Proof terms for simply typed higher order logic

Stefan Berghofer* and Tobias Nipkow

Technische Universität München, Institut für Informatik
http://www.in.tum.de/~berghofe/
http://www.in.tum.de/~nipkow/

Abstract. This paper presents proof terms for simply typed, intuitionistic higher order logic, a popular logical framework. Unification-based algorithms for the compression and reconstruction of proof terms are described and have been implemented in the theorem prover Isabelle. Experimental results confirm the effectiveness of the compression scheme.

1 Introduction

In theorem provers based on the LCF approach, theorems can only be constructed by a small set of primitive inference rules. Provided the implementation of these rules is correct, all theorems obtained in this way are sound. Hence it is often claimed that constructing an explicit proof term for each theorem is unnecessary. This is only partially true, however. If the core inference engine of a theorem prover is relatively large, correctness is difficult to ensure. Being able to verify proof terms by a small and independent proof checker helps to minimize the risks. Moreover, a precise notion of proof terms facilitates the exchange of proofs between different theorem proving systems. Finally, proof terms are a prerequisite for proof transformation and analysis or the extraction of computational content from proofs. Probably the most prominent application these days is proof-carrying code [5], a technique that can be used for safe execution of untrusted code. For these reasons we have extended Isabelle [9] with proof terms. However, apart from the actual implementation, our work is largely independent of Isabelle and most of this paper deals with the general topic of proof terms for simply typed, intuitionistic higher order logic (abbreviated to λHOL below), Isabelle's meta logic. Because other logics (e.g. full HOL) can be encoded in this meta logic, this immediately yields proof terms for those logics as well.

We start with a disclaimer: the idea of proof terms based on typed λ-calculus has been around for some time now and is the basis of a number of proof assistants for type theory, for example Coq [2]. Even more, with the advent of "pure type systems" and the λ-cube [1], it became clear what proof terms for λHOL look like in principle (although this seems to have had little impact on the HOL world). What we have done is to re-introduce the strict syntactic separation between terms, types, and proofs to make it more amenable to readers from a simply typed background. Thus our presentation of proof terms can be seen as a

* Supported by DFG Graduiertenkolleg *Logic in Computer Science*

partial evaluation of the corresponding pure type system, i.e. separating the layers. Things are in fact a bit more complicated due to the presence of schematic polymorphism in our term language.

The main original contribution of the paper is a detailed presentation of proof compression. A naive implementation of proof terms results in proofs of enormous size because with every occurrence of a proof rule the instantiations for its free variables are recorded. Thus it is natural to try and leave out some of those terms and to reconstruct them by unification during proof checking. Necula and Lee [6] have presented a scheme for proof compression in LF, another logical framework based on type theory. They analyze the proof rules of the object logic statically to determine what can be reconstructed by a weak form of unification. In contrast, we do a dynamic analysis of each proof term to determine what can be dropped. Reconstruction of missing information by unification is also available in other systems, e.g. Elf [12, 11], but none of them offers an automatic dynamic compression algorithm.

There has also been work on recording proofs in the HOL system [3, 14], but it is firmly based on a notion of proof that directly reflects the implementation of inferences as calls to ML functions. These proof objects lack the conciseness of λ-terms and it is less clear how to compress them other than textually.

We start by presenting the logical framework (§2) and its λ-calculus based proof terms (§3). In order to shrink the size of proofs we introduce partial proofs (§4), show how to collect equality constraints from a (partial) proof (§4.1), how to solve these constraints (§4.2) (to check that the proof is correct), and how to generate partial proofs from total ones, i.e. how to compress proofs (§4.3).

2 The logical framework

In a nutshell, Isabelle's meta logic [9, 8] is the minimal higher order logic of implication and universal quantification over simply typed λ-terms including schematic polymorphism. Thus types are first order only, which makes type reconstruction decidable. A type τ is either a variable α or a compound type expression $(\tau_1, \ldots, \tau_n)tc$, where tc is a type constructor and n is its arity. The (infix) constructor $\to$ for function types has arity 2. We assume implicitly that all types are well-formed, i.e. every type constructor is applied to the correct number of arguments. The set t of terms is defined in the usual way by

$$t = x \mid c \mid \lambda x :: \tau.\ t \mid t\ t$$

Formulae are terms of the primitive type prop. The logical connectives are:

universal quantification	$\bigwedge$	$::$	$(\alpha \to \mathsf{prop}) \to \mathsf{prop}$
implication	$\Longrightarrow$	$::$	$\mathsf{prop} \to \mathsf{prop} \to \mathsf{prop}$

Now we show how an object logic is formalized in this meta logic. As an example we have chosen a fragment of HOL. First we introduce new types and constants for representing the connectives of this logic:

Tr	$::$	$\mathsf{bool} \to \mathsf{prop}$	$\forall$	$::$	$(\alpha \to \mathsf{bool}) \to \mathsf{bool}$
$\longrightarrow$	$::$	$\mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}$	$\exists$	$::$	$(\alpha \to \mathsf{bool}) \to \mathsf{bool}$

Here, bool is the type of object level propositions. The function Tr establishes a connection between meta level and object level truth values: the expression $\mathsf{Tr}\ P$ should be read as "P is true". The application of Tr is occasionally dropped when writing down formulae. The inference rules for the meta logic consist of the usual introduction and elimination rules and are shown in §3.1 below.

Inference rules of object logics are usually written like this:

$$\frac{\phi_1 \quad \dots \quad \phi_n}{\psi}$$

In our meta logic they become nested implications:

$$\phi_1 \Longrightarrow \dots \Longrightarrow \phi_n \Longrightarrow \psi$$

Here are some examples:

$$\begin{array}{lll}
\mathsf{impI} & : \bigwedge A\ B. & (\mathsf{Tr}\ A \Longrightarrow \mathsf{Tr}\ B) \Longrightarrow \mathsf{Tr}\ (A \longrightarrow B) \\
\mathsf{impE} & : \bigwedge P\ Q\ R. & \mathsf{Tr}\ (P \longrightarrow Q) \Longrightarrow \mathsf{Tr}\ P \Longrightarrow (\mathsf{Tr}\ Q \Longrightarrow \mathsf{Tr}\ R) \Longrightarrow \mathsf{Tr}\ R \\
\mathsf{allI} & : \bigwedge P. & (\bigwedge x.\ \mathsf{Tr}\ (P\ x)) \Longrightarrow (\mathsf{Tr}\ (\forall x.\ P\ x)) \\
\mathsf{allE} & : \bigwedge P\ x\ R. & \mathsf{Tr}\ (\forall x.\ P\ x) \Longrightarrow (\mathsf{Tr}\ (P\ x) \Longrightarrow \mathsf{Tr}\ R) \Longrightarrow \mathsf{Tr}\ R \\
\mathsf{exI} & : \bigwedge P\ x. & \mathsf{Tr}\ (P\ x) \Longrightarrow \mathsf{Tr}\ (\exists x.\ P\ x) \\
\mathsf{exE} & : \bigwedge P\ Q. & \mathsf{Tr}\ (\exists x.\ P\ x) \Longrightarrow (\bigwedge x.\ \mathsf{Tr}\ (P\ x) \Longrightarrow \mathsf{Tr}\ Q) \Longrightarrow \mathsf{Tr}\ Q
\end{array}$$

Note that the introduction rules impI and allI are for object level implication and universal quantification are expressed by simply referring to the meta level counterpart of these connectives. The expression $\bigwedge x.\ \phi$ is just an abbreviation for $\bigwedge (\lambda x.\ \phi)$, and similarly for $\forall$ and $\exists$.

