Automating the Meta Theory of Deductive Systems

Thesis Proposal

Carsten Schürmann *
Carnegie Mellon University
School of Computer Science
carsten+@cs.cmu.edu

December 10, 1997

Abstract

Higher-order abstract syntax is a central representation technique in logical frameworks which maps variables of the object language into variables of the meta language. On the one hand, encodings using this idea are often extremely concise and elegant, on the other, higher-order representations are no longer inductive, which means that standard techniques for reasoning by induction do not apply.

In this proposal, we present a meta-logic for LF ($\mathcal{M}_\omega$) which overcomes this problem by representing inductive proofs as total functions. It uses recursion and pattern-matching to mimic strong induction principles. Because of the elegant representation of proofs it is possible to devise an efficient proof search algorithm for a fragment $\mathcal{M}_2$ of $\mathcal{M}_\omega$. It has been implemented as an automated theorem prover for the proof assistant Twelf, a reimplementation and extension of the logic programming language Elf. Twelf has been used to prove fully automatically (among others) type preservation for Mini-ML and Hilbert's deduction theorem. But $\mathcal{M}_\omega$ is less general than desired: Inductive arguments which require hypothetical reasoning — introducing new assumptions represented by parameters — cannot be represented. $\mathcal{M}_\omega$'s functions can only map closed objects to closed objects. Thus for example, the proof of the Church-Rosser theorem given in [Pfe93] is not representable in $\mathcal{M}_\omega$.

In my thesis, I propose to extend $\mathcal{M}_\omega$ to a meta logic whose functions are total but can map — in a well-defined way — open terms to open terms. I also plan to extend Twelf to an automated proof assistant for this new calculus. This way, I hope to achieve a fully automatic proof of the cut-elimination theorem for intuitionistic and classical logic [Pfe95].

1 Introduction

The logical framework LF formulated by Harper, Honsell, and Plotkin [HHP93] has been used as a suitable and appropriate meta language to represent deductive systems. Based on LF, the logic programming language Elf developed by Pfenning [Pfe89], allows the representation of many non-trivial examples such as the Church-Rosser theorem [Pfe93], compiler verification [HP92a], Mini-ML [Pfe94b], logics[Pfe94b], categorical structures [Geh94], occurrences of continuation

*This work was sponsored by NSF Grant CCR-9619584

parameters in CPS terms [DP95], and many others. Elf as a tool has also been successful in the discovery of new and elegant proofs as for example a proof of the cut-elimination theorem for classical, intuitionistic and linear logic [Pfe95].

Higher-order abstract syntax is a central representation technique in LF. Many of the aforementioned examples make use of this technique. The basic idea is to represent variables of the object language by variables of the meta-language. Consequently, object language constructs which bind variables must be represented by meta-language constructs which bind the corresponding variables.

This deceptively simple idea, which goes back to Church [Chu40], Martin-Löf's system of arities [NPS90], and Constable's Nuprl [C^+86], has far-reaching consequences for the methodology of logical frameworks. On the one hand, encodings of logical systems using this idea are often extremely concise and elegant, since common concepts and operations such as variable binding, variable renaming, capture-avoiding substitution, or parametric and hypothetical judgments are directly supported by the framework and do not need to be encoded separately in each application. On the other hand, higher-order representations are no longer inductive in the usual sense, which means that standard techniques for reasoning by induction do not apply.

In this work we take a first step in reconciling reasoning about LF signatures using higher-order abstract syntax with representing proofs in a formal system by proposing a meta logic of *total functions* which uses recursion and pattern-matching to mimic strong induction principles. We call this meta-logic $\mathcal{M}_\omega$, where ω expresses that arbitrary alternations between $\forall$ and $\exists$ quantifiers are allowed. For automated proof search, we then restrict $\mathcal{M}_\omega$ to a meta logic $\mathcal{M}_2$ of Π_2-formulas which allows only one alternation of quantifiers.

An automated theorem prover for this meta logic has been implemented as part of the Twelf system, a reimplementation and extension of Elf. It is able to prove value soundness and type preservation for Mini-ML fully automatically. In the setting of compiling a functional language into a continuation stack machine, it also proves the completeness direction of the equivalence theorem of both semantics. Moreover Twelf successfully proved the existence of a monomorphism mapping soundness proofs into completeness proofs. Other fully automatic proof examples include properties of uniform derivations and resolution (from the area of logic programming), the deduction theorem for propositional logic, and an embedding of Cartesian closed categories into the simply typed λ-calculus.

The system $\mathcal{M}_\omega$ allows the representation of proofs over languages using higher-order representation techniques. But $\mathcal{M}_\omega$ is more restricted then desired. The proof terms are total functions only with respect to closed inputs. Many inductive arguments require the introduction of new assumptions (which are represented by parameters), violating this basic property, as for example the proofs of the Church-Rosser theorem [Pfe93] and the cut-elimination theorem for intuitionistic logic [Pfe95]. The proofs of these theorems cannot be presented elegantly in $\mathcal{M}_\omega$.

In my thesis, I propose to extend $\mathcal{M}_\omega$ to a meta logic whose functions are still total, even though during execution they may be applied to open terms. These terms are defined in contexts which are regular in structure because of the way parameters are introduced. Initially, the arguments to the total function representing an entire proof must still be closed. In my thesis, I also plan to extend Twelf to an automated proof assistant for this new calculus. This way, I hope to achieve a full automatic proof of the cut-elimination theorem for intuitionistic logic.

The proposal is organized as follows: In Section 2 we describe briefly the logical framework LF and introduce two running examples: the programming language Mini-ML and the Hilbert calculus. In Section 3 we discuss important proof techniques using the type preservation theorem for Mini-ML and the deduction theorem for the Hilbert calculus as examples. The meta logic $\mathcal{M}_\omega$ is introduced in Section 4. A restricted version of $\mathcal{M}_\omega$ ($\mathcal{M}_2$) is implemented in the proof assistant Twelf which we discuss in Section 5. Section 6 mentions related work before I plan my thesis in Section 7.

2 The logical framework LF

The logical framework LF is an extension of the simply-typed λ-calculus by dependent types. It also appears under the name λP in Barendregt's λ-cube [Bar92]. It is defined by three syntactic categories of objects, types, and kinds [HHP93]:

$$
\begin{array}{lrcl}
\text{Kinds:} & K & ::= & \text{type} \mid \Pi x\!:\!A.\,K \\
\text{Atomic types:} & B & ::= & a \mid B\ M \\
\text{Types:} & A & ::= & B \mid \Pi x\!:\!A_1.\,A_2 \\
\text{Objects:} & M & ::= & c \mid x \mid \lambda x\!:\!A.\,M \mid M_1\ M_2 \\[4pt]
\text{Signatures:} & \Sigma & ::= & a\cdot \mid \Sigma, c : A \mid \Sigma, a : K \\
\text{Contexts:} & \Gamma & ::= & \cdot \mid \Gamma, x : A
\end{array}
$$

We use a for type constants, c for object constants, and x for variables. 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.

As typing judgments, we use $\Gamma \vdash M : A$ for objects and $\Gamma \vdash A : K$ for types. Well-formed contexts Γ satisfy $\vdash \Gamma$ ctx. The interested reader should consult [HHP93, Pfe94b] for the rules defining these judgments, which we will implicitly use throughout this proposal.

LF's main strength is the ability to represent deductive systems adequately. Judgments are represented by types, the inference rules by object declarations in the signature. Hence a derivation of some judgment is expressed by a corresponding object in LF. As one of two running examples, we use a very simple functional language, a subset of Mini-ML [Pfe94b]. It must be noted that the development throughout this paper has been extended to full Mini-ML. The other example is the Hilbert calculus, for which we prove the deduction theorem in the next section.

2.1 Mini-ML

Mini-ML illustrates the idea of higher-order abstract syntax. Higher-order abstract syntax exploits the full expressive power of LF where λ-abstraction provides the mechanism to represent binding. Mini-ML variables are represented by meta-language variables.

Example 1 (Mini-ML)

$$\text{Expressions:}\quad e \quad ::= \quad x \mid \mathbf{lam}\,x.\,e \mid e_1 @ e_2 \mid \mathbf{fix}\,x.\,e$$

$$
\begin{array}{rcll}
 & & & \text{exp} : \text{type} \\
\ulcorner \mathbf{lam}\,x.\,e \urcorner & = & \text{lam}\ (\lambda x\!:\!\text{exp}.\ulcorner e \urcorner) & \text{lam} : (\text{exp} \to \text{exp}) \to \text{exp} \\
\ulcorner e_1 @ e_2 \urcorner & = & \text{app}\ \ulcorner e_1 \urcorner\ \ulcorner e_2 \urcorner & \text{app} : \text{exp} \to \text{exp} \to \text{exp} \\
\ulcorner \mathbf{fix}\,x.\,e \urcorner & = & \text{fix}\ (\lambda x\!:\!\text{exp}.\ulcorner e \urcorner) & \text{fix}\ \ : (\text{exp} \to \text{exp}) \to \text{exp} \\
\ulcorner x \urcorner & = & x &
\end{array}
$$

Not every well-typed object of the meta-language directly represents an expression of the object language. For example, we can see that $\ulcorner e \urcorner$ will never contain a β-redex. Moreover, the argument to lam which has type exp $\to$ exp will always be a λ-abstraction. Thus the image of the translation in this representation methodology is always a β-normal η-long form. Following [HHP93] we call these forms canonical and write $\Gamma \vdash M \Uparrow A$ where $\Gamma \vdash M : A$.

Canonical forms play the role of "observable values" in a functional language: they are in one-to-one correspondence with the expressions we are trying to represent. For Mini-ML this is expressed by the following property, which is proved by simple inductions.

Example 2 (Compositional adequacy for untyped λ-expressions)

1. Let e be an expression with free variables among $x_1, \ldots, x_n$.

   Then $x_1 : \text{exp}, \ldots, x_n : \text{exp} \vdash \ulcorner e \urcorner \Uparrow \text{exp}$.

2. Let $x_1 : \text{exp}, \ldots, x_n : \text{exp} \vdash M \Uparrow \text{exp}$

   Then $M = \ulcorner e \urcorner$ for an expression e with free variables among $x_1, \ldots, x_n$.

3. $\ulcorner \cdot \urcorner$ is a bijection between expressions and canonical forms where $\ulcorner e[e'/x] \urcorner = \ulcorner e \urcorner[\ulcorner e' \urcorner/x]$.

Since every object in LF has a unique $\beta\eta$-equivalent canonical form, the meaning of every well-typed object is unambiguously given by its canonical form. Following [Pfe94b] we say that exp is a *level 0* type family. Expressions form the basic syntactic structure for which we define an operational semantics and a typing discipline.

Example 3 (Operational Semantics) Let $e \hookrightarrow v$ be the evaluation judgment. The rules are as follows.

$$\frac{}{\mathbf{lam}\, x.\, e \hookrightarrow \mathbf{lam}\, x.\, e}\,\mathsf{evLam} \qquad \frac{e_1 \hookrightarrow \mathbf{lam}\, x.\, e_1' \qquad e_2 \hookrightarrow v_2 \qquad e_1'[v_2/x] \hookrightarrow v}{e_1 @ e_2 \hookrightarrow v}\,\mathsf{evApp}$$

$$\frac{e[\mathbf{fix}\, x.\, e/x] \hookrightarrow v}{\mathbf{fix}\, x.\, e \hookrightarrow v}\,\mathsf{evFix}$$

Evaluation derivations can be represented adequately in LF. The representation of the judgment $\ulcorner e \hookrightarrow v \urcorner$ is a dependent type 'ev $\ulcorner e \urcorner \ulcorner v \urcorner$' (ev is already a *level 1* type family). Moreover

$$\ulcorner \frac{\begin{array}{c}\mathcal{D}\\ e[\mathbf{fix}\, x.\, e/x] \hookrightarrow v\end{array}}{\mathbf{fix}\, x.\, e \hookrightarrow v}\,\mathsf{evFix} \urcorner \quad = \quad \text{ev\_fix}\ (\lambda x{:}\text{exp}.\ulcorner e \urcorner)\ \ulcorner v \urcorner\ \ulcorner \mathcal{D} \urcorner.$$

The other rules are represented similarly, leading to a signature of the following form:

$$\begin{array}{lcl}
\text{ev} & : & \text{exp} \to \text{exp} \to \text{type} \\
\text{ev_lam} & : & \text{ev}\ (\text{lam}\ \mathbf{E})\ (\text{lam}\ \mathbf{E}) \\
\text{ev_app} & : & \text{ev}\ (\text{app}\ \mathbf{E_1}\ \mathbf{E_2})\ \mathbf{V} \\
 & & \leftarrow\ \text{ev}\ \mathbf{E_1}\ (\text{lam}\ \mathbf{E_1'}) \\
 & & \leftarrow\ \text{ev}\ \mathbf{E_2}\ \mathbf{V_2} \\
 & & \leftarrow\ \text{ev}\ (\mathbf{E_1'}\ \mathbf{V_2})\ \mathbf{V} \\
\text{ev_fix} & : & \text{ev}\ (\text{fix}\ \mathbf{E})\ \mathbf{V} \\
 & & \leftarrow\ \text{ev}\ (\mathbf{E}\ (\text{fix}\ \mathbf{E}))\ \mathbf{V}
\end{array}$$

In this example we reversed the clauses by using a $\leftarrow$ notation to improve readability. Since $\rightarrow$ is right associative, $\leftarrow$ is left associative:

$$A \leftarrow B \leftarrow C = \; (A \leftarrow B) \leftarrow C$$

In the case of 'ev_lam' one would expect that $\mathrm{ev\_lam} : \Pi e : \mathrm{exp} \rightarrow \mathrm{exp}.\, \mathrm{ev}\ (\mathrm{lam}\ e)\ (\mathrm{lam}\ e)$. As described elsewhere [Pfe94b], it is not necessary to write out these abstractions explicitly. They are implicitly given, indicated in this proposal by uppercase variable names in bold type. But it should be clear that this notation is merely an abbreviation, and can be expanded to a fully explicit LF type. In this paper we refer to these variables as *index-variables*, because they are used as indices to other type families, as for example, $e : \mathrm{exp} \rightarrow \mathrm{exp}$ is an index to '$\mathrm{ev}\ (\mathrm{lam}\ e)\ (\mathrm{lam}\ e)$'.

Mini-ML is a statically typed language. We distinguish between atomic types and function types which are adequately represented in LF.

Example 4 (Types)

$$\text{Types}: \quad \tau \quad ::= \quad a \mid \tau_1 \Rightarrow \tau_2$$

$$\begin{array}{lclll} & & & \mathrm{tp} & : \mathrm{type} \\ \ulcorner \tau_1 \Rightarrow \tau_2 \urcorner & = & \mathrm{arr}\ \ulcorner \tau_1 \urcorner \ulcorner \tau_2 \urcorner & \mathrm{arr} & : \mathrm{tp} \rightarrow \mathrm{tp} \rightarrow \mathrm{tp} \end{array}$$

The typing judgment for Mini-ML is a hypothetical judgment [Pfe94b] introducing new implicit assumptions. Consider for example the rule for typing Mini-ML functions: To show that $\lambda x : \tau_1 .\, e$ is well-typed, we have to introduce a new parameter x and assume its type to be τ_1 before we can complete the well-typedness proof.

$$\dfrac{\begin{array}{c} \dfrac{}{\triangleright\, x : \tau_1}\, u \\ \vdots \\ \triangleright e : \tau_2 \end{array}}{\triangleright\, \lambda x : \tau_1 .\, e : \tau_1 \Rightarrow \tau_2}\, \mathsf{ofLam}^u$$

In this proposal we prefer to collect all these implicit assumptions and make them explicit. They form a Mini-ML context Δ which provides an alternative way of writing the hypothetical typing judgment:

$$\Delta \triangleright e : \tau$$

In order to avoid confusion between the LF context and the Mini-ML context we will use Δ in this example to designate a Mini-ML context. The typing rules for Mini-ML are given by the following set of inference rules.

