checks and therefore can lead to inefficient proof validation.

We address the problem of proof size at its source. We eliminate the redundancy in the representation by omitting redundant copies and maintaining only one copy. It is a good idea to try to preserve only the copy contained in the proved predicate, if such a copy exists. This is motivated by the fact that the proved predicate, and all of its subterms, can be trusted to be well-formed.

During type-checking we reconstruct the missing copies from the subterms of the proven predicate. This addresses not only the proof size problem but also reduces drastically the effort involved in proof validation. The reason is that the reconstruction algorithm is designed so that whenever a missing copy of a subterm is reconstructed we know that the resulting term is well-typed, without having to typecheck it. Similarly the equivalence check does not normally have to be performed because of the properties of reconstruction.

The algorithm that we present in this section is able to analyze the term shown in Figure 13 and check that it represents the implicit form of a valid proof of the predicate shown in Figure 3. Compare this implicit representation with the full LF representation of a proof of the same predicate shown in Figure 12.

Before we plunge into the formal details of the type reconstruction we show how the algorithm works on a simple example. For the purpose of this example consider a new kind of object, called a placeholder, written as $*$ and marking a missing LF object. Consider now the proof of the predicate $P \supset P \wedge P$. If we apply the LF representation algorithm presented in the previous section, we obtain:

$$\texttt{impl\_i}\ \ulcorner P \urcorner\ (\texttt{and}\ \ulcorner P \urcorner\ \ulcorner P \urcorner)\ (\lambda u{:}\texttt{pf}\ \ulcorner P \urcorner.\texttt{and\_i}\ \ulcorner P \urcorner\ \ulcorner P \urcorner\ u\ u) \tag{1}$$

The redundancy of the representation is manifested through the presence of 6 copies of the term $\ulcorner P \urcorner$, which can be a sizeable term in general. It is easy to check the type of the representation above to be $\texttt{pf}\ (\texttt{impl}\ \ulcorner P \urcorner\ (\texttt{and}\ \ulcorner P \urcorner\ \ulcorner P \urcorner))$. We are going to replace all occurrences of $\ulcorner P \urcorner$ in the representation of the proof by placeholders and then show how type-reconstruction works for the resulting object.

The implicit representation of the considered proof is:

$$\texttt{impl\_i}\ *_1\ *_2\ (\lambda u{:}*_3.\texttt{and\_i}\ *_4\ *_5\ u\ u) \tag{2}$$

```
all_i *
      (λe1 : *
        all_i *
              (λe2 : *
                all_i *
                      (λe3 : *
                        (impl_i * *
                                (λu : *.
                                  (=tr * * *
                                       (-congr * * * *
                                               u
                                               (=id *))
                                       (=tr * * *
                                           (+-assoc * * *)
                                           (=tr * * *
                                                (+congr * * * *
                                                        (=id *)
                                                        (+inv *))
                                               (+id *)))))))))
```

Figure 13: Implicit LF representation of the proof shown in Figure 3.

where $*_i$ are 5 placeholders, which are labelled for demonstration purposes. We claim that the above LF term captures the essence of the proof. The remaining parts of the proof representation can be recovered while verifying that the the term has type $\texttt{pf}\ (\texttt{impl}\ \ulcorner P \urcorner\ (\texttt{and}\ \ulcorner P \urcorner\ \ulcorner P \urcorner))$.

Typechecking starts by recognizing the top-level constructor impl_i. The type of the entire term, $\texttt{pf}\ (\texttt{impl}\ \ulcorner P \urcorner\ (\texttt{and}\ \ulcorner P \urcorner\ \ulcorner P \urcorner))$, is "matched" against the result type of the impl_i constant, as given by the signature Σ. The result of this matching is an instantiation for placeholders 1 and 2 and a residual type-checking constraint for the explicit argument of impl_i:

$$\begin{array}{rlcl} & *_1 & \equiv & \ulcorner P \urcorner \\ & *_2 & \equiv & \texttt{and}\ \ulcorner P \urcorner\ \ulcorner P \urcorner \\ \vdash & (\lambda u : *_3 . \texttt{and\_i}\ *_4\ *_5\ u\ u) & : & \texttt{pf}\ \ulcorner P \urcorner \rightarrow \texttt{pf}\ (\texttt{and}\ \ulcorner P \urcorner\ \ulcorner P \urcorner) \end{array}$$

Now we continue with the remaining type-checking constraint. From its desired type we immediately obtain the value of placeholder 3 and a typing constraint for the body:

$$\begin{array}{rlcl} & *_3 & \equiv & \texttt{pf}\ \ulcorner P \urcorner \\ u : \texttt{pf}\ \ulcorner P \urcorner \vdash & \texttt{and\_i}\ *_4\ *_5\ u\ u & : & \texttt{pf}\ (\texttt{and}\ \ulcorner P \urcorner\ \ulcorner P \urcorner) \end{array}$$

Now we notice that and_i is the top-level constant and by matching its result type declared in the signature with the goal type we get the instantiation for placeholders 4 and 5 and two residual typing constraints:

$$\begin{array}{rlcl} & *_4 & \equiv & \ulcorner P \urcorner \\ & *_5 & \equiv & \ulcorner P \urcorner \\ u : \texttt{pf}\ \ulcorner P \urcorner \vdash & u & : & \texttt{pf}\ \ulcorner P \urcorner \\ u : \texttt{pf}\ \ulcorner P \urcorner \vdash & u & : & \texttt{pf}\ \ulcorner P \urcorner \end{array}$$

The remaining two typechecking constraints are solved by the variable typechecking rule, and this concludes typechecking the entire proof. We reconstructed the full representation of the proof by instantiating all the placeholders with well-typed LF objects. We know that these instantiations are well-typed because they are ultimately extracted from the original constraint type, which is assumed to contain only well-typed subterms.

Starting with the next section we define a type-system, called LF_i, which is very similar to the LF type-system but allows for placeholders. However, the LF_i type-system is not very useful for typechecking or type reconstruction. For this purpose we describe in Section 6 a reconstruction algorithm for LF_i.

5.1 The $\mathbf{LF}_i$ Type-System

We extend the syntax of LF objects with placeholders, written $*$. An object is fully reconstructed, or fully explicit, when it is placeholder-free. We write $\mathrm{PF}(M)$ to denote this property. We extend this notation to type environments and we write $\mathrm{PF}(\Gamma)$ to denote that all the types assigned in Γ to variables are placeholder-free. We also introduce the implicitly typed abstraction, written $\lambda x.M$, at the level of objects.

We call the resulting system of terms and types implicit LF and we refer to it by LF_i. The typing rules of LF_i are an extension of the LF typing rules with two new typing rules for dealing with implicit abstraction and placeholders, and one new β-equivalence rule dealing with implicit abstraction. These additions are shown in Figure 14. The LF_i typing judgment is written $\Gamma \overset{i}{\vdash} M : A$.

Objects :

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

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

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

Equivalence :

$$\overline{(\lambda x\!:\!A.M)N \equiv_\beta [N/x]M} \qquad \overline{(\lambda x.M)N \equiv_\beta [N/x]M}$$

Figure 14: The LF_i type-system as an extension of LF.

Note that according to the LF_i type-system placeholders cannot occur on a function position. This restriction is consistent with the previous example and is essential for the correctness of the reconstruction algorithm. Also note that we restrict the types to be placeholder-free in several rules. This restriction simplifies greatly the proofs of soundness and does not seem to get in the way when validating implicit representation of proofs.

A quick analysis of the LF_i type-system reveals that it is not very useful for type-checking or type-inference. The main reason is that typechecking an application involves "guessing" appropriate A and N. The type A can sometimes be recovered from the type of the application head, but the term N in an application to a placeholder cannot be found easily in general. This is not a problem for us because we are going to use the LF_i type-system just as a step in proving the correctness of the type-reconstruction algorithm presented in the next section, and not as the basis for an implementation of a typechecking algorithm.

The only property of interest of the LF_i type-system is that once we have a typing derivation we can reconstruct the object involved and a corresponding LF typing-derivation. To make this more precise we introduce the notation $M \nearrow M'$ to denote that M' is a fully-reconstructed version of the implicit object M (i.e., $\mathrm{PF}(M')$). This means that all the placeholders in M have been replaced by fully-explicit LF objects and similarly all the implicit abstractions have been replaced by explicit abstractions. Note that the reconstruction relation is not a function as there might be many reconstructions of a given implicit object or type.

Theorem 5.1 Soundness of $\mathbf{LF}_i$ typing *If* $\Gamma \overset{i}{\vdash} M : A$ *and* $\mathrm{PF}(\Gamma)$, $\mathrm{PF}(A)$, *then there exists* M' *such that* $M \nearrow M'$ *and* $\Gamma \overset{LF}{\vdash} M' : A$.

The proof of Theorem 5.1 is shown in Section 8.6.

To give an example of a situation where Theorem 5.1 is used consider that we have constructed—using the algorithm presented in the next section for example—a typing derivation $\cdot \overset{i}{\vdash} M : \mathtt{pf} \ulcorner P \urcorner$ for some predicate P and a term M possibly containing placeholders. Then Theorem 5.1 says that there exists an LF term M' such that $\cdot \overset{LF}{\vdash} M' : \mathtt{pf} \ulcorner P \urcorner$, or equivalently that there exists a proof of the predicate P. Theorem 5.1 reduces the problem of checking a proof to finding a LF_i typing-derivation for the implicit proof-representation M.

6 An Algorithm for $\mathbf{LF}_i$ Type Reconstruction

The LF_i type-system presented in the previous section has the benefit that can deal with implicit LF terms. However this type-system does not immediately suggest a type-checking algorithm, as explained before. We show in this section an algorithm that can be used to typecheck LF_i terms, or more precisely to reconstruct and typecheck LF_i terms.

We start with introducing some notation and then we present the algorithm at an abstract level, in terms of inference rules, and we deal with actual implementation issues in Section 10.

6.1 Notation

In addition to the placeholder constants introduced in the previous section we introduce a new brand of variables. These variables play similar role to that of placeholders in that they stand for missing terms. We shall use the letters x and y to denote traditional LF variables and the letter u to denote a placeholder variable. Also, we use the letter Δ to denote a type environment containing only type assignments for placeholder variables. The letter Γ will denote type environments containing any kind of variables. In the special case when an LF object M does not contain placeholder variables we write $\mathrm{PVF}(M)$. Note that placeholder variables always occur free.

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.

The main operation on placeholder variables is substitution with LF terms. We define the syntactic class of substitutions Ψ as follows:

$$\Psi \quad ::= \quad \cdot \quad | \quad u \leftarrow M \quad \text{Substitutions}$$

We denote by $\Psi(M)$ the term obtained by performing the substitution Ψ on M. We write $Dom(\Psi)$ to refer to the set of placeholder variables on which Ψ is defined. We write $\mathrm{PF}(\Psi)$ to mean that all the terms substituted for placeholder variables are placeholder-free.

We write $\Psi|_S$ to denote the substitution obtained from Ψ by restricting it to the set of placeholder variables S.

One of the key operations performed by the reconstruction algorithm is to compute substitutions through unification of terms or types. For a more precise presentation we use two flavors of unification shown below as unification of terms. The same notation is used for expressing unification of types.

$$\begin{array}{ll} M \approx_a M' \Rightarrow \Psi & \text{Atomic Unification} \\ M \approx M' \Rightarrow \Psi & \text{Unification} \end{array}$$

The last syntactic construct that we introduce is a list of type reconstruction constraints defined as follows:

$$C \quad ::= \quad \cdot \quad | \quad C, M : A \quad | \quad C, A \approx_a B$$

The reconstruction algorithm is described by the two unification judgments introduced above and three additional mutually recursive judgments shown below:

$$\begin{array}{ll} \Gamma \overset{r}{\vdash} M : A \Rightarrow \Psi & \text{Main reconstruction judgment} \\ \Gamma \overset{r}{\vdash} M \Rightarrow (\Delta \,;\, C \,;\, B) & \text{Collect constraints} \\ \Gamma \overset{r}{\vdash} C \Rightarrow \Psi & \text{Solve constraints} \end{array}$$

We continue now with the definition of the five judgments introduced above. The collection of these inference rules constitute an abstract description of the reconstruction algorithm.

6.2 Collecting Typechecking Constraints : $\Gamma \overset{r}{\vdash} M \Rightarrow (\Delta \,;\, C \,;\, B)$

This judgment is used for atomic objects M, which are constants or variables applied to zero or more canonical objects. A canonical object is a sequence of abstractions with an atomic body. The object M is scanned to find the application head, whose type is read from the signature Σ or from the variable type environment Γ. Then all arguments are collected in the residual type constraint list C together with their type. Placeholder variables are introduced for each placeholder arguments and are collected in Δ with their type. The resulting type B is the type of the whole application and might contain placeholder variables from Δ. Similarly, the types in the constraint list C might contain placeholders variables from Δ. Note that no placeholders can occur in types because of the side-condition on the rule pertaining to the application to a placeholder.

$$\frac{}{\Gamma \overset{r}{\vdash} c \Rightarrow (\cdot \,;\, \cdot \,;\, \Sigma(c))} \qquad \frac{}{\Gamma \overset{r}{\vdash} x \Rightarrow (\cdot \,;\, \cdot \,;\, \Gamma(x))}$$

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

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

In the next section we describe how the list of constraints is solved. The main reason we separate the tasks of collecting constraints and solving them is to allow an arbitrary order in solving the constraints. We discovered that this can greatly increase the effectiveness of the reconstruction algorithm, with the benefit that proof representations can be made smaller by making more subterms implicit.

The restriction in the explicit application rule ensures that the resulting type $[N/x]B$ does not have placeholders, provided that B does not have placeholders. This restriction simplifies the proof of correctness of the reconstruction and is also required in order to use the LF_i typing judgments, which are defined only on types without placeholders.

6.3 Solving Residual Constraints: $\Gamma \overset{r}{\vdash} C \Rightarrow \Psi$

This judgment describes the process of solving all the type-checking constraints in a list C in an arbitrary order. To accommodate arbitrary orders we introduce the constraint reordering rule. Note that in any particular case a given order is used so that the reconstruction is more effective. However, for the purpose of the reconstruction algorithm we do not care about the particular order used.

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

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

6.4 Type Reconstruction for Objects: $\Gamma \overset{r}{\vdash} M : A \Rightarrow \Psi$

This is the base judgment of the type-reconstruction algorithm. The term M can contain variables (but no placeholder variables) that are defined in Γ and constants that are defined in Σ. M can also contain placeholders and implicit abstractions. However, the restriction is that the type environment Γ and the type A do not contain placeholders. This is again required in order to relate the type reconstruction judgment to the LF_i typing judgment and has the beneficial effect of simplifying the proof of correctness.

Recall that a canonical LF_0 object is either an abstraction with a canonical body or an atomic object. The type-checking judgment deals directly with abstractions and invokes the constraint collecting and solving judgments for atomic objects.

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

Note that the implicit abstraction recovers the argument type from the goal type, which must be a type abstraction. Note also that no reconstruction is allowed under abstraction (the returned substitution must be empty). This restriction does not harm in any way the usefulness of the reconstruction algorithm for first-order logic proofs but simplifies the correctness arguments greatly by eliminating worries about the returned substitution containing the bound variable x.

In the case of a constant, a variable or an application the typing judgment first collects type reconstruction constraints that are then solved in an arbitrary order.

$$\frac{\Gamma \overset{r}{\vdash} M \Rightarrow (\Delta\,;C\,;B) \qquad \Gamma, \Delta \overset{r}{\vdash} C, A \approx_a B \Rightarrow \Psi \qquad \mathit{Dom}(\Delta) \subseteq \mathit{Dom}(\Psi)}{\Gamma \overset{r}{\vdash} M : A \Rightarrow \Psi\big|_{\mathit{Dom}(\Gamma)}}$$

M is not an abstraction

Note in the previous rule that it is required that the substitution returned be defined on all placeholder variables introduced by the current collecting operation. In this respect the presented type reconstruction algorithm is simpler, and potentially less powerful, than constraint based algorithms that allow reconstruction constraints to persist unsolved beyond the place where they were introduced. However, this restriction does not seem to limit the power of the algorithm for reconstructing first-order proof representation but on the other hand eliminates the need for all the machinery for managing persistent constraints.

Note also that the reconstruction algorithm does not check explicitly that the returned substitution is well-typed. However, this property holds because of the design of the unification, which we present next.

6.5 Unification: $M \approx_a M' \Rightarrow \Psi$ and $M \approx M' \Rightarrow \Psi$

The purpose of unification is to check the equivalence of two objects or two types. As a result, the unification constructs a substitution of terms for placeholder variables. Both terms being checked might contain implicit abstractions and might not be in canonical form, but they cannot contain placeholders.

The main restriction of the unification that we present here is that is does not try to unify terms where the placeholder variable is the head of an application. In such a case the resulting substitution would not be uniquely defined and thus we prefer to avoid it. However, this restriction does not seem to harm the power of the reconstruction of first-order logic proofs. In order to express this restriction we use two flavors of unification: atomic and normal. Only the atomic unification can be used for the head of an application and we restrict it so that a placeholder variable is not instantiated by an atomic unification. The unification judgments are presented below.

Atomic Unification

$$\frac{}{c \approx_a c \Rightarrow \cdot} \qquad \frac{}{x \approx_a x \Rightarrow \cdot}$$

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

Normal Unification

$$\frac{M \approx M' \Rightarrow \cdot}{\lambda x.M \approx \lambda x.M' \Rightarrow \cdot} \qquad \frac{M \approx_a M' \Rightarrow \Psi}{M \approx M' \Rightarrow \Psi}$$

$$\frac{u \notin \mathrm{FV}(M)}{u \approx M \Rightarrow u \leftarrow M}$$

The side condition on the instantiation rule is required for correctness. Checking this condition requires scanning the object M that is used to replace the placeholder denoted by the placeholder variable u. We would very much want to avoid this check. Recall that we had two arguments in favor of the implicit representation: smaller representation size and faster checking based on the fact that the reconstructed objects, such as M here, do not require checking because they are part of the trusted type. As expressed by the unification judgment the reconstructed fragments do not require checking but they require scanning. However, in many cases that occur in checking proofs of first-order logics the occurs check is not necessary. We view the removal of the occurs check as an optimization and we discuss it in Section 9.3.

6.6 Why does type reconstruction work?

The five reconstruction judgments described in the previous sections can be used directly as a type reconstruction algorithm. The process of reconstructing and typechecking an object M proceeds as follows:

- If M is an abstraction then use the abstraction rule and continue with the reconstruction of the body.
- Otherwise, M must be a variable or a constant applied to zero or more arguments. Use the collecting judgment to scan the arguments and replace the implicit ones with placeholder variables and collect the explicit arguments in a list of type reconstruction constraints. Add to the list of constraints the unification of the required type for the application and the type computed based on the type of the application head and arguments.
- Solve the list of constraints in a convenient order. As a result, return an instantiation for some placeholder variables.
- Verify that all local placeholders have an instantiation.

The above procedure can be implemented directly as described by the inference rule describing the reconstruction judgments. If all we are concerned is validating implicit representation of first-order proof we can further simplify and optimize the algorithm, as shown in Section 9.3. Before we discuss these optimization we prove the correctness of the reconstruction algorithm. Corollary 6.1 relates the reconstruction judgments to the implicit typing judgment. This is a sufficient correctness criterion for checking proofs because of Theorem 5.1, which in turn relates the LF_i typing judgment to the LF typing judgment shown to be adequate.

Corollary 6.1 (Correctness of proof reconstruction) *If M is an LF_i object such that* $\mathrm{PVF}(M)$ *and* $\cdot \overset{r}{\vdash} M : \mathtt{pf}\ \ulcorner P \urcorner \Rightarrow \cdot$ *then* $\cdot \overset{i}{\vdash} M : \mathtt{pf}\ \ulcorner P \urcorner$.

Unfortunately, the proof of Corollary 6.1 is nowhere as simple as its statement and therefore we devote the entire next section to it. Section 8 is necessarily very technical but the reader can safely skip Section 8 on the first reading and go to Section 9.3 where several optimizations of type reconstruction are discussed.

7 Algorithms for Implicit Representation

Section 6 presents a type reconstruction algorithm that is able to typecheck LF terms containing placeholders. It is obvious that not all subterms can be reconstructed by our algorithm and therefore not all implicit representations of a term can be typechecked using it. In this section we define three algorithms that can be used to eliminate redundant copies of subterms in such a way that the resulting implicit representation can be typechecked using our reconstruction algorithm.

In order to simplify the discussion of the representation algorithms we fix the order of constraint solving such that the type unification is always performed first followed by the typing constraints in the reversed order of their collection. Formally, we restrict our attention to an implementation of reconstruction where the constraint reordering rule is not used. This order is suited for the vast majority of reconstruction tasks. We shall discuss the few exceptions at the end of this section.

We present the implicit representation algorithms as erasure procedures, converting the full LF representations to implicit representations. The simplest erasure algorithms presented below translate easily to representation function defined as mappings from proofs to LF_i terms, following the model of the full LF representation function presented in Section 3.

We consider three algorithms with varying complexity and effectiveness. The output of each of these algorithms, as well as the fully-explicit representation, are equally well suited for typechecking using the reconstruction algorithm presented in Section 6. The only difference is the size of the proof representation and the running time of the reconstruction. The quantitative analysis of the erasure effectiveness is presented in Section 11.

The first algorithm is completely local in the sense that the erasure of a term does not depend on the context where the term occurs. Then we improve the representation by recording a one-bit information about the enclosing context. This is equivalent to having two erasure algorithms working in tandem. The last algorithm is fully global and it attempts to optimize the size of the representation by analyzing the entire enclosing context.