3 Proof terms

3.1 Basic concepts

The set of proof terms p is defined as follows:

$$p \ = \ h \mid c_{\overline{[\tau_n/\alpha_n]}} \mid \lambda h : \phi.\ p \mid \lambda x :: \tau.\ p \mid p\ p \mid p\ t$$

The letters h, c, x, t, ϕ and τ denote proof variables, proof constants, term variables, terms of arbitrary type, terms of type prop and types, respectively. Note that terms, types and proof terms are considered as separate concepts. This is in contrast to type theoretic frameworks, where these concepts are identified. We will write $\Gamma \vdash p : \phi$ for "p is a proof of ϕ in context Γ", where ϕ is a term of type prop, representing the logical proposition proved by p. The context Γ associates a proof variable with a term representing the proposition whose proof it denotes, and a term variable with its type. We require each context to be well-formed, i.e. every variable is associated with at most one term or type. Proof constants correspond to axioms or already proved theorems. The environment Σ maps proof constants to terms representing propositions. Our language of proof terms allows abstraction over proof and term variables, as well as application of proofs to proofs and terms. The abstractions correspond to the introduction of $\bigwedge$ and $\Longrightarrow$, while applications correspond to the elimination of these connectives. In contrast to polymorphic λ-calculi, no explicit application and abstraction is provided for types. To achieve a certain degree of polymorphism, we allow $\Sigma(c)$ to contain free type variables and introduce the notation $c_{[\overline{\tau_n/\alpha_n}]}$ to specify a suitable instantiation for them. The notion of provability can now be defined inductively as follows:

$$\frac{}{\Gamma, h : \phi, \Gamma' \vdash h : \phi} \qquad \frac{\Sigma(c) = \phi}{\Gamma \vdash c_{[\overline{\tau_n/\alpha_n}]} : \phi[\overline{\tau_n/\alpha_n}]}$$

$$\frac{\Gamma, h : \phi \vdash p : \psi \quad \Gamma \vdash \phi :: \mathsf{prop}}{\Gamma \vdash (\lambda h : \phi.\ p) : \phi \Longrightarrow \psi} \qquad \frac{\Gamma, x :: \tau \vdash p : \phi}{\Gamma \vdash (\lambda x :: \tau.\ p) : \bigwedge x :: \tau.\ \phi}$$

$$\frac{\Gamma \vdash p : \phi \Longrightarrow \psi \quad \Gamma \vdash q : \phi}{\Gamma \vdash (p\ q) : \psi} \qquad \frac{\Gamma \vdash p : \bigwedge x :: \tau.\ \phi \quad \Gamma \vdash t :: \tau}{\Gamma \vdash (p\ t) : \phi[t/x]}$$

The judgement $\Gamma \vdash t :: \tau$ used above expresses that the term t has type τ in context Γ. We will not give a formal definition of this judgement here, since it is well-known from simply typed lambda calculus.

3.2 Representing backward resolution proofs

This section explains how proof terms are constructed for proofs that are built up backwards by higher-order resolution as described by Paulson [8] and implemented in Isabelle. Although Isabelle also has LCF-like functions for forward proofs corresponding to the above inference rules, most proofs are constructed backwards without recourse to the forward rules. We now show how to augment these backward steps by proof terms. Thus the functions for backward resolution proofs need no longer be part of the trusted kernel of Isabelle.

In Isabelle, proof states are represented by theorems of the form

$$\psi_1 \Longrightarrow \cdots \Longrightarrow \psi_n \Longrightarrow \phi$$

where ϕ is the proposition to be proved and $\psi_1, \ldots, \psi_n$ are the remaining subgoals. Each subgoal is of the form $\bigwedge \overline{x}.\ \overline{A} \Longrightarrow P$, where $\overline{x}$ and $\overline{A}$ is a context of parameters and local assumptions.

Resolution
A proof of a proposition ϕ starts with the trivial theorem $\phi \Longrightarrow \phi$ whose proof term is $\lambda v : \phi.\ v$. The initial proof state is then refined successively using the resolution rule

$$\begin{array}{c} \dfrac{P_1 \ \ldots\ P_m}{C}\ R \\[2ex] \dfrac{P'_1 \ \ldots\ P'_i \ \ldots\ P'_{m'}}{C'}\ R' \end{array} \quad \longmapsto \quad \theta \left(\frac{P'_1 \quad \ldots \quad P'_{i-1} \quad P_1 \quad \ldots \quad P_m \quad P'_{i+1} \quad \ldots \quad P'_{m'}}{C'} \right)$$

$$\text{where} \quad \theta\ C = \theta\ P'_i$$

until a proof state with no more premises is reached. When refining a proof state having the proof term R' using a rule having the proof term R, the proof term for the resulting proof state can be expressed by

$$\theta\ (\lambda \overline{q_{i-1}}\ \overline{p_m}.\ R'\ \overline{q_{i-1}}\ (R\ \overline{p_m}))$$

where θ is a unifier of C and P_i'. The first $i-1$ abstractions are used to skip the first $i-1$ premises of R'. The next m abstractions correspond to the new subgoals introduced by R.

