Automated Theorem Proving in a Simple Meta-Logic for LF

Carsten Schürmann and Frank Pfenning *

Carnegie Mellon University
School of Computer Science
carsten@cs.cmu.edu fp@cs.cmu.edu

Abstract. Higher-order representation techniques allow elegant encodings of logics and programming languages in the logical framework LF, but unfortunately they are fundamentally incompatible with induction principles needed to reason about them. In this paper we develop a meta-logic $\mathcal{M}_2$ which allows inductive reasoning over LF encodings, and describe its implementation in Twelf, a special-purpose automated theorem prover for properties of logics and programming languages. We have used Twelf to automatically prove a number of non-trivial theorems, including type preservation for Mini-ML and the deduction theorem for intuitionistic propositional logic.

1 Introduction

The logical framework LF [HHP93] has been designed as a meta-language for representing deductive systems which are common in the study of logics and programming languages. It allows concise encodings of many common inference systems, such as natural deduction and sequent calculi, type systems, operational semantics, compilers, abstract machines, etc. (see [Pfe96] for a survey). These representations often lead directly to implementations, either via the constraint logic programming paradigm [Pfe94] or via general search using tactics and tacticals.

The logical framework derives its expressive power from the use of dependent types together with "higher-order" representation techniques which directly support common concepts in deductive systems, such as variable binding and capture-avoiding substitution, parametric and hypothetical judgments, and substitution properties. The fact that these notions are an integral part of the logical framework would seem to make it an ideal candidate not only for reasoning *within* various inference systems, but for reasoning *about* properties of such systems.

Unfortunately, higher-order representation techniques are fundamentally incompatible with the induction principles needed to reason about such encodings (see [DPS97] for a detailed analysis). In the literature three approaches have been studied in order to overcome these problems, while retaining the advantages a

* This work was supported by NSF Grant CCR-9619584

logical framework can offer. The first called *schema-checking* [Roh94,RP96] implements meta-theoretic proofs as relations whose operational reading as logic programs realizes the informal proofs. This has been applied successfully in many case studies (see [Pfe96]), but lacks automation. The second is based on reflection via a modal provability operator. At present it is unclear how this idea, developed for simple types in [DPS97], interacts with dependent types, and if it is flexible enough for many of the theorems that can be treated with schema-checking. The third is to devise an explicit (meta-)meta-logic for reasoning about logical framework encodings. For the simpler logical framework of hereditary Harrop formulas this approach has been followed by McDowell and Miller [MM97,McD97] (see Section 5 for a detailed comparison).

In this paper we follow the third approach and develop a simple meta-logic $\mathcal{M}_2$ for LF and sketch its implementation in the Twelf system. $\mathcal{M}_2$ was designed explicitly to support automated inductive theorem proving and has been applied successfully to prove, for example, value soundness and type preservation for Mini-ML, completeness of a continuation stack machine with respect to a natural semantics for Mini-ML, soundness and completeness of uniform derivations with respect to resolution (which is a critical step in the correctness of compilers for logic programming languages), the deduction theorem for intuitionistic propositional logic using Hilbert's axiomatization, and the existence of an embedding of Cartesian closed categories into the simply-typed λ-calculus. In each case we specified only the theorem and the induction variable, the proof was completely automatic in every other respect.

We view Twelf as a special-purpose automated theorem prover for the theory of programming languages and logics. It owes its success to the expressive power of the logical framework combined with the simplicity of the meta-logic which nonetheless allows direct expression of informal mathematical arguments. Its main current limitations are the lack of facilities for incorporating lemmas and for proving properties which require reasoning about *open* LF objects, i.e., objects which may contain free variables. We plan to address the former by adapting standard techniques from inductive and resolution theorem proving and the latter by borrowing successful ideas from schema-checking.

This paper is organized as follows: In Section 2 we briefly describe the logical framework LF and introduce a programming language Mini-ML and a type preservation result as running example. The meta-logic $\mathcal{M}_2$ is introduced in Section 3 which is implemented in the Twelf system which we discuss in Section 4. Section 5 compares the most closely related work before we assess the results and discuss future work.

2 The Logical Framework LF

The type theory underlying the logical framework LF is an extension of the simply-typed λ-calculus by dependent types. It is defined by three syntactic categories of objects, type families, and kinds [HHP93]. We use a for type family constants, c for object constants, and x for variables. Atomic types have the form

$a\ M_1 \ldots M_n$ and function types $\Pi x : A_1 . A_2$, which we may write as $A_1 \to A_2$ if x does not occur free in A_2. We assume that constants and variables are declared at most once in a signature and context, respectively. As usual we apply tacit renaming of bound variables to maintain this assumption and to guarantee capture-avoiding substitution.

The LF type theory is defined by a number of mutually dependent judgments which we only summarize here. The main typing judgment is $\Gamma \vdash_\Sigma M : A$ and expresses that object M has type A in context Γ with respect to signature Σ. We generally assume that signature Σ is valid and fixed and therefore omit it from the typing and other related judgments introduced below. We also need to explicitly require the validity of contexts, written as $\vdash \Gamma$ ctx. In a slight departure from [HHP93] we take $\beta\eta$-conversion as our notion of definitional equality, since this guarantees that every well-typed object has an equivalent *canonical form*, that is, a long $\beta\eta$-normal form. The requisite theory may be found in [Coq91].