We conclude this section with a brief discussion of the situations where each of the presented algorithms has practical value and when a combination of them makes sense.

7.1 A Local Algorithm

We start with a local erasure algorithm that defines the erasure of an LF term independently of its surrounding context. This algorithm can be viewed as an alternative mapping from proofs to LF_i terms, to be used instead of the full representation function shown in Section 3.

The local erasure algorithm is specified as a set of representation recipes, one for each constant in the signature. We write $R(c)$ to denote the representation recipe of the constant c. A representation recipe is a sequence of representation characters, each corresponding to an argument to which the constant is applied. The length of the representation recipe is dictated by the number of arguments that the constant can take. For the local algorithm we consider only two representation characters, $*$ and e. Each argument that corresponds to an $*$ representation character is replaced by a placeholder and each argument that corresponds to an e is replaced with its implicit representation.

For example, the representation recipe associated with the constant `and_i` is $**ee$, written $R(\texttt{and\_i}) = **ee$. This means that the implicit representation of the full LF term `and_i` $M_1\ M_2\ M_3\ M_4$ is `and_i` $*\ *\ M_3'\ M_4'$, where M_3' and M_4' are the implicit representations of M_3 and M_4 respectively.

Before we consider the details of computing representation recipes we show the definition of the local implicit representation as a function $M \xrightarrow{i} M'$. We shall use the auxiliary function

$M \xrightarrow{i} M' + R$ to say that the implicit representation of the application M is M' and the recipe for the remaining arguments of M is R. We also use the notation e^n to denote a fully-explicit representation recipe, which is used for the applications whose head is a variable.

$$\frac{M \xrightarrow{i} M'}{\lambda x{:}A.M \xrightarrow{i} \lambda x.M'} \qquad \frac{M \xrightarrow{i} M' + R}{M \xrightarrow{i} M'}$$

$$\frac{}{c \xrightarrow{i} c + R(c)} \qquad \frac{}{x \xrightarrow{i} x + e^n} \qquad \frac{M_1 \xrightarrow{i} M_1' + * R}{M_1\ M_2 \xrightarrow{i} M_1'\ * + R} \qquad \frac{M_1 \xrightarrow{i} M_1' + e\,R \qquad M_2 \xrightarrow{i} M_2'}{M_1\ M_2 \xrightarrow{i} M_1'\ M_2' + R}$$

Note that as part of the local erasure, the type of bound variables is removed in addition to those terms specified by the representation recipes.

The only missing detail of the local erasure algorithm is the representation recipe function. Recall that during the type reconstruction the constraint solving judgment must instantiate all local placeholder variables. Recall also that we are considering an implementation of type reconstruction that always solves first the unification judgment from a constraint list. For the local erasure algorithm, we impose the simple restriction that all the local placeholder variables are instantiated while solving the unification constraint. In this is true then, whenever a typing judgment is invoked as part of constraint solving, the type does not contain placeholder variables or placeholders. This invariant is clearly established initially because the top-level type cannot contain free variables and is preserved by constraint solving because the unification is always performed first.

In order to maintain the above mentioned restriction we must only introduce placeholder variables that are going to be instantiated while unifying the application result type with another type without placeholder variables. This determines which arguments in an application can be placeholders.

If we view the constants in the signature as representations of proof rules, then the local erasure algorithm says that in each rule we must explicitly specify at least the instantiations of logical variables that could not be recovered from the conclusion of the rule.

Before we give a general recipe function we discuss the cases of several constants. Consider first the constant `+` with the following type:

$$+ \quad : \quad \texttt{exp} \to \texttt{exp} \to \texttt{exp}$$

It is obvious that no placeholder variable can be instantiated as part of the unification of the result type (`exp`) with another type. Therefore, no implicit arguments are allowed for the constant `+`. This is denoted by the representation recipe $R(+) = e\,e$. Following the same line of reasoning we infer that the representation of expressions and predicates must be fully-explicit.

The next constant we consider is `and_i` with the type:

$$\texttt{and\_i} \quad : \quad \Pi p{:}\texttt{pred}.\Pi r{:}\texttt{pred}.\texttt{pf}\ p \to \texttt{pf}\ r \to \texttt{pf}\ (\texttt{and}\ p\ \ r)$$

If we unify the type $\texttt{pf}\ (\texttt{and}\ u_p\ \ u_r)$ with another type, and if the unification succeeds, then we obtain instantiations for u_p and for u_r and for nothing else. This suggests that the only arguments of `and_i` that can be implicit are the first two. This is denoted by the recipe $R(\texttt{and\_i}) = *\,*\,e\,e$.

Next we consider the case of the constant `and_er` with the type:

$$\texttt{and\_er} \quad : \quad \Pi p{:}\texttt{pred}.\Pi r{:}\texttt{pred}.\texttt{pf}\ (\texttt{and}\ p\ \ r) \to \texttt{pf}\ r$$

For the same reasons as above we see that only the argument r can be instantiated by solving the unification constraint, hence the representation recipe $R(\texttt{and\_er}) = e * e\, e$. Note that even when the argument p could be recovered by type inference on the third argument, the local erasure algorithm requires it to be explicit. It is in cases like this where more sophisticated erasure algorithms, such as the ones showed next, achieve smaller representations.

We conclude our examples with a slightly trickier case. Consider the constant `all_e` with the type:

$$\texttt{all\_e} \;:\; \Pi p\!:\!\texttt{exp} \to \texttt{pred}.\Pi e\!:\!\texttt{exp}.\texttt{pf}\ (\texttt{all}\ \ p) \to \texttt{pf}\ (p\ e)$$

In this case the type $\texttt{pf}\ (u_p\ u_e)$ does not unify with any another type because the placeholder variable u_p is in an application head position. Therefore at least the predicate p must be given explicitly in order to perform the unification. If this is the case, we consider unifying the type $\texttt{pf}\ (P\ u_e)$ for some P. We cannot assume that u_e is instantiated by this unification because, for example, u_e might not occur at all in the normal form of $(P\ u_e)$. Because of this, the expression e must also given explicitly. This justifies the all-explicit representation recipe $R(\texttt{all\_e}) = e\, e\, e$. Note that in a signature representing first-order logic there are only few cases where similar situations occur, namely the constants related to universal and existential quantification.

We could continue the case analysis for all the constants from Σ. Instead, we prefer to develop a general algorithm for computing the representation recipes.

As we have seen in the informal computation of representation recipes, it is important to know the set of placeholder variables that are certainly instantiated during the unification of a term. For this purpose we define a function *Inst* that computes the set of "instantiable" variables of a term.

The definition of *Inst* follows the structure of the unification judgment and has an atomic and a normal variant. We write $\mathit{Inst}(M) = S$ to mean that the variables in S are certainly instantiated by the successful normal unification of M with an LF term without placeholder variables. The atomic variant of the instantiation function is written $\mathit{Inst}_a(M) = S + \beta$, where β is an output parameter specifying whether the head of M is a placeholder variable. These judgments are defined by the following rules:

$$\frac{}{\mathit{Inst}(\lambda x\!:\!A.M) = \emptyset} \qquad \frac{}{\mathit{Inst}(u) = \{u\}} \qquad \frac{\mathit{Inst}_a(M) = S + \beta}{\mathit{Inst}(M) = S} \qquad \frac{\mathit{Inst}_a(M_1) = \emptyset + \mathbf{true}}{\mathit{Inst}_a(M_1\ M_2) = \emptyset + \mathbf{true}}$$

$$\frac{}{\mathit{Inst}_a(u) = \emptyset + \mathbf{true}} \qquad \frac{}{\mathit{Inst}_a(c) = \emptyset + \mathbf{false}} \qquad \frac{\mathit{Inst}_a(M_1) = S_1 + \mathbf{false} \qquad \mathit{Inst}(M_2) = S_2}{\mathit{Inst}_a(M_1\ M_2) = S_1 \cup S_2 + \mathbf{false}}$$

Note in the above rules that no variable is instantiated during the unification of an abstraction. Also, if the head of an application is a variable then the atomic instantiation function returns $\beta = \mathbf{true}$ and, as in the case of `all_e` discussed above, we must assume that no variable is instantiated as part of the unification. The precise relation between the instantiation judgments and the unification is described by the following theorem:

Theorem 7.1 *If* $\Psi_0(M) \approx_a N$ *and* $\mathrm{PVF}(\Psi_0)$ *and* $\mathrm{PVF}(N)$ *then* $\mathit{Inst}_a(M) \subseteq \mathit{Dom}(\Psi_0) \cup \mathit{Dom}(\Psi)$.

This theorem says that all the variables in $\mathit{Inst}_a(M)$ that are not already instantiated (by Ψ_0) are instantiated during unification with a type without placeholder variables. The theorem has an easy proof by induction on the structure of M.

It is convenient to define the complementary function $\mathit{NonInst}(A) = \mathrm{FV}(A) \backslash \mathit{Inst}_a(A)$, to denote the set of variables that are not necessarily instantiated as part of the unification of A.

Next we define the representation recipe function using the helper judgment $R_A(A) = r + S$, where A is a type, r is partial a representation recipe and S is a set of placeholder variables that are guaranteed to be instantiated while unifying the result part of A. The representation recipe of a constant c is computed based on its type $\Sigma(c)$ as follows:

$$R(c) = r \qquad \text{if } R_A(\Sigma(c)) = r + S$$

The R_A judgment is defined by the following rules:

$$\frac{R_A([u/x]B) = r + S \qquad u \in S}{R_A(\Pi x\!:\!A.B) = *\, r + S} \qquad \frac{R_A([u/x]B) = r + S \qquad u \notin S}{R_A(\Pi x\!:\!A.B) = e\, r + S}$$

$$\frac{A \ \ \textit{is atomic}}{R_A(A) = \cdot + \textit{Inst}(A)}$$

The R_A judgment scans the type of a constant, starting from the result type. It computes the set of instantiable variables of the result type and then works backward adding to the representation recipe a $*$ character if the corresponding argument is instantiated or an e character otherwise.

This completes the definition of the representation recipe function and with it the description of the local erasure algorithm. We present next a more sophisticated erasure algorithm that works in a similar way.

7.2 The One-bit Global Algorithm

The local erasure algorithm presented above has the advantage of simplicity and context independence. However, there are practical cases when its result is not satisfactory. Consider the fully-explicit term representing a proof of $P_1 \supset P_2$:

$$\texttt{impl\_i}\ P_1\ P_2\ (\lambda x\!:\!\texttt{pf}\ P_1.\texttt{impl\_e}\ P_1\ P_2\ D_1\ x) : \texttt{pf}\ (\texttt{impl}\ P_1\ P_2)$$

assuming that $D_1 : \texttt{pf}\ (\texttt{impl}\ P_1\ P_2)$ for some predicate representations P_1 and P_2. The local representation recipe for the two proof constants involved (defined in Figure 8 on page 9) are $R(\texttt{impl\_i}) = **e$ and $R(\texttt{impl\_e}) = e*ee$. Therefore the local erasure of the above term is:

$$\texttt{impl\_i} * * (\lambda x.\texttt{impl\_e}\ P_1 * D_1'\ x) : \texttt{pf}\ (\texttt{impl}\ P_1\ P_2)$$

assuming that D_1' is the local erasure of D_1. We immediately see that the first argument to `impl_e` could be reconstructed from the context in which it appears, for example from the fact that a representation of its proof is x of type `pf` P_1. This example is very simple and the potential savings are small. However, similar situations occur whenever there are proof rules for which some of the logical parameters cannot be recovered from the conclusion alone but from the surrounding context. This is the case with conjunction and implication elimination rules and many arithmetic rules such as the transitivity of equality.

The one-bit global erasure algorithm uses only one bit of information about the context or more precisely about the circumstances in which the type reconstruction will be invoked on the term. This bit of information says whether, at the time of type reconstruction, the type can contain placeholder variables or not. If the type cannot contain variables we say that the term must be represented for typechecking. Otherwise we say that it must be represented for type inference, and in such a way that type reconstruction on that term alone *must* succeed in finding an instantiation for all placeholder variables that appear in the type.

We express the one-bit global erasure algorithm in terms of representation recipes, in a similar manner as the local algorithm. We replace the e representation character with two separate characters e_c (for checking) and e_i (for inference). Due to the fact that now the erasure function uses one-bit of global information (whether it is checking or inference) we change the erasure function to $M \stackrel{i_p}{\to} M'$ where p can be either c or i. The only erasure rules that are changed are those for constants and for explicit application:

$$\frac{M_1 \stackrel{i_p}{\to} M_1' + e_c\, R \qquad M_2 \stackrel{i_c}{\to} M_2'}{M_1\ M_2 \stackrel{i_p}{\to} M_1'\ M_2' + R} \qquad \frac{M_1 \stackrel{i_p}{\to} M_1' + e_i\, R \qquad M_2 \stackrel{i_i}{\to} M_2'}{M_1\ M_2 \stackrel{i_p}{\to} M_1'\ M_2' + R} \qquad \frac{}{c \stackrel{i_p}{\to} c + R_p(c)}$$

The one-bit representation recipes now determine not only whether an argument must be implicit or not, but also whether the explicit arguments must be represented for checking or for inference. At this point we have reduced the erasure to the static computation of the representation recipes R_i and R_c for each constant in the signature. These functions are computed, as in the local case, using an auxiliary judgment $R^p_A(A) = r + S_i + S_e$. In addition to the set of variables S_i that are instantiated by constraint solving, the new judgment also contains a set of placeholder variables S_e that must be explicit. This judgment is defined by the following rules:

$$\frac{A \ \ \mathit{is\ atomic}}{R^c_A(A) = \cdot + \mathit{Inst}(A) + \mathit{NonInst}(A)} \qquad \frac{A \ \ \mathit{is\ atomic}}{R^i_A(A) = \cdot + \emptyset + \mathit{NonInst}(A)}$$

$$\frac{R^p_A([u/x]B) = r + S_i + S_e \qquad u \in S_i}{R^p_A(\Pi x\!:\!A.B) = *\, r + S_i + S_e}$$

$$\frac{R^p_A([u/x]B) = r + S_i + S_e \qquad u \notin S_i \qquad \mathit{Inst}(A) \subseteq S_i \cup S_e}{R^p_A(\Pi x\!:\!A.B) = e_c\, r + S_i + S_e \cup (\mathit{NonInst}(A) \backslash S_i)}$$

$$\frac{R^p_A([u/x]B) = r + S_i + S_e \qquad u \notin S_i \qquad \mathit{Inst}(A) \not\subseteq S_i \cup S_e}{R^p_A(\Pi x\!:\!A.B) = e_i\, r + S_i \cup (\mathit{Inst}(A) \backslash S_e) + S_e \cup (\mathit{NonInst}(A) \backslash S_i)}$$

The definition of R^p_A deserves some explanations. First, recall the assumption that the constraints are solved in the reverse order of introduction, starting with the unification constraint. In the case of checking, the unification is with a type without placeholder variables. The result is an instantiation for all instantiable placeholder variables (therefore $S_i = \mathit{Inst}(A)$) provided that all non-instantiable variables (collected in S_e) are explicit. This justifies the rule for $R^c_A(A)$ when A is atomic. In the case of inference the same unification happens but this time with a type that might contain placeholder variables. We have to assume conservatively that no placeholder variable is instantiated to a fully-explicit term, hence $S_i = \emptyset$. However, we also require that all non-instantiable variables are explicit as in the case of checking. This explains the rule for $R^i_A(A)$ when A is atomic. In case of $R^p_A(\Pi x : A.B)$, because of the constraint solving order we first deal with the arguments corresponding to B. If u is instantiated after dealing with B then it can be left implicit, hence the rule for $*\, r$. Otherwise, the argument corresponding to u must be left explicit and depending on whether the variables of A are all instantiated at this point we decide whether to use checking or inference on it. The set S_e is collecting the set of placeholder variables that cannot be instantiated because they occur in function position in some type. Note that the sets S_i and S_e are always disjoint.

A case where S_e is non-empty and is essential for the correctness of the erasure algorithm is for `all_e` of type:

$$\texttt{all\_e} \;:\; \Pi p\!:\!\texttt{exp} \to \texttt{pred}.\Pi e\!:\!\texttt{exp}.\texttt{pf}\ (\texttt{all}\ \ p) \to \texttt{pf}\ (p\ e)$$

In this case $p \in \mathit{NonInst}(\texttt{pf}\ (p\ e))$ but $p \in \mathit{Inst}(\texttt{pf}\ (\texttt{all}\ \ p))$. If we did not maintain the list S_e we could have the impression that p can be reconstructed by type inference on the term of type `pf` $(\texttt{all}\ \ p)$ without noticing that the unification constraint would have failed earlier because of a placeholder occurring in an application head position.

Using the function R^p_A we can define the representation recipes as in the local algorithm:

$$R_p(c) = r \qquad \mathit{if}\ \ R^p_A(\Sigma(c)) = r + S_i + S_e$$

In the table below we show examples of representation recipes as produced by the local algorithm and the one-bit representation algorithm. We notice that in the case of `+`, and similarly for all expression and predicate constants, all arguments must be explicit in both the local and the one-bit representation recipes. Also, in such cases, the inference recipe and should never be required during erasure. The next four cases shown in the table are examples where the one-bit global algorithm is able to remove more arguments than the local erasure algorithm.

c	$R(c)$	$R_c(c)$	$R_i(c)$
$+$	$e\,e$	$e_c\,e_c$	$e_c\,e_c$
`and_el`	$*\,e\,e$	$*\,*\,e_i$	$*\,*\,e_i$
`and_er`	$e\,*\,e$	$*\,*\,e_i$	$*\,*\,e_i$
`impl_e`	$e\,*\,e\,e$	$*\,*\,e_c\,e_i$	$*\,*\,e_i\,e_i$
$=$ `tr`	$*\,e\,*\,e\,e$	$*\,*\,*\,e_c\,e_i$	$*\,*\,*\,e_i\,e_i$
$+$`com`	$*\,*$	$*\,*$	$e_c\,e_c$
`all_e`	$e\,e\,e$	$e_c\,e_c\,e_c$	$e_c\,e_c\,e_c$

Table 2: Examples of representation recipes computed by the local and one-bit global algorithms.

In order to analyze the relation between the local and the one-bit recipes we remark that the local representation function is a special case of the one-bit representation recipes. The main difference is that the set S_e in the local recipe function is always the negation of the set S_i with respect to the universe of variables. This only needs to be enforced in the rule for R^c_A in the one-bit function in order to recover the local recipe function.

With the above remark we see that the size of set S_i in the local recipe function never increases beyond the initial size established for the result type. And as the size of S_i determines the maximum number of implicit arguments we conclude that the local erasure algorithm will always remove fewer arguments that the one-bit global algorithm for every particular application. This conclusion is supported by the examples shown in Table 2.

Furthermore we notice in Table 2 that $R_c(c)$ has more implicit arguments than $R_i(c)$ and also more e_i characters corresponding to the explicit arguments. This can be explained informally by remembering that there are additional constraints when representing a term for inference instead of just checking. In the case of the representation for inference, the term must contain enough subterms to allow the reconstruction of its type, while in the case of checking more subterms can be omitted and later reconstructed from the fully-explicit type.

We can characterize the one-bit global erasure algorithm as eliminating some explicit arguments remaining after the local erasure and redistributing the information required for their reconstruction to the subterms corresponding to sibling arguments, by marking them for inference. The additional information required for inference is more apparent at the leaves of the proof tree. Here there are no siblings to offload to information and hence the inference erasure is not able to erase any subterms while the checking erasure can remove all of them. This is illustrated by the case of `+com`. Even though the cost in the one-bit representation is pushed towards the leaves, the resulting representation is in general smaller than the local representation because part of the size of the erased argument is reconstructed from the structure of the proof of sibling arguments, and only a small part from explicit arguments at the leaves.

7.3 The Global Algorithm

The local erasure algorithm is able to recover some arguments of the representation of a proof-rule instance from the conclusion of the proof rule. Then in the one-bit algorithm we increase the reconstruction power by recovering additional arguments from the structure of some of the hypotheses of the proof-rule. In this section we present an even better algorithm that recovers additional arguments by examining the entire context in which a proof-rule is used. Also, the global erasure algorithm attempts to overcome another limitation of the previous algorithms, namely the loss of information due to the static conservative approximation of non-instantiable variables. The global algorithm considers each constant with its actual parameters and can therefore minimize the set of non-instantiable variables.

The basic idea of the global algorithm is to start by assuming that all arguments of the representation of a proof-rule instance are implicit and then simulate type reconstruction. Whenever an argument cannot be reconstructed it is made explicit and its implicit representation is made part of the final representation.

Because the global erasure algorithm works by simulating type reconstruction, it has the same overall structure. Unfortunately there are enough minor variations to require a new implementation. This make the global erasure algorithm expensive to implement.

The first change we have to make is to the unification judgments to consider the case of a placeholder variable that is the head of an application. In this case the placeholder variable must be instantiated with the term from the full representation that it replaces. For this to be possible the unification judgment must take a parameter Ψ_0 containing the instantiations for all placeholder variables that would produce the original fully-explicit term. Also the unification must return, in addition to the usual substitution, another substitution, called the explicit substitution, containing instantiations for the non-instantiable variables. We use the following notation to denote the new variant of unification $\Psi_0 \vdash A \approx_a B \Rightarrow \Psi + \Psi_e$. Similarly we add a new substitution argument and an additional substitution result to the main typing judgment. The resulting judgment is written $\Gamma + \Psi_0 \vdash M : A \Rightarrow \Psi + \Psi_e$. To this we add one additional result M', the resulting implicit representation of M, and we write the resulting judgment in functional style $GErase(\Gamma, \Psi_0, M, A) = (\Psi, \Psi_e, M')$. The global erasure algorithm is given informally in Figure 15 and a step-by-step explanation is made below. A formal description of the algorithm would follow the structure of the type reconstruction judgments.