Example 5 (Typing rules)

$$\dfrac{\Delta, x : \tau_1 \triangleright e : \tau_2}{\Delta \triangleright \mathbf{lam}\ x.\, e : \tau_1 \Rightarrow \tau_2}\, \mathsf{ofLam} \qquad \dfrac{\Delta \triangleright e_1 : \tau_2 \Rightarrow \tau_1 \qquad \Delta \triangleright e_2 : \tau_2}{\Delta \triangleright e_1 @ e_2 : \tau_1}\, \mathsf{ofApp}$$

$$\dfrac{\Delta, x : \tau \triangleright e : \tau}{\Delta \triangleright \mathbf{fix}\ x.\, e : \tau}\, \mathsf{ofFix}$$

The representation of a derivation $\mathcal{P} :: \Delta \triangleright e : \tau$ is different from previous examples. The Mini-ML context Δ is represented by the LF context. Without going into details, higher-order abstract syntax and hypothetical judgments are closely related. The interested reader might study [Pfe94b] which gives a very detailed account of this matter. It is safe to say that higher-order abstract syntax and the possibility to represent hypothetical judgments elegantly are the primary reasons which make LF such a powerful logical framework. The representation of the typing judgment is a *level 1* type family 'of', such that

$$\ulcorner\Delta\urcorner \vdash \ulcorner\mathcal{P}\urcorner : \mathrm{of}\,\ulcorner e\urcorner\,\ulcorner\tau\urcorner$$

where Δ is represented in the following way:

$$\begin{aligned} \ulcorner . \urcorner &= \cdot \\ \ulcorner \Delta, x : \tau \urcorner &= \ulcorner\Delta\urcorner, x : \mathrm{exp}, u : \mathrm{of}\ x\ \ulcorner\tau\urcorner \end{aligned}$$

To see how the typing rules are represented, consider the rule ofFix. Note that also here the representation technique of hypothetical judgments lets us safely omit Δ from the representation.

$$\ulcorner \dfrac{\begin{array}{c}\mathcal{E}\\ \Delta, x : \tau \triangleright e : \tau\end{array}}{\Delta \triangleright \mathbf{fix}\ x.\, e : \tau}\,\mathsf{ofFix} \urcorner = \mathrm{of\_fix}\ (\lambda x\!:\!\mathrm{exp}.\ulcorner e\urcorner)\ \ulcorner\tau\urcorner\ (\lambda x\!:\!\mathrm{exp}.\,\lambda u\!:\!\mathrm{of}\ x\ \ulcorner\tau\urcorner.\ulcorner\mathcal{E}\urcorner)$$

The complete list of all new LF constants representing the typing rules is given as follows.

$$\begin{array}{lll} \mathrm{of} & : & \mathrm{exp} \to \mathrm{tp} \to \mathrm{type} \\ \mathrm{of\_lam} & : & \mathrm{of}\ (\mathrm{lam}\ \mathbf{E})\ (\mathrm{arr}\ \mathbf{T_1}\ \mathbf{T_2}) \\ & & \leftarrow\ (\Pi x\!:\!\mathrm{exp}.\,\mathrm{of}\ x\ \mathbf{T_1} \to \mathrm{of}\ \mathbf{E}\ \mathbf{T_2}) \\ \mathrm{of\_app} & : & \mathrm{of}\ (\mathrm{app}\ \mathbf{E_1}\ \mathbf{E_2})\ \mathbf{T_1} \\ & & \leftarrow\ \mathrm{of}\ \mathbf{E_1}\ (\mathrm{arr}\ \mathbf{T_2}\ \mathbf{T_1}) \leftarrow\ \mathrm{of}\ \mathbf{E_2}\ \mathbf{T_2} \\ \mathrm{of\_fix} & : & \mathrm{of}\ (\mathrm{fix}\ \mathbf{E})\ \mathbf{T} \\ & & \leftarrow\ (\Pi x\!:\!\mathrm{exp}.\,\mathrm{of}\ x\ \mathbf{T} \to \mathrm{of}\ \mathbf{E}\ \mathbf{T}) \end{array}$$

2.2 Hilbert calculus

The Hilbert calculus has been introduced by Hilbert [HB34] as a proof theory for propositional logic. We consider only the intuitionistic implicational fragment which consists of two axioms and one inference rule, Modus Ponens. Among others it is used for the compilation of functional programming languages into combinators. The language of formulas is defined as follows and can be adequately represented in LF.

Example 6 (Hilbert Formulas)

$$\text{Formulas:}\quad F \quad ::= \quad P \mid F_1 \supset F_2$$

$$\ulcorner F_1 \supset F_2 \urcorner \quad = \quad \ulcorner F_1\urcorner\ \mathrm{imp}\ \ulcorner F_2\urcorner \qquad \begin{array}{l} \mathrm{o}\quad\ : \mathrm{type} \\ \mathrm{imp} : \mathrm{o} \to \mathrm{o} \to \mathrm{o} \end{array}$$

As usual, implication is right-associative: $A \supset B \supset C = A \supset (B \supset C)$. For the sake of readability, we use in the remainder of this proposal the LF-constant 'imp' as infix-operator. We say that a formula F is a provable in the calculus if there exists a derivation of $\vdash F$.

Example 7 (Hilbert Rules)

$$\frac{}{\vdash A \supset B \supset A}\,\mathsf{K} \qquad \frac{}{\vdash (A \supset B \supset C) \supset (A \supset B) \supset (A \supset C)}\,\mathsf{S}$$

$$\frac{\vdash A \supset B \qquad \vdash A}{\vdash B}\,\mathsf{MP}$$

The representation of the provability predicate yields a *level 1* type family: $\ulcorner \vdash F \urcorner = \mathrm{pf}\,\ulcorner F \urcorner$ with $\mathrm{pf} : \mathrm{o} \to \mathrm{type}$. Each rule is represented in the following way

$$\begin{aligned}
\ulcorner \frac{}{\vdash A \supset B \supset A}\,\mathsf{K} \urcorner &= \mathrm{K}\,\ulcorner A \urcorner\,\ulcorner B \urcorner \\
\ulcorner \frac{}{\vdash (A \supset B \supset C) \supset (A \supset B) \supset (A \supset C)}\,\mathsf{S} \urcorner &= \mathrm{S}\,\ulcorner A \urcorner\,\ulcorner B \urcorner\,\ulcorner C \urcorner \\
\ulcorner \frac{\begin{matrix}\mathcal{D}_1 \\ \vdash A \supset B\end{matrix} \qquad \begin{matrix}\mathcal{D}_2 \\ \vdash A\end{matrix}}{\vdash B}\,\mathsf{MP} \urcorner &= \mathrm{MP}\,\ulcorner A \urcorner\,\ulcorner B \urcorner\,\ulcorner \mathcal{D}_1 \urcorner\,\ulcorner \mathcal{D}_2 \urcorner
\end{aligned}$$

which yields the following signature.

$$\begin{array}{lcl}
\mathrm{pf} & : & \mathrm{o} \to \mathrm{type} \\
\mathrm{K} & : & \mathrm{pf}\,(\mathbf{A}\ \mathrm{imp}\ \mathbf{B}\ \mathrm{imp}\ \mathbf{A}) \\
\mathrm{S} & : & \mathrm{pf}\,((\mathbf{A}\ \mathrm{imp}\ \mathbf{B}\ \mathrm{imp}\ \mathbf{C})\ \mathrm{imp}\ (\mathbf{A}\ \mathrm{imp}\ \mathbf{B})\ \mathrm{imp}\ (\mathbf{A}\ \mathrm{imp}\ \mathbf{C})) \\
\mathrm{MP} & : & \mathrm{pf}\,(\mathbf{A}\ \mathrm{imp}\ \mathbf{B}) \to \mathrm{pf}\,\mathbf{A} \to \mathrm{pf}\,\mathbf{B}
\end{array}$$

And again, a simple inductive argument reveals the adequacy of representation. This proof is particularly easy, because provability does not involve higher-order abstract syntax. It is different if we work with a first-order calculus as described in [Pfe94b].

In this section we introduced a simple functional language, its operational semantics and its typing rules, and the Hilbert calculus and its notion of provability. All type families representing the different deductive systems are either *level 0* or *level 1*, i.e. either non-dependent type families, or dependent type families depending only on *level 0* type families. *level 2* type families are used to represent relations between derivations [Pfe94a].

3 Theorems, Proofs, and LF

Because of higher-order abstract syntax, LF is a very elegant logical framework. It is also very powerful because it allows the representation of proofs. To motivate this feature, we present in this section the type preservation theorem for Mini-ML, and the deduction theorem for the Hilbert calculus. For the sake of brevity we refer the interested reader to [Pfe94b] for the actual representation of these proofs in LF.

3.1 Type preservation theorem

Informal arguments about programming languages which contain a variable concept on their own often require auxiliary lemmas which weaken contexts or exchange the order of declarations, or substitute other objects for free variables. In this respect, the informal proof of the type preservation theorem for Mini-ML is not straightforward. Its proof requires that variables in Δ — introduced to make implicit assumptions explicit — are instantiated with well-typed objects, leading to a substitution lemma. When representing the proof in LF [Pfe94b] we can omit this lemma because we implicitly appeal to lemmas about LF [HHP93] and the adequacy of the encodings. For the not entirely trivial proof of the substitution lemma, we refer the reader to [Pfe94b].

Lemma 8 (Substitution) *For all contexts Δ:*
If $\Delta \triangleright e' : \tau'$ and $\Delta, x : \tau' \triangleright e : \tau$ then $\Delta \triangleright e[e'/x] : \tau$.

The operational semantics of Mini-ML preserves types.

Theorem 9 (Type preservation) *For all expressions e, v, all type τ:*
If $\mathcal{D} :: e \hookrightarrow v$ and $\mathcal{P} :: \triangleright e : \tau$ a derivation $\mathcal{Q} :: \triangleright v : \tau$.

Proof: by induction over $\mathcal{D}$:

Case: $\mathcal{D} = \dfrac{}{\mathbf{lam}\ x.\, e \hookrightarrow \mathbf{lam}\ x.\, e}\ \mathsf{evLam}$:

This case is immediate because we can use $\mathcal{P}$ for $\mathcal{Q}$.

Case: $\mathcal{D} = \dfrac{\begin{matrix}\mathcal{D}_1 \\ e_1 \hookrightarrow \mathbf{lam}\ x.\, e_1'\end{matrix} \qquad \begin{matrix}\mathcal{D}_2 \\ e_2 \hookrightarrow v_2\end{matrix} \qquad \begin{matrix}\mathcal{D}_3 \\ e_1'[v_2/x] \hookrightarrow v\end{matrix}}{e_1 \mathbf{@} e_2 \hookrightarrow v}\ \mathsf{evApp}$:

We have $\mathcal{P} :: \triangleright e_1 \mathbf{@} e_2 : \tau$. By inversion, we obtain $\mathcal{P}_1 :: \Delta \triangleright e_1 : \tau_2 \Rightarrow \tau$ and $\mathcal{P}_2 :: \Delta \triangleright e_2 : \tau_2$ for some τ_2. By induction hypothesis on $\mathcal{D}_1$ and $\mathcal{P}_1$ we obtain $\mathcal{Q}_1 :: \triangleright \mathbf{lam}\ x.\, e_1' : \tau_2 \Rightarrow \tau$. Another inversion step gives us that $\mathcal{Q}_1' :: x : \tau_2 \triangleright e_1' : \tau$. A second appeal to the induction hypothesis on $\mathcal{D}_2$ and $\mathcal{P}_2$ yields that $\mathcal{Q}_2 :: \triangleright v_2 : \tau_2$. The application of the substitution lemma gives us a $\mathcal{Q}_2' :: \triangleright e_1'[v_2/x] : \tau$. Finally, applying the induction hypothesis to $\mathcal{D}_3$ and $\mathcal{Q}_2'$ results in the required derivation $\mathcal{Q}_3 :: \triangleright v : \tau$.

Case: $\mathcal{D} = \dfrac{\begin{matrix}\mathcal{D}_1 \\ e[\mathbf{fix}\ x.\, e/x] \hookrightarrow v\end{matrix}}{\mathbf{fix}\ x.\, e \hookrightarrow v}\ \mathsf{evFix}$:

We have $\mathcal{P} :: \triangleright \mathbf{fix}\ x.\, e : \tau$. By inversion we immediately obtain a $\mathcal{P}_1 :: x : \tau \triangleright e : \tau$. Applying the substitution lemma yields $\mathcal{P}_1' :: \triangleright e[\mathbf{fix}\ x.\, e/x] : \tau$. By applying the induction hypothesis to $\mathcal{D}_1$ and $\mathcal{P}_1'$ we obtain the desired derivation of $\mathcal{Q}_1 :: \triangleright v : \tau$.

□

A first useful observation is that this proof is constructive. It implicitly contains a method for constructing a derivation $\mathcal{Q}$ from derivations $\mathcal{D}$ and $\mathcal{P}$. This method constitutes already proof, because of an argument analogous to the following observation: In the setting of natural deduction, the Curry-Howard isomorphism guarantees the realizability of proofs by total functions in the simply-typed λ-calculus and it also guarantees the converse: each total function corresponds to a natural deduction proof.

Similarly, in this setting, each total function f mapping closed derivations of $e \hookrightarrow v$ and $\triangleright e : \tau$ to a derivation of $\triangleright v : \tau$ realizes a proof of the type preservation theorem. The meta logic $\mathcal{M}_2$ (and we believe also $\mathcal{M}_\omega$) is a system of total functions, at least one of them corresponds to this proof. Hence, the type of this total function is (informally)

$$\text{tps} : \text{ev } \mathbf{E}\ \mathbf{V} \Rightarrow \text{of } \mathbf{E}\ \mathbf{T} \Rightarrow \text{of } \mathbf{V}\ \mathbf{T}.$$

Applied to two closed objects representing derivations of 'ev $\mathbf{E}\ \mathbf{V}$' and 'of $\mathbf{E}\ \mathbf{T}$', respectively, the function constructs a closed object M of type 'of $\mathbf{V}\ \mathbf{T}$'. We write $A_1 \Rightarrow A_2$ for the primitive recursive function space from type A_1 to type A_2 to distinguish it from the the parametric function space $A_1 \rightarrow A_2$ provided by LF [DPS96]. In an ML-like informal notation, we expect the proof to have the following form (for the sake of simplicity, we omit implicit variables, and write index variables in bold type):

$$
\begin{array}{l}
\textbf{fun}\ \text{tps ev\_lam}\ P = P \\
\quad \mid \text{tps (ev\_app}\ D_3\ D_2\ D_1)\ (\text{of\_app}\ P_2\ P_1) = \\
\qquad \textbf{let} \\
\qquad\quad \textbf{val}\ (\text{of\_lam}\ Q_1') = \text{tps}\ D_1\ P_1 \\
\qquad\quad \textbf{val}\ Q_2 = \text{tps}\ D_2\ P_1 \\
\qquad\quad \textbf{val}\ Q_3 = \text{tps}\ D_3\ (Q_1'\ \mathbf{V_2}\ Q_2) \\
\qquad \textbf{in} \\
\qquad\quad Q_3 \\
\qquad \textbf{end} \\
\quad \mid \text{tps (ev\_fix}\ D_1)\ (\text{of\_fix}\ P_1) = \\
\qquad \textbf{let} \\
\qquad\quad \textbf{val}\ Q_1 = \text{tps}\ D_1\ (P_1\ \mathbf{E_1}\ (\text{of\_fix}\ P_1)) \\
\qquad \textbf{in} \\
\qquad\quad Q_1 \\
\qquad \textbf{end}
\end{array}
$$

The first argument corresponds to the subject of induction. Appeals to the induction hypothesis are represented in the function above by recursive calls on "smaller" arguments. After selecting the right case, inversion must be performed on the second argument in the 'ev_app' and 'ev_fix' case. In the informal function notation both steps are combined to one pattern-matching step.