As a running example we will use Mini-ML in the formulation of [Pfe92] which goes back to [MP91], culminating in an automatic proof of type preservation. While space only permits showing the fragment including abstraction, application, and recursion, our automatic proof also treats the remaining features of Mini-ML including polymorphism and an inductively defined type.

Mini-ML is defined through expressions e, types τ, a typing judgment $\Delta \triangleright e : \tau$, and an evaluation judgment $e \hookrightarrow v$, which are represented as type families

$\mathrm{exp} : \mathrm{type}$,
$\mathrm{tp} : \mathrm{type}$,
$\mathrm{of} : \mathrm{exp} \to \mathrm{tp} \to \mathrm{type}$, and
$\mathrm{ev} : \mathrm{exp} \to \mathrm{exp} \to \mathrm{type}$,

respectively. Expressions, types, typing rules, and evaluation rules are encoded as object-level constants. The encoding is adequate in the sense that there is a *compositional bijection* between derivations and well-typed objects of appropriate type. For example, using $\ulcorner . \urcorner$ for a generic representation function, we have that derivations of $e \hookrightarrow v$ are in bijective correspondence with closed canoncial objects of type $\mathrm{ev} \ulcorner e \urcorner \ulcorner v \urcorner$.

Compositionality of the encoding gives us the following substitution lemma "for free", since it can be represented simply by substitution in LF, whose correctness has been proven once and for all [HHP93].

Lemma 1 (Substitution). *If* $\Delta \triangleright e' : \tau'$ *and* $\Delta, x : \tau' \triangleright e : \tau$ *then* $\Delta \triangleright e[e'/x] : \tau$.

A substitution lemma of this or a similar form is an important ingredient in many theorems in logic (e.g., cut-elimination, normalization, or the Church-Rosser theorem) or the theory of programming languages (e.g., subject reduction or type preservation).

To demonstrate our theorem prover, we consider the type preservation theorem for Mini-ML. It is proven by structural induction, with repeated applications of *inversion*, which is applicable when the shape of the conclusion determines the inference rule which must have been applied last [Pfe92]. (This proof is also *exactly* the proof found automatically rendered into informal notation.) We write $\mathcal{D} :: J$ if $\mathcal{D}$ is a derivation of a judgment J to avoid two-dimensional notation.

Theorem 1 (Type preservation). *For all expressions e, v, types τ, and derivations $\mathcal{D} :: (e \hookrightarrow v)$ and $\mathcal{P} :: (\cdot \triangleright e : \tau)$, there exists a derivation $\mathcal{Q} :: (\cdot \triangleright v : \tau)$.*

The inductive proof of this theorem is constructive and contains a method for constructing a derivation $\mathcal{Q} :: (\cdot \triangleright v : \tau)$ from $\mathcal{D} :: (e \hookrightarrow v)$ and $\mathcal{P} :: (\cdot \triangleright e : \tau)$. By an extension of the Curry-Howard correspondence one might hope to represent this as an LF function

$$\text{tps} : \Pi E : \text{exp}.\ \Pi V : \text{exp}.\ \Pi T : \text{tp}.\ \text{ev}\ E\ V \to \text{of}\ E\ T \to \text{of}\ V\ T.$$

In fact, if we could exhibit a total function of this type, we would know that type preservation holds. Unfortunately, such a function does not exist in LF, since it would have to be defined by primitive recursion over its fourth argument (the derivation of ev $E\ V$), and primitive recursion is not available in LF. Moreover, straightforward attempts to add primitive recursion render higher-order representations inadequate, as discussed in [DPS97]. Instead we define a meta-logic for LF in which it is possible to express and prove (over the signature encoding Mini-ML):

> For all closed LF objects E : exp, V : exp, T : tp, D : ev $E\ V$, and P : of $E\ T$ there exists a closed LF object Q : of $V\ T$.

By the adequacy of the encodings, the existence of such an LF object Q implies the existence of a typing derivation $\mathcal{Q}$ of $\cdot \triangleright v : \tau$, where $\ulcorner v \urcorner = V$ and $\ulcorner \tau \urcorner = T$, thereby guaranteeing the type preservation property for Mini-ML.

3 The Meta-Logic $\mathcal{M}_2$

The purpose of the meta-logic $\mathcal{M}_2$ is formal reasoning about properties of LF signatures, with the goal of automating the proof of such properties. Since LF signatures implement object languages and their semantics, this provides for automatic proofs of properties of logics and programming languages.

$\mathcal{M}_2$ is a restricted constructive first-order logic where quantifiers range over closed LF objects constructed over a given signature Σ. Its formal definition is a sequent calculus endowed with realizing proof terms.

The formal system of $\mathcal{M}_2$ in its full generality is rather complex. We therefore present here only a restriction of $\mathcal{M}_2$, where pattern matching subjects must be of atomic type. For a complete presentation of the meta-logic we refer the interested reader to the technical report [SP98]. We introduce $\mathcal{M}_2$ in four steps: in Section 3.1 we describe a constructive logic over LF with proof terms which we augment by well-founded recursion in Section 3.2. In Section 3.3 we introduce definition by cases and in Section 3.4 we state the meta-theoretic properties of $\mathcal{M}_2$ which make it an appropriate meta-logic.