In the case of an abstraction the erasure is done recursively on the body of the abstraction and the type of the bound variable is omitted from the resulting representation.

In the case of a constant application there is an opportunity to leave some of the arguments implicit. We start in step 2(a) by introducing a placeholder variable for each argument, trying therefore to maximize the number of implicit terms. In step 2(b) we record the original values of the newly introduced placeholder variables in the substitution Ψ_0.

$GErase(\Gamma, \Psi_0, M, A) =$

1. *If* $M = \lambda x\!:\!A_1.M_1$ *and* $A = \Pi x\!:\!A_1.A_2$ *with* $GErase(\Gamma, x : A_1, \Psi_0, M_1, A_2) = (\cdot, \Psi_e, M_1')$ *then the result is* $(\cdot, \Psi_e, \lambda x.M_1')$
2. *If* $M = c\, M_1 \ldots M_n$ *and* $\Sigma(c) = \Pi x_1\!:\!A_1.\ldots\Pi x_n\!:\!A_n.A_{n+1}$ *then*
   (a) *Let* $\Delta = u_1 : A_1, u_2 : [u_1/x_1]A_2, \ldots, u_n : [u_{n-1}/x_{n-1}]\ldots[u_1/x_1]A_n$, *and*
   (b) *Let* $\Psi_0 = \Psi_0, u_1 \leftarrow M_1, \ldots, u_n \leftarrow M_n$, *and*
   (c) *Let* $\Psi_0 \vdash A \approx_a [u_n/x_n]\ldots[u_1/x_1]A_{n+1} \Rightarrow \Psi + \Psi_e$
3. *Let* $\Delta = \Psi \circ \Psi_e(\Delta)$
4. *If* $\Delta = \cdot$, *then return with* $(\Psi\big|_{Dom(\Gamma)}, \Psi_e\big|_{Dom(\Gamma)}, c\, M_1' \ldots M_n')$ *where*

$$M_i' = \begin{cases} \Psi_e(u_i) & \textit{if} \;\; u_i \in Dom(\Psi_e) \\ * & \textit{otherwise} \end{cases}$$

5. (a) *Let* $\Delta = \Delta', u_k : A_k'$, *and*
   (b) *Let* $GErase((\Gamma, \Delta'), \Psi_0, \Psi_0(u_k), A_k') = (\Psi', \Psi_e', M_k')$, *and*
   (c) *Let* $\Psi_e = [u_k \leftarrow M_k'] \circ \Psi_e' \circ \Psi_e$, *and*
   (d) *Let* $\Psi = \Psi' \circ \Psi$, *and*
   (e) *Let* $\Delta = \Delta'$, *and*
   (f) *Go to step 3*

Figure 15: The informal description of the global erasure algorithm.

The purpose of the remaining steps is to build the substitution Ψ_e giving an instantiation for those arguments that cannot possibly be implicit. Recall that constraint solving is done in reverse order, starting with the unification constraint. We start therefore with the unification and in step 2(c) we compute both the set of variables that must be explicit in order for the unification to succeed (Ψ_e) and the set of variables successfully instantiated by the unification.

Then in step 3 we remove these variables from Δ. If no variables remain, then we conclude that all placeholder variables introduced in step 2(a) can be either recovered or must be explicit in order for the constraint list to be solvable. We return with an implicit representation of M' as dictated by Ψ_e.

If there are placeholder variables left in Δ then at least the last one must be explicit. If this were not the case we would have no possibility of recovering it by solving the remaining constraints. In order to compute its implicit representation we recursively invoke the global erasure on the original value of the placeholder variable $\Psi_0(u_k)$. The resulting implicit representation is added to Ψ_e such that it can be inserted in the final result in step 4. However, during the computation of the global erasure of $\Psi_0(u_k)$ we might have instantiated further variables (Ψ') and found variables that must be explicit (Ψ_e'). These variables are removed from Δ in step 3 and we cycle until all the variables in Δ are dealt with.

The global erasure algorithm achieves an optimal representation given a specific constraint solving order. This is the case because, by design, it starts with all arguments implicit and it only reverts an argument to explicit status when the reconstruction would not work otherwise.

7.4 Discussion

In all three erasure algorithms we have assumed a specific order of constraint solving. In our experiments this fixed order performed surprisingly well. However, in selected cases the representation can be made even smaller if a different order is adopted. The largest benefit can be achieved if we allow a different reconstruction order for each application instance.

We have not tried this approach but instead we have selected a few constants for which a non-default order has the potential of decreasing the size of the representation. An example is the constant `all_e` where it is better to typecheck the constraint corresponding to `pf` (`all` p), with the hope of recovering p, before solving the unification constraint on `pf` ($p\ e$), which invariably fails if p is not instantiated. This approach produced such minor improvements that we did not consider it further.

Despite its limited knowledge of the context, the local erasure algorithm produces implicit terms that are close to optimal for most logic proof rules and it might be a good practical choice in situations where absolute optimality is not required. In situations where the size of the proof or the its validation time is crucial we can use one of the global algorithms.

As we mentioned before, the global erasure algorithm is optimal for a given reconstruction order. However it requires as one of its inputs the whole LF term, which might be very large. Another disadvantage surfaces in the cases where the proof is build gradually from small pieces at a time, as is the case in a theorem prover that outputs a proof representation. These small pieces cannot be optimized until the whole proof is constructed.

We show in Section 11 that we din not noticed major improvements in the size of the representation produced by the global algorithm when compared to the one-bit algorithm. This, in conjunction with the disadvantages of the global erasure algorithm, makes us conclude that it is not worth using, or in other words that the simpler one-bit algorithm achieves near optimal results.

We believe that the practical way to build implicit representations of proofs is to use the local algorithm for maintaining small intermediary fragments of proofs. When the whole proof is assembled we can apply the one-bit global erasure algorithm to it. The result will be the same as if we started from the fully-explicit representation but without the expense of building the large intermediary proof object. Another benefit is that the overall running time is usually smaller when running the erasure algorithms in sequence because the more primitive the algorithm the smaller the cost of erasing one subterm.

The correctness criterion for an erasure algorithm is that the type reconstruction algorithm succeeds on the resulting representation. We did not state formal correctness results for the erasure algorithm presented in this section for two main reasons. One is that we motivated each design choice in the erasure algorithms with respect to the type reconstruction algorithm. Also, for practical reasons, the correctness of the erasure algorithm is not crucial. Its result can always be validated by running the type reconstruction algorithm.

8 The Correctness Proof of Type Reconstruction

In this section we prove that the reconstruction algorithm presented in the previous section is adequate for checking the validity of proofs. One of the advantages of developing the proof checking algorithm as a variation on LF typechecking is that we have a well-understood formal framework that can be used for the succinct expression of the correctness criteria and also for proving them.

We want to stress that the proof of correctness is not just an exercise in type-theoretic proofs. Besides the obvious purpose of ensuring the correctness of the type reconstruction algorithm—which is not obvious by inspection only—the correctness proof also constitutes a thorough analysis of the strengths limitations of our algorithm.

The algorithm that we presented in the previous section is limited by the side-conditions present in some of the rules, most notably the collection of explicit parameters, the typing of abstractions and the instantiation rule. The correctness proof can serve as a reference documenting the need for every such side-condition. In many practical cases the reconstruction is used in circumstances when some of the side-conditions do not need to be enforced. These circumstances can be discovered only after a deep understanding of the purpose of the side conditions in the correctness proof.

In Section 10 we show how we can significantly improve the performance of the reconstruction algorithm by removing some of the side conditions in the special case when we are checking first-order logic proofs.

The rest of this section is very technical and can be skipped on first reading. The only reference back to the material in the rest of the section is to justify some of the optimizations that we implemented for the reconstruction.

8.1 Notation and Conventions

We write $\Psi(\Gamma)$ to denote the result of applying the substitution Ψ to a type environment Γ. The resulting type environment is defined on $Dom(\Psi(\Gamma)) = Dom(\Gamma) \backslash Dom(\Psi)$ as follows:

$$\Psi(\Gamma)(x) = \Psi(\Gamma(x)) \qquad\qquad \forall x \in Dom(\Gamma) \backslash Dom(\Psi)$$

Note that the type declarations for the placeholder variables defined by the substitution are removed from the resulting type environment.

We write $\Gamma \vdash \Psi$ to denote that the substitution Ψ is well-typed according to a type environment Γ, using the LF_i typing rules. As a general note, our reference typing system is LF_i. Formally,

$$\Gamma \vdash \Psi \ \text{ iff } \ \forall u \in Dom(\Psi) \ \text{ then } \ u \in Dom(\Gamma) \ \text{ and } \ \Psi(\Gamma) \overset{i}{\vdash} \Psi(u) : \Psi(\Gamma(u))$$

In order to simplify the presentation of the correctness proofs in this section we make the convention that *all the types involved are placeholder-free.* This applies to types given as part of the theorem hypothesis or types mentioned in the conclusion of helper lemmas and theorems. Whenever new types are created we shall check this property but we do not express it explicitly in the statements of the theorems. Because of this convention we can use freely the LF_i typing rules without checking that the PF side-conditions hold.

In order to keep the presentation focused we have segregated a number of helper lemmas in Section 8.7 at the end of this section.

8.2 Correctness of the Type Reconstruction Judgment

The type reconstruction algorithm is expressed as five mutually recursive judgments. Not surprisingly the correctness proof of the reconstruction algorithm consists of five mutually dependent correctness proofs for the constituent judgments. These proofs are by induction on the structure of the reconstruction judgments. Occasionally, a proof invokes the theorem for a related judgment, in the same way as the corresponding judgment invokes a related judgment. For the five proofs at hand the chain of theorem invocations is circular but the derivations involved are structurally smaller, therefore the induction is well-founded.

We start with the correctness theorem about the main reconstruction judgment. We cannot prove the Corollary 6.1 directly. Instead we have to strengthen the statement of the theorem so that the induction goes through. In particular we have to allow for arbitrary typing environments and types. Note that there might be placeholder variables in Γ and A but no placeholders. The correctness theorem is stated formally bellow:

Theorem 8.1 (Correctness of the Type Reconstruction) *If* Γ *and* A *are a type environment and a type respectively such that* $\mathrm{PF}(\Gamma)$ *and* $\mathrm{PF}(A)$ *and* $\Gamma \overset{i}{\vdash} A : \mathtt{Type}$ *and* M *is a term such that* $\mathrm{PVF}(M)$ *and if* $\Gamma \overset{r}{\vdash} M : A \Rightarrow \Psi$ *then*

- $\Gamma \vdash \Psi$, *and*
- $\mathrm{PF}(\Psi)$, *and*
- $\Psi(\Gamma) \overset{i}{\vdash} M : \Psi(A)$.

From the Theorem 8.1 we can immediately prove the Corollary 6.1 for the empty type environment and the empty substitution, if we note that $\mathrm{PF}(\mathtt{pf}\ \ulcorner P \urcorner)$ by the definition of the representation function.

Proof: (of Theorem 8.1) The proof is by induction on the structure of the derivation $\mathcal{D} :: \Gamma \overset{r}{\vdash} M : A \Rightarrow \Psi$. There are two cases, depending on the last rule used in $\mathcal{D}$.

Case: If M is an abstraction:

$$\mathcal{D} = \cfrac{\begin{array}{c}\mathcal{D}_1 \\ \Gamma, x : A \overset{r}{\vdash} M : B \Rightarrow \cdot\end{array}}{\Gamma \overset{r}{\vdash} \lambda x.M : \Pi x\!:\!A.B \Rightarrow \cdot}$$

It is obvious that the empty substitution is well-typed and placeholder-free. Because $\mathrm{PF}(\Gamma)$ and $\mathrm{PF}(\Pi x\!:\!A.B)$ it follows that we can apply the induction hypothesis on $\mathcal{D}_1$ and infer that $\Gamma, x : A \overset{i}{\vdash} M : B$. Therefore we can also infer that $\Gamma \overset{i}{\vdash} \lambda x.M : \Pi x\!:\!A.B$, which is the desired conclusion.

Case: If M is not an abstraction:

$$\mathcal{D} = \cfrac{\begin{array}{c}\mathcal{D}_1 \\ \Gamma \overset{r}{\vdash} M \Rightarrow (\Delta\ ;C\ ;B)\end{array} \qquad \begin{array}{c}\mathcal{D}_2 \\ \Gamma, \Delta \overset{r}{\vdash} C, A \approx_a B \Rightarrow \Psi\end{array} \qquad \mathit{Dom}(\Delta) \subseteq \mathit{Dom}(\Psi)}{\Gamma \overset{r}{\vdash} M : A \Rightarrow \Psi}$$

We follow the sequence of deduction steps shown below:

1. $\Gamma \overset{i}{\vdash} A : \mathtt{Type}$ (hypothesis),
2. $\mathrm{PF}(\Gamma)$ and $\mathrm{PF}(A)$ and $\mathrm{PVF}(M)$ (hypothesis),
3. Using Theorem 8.4 on $\mathcal{D}_2$ we infer that

4. For all $N : A'$ from C there exist Ψ_1 and Ψ_2 such that $\Psi_1(\Gamma, \Delta) \overset{i}{\vdash} \Psi_1(N) : \Psi_1(A') \Rightarrow \Psi_2$
5. With 2, 4 we apply Theorem 8.5 on $\mathcal{D}_1$ (with $\Delta' = \emptyset$) and infer that
6. $\Gamma, \Delta \overset{i}{\vdash} B : \mathtt{Type}$, and
7. PF(Δ), and
8. PF(B), and
9. For all $N : A'$ in C we have PF(A') and PVF(N) and $\Gamma, \Delta \overset{i}{\vdash} A' : \mathtt{Type}$
10. From 1 we infer that $\Gamma, \Delta \overset{i}{\vdash} A : \mathtt{Type}$.
11. With 2, 6, 7, 8, 9 and 10 we apply Theorem 8.6 on $\mathcal{D}_2$ and infer that
12. $\Gamma, \Delta \vdash \Psi$, and
13. $\Psi(A) \equiv_\beta \Psi(B)$, and
14. PF(Ψ), and
15. For all $N : A'$ in C we have $\Psi(\Gamma, \Delta) \overset{i}{\vdash} N : \Psi(A')$. Also using Lemma 8.20 with $Dom(\Delta) \subseteq Dom(\Psi)$ we infer that $\Psi(\Gamma) \overset{i}{\vdash} N : \Psi(A')$.
16. With 9, 12, 14, 15 we apply Theorem 8.7 and infer that
17. $\Psi(\Gamma) \overset{i}{\vdash} M : \Psi(B)$
18. Because all placeholder variables from Δ are new, they cannot occur in Γ or in B. Therefore $\Psi(Dom(\Gamma)) = \Psi\big|_{Dom(\Gamma)}(\Gamma)$ and $\Psi(B) = \Psi\big|_{Dom(\Gamma)}(B)$. Therefore we get one of the desired conclusions.
19. With 12 and using Lemma 8.20 we prove that $\Gamma \vdash \Psi\big|_{Dom(\Gamma)}$. Also from 14 we can easily prove that PF($\Psi\big|_{Dom(\Gamma)}$).

$\square$

8.3 Correctness of Unification

The most important judgments for the type reconstruction algorithm presented in the previous section are the unification judgments. Their properties are crucial for the correctness of the reconstruction and their implementation determine the performance of the reconstruction. The key property of the unification judgments is that the resulting substitution preserves the types of the placeholder variables, and as a consequence the algorithm does not need to typecheck the substitution.

The properties of interest of the unification judgments are expressed in Theorem 8.2. The first two parts of the theorem deal with atomic unification of types and objects respectively. The last part deals with normal unification of objects.

Theorem 8.2 (Correctness of Unification) *All the types mentioned in the statement below are assumed to be placeholder-free.*

(a) *If* $A_1 \approx_a A_2 \Rightarrow \Psi$ *such that* $\Gamma \overset{i}{\vdash} A_1 : K_1$ *and* $\Gamma \overset{i}{\vdash} A_2 : K_2$ *then*

- $\Gamma \vdash \Psi$, *and*
- $\mathrm{PF}(\Psi)$, *and*
- $\Psi(A_1) \equiv_\beta \Psi(A_2)$, *and*
- $\Psi(K_1) \equiv_\beta \Psi(K_2)$

(b) *If* $M_1 \approx_a M_2 \Rightarrow \Psi$ *and* $\mathrm{PF}(M_1)$ *and* $\mathrm{PF}(M_2)$ *such that* $\Gamma \overset{i}{\vdash} M_1 : A_1$ *and* $\Gamma \overset{i}{\vdash} M_2 : A_2$ *then*

- $\Gamma \vdash \Psi$, *and*
- $\mathrm{PF}(\Psi)$, *and*
- $\Psi(M_1) \equiv_\beta \Psi(M_2)$, *and*
- $\Psi(A_1) \equiv_\beta \Psi(A_2)$

(c) *If* $M_1 \approx M_2 \Rightarrow \Psi$ *and* $\mathrm{PF}(M_1)$ *and* $\mathrm{PF}(M_2)$ *such that* $\Gamma \overset{i}{\vdash} M_1 : A$ *and* $\Gamma \overset{i}{\vdash} M_2 : A$ *then*

- $\Gamma \vdash \Psi$, *and*
- $\mathrm{PF}(\Psi)$, *and*
- $\Psi(M_1) \equiv_\beta \Psi(M_2)$

The complicated statement above is required in order to prove the theorem by induction. All the actual uses of the Theorem 8.2 are in the form of a much simpler corollary shown below. The corollary follows immediately from the case (a) of Theorem 8.2 by taking $K_1 = K_2 = \mathtt{Type}$.

Corollary 8.3 *If* $A_1 \approx_a A_2 \Rightarrow \Psi$ *such that* $\Gamma \overset{i}{\vdash} A_1 : \mathtt{Type}$ *and* $\Gamma \overset{i}{\vdash} A_2 : \mathtt{Type}$ *then*

1. $\Gamma \vdash \Psi$, *and*
2. $\mathrm{PF}(\Psi)$, *and*
3. $\Psi(A_1) \equiv_\beta \Psi(A_2)$.

Proof: (of Theorem 8.2) The proof is by induction on the structure of the unification derivations. We only show here the cases for the unification of objects (cases b and c). The proof for atomic unification of types follows the same patterns.
Case:

$$\mathcal{D} = \frac{}{c \approx_a c \Rightarrow \cdot}$$

The empty substitution is well-typed in any environment and is also placeholder-free. By hypothesis we know that $\Gamma \overset{i}{\vdash} c : A_1$ and $\Gamma \overset{i}{\vdash} c : A_2$. From Lemma 8.14 we conclude that $A_1 \equiv_\beta A_2$. The rest of the conclusion follows immediately.
Case:

$$\mathcal{D} = \frac{}{x \approx_a x \Rightarrow \cdot}$$

Again, the typing condition on the resulting substitution is vacuously true. By hypothesis we know that $\Gamma \overset{i}{\vdash} x : A_1$ and $\Gamma \overset{i}{\vdash} x : A_2$. From Lemma 8.14 we infer that $A_1 \equiv_\beta A_2$, which concludes the proof of this case.

Case:

$$\mathcal{D} = \cfrac{\begin{array}{c}\mathcal{D}_1\\ [N/x]M \approx_a M' \Rightarrow \Psi\end{array}}{(\lambda x.M)\ N \approx_a M' \Rightarrow \Psi}$$

For the purpose of the correctness proof we only consider the β-reduction case with only one argument ($n = 1$). The general case is proved in a similar way.
By hypothesis we have that $\Gamma \overset{i}{\models} (\lambda x.M)\ N : A_1$. From Lemma 8.14 we infer that $\Gamma \overset{i}{\models} \lambda x.M : \Pi x : A.B$ and $\Gamma \overset{i}{\models} N : A$ and $[N/x]B \equiv_\beta A_1$. Because $\mathrm{PF}((\lambda x.M)\ N)$ we infer that $\mathrm{PF}(\lambda x.M)$ and $\mathrm{PF}(N)$. Therefore $\mathrm{PF}([N/x]M)$. Now we can use the Lemma 8.15 to infer that $\Gamma \overset{i}{\models} [N/x]M : [N/x]B$ and therefore $\Gamma \overset{i}{\models} [N/x]M : A_1$. Now we apply the induction hypothesis on $\mathcal{D}_1$ and infer that $\Gamma \vdash \Psi$ and $\Psi(A_1) \equiv_\beta \Psi(A_2)$ and that $\mathrm{PF}(\Psi)$.
We also conclude from the induction hypothesis that $\Psi([N/x]M) \equiv_\beta \Psi(M')$. But $\Psi([N/x]M) = [\Psi(N)/x]\Psi(M) \equiv_\beta \Psi((\lambda x.M)N)$, which completes the proof of this case.
Case:

$$\mathcal{D} = \cfrac{\begin{array}{c}\mathcal{D}_1\\ M \approx_a M' \Rightarrow \Psi\end{array} \qquad \begin{array}{c}\mathcal{D}_2\\ \Psi(N) \approx \Psi(N') \Rightarrow \Psi'\end{array}}{M\ N \approx_a M'\ N' \Rightarrow \Psi' \circ \Psi}$$

We follow a sequence of deductions:

1. From $\Gamma \overset{i}{\models} M\ N : A_1$ (hypothesis) and Lemma 8.14 we get
2. $\Gamma \overset{i}{\models} M : \Pi x\!:\!A.B$, and
3. $\Gamma \overset{i}{\models} N : A$, and
4. $[N/x]B \equiv_\beta A_1$.
5. From $\Gamma \overset{i}{\models} M'\ N' : A_2$ (hypothesis) and Lemma 8.14 we get
6. $\Gamma \overset{i}{\models} M' : \Pi x\!:\!A'.B'$, and
7. $\Gamma \overset{i}{\models} N' : A'$, and
8. $[N'/x]B' \equiv_\beta A_2$.
9. Using 2 and 6 we apply the induction hypothesis on $\mathcal{D}_1$. We conclude:
10. $\Gamma \vdash \Psi$, and
11. $\mathrm{PF}(\Psi)$, and
12. $\Psi(\Pi x\!:\!A.B) \equiv_\beta \Psi(\Pi x\!:\!A'.B')$, and
13. $\Psi(M) \equiv_\beta \Psi(M')$.
14. With 3, 10 and 11 we use Corollary 8.17 and deduce that $\Psi(\Gamma) \overset{i}{\models} \Psi(N) : \Psi(A)$.
15. With 7, 10 and 11 we use Corollary 8.17 and deduce that $\Psi(\Gamma) \overset{i}{\models} \Psi(N') : \Psi(A')$, and using 12 we also deduce that $\Psi(\Gamma) \overset{i}{\models} \Psi(N') : \Psi(A)$.
16. From 11 we infer that $\mathrm{PF}(\Psi(N))$ and $\mathrm{PF}(\Psi(N'))$. This together with 14 and 15 let us apply the induction hypothesis on $\mathcal{D}_2$. We deduce that:

17. $\Psi(\Gamma) \vdash \Psi'$, and

18. $\mathrm{PF}(\Psi')$, and

19. $\Psi'(\Psi(N)) \equiv_\beta \Psi'(\Psi(N'))$.