Proof by assumption
If the formula P_j in a subgoal $\bigwedge \overline{x_k}.\ \overline{P_n} \Longrightarrow P_j$ of a proof state having the proof term R equals one of the assumptions in $\overline{P_n}$, this subgoal trivially holds and can therefore be removed from the proof state

$$\frac{Q_1\ \ldots\ Q_{i-1} \quad \bigwedge \overline{x_k}.\ \overline{P_n} \Longrightarrow P_j \quad Q_{i+1}\ \ldots\ Q_m}{C}\ R \quad \longmapsto \quad \frac{Q_1\ \ldots\ Q_{i-1} \quad Q_{i+1}\ \ldots\ Q_m}{C}$$

$$\text{where} \quad 1 \leq j \leq n$$

The proof term of the new proof state is obtained by supplying a suitable projection function as an argument to R:

$$\lambda \overline{q_{i-1}}.\ R\ \overline{q_{i-1}}\ (\lambda \overline{x_k}\ \overline{p_n}.\ p_j)$$

Lifting rules into a context
Before a subgoal of a proof state can be refined by resolution with a certain rule, the context of both the premises and the conclusion of this rule has to be augmented with additional parameters and assumptions in order to be compatible with the context of the subgoal. This process is called *lifting*. Isabelle distinguishes between two kinds of lifting: lifting over assumptions and lifting over parameters. The former simply adds a list of assumptions $\overline{Q_n}$ to both the premises and the conclusion of a rule:

$$\frac{P_1 \quad \ldots \quad P_m}{C}\ R \quad \longmapsto \quad \frac{\overline{Q_n} \Longrightarrow P_1 \quad \ldots \quad \overline{Q_n} \Longrightarrow P_m}{\overline{Q_n} \Longrightarrow C}$$

The proof term for the lifted rule is

$$\lambda \overline{r_m}\ \overline{q_n}.\ R\ (\overline{r_m\ \overline{q_n}})$$

where the first m abstractions correspond to the new premises (with additional assumptions) and the next n abstractions correspond to the additional assumptions.

Lifting over parameters replaces all free variables a_i in a rule $R\,[\overline{a_k}]$ by new variables a_i' of function type, which are applied to a list of new parameters $\overline{x_n}$. The new parameters are bound by universal quantifiers.

$$\frac{P_1\,[\overline{a_k}] \quad \ldots \quad P_m\,[\overline{a_k}]}{C\,[\overline{a_k}]}\ R\,[\overline{a_k}] \quad \longmapsto \quad \frac{\bigwedge \overline{x_n}.\ P_1\left[\overline{a_k'\ \overline{x_n}}\right] \quad \ldots \quad \bigwedge \overline{x_n}.\ P_m\left[\overline{a_k'\ \overline{x_n}}\right]}{\bigwedge \overline{x_n}.\ C\left[\overline{a_k'\ \overline{x_n}}\right]}$$

The proof term for the lifted rule looks similar to the one in the previous case:

$$\lambda \overline{r_m}\ \overline{x_n}.\ R\left[\overline{a_k'\ \overline{x_n}}\right]\ (\overline{r_m\ \overline{x_n}})$$

3.3 Constructing an example proof

We will now demonstrate how a proof term can be synthesized incrementally while proving a theorem in backward style. A proof term corresponding to a proof state will have the general form

$$\lambda(g_1 : \phi_1) \ldots (g_n : \phi_n).\ \ldots\ (g_i\ \overline{x^i}\ \overline{h^i})\ \ldots$$

where the bound variables g_1, $\ldots$, g_n stand for proofs of the current subgoals which are still to be found. The $\overline{x^i}$ and $\overline{h^i}$ appearing in the proof term $(g_i\ \overline{x^i}\ \overline{h^i})$ are parameters and assumptions which may be used in the proof of subgoal i. As an example, the construction of a proof term for the theorem

$$(\exists x.\ \forall y.\ P\ x\ y) \longrightarrow (\forall y.\ \exists x.\ P\ x\ y)$$

will be shown by giving a proof term for each proof state. The parts of the proof terms, which are affected by the application of a rule will be underlined. Initially, the proof state is the trivial theorem:

step 0, remaining subgoal: $(\exists x.\ \forall y.\ P\ x\ y) \longrightarrow (\forall y.\ \exists x.\ P\ x\ y)$

$$\lambda g : ((\exists x.\ \forall y.\ P\ x\ y) \longrightarrow (\forall y.\ \exists x.\ P\ x\ y)).\ g$$

We first apply rule impI. Applying a suitable instance of this rule to the trivial initial proof term yields

$$\begin{array}{l} \lambda g : (\exists x.\ \forall y.\ P\ x\ y) \Longrightarrow (\forall y.\ \exists x.\ P\ x\ y). \\ \quad (\lambda g' : ((\exists x.\ \forall y.\ P\ x\ y) \longrightarrow (\forall y.\ \exists x.\ P\ x\ y)).\ g') \qquad \} \text{ proof term from step 0} \\ \quad\ \underbrace{(\mathsf{impI}\ (\exists x.\ \forall y.\ P\ x\ y)\ (\forall y.\ \exists x.\ P\ x\ y)}_{\text{instance of } \mathsf{impI}}\ g) \end{array}$$

and by $\beta\eta$ reduction of this proof term we obtain

step 1, remaining subgoal: $(\exists x.\ \forall y.\ P\ x\ y) \Longrightarrow (\forall y.\ \exists x.\ P\ x\ y)$

$$\mathsf{impI}\ (\exists x.\ \forall y.\ P\ x\ y)\ (\forall y.\ \exists x.\ P\ x\ y)$$

We now apply allI to the above proof state. Before resolving allI with the proof state, its context has to be augmented with the assumption $\exists x.\ \forall y.\ P\ x\ y$ of the current goal. The resulting proof term is

$$\begin{array}{l} \lambda g : (\bigwedge y.\ \exists x.\ \forall y.\ P\ x\ y \Longrightarrow \exists x.\ P\ x\ y). \\ \quad \mathsf{impI}\ (\exists x.\ \forall y.\ P\ x\ y)\ (\forall y.\ \exists x.\ P\ x\ y) \qquad \} \text{ proof term from step 1} \\ \quad\quad \left.\begin{array}{l} ((\lambda h_2 : (\exists x.\ \forall y.\ P\ x\ y \Longrightarrow \bigwedge y.\ \exists x.\ P\ x\ y). \\ \quad \lambda h_1 : (\exists x.\ \forall y.\ P\ x\ y). \\ \quad\quad \mathsf{allI}\ (\lambda y.\ \exists x.\ P\ x\ y)\ (h_2\ h_1)) \end{array}\right\} \text{lifted instance of } \mathsf{allI} \\ \quad\quad\quad\quad \left.\begin{array}{l} (\lambda h_3 : (\exists x.\ \forall y.\ P\ x\ y) \\ \quad \lambda y :: \beta.\ g\ y\ h_3)) \end{array}\right\} \text{rearranging quantifiers} \end{array}$$

as before, we apply β reduction, which yields *step 2*, remaining subgoal: $\bigwedge y.\ \exists x.\ \forall y.\ P\ x\ y \Longrightarrow \exists x.\ P\ x\ y$

$$
\begin{array}{l}
\lambda g : (\bigwedge y.\ \exists x.\ \forall y.\ P\ x\ y \Longrightarrow \exists x.\ P\ x\ y). \\
\quad \mathsf{impI}\ (\exists x.\ \forall y.\ P\ x\ y)\ (\forall y.\ \exists x.\ P\ x\ y) \\
\quad\quad (\lambda h_1 : (\exists x.\ \forall y.\ P\ x\ y). \\
\quad\quad\quad \mathsf{allI}\ (\lambda y.\ \exists x.\ P\ x\ y)\ (\lambda y :: \beta.\ \underline{g\ y\ h_1}))
\end{array}
$$