$$\frac{\Gamma \vdash \sigma : \Gamma_1 \quad \Gamma; (\Delta_1, \mathbf{x} \in \forall \Gamma_1.\, F_1, \Delta_2, \mathbf{y} \in F_1[\sigma]) \vdash P \in F_2}{\Gamma; (\Delta_1, \mathbf{x} \in \forall \Gamma_1.\, F_1, \Delta_2) \vdash \text{let } \mathbf{y} = \mathbf{x}\, \sigma \text{ in } P \in F_2}\,\forall\mathsf{L} \qquad \frac{(\Gamma, \Gamma_1); \Delta \vdash P \in F}{\Gamma; \Delta \vdash \Lambda \Gamma_1.\, P \in \forall \Gamma_1.\, F}\,\forall\mathsf{R}^*$$

$$\frac{(\Gamma, \Gamma_1); (\Delta_1, \mathbf{x} \in \exists \Gamma_1.\, \top, \Delta_2) \vdash P \in F}{\Gamma; (\Delta_1, \mathbf{x} \in \exists \Gamma_1.\, \top, \Delta_2) \vdash \text{split } \mathbf{x} \text{ as } \langle \Gamma_1 \rangle \text{ in } P \in F}\,\exists\mathsf{L}^* \qquad \frac{\Gamma \vdash \sigma : \Gamma_1}{\Gamma; \Delta \vdash \langle \sigma \rangle \in \exists \Gamma_1.\, \top}\,\exists\mathsf{R}$$

* Eigenvariable condition: $\vdash \Gamma, \Gamma_1$ ctx

Fig. 1. $\mathcal{M}_2$ without recursion or pattern matching

3.1 A Constructive Sequent Calculus Over LF

Formulas in $\mathcal{M}_2$ have the form $\forall x_1 : A_1 \ldots \forall x_n : A_n.\, \exists y_1 : B_1 \ldots \exists y_m : B_m.\, \top$ (which we write as $\forall \Gamma_1.\, \exists \Gamma_2.\, \top$, where $\Gamma_1 = x_1 : A_1, \ldots, x_n : A_n$ and $\Gamma_2 = y_1 : B_1, \ldots, y_m : B_m$). Here all A_i and B_j are LF types, and for a formula to be well-formed the combined context Γ_1, Γ_2 must be a valid LF context.

While this may not seem very expressive, it is sufficient for many theorems in the realm of logic and the theory of programming languages we have examined, since other connectives (such as disjunction) and even more complex quantifier alternations can be incorporated at the level of LF. The main limitation is that the quantifiers range only over closed LF objects of the given types; a generalization is the subject of current research. Assumptions are labelled with proof term variables $\mathbf{x}$ which are used in the proof terms P.

$$\begin{array}{ll} \textit{Formulas} & F ::= \forall \Gamma_1.\, \exists \Gamma_2.\, \top \\ \textit{Assumptions} & \Delta ::= \cdot \mid \Delta, \mathbf{x} \in F \end{array}$$

The main judgment of this sequent calculus is $\Gamma; \Delta \vdash P \in F$, where the LF context Γ makes all Eigenvariables explicit together with their types. The judgment is also indexed by an LF signature Σ which we suppress for the sake of brevity.

The rules for the judgment are in the form of a sequent calculus and defined in Figure 1. Because of the way our search engine actually works and the restriction on quantifier alternations, it is convenient to instantiate all quantified variables of the same kind simultaneously by means of a substitution σ explained below. This applies to the $\forall\mathsf{L}$ and $\exists\mathsf{R}$ rules, where the latter also incorporates an axiom rule for $\top$. The reader may wish to ignore the proof terms in the first reading, which are not essential until recursion is introduced in Section 3.2.

$$\textit{Substitutions}\ \sigma ::= \cdot \mid \sigma, M/x$$

Valid substitutions map variables in a context Γ' to valid objects in a context Γ. This judgment is written as $\Gamma \vdash \sigma : \Gamma'$ and defined by the following inference rules, which guarantee that dependencies are respected.

$$\frac{}{\Gamma \vdash \cdot : \cdot}\,\mathsf{subId} \qquad \frac{\Gamma \vdash \sigma : \Gamma' \qquad \Gamma \vdash M : A[\sigma]}{\Gamma \vdash (\sigma, M/x) : (\Gamma', x : A)}\,\mathsf{subDot}$$

When $\Gamma \vdash \sigma : \Gamma'$ and $\Gamma' \vdash M : A$ then we write $M[\sigma]$ for the result of applying the substitution σ to M, and similarly for types, contexts, etc. The result satisfies $\Gamma \vdash M[\sigma] : A[\sigma]$. This is also reflected in our implementation of the system, which employs dependently typed explicit substitutions. We write id_Γ for the identity substitution on Γ satisfying $\Gamma \vdash \mathrm{id}_\Gamma : \Gamma$.

The formulation of the calculus incorporates the structural rules: weakening is implicit in $\exists\mathsf{R}$, contraction and exchange are implicit in the left rules $\forall\mathsf{L}$ and $\exists\mathsf{L}$. The type preservation theorem (Theorem 1) can be expressed in $\mathcal{M}_2$ as

$$\forall E : \mathrm{exp}, V : \mathrm{exp}, T : \mathrm{tp}, D : \mathrm{ev}\ E\ V, P : \mathrm{of}\ E\ T.\, \exists Q : \mathrm{of}\ V\ T.\, \top$$

The variables E, V and T appear as index objects in the types of D, P, and Q and are therefore called *index variables*. Index variables are treated differently from other variables during proof search, as we will see in Section 4. We adopt the convention to omit their quantifier and denote them with bold uppercase names. In this way the theorem can be abbreviated to $\forall D : \mathrm{ev}\ \mathbf{E}\ \mathbf{V}, P : \mathrm{of}\ \mathbf{E}\ \mathbf{T}.\, \exists Q : \mathrm{of}\ \mathbf{V}\ \mathbf{T}.\, \top$.