20. From 10, 11, 17 and 18 with Lemma 8.19 we get the first part of the conclusion: $\Gamma \vdash \Psi' \circ \Psi$ and $\mathrm{PF}(\Psi' \circ \Psi)$.

21. From 13 and using Lemma 8.21 we deduce that $\Psi' \circ \Psi(M) \equiv_\beta \Psi' \circ \Psi(M')$. This and 19 allow us to prove more of the conclusion: $\Psi' \circ \Psi(M\ N) \equiv_\beta \Psi' \circ \Psi(M'\ N')$.

22. From 4 and using Lemma 8.21 we deduce that $\Psi' \circ \Psi(A_1) \equiv_\beta [\Psi' \circ \Psi(N)/x]\Psi' \circ \Psi(B)$. Similarly from 8 we get that $\Psi' \circ \Psi(A_2) \equiv_\beta [\Psi' \circ \Psi(N')/x]\Psi' \circ \Psi(B')$. Now we can use Lemma 8.22 with 12 and 19 to get the last part of the conclusion: $\Psi' \circ \Psi(A_1) \equiv_\beta \Psi' \circ \Psi(A_2)$.

This concludes all the cases regarding the atomic unification. The rest of the cases are for normal unification. The only interesting cases here are the abstraction and instantiation cases.
Case:

$$\mathcal{D} = \frac{\begin{array}{c}\mathcal{D}_1 \\ M \approx M' \Rightarrow \cdot\end{array}}{\lambda x.M \approx \lambda x.M' \Rightarrow \cdot}$$

The empty substitution is well-typed and also placeholder-free. By hypothesis $\Gamma \overset{i}{\vdash} \lambda x.M : A$ and $\Gamma \overset{i}{\vdash} \lambda x.M' : A$. From Lemma 8.14 we have that $A = \Pi x : A_1.A_2$ and $\Gamma, x : A_1 \overset{i}{\vdash} M : A_2$ and also $\Gamma, x : A_1 \overset{i}{\vdash} M' : A_2$. Now we can use the induction hypothesis on $\mathcal{D}_1$ and infer the required conclusion: $M \equiv_\beta M'$.
Case:

$$\mathcal{D} = \frac{u \notin \mathrm{FV}(M)}{u \approx M \Rightarrow u \leftarrow M}$$

Let $\Psi = u \leftarrow M$. By hypothesis we have that $\Gamma \overset{i}{\vdash} u : A$ and $\Gamma \overset{i}{\vdash} M : A$ and $\mathrm{PF}(M)$. From Corollary 8.18 we deduce that $\Gamma \vdash \Psi$. Because $\mathrm{PF}(M)$ we can also infer that $\mathrm{PF}(\Psi)$. The rest of the conclusion is trivial: $\Psi(u) \equiv_\beta \Psi(M)$ because $\Psi(M) = M$.

□

8.4 Correctness of Constraint Collection and Constraint Solving

Due to the presence of dependent types there is a very close relationship between constraint collection and constraint solving. This is mainly because the properties for constraint solving are only defined if the type involved is well-typed of kind `Type`. However, due to the dependency of types on terms the latter property of types may depend on some of the other typing constraints.

Things are complicated because of the reconstruction whose results, and therefore the exact shape of future typing constraints, depends on the particular order chosen for solving. If the constraints were solved in order then by the time a constraint is about to be solved we could prove that the type involved is well-formed of kind `Type`. We cannot assume in-order solving of constraints because much of the power of the reconstruction originates in the ability to solve the constraints out-of-order.

This complicates the correctness proof substantially. We structure the proof as a chain of dependent theorems, two for each of the constraint collection and constraint solving judgments. Note how the conclusion of one theorem establishes the assumptions of the next one.

We start with a theorem about the goal solving judgment, saying that during constraint list solving each typing constraint is eventually solved. However, depending on the particular order of solving some placeholder variables may be already instantiated, hence the Ψ_1.

Theorem 8.4 *If* $\Gamma \overset{r}{\vdash} C \Rightarrow \Psi$ *then for each* $N : A'$ *from* C *there exist* Ψ_1 *and* Ψ_2 *such that* $\Psi_1(\Gamma) \overset{r}{\vdash} \Psi_1(N) : \Psi_1(A') \Rightarrow \Psi_2$.

Proof: (of Theorem 8.4) The proof is by induction on the structure of the derivation $\mathcal{D} :: \Gamma \overset{r}{\vdash} C \Rightarrow \Psi$. The conclusion is vacuously true if C is empty. Also, if the last rule in $\mathcal{D}$ is the reordering rule, the induction hypothesis proves the conclusion directly. The other two case are similar so we only show here the case when the last rule in $\mathcal{D}$ is solving a typing constraint.
Case:

$$\mathcal{D} = \cfrac{\begin{array}{cc} \mathcal{D}_1 & \mathcal{D}_2 \\ \Gamma \overset{r}{\vdash} M : A \Rightarrow \Psi & \Psi(\Gamma) \overset{r}{\vdash} \Psi(C) \Rightarrow \Psi' \end{array}}{\Gamma \overset{r}{\vdash} C, M : A \Rightarrow \Psi' \circ \Psi}$$

We show the conclusion separately for $M : A$ and then for all the other $N : A'$ from C. If $M : A$ is the considered constraint in $C, M : A$ then the conclusion follows immediately with $\Psi_1 = \cdot$ and $\Psi_2 = \Psi$. Otherwise let $N : A'$ be a typing constraint in C. Then $\Psi(N) : \Psi(A')$ is a constraint in $\Psi(C)$. By induction hypothesis on $\mathcal{D}_2$ we get that there exists Ψ'_1 and Ψ'_2 such that $\Psi'_1(\Psi(\Gamma)) \overset{r}{\vdash} \Psi'_1(\Psi(N)) : \Psi'_1(\Psi(A')) \Rightarrow \Psi'_2$. This proves our conclusion with the required substitutions $\Psi_1 = \Psi'_1 \circ \Psi$ and $\Psi_2 = \Psi'_2$.

□

We continue now with a theorem about the constraint collection judgment. Using the result of the previous theorem we show that all the types in the resulting constraint list are well-formed and of kind `Type`.

Theorem 8.5 *If* $\Gamma \overset{r}{\vdash} M \Rightarrow (\Delta \,;\, C \,;\, B)$ *with* $\mathrm{PF}(\Gamma)$ *and* $\mathrm{PVF}(M)$ *and for all* $N : A'$ *in* C *there exist* Ψ_1, Ψ_2 *and* Δ' *such that* $\Psi_1(\Gamma, \Delta, \Delta') \overset{r}{\vdash} \Psi_1(N) : \Psi_1(A') \Rightarrow \Psi_2$ *then*

- $\Gamma, \Delta \overset{i}{\vdash} B : \mathtt{Type}$, *and*
- $\mathrm{PF}(\Delta)$, *and*
- $\mathrm{PF}(B)$, *and*
- *For all* $N : A'$ *in* C *we have*
  - $\mathrm{PF}(A')$, *and*
  - $\mathrm{PVF}(N)$, *and*
  - $\Gamma, \Delta \overset{i}{\vdash} A' : \mathtt{Type}$.

Proof: (of Theorem 8.5) The proof is by induction on the structure of the derivation $\mathcal{D} : \Gamma \overset{r}{\vdash} M \Rightarrow (\Delta \,;\, C \,;\, B)$. There are 4 cases depending on the last rule used in $\mathcal{D}$. The cases for constants and variables follow immediately as the Δ and the list of constraints are empty and because $\mathrm{PF}(\Gamma)$. The only interesting cases are those that deal with application.
Case:

$$\mathcal{D} = \cfrac{\begin{array}{c} \mathcal{D}_1 \\ \Gamma \overset{r}{\vdash} M \Rightarrow (\Delta \,;\, C \,;\, \Pi x{:}A.B) \end{array}}{\Gamma \overset{r}{\vdash} M * \Rightarrow (\Delta, u : A \,;\, C \,;\, [u/x]B)} \qquad u \text{ is a new placeholder variable}$$

We can immediately apply the induction hypothesis on $\mathcal{D}_1$ and obtain that

1. $\Gamma, \Delta \overset{i}{\vdash} \Pi x\!:\!A.B : \texttt{Type}$, and
2. $\mathrm{PF}(\Delta)$, and
3. $\mathrm{PF}(\Pi x\!:\!A.B)$, and
4. For all $N : A'$ in C we have $\Gamma, \Delta \overset{i}{\vdash} A' : \texttt{Type}$, and $\mathrm{PF}(A)$ and $\mathrm{PVF}(N)$.
5. Because u is new we can transform 4 in $\Gamma, \Delta, u : A \overset{i}{\vdash} A' : \texttt{Type}$, which proves part of the conclusion.
6. Similarly, from 1 we deduce that $\Gamma, \Delta, u : A \overset{i}{\vdash} [u/x]B : \texttt{Type}$, which is another part of the conclusion.
7. From 2 and 3 we can show that $\mathrm{PF}(\Delta, u : A)$
8. From 3 we show that $\mathrm{PF}([u/x]B)$, which concludes the proof of this case.

Case:

$$\mathcal{D} = \cfrac{\begin{array}{c}\mathcal{D}_1\\ \Gamma \overset{r}{\vdash} M \Rightarrow (\Delta \,;\, C \,;\, \Pi x\!:\!A.B)\end{array} \qquad x \in \mathrm{FV}(B) \supset (\mathrm{PF}(N) \textit{ and } \mathrm{PVF}(A) \textit{ and } \mathrm{PVF}(\Gamma(\mathrm{FV}(N))))}{\Gamma \overset{r}{\vdash} M\ N \Rightarrow (\Delta \,;\, C, N : A \,;\, [N/x]B)}$$

We can immediately apply the induction hypothesis on $\mathcal{D}_1$ and infer that:

1. $\Gamma, \Delta \overset{i}{\vdash} \Pi x\!:\!A.B : \texttt{Type}$, and
2. $\mathrm{PF}(\Delta)$ (part of the conclusion), and
3. $\mathrm{PF}(\Pi x\!:\!A.B)$, and
4. For all $N' : A'$ in C we have $\Gamma, \Delta \overset{i}{\vdash} A' : \texttt{Type}$ and $\mathrm{PF}(A')$ and $\mathrm{PVF}(N')$ (part of the conclusion).
5. From 3 we can deduce that $\mathrm{PF}([N/x]B)$ (part of the conclusion). Note that we have used the fact that if $x \in \mathrm{FV}(B)$ then $\mathrm{PF}(N)$.
6. The only part of the conclusion that remains to be proved is that $\Gamma, \Delta \overset{i}{\vdash} [N/x]B : \texttt{Type}$. If $x \notin \mathrm{FV}(B)$ then this follows immediately from 1. Otherwise is suffices to prove that $\Gamma \overset{i}{\vdash} N : A$.
7. By hypothesis we have that there exist Ψ_1, Ψ_2 and Δ' such that $\Psi_1(\Gamma, \Delta, \Delta') \overset{r}{\vdash} \Psi_1(N) : \Psi_1(A) \Rightarrow \Psi_2$.
8. But $\mathrm{PVF}(A)$ therefore we get $\Psi_1(A) = A$.
9. Because $\mathrm{PVF}(M\ N)$ (hypothesis) we know that $\mathrm{PVF}(N)$.
10. From 9 we conclude that $\Psi_1(N) = N$.
11. With 8 and 10 we transform 7 to $\Psi_1(\Gamma, \Delta, \Delta') \overset{r}{\vdash} N : A \Rightarrow \Psi_2$
12. But because $\mathrm{PVF}(N)$ then none of the placeholder variables from Δ and Δ' occur in N. Therefore we can transform 11 to $\Psi_1(\Gamma) \overset{r}{\vdash} N : A \Rightarrow \Psi_2$.

13. We know that $\mathrm{PVF}(\Gamma(\mathrm{FV}(N)))$ therefore we can transform 12 to $\Gamma \overset{r}{\vdash} N : A \Rightarrow \Psi_2$.

14. With 3 we apply Theorem 8.8 on 13 and deduce that $\Gamma \overset{i}{\vdash} N : A$. As motivated at 6 this concludes the proof of this case.

□

We continue with a theorem that shown that the constraint solving judgment, when presented with constraints whose types are well-formed, produces a well-typed substitution that satisfies the constraints.

Theorem 8.6 *If* $\Gamma \overset{r}{\vdash} C \Rightarrow \Psi$ *with*

- $\mathrm{PF}(\Gamma)$, *and*
- *For* $A \approx_a B$ *in* C *we have* $\mathrm{PF}(A)$, $\mathrm{PF}(B)$ *and* $\Gamma \overset{i}{\vdash} A : \mathtt{Type}$ *and* $\Gamma \overset{i}{\vdash} B : \mathtt{Type}$, *and*
- *For all* $N : A'$ *in* C *we have* $\mathrm{PVF}(N)$ *and* $\mathrm{PF}(A')$ *and* $\Gamma \overset{i}{\vdash} A' : \mathtt{Type}$, *and*

then the following are true:

- $\Gamma \vdash \Psi$, *and*
- $\mathrm{PF}(\Psi)$, *and*
- *For* $A \approx_a B$ *in* C *we have* $\Psi(A) \equiv_\beta \Psi(B)$, *and*
- *For all* $N : A'$ *in* C *we have* $\Psi(\Gamma) \overset{i}{\vdash} N : \Psi(A')$

Proof: (of Theorem 8.6) The proof is by induction on the structure of the derivation $\mathcal{D} : \Gamma \overset{r}{\vdash} C, A \approx_a B \Rightarrow \Psi$.
The case when the last rule in $\mathcal{D}$ is the reordering rule poses no problems. Similarly the case of an empty constraint list is trivial. We show next the case when the last rule in $\mathcal{D}$ is solving a typing constraint.
Case:

$$\mathcal{D} = \cfrac{\begin{array}{cc} \mathcal{D}_1 & \mathcal{D}_2 \\ \Gamma \overset{r}{\vdash} M : A \Rightarrow \Psi & \Psi(\Gamma) \overset{r}{\vdash} \Psi(C) \Rightarrow \Psi' \end{array}}{\Gamma \overset{r}{\vdash} C, M : A \Rightarrow \Psi' \circ \Psi}$$

Because $\mathrm{PF}(\Gamma)$, $\mathrm{PF}(A)$ and $\mathrm{PVF}(M)$ and $\Gamma \overset{i}{\vdash} A : \mathtt{Type}$ we can apply Theorem 8.1 on $\mathcal{D}_1$ and infer that $\Gamma \vdash \Psi$, $\mathrm{PF}(\Psi)$ and $\Psi(\Gamma) \overset{i}{\vdash} M : \Psi(A)$.
Because the substitution Ψ is without placeholders and is well-typed, the constraint list $\Psi(C)$ satisfies all the conditions for applying the induction hypothesis on $\mathcal{D}_2$ with respect to the type environment $\Psi(\Gamma)$. We deduce from the induction hypothesis that $\Psi(\Gamma) \vdash \Psi'$ and $\mathrm{PF}(\Psi')$ and for all $N : A'$ in C that $\Psi' \circ \Psi(\Gamma) \overset{i}{\vdash} N : \Psi' \circ \Psi$. Now we only need to use Lemma 8.19 to conclude the proof of this case.
The case when a unification constraint is solved is very similar with the difference that Theorem 8.2 is invoked to show that the unification returns a well-typed substitution.

□

The last theorem in the correctness proof of the reconstruction algorithm shows why the existence of a well-typed substitution defined on all locally introduced placeholder variables is enough to guarantee the well-typedness of the original term.

Theorem 8.7 *If* $\Gamma \stackrel{r}{\vdash} M \Rightarrow (\Delta\ ;\ C\ ;\ B)$ *and*

- $\Gamma, \Delta \vdash \Psi$*, and*
- $\mathrm{PF}(\Psi)$*, and*
- $Dom(\Delta) \subseteq Dom(\Psi)$*, and*
- *For all* $N : A'$ *in* C *we have* $\mathrm{PVF}(N)$ *and* $\Psi(\Gamma) \stackrel{i}{\vdash} N : \Psi(A')$

then $\Psi(\Gamma) \stackrel{i}{\vdash} M : \Psi(B)$.

Proof: (of Theorem 8.7) The proof is by induction on the structure of the derivation $\mathcal{D} :: \Gamma \stackrel{r}{\vdash} M \Rightarrow (\Delta\ ;\ C\ ;\ B)$. The cases of a constant or a variable follow immediately from the hypothesis. We consider next the cases of application to explicit terms and to placeholders next.
Case:

$$\mathcal{D} = \cfrac{\overset{\mathcal{D}_1}{\Gamma \stackrel{r}{\vdash} M \Rightarrow (\Delta\ ;\ C\ ;\ \Pi x{:}A.B)} \qquad x \in \mathrm{FV}(B) \supset (\mathrm{PF}(N)\ \mathit{and}\ \mathrm{PVF}(A)\ \mathit{and}\ \mathrm{PVF}(\Gamma(\mathrm{FV}(N))))}{\Gamma \stackrel{r}{\vdash} M\ N \Rightarrow (\Delta\ ;\ C, N : A\ ;\ [N/x]B)}$$

We can immediately apply the induction hypothesis on $\mathcal{D}_1$ and we conclude that $\Psi(\Gamma) \stackrel{i}{\vdash} M : \Psi(\Pi x{:}A.B)$. From the hypothesis we have that $\mathrm{PVF}(N)$ and $\Psi(\Gamma) \stackrel{i}{\vdash} N : \Psi(A)$. From the application rule in LF_i, and because $\mathrm{PVF}(N)$ we deduce that $\Psi(\Gamma) \stackrel{i}{\vdash} M\ N : \Psi([N/x]B)$. Note that the newly introduced type $[N/x]B$ is without placeholders because if x occurs in B then $\mathrm{PF}(N)$.
Case:

$$\mathcal{D} = \cfrac{\overset{\mathcal{D}_1}{\Gamma \stackrel{r}{\vdash} M \Rightarrow (\Delta\ ;\ C\ ;\ \Pi x{:}A.B)}}{\Gamma \stackrel{r}{\vdash} M * \Rightarrow (\Delta, u : A\ ;\ C\ ;\ [u/x]B)} \qquad u \text{ is a new placeholder variable}$$

Let $\Psi' = \Psi|_{\Gamma,\Delta}$. We follow a sequence of deductions as follows:

1. From $\Gamma, \Delta, u : A \vdash \Psi$ (hypothesis) and $Dom(\Delta, u : A) \subseteq Dom(\Psi)$ (hypothesis) we deduce:
2. $\Psi(\Gamma, \Delta, u : A) \stackrel{i}{\vdash} \Psi(u) : \Psi(A)$, and
3. $Dom(\Delta) \subseteq Dom(\Psi')$.
4. Because $Dom(\Delta, u : A) \subseteq Dom(\Psi)$ we have that $\Psi(\Gamma, \Delta, u : A) = \Psi(\Gamma)$.
5. From 2 and 4 we infer that $\Psi(\Gamma) \stackrel{i}{\vdash} \Psi(u) : \Psi(A)$.
6. From Lemma 8.20 and 1 we have that $\Gamma, \Delta \vdash \Psi'$. Also because u is new it cannot appear in C or in Γ. Therefore we can deduce that for all $N : A'$ in C we have $\Psi'(\Gamma) \stackrel{i}{\vdash} N : \Psi'(A')$. Thus we can apply the induction hypothesis on $\mathcal{D}_1$ and conclude that
7. $\Psi'(\Gamma) \stackrel{i}{\vdash} M : \Pi x{:}\Psi'(A).\Psi'(B)$.
8. Again, because u is new it cannot appear in Γ, M, A or B. Therefore, from 7 we deduce that $\Psi(\Gamma) \stackrel{i}{\vdash} M : \Pi x{:}\Psi(A).\Psi(B)$.
9. Recall that we assume that all types involved are without placeholders. Thus, $\mathrm{PF}(\Psi'(A))$ and therefore $\mathrm{PF}(\Psi(A))$.