By eliminating the existence quantifier using exE we get

step 3, remaining subgoal: $\bigwedge y\ x.\ \forall y.\ P\ x\ y \Longrightarrow \exists x.\ P\ x\ y$

$$
\begin{array}{l}
\lambda g : (\bigwedge y\ x.\ \forall y.\ P\ x\ y \Longrightarrow \exists x.\ P\ x\ y). \\
\quad \mathsf{impI}\ (\exists x.\ \forall y.\ P\ x\ y)\ (\forall y.\ \exists x.\ P\ x\ y) \\
\quad\quad (\lambda h_1 : (\exists x.\ \forall y.\ P\ x\ y). \\
\quad\quad\quad \mathsf{allI}\ (\lambda y.\ \exists x.\ P\ x\ y) \\
\quad\quad\quad\quad (\lambda y :: \beta.\ \mathsf{exE}\ (\lambda x.\ \forall y.\ P\ x\ y)\ (\exists x.\ P\ x\ y)\ h_1\ (\underline{g\ y})))
\end{array}
$$

Applying the introduction rule exI for the existential quantifier results in

step 4, remaining subgoal: $\bigwedge y\ x.\ \forall y.\ P\ x\ y \Longrightarrow P\ (?x\ y\ x)\ y$

$$
\begin{array}{l}
\lambda g : (\bigwedge y\ x.\ \forall y.\ P\ x\ y \Longrightarrow P\ (?x\ y\ x)\ y). \\
\quad \mathsf{impI}\ (\exists x.\ \forall y.\ P\ x\ y)\ (\forall y.\ \exists x.\ P\ x\ y) \\
\quad\quad (\lambda h_1 : (\exists x.\ \forall y.\ P\ x\ y). \\
\quad\quad\quad \mathsf{allI}\ (\lambda y.\ \exists x.\ P\ x\ y) \\
\quad\quad\quad\quad (\lambda y :: \beta.\ \mathsf{exE}\ (\lambda x.\ \forall y.\ P\ x\ y)\ (\exists x.\ P\ x\ y)\ h_1 \\
\quad\quad\quad\quad\quad (\lambda x :: \alpha. \\
\quad\quad\quad\quad\quad\quad \lambda h_2 : (\forall y.\ P\ x\ y). \\
\quad\quad\quad\quad\quad\quad\quad \mathsf{exI}\ (\lambda x.\ P\ x\ y)\ (?x\ y\ x)\ (\underline{g\ y\ x\ h_2}))))
\end{array}
$$

We now eliminate the universal quantifier using allE, which yields

step 5, remaining subgoal: $\bigwedge y\ x.\ P\ x\ (?y\ y\ x) \Longrightarrow P\ (?x\ y\ x)\ y$

$$
\begin{array}{l}
\lambda g : (\bigwedge y\ x.\ P\ x\ (?y\ y\ x) \Longrightarrow P\ (?x\ y\ x)\ y). \\
\quad \mathsf{impI}\ (\exists x.\ \forall y.\ P\ x\ y)\ (\forall y.\ \exists x.\ P\ x\ y) \\
\quad\quad (\lambda h_1 : (\exists x.\ \forall y.\ P\ x\ y). \\
\quad\quad\quad \mathsf{allI}\ (\lambda y.\ \exists x.\ P\ x\ y) \\
\quad\quad\quad\quad (\lambda y :: \beta.\ \mathsf{exE}\ (\lambda x.\ \forall y.\ P\ x\ y)\ (\exists x.\ P\ x\ y)\ h_1 \\
\quad\quad\quad\quad\quad (\lambda x :: \alpha. \\
\quad\quad\quad\quad\quad\quad \lambda h_2 : (\forall y.\ P\ x\ y). \\
\quad\quad\quad\quad\quad\quad\quad \mathsf{exI}\ (\lambda x.\ P\ x\ y)\ (?x\ y\ x) \\
\quad\quad\quad\quad\quad\quad\quad\quad (\mathsf{allE}\ (P\ x)\ (?y\ y\ x)\ (P\ (?x\ y\ x)\ y)\ h_2\ (\underline{g\ y\ x})))))
\end{array}
$$

We can now prove the remaining subgoal by assumption, which is done by substituting the projection function $\lambda(y :: \beta)\ (x :: \alpha).\ \lambda h_3 : (P\ x\ y).\ h_3$ for g:

step 6, no subgoals

$$
\begin{array}{l}
\mathsf{impI}\ (\exists x.\ \forall y.\ P\ x\ y)\ (\forall y.\ \exists x.\ P\ x\ y) \\
\quad (\lambda h_1 : (\exists x.\ \forall y.\ P\ x\ y). \\
\quad\quad \mathsf{allI}\ (\lambda y.\ \exists x.\ P\ x\ y) \\
\quad\quad\quad (\lambda y :: \beta.\ \mathsf{exE}\ (\lambda x.\ \forall y.\ P\ x\ y)\ (\exists x.\ P\ x\ y)\ h_1 \\
\quad\quad\quad\quad (\lambda x :: \alpha. \\
\quad\quad\quad\quad\quad \lambda h_2 : (\forall y.\ P\ x\ y). \\
\quad\quad\quad\quad\quad\quad \mathsf{exI}\ (\lambda x.\ P\ x\ y)\ x \\
\quad\quad\quad\quad\quad\quad\quad (\mathsf{allE}\ (P\ x)\ y\ (P\ x\ y)\ h_2\ (\lambda h_3 : (P\ x\ y).\ h_3)))))
\end{array}
$$

4 Partial proof terms

Proof terms are large, contain much redundant information, and need to be compressed. The solution is simple: leave out everything that can be reconstructed. But since we do not want to complicate reconstruction too much, it should not degenerate into proof search. Thus we have to keep the skeleton of the proof. What can often be left out are the ϕ, τ and t in $\lambda h{:}\phi.\ p$, $\lambda x{::}\tau.\ p$ and $(p\ t)$.