The function 'tps' maps closed objects to closed objects. This invariant is preserved during evaluation: For example, in the 'ev_app' case, $\mathbf{V_2}, D_1, D_2, D_3, P_1, P_2$ are closed because 'ev_app $D_3\ D_2\ D_1$' and 'of_app $P_2\ P_1$' are closed. Consequently, Q_1' and Q_2 are closed. The application of Q_1' to $\mathbf{V_2}$ and Q_2 yields that Q_3 is closed.

To show that 'tps' is total, we must ensure three properties: type correctness, progress, and termination. Intuitively, the informal notation of the function is type correct. The function will also terminate, because the recursive calls are always applied to subderviations of the first argument representing the evaluation derivation. The progress property is less obvious. We must show that the patterns cover all possible cases.

There are several places in this function were pattern-matching is used. First, we examine pattern-matching on the represented induction subject of type 'ev $\mathbf{E}$ $\mathbf{V}$'. To ensure that all cases are covered, we examine all of its possible "shapes". This is done by examining each constant in the signature: If its target type (the type of the constant applied to the maximal number of fresh variables) unifies with 'ev $\mathbf{E}$ $\mathbf{V}$' it must be considered as a case for pattern-matching. In the example of 'tps', we obtain three possible "shapes", ensuring that all cases are covered. Note, that this argument relies on the fact that the term representing the induction subject is closed.

$$\begin{array}{ll} \text{ev\_lam} & : \text{ev } (\text{lam } \mathbf{E_1}) \ (\text{lam } \mathbf{E_1}) \\ \text{ev\_app } D_3 \ D_2 \ D_1 & : \text{ev } (\text{app } \mathbf{E_1} \ \mathbf{E_2}) \ \mathbf{V'} \\ \text{ev\_fix } D_1 & : \text{ev } (\text{fix } \mathbf{E_1}) \ \mathbf{V'} \end{array}$$

This idea is inspired by Schroeder-Heister's rule of definitional reflection [SH93b], which also relies on a complete coverage of all possible cases, but only in the realm of a first-order logic without proof terms. A complete coverage of cases must be also guaranteed for the proof editor Alf, a system which is due to Magnusson [Mag95].

As second example, consider pattern-matching on the object representing the typing derivation in the case 'ev_app D_3 D_2 D_1'. To guarantee progress, we must show that all cases due to inversion are covered. Originally, the object representing the typing derivation has type 'of $\mathbf{E}$ $\mathbf{T}$'. Since we are discussing the case for 'ev_app', $\mathbf{E}$ has been instantiated to 'app $\mathbf{E_1}$ $\mathbf{E_2}$'. An inspection of the signature yields, that only three constants yield a "shape" of the type family 'of'

$$\begin{array}{ll} \text{of\_lam } P_1 & : \text{of } (\text{lam } \mathbf{E'_1}) \ (\text{arr } \mathbf{T_1} \ \mathbf{T_2}) \\ \text{of\_app } P_2 \ P_1 & : \text{ev } (\text{app } \mathbf{E'_1} \ \mathbf{E'_2}) \ \mathbf{T_1} \\ \text{of\_fix } P_1 & : \text{of } (\text{fix } \mathbf{E'_1}) \ \mathbf{T'} \end{array}$$

and only the type of 'of_app P_2 P_1' unifies with 'of (app $\mathbf{E_1}$ $\mathbf{E_2}$) $\mathbf{T}$'. Hence all possible cases are covered for this occurrence of pattern-matching in 'tps'.

In other proof assistants, as for example early versions of Coq [C$^+$95], proofs by induction have always the same rigid form determined by the induction principles. All cases for inversion must be considered, even the impossible ones. That they are indeed impossible must then be shown explicitly.

The unification problem is higher-order, and hence is in general not decidable. But in practice, most of the unification problems lie in the pattern fragment described first by Miller [Mil89]. A sound and complete unification algorithm for the extended fragment by dependent types goes back to Pfenning [Pfe91]. If it exists, the algorithm finds a most general unifier mgu (U) (where U is a unification problem, see Snyder [Sny91]). For the meta-theoretical discussion, it is important to have only one most general unifier, otherwise one has to consider possibly infinite complete sets of unifiers (in the sense of Huet [Hue75]), as noted by Eriksson [Eri91].

Using the same technique, it can be also verified that inversion on the result of the first recursive call is uniquely determined, as is the inversion on the typing derivation in the 'ev_fix' case. Hence all occurrences of pattern-matching in 'tps' cover all possible cases, and hence its application to well-typed and closed LF-objects is guaranteed to make progress. Therefore it is a total function, representing a proof of the type preservation theorem.

Another possibility developed in the past, is to represent the total function as relation in LF. The relation is encoded by the type family

$$\text{tps} : \text{ev}\ \mathbf{E}\ \mathbf{V} \rightarrow \text{of}\ \mathbf{E}\ \mathbf{T} \rightarrow \text{of}\ \mathbf{V}\ \mathbf{T} \rightarrow \text{type}.$$

For a definition of this relation in LF, the interested reader is invited to consult [Pfe94b]. There is no guarantee that these representations are proofs because for example cases could have been omitted, but they can be executed as logic programs.

3.2 Deduction theorem

We return to the Hilbert calculus to prove the deduction theorem. The algorithm realizing the proof of this theorem is important for translating pure functional programs into combinators. It is called bracket abstraction. The deduction theorem shows that abstracting over formulas in the context is an admissible rule:

Theorem 10 (Deduction) *If* $\begin{array}{c}\overline{\vdash A}\,u \\ \mathcal{D} \\ \vdash B\end{array}$ *(which reads as "under the assumption that $\vdash A$ holds, $\mathcal{D}$ is a derivation for $\vdash B$"), then $\vdash A \supset B$ is provable.*

Proof: by induction over $\mathcal{D}$.

Case: $\mathcal{D} = \dfrac{}{\vdash A}\,u$: We can construct a derivation with $(\mathsf{MP}\ (\mathsf{MP}\ \mathsf{S}\ \mathsf{K})\ \mathsf{K})$.

Case: $\mathcal{D} = \dfrac{}{\vdash A' \supset B' \supset A'}\,\mathsf{K}$: We can construct a derivation with $(\mathsf{MP}\ \mathsf{K}\ \mathsf{K})$.

Case: $\mathcal{D} = \dfrac{}{\vdash (A' \supset B' \supset C') \supset (A' \supset B') \supset (A' \supset C')}\,\mathsf{S}$:

We can construct a derivation with $(\mathsf{MP}\ \mathsf{K}\ \mathsf{S})$.

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\overline{\vdash A}\,u \\ \mathcal{D}_1 \\ \vdash A' \supset B\end{array} \qquad \begin{array}{c}\overline{\vdash A}\,u \\ \mathcal{D}_2 \\ \vdash A'\end{array}}{\vdash B}\,\mathsf{MP}$:

By applying the induction hypothesis to $\mathcal{D}_1$ and $\mathcal{D}_2$ respectively, we obtain two derivations:

$$\begin{array}{c}\mathcal{P}_1 \\ \vdash A \supset (A' \supset B)\end{array} \qquad \begin{array}{c}\mathcal{P}_2 \\ \vdash A \supset A'\end{array}$$

We can hence construct a derivation with $(\mathsf{MP}\ (\mathsf{MP}\ \mathsf{S}\ \mathcal{P}_1)\ \mathcal{P}_2)$.

□

The proof is again a total function mapping the representation of the hypothetical derivation $\mathcal{D}$ into a derivation of $\vdash A \supset B$. Since the given derivation $\mathcal{D}$ is represented in LF as parametric function

$$\ulcorner \begin{array}{c} \overline{\vdash A}\, u \\ \mathcal{D} \\ \vdash B \end{array} \urcorner = \lambda u\!:\!\ulcorner\vdash A\urcorner.\, \ulcorner\mathcal{D}\urcorner : \ulcorner\vdash A\urcorner \to \ulcorner\vdash B\urcorner.$$

the type of the total function representing the deduction theorem is

$$\mathrm{abs} : (\mathrm{pf}\ \mathbf{A} \to \mathrm{pf}\ \mathbf{B}) \Rightarrow \mathrm{pf}\ (\mathbf{A}\ \mathrm{imp}\ \mathbf{B}).$$

The proof proceeds by induction over the hypothetical derivation $\mathcal{D}$. It is crucial that the representation of $\mathcal{D}$ is a parametric function because all of its possible "shapes" must be analyzed. In contrast, the primitive recursive function space 'pf $\mathbf{A} \Rightarrow$ pf $\mathbf{B}$' contains too many functions, which makes a case analysis of the "shape" of its elements impossible [DPS96].

Applying 'abs' to a closed parametric function of type 'pf $\mathbf{A} \to$ pf $\mathbf{B}$' yields a derivation of type 'pf ($\mathbf{A}$ imp $\mathbf{B}$)'. In our informal ML-like language the proof of the deduction theorem can be written as:

$$\begin{array}{l}
\textbf{fun}\ \mathrm{abs}\ (\lambda u\!:\!\mathrm{pf}\ \mathbf{A}.\, u) = \mathrm{MP}\ (\mathrm{MP}\ \mathrm{S}\ \mathrm{K})\ \mathrm{K} \\
\quad \mid \mathrm{abs}\ (\lambda u\!:\!\mathrm{pf}\ \mathbf{A}.\, \mathrm{K}) = \mathrm{MP}\ \mathrm{K}\ \mathrm{K} \\
\quad \mid \mathrm{abs}\ (\lambda u\!:\!\mathrm{pf}\ \mathbf{A}.\, \mathrm{S}) = \mathrm{MP}\ \mathrm{K}\ \mathrm{S} \\
\quad \mid \mathrm{abs}\ (\lambda u\!:\!\mathrm{pf}\ \mathbf{A}.\, \mathrm{MP}\ (D_1\ u)\ (D_2\ u)) = \\
\qquad \textbf{let} \\
\qquad\quad \textbf{val}\ P_1 = \mathrm{abs}\ (\lambda u\!:\!\mathrm{pf}\ \mathbf{A}.\, D_1\ u) \\
\qquad\quad \textbf{val}\ P_2 = \mathrm{abs}\ (\lambda u\!:\!\mathrm{pf}\ \mathbf{A}.\, D_2\ u) \\
\qquad \textbf{in} \\
\qquad\quad \mathrm{MP}\ (\mathrm{MP}\ \mathrm{S}\ P_1)\ P_2 \\
\qquad \textbf{end}
\end{array}$$

To see that this function is total, consider the application of 'abs' to a closed parametric function of type 'pf $\mathbf{A} \to$ pf $\mathbf{B}$'. It is type correct and it terminates: the first three cases simply return an LF-object and the recursive calls in the fourth case terminate because 'λu:pf $\mathbf{A}$. $D_1\ u$' and 'λu : pf $\mathbf{A}$. $D_2\ u$' are "smaller" than 'λu : pf $\mathbf{A}$. MP ($D_1\ u$) ($D_2\ u$)'. To see this, compare the bodies under the λ-abstraction: '$D_1\ u$' and '$D_2\ u$' are subterms of 'MP ($D_1\ u$) ($D_2\ u$)'. A precise definition of the subterm relation is given in [RP96].

To show progress, we must ensure that the four given patterns cover all possible cases. Without loss of generality, the parametric function is canonical [HHP93]. That means, it has the form λu:pf $\mathbf{A}$. M, where M is an object of atomic type. An inspection of the local context u : pf $\mathbf{A}$ and the signature yields, that there are four possible "shapes" of parametric functions, whose types unify with 'pf $\mathbf{A} \to$ pf $\mathbf{B}$'.

$$\begin{array}{ll}
\lambda u\!:\!\mathrm{pf}\ \mathbf{A}.\, u & : \mathrm{pf}\ \mathbf{A} \to \mathrm{pf}\ \mathbf{A} \\
\lambda u\!:\!\mathrm{pf}\ \mathbf{A}.\, \mathrm{K} & : \mathrm{pf}\ \mathbf{A} \to \mathrm{pf}\ (\mathbf{A}'\ \mathrm{imp}\ \mathbf{B}'\ \mathrm{imp}\ \mathbf{A}') \\
\lambda u\!:\!\mathrm{pf}\ \mathbf{A}.\, \mathrm{S} & : \mathrm{pf}\ \mathbf{A} \\
 & \quad \to \mathrm{pf}\ ((\mathbf{A}'\ \mathrm{imp}\ \mathbf{B}'\ \mathrm{imp}\ \mathbf{C}')\ \mathrm{imp}\ (\mathbf{A}'\ \mathrm{imp}\ \mathbf{B}')\ \mathrm{imp}\ (\mathbf{A}'\ \mathrm{imp}\ \mathbf{C}')) \\
\lambda u\!:\!\mathrm{pf}\ \mathbf{A}.\, \mathrm{MP}\ (D_1\ u)\ (D_2\ u) & : \mathrm{pf}\ \mathbf{A} \to \mathrm{pf}\ \mathbf{B}'
\end{array}$$

This verifies that 'abs' makes always progress when applied to closed and well-typed parametric functions. Hence 'abs' is a total function, realizing the proof of the deduction theorem.

As alternative to the presentation of the function in an ML-like formalism, it can also been encoded in LF as a relation [Pfe94b] by the following type family:

$$\text{abs} : (\text{pf}\ \mathbf{A} \to \text{pf}\ \mathbf{B}) \to \text{pf}\ (\mathbf{A}\ \text{imp}\ \mathbf{B}) \to \text{type}$$

We refer the interested reader to [Pfe94b] for the signature representing the proof of the deduction theorem.

3.3 Summary

Higher-order representation techniques allow elegant and adequate representations of formal languages, theorems, and proofs in LF. But the representations are no longer inductive, which means that standard techniques for reasoning by induction do not apply. The proofs we have encountered in this section are constructive and can be realized by total functions. In the next section we define a meta logic $\mathcal{M}_\omega$, in which we can represent these realizing functions formally.

4 The Meta Logic $\mathcal{M}_\omega$

The purpose of the meta logic $\mathcal{M}_\omega$ is to provide a formal framework for the representation of total functions realizing informal proofs (as for example 'tps' and 'abs') with the goal to automate their derivation. $\mathcal{M}_\omega$ is a first-order logic where formulas are used to represent theorems and quantifiers range over LF-objects generated by a given but fixed LF-signature.

In general, strong induction principles do not exist for the types we intend to reason about. Hence our meta logic $\mathcal{M}_\omega$ contains a different mechanism for inductive reasoning: We propose to endow the meta logic with proof terms, each of which is a *total* function (under a well-defined operational semantics). The property that they are *total* guarantees that each proof term represents an informal proof. Inductive arguments are represented in $\mathcal{M}_\omega$ by well-founded recursion and case analysis.

On the one hand, $\mathcal{M}_\omega$ is more general than needed, because its formulation allows arbitrary alternation of universal and existential quantifiers prefixing a theorem (Π_ω-formulas). This is indicated by the index ω in $\mathcal{M}_\omega$. For the purpose of automated proof search, we restrict the formulas of $\mathcal{M}_\omega$ to Π_2-formulas which allow only one alternation of quantifiers. Π_2-formulas are prefixed by a block of universal, followed by a block of existentials quantifiers. In the technical report [Sch97], we define this system $\mathcal{M}_2$ formally, embed it into $\mathcal{M}_\omega$, and prove that all proof terms are total. Since the design of $\mathcal{M}_2$ is very specialized and it is very likely to change fundamentally in future work, we present in this proposal only $\mathcal{M}_\omega$.

On the other hand, $\mathcal{M}_\omega$ is less general than desired, because it cannot be used to realize proofs which do introduce new assumptions into the proof process. It is the main objective of my thesis, to generalize $\mathcal{M}_\omega$ to a system which can.