The system presented in Figure 1 is the core of the meta logic $\mathcal{M}_2$ for LF. In the next two sections we strengthen $\mathcal{M}_2$ by introducing well-founded recursion and definition by cases for closed LF objects. This will allow us to represent many proofs by structural induction, case distinction, and inversion in $\mathcal{M}_2$. A further extension of $\mathcal{M}_2$ is the introduction of conjunction which is required for the representation of mutually inductive proofs, but omitted here for the sake of brevity (see [SP98]).

3.2 Adding Recursion

The recursion operator $\mu\mathbf{x} \in F.\, P$ is the standard fixed point operator at the level of proof terms with the following introduction rule.

$$\frac{\Gamma; \Delta, \mathbf{x} \in F \vdash P \in F}{\Gamma; \Delta \vdash \mu\mathbf{x} \in F.\, P \in F}\,\mathsf{fix} \quad (\text{where } \mu\mathbf{x} \in F.\, P \text{ terminates in } \mathbf{x})$$

It is obvious that a proof term represents a total function only if it terminates independently of the arguments it is applied to. Thus the side condition on the rule. For termination we use arbitrary lexicographic extensions of the subterm ordering on LF objects described in [RP96], all of which are well-founded orderings and easy to check due to the restricted nature of our meta-logic.

3.3 Adding Case Analysis

The context of Eigenvariables Γ in the judgment $\Gamma; \Delta \vdash P \in F$ represents all LF variables which might occur free in the proof term P. Because of the assumption that proof terms are only applied to closed LF objects, all variables in Γ stand for closed LF objects. It is therefore possible to determine all possible cases for the top-level constructor of such objects.

Assume we would like to distinguish all possible cases for a given LF variable x of type A declared in Γ. For simplicity, we assume that A is a base type, even though in the full system [SP98] function types are also permitted which is needed, for example, in the proof of the deduction theorem. The top-level structure of a closed canonical term of base type is always $c\ x_1 \ldots x_n$, where x_i are new variables. If c has type $\Pi x_1 : A_1 . \ldots \Pi x_n : A_n . B$, then this is a possible candidate for the shape of $x : A$ if B unifies with A.

This idea is very similar to the realization of partial inductive definitions and definitional reflection [SH93], except that dependent types can eliminate more cases statically. Also, because of the higher-order nature of the term language, we need to deal with the undecidability of the full higher-order unification problem. Our solution is to restrict the analysis of possible cases to Miller's higher-order patterns, generalized to the setting of dependent types [Pfe91]. However, we do not restrict our system to patterns statically, since this would preclude, for example, a direct appeal to substitution or substitution lemmas at the level of LF. Instead, we simply rule out definition by cases where determining the possible cases would require unification beyond the pattern fragment.

Formally, we extend the language of proof terms by a case construct.

$$\begin{array}{ll} \textit{Patterns} & R ::= \Gamma'; \Gamma'' \triangleright M \\ \textit{Cases} & \Omega ::= \cdot \mid \Omega, R \mapsto P \\ \textit{Proof Terms} & P ::= \ldots \mid \text{case } x \text{ of } \Omega \end{array}$$

The objects M in patterns are strongly restricted by the rules which check valid patterns; usually it will be a constant applied to variable arguments, but because of dependencies, it might be more complex than that. Contexts Γ' and Γ'' are separated for technical reasons, where Γ' contains the variables which will be instantiated when the case subject is matched against the object M, while Γ'' contains those variables which will not be instantiated (although their types could still be instantiated). We always have that $\Gamma', \Gamma'' \vdash M : A'$ for some type A' which is equal to or more specific than the type A of the case subject x.

The judgment for checking the validity of a case construct has the form $\Gamma_1; x : B_x; \Gamma_2; \Delta \vdash_\Sigma \ \Omega \in F$, where we maintain the invariant that B_x depends on all variables in Γ_1, which therefore collects the variables which will be instantiated by pattern matching. By using the limited permutation properties of LF [HHP93] this can always be established. The following rule then completes the definition of derivability in $\mathcal{M}_2$.

$$\frac{\Gamma(x) = B_x \qquad \Gamma_1; x : B_x; \Gamma_2; \Delta \vdash_\Sigma \ \Omega \in F}{\Gamma; \Delta \vdash \text{case } x \text{ of } \Omega \in F}\,\mathsf{case}$$

where $\Gamma_1, x : B_x, \Gamma_2$ must be a valid permutation of Γ, and B_x depends on all variables in Γ_1. The judgment $\Gamma_1; x : B_x; \Gamma_2; \Delta \vdash_\Sigma \ \Omega \in F$ selects all constants from Σ which are possible constructors for a closed object of type B_x. The rules for the judgment are given in Figure 2. This judgment iterates through the signature Σ, trying each constant c in turn. If the target type B_c unifies with

$$\frac{}{\Gamma_1; x : B_x; \Gamma_2; \Delta \vdash_{\cdot} \ \cdot \in F}\ \mathsf{sigempty}$$