Since we will have to reconstruct the missing information later on, it is conceptually simpler to model the missing information by *unification variables*. These are simply a new class of free (term and type) variables, syntactically distinguished by a leading "?", as in $?f$ and $?\alpha$. We will sometimes write $?f_\tau$ to emphasize that $?f$ has type τ. *Substitutions* are functions that act on unification variables, e.g. $\theta = \{?f \mapsto \lambda x.x, ?\alpha \mapsto \tau\}$.

In the remainder of this section we work with *partial* proofs, where terms and types may contain unification variables, as in $(p\ ?f)$. Note that term unification variables that occur within the scope of a λ need to be "lifted" as in $\lambda x{::}\tau.\ (p\,(?f\,x))$. Because of this lifting, this partial information may take up more space than it saves. Therefore an actual implementation is bound to introduce separate new constructors for proof trees, e.g. $\lambda h{:}\_.\ p$, $\lambda x{::}\_.\ p$ and $(p\ \_)$, where _ represents the missing information. This is in fact what Necula and Lee describe [6]. However, it turns out that our partial proofs are easier to treat mathematically, not far from the "_"-version, and also allow to drop only part of a term (although we will not make use of this feature).

Of course, we cannot check partial proofs with the rules of §3.1. In fact, we may not be able to check them at all, because too much information is missing. But we can collect equality constraints that need to hold in order for the proof to be correct. Such equality constraints are of the form $T_1 =^? T_2$, where T_1 and T_2 are either both terms or both types. A substitution *solves* a constraint if the two terms become equal modulo $\beta\eta$-conversion, or if the two types become identical. Sets of such equality constraints are usually denoted by the letters C and D. To separate C into term and type constraints, let C_t denote the term and C_τ the type part. The subscripts t and τ do *not* refer to variable names but are simply keywords.

We will now show how to extract a set of constraints from a partial proof; how to solve those constraints is discussed later on.

4.1 Collecting constraints

The relation $\Gamma \vdash p \rhd (\phi,\ C)$ is a partial function taking Γ and a partial proof p and producing a formula ϕ (which may contain unification variables) and a set of constraints C. The function will be defined such that, if θ solves C, then $\theta(p)$ proves $\theta(\phi)$. The notation $\overline{V_\Gamma}$ denotes the list of all term variables declared in

Γ, and $\overline{\tau_\Gamma}$ denotes the list of their types, i.e. $\overline{V_\Gamma} = x_1 \ldots x_n$ and $\overline{\tau_\Gamma} = \tau_1 \ldots \tau_n$ for $\Gamma = x_1 :: \tau_1 \ldots x_n :: \tau_n$.

$$\frac{}{\Gamma', h : \phi, \Gamma \vdash h \rhd (\phi,\ \emptyset)} \qquad \frac{\Sigma(c) = \phi}{\Gamma \vdash c_{\overline{[\tau_n/\alpha_n]}} \rhd (\phi\overline{[\tau_n/\alpha_n]},\ \emptyset)}$$

$$\frac{\Gamma, h : \phi \vdash p \rhd (\psi,\ C) \quad \Gamma \vdash \phi \rhd (\tau,\ D)}{\Gamma \vdash (\lambda h{:}\phi.\ p) \rhd (\phi \Longrightarrow \psi,\ C \cup D \cup \{\tau =^? \mathsf{prop}\})}$$

$$\frac{\Gamma, x :: \tau \vdash p \rhd (\phi,\ C)}{\Gamma \vdash (\lambda x{::}\tau.\ p) \rhd (\bigwedge x :: \tau.\ \phi,\ \{\lambda x :: \tau.\ r =^? \lambda x :: \tau.\ s \mid (r =^? s) \in C_t\} \cup C_\tau)}$$

$$\frac{\Gamma \vdash p \rhd (\phi,\ C) \quad \Gamma \vdash q \rhd (\psi,\ D)}{\Gamma \vdash (p\ q) \rhd (?f_{\overline{\tau_\Gamma} \to \mathsf{prop}}\ \overline{V_\Gamma},\ \{\phi =^? (\psi \Longrightarrow ?f_{\overline{\tau_\Gamma} \to \mathsf{prop}}\ \overline{V_\Gamma})\} \cup C \cup D)}$$

$$\frac{\Gamma \vdash p \rhd (\phi,\ C) \quad \Gamma \vdash t \rhd (\tau,\ D)}{\Gamma \vdash (p\ t) \rhd (?f_{\overline{\tau_\Gamma} \to \tau \to \mathsf{prop}}\ \overline{V_\Gamma}\ t,\ \{\phi =^? \bigwedge x :: \tau.\ ?f_{\overline{\tau_\Gamma} \to \tau \to \mathsf{prop}}\ \overline{V_\Gamma}\ x\} \cup C \cup D)}$$

As usual, the unification variables $?f$ must be "new" in each case.

The above rules follow those in §3.1 very closely. For example, the intuition behind the rule for the application $(p\ q)$ is the following: if p proves proposition ϕ, then ϕ must be some implication and the proposition ψ proved by q must be the premise of this implication. Moreover, the proposition proved by $(p\ q)$ is the conclusion of the implication. The set of constraints for $(p\ q)$ is the union of the constraints for p and q, plus one additional constraint expressing that ϕ is a suitable implication. One point to note is the judgement $\Gamma \vdash t \rhd (\tau,\ D)$ used in two premises of the constraint collection rules. It corresponds to $\Gamma \vdash t :: \tau$ just like $\Gamma \vdash p \rhd (\phi,\ C)$ corresponds to $\Gamma \vdash p :: \phi$, i.e. D is a set of type constraints whose solvability implies that t has type τ. The rules for $\Gamma \vdash t \rhd (\tau,\ D)$ are not given because they are well-known: both from the literature about type inference for simply typed terms and because they closely resemble the rules above, just one level down. In a setting where types, terms and proofs are not syntactically distinguished, we would only have one judgement $. \vdash . \rhd (.,\ .)$.

We introduce the notation $(\phi, C) = \mathit{collect}(\Gamma, p)$ as a functional variant of $\Gamma \vdash p \rhd (\phi,\ C)$.