The formal system of $\mathcal{M}_\omega$ in its full generality is rather complex, its formal development quite tedious. We therefore introduce $\mathcal{M}_\omega$ in four steps. We introduce in Section 4.1 an intuitionistic logic for LF. In Section 4.2 we endow it with proof terms and strengthen the system to allow the representation of inductive arguments by recursion in Section 4.3 and by case analysis in Section 4.4. A detailed meta-theoretic investigation of $\mathcal{M}_\omega$ lies outside the scope of this proposal. We hence refer the interested reader to the more detailed forthcoming technical report [Sch97].

$$\frac{}{\Gamma; \Delta \vdash \top}\,\mathsf{axunit} \qquad \frac{}{\Gamma; \Delta_1, F, \Delta_2 \vdash F}\,\mathsf{axvar}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma; \Delta_1, \forall x\!:\!A.\,F, \Delta_2, F[M/x] \vdash G}{\Gamma; \Delta_1, \forall x\!:\!A.\,F, \Delta_2 \vdash G}\,\forall\mathsf{L} \qquad \frac{\Gamma, u : A; \Delta \vdash F[u/x]}{\Gamma; \Delta \vdash \forall x\!:\!A.\,F}\,\forall\mathsf{R}^*$$

$$\frac{\Gamma, u : A; \Delta_1, \exists x\!:\!A.\,F, \Delta_2, F[u/x] \vdash G}{\Gamma; \Delta_1, \exists x\!:\!A.\,F, \Delta_2 \vdash G}\,\exists\mathsf{L}^* \qquad \frac{\Gamma \vdash M : A \qquad \Gamma; \Delta \vdash F[M/x]}{\Gamma; \Delta \vdash \exists x\!:\!A.\,F}\,\exists\mathsf{R}$$

$$\frac{\Gamma; \Delta_1, F_1 \wedge F_2, \Delta_2, F_1, F_2 \vdash G}{\Gamma; \Delta_1, F_1 \wedge F_2, \Delta_2 \vdash G}\,\wedge\mathsf{L} \qquad \frac{\Gamma; \Delta \vdash F_1 \qquad \Gamma; \Delta \vdash F_2}{\Gamma; \Delta \vdash F_1 \wedge F_2}\,\wedge\mathsf{R}$$

* Eigenvariable condition: $\vdash \Gamma, u : A\ \mathrm{ctx}$

Figure 1: Intuitionistic logic for LF

4.1 Intuitionistic Sequent Calculus for LF

$\mathcal{M}_\omega$ is a first-order intuitionistic logic. Formulas express theorems about LF-objects, which represent derivations in some deductive system as for example the operational semantics of Mini-ML or its type system. The language of LF-objects hence forms a term algebra which is generated from a given and fixed signature Σ. We define provability in $\mathcal{M}_\omega$ using a sequent calculus formulation [Gen35].

The language of formulas of this logic is very restricted. We only allow universal and existential quantification, and conjunction. The only propositional constant is truth. The universal quantifier $\forall x\!:\!A.\,F$ and the existential quantifier $\exists x\!:\!A.\,F$ range over *closed* LF-objects only. The reason to assume these objects to be closed stems from the fact that we intend to analyze the different "shapes" the LF-variable x can take when instantiated. We must guarantee complete analysis of all cases, otherwise proof terms (which we define in Section 4.2) are *not* necessarily total.

$$\begin{array}{lcll} \text{Types:} & A & ::= & \text{LF types generated } \Sigma \\ \text{Terms:} & M & ::= & \text{LF objects generated by } \Sigma \\ \text{Formulas:} & F & ::= & \top \mid \forall x\!:\!A.\,F \mid \exists x\!:\!A.\,F \mid F_1 \wedge F_2 \end{array}$$

The judgment for provability for this sequent calculus is $\Gamma; \Delta \vdash F$, where Δ stands for a set of assumptions, and Γ makes all Eigenvariables explicit. The judgment is actually indexed by a fixed LF-signature Σ, but for the sake of brevity, we omit this index. Because the Eigenvariables are typed LF-variables, Γ is a standard LF-context.

$$\text{Assumptions:} \quad \Delta \quad ::= \quad \cdot \mid \Delta, F$$

The rules for the judgement are defined in Figure 1. For every formula constructor there is a left and a right rule. The formulation of the calculus incorporates the structural rules: weakening is implicit in the axioms axunit and axvar, contraction and exchange are implicit in the left rules $\forall$L, $\exists$L, and $\wedge$L.

Example 11 (Formalization of the type preservation theorem) The "type" of the function realizing the proof of the type preservation theorem (Theorem 9) is informally 'ev **E V** $\Rightarrow$

$$\frac{}{\Gamma;\Delta \vdash \langle\rangle \in \top}\,\mathsf{axunit} \qquad \frac{}{\Gamma;\Delta_1,\mathbf{x}\in F,\Delta_2 \vdash \mathbf{x} \in F}\,\mathsf{axvar}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma;\Delta_1,\mathbf{y}\in \forall x\!:\!A.\,F,\Delta_2,\mathbf{z}\in F[M/x] \vdash P \in G}{\Gamma;\Delta_1,\mathbf{y}\in\forall x\!:\!A.\,F,\Delta_2 \vdash \text{let } \mathbf{z}=\mathbf{y}\ M \text{ in } P \in G}\,\forall\mathsf{L} \qquad \frac{\Gamma,u:A;\Delta \vdash P \in F[u/x]}{\Gamma;\Delta \vdash \Lambda x\!:\!A.\,P \in \forall x\!:\!A.\,F}\,\forall\mathsf{R}^*$$

$$\frac{\Gamma,u:A;\Delta_1,\mathbf{y}\in\exists x\!:\!A.\,F,\Delta_2,\mathbf{z}\in F[u/x] \vdash P \in G}{\Gamma;\Delta_1,\mathbf{y}\in\exists x\!:\!A.\,F,\Delta_2 \vdash \text{split } \mathbf{y} \text{ as } [x,\mathbf{z}] \text{ in } P \in G}\,\exists\mathsf{L}^* \qquad \frac{\Gamma \vdash M : A \qquad \Gamma;\Delta \vdash P \in F[M/x]}{\Gamma;\Delta \vdash [M,P] \in \exists x\!:\!A.\,F}\,\exists\mathsf{R}$$

$$\frac{\Gamma;\Delta_1,\mathbf{x}\in F_1\wedge F_2,\Delta_2,\mathbf{y}\in F_1,\mathbf{z}\in F_2 \vdash P \in G}{\Gamma;\Delta_1,\mathbf{x}\in F_1\wedge F_2,\Delta_2 \vdash \text{split } \mathbf{x} \text{ as } \langle\mathbf{y},\mathbf{z}\rangle \text{ in } P \in G}\,\wedge\mathsf{L} \qquad \frac{\Gamma;\Delta \vdash P_1 \in F_1 \qquad \Gamma;\Delta \vdash P_2 \in F_2}{\Gamma;\Delta \vdash \langle P_1,P_2\rangle \in F_1\wedge F_2}\,\wedge\mathsf{R}$$

* Eigenvariable condition: $\vdash \Gamma, u : A$ ctx

Figure 2: Intuitionistic logic for LF with proof terms

of $\mathbf{E}\ \mathbf{T} \Rightarrow$ of $\mathbf{V}\ \mathbf{T}$'. Applied to two closed objects representing derivations of 'ev $\mathbf{E}\ \mathbf{V}$' and 'of $\mathbf{E}\ \mathbf{T}$', respectively, the function constructs a closed object M of type 'of $\mathbf{V}\ \mathbf{T}$'. In $\mathcal{M}_\omega$, we write $\forall D : \text{ev } \mathbf{E}\ \mathbf{V}.\,\forall P : \text{of } \mathbf{E}\ \mathbf{T}.\,\exists Q : \text{of } \mathbf{V}\ \mathbf{T}.\,\top$ for this theorem (for the sake of simplicity, we omit implicit variables, and write index variables in bold type: The verbose version is as follows: $\forall E\!:\!\text{exp}.\,\forall V\!:\!\text{exp}.\,\forall T\!:\!\text{tp}.\,\forall D\!:\!\text{ev } E\ V.\,\forall P\!:\!\text{of } E\ T.\,\exists Q\!:\!\text{of } V\ T.\,\top$).

Example 12 (Formalization of the deduction theorem) The "type" of the function realizing the proof of the deduction theorem (Theorem 10) is informally written as '(pf $\mathbf{A} \to$ pf $\mathbf{B}$) $\Rightarrow$ pf ($\mathbf{A}$ imp $\mathbf{B}$)'. Applied to a closed parametric function of type 'pf $\mathbf{A} \to$ pf $\mathbf{B}$' it constructs a derivation of type 'pf ($\mathbf{A}$ imp $\mathbf{B}$)'. In $\mathcal{M}_\omega$, we write $\forall D : \text{pf } \mathbf{A} \to \text{pf } \mathbf{B}.\,\exists P : \text{pf } \mathbf{A} \text{ imp } \mathbf{B}.\,\top$ for this theorem.

4.2 Intuitionistic Sequent Calculus for LF with proof terms

$\mathcal{M}_\omega$ extends the calculus in Figure 1 by proof terms which are constructs witnessing derivations. Endowing a calculus with proof terms means to extend the provability judgment by a new syntactical construct. In $\mathcal{M}_\omega$ we denote proof terms with P and extend the judgment $\Gamma;\Delta \vdash F$ to $\Gamma;\Delta \vdash P \in F$. Usually, the proof term in the conclusion of an inference rule is constructed in terms of the proof terms of the premiss.

The introduction of proof terms makes it necessary to name assumptions: consider for example the rule axvar. The proof term associated with this rule is simply the name of F. We use bold lowercase variables as assumption names (for example $\mathbf{x}$) to distinguish them from LF-variable names which are written in the standard way (for example x).

$$\text{Assumptions:} \quad \Delta \ ::= \ \cdot \mid \Delta, \mathbf{x} \in F$$

The result of endowing all inference rules from Figure 1 with proof terms is given in Figure 2. Besides axvar, every rule introduces a new proof term constructor.

$$\begin{array}{lrcl} \text{Proof terms:} & P & ::= & \langle\rangle \mid \mathbf{x} \mid \Lambda x\!:\!A.\,P \mid \text{let } \mathbf{x} = \mathbf{y}\ M \text{ in } P \\ & & & \mid [M,P] \mid \text{split } \mathbf{x} \text{ as } [x_1,\mathbf{x}_2] \text{ in } P \\ & & & \mid \langle P_1,P_2\rangle \mid \text{split } \mathbf{x} \text{ as } \langle\mathbf{x}_1,\mathbf{x}_2\rangle \text{ in } P \end{array}$$

We can easily argue that all proof terms correspond to total functions under the standard operational semantics. Since evaluating a proof term triggers the evaluation only of its subterms, termination is guaranteed and we can conclude that all proof terms are total.

Example 13 (Using the deduction theorem) Assume we have a proof of the deduction theorem and are supposed to prove the following property:

Let a be a formula. Show that $\vdash a \supset a$ is derivable in the Hilbert calculus.

There are two possible ways to tackle this problem: Either one constructs the derivation of 'pf (a imp a)' by hand, or one appeals to the deduction theorem with the representation of

$$\ulcorner \begin{array}{c} \overline{\vdash a}\,u \\ u \\ \vdash a \end{array} \urcorner = \lambda u\!:\!\mathrm{pf}\,a.\,u : \mathrm{pf}\,a \to \mathrm{pf}\,a$$

as argument. It is easy to see that

$$\cdot; \mathbf{x} \in \forall A\!:\!\mathrm{o}.\,\forall B\!:\!\mathrm{o}.\,\forall D\!:\!\mathrm{pf}\,A \to \mathrm{pf}\,B.\,\exists E\!:\!\mathrm{pf}\,A \;\mathrm{imp}\; B.\,\top \vdash P \in \forall a\!:\!\mathrm{o}.\,\exists x\!:\!\mathrm{pf}\,(a \;\mathrm{imp}\; a).\,\top$$

is derivable in $\mathcal{M}_\omega$. The proof term has the following form:

$$\begin{array}{rl} P = & \Lambda a\!:\!\mathrm{o}. \\ & \quad \mathrm{let}\ \mathbf{x}_1 = \mathbf{x}\ a\ \mathrm{in} \\ & \quad\quad \mathrm{let}\ \mathbf{x}_2 = \mathbf{x}_1\ a\ \mathrm{in} \\ & \quad\quad\quad \mathrm{let}\ \mathbf{x}_3 = \mathbf{x}_2\ (\lambda u\!:\!\mathrm{pf}\,a.\,u)\ \mathrm{in} \\ & \quad\quad\quad\quad \mathbf{x}_3 \end{array}$$

Applying P to some formula a will evaluate to $[M, \langle\rangle]$, a proof term for $\exists x\!:\!\mathrm{pf}\,(a \;\mathrm{imp}\; a).\,\top$. The LF-object M is hence a representation of the desired deduction $\vdash a \supset a$ in the Hilbert calculus.

4.3 Adding Recursion

$\mathcal{M}_\omega$ as defined in Figure 2 is a standard intuitionistic logic with proof terms. We cannot extend the rules by induction principles because they do not necessarily exist, but we can extend the proof term calculus by recursion and case analysis and hence recovering the power of induction principles. In this section we address the extension of $\mathcal{M}_\omega$ by a general recursion operator.

The recursion operator $\mu\mathbf{x}\!:\!F.\,P$ is a fixed point operator which makes the formula F which is to be shown an assumption, and names it $\mathbf{x}$.

$$\text{Proof terms:} \quad P \quad ::= \quad ... \mid \mu\mathbf{x}\!:\!F.\,P$$

The inference rule introduces the recursion operator on the level of proof terms, very similar to the typing rule 'of_fix' in Mini-ML in Section 2. It is obvious, that a general recursion rule without any side condition violates the totality requirement.

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

We use a lexicographical extension of the termination ordering introduced by Rohwedder and Pfenning [RP96] to ensure that the sequence of recursive calls terminates. In the technical report [Sch97] we precisely characterize a termination ordering and show termination for the $\mathcal{M}_2$-fragment of $\mathcal{M}_\omega$. We suspect that this argument can be extended to the full system $\mathcal{M}_\omega$ which we leave for future work.

4.4 Adding Case Analysis

The context of all Eigenvariables Γ in the judgment $\Gamma; \Delta \vdash P \in G$ represents all free LF-variables which might occur in the proof term P. Because of the assumption that proof terms are only applied to closed LF-objects, all variables in Γ are bound to closed LF-objects during evaluation. It is hence possible to distinguish cases according to the shapes of these objects. Recall, that without this assumption case analysis is not an admissible rule, because a complete coverage of all cases and hence progress cannot be guaranteed which means that proof terms need not to be total.

Because of higher-order representation techniques, as we have seen in Example 12, it is necessary to distinguish cases over functions. These functions must be parametric, because their "shapes" must be analyzed.

The formal development of the rules which define case analysis for $\mathcal{M}_\omega$ is very complicated and tedious. The rules are engineered in such a way, that impossible cases are automatically excluded. Some information about a derivation — in its broadest sense — is encoded in dependent types, which makes unification as used for the examples in Section 3 succeed and fail for the appropriate cases. We attribute the high degree of automation of Twelf to this property. The attentive reader should be warned, that the remainder of this section is very technical (even though we abstracted in this presentation already from a lot of details). A detailed presentation of this material can be found in the forthcoming technical report [Sch97].