$$\frac{\Gamma_1; x : B_x; \Gamma_2; \Delta \vdash_{\Sigma} \ \Omega \in F}{\Gamma_1; x : B_x; \Gamma_2; \Delta \vdash_{\Sigma, c : \Pi \Gamma_c.\, B_c} \ \Omega \in F}\ \mathsf{signonuni} \quad (B_x, B_c \text{ do not unify})$$

$$\frac{\Gamma', \Gamma_2[\sigma]; \Delta[\sigma'] \vdash P \in F[\sigma'] \qquad \Gamma_1; x : B_x; \Gamma_2; \Delta \vdash_{\Sigma} \ \Omega \in F}{\Gamma_1; x : B_x; \Gamma_2; \Delta \vdash_{\Sigma, c : \Pi \Gamma_c.\, B_c} \ \Omega, (\Gamma'; \Gamma_2[\sigma] \triangleright (c\ \Gamma_c)[\sigma] \mapsto P) \in F}\ \mathsf{siguni}$$

$$\Gamma' \vdash \sigma = \mathrm{mgu}\,(B_x \doteq B_c, x \doteq c\ \Gamma_c) : (\Gamma_1, x : B_x, \Gamma_c)$$
$$\Gamma', \Gamma_2[\sigma] \vdash \sigma' = (\sigma, \mathrm{id}_{\Gamma_2}) : (\Gamma_1, x : B_x, \Gamma_c, \Gamma_2)$$

Fig. 2. Selection rules for $\Gamma_1; x : B_x; \Gamma_2; \Delta \vdash_{\Sigma} \ \Omega \in F$

the type B_x of the case subject (siguni), a new case is added to Ω. Otherwise, c cannot be a top-level constructor for a closed term M of type B_x and no case is added (signonuni).

In the rules we use $\Pi \Gamma_c.\, B_c$ as a compact notation for the type of the object constant c, where B_c is an atomic type. We write $c\ \Gamma_c$ for the result of applying c to the variables in Γ_c in order, which gives us the most general form of a term in canonical form whose head is c. The side conditions of siguni determine a substitution σ', which instantiates all variables in Γ_1 according to the unification of B_c and B_x, x by $c\ \Gamma_c$, and acts on all variables in Γ_2 as the identity substitution.

3.4 Properties of $\mathcal{M}_2$

The principal property of $\mathcal{M}_2$ which justifies its use for reasoning about closed LF objects is the following.

Theorem 2. *If* $\cdot; \cdot \vdash P \in \forall \Gamma_1.\, \exists \Gamma_2.\, \top$ *is derivable for some* P*, then for every closed substitution* $\cdot \vdash \sigma_1 : \Gamma_1$ *there exists a substitution* $\cdot \vdash \sigma_2 : \Gamma_2[\sigma_1]$.

As indicated at the end of Section 2, this, together with the adequacy of the encodings, guarantees the meta-theoretic properties of the object languages we can express in $\mathcal{M}_2$. Note that this is different from and in many ways simpler than a full cut-elimination result for $\mathcal{M}_2$.

The proof of this central property is non-trivial. What we show is that the realizing proof terms P can be used to calculate σ_2 from σ_1. For this purpose, we define a small-step, call-by-value, continuation-passing operational semantics for proof terms P with explicit environments and establish the following three properties.

Type Soundness: Each step in the evaluation of P preserves types and provability in $\mathcal{M}_2$ (the critical idea here is the use of explicit environments rather than substitution, since substitution may render some branches in a case distinction inapplicable, thereby invalidating it).

Progress: At each step we either have a final result, or a rule in the operational semantics applies (the critical step here shows that all possibilities are covered in a definition by cases).

Termination: All reduction sequences terminate (the critical step here uses the well-foundedness restriction on recursion).

Unfortunately, space does not permit us to show the details of this proof or even the definition of the operational semantics. The interested reader is referred to [SP98].

4 Twelf

Twelf is a theorem prover for LF which directly implements the meta-logic $\mathcal{M}_2$ (including mutual induction and distinction by cases over functions). It provides an interactive mode for experimentation and an automatic mode in which only the theorem and the termination ordering are specified. The deduction engine implements only a few elementary operations which are used to formalize the three important basic proof principles: inversion (that is, determining all possible shapes of an LF object from its type), direct proofs (that is, direct construction of an LF object), and appeals to the induction hypothesis. The interactive mode also supports lemma application.

4.1 Elementary Operations

We discuss the elementary operations using the proof of the type preservation theorem as an example. The initial goal

$$\forall D : \mathrm{ev}\ \mathbf{E}\ \mathbf{V}, P : \mathrm{of}\ \mathbf{E}\ \mathbf{T}.\, \exists Q : \mathrm{of}\ \mathbf{V}\ \mathbf{T}.\, \top$$

and the induction principle (induction over D) are specified by the user. Twelf uses only outermost induction, so there is an implicit application of the recursion rule before the real proof process is started. Then Twelf generates subgoals by applying its elementary operations until all subgoals are solved, using the strategy described in Section 4.2.

The most basic step directly constructs a substitution for the existentially quantified variables using the constants from the signature and the universally quantified variables. We call this step *filling*. It is basically a straightforward, iterative-deepening search over an LF signature and is derived from a related implementation of resolution for logic programming [Pfe94].

In our example, such a substitution does not exist for the current state, so the system applies the *splitting* operation which performs a case analysis: it inspects the signature for possible constructors for D and generates a list of three subgoals, automatically updating the context of universal variables.