10. Now we can use the implicit application rule of LF_i with 5, 8 and 9 and with the placeholder replacement being $\Psi(u)$. The resulting type is $[\Psi(u)/x]\Psi(B) = \Psi([u/x]B)$.
11. This case is not complete until we verify that the newly introduced type $\Psi([u/x]B)$ is without placeholders. This follows immediately from the $\mathrm{PF}(\Psi(\Pi x\!:\!A.B))$ and $\mathrm{PF}(\Psi)$.

Note that this is the place where we require the property that Ψ be well-typed and defined for all variables in Δ.

□

This concludes the skeleton of the correctness proof for the reconstruction algorithm. In the rest of this section we have the proofs of the helper lemmas used in the correctness proofs. We start first with a family of theorems mirroring the correctness proof but in the special case when the terms involved are fully-reconstructed.

8.5 Correctness in the Fully-Explicit Case

As part of the correctness of the reconstruction algorithm we make use of the correctness of the algorithm in the case when both the LF term and type involved do not contain placeholders or placeholder variables. The correctness proof in the fully-explicit form follows the same pattern as the proof in the general case, with some simplifications. We do not show here a complete proof of this case. We just state the lemmas involved.

Theorem 8.8 *If* $\Gamma \overset{r}{\vdash} M : A \Rightarrow \Psi$ *and* $\mathrm{PF}(\Gamma)$, $\mathrm{PF}(M)$, $\mathrm{PF}(A)$, $\mathrm{PVF}(M)$, $\mathrm{PVF}(A)$ *and* $\mathrm{PVF}(\Gamma(\mathrm{FV}(M)))$ *then* $\Psi = \cdot$ *and* $\Gamma \overset{i}{\vdash} M : A$.

Proof: The proof of this theorem is done similarly to that of Theorem 8.1 by induction on the derivation $\mathcal{D} : \Gamma \overset{r}{\vdash} M : A \Rightarrow \Psi$. The abstraction case is simple. For the application case we need a series of auxiliary lemmas about the constraint collection and solving judgments in the case of fully-explicit terms and types. These lemmas are stated without proof below.

□

Lemma 8.9 *If* $\Gamma \overset{r}{\vdash} M \Rightarrow (\Delta\, ;\, C\, ;\, B)$ *and* $\mathrm{PF}(\Gamma)$, $\mathrm{PF}(M)$, $\mathrm{PVF}(M)$ *and* $\mathrm{PVF}(\Gamma(\mathrm{FV}(M)))$ *then*

- $\Delta = \cdot$, *and*
- $\mathrm{PF}(B)$ *and* $\mathrm{PVF}(B)$, *and*
- *For all* $N : A'$ *in* C *we have that* $\mathrm{PF}(N)$, $\mathrm{PVF}(N)$, $\mathrm{PF}(A')$ *and* $\mathrm{PVF}(A')$ *and* $\mathrm{PVF}(\Gamma(\mathrm{FV}(N)))$.

The intuition behind Lemma 8.9 is that because M does not have placeholders, no placeholder variables are introduced, hence $\Delta = \cdot$. Also the terms in C are subterms of M and therefore do not contain placeholders or placeholder variables and also all their free variables have types that do not contain placeholder variables. The types in C and B are constructed from fully-explicit types (because $\mathrm{PF}(\Gamma)$ and $\mathrm{PVF}(\Gamma(\mathrm{FV}(M))))$ with subterms of M, hence the condition on types.

Lemma 8.10 *If* $A \approx_a B \Rightarrow \Psi$ *and* $\mathrm{PVF}(A)$ *and* $\mathrm{PVF}(B)$ *then*

- $\Psi = \cdot$, *and*
- $A \equiv_\beta B$

The lemma above is proved by induction on the structure of the unification judgment. The intuition behind it is that if the terms to be unified do not contain placeholder variables then the resulting substitution must be empty. In this case the two terms are β-equivalent.

Lemma 8.11 *If* $\Gamma \overset{r}{\vdash} C, A \approx_a B \Rightarrow \Psi$ *and* $\mathrm{PF}(\Gamma)$, $\mathrm{PF}(A)$, $\mathrm{PVF}(A)$, $\mathrm{PF}(B)$, $\mathrm{PVF}(B)$ *and for all* $N : A'$ *in* C *we have* $\mathrm{PF}(N)$, $\mathrm{PVF}(N)$, $\mathrm{PF}(A')$, $\mathrm{PVF}(A')$ *and* $\mathrm{PVF}(\Gamma(\mathrm{FV}(N)))$ *then*

- $\Psi = \cdot$, *and*
- $A \equiv_\beta B$, *and*
- $\Gamma \overset{i}{\vdash} N : A'$ *for all* $N : A'$ *in* C

The proof of Lemma 8.11 is by induction on the structure of the derivation $\Gamma \overset{r}{\vdash} C, A \approx_a B \Rightarrow \Psi$. When a unification is solved we use Lemma 8.10 to conclude that the resulting substitution is empty and that the unified types are β-equivalent. When a typing constraint is solved then the hypothesis provides all the conditions necessary to apply Theorem 8.8 and conclude again that the substitution is empty and that the typing constraints are satisfied.

Lemma 8.12 *If* $\Gamma \overset{r}{\vdash} M \Rightarrow (\cdot\ ;\ C\ ;\ B)$ *and* $A \equiv_\beta B$ *with* $\mathrm{PF}(A)$ *and for all* $N : A'$ *in* C *we have* $\Gamma \overset{i}{\vdash} N : A'$ *and* $\mathrm{PF}(A')$ *then* $\Gamma \overset{i}{\vdash} M : A$.

We prove Lemma 8.12 by induction on the structure of the collection derivation. Again we cannot have placeholders and because the typing constraints from C are satisfied we can immediately prove the conclusion using the typing rules of LF_i.

8.6 Soundness of $\mathbf{LF}_i$ typing

We have proved the correctness of the reconstruction algorithm with respect to the LF_i typing system. In order to complete the proof of adequacy of the reconstruction algorithm for checking proofs we still have to prove Theorem 5.1, that is the soundness of LF_i typing with respect to LF typing. We do this in the rest of this section. For clarity we restate the theorem here.

Theorem 8.13 Soundness of $\mathbf{LF}_i$ typing *If* $\Gamma \overset{i}{\vdash} M : A$ *and* $\mathrm{PF}(\Gamma)$, $\mathrm{PF}(A)$, *then there exists* M' *such that* $M \nearrow M'$ *and* $\Gamma \overset{LF}{\vdash} M' : A$.

Proof: The proof is by induction on the structure of the derivation $\mathcal{D} : \Gamma \overset{i}{\vdash} M : A$. The case of a constant or a variable is trivial. In the case of the β-equivalence rule we use the rule hypothesis $\mathrm{PF}(A)$ to apply the induction hypothesis and then we use the LF β-equivalence rule.
In the case of an abstraction again we use the theorem hypothesis $PF(\Pi x : A.B)$ to ensure that we can apply the induction hypothesis. The remaining cases deal with the application to a term or a placeholder.
Case:

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

Because of the rule hypothesis $\mathrm{PF}(A)$ we can apply the induction hypothesis on $\mathcal{D}_2$ and deduce that there exists N' such that $N \nearrow N'$ and $\Gamma \overset{LF}{\vdash} N' : A$.
From the theorem hypothesis we have that $\mathrm{PF}([N/x]B)$. From here we infer that $\mathrm{PF}(B)$ and then that $\mathrm{PF}(\Pi x{:}A.B)$. This justifies applying the induction hypothesis to $\mathcal{D}_1$ and inferring that there

exists M' such that $M \nearrow M'$ and $\Gamma \overset{LF}{\models} M' : \Pi x{:}A.B$. Now using the LF application rule we infer that $\Gamma \overset{LF}{\models} M'\ N' : [N'/x]B$. It is evident that $M * \nearrow M'\ N'$. What remains to be proved is that $[N'/x]B \equiv_\beta [N/x]B$. This is that case if $x \notin \mathrm{FV}(B)$. Otherwise, we know from the hypothesis that $\mathrm{PF}([N/x]B)$ which implies that $\mathrm{PF}(N)$ and then that $N = N'$.

The case when the last rule in $\mathcal{D}$ is an application to a term is very similar to the case presented above.

□

8.7 Auxiliary Lemmas

The following lemmas are results that were used in the correctness proof of the type reconstruction algorithm. Most of them are trivial to prove and therefore we omit their proof. Recall also that we made the convention that all the types involved in the statements of the theorems and lemmas are placeholder-free.

The first lemma establishes some canonical forms of types in LF_i judgments.

Lemma 8.14 *If $\Gamma \overset{i}{\models} M : A$ then the following are true:*

- *If $M = x$ then $\Gamma(x) \equiv_\beta A$*
- *If $M = c$ then $\Sigma(c) \equiv_\beta A$*
- *If $M = M_1\ M_2$ then $\Gamma \overset{i}{\models} M_1 : \Pi x{:}A_1.A_2$ and $\Gamma \overset{i}{\models} M_2 : A_1$ and $[M_2/x]A_2 \equiv_\beta A$*
- *If $M = \lambda x.N$ then $A = \Pi x{:}A_1.A_2$ and $\Gamma, x : A_1 \overset{i}{\models} M : A_2$.*

Next we have a lemma saying that β-reduction "preserves" the type of the expression. Because of the dependent types the resulting type is actually obtained by substitution itself.

Lemma 8.15 *If $\Gamma \overset{i}{\models} \lambda x.M : \Pi x{:}A.B$ and $\Gamma \overset{i}{\models} N : A$ with $\mathrm{PF}(N)$ then $\Gamma \overset{i}{\models} [N/x]M : [N/x]B$.*

We continue with a crucial lemma used throughout the proof of correctness. This lemma says that if a substitution is well-typed on certain placeholder variables then it preserves the typing relation.

Lemma 8.16 *Let $D = Dom(\Psi) \cap \mathrm{FV}(M)$ if $\mathrm{PF}(M)$ and $Dom(\Psi)$ otherwise. If $\mathrm{PF}(\Psi)$ and $\Psi(\Gamma) \overset{i}{\models} \Psi(u) : \Psi(\Gamma(u))$ for all $u \in D$ and if $\Gamma \overset{i}{\models} M : A$ then $\Psi(\Gamma) \overset{i}{\models} \Psi(M) : \Psi(A)$.*

Proof: (of Lemma 8.16) The proof is by induction on the structure of the derivation $\mathcal{D} :: \Gamma \overset{i}{\models} M : A$. The case of a constant is trivial because the constant and its type do not change by substitution. More interesting cases are when M is a variable or a placeholder variable. In the case of a normal variable, or a placeholder variable outside $Dom(\Psi)$, the conclusion follows immediately from the definition of $\Psi(\Gamma)$. In the case of a placeholder variable that is in $Dom(\Psi)$ the conclusion follows from the hypothesis because that variable is also in $\mathrm{FV}(M)$ and therefore in D.
We consider the other cases below:
Case:

$$\mathcal{D} = \cfrac{\begin{array}{c}\mathcal{D}_1 \\ \Gamma \overset{i}{\models} M : A\end{array} \qquad A \equiv_\beta B \qquad \mathrm{PF}(A)}{\Gamma \overset{i}{\models} M : B}$$

In this case we can apply the induction hypothesis on $\mathcal{D}_1$ and infer that $\Psi(\Gamma) \overset{i}{\models} \Psi(M) : \Psi(A)$. Note that we have used the hypothesis $\mathrm{PF}(A)$ to ensure that our implicit convention about types is preserved. By Lemma 8.21 we get that $\Psi(A) \equiv_\beta \Psi(B)$. Because $\mathrm{PF}(\Psi)$ and $\mathrm{PF}(A)$ we deduce that $\mathrm{PF}(\Psi(A))$. We can therefore use the LF_i rule for beta-equivalence and infer that $\Psi(\Gamma) \overset{i}{\vdash} \Psi(M) : \Psi(B)$.

Case:

$$\mathcal{D} = \frac{\Gamma, x : A \overset{i}{\vdash} M : B}{\Gamma \overset{i}{\vdash} \lambda x.M : \Pi x{:}A.B}$$

The set of placeholder variables D is the same for $\lambda x.M$ and M. We know that for all $u \in D$ we have $\Psi(\Gamma) \overset{i}{\vdash} \Psi(u) : \Psi(\Gamma(u))$. Because x cannot occur in Ψ we deduce that $\Psi(\Gamma, x : A) \overset{i}{\vdash} \Psi(u) : \Psi((\Gamma, x : A)(u))$ for all $u \in D$. We can therefore apply the induction hypothesis and conclude that $\Psi(\Gamma, x : A) \overset{i}{\vdash} \Psi(M) : \Psi(B)$. From here we can use the abstraction rule of LF_i and deduce the desired conclusion $\Psi(\Gamma) \overset{i}{\vdash} \Psi(\lambda x.M) : \Psi(\Pi x{:}A.B)$.

Case:

$$\mathcal{D} = \frac{\overset{\mathcal{D}_1}{\Gamma \overset{i}{\vdash} M : \Pi x{:}A.B} \qquad \overset{\mathcal{D}_2}{\Gamma \overset{i}{\vdash} N : A} \qquad \mathrm{PF}(A)}{\Gamma \overset{i}{\vdash} M\ N : [N/x]B}$$

We know that $\mathrm{PF}(M\ N)$ iff $\mathrm{PF}(M)$ and $\mathrm{PF}(N)$. This and the fact that the free variables of M and N are among those of $M\ N$ allow us to use the induction hypothesis both on $\mathcal{D}_1$ and $\mathcal{D}_2$. Because $\mathrm{PF}(A)$ and $\mathrm{PF}(\Psi)$ we have that $\mathrm{PF}(\Psi(A)$ which can be used together with the induction hypotheses to infer the desired conclusion $\Psi(\Gamma) \overset{i}{\vdash} \Psi(M\ N) : \Psi([N/x]B)$. Note that by our implicit convention on types we have that $\mathrm{PF}([N/x]B)$ which implies that $\mathrm{PF}(B)$ and also that $\mathrm{PF}(\Pi x{:}A.B)$.

Case:

$$\mathcal{D} = \frac{\overset{\mathcal{D}_1}{\Gamma \overset{i}{\vdash} M : \Pi x{:}A.B} \qquad \overset{\mathcal{D}_2}{\Gamma \overset{i}{\vdash} N : A} \qquad \mathrm{PF}(A)}{\Gamma \overset{i}{\vdash} M * : [N/x]B}$$

Note that in this case we do not have $\mathrm{PF}(M *)$ and therefore we must have $\Gamma \vdash \Psi$. We can therefore apply the induction hypothesis for the derivation $\mathcal{D}_1$ and infer that $\Psi(\Gamma) \overset{i}{\vdash} \Psi(M) : \Pi x{:}\Psi(A).\Psi(B)$. From the induction hypothesis on $\mathcal{D}_2$ we infer that $\Psi(\Gamma) \overset{i}{\vdash} \Psi(N) : \Psi(A)$. From here we follow the same steps as in the previous case. Note that $\Psi(M *) = \Psi(M) *$.

□

The Lemma 8.16 is not actually used in that form. All its uses are in the form of two corollaries stated below.

Corollary 8.17 *If* $\Gamma \overset{i}{\vdash} M : A$ *and* $\Gamma \vdash \Psi$ *with* $\mathrm{PF}(\Psi)$ *then* $\Psi(\Gamma) \overset{i}{\vdash} \Psi(M) : \Psi(A)$.

Proof: (of Corollary 8.17) The corollary follows immediately from Lemma 8.16 if we note that $D \subseteq \mathit{Dom}(\Psi)$ and that $\Gamma \vdash \Psi$ implies that for all $u \in \mathit{Dom}(\Psi)$ we have $\Psi(\Gamma) \overset{i}{\vdash} \Psi(u) : \Psi(\Gamma(u))$.

□

Corollary 8.18 *If* $\Gamma \overset{i}{\vdash} u : A$ *and* $\Gamma \overset{i}{\vdash} M : A$ *with* $\mathrm{PF}(M)$ *and* $u \notin \mathrm{FV}(M)$ *then* $\Gamma \overset{i}{\vdash} u \leftarrow M$.

Proof: (of Corollary 8.18) Let $\Psi = u \leftarrow M$. Because $\mathrm{PF}(M)$ we can apply Lemma 8.16 with $D = \mathit{Dom}(\Psi) \cap \mathrm{FV}(M) = \emptyset$ and we infer that $\Psi(\Gamma) \overset{i}{\vdash} \Psi(M) : \Psi(A)$. But because $u \notin \mathrm{FV}(M)$ we have that $\Psi(M) = M = \Psi(u)$. Therefore $\Gamma \vdash \Psi$.

□

We continue with two lemmas dealing with typing substitution. The first is concerned with the well-typedness of a composition of two substitutions. The second one deals with restricted substitutions.

Lemma 8.19 *If* $\Gamma \vdash \Psi$ *and* $\Psi(\Gamma) \vdash \Psi'$ *with* $\mathrm{PF}(\Psi)$ *and* $\mathrm{PF}(\Psi')$ *then* $\Gamma \vdash \Psi' \circ \Psi$ *and* $\mathrm{PF}(\Psi' \circ \Psi)$*.*

Proof: It is obvious that $\mathrm{PF}(\Psi' \circ \Psi)$. To prove that $\Gamma \vdash \Psi' \circ \Psi$ consider a placeholder variable $u \in Dom(\Psi' \circ \Psi)$. Then either $u \in Dom(\Psi)$, in which case $\Psi(Gamma) \stackrel{i}{\vdash} \Psi(u) : \Psi(\Gamma(u))$ and by Lemma 8.16 we get the desired conclusion, or $u \in Dom(\Psi') \backslash Dom(\Psi)$, in which case $\Psi'(\Psi(\Gamma)) \stackrel{i}{\vdash} \Psi'(u) : \Psi'(\Psi(\Gamma(u)))$.

$\square$

Lemma 8.20 *If* $\Gamma, \Delta \vdash \Psi$ *and* $Dom(\Delta) \subseteq Dom(\Psi)$ *then* $\Psi(\Gamma, \Delta) = \Psi(\Gamma) = \Psi\big|_{Dom(\Gamma)}(\Gamma)$ *and* $\Gamma \vdash \Psi\big|_{Dom(\Gamma)}$*.*

Proof: It is easy to prove that $\Psi(\Gamma, \Delta) = \Psi(\Gamma)$ using the definition of substitution applied to type environments. Also it must be the case that Γ does not contain any placeholder variable contained in Δ, thus we get $\Psi(\Gamma) = \Psi\big|_{Dom(\Gamma)}(\Gamma)$.
For the second part, let $u \in Dom(\Psi) \cap Dom(\Gamma)$. We have that $\Psi(\Gamma) \stackrel{i}{\vdash} \Psi(u) : \Psi(\Gamma(u))$. Because nothing in Δ can occur in Γ we conclude that $\Gamma \vdash \Psi\big|_{Dom(\Gamma)}$.

$\square$

The last lemmas required are concerned with β-equivalence. Their proof is trivial so we just state them here.

Lemma 8.21 *If* $M \equiv_\beta N$ *then* $\Psi(M) \equiv_\beta \Psi(N)$*.*

Lemma 8.22 *If* $M \equiv_\beta N$ *and* $M' \equiv_\beta N'$ *then* $[M/x]M' \equiv_\beta [N/x]N'$*.*

9 An Optimized Version of Type Reconstruction

In this section we examine several optimizations to the reconstruction algorithm presented in Section 6. We chose to present these optimizations separately from the main algorithm to simplify the presentation of the algorithm and its correctness proof, and also because the behavior and power of the reconstruction is not influenced by these optimizations. Their only purpose is to lay the grounds for an efficient implementation of the algorithm, which will be discussed in Section 10.

Because the optimizations presented in this section were not proved correct as part of the main correctness theorem we have the obligation of proving their correctness here. However, in order to simplify the presentation we are only showing informal correctness arguments.

We consider two classes of optimizations. The first class consists of signature-independent optimizations whose focus is to optimize the checking of the side conditions present in the reconstruction algorithm. Examples of signature-independent optimizations are the occurs check optimization and an optimization of the complicated side condition from the collection rule for explicit parameters. We show in Figure 16 on page 51 a variant of the reconstruction algorithm that incorporates these optimizations.

The second class of optimizations are dependent on the particular signature that is at the base of reconstruction. The reason we are interested in such optimizations is that in all of our current experiments we are interested in using the reconstruction as the basis for validating first-order logic proofs. In such conditions we can eliminate some of the side-condition checks without compromising the correctness of the reconstruction.

9.1 Optimizing the Occurs Check

One of the most important features of our reconstruction algorithm is that the reconstructed subterms do not have to be typechecked because, by design, the algorithm ensures that they are well-typed. This saves a lot of time when validating proofs.

If we examine the unoptimized algorithm we notice that the basic reconstruction step is a placeholder-variable instantiation in the unification judgment. Whenever a new instantiation is made, the algorithm scans the reconstructed term to perform the occurs check. This operation is less expensive than complete typechecking but still detracts from the performance advantage of the reconstruction versus the validation of fully-explicit terms.

We have discovered that in most of our experiments the occurs check is not necessary. This section presents the results of our efforts to isolate as precisely as possible the cases when the absence of the occurs check could lead to inconsistencies. The optimization presented in this section recovered about 90% of the cost of occurs check in our experiments with proof validation for first-order logic.