We start the discussion with the introduction of some simplifying notation to make the rules we are introducing later in this section readable: Consider the term 'ev_app $D_3\ D_2\ D_1$' in the 'ev_app' case in the proof of the type preservation theorem in Section 3.1. Using a context $\Gamma_{\text{ev\_app}} = D_3 : \text{ev } (\mathbf{E_1'}\ \mathbf{V_2})\ \mathbf{V}', D_2 : \text{ev } \mathbf{E_2}\ \mathbf{V_2}, D_1 : \text{ev } \mathbf{E_1}\ (\text{lam } \mathbf{E_1'})$, we abbreviate it with 'ev_app $\Gamma_{\text{ev\_app}}$': the constant 'ev_app' is applied to a context, an application which is formally defined as follows:

Definition 14 (Contexts as arguments)

$$\begin{array}{rcl} M\ \cdot & = & M \\ M\ (\Gamma, x : A) & = & (M\ \Gamma)\ x \end{array}$$

Similarly, we introduce an abbreviating notation for Π- and λ-abstractions. The argument type of 'abs' in Section 3.2 is a parametric function of type 'pf $\mathbf{A} \to$ pf $\mathbf{B}$'. In general, the type of the case subject can be an arbitrary parametric function type. Since in LF each function has a η-long and β-normal form, we can simplify the notation for the general case by allowing Π- and λ-abstractions over contexts: Let $\Gamma_D = u : \text{pf } \mathbf{A}$. We write $\Pi\Gamma_D.\, \text{pf } \mathbf{B}$ for pf $\mathbf{A} \to$ pf $\mathbf{B}$ and $\lambda\Gamma_D.\, u, \lambda\Gamma_D.\, \text{K}, \ldots$ for the canonical forms.

Definition 15 (Abstraction contexts)

$$\begin{array}{rclcrcl} \lambda\cdot.\, M & = & M & \quad & \Pi\cdot.\, A & = & A \\ \lambda(\Gamma, x : A).\, M & = & \lambda\Gamma.\, (\lambda x{:}A.\, M) & \quad & \Pi(\Gamma, x : A').\, A & = & \Pi\Gamma.\, (\Pi x{:}A'.\, A) \end{array}$$

The formal notation for patterns is very similar to the informal notation used in Section 3. Consider pattern-matching over objects of type 'ev $\mathbf{E}$ $\mathbf{V}$' in the type preservation example in Section 3.1. For the meta-theoretic investigation, we must record with the pattern a context Γ' representing all index variables of the type 'ev $\mathbf{E}$ $\mathbf{V}$' after unification (the variables which will be instantiated by pattern-matching in the operational semantics) and a subcontext[1] Γ'' wrt. Γ' representing all other *non*-index variables (which will not be instantiated by pattern-matching). In a simply-typed setting, we need not to keep track of variables in Γ'', but in a dependently-typed setting we must: index-variables used in the types of variables in Γ'' might be instantiated as we demonstrate below using the proof of the type preservation theorem.

This leads to the syntactic form of a pattern $\Gamma'; \Gamma'' \triangleright M$, where M is the part of the pattern which is matched against during execution. The form of M is determined, either by some constant from the LF-signature Σ or by some parameter from the local context (in case of functional case subjects!).

$$\text{Patterns:} \quad R \quad ::= \quad \Gamma'; \Gamma'' \triangleright \lambda\Gamma_1.\, c\; \Gamma_2 \mid \Gamma'; \Gamma'' \triangleright \lambda\Gamma_1.\, x\; \Gamma_2$$

We motivate the notion of patterns using the examples from Section 3. Consider again the formalization of the proof of the type preservation theorem from Section 3.1. The main case distinction leads to three patterns:

Γ'	$;\Gamma''$	$\triangleright M$
E_1 : exp $\to$ exp	; P : of (lam E_1) $\mathbf{T}$	$\triangleright$ ev_lam
D_3 : ev ($\mathbf{E_1'}$ $\mathbf{V_2}$) $\mathbf{V'}$, D_2 : ev $\mathbf{E_2}$ $\mathbf{V_2}$, D_1 : ev $\mathbf{E_1}$ (lam $\mathbf{E_1'}$)	; P : of (app E_1 E_2) $\mathbf{T}$	$\triangleright$ ev_app D_3 D_2 D_1
D_1 : ev ($\mathbf{E_1}$ (fix $\mathbf{E_1}$)) $\mathbf{V'}$	; P : of (fix E_1) $\mathbf{T}$	$\triangleright$ ev_fix D_1

Bold variable names in the left context are index variables implicitly declared in the left context, bold variable names in the right context are correspondingly index variables implicitly declared in the right context. All declared variables and all index variables in the left context *will be* instantiated by pattern-matching (emphasized by the grey background), all declared variables and all index variables in the right context *will not be* instantiated.

Consider the second case 'ev_app D_3 D_2 D_1': intuitively, D_3, D_2, and D_1 represent subderivations of a derivation ending in the evApp-rule. After pattern-matching D_3, D_2, D_1, and all index variables in the left context (all variable names on a grey background) are bound to closed objects. All other variables in the right context, in this example P and the index variable $\mathbf{T}$, are still bound to the same closed objects after pattern-matching as they were before. But due to case analysis more information about these objects is available: Intuitively, before case analysis P represents a derivation of $\Delta \triangleright e : \tau$, after case analysis e is determined, and hence it is a derivation of $\Delta \triangleright e_1 @ e_2 : \tau$. This additional information is recorded in the type P: 'of E $\mathbf{T}$' is refined to 'of (app E_1 E_2) $\mathbf{T}$', where E_1 and E_2 are still unknowns, but they will be instantiated during evaluation by pattern-matching.

In the proof of Theorem 9 we applied then inversion to the derivation of $\Delta \triangleright e_1 @ e_2 : \tau$ which must end in the rule ofApp. We obtained two subderivations $\Delta \triangleright e_1 : \tau_2 \Rightarrow \tau$ and $\Delta \triangleright e_2 : \tau_2$.

[1] A context Γ_1, Γ_2 always satisfies $\vdash \Gamma_1, \Gamma_2$ ctx. But Γ_2 alone is not a context. We call Γ_2 a subcontext wrt. Γ_1, satisfying $\Gamma_1 \vdash \Gamma_2$ subctx. The rules for this judgment are omitted.

Formally, inversion is represented as a case analysis of the possible "shapes" of closed objects bound to P. It yields, that there is only one possibility: '$P = \text{of\_app}\ P_2\ P_1$', where P_1 represents the first subderivation, and P_2 represents the second. As above the grey backgrounds mark variables which will be instantiated by pattern-matching during execution.

Γ'	$;\Gamma''$	$\triangleright M$
	$D_3 : \text{ev}\ (\mathbf{E_1'}\ \mathbf{V_2})\ \mathbf{V}',$	
$P_2 : \text{of}\ \mathbf{E_2}\ \mathbf{T_2},$	$D_2 : \text{ev}\ E_2\ \mathbf{V_2},$	
$P_1 : \text{of}\ \mathbf{E_1}\ (\text{arr}\ \mathbf{T_2}\ \mathbf{T_1})$	$; D_1 : \text{ev}\ E_1\ (\text{lam}\ \mathbf{E_1'})$	$\triangleright \text{of\_app}\ P_2\ P_1$

As a last example, we return to the proof of the deduction theorem (Theorem 10) in Section 3.2. Distinction by cases has led us to consider four cases: Intuitively, each of the patterns corresponds to a case in the proof.

Γ'	$;\Gamma'' \triangleright M$
$A : \text{o}$	$;\cdot\ \triangleright \lambda u{:}\text{pf}\ A\,.\,u$
$A : \text{o},\ A' : \text{o},\ B' : \text{o}$	$;\cdot\ \triangleright \lambda u{:}\text{pf}\ A\,.\,\text{K}$
$A : \text{o},\ A' : \text{o},\ B' : \text{o},\ C' : \text{o}$	$;\cdot\ \triangleright \lambda u{:}\text{pf}\ A\,.\,\text{S}$
$D_1 : \text{pf}\ \mathbf{A} \to \text{pf}\ (\mathbf{A'}\ \text{imp}\ \mathbf{B'}),$ $D_2 : \text{pf}\ \mathbf{A} \to \text{pf}\ \mathbf{A'}$	$;\cdot\ \triangleright \lambda u{:}\text{pf}\ A\,.\,\text{MP}\ (D_1\ u)\ (D_2\ u)$

Having introduced all necessary syntactical constructs, we can now complete the definition of the language of $\mathcal{M}_\omega$ by introducing a 'case' proof term for pattern-matching. 'Case' proof terms are defined with respect to a list of cases Ω, each case $R \mapsto P$ represents what has to happen if R matches the case subject.

$$\begin{array}{llll} \text{Proof terms:} & P & ::= & \ldots \mid \text{case}\ x\ \text{of}\ \Omega \\ \text{Cases:} & \Omega & ::= & \cdot \mid \Omega, R \mapsto P \end{array}$$

We start now with the discussion of the typing rules for 'case' and Ω which requires a new typing judgment. In this proposal we can only try to convey the idea behind these rules, since a detailed discussion of the rules is beyond the scope of this proposal. The interested reader is invited to study their presentation in [Sch97].

Let Γ be the usual LF-context, representing Eigenvariables, Δ be a set of assumptions, and x the subject of case. x must be defined in Γ: $\Gamma(x) = A$. During evaluation it will be bound to a closed term. Let R be a pattern in Ω which matches this term. R's definition contains a context and a subcontext: matching against R means to instantiate all variables from the context, but none from the subcontext.

To derive the context and the subcontext of R formally, we must characterize the context of index variables of A. Because of deep theoretical properties of LF [HHP93] this is indeed possible: the context Γ can be permuted to a context $\Gamma' = \Gamma_1, x : A, \Gamma_2$ where Γ_1 contains all index variables of A. Γ' is called *dependency context* of Γ with respect to $x : A$.

Definition 16 (Dependency context) $\Gamma' = \Gamma_1, x : A, \Gamma_2$ *is a dependency context of* Γ *with respect to* $x : A$, *iff* Γ' *is a permutation of* Γ, $\vdash \Gamma'$ *ctx and* A *depends on all variables* $y : A' \in \Gamma_1$.

In general, the type of x is $\Pi\Gamma_x.\, B_x$, a more sophisticated way of writing A. For example, $\Gamma_x = \cdot$ for all case distinctions in the body of the function 'tps' from Section 3.1, and $\Gamma_x = u : \mathrm{pf}\ A$ for the function 'abs' from Section 3.2. During evaluation x is bound to a closed object of the form $\lambda\Gamma_x.\, M$. In this case Γ_x provides all parameters which can possibly occur in M. For the purpose of case analysis it must hence be interpreted as an extension of the signature Σ. The judgment for cases relates a dependency context of Γ with respect to $x : \Pi\Gamma_x.\, B_x$, a combined signature $\Sigma; \Gamma_S$, (where constants are defined in Σ and parameters are defined in Γ_S) and the type of the case object G: $\Gamma_1; x : \Pi\Gamma_x.\, B_x; \Gamma_2; \Delta \vdash_{[\Sigma;\Gamma_S]} \ \Omega \in G$. We use this judgment in the inference rule for case which extends the rules in Figure 2 and completes the set of inference rules for the judgment $\Gamma; \Delta \vdash P \in F$.

$$\frac{\Gamma(x) = \Pi\Gamma_x.\, B_x \qquad \Gamma_1; x : \Pi\Gamma_x.\, B_x; \Gamma_2; \Delta \vdash_{[\Sigma;\Gamma_x]} \ \Omega \in G}{\Gamma; \Delta \vdash \mathrm{case}\ x\ \mathrm{of}\ \Omega \in G}\,\mathsf{case}$$

where $\Gamma_1, x : \Pi\Gamma_x.\, B_x, \Gamma_2$ is a dependency context of Γ with respect to x

Next, we discuss the typing rules for Ω (see [Sch97] for details). The judgment $\Gamma_1; x : \Pi\Gamma_x.\, B_x; \Gamma_2; \Delta \vdash_{[\Sigma;\Gamma_S]} \ \Omega \in G$ selects all constants from Σ and all parameters from Γ_S, which represent the possible "shapes" of the closed object bound to x. Since x might be a parametric function, the form of the rules is very complicated. The target type of x and the target type of each pattern must unify, which is a higher-order unification problem as discussed in Section 3.

Reconsider the proof of the deduction theorem from Section 3.2 again. The object $\lambda u : \mathrm{pf}\ \mathbf{A}.\, \mathrm{MP}\ (D_1\ u)\ (D_2\ u)$ has type 'pf $\mathbf{A} \to$ pf $\mathbf{B}'$'. The arguments of 'MP' are two objects, $D_1\ u$ and $D_2\ u$. In contrast, consider the constant declaration $\mathrm{MP} : \Pi\Gamma_{\mathrm{MP}}.\, \mathrm{pf}\ B'$ in Σ. The context Γ_{MP} has the form $D_1' : \mathrm{pf}\ (\mathbf{A}'\ \mathrm{imp}\ \mathbf{B}'), D_2' : \mathrm{pf}\ \mathbf{A}'$. To obtain the new context declaring D_1 and D_2, we must raise the type of the variables D_1' and D_2' by $\Gamma_x = u : \mathrm{pf}\ A$. The result is the context $\Gamma_{\mathrm{MP}}' = D_1 : \mathrm{pf}\ A \to \mathrm{pf}\ (\mathbf{A}'\ \mathrm{imp}\ \mathbf{B}'), D_2 : \mathrm{pf}\ A \to \mathrm{pf}\ \mathbf{A}'$ [2]. In general, a constant $c : \Pi\Gamma_c.\, B_c$ can be raised by a local subcontext Γ_x by abstracting each type in Γ_c over Γ_x, written as $\Gamma_c' = \Pi\Gamma_x.\, \Gamma_c$. This operation is defined as follows:

Definition 17 (Abstracting contexts)

$$\begin{aligned} \Pi\Gamma_x.\, \cdot \ &= \ \cdot \\ \Pi\Gamma_x.\, (\Gamma, y : A) \ &= \ (\Pi\Gamma_x.\, \Gamma), y' : \Pi\Gamma_x.\, A \end{aligned}$$

We use a substitution (a concept which is defined as usual, we write id_Γ for the identity substitution on a context Γ, and $M[\sigma]$ for application, if σ is a substitution) to transform the body of a function '$c\ \Gamma_c$' to its raised version. In our example 'MP $D_1'\ D_2'$' is transformed to 'MP $(D_1\ u)\ (D_2\ u)$' by a *raising substitution* which replaces D_1' by $D_1\ u$ and D_2' by $D_2\ u$. We denote it with τ.

[2] In this formulation of the system $\mathcal{M}_\omega$, we also raise the types of the implicit parameter A'. A' has hence type pf $A \to \mathrm{o}$ and not o as we assume above. But a careful inspection of the subordination relation (for its definition see Virga [Vir96], who calls it *dependency relation* of type families) yields, that formulas o cannot depend on any object representing a proof. Therefore, it can be safely assumed that A' has type o, even though $\mathcal{M}_\omega$ does not account for this fact. It is possible to extend Definition 17 and Definition 18 to include the subordination relation, but we omitted this discussion for the sake of clarity. The implementation described in Section 5 is based on this extended calculus.

Definition 18 (Raising substitution)

$$\begin{array}{rcl} \cdot \bullet \Gamma_x & = & \cdot \\ \Gamma, y' : A \bullet \Gamma_x & = & (\Gamma \bullet \Gamma_x), (y' \; \Gamma_x)/y \end{array}$$

In the example above τ is defined as $\Gamma_{\mathrm{MP}} \bullet (u : \mathrm{pf}\, A)$.

Everything is prepared to formally define the provability judgment for cases Ω. The rules for the judgment are given in Figure 3. Operationally speaking, these rules can be read as follows: The judgement is "activated" by the case rule, with $\Sigma; \Gamma_x$ as index to the $\vdash$ symbol. The type of the case subject is made explicit: it is $x : \Pi\Gamma_x.\, B_x$. The rules traverse first the local context Γ_x, and then the signature Σ to retrieve all declarations which provide possible patterns. Before we can determine if a case is possible or not, we must first calculate its type. This is a non-trivial problem, because case subjects can be parametric functions and hence variables must be raised using the raising substitution τ as discussed above. The type of the pattern is $\Pi\Gamma_x.\, B_c[\tau]$ which must unify with $\Pi\Gamma_x.\, B_x$. If a most general unifier σ can be found, the case is applicable, and either siguni or ctxuni is triggered. If not, either signonuni or ctxnonuni is triggered and the current declaration is ignored.