Case: $D = \mathrm{ev\_lam}$:

$$\forall P : \mathrm{of}\ (\mathrm{lam}\ \mathbf{E})\ \mathbf{T}.\, \exists Q : \mathrm{of}\ (\mathrm{lam}\ \mathbf{E})\ \mathbf{T}.\, \top$$

Case: $D = \text{ev\_app}\ D_3\ D_2\ D_1$:

$$\forall D_3 : \text{ev}\ (\mathbf{E_1'}\ \mathbf{V_2})\ \mathbf{V},\ D_2 : \text{ev}\ \mathbf{E_2}\ \mathbf{V_2}, D_1 : \text{ev}\ \mathbf{E_1}\ (\text{lam}\ \mathbf{E_1'}),\\ P : \text{of}\ (\text{app}\ \mathbf{E_1}\ \mathbf{E_2})\ \mathbf{T}.\ \exists Q : \text{of}\ \mathbf{V}\ \mathbf{T}.\ \top$$

Case: $D = \text{ev\_fix}\ D_1$:

$$\forall D_1 : \text{ev}\ (\mathbf{E}\ (\text{fix}\ \mathbf{E}))\ \mathbf{V}, P : \text{of}\ (\text{fix}\ \mathbf{E})\ \mathbf{T}.\ \exists Q : \text{of}\ \mathbf{V}\ \mathbf{T}.\ \top$$

For the sake of brevity, we skip the discussion of the first two subgoals, and continue with the third. Inversion is now applied to P in the informal proof, since there is only one typing rule with a conclusion of the form $\cdot \triangleright \mathbf{fix}\ x.\, e : \tau$. In Twelf, inversion is realized by another splitting operation which generates only one subgoal in this example. The other two potential cases (of_lam, of_app) do not need to be considered by Twelf, because their types are incompatible with the type of P. This leaves the subgoal

$$\forall D_1 : \text{ev}\ (\mathbf{E}\ (\text{fix}\ \mathbf{E}))\ \mathbf{V},\ P_1 : \Pi x{:}\text{exp.}\ \text{of}\ x\ \mathbf{T} \to \text{of}\ (\mathbf{E}\ x)\ \mathbf{T}.\ \exists Q : \text{of}\ \mathbf{V}\ \mathbf{T}.\ \top.$$

Note, that in this goal the variable P_1 is functional and represents a hypothetical derivation.

It is now possible to appeal to the induction hypothesis in an operation we call *recursion*. The termination condition of the fix-rule requires that it is only applied to a term smaller than $D = \text{ev\_fix}\ D_1$. According to the termination ordering in [RP96] there is only one possibility, namely D_1.

We cannot appeal to the induction hypothesis without providing a typing derivation as second argument. Formally, the representation of this derivation must be of type 'of ($\mathbf{E}$ (fix $\mathbf{E}$)) $\mathbf{T}$'. Twelf searches and finds the term 'P_1 (fix $\mathbf{E}$) (of_fix P_1)' which represents the result of applying the substitution lemma (Lemma 1) as used in the proof of Theorem 1. If we call the result of the appeal to the induction hypothesis Q_2, we obtain the following subgoal.

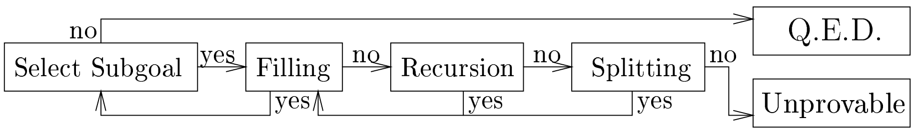
$$\forall D_1 : \text{ev}\ (\mathbf{E}\ (\text{fix}\ \mathbf{E}))\ \mathbf{V},\ P_1 : \Pi x{:}\text{exp.}\,\text{of}\ x\ \mathbf{T} \to \text{of}\ (\mathbf{E}\ x)\ \mathbf{T},\ Q_2 : \text{of}\ \mathbf{V}\ \mathbf{T}.\\ \exists Q_1 : \text{of}\ \mathbf{V}\ \mathbf{T}.\ \top$$

Twelf is now able to determine in a simple filling step that Q_2 is a possible instantiation for Q_1, thereby completing the ev_fix-branch of the proof. The other two branches can be solved similarly. Twelf then reports the proof term (currently shown in a more readable relational notation as an LF signature, rather than in the functional notation used to define $\mathcal{M}_2$).

4.2 Strategy

The proof strategy of Twelf is a simple combination of the three elementary operations. But each operation must be applied with care because they are inherently expensive in time and space. In particular, we completely avoid backtracking except locally during the filling step. Splitting, filling, and recursion use unification to analyze cases and to select constants. Recursion triggers the calculation of possible recursion arguments according to the termination ordering [RP96].

For a given theorem and induction principle, Twelf attempts to construct a derivation in $\mathcal{M}_2$ using the following strategy:



There is a global store of yet to be proven subgoals, initialized with the formula representing the theorem. Once the automated proof process is started, the strategy activates a subgoal and tries to apply a filling operation.

Filling: The filling operation corresponds to an application of the $\exists$R-rule: it is applicable if a substitution instantiating all existentially quantified variables can be constructed. Because index variables occur in the types of non-index variables, it is already enough to determine instantiations for all non-index variables (see Section 2). In general, infinitely many substitutions must be examined, but since our strategy is parameterized by a number to limit the depth of the search space, the employed search algorithm is incomplete but will always terminate (even though failure is sometimes slow).