The basic idea behind the occurs-check optimization is that most placeholder variables occur at most once and in at most one of the terms involved in a unification operation. When such a variable is instantiated the occurs check is obviously not necessary. The rest of the section presents how we identify the placeholder variables that have this linearity attribute and how we ensure that the attribute is preserved as the unification goes along.

We attach a linearity attribute to each placeholder variable to identify those placeholder variables that occur at most once and in at most one of the unified terms. We write u^l to denote that the placeholder variable u is linear and we write u^n to denote that u is non-linear. We reserve the notation u for situations where the linearity attribute of u is not relevant. The purpose of this labelling is to be able to isolate the instantiation operations that do not require an occurs check by means of a new instantiation rule:

$$\overline{u^l \approx M \Rightarrow u^l \leftarrow M}$$

The first task is to label the placeholder variables as they are introduced in the collection rule for implicit application. In order to determine the labelling we define the judgment $linear(x, \Pi x_1 : A_1. \ldots \Pi x_n : A_n.A_{n+1})$ to denote that the variable x occurs at most once in any A_i component of the type. This notion is formalized below using the notation $x \in_{\leq 1} \mathrm{FV}(A)$ to denote that x has at most one free occurrence in A.

$$\frac{linear(x, B) \qquad x \in_{\leq 1} \mathrm{FV}(A)}{linear(x, \Pi y : A.B)} \qquad \frac{x \in_{\leq 1} \mathrm{FV}(A)}{linear(x, A)}$$

The variable labelling is done using an additional modified version of the collection rule which labels linear placeholder variables:

$$\frac{\Gamma \overset{r}{\vdash} M \Rightarrow (\Delta\ ;\ C\ ;\ \Pi x : A.B) \qquad linear(x, B)}{\Gamma \overset{r}{\vdash} M * \Rightarrow (\Delta, u^l : A\ ;\ C\ ;\ [u^l/x]B)}$$

In most cases the linearity check has a small run-time cost. This is because whenever the head of M is a constant the linearity predicate can be computed statically and cached in the signature. For example if the head of M is the constant `and_i` of type

$$\Pi x_1 : \mathtt{pred}\ .. \Pi x_2 : \mathtt{pred}\ .. \Pi x_3 : \mathtt{pf}\ x_1 .. \Pi x_4 : \mathtt{pf}\ x_2 .. \mathtt{pf}\ (\mathtt{and}\ x_1\ x_2)$$

the all the variables introduced in the collection for M are linear. However, if the head is `eq_id` of type

$$\Pi x_1 : \mathtt{exp}\ .. \mathtt{pf}\ (= x_1\ x_1)$$

the variable x_1 is not linear because it occurs twice in the same atomic component of the type. If the head of M is a variable we can simply use the usual non-linear collection rule and avoid the check. Note that we do not jeopardize the correctness of the algorithm by having too few linearity labels.

We can state the linearity property in terms of the logic inference rules that the constants in Σ represent. A logic variable is linear in an inference rule if it appears at most once in the conclusion and at most once in each of the hypotheses of the inference rule.

Just identifying the linear placeholder variables is not sufficient. In order to motivate the other changes to the reconstruction rules we first state our goals in the form of a list of invariants that we want to preserve throughout reconstruction. These invariants also serve as the basis for the informal correctness arguments, and can be viewed as informal statements of a series of correctness theorems.

Invariants:

1. *In all judgments we never have simultaneous occurrences of both the linear and the non-linear versions of a placeholder variable. That is, the linearity property is attached to a variable and not to its occurrences.*

2. *Whenever a unification judgment is invoked all linear variables occur at most once in at most one of the terms to be unified.*

3. *All substitutions have the property that non-linear variables are instantiated to terms containing only non-linear variables.*

It is clear that if the above invariants hold then it is safe to omit the occurs check when a linear variable is unified. Next we show the modifications that are required in order to preserve these invariants.

Substitutions are produced by the instantiation rule and are then transformed using composition and restriction. It is easy to see that substitution composition and restriction preserve the substitution invariant. The only remaining fact to check about substitutions is that the invariant holds on a newly produced instantiations. Substitutions introduced by the linear instantiation rule are not a problem. However, in order to establish the invariant in the case of a non-linear instantiation we must change the instantiation rule to instantiate non-linear variables only to terms containing non-linear variables.

For this purpose we define the operation $\mathit{Nonlinear}(M) \Rightarrow \Psi$ to produce a substitution containing only mappings of the form $u^l \leftarrow u^n$ for all linear variables u^l occurring in M. The modified version of the non-linear instantiation rule is:

$$\frac{u \notin \mathrm{FV}(M) \qquad \mathit{Nonlinear}(M) \Rightarrow \Psi}{u^n \approx M \Rightarrow \Psi \circ [u^n \leftarrow M]}\,\texttt{u\_inst}$$

Similarly we need to modify the β-reduction unification rule to prevent linear variables to be duplicated by β-reduction.

$$\frac{\mathit{Nonlinear}(N) \Rightarrow \Psi \qquad [\Psi(N)/x]M \approx_a M' \Rightarrow \Psi'}{(\lambda x.M)\ N \approx_a M' \Rightarrow \Psi' \circ \Psi}\,\texttt{au\_beta}$$

Note that whenever we change the linearity attribute of a variable, the fact is recorded in the substitution produced by the *Nonlinear* function so that we can do the change globally. This ensures that the linearity attribute is global.

The most important invariant from the list above is the one about unification. We can show that this invariant is preserved throughout unification. The only difficult case here is the application case. We can prove the invariant preservation in this case if we observe the following properties of substitutions produced by unification:

- If $M \approx N \Rightarrow \Psi$ then Ψ can contain only variables that occur in M or N.
- If $M \approx N \Rightarrow \Psi$ then any linear variable appears at most once in at most one of the instantiations from Ψ.

The most difficult property to show is that when unification is invoked from the constraint solving judgment the unification invariant holds. In order to do this we must observe that the following invariants hold:

Invariants:

4. *If $M \approx N \Rightarrow \Psi$ and $u^l \in \mathit{Dom}(\Psi) \cap \mathrm{FV}(M)$ and $v^l \in \mathrm{FV}(\Psi(u^l))$ then $v^l \in \mathrm{FV}(N)$.*
5. *No placeholder variable that appears in $\Gamma(x)$ for some x, is instantiated until $x : A$ is retracted from Γ.*

6. *Whenever $\Gamma \overset{r}{\vdash} M : A \Rightarrow \Psi$ is invoked, all linear variables occur at most once in A. Also, the resulting substitution does not contain instantiations $\Psi(u)$ that contain linear variables.*

The first invariant above is shown by induction on the unification judgment using the fact that linear variables appear at most once in at most one side of the unification. The second invariant is motivated by the fact that only the empty substitution can be the result of type-checking an abstraction.

In order to show the last invariant we note that type-checking judgments are only invoked from the constraint solving judgment, which in turn is invoked after a constraint list C was produced by the judgment $\Gamma \overset{r}{\vdash} M \Rightarrow (\Delta ; C ; B)$. If $N' : A'$ is a component of a constraint list C we notice that A' has the linearity attribute and furthermore it only contains linear variables from the newly introduced set Δ.[1] Also, if $A \approx_a B$ is a unification constraint in C we notice that the linearity invariant is established initially and furthermore the linear variables in A are not in Δ, while all the linear variables in B are in Δ.

The last proof obligation is to show that while doing constraint solving followed by the transformation of the remaining list with the intermediate substitution, the linearity invariant is preserved. This can be done separately for the cases when the first goal solved is a unification goal or a typing goal. For this purpose we use the substitution invariants mentioned above.

As shown in Section 11, this occurs-check optimization is very effective when checking proofs in first-order logics. This is because all the applications have a constant head and we do not loose anything by not optimizing the case of variable heads. Another and a more important reason is that the vast majority of inference rule in a typical logic are completely linear. For example all the inference rule of first-order predicate logic are linear. In our experiments we only use non-linear rules that deal with arithmetic and even in that case only a part of the variables are non-linear. We have measured reductions of up to 90% in the cost of the "occurs" check when checking proofs of first-order logic. This translated to reductions of up to 60% in the total cost of proof validation. These numbers were measured for an implementation that contains other optimizations, some of which make the cost of computing the *Nonlinear* function negligible.

9.2 Optimizing the Side Conditions

The largest opportunity for optimization in the reconstruction algorithm is given by the side conditions. In the previous section we presented such an optimization for the occurs-check side condition. Another side condition that can be optimized is the one in the collection rule for application to an explicit term.

This side condition involves four checks in the worst case:

$$x \in \mathrm{FV}(B) \supset (\mathrm{PF}(N) \textit{ and } \mathrm{PVF}(A) \textit{ and } \mathrm{PVF}(\Gamma(\mathrm{FV}(N))))$$

We note first that the check $x \in \mathrm{FV}(B)$ can be precomputed in the cases when the head of M is a constant in a similar way as we precomputed the linearity predicate for the occurs check optimization.

We focus on the checks on N and we ignore here the check $\mathrm{PVF}(A)$. The basic idea here is that N is collected in the typing constraint list and is eventually type-checked. Therefore, instead of checking the conditions on N eagerly at the time of collection, we just mark that they should be

[1] Except perhaps some linear variable occurring in some $\Gamma(x)$. But we argued before that these are never instantiated while solving the current constraint list, and therefore we can ignore them here.

checked at the time N is checked. As shown in Section 11, this is an effective optimization because delaying the checks has the potential of collapsing multiple checks into a single one.

Whenever N is added to the typing constraint list we set a flag marking that when N is typechecked we should also verify that it does not contain placeholders and all variables free in it have types that do not contain placeholder variables. We call this flag p and we write the new reconstruction judgment as $\Gamma \vdash^{r}_{p} M : A \Rightarrow \Psi$. The flag p can have either the value `full`, meaning that the term M cannot contain placeholders, or `implicit` otherwise. The initial value of the flag is `implicit`.

With this extra flag we can delay the side-condition check from the collection rule to when the explicit argument is typechecked. The new reconstruction judgments, implementing this optimization and also the occurs-check optimization are shown in Figure 16.

9.3 Optimizations Specific to First-Order Logics

The optimization presented in Section 9.2 optimized the checking of a complicated side condition from the constraint collecting rule. As a result of this optimization, the side condition was split in several parts. Among these the most expensive are the side conditions from the rules `cc_x` (checking that the type of a variable does not contain placeholder variables) and `cc_dep` (checking that a type does not contain placeholder variables).

In certain situations these residual checks can be omitted altogether. This section shows that this is the case if we restrict our attention to checking proofs of first-order logic predicates.

For the purpose of this section we restrict our attention to reconstruction in the case when the signature Σ is the representation of a first-order logic, using as an example the representation of the logic $\mathcal{L}$ introduced in Section 2.

By inspection of the signature Σ we realize that the only dependent type family is `pf` $:$ `pred` $\rightarrow$ `Type`. Also by inspecting the predicate constructors we notice that only objects of type `exp`, `pred` and `exp` $\rightarrow$ `pred` can occur in normal-form predicates. From this we conclude that the only type dependency in Σ is on objects of type `pred`, `exp` and `exp` $\rightarrow$ `pred`.

Therefore, in the `cc_dep` rule because we have $x \in \mathrm{FV}(B)$ it must be that A is one of the above mentioned types, none of which can contain placeholder variables. Therefore we can omit the check $\mathrm{PVF}(A)$.

Similarly we infer that any object N of type `exp`, `pred` or `exp` $\rightarrow$ `pred` can only contain variables of type `exp`. We deduce therefore that the condition $p = \texttt{full} \supset \mathrm{PVF}(\Gamma(x))$ is always satisfied in the rule `cc_x` and can be omitted.

The reasoning behind these optimizations depends only on the syntactic structure of expressions and predicates and on the fact that the only dependent type is that of proofs. It is reasonable to expect that these properties hold for a very large class of first-order logics, and therefore the optimizations presented here are not specific to $\mathcal{L}$ but can be applied to many other first-order logics.

Main Reconstruction :

$$\frac{\Gamma, x : A \vdash^{r}_{p} M : B \Rightarrow \cdot}{\Gamma \vdash^{r}_{p} \lambda x.M : \Pi x{:}A.B \Rightarrow \cdot}\,\texttt{tp\_lam}$$

$$\frac{\Gamma \vdash^{r}_{p} M \Rightarrow (\Delta \,;\, C \,;\, B) \qquad \Gamma, \Delta \vdash^{r} C, A \approx_a B \Rightarrow \Psi \qquad \mathit{Dom}(\Delta) \subseteq \mathit{Dom}(\Psi)}{\Gamma \vdash^{r}_{p} M : A \Rightarrow \Psi\big|_{\mathit{Dom}(\Gamma)}}\,\texttt{tp\_app} \quad \text{M is not an abstraction}$$

Constraint Collection:

$$\frac{}{\Gamma \vdash^{r}_{p} c \Rightarrow (\cdot \,;\, \cdot \,;\, \Sigma(c))}\,\texttt{cc\_c} \qquad \frac{p = \texttt{full} \supset \mathrm{PVF}(\Gamma(x))}{\Gamma \vdash^{r}_{p} x \Rightarrow (\cdot \,;\, \cdot \,;\, \Gamma(x))}\,\texttt{cc\_x}$$

$$\frac{\Gamma \vdash^{r}_{p} M \Rightarrow (\Delta \,;\, C \,;\, \Pi x{:}A.B) \qquad p \neq \texttt{full}}{\Gamma \vdash^{r}_{p} M * \Rightarrow (\Delta, u^n : A \,;\, C \,;\, [u^n/x]B)}\,\texttt{cc\_p}$$

$$\frac{\Gamma \vdash^{r}_{p} M \Rightarrow (\Delta \,;\, C \,;\, \Pi x{:}A.B) \qquad \mathit{linear}(x, B) \qquad p \neq \texttt{full}}{\Gamma \vdash^{r}_{p} M * \Rightarrow (\Delta, u^l : A \,;\, C \,;\, [u^l/x]B)}\,\texttt{cc\_linp}$$

$$\frac{\Gamma \vdash^{r}_{p} M \Rightarrow (\Delta \,;\, C \,;\, \Pi x{:}A.B) \qquad x \in \mathrm{FV}(B) \qquad \mathrm{PVF}(A)}{\Gamma \vdash^{r}_{p} M\ N \Rightarrow (\Delta \,;\, C, N :_{\texttt{full}} A \,;\, [N/x]B)}\,\texttt{cc\_dep}$$

$$\frac{\Gamma \vdash^{r}_{p} M \Rightarrow (\Delta \,;\, C \,;\, \Pi x{:}A.B) \qquad x \notin \mathrm{FV}(B)}{\Gamma \vdash^{r}_{p} M\ N \Rightarrow (\Delta \,;\, C, N :_p A \,;\, B)}\,\texttt{cc\_indep}$$

Constraint Solving:

$$\frac{}{\Gamma \vdash^{r} \cdot \Rightarrow \cdot}\,\texttt{cs\_0} \qquad \frac{\Gamma \vdash^{r} C_1, C_2, C_3 \Rightarrow \Psi}{\Gamma \vdash^{r} C_2, C_1, C_3 \Rightarrow \Psi}\,\texttt{cs\_ord}$$

$$\frac{\Gamma \vdash^{r}_{p} M : A \Rightarrow \Psi \qquad \Psi(\Gamma) \vdash^{r} \Psi(C) \Rightarrow \Psi'}{\Gamma \vdash^{r} C, M :_p A \Rightarrow \Psi' \circ \Psi}\,\texttt{cs\_tp} \qquad \frac{A \approx_a B \Rightarrow \Psi \qquad \Psi(\Gamma) \vdash^{r} \Psi(C) \Rightarrow \Psi'}{\Gamma \vdash^{r} C, A \approx_a B \Rightarrow \Psi' \circ \Psi}\,\texttt{cs\_unif}$$

Atomic Unification:

$$\frac{}{c \approx_a c \Rightarrow \cdot}\,\texttt{au\_c} \qquad \frac{}{x \approx_a x \Rightarrow \cdot}\,\texttt{au\_x} \qquad \frac{M \approx_a M' \Rightarrow \Psi \qquad \Psi(N) \approx \Psi(N') \Rightarrow \Psi'}{M\ N \approx_a M'\ N' \Rightarrow \Psi' \circ \Psi}\,\texttt{au\_app}$$

Normal Unification

$$\frac{M \approx M' \Rightarrow \cdot}{\lambda x.M \approx \lambda x.M' \Rightarrow \cdot}\,\texttt{u\_lam} \qquad \frac{M \approx_a M' \Rightarrow \Psi}{M \approx M' \Rightarrow \Psi}\,\texttt{u\_at}$$

$$\frac{\mathit{Nonlinear}(N_i) \Rightarrow \Psi_i \qquad [\Psi_n(N_n)/x_n] \ldots [\Psi_1(N_1)/x_1]M \approx_a M' \Rightarrow \Psi'}{(\lambda x_1. \ldots \lambda x_n.M)\ N_1 \ldots N_n \approx_a M' \Rightarrow \Psi' \circ \Psi_n \circ \ldots \circ \Psi_1}\,\texttt{u\_beta}$$

$$\frac{}{u^l \approx M \Rightarrow u^l \leftarrow M}\,\texttt{u\_linst} \qquad \frac{u \notin \mathrm{FV}(M) \qquad \mathit{Nonlinear}(M) \Rightarrow \Psi}{u^n \approx M \Rightarrow \Psi \circ [u^n \leftarrow M]}\,\texttt{u\_inst}$$

Figure 16: The optimized type reconstruction algorithm.

10 The Implementation of Type Reconstruction

In this section we present the main implementation techniques that we currently use in the implementation of the type reconstruction shown in Figure 16 on the previous page. We touch issues ranging from efficient implementation of λ calculi to memory management and a portable and compact binary representation. For each implementation choice we describe how it affects the type reconstruction algorithm.

With the implementation techniques described here we were able to obtain efficient proof checking and maintain the size of the proof representation at reasonable levels, as shown in Section 11. However, our efforts in this area were limited and we therefore believe that more can be done towards smaller representations and faster proof validation.

10.1 De Bruijn Notation and Explicit Substitutions

In the abstract presentation of the type reconstruction algorithm we assumed certain meta-properties of variables and substitutions such as term identity modulo renaming of bound variables and capture-avoiding substitutions. A concrete implementation must deal with these issues explicitly. The essence of our implementation choices is best modeled abstractly by a normal-form variant of the $\lambda\sigma$ calculus with explicit substitutions and De Bruijn indices as presented in [1]. However, the presence of type reconstruction makes our implementation more complex than the one suggested by [1].

For the implementation of bound variables we use De Bruijn[2] indices. In such an implementation each bound variable is represented as a positive integer. In this representation the De Bruijn index n denotes the variable bound by the n^{th} enclosing λ-abstraction. Also, in such a representation there are variable names and the abstractions are denoted simply by the symbol λ.

The De Bruijn notation captures the essence of a term without regard to irrelevant details such as bound variable names. Thus α-equivalence is obtained directly by representation. For example, both $\lambda x.\lambda y..xy$ and $\lambda y.\lambda x.yx$ have the same De Bruijn representation, namely $\lambda\lambda\mathbf{21}$.

Although the De Bruijn notation is not easily readable it leads to simple implementations. A more serious disadvantage of the De Bruijn notation is that implementing substitution becomes more complex. For example in order to perform the substitution $[M/x]N$ it is not sufficient to substitute M for x in N. Consider the situation where there are occurrences of x in an abstraction contained in N. The correct procedure in this case is to increase by one all De Bruijn indices from M. Similar difficulties occur with β-reduction.

Another problem that we must face when implementing the type reconstruction is the implementation of substitution. We mentioned before the complications due the interaction between substitution and bound variables. In addition, the direct implementation of the substitution $[M/x]N$ is expensive because it involves copying the term M for each occurrence of x in N and in most cases most of the term N must be copied also. This causes a size explosion and excessive memory usage unless sophisticated structure-sharing mechanisms are used [20].

We chose to perform substitution in a lazy fashion, postponing a substitution until we come across a substituted variable. At that point, we lookup the substitution and we continue our current operation on the body of the substitution. Such an implementation is modeled very closely by the $\lambda\sigma$ calculus of explicit substitutions [1].

Our variant of the $\lambda\sigma$ calculus is slightly more complicated than that presented in [1] because we want it to model closely our implementation of unification. On the other hand, during type reconstruction all the terms are preserved in normal form and therefore our variant uses only one substitution operator.

We present next the syntax of the $\lambda\sigma$ calculus and proceed then to show how it is used in the implementation of the type reconstruction algorithm. See [1] for formal results about the adequacy of the calculus for implementing dependent-typed λ calculus.

At the level of syntax we replace variables with De Bruijn indices, we change the formal of term and type abstraction accordingly and we also introduce a distinguished term constant $*$. Explicit substitutions are a sequence of *locations* whose values are given by a *state*. As the terminology suggests, this indirection is exposed because we desire to model the side-effects of unification. This is the main complication added to the traditional $\lambda\sigma$ calculus. The syntax is summarized in the table below:

$$\begin{array}{llll}
\textit{Terms} & M & ::= & \mathbf{n} \mid c \mid * \mid M_1\,M_2 \mid \lambda M \\
\textit{Types} & A & ::= & a \mid A\,M \mid \Pi A.B \\
\textit{Locations} & l & & \\
\textit{Substitutions} & s & ::= & \cdot \mid l \cdot s \\
\textit{State} & \Omega & ::= & \cdot \mid \Omega, l \leftarrow M\{s\}
\end{array}$$

In preparation for the discussion of how the type reconstruction rules are influenced by the explicit substitutions we find it useful to give a few concrete implementation details. Note that substitutions as defined by the syntax above are just lists. In fact they are implemented as lists. Locations are just names (implemented as pointers) for the *car* fields of lists. The substitution state is implemented by the computer memory by storing the corresponding values in the appropriate *car* cells.