Because of reasons, we are not going to discuss here (see [Sch97]), the unification problem is a genuine unification problem and not a simple matching problem. This means, that if a most general unifier σ can be found, it might instantiate variables in the pattern $\lambda\Gamma_x.\, (c\; \Gamma_c)[\tau]$ which must hence be considered under the substitution σ. All free variables of $(\lambda\Gamma_x.\, (c\; \Gamma_c)[\tau])[\sigma]$ are therefore declared in Γ', all others in $\Gamma_2[\sigma]$. We must refer the interested reader for a more detailed presentation and discussion and the proof of well-formedness for the rules in Figure 3 to [Sch97].

4.5 Meta-theory

In this section we describe briefly and informally the properties we have shown for this system (see also the forthcoming tech report [Sch97]). Even though the goal is to prove that every proof term in $\mathcal{M}_\omega$ is total, we only show that every proof term in $\mathcal{M}_2$ is total for the purpose of this proposal. But we strongly believe, that this result can be extended to $\mathcal{M}_\omega$. We restricted the meta theory to this fragment of $\mathcal{M}_\omega$, because Twelf (which we present in the Section 5) generates only proof terms in this fragment, and the termination proof is direct. To achieve this result, we define an operational semantics for $\mathcal{M}_\omega$, and prove type preservation. For a restricted fragment of $\mathcal{M}_\omega$, we show that all proof terms with one outermost recursion operator make progress when evaluated. In addition, all proof terms in $\mathcal{M}_2$ terminate. An extension of the termination argument to full $\mathcal{M}_\omega$ is very likely to require an argument by logical relations.

The operational semantics for $\mathcal{M}_\omega$ is a standard big step call-by-value natural semantics [Pfe94b] with environments as it is defined for Standard ML [HS97]. It must be defined with respect to environments because the proof calculus for $\mathcal{M}_\omega$ does not contain a cut-rule: Derivations in the calculus always correspond to normalized, cut-free derivations in a more general sequent calculus. This property manifests itself already in the syntactical form of the proof terms. Split and case subjects are always variables $\mathbf{x}$ or x but never proof terms or LF-objects.

A well-typed proof term in $\mathcal{M}_2$ is total if it satisfies progress and termination. This means, evaluating a proof term will always make progress, and eventually terminate with a result. The big step semantics is inappropriate to show these properties because it does not allow an explicit notion of partial computation.

$$\frac{}{\Gamma_1; x : A; \Gamma_2; \Delta \vdash_{[\cdot;\cdot]} \ \cdot \in G}\ \mathsf{sigempty}$$

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

$$\begin{array}{r}
\Gamma'_c = \Pi\Gamma_x.\, \Gamma_c \\
\Gamma_1, \Gamma'_c, \Gamma_x \vdash \tau = \mathrm{id}_{\Gamma_1}, \mathrm{id}_{\Gamma_x}, \Gamma'_c \bullet \Gamma_x : \Gamma_1, \Gamma_x, \Gamma_c \\
\Gamma' \vdash \sigma = \mathrm{mgu}\ (\Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B_c[\tau], x \approx (\lambda\Gamma_x.\, (c\ \Gamma_c)[\tau])) : \Gamma_1, x : \Pi\Gamma_x.\, B_x, \Gamma'_c \\
\Gamma', \Gamma_2[\sigma] \vdash \sigma' = \sigma, \mathrm{id}_{\Gamma_2} : \Gamma_1, x : \Pi\Gamma_x.\, B_x, \Gamma'_c, \Gamma_2
\end{array}$$

$$\frac{\Gamma', \Gamma_2[\sigma]; \Delta[\sigma'] \vdash P \in G[\sigma'] \qquad \Gamma_1; x : \Pi\Gamma_x.\, B_x; \Gamma_2; \Delta \vdash_{[\Sigma;\Gamma_S]} \ \Omega \in G}{\Gamma_1; x : \Pi\Gamma_x.\, B_x; \Gamma_2; \Delta \vdash_{[\Sigma;\Gamma_S, p:\Pi\Gamma_p.\, B_p]} \ \Omega, (\Gamma'; \Gamma_2[\sigma] \triangleright (\lambda\Gamma_x.\, (p\ \Gamma_p)[\tau])[\sigma] \mapsto P) \in G}\ \mathsf{ctxuni}$$

$$\begin{array}{r}
\Gamma'_p = \Pi\Gamma_x.\, \Gamma_p \\
\Gamma_1, \Gamma'_p, \Gamma_x \vdash \tau = \mathrm{id}_{\Gamma}, \mathrm{id}_{\Gamma_x}, \Gamma'_p \bullet \Gamma_x : \Gamma_1, \Gamma_x, \Gamma_p \\
\Gamma' \vdash \sigma = \mathrm{mgu}\ (\Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B_p[\tau], x \approx (\lambda\Gamma_x.\, (p\ \Gamma_p)[\tau])) : \Gamma_1, x : \Pi\Gamma_x.\, B_x, \Gamma'_p \\
\Gamma', \Gamma_2[\sigma] \vdash \sigma' = \sigma, \mathrm{id}_{\Gamma_2} : \Gamma_1, x : \Pi\Gamma_x.\, B_x, \Gamma'_p, \Gamma_2
\end{array}$$

$$\frac{\Gamma_1; x : \Pi\Gamma_x.\, B_x; \Gamma_2; \Delta \vdash_{[\Sigma;\cdot]} \ \Omega \in G}{\Gamma_1; x : \Pi\Gamma_x.\, B_x; \Gamma_2; \Delta \vdash_{[\Sigma, c:\Pi\Gamma_c.\, B_c;\cdot]} \ \Omega \in G}\ \mathsf{signonuni}$$

$$\begin{array}{rcl}
\Gamma'_c & = & \Pi\Gamma_x.\, \Gamma_c \\
\Gamma_1, \Gamma'_c, \Gamma_x \vdash \tau & = & \mathrm{id}_{\Gamma}, \mathrm{id}_{\Gamma_x}, \Gamma'_c \bullet \Gamma_x : \Gamma_1, \Gamma_x, \Gamma_c \\
\multicolumn{3}{c}{\Pi\Gamma_x.\, B_x, \Pi\Gamma_x.\, B_c[\tau] \text{ not unifiable}}
\end{array}$$

$$\frac{\Gamma_1; x : \Pi\Gamma_x.\, B_x; \Gamma_2; \Delta \vdash_{[\Sigma;\Gamma_S]} \ \Omega \in G}{\Gamma 1; x : \Pi\Gamma_x.\, B_x; \Gamma_2; \Delta \vdash_{[\Sigma;\Gamma_S, p:\Pi\Gamma_p.\, B_p]} \ \Omega \in G}\ \mathsf{ctxnonuni}$$

$$\begin{array}{rcl}
\Gamma'_p & = & \Pi\Gamma_x.\, \Gamma_p \\
\Gamma_1, \Gamma'_p, \Gamma_x \vdash \tau & = & \mathrm{id}_{\Gamma}, \mathrm{id}_{\Gamma_x}, \Gamma'_p \bullet \Gamma_x : \Gamma_1, \Gamma_x, \Gamma_p \\
\multicolumn{3}{c}{\Pi\Gamma_x.\, B_x, \Pi\Gamma_x.\, B_p[\tau] \text{ not unifiable}}
\end{array}$$

Figure 3: Typing rules for $\Gamma_1; x : \Pi\Gamma_x.\, B_x; \Gamma_2; \Delta \vdash_{[\Sigma;\Gamma_S]} \ \Omega \in G$

For progress and termination, we refine the big step semantics to a small step semantics in the sense of [Plo75]. It is defined with respect to an abstract continuation machine [FSDF93] by a set of reduction rules, using a continuation stack to represent intermediate results. We can then show that proof terms in $\mathcal{M}_\omega$ (restricted to only allow only one outermost recursion operator), will always make progress: one reduction rule is always applicable.

For the proof of termination we devised a suitable termination condition for the fix-rule based on a lexicographical extension of Rohwedder's and Pfenning's subterm ordering [RP96]. Hence all proof terms in $\mathcal{M}_2$ are total functions, but we strongly believe that $\mathcal{M}_\omega$ itself can be already a meta logic of total functions.

4.6 Summary

The meta logic $\mathcal{M}_\omega$ is a logic of total functions, given a suitable termination condition for the recursion rule. On the one hand, $\mathcal{M}_\omega$ is too general for our purposes, but on the other it is not general enough. It is too general in the sense that the representation of Π_2-formulas (adequate for automatic proof search) does not require a meta logic which allows arbitrary quantifier alterations. It is not general enough, because we would also like to represent proofs which introduce new assumptions (parameters) into the proof process, hence violating the condition that all objects must be closed. We address the former point in [Sch97] where we embed $\mathcal{M}_2$ into $\mathcal{M}_\omega$, and for which we define a suitable termination condition leading to a formal proof of totality. The latter is proposed work.

5 Twelf

Twelf is an automated proof assistant for LF. Its proof search algorithm constructs total $\mathcal{M}_2$-proof terms[3] (with one outermost recursion operator) of Π_2-formulas. It is a reimplementation and extension of the logic programming language Elf [Pfe89] using explicit substitutions [DHKP96] and an internal spine notation of terms [CP97].

Besides the automatic component, Twelf also contains a interactive proof component which we do not discuss here. This section is organized as follows. In Section 5.1 we examine the proofs introduced in Section 3 for proof techniques, and relate them to the syntactical constructs of $\mathcal{M}_2$. We then address the question of proof strategies and describe the one which is implemented in Twelf in Section 5.2. After a brief discussion about the correctness of the implementation in Section 5.3, experimental results are assessed in Section 5.4. Even though Twelf works very well on a large variety of examples, there are a lot of improvements possible and necessary which we discuss in Section 5.5.

5.1 Proof techniques

Reviewing the proof of the type preservation theorem and the proof of the deduction theorem from Section 3, we can extract four main reoccurring proof principles:

Selection the of induction principle: Induction is typically done by inducing over one (structural induction) or more derivations (nested structural induction). It is guaranteed that the induction hypothesis is only applied to subderivations. The proof of the theorem is then done by considering different cases. Distinguishing cases is represented in $\mathcal{M}_2$ by 'case'.

Inversion: The proof of type preservation requires inversion. Inversion is applied to a derivation if the last rule of some derivation can be determined from the shape of its conclusion (uniquely or by considering different cases) to recover the different premisses. This step is represented in $\mathcal{M}_2$ also by 'case'.

Construction: In both proofs new derivations must be constructed as for example $(\mathsf{MP}\ (\mathsf{MP}\ \mathsf{S}\ \mathsf{K})\ \mathsf{K})$ in the first case of the deduction theorem. Such a derivation is either a result derivation or an argument to the induction hypothesis. The representation of this step in $\mathcal{M}_2$ is a well-typed LF object.

Appeal to the induction hypothesis: Both proofs require the application of induction hypothesis to "smaller" derivations, for all other arguments appropriate derivations are provided. In general there are only few possible applications of the induction hypothesis,

[3]Twelf is actually based on a an extension of $\mathcal{M}_2$ taking type subordination into account (see also previous footnote [2]).

because there are only few subderivations of the argument derivation. This step is represented in $\mathcal{M}_2$ by several nested 'let's. The result must be 'split' to recover the proof objects for the existentials.

As example we discuss the type preservation theorem (Theorem 9), taking snapshots of the trace produced by the automated theorem proving component. Each snapshot represents a partially constructed $\mathcal{M}_2$-function. For a clean presentation we introduce a new and simpler notation (which is also used in Twelf, when communicating with the user): Any proof state consists of a list of assumptions annotated with $\forall$, and a list of variables to be constructed, annotated with $\exists$. These annotations are very similar to mode annotations which have been introduced earlier by Rohwedder and Pfenning [RP96].

In the initial state the derivation of $\ulcorner e \hookrightarrow v \urcorner$, and hence the objects $\ulcorner e \urcorner$ and $\ulcorner v \urcorner$ are supplied as is $\ulcorner \Delta \triangleright e : \tau \urcorner$ and the object $\ulcorner \tau \urcorner$. Goal of the construction process is to find a method to construct always a derivation of type $\ulcorner \Delta \triangleright v : \tau \urcorner$. (Note, that we again omit the index variables as we did in the previous sections.)

$$\frac{\begin{array}{l}\forall E : \mathrm{exp},\ \forall V : \mathrm{exp},\ \forall T : \mathrm{tp},\\ \forall D : \mathrm{ev}\ E\ V,\ \forall P : \mathrm{of}\ E\ T,\ \exists Q : \mathrm{of}\ V\ T\end{array}}{\mathrm{tps}\ D\ P\ Q}$$

For the sake of readability, the type "tps D P Q" in the last line is a relational encoding of the partial $\mathcal{M}_2$-function, but not the partial $\mathcal{M}_2$-function itself.

5.1.1 Distinction of cases

The informal proof proceeds with distinction by cases over the derivation of $\ulcorner e \hookrightarrow v \urcorner$, mimicked by a technique called *Splitting*. We split the induction variable D, leading to three different cases. Without going into details, Twelf automatically orders all quantified variables in the context in such a way, that they always reflect the correct dependency order. This means that Q in all three subgoals can depend on any $\forall$ assumption declared to the left of it. Note, that this order is not uniquely determined.

Case: ev_lam:

$$\frac{\begin{array}{l}\forall E : \mathrm{exp} \to \mathrm{exp},\ \forall T : \mathrm{tp},\\ \forall P : \mathrm{of}\ (\mathrm{lam}\ E)\ T,\ \exists Q : \mathrm{of}\ (\mathrm{lam}\ E)\ T\end{array}}{\mathrm{tps}\ \mathrm{ev\_lam}\ P\ Q}$$

Case: ev_app:

$$\frac{\begin{array}{l}\forall E_1 : \mathrm{exp},\ \forall E_2 : \mathrm{exp},\ \forall V : \mathrm{exp},\ \forall T : \mathrm{tp},\\ \forall E_1' : \mathrm{exp} \to \mathrm{exp},\ \forall V_2 : \mathrm{exp},\\ \forall D_3 : \mathrm{ev}\ (E_1'\ V_2)\ V,\ \forall D_2 : \mathrm{ev}\ E_2\ V_2,\ \forall D_1 : \mathrm{ev}\ E_1\ (\mathrm{lam}\ E_1'),\\ \forall P : \mathrm{of}\ (\mathrm{app}\ E_1\ E_2)\ T,\ \exists Q : \mathrm{of}\ V\ T\end{array}}{\mathrm{tps}\ (\mathrm{ev\_app}\ D_3\ D_2\ D_1)\ P\ Q}$$

Case: ev_fix:

$$\frac{\begin{array}{l}\forall E : \mathrm{exp} \to \mathrm{exp},\ \forall V : \mathrm{exp},\ \forall T : \mathrm{tp},\\ \forall D_1 : \mathrm{ev}\ (E\ (\mathrm{fix}\ E))\ V,\\ \forall P : \mathrm{of}\ (\mathrm{fix}\ E)\ T,\ \exists Q : \mathrm{of}\ V\ T\end{array}}{\mathrm{tps}\ (\mathrm{ev\_fix}\ D_1)\ P\ Q}$$

5.1.2 Inversion

For the sake of brevity, we skip the discussion of the first two subgoals, but continue with the third. The next step in the proof is an inversion step. Very similar to the splitting operation, different cases of the derivation P must be considered. As we can expect from the discussion of Example 11, applying the splitting operation yields only one subgoal. The other two possibilities were rejected by Twelf, because their types are incompatible with the type of P. This restriction of cases can be attributed to the additional information about P which is encoded in P's dependent type.

$$\frac{\begin{array}{l}\forall E : \text{exp} \to \text{exp},\ \forall V : \text{exp},\ \forall T : \text{tp},\\ \forall D_1 : \text{ev}\ (E\ (\text{fix}\ E)))\ V\\ \forall P_1 : \Pi x\!:\!\text{exp}.\ \text{of}\ x\ T \to \text{of}\ (E\ x)\ T,\ \exists Q : \text{of}\ V\ T\end{array}}{\text{tps}\ (\text{ev\_fix}\ D_1)\ (\text{of\_fix}\ P_1)\ Q}$$

5.1.3 Appeal to the induction hypothesis

A splitting operation can only be applied to a $\forall$ assumption. In the current situation, there are many $\forall$ assumptions available, but splitting will not help to complete the proof of this subgoal. It is necessary to appeal to the induction hypothesis. The termination condition of the fix-rule requires that the induction hypothesis is applied to a term smaller than the originally supplied term $\forall D$. According to the termination ordering in [RP96] there is only one possibility: D_1. We must also construct an argument of type 'of $(E\ (\text{fix}\ E))\ T$'.