If Twelf succeeds in constructing the substitution, the current subgoal is successfully completed and the next subgoal is selected if available, otherwise Twelf stops (Q.E.D.). If Twelf fails to construct the desired substitution the strategy tries to apply the recursion operation.

Recursion: The recursion operation corresponds to an application of the $\forall$L-rule, immediately followed by an application of the $\exists$L-rule: Twelf generates all possible recursive calls by constructing substitutions which correspond to the arguments of the recursive call. These substitutions must satisfy the side condition of the fix-rule. Because lower-ranked arguments in a lexicographic termination order actually may increase in size, there are potentially infinitely many different ways to appeal to the induction hypothesis. Moreover results of recursive calls can be used to form new ones. Hence, to avoid an infinite chain of applications of induction hypotheses, our strategy is parameterized by an upper bound on the number of recursive calls. If no new recursive calls can be generated, the strategy tries to apply the splitting operation.

Splitting: The splitting operation corresponds to an application of the case-rule. Twelf selects a universally quantified variable which is not an index variable (see Section 2). Its type is then used to determine all of its possible shapes (sigempty, siguni, and signonuni). For each shape, a new subgoal is created. Twelf then selects among those an active subgoal and tries to apply the filling operation.

Experiment	Ind	Lim	Filling	Splitting	Recursion	Total
Cartesian Closed Categories	1	4	1.000	0.004	0.036	1.099
CPM Completeness	1	20	0.916	0.010	0.117	1.134
CPM Proof equivalence: $\Rightarrow$	1	6	0.226	0.034	0.442	0.951
CPM Proof equivalence: $\Leftarrow$	1	6	0.280	0.033	0.647	1.235
Horn LP Soundness	3	4	4.336	0.004	0.049	4.501
Horn LP Canonical forms	3	4	0.028	0.009	0.107	0.303
Horn LP Completeness	2	4	0.015	0.005	0.039	0.195
Mini-ML Value soundness	1	3	0.016	0.041	0.061	0.172
Mini-ML Type preservation	1	6	0.062	0.521	0.150	0.799
Mini-ML Evaluation/Reduction	1	9	25.397	0.007	0.078	25.546
Hilbert's abstraction theorem	1	4	0.197	0.004	0.010	0.322
Associativity of +	1	3	0.009	0.012	0.016	0.063
Commutativity of +	2	3	0.092	0.609	4.139	4.877

Fig. 3. Experimental results (in CPU seconds)

Among all universally quantified variables Twelf selects the one which generates the least number of subgoals first (which could be zero if a variable has a dependent type which does not unify with any constructor type—the subgoal succeeds immediately in that case). This heuristic works surprisingly well in all our examples, we leave a refinement to future research. To avoid an infinite loop of splits (applying splits to the children of a previous split), Twelf is parametrized by a splitting limit. Hence, there are two cases when the strategy may stop unsuccessfully: Either there are no further splittable universally quantified variables available, or their types fall outside Miller's pattern fragment. In both cases the strategy stops with the message that a proof could not be found.

4.3 Experimental Results

Twelf has successfully proved several non-trivial theorems automatically. In Figure 3 we give an overview over the experimental results from the areas of programming languages and logics. "Ind" states how many simultaneous induction hypotheses are necessary and "Lim" the maximal size for LF objects (counting variables and constants, excluding index objects). In all examples the splitting limit is 2, and the number of recursive calls in each case is limited to 10. "Total" summarizes time spent for filling, splitting, recursion and miscellaneous tasks such as parsing, and type reconstruction.

All timings are in CPU seconds, include garbage collection, and have been taken on a 300 Mhz Pentium-II machine, running Linux 2.30, New Jersey SML 110, and Twelf 1.2.

In the area of Mini-ML, Twelf was used to prove value soundness, i.e., if $e \hookrightarrow v$ then v is a value, and type preservation (Theorem 1). The third related theorem, namely that if $e \hookrightarrow v$ then e reduces to v was particularly difficult with our strategy, since the search space for reductions is rather unstructured. Most of the time here is spent in failed attempts to fill incomplete subgoals before appeals to the induction hypothesis generate the necessary auxiliary reductions. Twelf also proved completeness of a continuation passing machine (CPM) with respect to a natural semantics for Mini-ML. The proof constitutes a mapping from Mini-ML evaluations to computation traces of the abstract machine. But Twelf cannot verify the soundness direction, because the proof requires complete induction which is currently not supported. Nonetheless, Twelf could prove that the soundness proof (coded by hand) can be mapped onto the completeness proof and vice versa.

In the area of logic, Twelf was used to prove the deduction theorem for intuitionistic propositional logic using Hilbert's axiomatization which is used to translate pure functional programs into combinators. It also proved soundness and completeness of uniform derivations with respect to resolution for Horn-logic. From the area of category theory, it proved that Cartesian closed categories can be embedded into the simply-typed λ-calculus. Finally, we have carried out some more traditional inductions, proving the associativity and commutativity of addition on unary natural numbers. Especially the latter is interesting, since Twelf spends most of its time exploring various ways to apply the rather general induction hypothesis, while in most other examples filling is the most expensive operation.