For example, the transition from substitution s in state Ω to substitution $l \cdot s$ in state $\Omega, l \leftarrow M\{s'\}$ is implemented by consing a new list cell to the list s and storing in the *car* field the values M and s'. We note here that each list cell is in fact 3 memory words. Keep in mind also that there is a significant amount of sharing among the lists and that changing the contents of a substitution location does affect many terms.

We do not show here the complete description of the reconstruction algorithm in the presence of explicit substitutions. Instead we give some general rules that govern the manipulation of explicit substitutions and we exemplify them by showing the new version of unification.

As a general rule, in the new version of the algorithm all types and terms occurring in judgments are accompanied by a substitution object that records all delayed substitutions for that particular term or type. We write $M\{s\}$ to denote that the substitution s corresponds to the term M. During type reconstruction the pair of a term and substitution is always closed. For example, if the term M contains n free variables, then it is always accompanied by a substitution that is at least of length n. Then the free variable with index $\mathbf{1}$ in M refers to the top of the substitution list, index $\mathbf{2}$ refers to the second element, and so on. This means that the pair $\mathbf{1}\{l_1 \cdot s\}$ represents the same term as the one bound to l_1 in the current state. Similarly the term $\mathbf{2}\{l_1 \cdot l_2 \cdot s\}$ represents the term bound to l_2 in the current state.

Another general rule is that all the judgments have a potential side-effect on the substitution state. Thus all the judgments are changed to take a state in input (written usually at the left of the $\vdash$ symbol) and return a new state value (written at the right end of the judgment, usually after a $\Rightarrow$ symbol).

As an introduction to explicit substitution in the indirect form, we show how the β-reduction rule would be implemented. For a gentler introduction and formal correctness proofs see [1]. Consider that the current substitution state is Ω and the β-redex is $(\lambda M)N$ accompanied by the substitution s. The β-reduction rule is shown below:

$$\Omega \vdash (\lambda M)N\{s\} \to_\beta M\{l \cdot s\} \Rightarrow \Omega, l \leftarrow N\{s\} \qquad \textit{l is new to } \Omega$$

This rule is actually very close to the actual implementation. In order to perform β-reduction we cons to s a new list cell and let l be the address of the new *car* field. Then we store the term N accompanied by its substitution in the *car* field and return the resulting substitution together with M.

As we can see the immediate cost of performing the β-reduction is very small. The hidden cost of β-reduction surfaces when, during the processing of the result, we encounter the substituted variable, or more precisely its De Bruijn index. At this point we must lookup the substitution paired with the term and continue with the processing of the substituted term. As a general rule, this lookup operation is performed whenever the term is a variable, possibly multiple times in sequence. The following two rules describe this lookup operation as a judgment $\Omega \vdash M\{s\} = M'\{s'\} \Rightarrow \Omega'$. Ignore for a moment the side-condition on the second lookup rule.

$$\frac{\Omega \vdash \mathbf{n}\{s\} = M'\{s'\} \Rightarrow \Omega'}{\Omega \vdash \mathbf{n+1}\{l \cdot s\} = M'\{s'\} \Rightarrow \Omega'} \qquad \frac{\Omega(l) \neq *\{\cdot\} \qquad \Omega \vdash \Omega(l) = M'\{s'\} \Rightarrow \Omega'}{\Omega \vdash \mathbf{1}\{l \cdot s\} = M'\{s'\} \Rightarrow \Omega'}$$

Focusing on unification we extend the above equality judgment with one that incorporates a notion of weak head reduction. This means that enough β redices are reduced so that an abstraction or an application with the head either a constant or a variable is exposed. The following rule describes the weak head reduction. Notice in the second hypothesis an instance of the β reduction rule presented above.

$$\frac{\Omega \vdash M_1\{s\} = \lambda M_1'\{s_1'\} \Rightarrow \Omega' \qquad \Omega', l \leftarrow M_2\{s\} \vdash M_1'\{l \cdot s_1'\} = M'\{s'\} \Rightarrow \Omega''}{\Omega \vdash M_1\ M_2\{s\} = M'\{s'\} \Rightarrow \Omega''} \qquad \textit{l is new to } \Omega'$$

After the unification judgment reduces both terms involved according to the equality relation defined above the resulting terms are matched according to the unification relation defined below. Consider the case when the resulting terms are both abstractions.

$$\frac{\Omega, l_1 \leftarrow c\{\cdot\}, l_2 \leftarrow c\{\cdot\} \vdash M\{l_1 \cdot s\} \approx M'\{l_2 \cdot s'\} \Rightarrow \Omega'}{\Omega \vdash \lambda M\{s\} \approx \lambda M'\{s'\} \Rightarrow \Omega'} \qquad \begin{array}{l} l_1 \textit{ and } l_2 \textit{ are new to } \Omega \\ c \textit{ is a new constant} \end{array}$$

In this case both abstractions are opened with the bound variable substituted by a newly generated constant. This trick helps preserve the invariant that all terms are closed and also ensures that, down the road, when we compare two instances of the bound variable, the unification succeeds with the constant unification rule. Note that we allocate two new locations to which we assign the same value and not just one. This decision is with an eye to the implementation of substitutions as lists. In such an implementation we cannot cons the same cell to two different lists. Exactly the same procedure is followed whenever an abstraction is opened.

Next we have the instantiation rule. We ignore here the details of atomic versus normal unification and of the occurs check optimization.

$$\frac{\Omega(l) = *\{\cdot\} \qquad l \notin M'\{s'\}}{\Omega \vdash \mathbf{1}\{l \cdot s\} \approx M'\{s'\} \Rightarrow \Omega, l \leftarrow M'\{s'\}} \qquad \textit{l is new to } \Omega$$

First note that the instantiation does not work on the term $*\{\cdot\}$ but on $\mathbf{1}\{l\}$ where l currently has the placeholder value. This is because the essence of the instantiation is to side-effect the state by changing the value associate with the location l. In fact this does not explicitly collect a substitution Ψ but immediately performs it on all the terms of the current reconstruction, and all this with just a memory store. Note that the occurs check verifies that the term $M'\{s'\}$ does not contain l and therefore the unification does not lead to circularity.

Finally we have the unification rule for application.

$$\frac{\Omega \vdash M\{s\} \approx_a M'\{s'\} \Rightarrow \Omega' \qquad \Omega' \vdash N\{s\} \approx N\{s'\} \Rightarrow \Omega''}{\Omega \vdash M\ N\{s\} \approx_a M'\ N'\{s'\} \Rightarrow \Omega''}$$

We notice here that we do not have to perform the expensive substitutions $\Psi(N)$ and $\Psi(N')$ or the substitution composition. All these are taken care of in the instantiation rules and are recorder in the substitution state.

This ends our implementation sketch. Even though we only discussed the unification, all the other judgments apply exactly the same techniques for manipulating the explicit substitutions.

10.2 Implementing the Occurs Check Optimization

The occurs check optimization presented in Section 9 uses the notion of linear placeholder variables and the function *nonlinear*. In the actual implementation the linearity attribute is not attached to the placeholder itself but to the location that contains its instantiation. For this purpose we extend the syntax of explicit substitutions with the notion of linear and non-linear locations. As locations are introduced in the collecting rule they are labelled in a similar way to the placeholder variables.

This setup permits a very efficient implementation of the function *Nonlinear* used in the instantiation rule `u_inst` and in the rule `u_beta`. Instead of computing the *Nonlinear* function eagerly we just reset the linearity attribute of the variables u in the `u_inst` rule or x in the *u_beta* rule are non-linear.

The linearity attribute of a location is used during unification in the following manner. The implementation of unification maintains a boolean flag that says whether an occurs check must be made when an instantiation is performed. The flag is initially false and is set to true whenever the unification unrolls a substitution corresponding to a non-linear location. Finally, an occurs check is made for every instantiation that happens when the flag set to true.

With this implementation, everything that is substituted for a non-linear placeholder variable is treated as non-linear, without having to compute the *Nonlinear* substitution. As an additional benefit this implementation indirectly achieves a new occurs check optimization. The first time a non-linear location is instantiated no occurs check is made. However if unification ever crosses the same location again, it sets the occurs check flag so that all variables that were substituted for the non-linear placeholder variable behave as if they were non-linear.

Thus explicit substitutions not only help amortize the cost of performing substitutions but prove to be excellent vehicles for implementing the occurs check optimization.

10.3 Memory Management

One of the main advantages of using explicit substitutions is that the memory footprint of the reconstruction algorithm is very small. The only memory allocation required is for the substitution lists. In our implementation this amounts to 3 words for each substitution performed.

Moreover if no instantiation is performed while checking a subterm then the only modification to the state is allocation and the entire memory allocated for checking that subterm can be deallocated. In such a situation the memory usage follows a stack pattern. To exploit this pattern our implementation contains a stack-based memory allocator. Whenever we start processing a subterm we place a marker on the allocator's stack. When we are done with that subterm we deallocate at once all the memory used while checking the current subterm. This implementation achieves very fast memory allocation and deallocation and also has good cache locality.

However, if we are doing reconstruction and not just checking fully-explicit terms then there can be side-effects to the state. Because instantiations create sharing relationships between substitution locations it is not correct in general to deallocate all the memory used while processing a subterm.

In order to identify the situations when we can still safely deallocate memory the implementation of unification, goal solving and type checking keep track of how many distinct placeholder variables are in the type environment Γ and in the type A to be verified. The counter is incremented with each placeholder variable introduced. The counter is decremented with each instantiation. Note that it is not possible for two distinct instantiations to instantiate the same placeholder variable. Using this simple counter we can conservatively detect the situations when no instantiation will happen during the processing of a subterm because there are no placeholder variables. In these situations we deallocate memory.

This memory optimization is fairly conservative because it fails to deallocate when there are placeholder variables but none is instantiated. In these situations the memory deallocation is delayed until all the current placeholders are instantiated. This eventually happens because at the top level there are no placeholders. Even such a conservative implementation performs well in practice as shown in Section 11.

10.4 The Flat Binary Representation

The last implementation detail that we present here is the actual representation of the syntax of LF terms and types. There are four main classes of terms: variables, constants, applications and abstractions. For the purpose of representation we ignore the difference between a terms and types.

The basic representation cell is 32 bits and has the following bit structure:

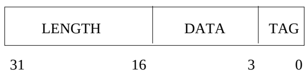

Figure 17: The basic representation cell.

The *tag* field distinguishes between a variable, a constant, an application or an abstraction. All the 4 tag values are odd because we reserve cells with even values to represent pointers to other representation cells.

In the case of a variable the data portion contains the De Bruijn index of the variable. In the case of a constant the data consists of an index into the signature table. The length field for variables and for constants is always 4 because they occupy only one cell.

An application $M_0\ M_1 \ldots M_n$ is represented as a representation cell with an application tag, the data value equal to $n + 1$ (to the number of terms in the application) and followed by the representations of M_i for $i = 0, \ldots, n$. The length field for the application is equal to the number of bytes used to represent the entire application term including its subterms. This value is redundant but it is used to speed up the breadth-first scanning of terms.

An abstraction $\lambda x_1 : A_1.\lambda x_2 : A_2.\ldots\lambda x_n : A_n.M$ is represented as a representation cell with an abstraction tag, the data field equal to $n + 1$ and the length field equal to the length of the entire abstraction. This cell is followed in order by the representations of A_i for $i = 1, \ldots, n$ and then by the representation of M.

For example, we show in Figure 18 the actual representation of the term $c_1\ c_2\ (\lambda x : a.c_3\ x)$.

32	3	APP	4	c_1	CT	4	c_2	CT	20	2	ABS	4	a	CT	12	2	APP	4	c_3	CT	4	1	VAR

Figure 18: The flat representation of the term $M = c_1\ c_2\ (\lambda x : a.c_3\ x)$. The labels c_i and a stand for integer indices into the signature table.

The main characteristic of this representation is that it avoids the use of pointers. For this reason the representation is compact and has good spatial locality of reference. Most of the time the terms are scanned depth-first exactly in the order they are laid out in memory . In the rare cases when this does not happen the *length* field allows for efficient skipping of subterms.

Due to the flatness of representation it is not possible to share subterms, and thus a size explosion occurs if a subterm has many instances in the representation. This situation does not occur often in our experiments, mainly because we do not perform substitution eagerly. However, there is one proviso to the representation rules that allow for sharing. If the low-order bit of the tag is zero, the the content of the representation cell is a pointer to a subterm. For this reason we select odd values for the variable, constant, application and abstraction tags and we reserve the even value to signal indirection.

The binary representation can be made even smaller because most of the time the *data* field uses significantly less than 13 bits, and also because the *length* field is redundant. However, we keep the representation cells 32-bit long because we occasionally want to fit machine pointers in them and also due to alignment constraints on the physical machines we use.

However, these justification do not apply to an external representation of LF terms, on the file system or in the network for example. The external representation is very similar to the internal representation with the difference that the cells do not contain a *length* field and are only 16-bit long. In the external representation the indirection pointers are replaced with even indices into an indirection table that accompanies the LF term. The *length* field is easily re-generated when the external representation is converted to the internal form.

11 Performance Measurements

In this section we compare the implicit LF representations obtained using the three erasure algorithms presented in Section 7 among themselves and with the fully-explicit representation. The parameters that we measure for each representation are the size, the reconstruction time and the dynamically allocated memory required for reconstruction. Then, we conclude the performance analysis with the presentation of the benefits gained using the optimizations presented in Section 9.

The data set used for the measurements presented in this report is obtained from our case studies of Proof-Carrying Code (PCC) [12]. PCC is a technique that a server can use to ensure with absolute certainty that the code provided by an untrusted client is safe to run, for a previously defined notion of safety. The idea behind PCC is that the client provides, together with the code, the LF representation of a formal proof of safety. In the context of PCC it is important for the proof representations to be small because they accompany the code and it is also important for the reconstruction to be fast because it is executed by the server in the critical path.

We extracted the PCC safety proof for 32 experimental programs written in the DEC Alpha assembly language. These programs include several variants of network packet filters [14], several variants of the IP checksum routine [13] and other PCC experiments presented in [13]. The sizes of these programs range from 10 to 50 assembly language instructions. In all the experiments considered in this report the LF objects to be reconstructed are implicit representations of proofs in first-order logics similar to the logic $\mathcal{L}$ presented in Section 2.

For each of the 32 proofs we considered four representations. The basic representation is the fully-explicit one. We conjecture that the performance of the type-reconstruction algorithm on explicit representations is a good estimate of the cost of LF type-checking because there is a close similarity between the number and the nature of the operations performed in both cases. The other three representations that we consider are those obtained through the local, one-bit global and global erasures from the explicit representations.

All the size measurements are of the binary representation of LF terms, as presented in Section 10.4. Recall that all LF constants are represented as indices into the signature table. We include in the size of the representation the size of the linkage information, which is a list of the constant names in the signature. The linkage table is only required if there is no preestablished order of individual constants in the signature. If we omit the size of the linkage table, the results presented in the rest of this section are improved.

The size of the explicit representation serves as the basis of evaluation for the erasure algorithms. For each of the implicit representations we measure the improvement in the reconstruction time and the size of the dynamically allocated memory used during type reconstruction. All the measurements were done on a DEC Alpha workstation with a 175-MHz Alpha 21064 processor and a 2-Mbytes board-level cache. The measurements are performed with a warm cache and we average the data over at least consecutive 1000 runs.

Table 3 shows absolute performance data for the fully-explicit representation and the one-bit global representation. We only show in this table only a handful of representative cases. The experiment entitled *Cap* is the resource access service from [14]. The packet filters 3 and 4 are also described in [13]. The *Ping* experiment is the largest PCC experiment to date and involves checking both safety and liveness properties of a simple implementation of the ping network protocol. The experiment *Safe ML* is described in [12]. In the rest of this section we shall focus on relative performance comparisons. All the reported results are based on 32 experiments that include all those described here and others that are similar but generally smaller.

In order to compute the reduction in the representation size and reconstruction time, we com-

Experiment	Explicit			One-bit Global		
	Size (bytes)	Rec. Time (ms)	Rec. Heap (bytes)	Size (bytes)	Rec. Time (ms)	Rec. Heap (bytes)
Cap	3049	7.41	5896	257	1.14	4984
IP Checksum	23198	56.84	20088	1037	5.77	15432
Packet Filter 3	15898	36.75	17648	561	2.80	13136
Packet Filter 4	14502	33.57	15016	547	2.60	10216
Ping	181527	458.27	53376	4149	24.86	38928
Safe ML	13758	34.72	12152	707	4.30	11144

Table 3: Absolute performance data for the type reconstruction of fully-explicit and one-bit global representations.

pute the geometric means of corresponding ratios. The resulting values are shown in Table 4. Based on these results we conclude that all implicit representations achieve significant reductions in the size of representation and the validation time. The memory required for validation is reduced by only approximately 35%. Such a small improvement in memory usage is motivated by the fact that the more implicit the representation, the more memory is required for holding the reconstructed subterms, and the fewer opportunities to deallocate memory eagerly (see the description of the memory optimization in Section 9).

The major surprise revealed by the data from Table 4 is that the global representation algorithm is not significantly more effective than the one-bit global algorithm in terms of representation size and reconstruction time. In addition, it increases slightly the amount of memory required for reconstruction. The main reason for such a dismal improvement over the one-bit algorithm is that the one-bit erasure succeeds in removing the vast majority of redundant subterms and achieves a near optimal representation, leaving very little room for improvement. Interestingly, the one-bit global erasure algorithm achieved optimal results on the largest experiments we performed. These observation, combined with the substantial complexity of the global erasure algorithm, lead to the conclusion that the global erasure algorithm is not of practical value for the implicit representation of first-order proofs.

Ratio	Local	One-bit	Global
Representation size	5.08	15.42	15.83
Reconstruction time	3.71	7.71	7.81
Reconstruction space	1.31	1.36	1.32

Table 4: The reduction in representation size, reconstruction time and space due to the implicit representation.

While the ratios shown in Table 4 clearly demonstrate the effectiveness of the implicit representations for first-order logic proofs, they do not show the whole power of reconstruction. We observed that the improvements due to implicit representations are larger in relative terms as the proofs get larger. This suggests that a simple ratio does not convey the true performance benefit to be gained through reconstruction.

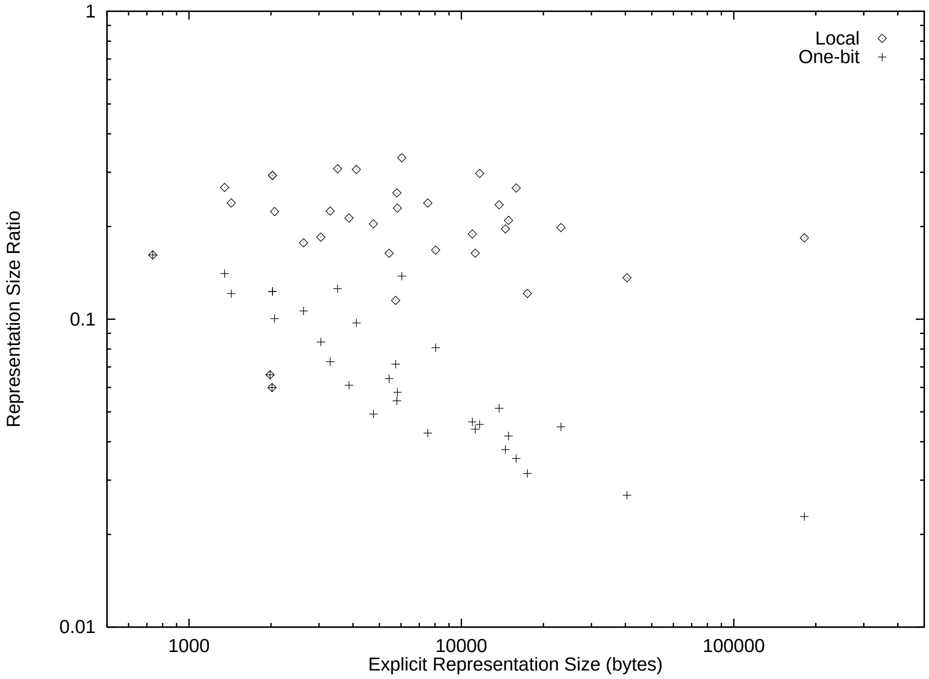

Figure 19: The representation-size reduction as a function of the explicit representation on logarithmic scale.