Twelf possesses a proof term search engine, which allows to construct LF-objects of given type. Among others, it constructs the term '$P_1\ (\text{fix}\ E)\ (\text{of\_fix}\ P_1)$' which has this type. Note, that this object corresponds to the result of the substitution lemma (Lemma 8) as used in the proof of Theorem 9. Applying the induction hypothesis to D_1 and this argument yields the following state:

$$\frac{\begin{array}{l}\forall E : \text{exp} \to \text{exp},\ \forall V : \text{exp},\ \forall T : \text{tp},\\ \forall D_1 : \text{ev}\ (E\ (\text{fix}\ E))\ V,\ \forall P_1 : \Pi x\!:\!\text{exp}.\, \text{of}\ x\ T \to \text{of}\ (E\ x)\ T,\\ \forall Q_2 : \text{of}\ V\ T,\ \exists Q_1 : \text{of}\ V\ T\end{array}}{\begin{array}{l}\text{tps}\ (\text{ev\_fix}\ D_1)\ (\text{of\_fix}\ P_1)\ Q_1\\ \leftarrow\ \text{tps}\ D_1\ (P_1\ (\text{fix}\ E)\ (\text{of\_fix}\ P_1))\ Q_2\end{array}}$$

We refer to this operation as *recursion*.

5.1.4 Construction

The final step for this proof branch is to determine the proof term for Q_1. The proof term search engine suggests Q_2. The instantiation (it is called *filling*) completes this branch of the proof.

$$\frac{\begin{array}{l}\forall E : \text{exp} \to \text{exp},\ \forall V : \text{exp},\ \forall T : \text{tp},\\ \forall D_1 : \text{ev}\ (E\ (\text{fix}\ E))\ V,\ \forall P_1 : \Pi x\!:\!\text{exp}.\, \text{of}\ x\ T \to \text{of}\ (E\ x)\ T,\\ \forall Q_2 : \text{of}\ V\ T\end{array}}{\begin{array}{l}\text{tps}\ (\text{ev\_fix}\ D_1)\ (\text{of\_fix}\ P_1)\ Q_2\\ \leftarrow\ \text{tps}\ D_1\ (P_1\ (\text{fix}\ E)\ (\text{of\_fix}\ P_1))\ Q_2\end{array}}$$

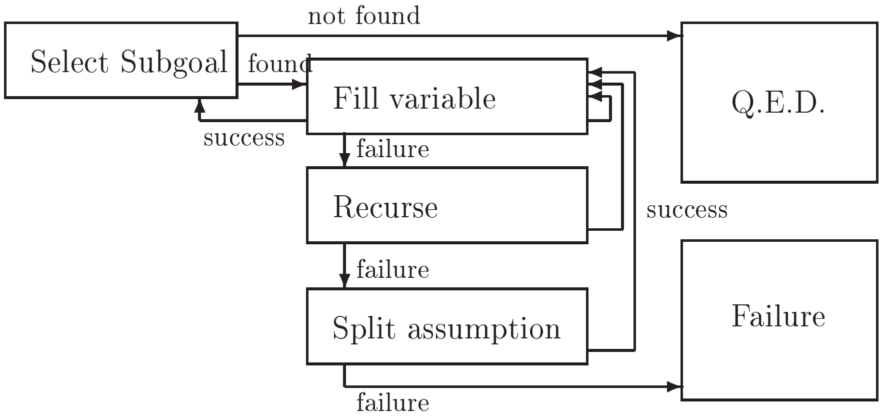

Figure 4: Proof strategy

This state is final because all ∃ variables are instantiated. The interested reader is invited to convince himself, that the other two subgoals (introduced by the first splitting step over the induction variable D) can easily be solved using "splitting", "recursion", and "filling" operations.

5.2 Strategy

The high degree of automation which we want to achieve with Twelf requires an intelligent combination of the three basic proof operations. Each operation must be applied with care because they are inherently expensive in time and space. This is evident, because every step involves unification ("splitting" and "filling"), or the calculation of subterms ("recursion"). Hence, under all circumstances, Twelf must avoid back-tracking. The strategy currently employed in Twelf does avoid it.

The goal is to construct an $\mathcal{M}_2$-function (which corresponds to instantiating all ∃ variables by well-typed LF-objects) for a given theorem. The strategy of achieving this goal is presented in Figure 4. The global state of the proof is a set of subgoals, introduced either by initialization (for proofs by simultaneous induction) or during a splitting operation. The strategy *activates* a subgoal and attempts to prove it. During the proof of the active subgoal, new subgoals might arise. Once all ∃ variables of the active subgoal are filled, it is *solved* and Twelf proceeds with the next unsolved subgoal. The theorem is proven, if all subgoals are solved.

Fill: The ∃ variables must be filled with a well-typed LF-objects. On the one hand, the search space of well-typed LF-objects is in general infinite: There are infinitely many well-typed LF-objects of type 'of $\ulcorner e \urcorner$ $\ulcorner \tau \urcorner$'. Hence the search problem is only semi-decidable. On the other hand in many examples, the size of LF-objects to be constructed is not very big. Trading feasibility for completeness, the Twelf proof search strategy is parametrized by an upper bound for the number of constructors to be used when searching for LF-objects. Consequently, the problem becomes decidable (even though failure is sometimes very slow).

The first step of the strategy is to locate all non-index ∃ variables in the context (which does not index any type in the context). The proof term search engine tries to simultaneously construct proof terms for all of them. If none could be found the strategy proceeds with

Recurse, otherwise this branch of the proof has been successfully completed and the next subgoal is selected. If none found, the strategy stops and the proof is completed.

Filling is represented in $\mathcal{M}_2$ as the proof term $[M_1, [M_2, ..[M_n, \langle\rangle]]]$, where M_i is the constructed LF-object for the ith $\exists$ variable).

Recurse: Twelf generates all possible recursive calls — only one is applied. First, all subterms of the induction terms are collected, then the remaining non-inductive universal variables of the induction hypothesis are filled with all possible well-typed LF-objects, each leading to a different recursive call. The vector of those LF terms is guaranteed to reduce the rank of the environment in case of execution (the rank of an environment is defined as the vector of the values of all induction variables in some prespecified lexicographical order). Out of all generated recursive calls, one which has not been applied yet is selected and applied. The strategy returns to step **Fill**.

Since only finitely many objects are considered for the recursive call and search always terminates Twelf can effectively calculate all finitely many recursive calls. If none are applicable, Twelf continues with step **Split**.

Recursion is represented in $\mathcal{M}_2$ by nested "lets"s, followed by some "split" operations to unwrap the witness LF-objects for the existentials

Split: For splitting, Twelf locates a non-index $\forall$ assumption in the current context (which does not index any type in the context). To split it, it locates all constants in the global signature and all parameters in the local context matching the variable's type. It then refines the current subgoal accordingly, possibly generating new subgoals, to be solved later. Among all $\forall$ assumptions, it selects the $\forall$ assumption which generates the least number of subgoals first. This heuristic works for all our examples surprisingly well, we leave the refinement of this heuristic to future research.

Splitting is represented in $\mathcal{M}_2$ by "case".

It is this strategy which has been used for all the proofs automatically found by Twelf. The strategy successively defines a function in $\mathcal{M}_2$.

5.3 Correctness

Randy Pollack writes in [Pol97] that it is impossible to prove a theorem prover totally correct. There is a vicious circle of dependencies which can only be unbroken by trusting in some fundamental truth. Formal verification, in Pollack's sense, cannot provide a rigorous proof of correctness because the machine running the verification process might be faulty, or the compiler used to compile the verifier buggy. To escape this obvious dilemma, Pollack distinguishes between the correctness of the method and the correctness of the proofs.

5.3.1 Correctness of the method

We claim that Twelf is correct because it constructs $\mathcal{M}_2$ proof terms. But there are also other ways how one could proceed with the correctness proof of the method.

One could for example extend $\lambda^\Box$ [DPS96] by dependent types to a meta logic of Twelf. $\lambda^\Box$ is a conservative extension of the simply-typed λ-calculus by a modal operator allowing the coexistence of primitive recursion and adequacy of higher-order representation. Each primitive recursive function representable in $\lambda^{\Box}$ is total.

Another possibility is to follow Rohwedder's approach to prove schema checking correct [Roh94]. He characterizes a logic programming interpretation of LF. As correctness proof one has to show the totality of the generated logic program.

Higher-order abstract syntax disallows a translation of Twelf proofs into a proof calculus with strong induction principles. Using de Bruijn indices [dB72], higher-order terms can be directly translated into first-order terms, for which we have induction principles. In this approach the transformation function must be adequate, and the image of a higher-order Twelf proof must be a first-order proof of the transformed property.

One could also envision to define a meta logic, inspired by $FO\lambda^{\Delta I\!N}$, a sequent calculus for intuitionistic logic with cut enriched by definitions and induction over natural numbers [McD97] which allows the representation of meta-languages (see also Section 6). $FO\lambda^{\Delta I\!N}$ would have to be enriched by principles to allow general structural induction. It is not clear how and if McDowell's and Miller's system can be extended.

5.3.2 Correctness of proofs

Trusting the correctness of the method might not be enough to convince a critical Twelf user of the correctness of a constructed proof. Pollack suggests to use a small trusted proof checker in order to check its correctness.

There are many (more or less) trusted proof checkers, as for example HOL [Ge93], LCF/ML [Pau87], Coq [C^+95], Lego [Pol94], Isabelle [Pau94], or the schema checker for Elf [Roh94] and many others. To employ these proof checkers one has to define an interface converting a Twelf proof into the appropriate syntax, so that the trusted proof checker can easily decide its correctness. Such an interface can be set up in different ways:

As described above one can translate Twelf proofs into de Bruijn notation making it first-order and hence adequate for the representation in a logic with induction principles. These proofs can then be checked with Coq, Lego, or Isabelle.

A second possibility is to represent LF itself in a proof assistant like Coq. The Twelf proof is then being transformed into a represented signature of the represented LF in Coq. Instead of transforming the proof itself one would have to transform the proof of the totality of its functional interpretation.

There exists already a proof checker for LF: the schema checker [Roh94] checks if a LF signature represents a total function. Unfortunately, the schema checker is neither small nor has been completely proven correct.

Finally, the translation of proofs into natural language can greatly improve one's confidence. The still unconvinced user can only be convinced on a semantic level.

5.4 Results

Twelf has been successfully employed to prove several non-trivial theorems *fully* automatically. In the area of Mini-ML it was used to prove value soundness in 0.372 sec and type preservation (Theorem 9) in 1.999 sec. Moreover it was used to show the completeness of refining the standard big step operational semantics into a continuation based small step operational semantics in 8.332 sec. There are at least two equivalent formulations of the completeness theorem, one is automatically provable, the other is not for reasons we mention briefly in Section 5.5. Twelf could not prove the soundness direction of this refinement, because the proof requires complete induction which is not supported by Twelf, yet. In one formulation there exists a monomorphism mapping soundness proofs into completeness proofs, a property which requires a *level 3* type family when expressed in LF. Twelf successfully derived its existence in 23.838 sec.

In the area of logic, Twelf was used to derive, among others, the proof of the deduction theorem (Theorem 10) in 1.606 sec, the soundness and completeness (in 0.533 sec) direction of the equivalence of uniform derivations and resolution in a fragment of hereditary Harrop formulas, which does not require hypothetical meta-reasoning. It took Twelf 120.216 sec to prove that there exists an embedding of Cartesian closed categories into the simply-typed λ-calculus. All timings have been taken on a 300 Mhz Pentium-II machine, running Linux 2.30, New Jersey SML 109.26.1, and Twelf 5.0.

5.5 Limitations

The more restricted the meta logic $\mathcal{M}_2$, the higher is the degree of automation. For some examples, the proposed strategy from Section 5.2 is too restrictive: one formulation of the completeness theorem mentioned in the previous section is not provable with the current strategy, because at one point no LF-objects can be constructed for the non-inductive arguments to the induction hypothesis (due to undecidability of higher-order unification). But this problem could be solved by postponing their instantiation, a feature not support by the current strategy.

Another limitation of the strategy is the interpretation of failure. Currently, Twelf's strategy reaches the failure state only if neither **Fill** nor **Recurse** nor **Split** are applicable. Since in general a proof state contains always many $\forall$ assumptions, the **Split** operation will continue splitting until it stops (no further $\forall$ assumptions are available) or it runs forever. This behavior is undesired, because one of the objectives of the Twelf project is to use the theorem proving component as a mean to find mistakes in data structures (*level 0* types), properties (*level 1* type families) or the formulation of theorems (*level 2* type families). The strategy must be refined in such a way, that it recognizes failure situations much earlier than it currently does. Then Twelf can be used to find the case in a proof, where a theorem breaks, which may lead to a generalization of the induction hypothesis or a refinement of the signature.

Another major restriction of the presented strategy is that it does not allow lemma application. The application of lemmas is very similar to the application of the induction hypothesis. But, it is far less restricted: Lemmas need not to be applied to objects "smaller" then the original ones, because it is already guaranteed to terminate. Because of the absence of this restriction, lemma application is incompatible with the current Twelf proof strategy. Many theorems are hence not automatically provable with the current system. We hope to solve this problem by employing standard techniques form inductive theorem proving, as for example difference unification [Bas93] and rippling [Hut97, BSvH$^+$93].

The current algorithm employed in the proof term search engine is a straightforward, brute force tree traversal of a limited depth search space. It requires a complete traversal to decide that no well-typed LF-object was found. For many examples, the use of dependent types restricts the search space generously, but not for all. Proving for example the existence of an embedding of Cartesian closed categories into the simply-typed λ-calculus requires the search to a depth that the algorithm takes several seconds to traverse the complete search space once. For these examples, the current version of the search algorithm is very unattractive. Indexing techniques as proposed by Graf [Gra96] might help to improve the currently naive design by replacing it with a resolution-like technique.

6 Related work

The logical framework LF defined by Honsell, Harper, and Plotkin [HHP93] is an extension of the simply-typed λ-calculus by dependent types. It is strictly subsumed by the calculus of constructions [CH88], which also allows polymorphism and type constructors. The usefulness of induction has been recognized by Paulin-Mohring and others, leading to the calculus of inductive constructions, an extension of the standard calculus by inductive datatypes [PM93]. It has been implemented as the proof assistant Coq [C$^+$95]. Coq contains a powerful inductive theorem prover component based on strong induction principles and an extensive library of tactics.

LF itself has been implemented by Pfenning [Pfe89] as the logic programming language Elf. Elf is not a proof assistant but its operational semantics allows experiments with represented deductive systems and their meta theory. The adequacy of higher-order representation guarantees that every object constructed in Elf corresponds to a derivation in the original deductive system. Higher-order abstract syntax [PE88] is the central representation technique in this framework which allows an elegant representation of hypothetical judgments and languages which contain a variable concept on their own. But languages using higher-order abstract syntax are no longer inductive. No strong induction principles exist. Hence this technique is only of limited use in the setting of automated proof assistants relying on the existence of strong induction principles as for example Coq, Nuprl [C$^+$86], Lego [Pol94], Alf [AGNvS94, Mag95], PVS [ORS92] and many others. Languages using higher-order abstract syntax can still be represented in these systems as defined types, but they cannot be used for inductive reasoning. Attempts to simulate higher-order abstract syntax in the setting of strong induction principles as for example [DH94, DFH95] have been unsatisfactory from a technical and theoretical point of view.