5 Related Work and Future Work

There have been many mechanized proofs of meta-theoretic properties of logics or programming languages in the literature (see the survey [Pfe96]). Most of these do not use techniques from logical frameworks, but represent the languages via standard inductive types and their semantics by inductively defined predicates. A popular choice for such encodings are de Bruijn indices, since they eliminate the problem of α-conversion from consideration. However, various lemmas regarding substitution must still be shown and used, which severely limits the degree of automation which can be achieved. Most closely related to our own efforts in this area is the work on ALF [Mag95], since ALF also employs dependently typed pattern matching and termination orderings, although without the benefits of higher-order abstract syntax.

Another approach is to represent meta-theoretic proofs as relations in LF, which leaves the progress and termination properties above to an external check on relations [PR92]. In this approach, there is no automation besides type reconstruction. The expressive power of LF makes this feasible, but it remains tedious.

Most closely related to our approach is work by McDowell and Miller [MM97] who also define a higher-order meta-logic $FO\lambda^{\Delta I\!N}$ for a logical framework (hereditary Harrop formulas) and then reason in the meta-logic. Their approach is based entirely on simple types and does not incorporate proof terms, which makes it less suitable for automation. Moreover, in order to establish consistency for their meta-logic, they limit induction to natural numbers, which also complicates automation. In fact, their implementation based on the Pi proof editor [Eri94] is entirely interactive. On the other hand, $FO\lambda^{\Delta I\!N}$ does not restrict itself to Π_2-formulas. In addition, McDowell has demonstrated the flexibility of his approach in his thesis [McD97] where he also treats a logical framework incorporating linearity. Since the overall architecture is quite similar, this gives us confidence that our approach may be extended to a linear logical framework [CP96], which is planned in future work. We believe that the separation between logical framework and meta-logic, and the separation between definition by cases and well-founded recursion are all critical ingredients in making this idea successful for even richer logical frameworks than LF.

While the set of theorems we can prove at present is already surprisingly rich, they are limited by three factors: (1) we do not attempt to automatically use lemmas, (2) only lexicographic extensions of subterm orderings are permitted to show termination, and (3) $\mathcal{M}_2$ does not support reasoning about *open* LF objects. We believe that (1) and (2) can be addressed by incorporating standard techniques from inductive theorem proving, efficiency improvements such as indexing, and simply allowing more complex termination orderings. Nonetheless, we have currently no plans for developing Twelf into a general-purpose theorem prover, because we feel that its present success owes mostly to its design as a special-purpose prover for properties of programming languages and logics. We are currently investigating how to incorporate ideas from schema-checking [Roh94] and primitive recursion over higher-order abstract syntax [DPS97] into our meta-logical framework in order to make progress on item (3), that is, allow reasoning over terms which may have free variables from certain regular contexts which arise in many practical examples.

References

[Coq91] Thierry Coquand. An algorithm for testing conversion in type theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, 1991.

[CP96] Iliano Cervesato and Frank Pfenning. A linear logical framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[DPS97] Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In R. Hindley, editor, *Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, pages 147–163, Nancy, France, April 1997. Springer-Verlag LNCS.

[Eri94] Lars-Henrik Eriksson. Pi: An interactive derivation editor for the calculus of partial inductive definitions. In Alan Bundy, editor, *Proceedings of the Twelfth International Conference on Automated Deduction*, pages 821–825. Springer-Verlag LNAI 814, June 1994.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[Mag95] Lena Magnusson. *The Implementation of ALF—A Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution.* PhD thesis, Chalmers University of Technology and Göteborg University, January 1995.

[McD97] Raymond McDowell. *Reasoning in a logic with definitions and induction.* PhD thesis, University of Pennsylvania, 1997.

[MM97] Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax: An extended abstract. In Glynn Winskel, editor, *Proceedings of the Twelfth Annual Symposium on Logic in Computer Science*, Warsaw, Poland, June 1997. To appear.

[MP91] Spiro Michaylov and Frank Pfenning. Natural semantics and some of its meta-theory in Elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, pages 299–344, Stockholm, Sweden, January 1991. Springer-Verlag LNAI 596.

[Pfe91] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.

[Pfe92] Frank Pfenning. Computation and deduction. Unpublished lecture notes, 277 pp. Revised May 1994, April 1996, May 1992.

[Pfe94] Frank Pfenning. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815, Nancy, France, June 1994. Springer-Verlag LNAI 814. System abstract.

[Pfe96] Frank Pfenning. The practice of logical frameworks. In Hélène Kirchner, editor, *Proceedings of the Colloquium on Trees in Algebra and Programming*, pages 119–134, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1059. Invited talk.

[PR92] Frank Pfenning and Ekkehard Rohwedder. Implementing the meta-theory of deductive systems. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, pages 537–551, Saratoga Springs, New York, June 1992. Springer-Verlag LNAI 607.

[Roh94] Ekkehard Rohwedder. Verifying the meta-theory of deductive systems. Thesis Proposal, February 1994.

[RP96] Ekkehard Rohwedder and Frank Pfenning. Mode and termination checking for higher-order logic programs. In Hanne Riis Nielson, editor, *Proceedings of the European Symposium on Programming*, pages 296–310, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1058.

[SH93] Peter Schroeder-Heister. Rules of definitional reflection. In M. Vardi, editor, *Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 222–232, Montreal, Canada, June 1993.

[SP98] Carsten Schürmann and Frank Pfenning. Automated theorem proving in a simple meta-logic for LF. Technical Report CMU-CS-98-123, Carnegie Mellon University, 1998.