Example 1. Let $p = \lambda x :: ?\alpha_1.\ \lambda h_1 : (?f_{?\alpha_2}\ x).\ \lambda y :: ?\alpha_3.\ \lambda h_2 : (?f'_{?\alpha_4}\ x\ y).\ h_1\ h_2\ y$ Then

$$\begin{aligned}
\mathit{collect}([], p) = (&\textstyle\bigwedge x :: ?\alpha_1.\ ?f_{?\alpha_2}\ x \Longrightarrow \bigwedge y :: ?\alpha_3.\ ?f'_{?\alpha_4}\ x\ y \Longrightarrow \\
&\quad ?g'_{?\alpha_1 \to ?\alpha_3 \to ?\alpha_3 \to \mathsf{prop}}\ x\ y\ y, \\
&\{\lambda x :: ?\alpha_1.\lambda y :: ?\alpha_3.\ ?f_{?\alpha_2}\ x\ =^? \\
&\quad \lambda x :: ?\alpha_1.\lambda y :: ?\alpha_3.\ ?f'_{?\alpha_4}\ x\ y \Longrightarrow ?g_{?\alpha_1 \to ?\alpha_3 \to \mathsf{prop}}\ x\ y, \\
&\quad \lambda x :: ?\alpha_1.\lambda y :: ?\alpha_3).\ ?g_{?\alpha_1 \to ?\alpha_3 \to \mathsf{prop}}\ x\ y\ =^? \\
&\quad \lambda x :: ?\alpha_1.\lambda y :: ?\alpha_3).\ \textstyle\bigwedge z :: ?\alpha_3.\ ?g'_{?\alpha_1 \to ?\alpha_3 \to ?\alpha_3 \to \mathsf{prop}}\ x\ y\ z, \\
&\quad ?\alpha_2 =^? ?\alpha_1 \to ?\beta_1,\ ?\beta_1 =^? \mathsf{prop} \\
&\quad ?\alpha_4 =^? ?\alpha_1 \to ?\alpha_3 \to ?\beta_2,\ ?\beta_2 =^? \mathsf{prop}\})
\end{aligned}$$

where $?g$, $?g'$ and $?\beta_i$ are new variables generated during constraint collection.

Theorem 1. *(Soundness and completeness)*

1. *If* $\Gamma \vdash p \rhd (\phi',\ C)$ *and* θ *solves* C *then* $\theta(\Gamma) \vdash \theta(p) : \theta(\phi')$.
2. *If* $\Gamma \vdash p : \phi$ *and* $\Gamma \vdash p \rhd (\phi',\ C)$ *then* $C \cup \{\phi =^? \phi'\}$ *is solvable.*

This theorem shows the advantage of working with partial proofs as opposed to proofs containing "_": we can produce the full proof by instantiation from the partial one.

Thus there are two possible system architectures for checking partial proofs:

- either constraint collection $\Gamma \vdash p \rhd (\phi',\ C)$ and constraint solving are part of the trusted kernel and $\theta(\phi')$ is accepted as the correct answer;
- or, for the security conscious, neither collecting nor solving is part of the trusted kernel and their result is checked by checking $\theta(\Gamma) \vdash \theta(p) : \theta(\phi')$.

4.2 Solving constraints

Our constraints are a mixture of term and type constraints. Type constraints can be solved by first-order unification and thus do not need to be discussed here: they can be solved at any time in any order. Therefore we concentrate on term constraints in this subsection.

Since higher-order unification is undecidable, we restrict to unification of so called (higher-order) *patterns* [4], i.e. λ-terms where each occurrence of a unification variable is applied only to distinct bound variables. For example $\lambda xy.?F\ y\ x$ is a pattern, whereas $\lambda x.?F\ x\ x$ and $?Fa$ are not patterns. The set of all patterns is denoted by $\mathcal{P}at$. For us, the key property of patterns is that their unification is decidable and that solvable pattern unification problems have most general unifiers [4, 12, 7, 13]. Thus we may assume a function mgu taking two patterns as arguments and either failing or returning the most general unifier of its arguments.

Since the constraints C generated by $\Gamma \vdash p \rhd (\phi,\ C)$ may contain non-patterns, we have to delay solving those constraints until (hopefully) they are turned into patterns by the solution of other pattern constraints. Of course, in the worst case C may not contain any patterns at all, in which case we have to give up. Thus we have to take care when constructing partial proofs to make sure the complete proof can be reconstructed by pattern unification.

Example 2. The first constraint in $\{?f\,?z =^? ?z,\ \lambda xy.?f\,x =^? \lambda xy.?f\,y\}$ is not a pattern constraint, but if the second constraint is solved first, it yields the substitution $?f \mapsto \lambda x.?u$, which turns the first constraint into the trivial $?u =^? ?z$.

We say that a set of equality constraints C *can be solved by pattern unification* if $C \longrightarrow^* \emptyset$, where $\longrightarrow$ is defined by the rewrite rule

$$C \cup \{s =^? t\} \longrightarrow \theta(C)$$

where s and t are patterns and $mgu(s,t)$ returns the unifier θ. Note that the choice of which pair $s =^? t$ to solve at which point is immaterial because $\longrightarrow$ is confluent. This is well-known for first-order terms, and holds for patterns as well because of the existence of most general unifiers and the following easy results:

Lemma 1. *1. If $s, t \in \mathcal{P}at$ and $mgu(s,t)$ returns θ then $\mathcal{R}an(\theta) \subseteq \mathcal{P}at$.*
2. If $\mathcal{R}an(\theta) \subseteq \mathcal{P}at$ and $u \in \mathcal{P}at$ then $\theta(u) \in \mathcal{P}at$.

4.3 Compressing proof terms

The basic idea here is straightforward: given a proof term, remove all of the information that can be reconstructed by unification. If our meta logic were Prolog (i.e. proof terms contain no λs), we could drop all terms t in an application $(p\ t)$, because first-order unification can reconstruct them. In our setting we require that pattern unification should be able to reconstruct the missing terms.

Compression is performed in three phases. First, all terms and types in the proof are replaced by suitably lifted unification variables. A substitution reversing the term abstractions is constructed as well.

$$\begin{array}{rcl}
\mathit{varify}(\overline{x}, \overline{\tau}, \lambda v{:}\phi.\ p) & = & \texttt{let}\ (p', \theta) = \mathit{varify}(\overline{x}, \overline{\tau}, p) \\
& & \texttt{in}\ (\lambda v{:}(?f_{\overline{\tau} \to \mathsf{prop}}\ \overline{x}).\ p',\ \ \theta \cup \{?f \mapsto \lambda \overline{x}.t\}) \\
\mathit{varify}(\overline{x}, \overline{\tau}, \lambda y{::}\tau.\ p) & = & \texttt{let}\ (p', \theta) = \mathit{varify}(\overline{x}y, \overline{\tau}\,?\alpha, p) \\
& & \texttt{in}\ (\lambda y{::}?\alpha.\ p',\ \ \theta) \\
\mathit{varify}(\overline{x}, \overline{\tau}, (p\ q)) & = & \texttt{let}\ (p', \theta) = \mathit{varify}(\overline{x}, \overline{\tau}, p);\ \ (q', \theta') = \mathit{varify}(\overline{x}, \overline{\tau}, q) \\
& & \texttt{in}\ ((p'\ q'), \theta') \\
\mathit{varify}(\overline{x}, \overline{\tau}, (p\ t)) & = & \texttt{let}\ (p', \theta) = \mathit{varify}(\overline{x}, \overline{\tau}, p) \\
& & \texttt{in}\ ((p'\ (?f_{\overline{\tau} \to ?\beta}\ \overline{x})),\ \ \theta \cup \{?f \mapsto \lambda \overline{x}.t\}) \\
\mathit{varify}(\overline{x}, \overline{\tau}, c_{\overline{[\tau_n/\alpha_n]}}) & = & (c_{\overline{[?\alpha_n/\alpha_n]}}, \emptyset)
\end{array}$$