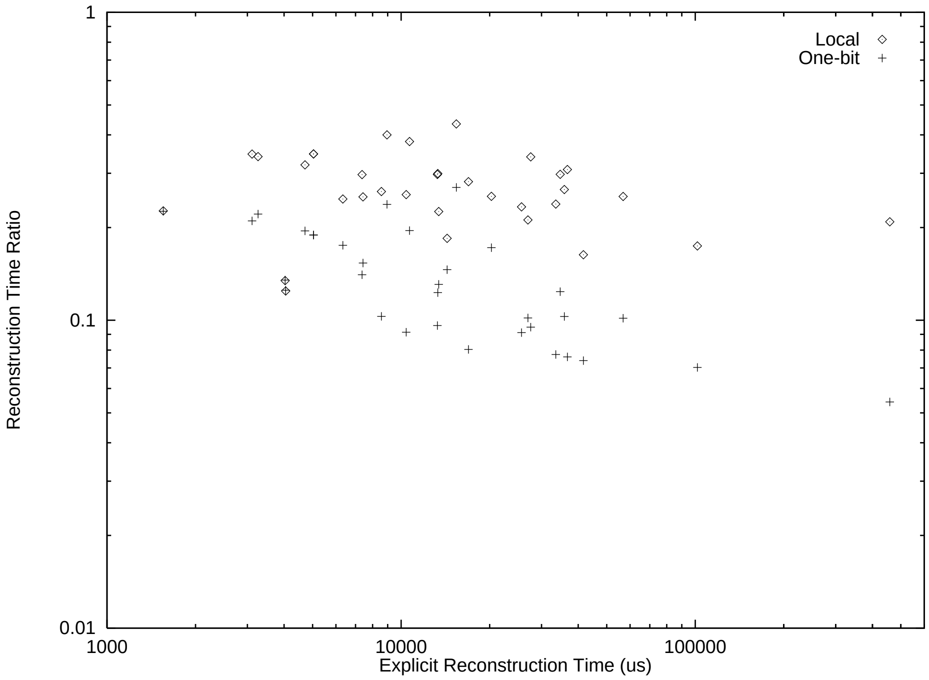

Figure 20: The reconstruction-time reduction as a function of the explicit reconstruction on logarithmic scale.

In Figure 19 and 20 we plot on logarithmic scale the improvement ratios of the representation size and the validation time as a function of the explicit representation size and reconstruction time, for the 32 experiments considered. These figures show that, at least for the one-bit global representation, the improvements are superscalar. We have computed logarithmic correlations of the implicit and explicit values of the representation size and the reconstruction time. The results suggest that, for the local erasure, the improvements are scalar and are accurately described by the simple ratios shown in Table 4. For the one-bit global erasure, however, we have found that the following formulas are more accurate for the range of term sizes covered by our experiments:

$$\textit{One-bit size} = 2.03 * \textit{Explicit size}^{0.60} \qquad \text{with a correlation of } 0.94$$

$$\textit{One-bit time} = 1.81 * \textit{Explicit time}^{0.72} \qquad \text{with a correlation of } 0.95$$

These formulas suggest that as the size of the problem increases, the improvement ratio of the one-bit global representation compared to the fully-explicit representation also increases. More experiments with large LF terms are required to substantiate these findings.

11.1 Effectiveness of Optimizations

The results of the previous section are obtained with all the optimizations presented in Section 9. In the rest of this section we show our findings regarding the effectiveness of these optimizations. To evaluate an optimization we measure the reconstruction time and space with only that optimization disabled and then with all the optimizations enabled, for the four LF representations. We then compute the geometric mean, over the 32 experiments, of the ratio between the optimal value of the performance parameter and the value with the optimization disabled. We report, in Table 5 and 6, the result of subtracting from 100% the mean ratio for the reconstruction time and space respectively. A value of 10% in these tables means that, on the average, the optimization reduces the reconstruction time or space to 90% of the original value.

Optimization	Explicit	Local	One-bit	Global
Occurs Check	0.07%	24.23%	43.75%	43.88%
Memory Management	-1.46%	-0.59%	-0.44%	0.11%
Side-Condition	13.62%	8.90%	1.73%	1.96%
First-Order Logic	8.51%	5.40%	0.94%	1.72%
All	15.91%	30.49%	44.16%	44.71%

Table 5: The effect of the optimizations on the reconstruction time. Each value is obtained by subtracting from 100% the geometric mean over 32 experiments of the ratio between the optimal time and the time with one optimization disabled.

We notice that the occurs check optimization has a major effect on the reconstruction time and it is more effective for the representation with more implicit subterms. This is because the occurs check is only required for the instantiation of implicit subterms. The occurs check optimization has also a minor improvement on the memory used during reconstruction, because it avoids memory allocation that is normally part of the occurs check.

The occurs check is one of the most expensive operations performed during type reconstruction. To assess the cost of occurs check for type reconstruction we measured the reconstruction time for each of the four LF representations in three separate circumstances: with the occurs check turned

Optimization	Explicit	Local	One-bit	Global
Occurs Check	0.00%	0.64%	0.77%	0.77%
Memory Management	88.92%	66.91%	35.56%	35.86%
Side-Condition	3.96%	0.00%	0.00%	0.00%
First-Order Logic	0.00%	0.00%	0.00%	0.00%
All	88.99%	67.12%	36.04%	35.35%

Table 6: The effect of the optimizations on the reconstruction space. Each value is obtained by subtracting from 100% the geometric mean over 32 experiments of the ratio between the optimal space and the space with one optimization disabled.

off[2], with it turned always on and with the optimized occurs check presented in Section 9. In Table 7 we show for each representation the average percentage of time spent doing the occurs check. We see that, in the unoptimized case, the reconstruction algorithm spends about 45% of the time performing the occurs check. However, with the simple linearity optimization the cost of the occurs check is reduced to less than 2%, making further optimization uninteresting.

Parameter	Explicit	Local	One-bit	Global
Cost of occurs check	0.09%	24.70%	44.28%	44.67%
Optimized cost of occurs check	0.02%	0.63%	2.01%	1.40%

Table 7: The percentage of time spent doing the occurs check, in the unoptimized and optimized cases.

The memory usage optimization has a major effect on the dynamically allocated memory used during reconstruction. Recall that memory deallocation can be done only after processing a subterm whose type contains no placeholder variables. The more implicit the representation, the more placeholder variables are introduced in types. This explains the diminishing effect of the memory optimization as the representation becomes more implicit. Even so, the memory usage is reduced by approximately 35% in the case of the one-bit global representation. From the viewpoint of the reconstruction time, the deallocation operation increases the overall running time by about 0.44% for the one-bit global representation.

The side-condition and the first-order logic optimizations do not bring significant improvements in the reconstruction space and time, especially in the case of the implicit representations. Recall that the effect of these optimizations is to reduce the cost of checking the side-condition when collecting an explicit argument that occurs in the result type. Such arguments are reduced to a minimum by the erasure algorithms, eliminating the opportunities for these optimizations. Nevertheless we consider that these optimizations are useful because they reduce the complexity of the reconstruction algorithm. If these optimizations are applied then the reconstruction algorithm does not require the procedures $\mathrm{PVF}(A)$ and the more complex $\mathrm{PVF}(\Gamma(\mathrm{FV}(N))$.

In addition of the effect of individual optimizations we also measured their combined effect. For this purpose we measured the reconstruction time and space with all the optimizations enabled and

[2]This makes reconstruction unsound but is useful as a basis for performance comparisons.

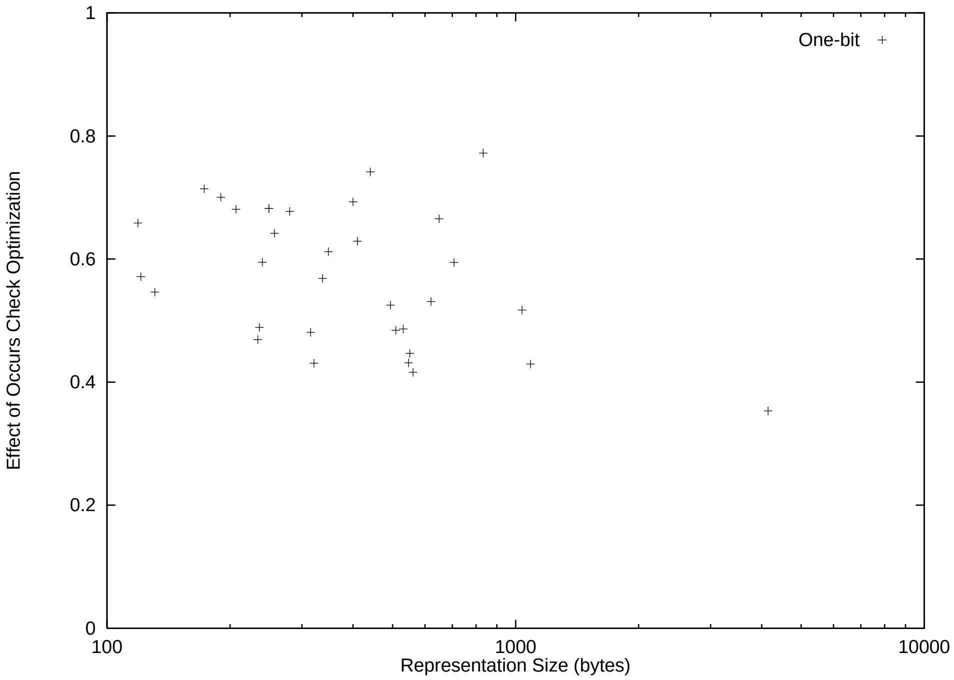

Figure 21: The correlation between the ratio of the optimized reconstruction time to the time without the occurs check optimization and the size of the term.

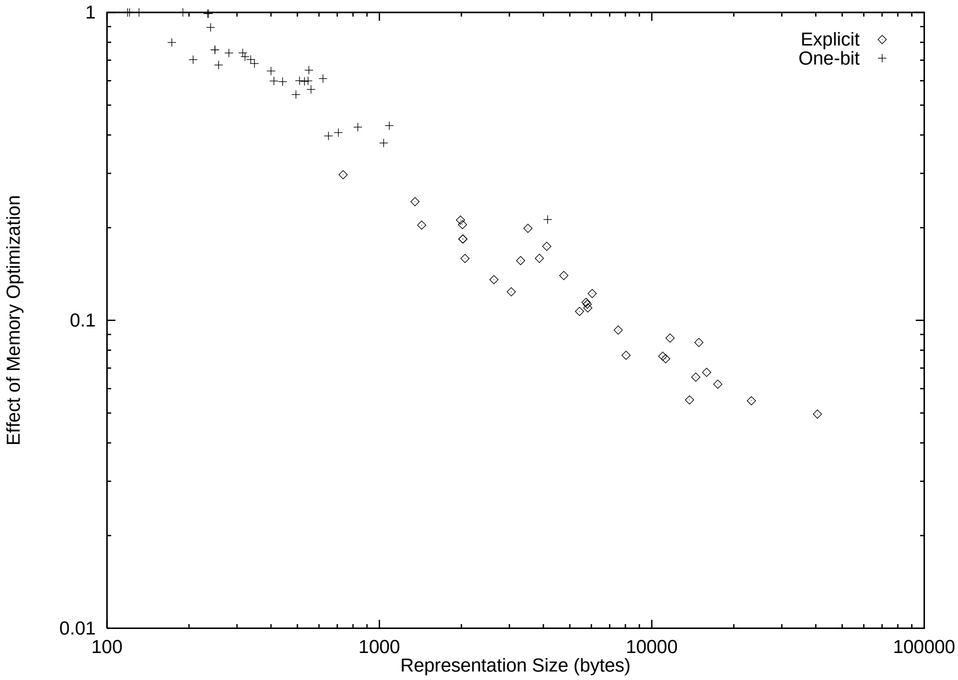

Figure 22: The correlation between the ratio of the optimized reconstruction space to the space without the memory optimization and the size of the term.

then with them disabled. We notice that with all the optimizations enabled the reconstruction time is reduced by 44% on the average and the reconstruction space by 36%. These effects are mostly due to the occurs check optimization for the time improvement and the memory optimization for the memory usage improvement.

We noticed that the two most important optimizations, the occurs check and the memory optimization, have the tendency to achieve better improvements for larger benchmarks. We plotted in Figure 21 the ratio between the optimal reconstruction time to the time with the occurs check optimization disabled as a function of the term size, on a logarithmic scale. We notice on the graph that, indeed, the occurs check optimization becomes more effective for larger terms. A similar and more pronounced correlation exists between the effectiveness of the memory optimization and the size of the term, both for the explicit representation and the one-bit global representation (Figure 22). This suggests that these optimizations are more effective for larger terms than for small ones, although more large experiments are required to ascertain this tendency.

11.2 Correlation between the Reconstruction Time and Term Size

Although not directly related to the performance of the erasure algorithms or the effectiveness of optimizations, we briefly discuss the relation between the term size and the reconstruction time. We plot in Figure 23 this correlation for the explicit representation and the one-bit global representation. As can be seen from the figure, in our current experiments involving validation of first-order logic proofs, we observe a linear dependency of the reconstruction time with the size of the proof representation. This property is true both for the fully-explicit representation and the one-bit global implicit representation.

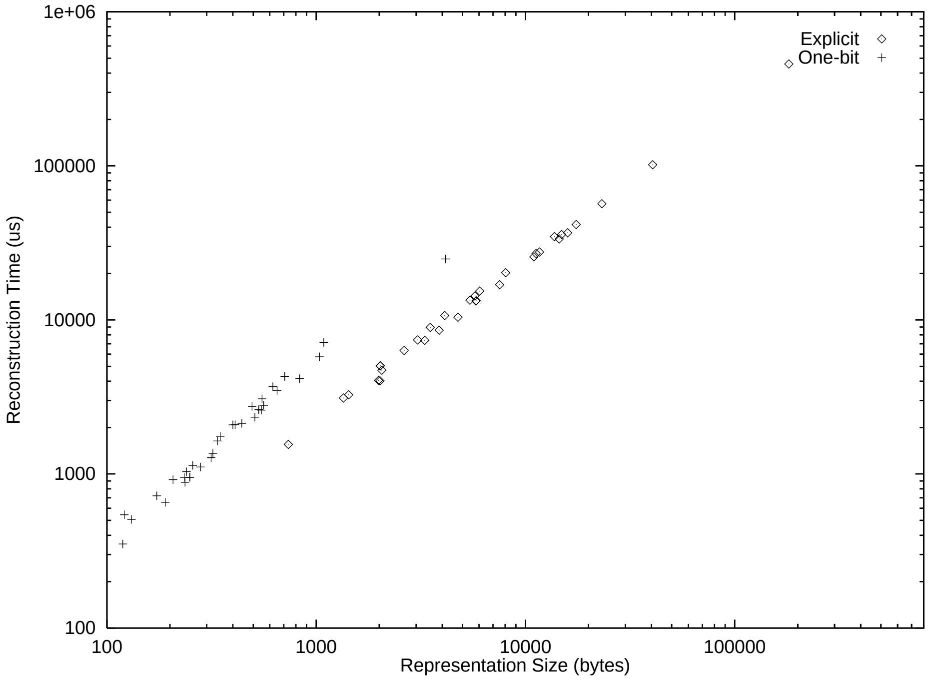

Figure 23: The correlation between the representation size and the reconstruction time.

These results are not possible to generalize to all LF reconstruction tasks because, in the worst case, the complexity of the reconstruction is superexponential. This is hinted by the fact that reconstruction involves β-reduction which has been shown by Statman to be not elementary recursive [19].

There are several aspects of using reconstruction for validating first-order logic proofs that suggest a better behavior that in the general case. First, the reconstruction algorithm only works on canonical terms. Because of substitution in types the reconstruction creates β-redices. Such situations occur, in the context of first-order logic, only when checking instances of rules for universal and existential quantification. In these cases, the β-reduction substitutes a LF term of type `exp` in the body of the redex and therefore, the result cannot contain additional β-redices. This suggests that it is never the case that the β-reductions are multiplicated in avalanche, and therefore that Statman's theorem does not apply in this case.

However, that the reconstruction effort is linear in the term size remains a surprise that we attribute to the fact that our experiments do not make extensive use of quantification. We expect similar situations to occur in most practical situations involving PCC and first-order logics.

12 Related Work

We are not the first to address the elimination of redundancy in proof representations. Miller [11] approaches the problem by noting that it should be possible to greatly simplify the representation of proofs in classical logic by simply recording the substitutions that were performed when building the proof instead of recording all the basic proof steps. For this purpose he introduces a new proof structure called expansion trees. While expansion tree representation of proofs are indeed compact, proof checking is more expensive because some (directed) search still needs to be done.

The redundancy of representation that we noticed for LF proof representation generates usability problems for implementations of logic programming languages or proof assistants based on LF or related type-systems. Without a form of implicit representation the interaction with these systems is very verbose. This led the implementors to consider implicit representations that are similar in spirit to the one presented in this report.

For example, the LEGO [7, 18] and Coq [3] proof assistants implement algorithms for argument synthesis and term reconstruction. However, for the task of representing proofs these algorithms are both less effective, in the sense that fewer proof subterms can be omitted, and less efficient. They are less effective because the only arguments that can be omitted from an application are those that can be inferred from other arguments of the same application. This means that an application of the term $\texttt{eqId} : \Pi e : \texttt{exp}.\texttt{pf}\ (=\ e\ e)$ must always be explicit even though the argument can usually be recovered from the context. These algorithms are also less efficient than ours because they implement more general unification algorithms that lift some of the syntactic restrictions that we impose.

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 arguments can be recovered also from the context, not only from the other arguments. In fact, the Elf reconstruction algorithm is much more powerful than ours because it does not impose any restrictions on which types and terms can be missing from the proof. In particular, the whole proof might be missing, in which case Elf tries to reconstruct it. To achieve this level of flexibility, Elf type reconstruction employs depth-first search and higher-order unification with dependent types [4], followed by a constraint solver [16, 15]. An alternative characterization of our proof checking process is as a proof reconstruction instance where enough of the structure of the proof is given to avoid the need for search, to reduce higher-order unification to a simple extension of first-order unification that respects bound variables and to ensure that all constraints that are generated have the simple rigid-rigid or flex-rigid form that can be solved eagerly. These simplifications enable us to implement a more efficient reconstruction algorithm.

Because higher-order unification is undecidable [6] and expensive in general, Miller [10] proposes syntactic restrictions so that to ensure that the only unification problems that occur can be solved by a simple extension of the first-order unification, as in our case. This approach is the approach taken in the language L_λ. Unfortunately, these restrictions are too strict for our purposes because they prevent the free use of higher-order abstract syntax for the representation of predicates and proofs. This obstacle can be overcome by implementing term level substitution, but only at the expense of more complicated programs and a significant loss in performance [8, 9]. We too recognize the benefits of implementing the reconstruction algorithm for a syntactically-restricted subset of LF, but we do it in such a way that any (explicit) LF term can still be type-checked in our system. This does not limit the available language and programming techniques but might reduce the effectiveness of the term compaction algorithms in certain cases.

Even though there has been a significant amount of work in term reconstruction, we were not able to identify previous work on the dual problem of proof compaction.

13 Conclusion

In this report we have presented algorithms for the representation and validation of logical proofs. Such techniques are important in situations where logical proofs must be manipulated explicitly such as in the case of a theorem prover that generates witnesses of successful derivations, or in the case of Proof-Carrying Code.

The algorithms presented here are derived from the representation and typechecking algorithm of the Edinburgh Logical Framework. The major advantages that we achieve with this choice is that the algorithms are parameterized by the logic of interest and therefore only one implementation is required for a large range of logics. In particular this is the case for all our current PCC experiments, ranging from extensions to operating system kernel to extensions to safe programming languages to active network components. Another major benefit of LF is that all the complications due to parametric and hypothetical proofs are shifted to the framework by using higher-order syntax.

Even though the pure LF representation is not optimal in terms of size and validation time, our implicit representations achieve much improved results. It is an interesting exercise to analyze the behavior of the representation and reconstruction algorithms for each rule in a given logic. Such an analysis reveals that for all first-order logic constructs, both the implicit representation and reconstruction are optimal in the sense that nothing more could be saved by algorithms that are specialized to the given logic. This practically says that we achieve performance comparable to special purpose algorithms by using general algorithms parameterized by the logic.

A major contribution of the work presented in this report is a sound reconstruction algorithm that is able to reconstruct and typecheck LF terms with missing subterms. The power of the algorithm consists on being able to work with implicit representations that are 15 times smaller than the original LF representations. A notable feature of the reconstruction algorithm is that the reconstruction time is much smaller than the time required for checking the full LF representation, in the cases when the type is known to be well-formed. We have measured reductions of about on order of magnitude in the typechecking time and of about 35% in the space required for typechecking when using explicit substitutions.

Accompanying the reconstruction algorithm are several algorithms that can be used to produce implicit representations of full LF terms. Among these, two are very simple and syntax directed and produced near optimal results.

On the implementation front, we adapted the explicit substitution method to the case of dependent type reconstruction and we integrated it with an effective occurs-check optimization and a memory management optimization. Finally we described a portable binary encoding that attempts to further minimize the size of LF terms.

Although we have made significant progress in the size of the representation and the validation effort over pure LF, there is certainly room for further improvement. To discover these opportunities, we want to continue experimenting with the representation and reconstruction algorithms for more diverse and for larger problems. The purpose of such experiments would be to further substantiate our expectations that the improvements due to the implicit representations are super-linear and that the reconstruction time is linear on the problem size.

One interesting direction for future work is to experiment with representations of proofs in higher-order logics. It is likely that the improvements that we measured for first-order logics do not translate directly to higher-order logics, and that even more sophisticated reconstruction algorithms can be devised. It would be interesting to see if the reconstruction time is a linear function of the term size also in the case of proofs in higher-order logics.

We have claimed that the local and the one-bit global erasure algorithms can be at the base of efficient representation algorithms that achieve incremental construction of proof representations without ever generating the full representation. An interesting experiment would be to instrument an actual theorem prover to maintain proof representations of the intermediary results and to combine them as it progresses toward the final proof.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

[13] Necula, G. C., and Lee, P. Proof-carrying code. Technical Report CMU-CS-96-165, Computer Science Department, Carnegie Mellon University, Dec. 1996. Also appeared as FOX memorandum CMU-CS-FOX-96-03.

[14] 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.

[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.

[19] Statman, R. The typed λ-calculus is not elementary recursive. *Theoretical Computer Science 9* (1979), 73–81.

[20] Wadsworth, C. P. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Oxford University, 1971.