Thus $\mathit{varify}(\overline{x}, \overline{\tau}, p) = (p', \theta)$ does not quite imply $\theta(p') = p$ because θ does not reverse the type abstractions: types are first-order and thus they can be reconstructed uniquely by unification.

Then the constraints are extracted from the resulting partial proof: $(\phi', C) = \mathit{collect}(\Gamma, p')$. Finally we compute (with the help of function *solves*) a minimal set of term variables $V \subseteq \mathcal{D}om(\theta)$ such that $\theta(C)$ can be solved by pattern unification. Thus the overall algorithm for compressing a proof p is

$$\begin{array}{rll}
\mathit{compress}(p, \phi) = & \texttt{let} & (p', \theta) = \mathit{varify}([], [], p) \\
& & (\phi', C) = \mathit{collect}([], p') \\
& & V = \mathit{solves}(C \cup \{\phi =^? \phi'\}, \theta) \\
& \texttt{in} & \theta|_V(p')
\end{array}$$

where $\theta|_V$ is the restriction of θ to V. Function $\mathit{solves}(D, \theta)$ returns $V \subseteq \mathcal{D}om(\theta)$ such that $\theta|_V(D)$ is solvable by pattern unification. The details are explained below.

The main correctness theorem expresses that compression does not lose any information in the following sense: the constraints collected from the compressed version of a valid proof are solvable by pattern unification and any solution yields a proof of the original formula.

Theorem 2. *Let ϕ be ground. If $\vdash p : \phi$, $q = \mathit{compress}(p, \phi)$ and $(\psi, C) = \mathit{collect}([], q)$ then*

1. $C \cup \{\phi =^? \psi\}$ *is solvable by pattern unification and*
2. *if* θ *solves* $C \cup \{\phi =^? \psi\}$ *then* $\vdash \theta(q) : \phi$.

The second part of the theorem follows directly from the soundness of *collect* (part 1 of Theorem 1) because ϕ is ground, i.e. $\theta(\phi) = \phi$.

The fact that $C \cup \{\phi =^? \psi\}$ is solvable by pattern unification is a bit more subtle. From $q = \mathit{compress}(p, \phi)$ it follows by definition of *compress* that there are p', θ, ϕ', C' and V such that $(p', \theta) = \mathit{varify}([], [], p)$, $(\phi', C') = \mathit{collect}([], p')$, $V = \mathit{solves}(C' \cup \{\phi =^? \phi'\}, \theta)$ and $q = \theta|_V(p')$. Because $(\psi, C) = \mathit{collect}([], q) = \mathit{collect}([], \theta|_V(p'))$ and $(\phi', C') = \mathit{collect}([], p')$ it can be shown that $\psi = \theta|_V(\phi')$ and $C = \theta|_V(C')$. Thus $C \cup \{\phi =^? \psi\} = \theta|_V(C' \cup \{\phi =^? \phi'\})$. It can be shown that $\theta(C' \cup \{\phi =^? \phi'\})$ is solvable by pattern unification. Appealing to Theorem 3 below, it follows that so is $C \cup \{\phi =^? \psi\}$.

Theorem 3. *If* $\theta(C)$ *is solvable by pattern unification, then* $\mathit{solves}(C, \theta)$ *terminates successfully with a set of variables* V *such that* $\theta|_V(C)$ *is solvable by pattern unification.*

This can be viewed as a specification of *solves* or as its main correctness theorem. Of course there are trivial implementations of *solves* that simply return $\mathcal{D}om(\theta)$. What we want is a minimal set V with the stated property. Note that in general there is no least such V:

Example 3. Let $C = \{\lambda x.?f(?g(x)) =^? \lambda x.f(x)\}$ and $\theta = \{?f \mapsto f, ?g \mapsto \lambda x.x\}$. Then $\theta(C)$, $\theta|_{\{?f\}}(C)$ and $\theta|_{\{?g\}}(C)$ are solvable by pattern unification.

Therefore *solves* nondeterministically computes a minimal set of variables by simulating the process of solving the constraints by pattern unification. Every time the process gets stuck, i.e. no more pattern constraints are left, a minimal set of additional variables is instantiated.

The main complication is that pattern unification may introduce new variables. Thus we need to keep track of where they came from, i.e. which original variable needs to be instantiated in order to ensure that the new variable becomes instantiated. Again, there may be a choice:

Example 4. Let $C = \{\lambda xyz.?f\,z\,x =^? \lambda xyz.?g\,x\,y\} \cup C'$ and let $\theta = \{?f \mapsto \lambda xy.f\,y, ?g \mapsto \lambda xy.f\,x\} \cup \theta'$. Solving the first constraint in C yields the substitution $\sigma = \{?f \mapsto \lambda xy.?h\,y, ?g \mapsto \lambda xy.?h\,x\}$. Now we need to compute the value of $?h$ (in case it is required later on in order to continue) and we need to record that $?h$ depends on either $?f$ or $?g$, i.e. instantiating $?f$ or $?g$ will instantiate $?h$. We can chose to record either dependency.

In the algorithm below, this dependency relation is stored in a partial function D mapping new variables to old ones they depend on.

Finally we introduce some terminology to select those variables that occur in non-pattern positions: given an equality constraint st, $\mathcal{NPV}ars(st)$ is the set of variables $?f$ that occur in subterms of the form $?f\ \overline{u}$ in st, such that $\overline{u}$ is not a list of distinct bound variables.

Now we can describe $solves(C, \theta)$ as an imperative algorithm. As long as there are pattern problems left in C, we solve them and propagate the solution. Once we are left only with non-pattern problems, we pick one such that values for all its non-pattern variables are known (via θ). All those variables are then instantiated and recorded in V. If there is no such non-pattern problem either, the algorithm fails. Luckily, Theorem 3 tells us that this will not happen in the cases we are interested in.

$V := \emptyset; D := \{?f \mapsto ?f \mid ?f \in \mathcal{V}ar(C)\}$
`while` $C \neq \emptyset$ `do`
  `if` there is a pattern problem $(s =^? t) \in C$
  `then` $\sigma := mgu(s, t);\ C := \sigma(C - \{s =^? t\});$
    `forall` $(?f \mapsto t) \in \sigma$ `do`
      $\delta := mgu(\theta(?f), \theta(t));$
      $D := D \cup \{?h \mapsto D(?f) \mid ?h \in \mathcal{D}om(\delta)\}$
    `od`
  `else` pick some $st \in C$ such that $\mathcal{NPV}ars(st) \subseteq \mathcal{D}om(\theta)$;
    $V := V \cup \{D(?f) \mid ?f \in \mathcal{NPV}ars(st)\}$;
    $C := \theta|_{\mathcal{NPV}ars(st)}(C)$;
`od`;
`return` V

Note that the else-case does not necessarily compute a minimal V: if $st = (\lambda xy.?f\, x\, (?g\, y\, y) =^? \lambda xy.x\}$ and $\theta(?f) = \lambda xy.x$ and $\theta(?g) = \lambda xy.x$ then it suffices to instantiate either $?f$ or $?g$, whereas above both are added to V. The above algorithm can be refined by instantiating st stepwise from the top and normalizing the result each time.

5 Implementation

The algorithms for compression and reconstruction of proofs have been implemented in ML as a part of the theorem prover Isabelle. During the proof of a theorem, the corresponding proof term is built up incrementally. Since this may slow down the execution of proof scripts, the generation of proof terms can be switched on and off as needed: during the interactive development of a proof this feature may be switched off. When the proof is completed, the proof script may be re-run with proof generation switched on and the resulting proof term could be exported.

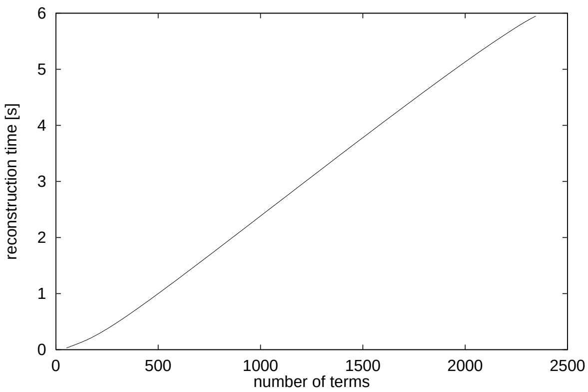
We have tested the implementation on several proofs of theorems in Pelletier's collection [10], which were generated by Isabelle's tableau prover. The following table summarizes some results[1]. It shows the number of terms (i.e. terms such as ϕ and t in $\lambda h{:}\phi.\ p$ and $(p\ t)$) occurring in the uncompressed proof term, as well as the number of terms occurring in the compressed proof term (i.e. terms not replaced by placeholders "_", as explained in §4). In all cases, the compression

[1] These measurements were done on a Pentium II with 400 MHz and 512 MB RAM

ratio was more than 90%. The compression rate reported by Necula and Lee [6] appears to be a little better, but that is probably because we counted only the number of terms, not their size: our dynamic compression scheme should drop at least as much as Necula and Lee's static scheme.

number of terms			reconstruction
uncompressed	compressed	compression	time [s]
52	4	92.3%	0.030
116	6	94.8%	0.120
170	9	94.7%	0.190
316	16	94.9%	0.360
425	31	92.7%	0.710
1948	142	92.7%	5.220
2345	153	93.5%	5.950

The following diagram shows the correspondence between the number of terms in the uncompressed proof term and the time needed for reconstruction.



A crucial point is the efficient handling of large sets of constraints: when having computed a unifier for a constraint, one possibility is to apply this unifier to all the remaining constraints. Another possibility is to accumulate the unifiers and only apply them when needed. The first solution can be rather slow when having a large number of constraints, while the second solution—which has been chosen in our implementation—requires efficient data structures for storing substitutions. To speed up reconstruction, our implementation of function *collect* described in §4.1 tries to solve newly introduced constraints immediately, instead of collecting all constraints first and then solving them at the end.

6 Conclusion

We have given a first presentation of proof terms for simply typed intuitionistic higher-order logic, an important logical framework. We hope that by unfolding the underlying type theory and explicitly isolating the simply typed components familiar to users of Isabelle or HOL, this may popularize the λ-calculus view of proofs in those quarters as well. We have also presented what appears to be a novel compression scheme for proofs. Hence our work provides a new and promising basis for exchanging proofs in simply typed logics, in particular HOL, both among theorem provers (especially automatic and interactive) and in the realm of proof carrying code.

References

[1] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 118–309. Oxford University Press, 1992.

[2] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, H. Laulhère, C. Muñoz, C. Murthy, C. Parent-Vigouroux, P. Loiseleur, C. Paulin-Mohring, A. Saïbi, and B. Werner. The Coq proof assistant reference manual – version 6.3.1. Technical report, INRIA, 1999.

[3] M. J. C. Gordon, J. M. J. Herbert, R. W. S. Hale, J. Harrison, W. Wong, and J. von Wright. Self-checking prover study – final report. Technical report, SRI, 1995. Available at `http://www.csl.sri.com/reports/postscript/proofchecker.ps.gz`.

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

[5] G. C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, New York, 1997.

[6] G. C. Necula and P. Lee. Efficient representation and validation of proofs. In *13th IEEE Symp. Logic in Computer Science (LICS'98)*, pages 93–104. IEEE Computer Society Press, 1998.

[7] T. Nipkow. Functional unification of higher-order patterns. In *8th IEEE Symp. Logic in Computer Science*, pages 64–74. IEEE Computer Society Press, 1993.

[8] L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

[9] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.

[10] F. J. Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2:191–216, 1986.

[11] F. Pfenning. Logic programming in the LF Logical Framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 66–78. Cambridge University Press, 1991.

[12] F. Pfenning. Unification and anti-unification in the calculus of constructions. In *6th IEEE Symposium on Logic in Computer Science*, pages 74–85. IEEE Computer Society Press, 1991.

[13] Z. Qian. Unification of higher-order patterns in linear time and space. *Journal of Logic and Computation*, 6:315–341, 1996.

[14] W. Wong. Recording and checking HOL proofs. In E. Schubert, P. Windley, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications. 8th International Workshop*, volume 971 of *LNCS*, pages 353–68. Springer-Verlag, Berlin, 1995.