The logical framework Isabelle [Pau94] is implemented as a generic theorem prover. Object logics are represented in Isabelle by extending its meta logic which is based on the simply-typed λ-calculus. The same way, datatypes with strong induction principles can be introduced. Because of the same reasons as for Coq, proofs over languages using higher-order representation techniques cannot be elegantly represented in Isabelle. Isabelle's automated theorem proving component offers a powerful selection of tactics and tacticals.

In many examples, LF is used to represent meta theorems and their proofs (for example Church-Rosser theorem [Pfe93], compiler verification [HP92a], Mini-ML [Pfe94b], logics[Pfe94b], categorical structures [Geh94], occurrences of continuation parameters in CPS terms [DP95], and many others). A represented proof *must* correspond to a total function, a property which is not easily verifiable by hand. To verify this property automatically, Rohwedder devised a technique called schema checking [Roh94]. The schema checker does not support the search for the proof. The proof must be constructed by hand, and then typed in by the user before it can be verified.

The present proposal describes the meta logic $\mathcal{M}_\omega$ and its implementation as the system Twelf, which is an automated theorem prover over LF signatures for which no strong induction principles exist. In earlier work [DPS96], we presented an extension of the simply typed λ-calculus by a modal operator which showed that primitive recursion and higher-order abstract syntax can coexist in the same framework. $\lambda^\Box$ is restricted to simple types only, it is still an open question whether it can be elegantly extended to dependent types.

Different from $\lambda^{\Box}$, $\mathcal{M}_\omega$ is a pattern-matching calculus, which greatly simplifies the writing of programs for larger applications. Similar ideas have already been presented by Coquand [Coq92] and by Jouannaud and Okada [JO91]. Pattern-matching calculi must ensure a complete coverage of the defining patterns by definitional reflection. Among others, this property has been extensively studied by Schröder-Heister [SH93a] and Magnusson for the proof assistant Alf [Mag95].

In earlier work we defined a meta logic MLF for the Horn fragment of LF [Sch95] which extends Gallier's system $\mathcal{G}^{\supset,\wedge,\vee,\forall,\exists,\bot,\mathrm{cut}}$[Gal93] by constructs to allow reasoning about LF. The main insufficiency of MLF is that a general cut elimination theorem has not been proven and hence its consistency is unclear.

We see our system $\mathcal{M}_\omega$ as a first step towards a meta logic which can also represent proofs using meta-hypothetical reasoning. Proofs are represented by total functions, which are possibly applied to open objects. Assumptions introduce new parameters in the proof process, a feature which is currently not supported but proposed work. Some ideas of how to treat local assumptions have been discussed in [DPS96].

McDowell and Miller avoid this problem of local assumptions in their system $FO\lambda^{\Delta I\!N}$ [MM97], a meta logic based on the a sequent calculus formulation of intuitionistic logic extended by induction over natural numbers and definitions. It is more general than $\mathcal{M}_\omega$ because definitions allow the embedding of many different meta languages (which they call object logics) into $FO\lambda^{\Delta I\!N}$ (as for example the second order fragment of minimal logic, or its extension with linear constructs). In contrast, $\mathcal{M}_\omega$ is designed solely as a meta logic for LF.

Because of this generality, $FO\lambda^{\Delta I\!N}$ has the disadvantage, that it cannot exploit properties which are special to a particular meta language, as for example the way proofs by structural induction are represented in LF. McDowell [McD97] recognizes this problem which becomes apparent in two places: First, many proofs represented in LF make use of the higher-order properties of LF, for which natural numbers induction must be extended to richer induction schemes, and second the subformula property does not hold for the $nat\mathcal{L}$ rule which makes automated proof search for proofs in $FO\lambda^{\Delta I\!N}$ intractable.

According to our knowledge the meta logic $\mathcal{M}_\omega$ of Twelf is also a first attempt to allow the methodology of LF to be employed in an automated theorem proving system. This raises the issue of how to apply lemmas. From a given set of assumptions Twelf must construct a sequence of lemma applications which constructs an LF-object of given type. This is closely related to the problem of difference unification [Bas93] and rippling [Hut97, BSvH+93] as implemented in the theorem prover Inka [Hut96] and Oyster-Clam [BvHHS90]. In addition, Twelf's proof term search engine can be greatly improved by using indexing techniques as proposed by Graf[Gra96].

Magnusson developed in her thesis the system Alf [Mag95] a syntax-directed proof editor which inspired our proof search technique for Twelf. Our implementation uses explicit substitutions [Ros96] — so does Alf — and a spine calculus representation of terms, refining an earlier implementation by Harper and Pfenning of a module system for Elf [HP92b] and incorporates ideas as presented by Cervesato and Pfenning [CP97].

7 Proposed work

In my thesis, I propose to extend $\mathcal{M}_\omega$ to a meta logic which allows meta-hypothetical reasoning. Its proof terms must be provably total even when applied to arguments which are not necessarily closed. I will also extend the implementation of Twelf to a proof assistant which supports meta-hypothetical reasoning as described in Section 7.1. If time permits I hope to equip Twelf with an automated lemma application component, so that the proof of cut-elimination for intuitionistic logic can be derived fully automatically, as described in Section 7.2.

But if it should happen, that for some currently unforeseeable reason, $\mathcal{M}_\omega$ is not extensible, I propose to make Twelf an efficient, reliable and competitive automated theorem prover, also incorporating automatic lemma applications (Section 7.2) and some other major improvements which are discussed in Section 7.3.

7.1 Extension of $\mathcal{M}_\omega$ to meta-hypothetical reasoning

$\mathcal{M}_\omega$ is a meta logic of total functions. Our argument for totality has been based on the assumption they are only applied to closed arguments. But this is a major restriction, because many represented proofs in LF introduce new parameters (and hence the arguments can be open) as for example the proof of the Church-Rosser theorem [Pfe93] or the various cut-elimination theorems [Pfe95]. This has been discussed in [Pfe94b] in great depth of detail.

In my thesis, I propose to extend $\mathcal{M}_\omega$ to a meta logic, which can guarantee totality, even if the argument objects are open with respect to a well-structured context. This context represents local assumptions, which have been introduced during the proof process. It grows in a well-defined predictable way, as preliminary experiments have shown. In my thesis, I will also extend the implementation of Twelf to a proof assistant which allows hypothetical reasoning by introducing assumptions.

Hypothetical meta-reasoning should not be confused with reasoning about languages which use hypothetical judgments and higher-order abstract syntax as higher-order representation techniques. In general, when one uses higher-order representation techniques for formal systems, it is very likely, that meta-theoretic considerations require meta-hypothetical reasoning. From the discussions in this proposal it is clear that the current design supports neither the introduction of new parameters nor a strategic decision when to do it.

7.2 Automated lemma application

In its current version $\mathcal{M}_\omega$ does not allow lemma application, absolutely necessary for the proof process. Because lemmas can be applied to arbitrary arguments, one cannot use the termination ordering to limit the number of possibilities. Recall that this is different for the application of the induction hypothesis.

In Section 5, assumptions were annotated with $\forall$ and variables to be constructed with $\exists$. I propose to use a technique similar to rippling [BSvH$^+$93, Hut97] to construct the right order of lemmas which provide the right $\forall$ assumptions to eventually instantiate $\exists$ variables. Rippling is defined in the realm of equational reasoning only. By characterizing wave fronts, one is able to direct equations in such a way that wave fronts are pushed inwards when applying the rule. I believe, that applying this idea in a setting with dependent types limits the number of possible lemma applications drastically (even more than in the standard equational case), so that one has a good chance of incorporating automated lemma application into the proof process.

A new strategy has to be engineered in such a way, that it can incorporate at the very least meta-hypothetical reasoning and rippling, and hopefully also a relaxed version of induction hypothesis application: The current implementation requires all universal variables to be instantiated before applying the induction hypothesis. Delaying this choice might enlarge the number of provable theorems (as for example the original formulation of the completeness theorem for natural and continuation style semantics, Section 5.5).

Meta-hypothetical reasoning and automated lemma application lead to the definition of the generalized meta logic $\mathcal{M}$ and an extended implementation of Twelf which should allow the automatic deduction of many theorems. Because of the results presented in this proposal and earlier results achieved by Rohwedder [Roh94], I am convinced, that this project is feasible. Eventually, I hope to prove cut-elimination for intuitionistic and classical logic — fully automatically.

7.3 Improvements

The proof term search engine should be updated by a saturation based resolution technique, using indexing techniques as described by Graf in [Gra96]. The current search strategy is brute force and limited only in depth. It is currently the bottleneck of Twelf, drastically reducing its performance. This improvement will require some more research.

If a theorem is not provable, Twelf might halt or might not halt. In both cases it is difficult or impossible to trace back the actual cause of failure. To make Twelf more robust it has to be extended by failure heuristics to improve feed-back to the user. A clean distinction between induction variables and non-induction variables will simplify the formulation of such a heuristic. It is crucial for the project to assign meaning to the failure of proofs.

The soundness proof of the compiler example (relating natural and continuation stack semantics) is not provable by structural induction alone. One has to use complete induction. Twelf can be improved by extending the termination order towards a complete termination order. The more induction hypotheses Twelf generates, the more theorems it can prove in principle. There is a trade-off between completeness and efficiency I will have to investigate.

References

[AGNvS94] Thorsten Altenkirch, Veronica Gaspes, Bengt Nordström, and Björn von Sydow. *A User's Guide to ALF*. Chalmers University of Technology, Sweden, May 1994.

[Bar92] Henk Barendregt. Lambda calculi with types. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of logic in Computer Science*, volume II, pages 118–309. Oxford University Press, 1992.

[Bas93] David Basin. Difference unification. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, 1993.

[BSvH+93] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993.

[BvHHS90] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The oyster-clam system. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990.

[C+86] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[C+95] Christina Cornes et al. *The Coq Proof Assistant, Reference Manual, Version 5.10.* INRIA, CNRS, France, 1995.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3), 1988.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Coq92] Thierry Coquand. Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 71–83, Båstad, Sweden, 1992.

[CP97] Iliano Cervesato and Frank Pfenning. A linear spine calculus. Technical Report CMU-CS-97-125, CMU, 1997.

[dB72] N.G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formulamanipulation with application to the church-rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[DFH95] Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 124–138, Edinburgh, Scotland, April 1995. Springer-Verlag LNCS 902.

[DH94] Joëlle Despeyroux and André Hirschowitz. Higher-order abstract syntax with induction in Coq. In Frank Pfenning, editor, *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, pages 159–173, Kiev, Ukraine, July 1994. Springer-Verlag LNAI 822.

[DHKP96] Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In *Joint International Conference and Symposium on Logic Programming (JICSLP'96), Bonn, Germany*, 1996.

[DP95] Olivier Danvy and Frank Pfenning. The occurrence of continuation parameters in cps terms. Technical Report CMU-CS-95-121, CMU, 1995.

[DPS96] Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In *Typed Lambda Calculi and Applications, Nancy.* Springer, April 1996.

[Eri91] L.-H. Eriksson. A finitary version of the calculus of partial inductive definitions. In *Second International Workshop on Extensions of Logic Programming, Stockholm*, pages 89–134. Springer, January 1991.

[FSDF93] Cormac Flanagan, Amry Sbray, Bruce Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Conference on Porgramming Language Design and Implementation. Albuquerque, New Mexico*, June 21-25 1993.

[Gal93] Jean Gallier. Constructive logics Part I: A tutorial on proof systems and typed λ-calculi. *Theoretical Computer Science*, 110:249–339, 1993.

[Ge93] M.J.C. Gordon and T.F. Melham (editors). *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[Geh94] Wolfgang Gehrke. Proof of the decidability of the uniform word problem for monads assisted by Elf. Technical Report 94-66, RISC, Linz, Austria, August 1994.

[Gen35] G. Gentzen. Untersuchungen über das logische Schliessen. In *Mathematische Zeitschrift*, volume 39, 1935. English translation, Investigation into logical deduction, in [Sza69].

[Gra96] Peter Graf. *Term Indexing.* Springer LNAI 1053, 1996.

[HB34] D. Hilbert and P. Bernays. *Grundlagen der Mathematik*, volume I. Springer Verlag, 1934.

[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.

[HP92a] John Hannan and Frank Pfenning. Compiler verification in LF. In Andre Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992.

[HP92b] Robert Harper and Frank Pfenning. A module system for a programming language based on the LF logical framework. Technical Report CMU-CS-92-191, Carnegie Mellon University, Pittsburgh, Pennsylvania, September 1992.

[HS97] Robert Harper and Chris Stone. An interpretation of Standard ML in type theory. Technical Report CMU–CS–97–147, Carnegie Mellon University, Pittsburgh, PA, June 1997. (Also published as Fox Memorandum CMU–CS–FOX–97–01.).

[Hue75] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Hut96] Dieter Hutter. Inka - the next generation. In *13th International Conference on Automated Deduction, New Brunswick, USA*, 1996.

[Hut97] Dieter Hutter. Equalizing terms by difference reduction techniques. In *14th International Conference on Automated Deduction, Townsville, Australia*, 1997.

[JO91] Jean-Pierre Jouannaud and Mitsuhiro Okada. A computation model for executable higher-order algebraic specification languages. In Gilles Kahn, editor, *Proceedings of the 6th Annual Symposium on Logic in Computer Science*, pages 350–361, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[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.

[Mil89] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In Peter Schroeder-Heister, editor, *Proceedings of the International Workshop on Extensions of Logic Programming*, pages 253–281, Tübingen, Germany, 1989. Springer-Verlag LNAI 475.

[MM97] Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax. In *Symposium on Logic in Computer Science, Warsaw, Poland*, 1997.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*, volume 7 of *International Series of Monographs on Computer Science*. Oxford University Press, 1990.

[ORS92] Sam Owre, John Rusby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Conference on Auotmated deduction (CADE)*, volume 607 of *LNCS*, pages 748–752, Saratoga, NY, 1992.

[Pau87] Larry Paulson. *Logic and Computation: Interactive proof with Cambridge LCF*. Cambridge University Press, 1987.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer-Verlag LNCS 828, 1994.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.

[Pfe89] Frank Pfenning. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[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.

[Pfe93] Frank Pfenning. A proof of the Church-Rosser theorem and its representation in a logical framework. *Journal of Automated Reasoning*, 1993. To appear. A preliminary version is available as Carnegie Mellon Technical Report CMU-CS-92-186, September 1992.

[Pfe94a] F. Pfenning. Structural cut elimination in linear logic. Technical Report CMU-CS-94-222, Carnegie Mellon University, Department of Computer Science, December 1994.

[Pfe94b] Frank Pfenning. Computation and deduction. Unpublished lecture notes, May 1994.

[Pfe95] Frank Pfenning. Structural cut elimination. In D. Kozen, editor, *Proceedings of the Tenth Annual Symposium on Logic in Computer Science*, pages 156–166, San Diego, California, June 1995. IEEE Computer Society Press.

[Plo75] Gordon Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science 1*, 1:125–159, 1975.

[PM93] Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J.F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 328–345, Utrecht, The Netherlands, March 1993. Springer-Verlag LNCS 664.

[Pol94] Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.

[Pol97] Randy Pollack. *Twenty Five Years of Constructive Type Theory*, chapter How to Believe a Machine-Checked Proof. Oxford University Pres, 1997. to appear.

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

[Ros96] Kristoffer Rose. Explicit substitutions — tutorial and survey. Technical Report BRICS LS-96-3, BRICS, 1996.

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

[Sch95] Carsten Schürmann. A computational meta logic for the Horn fragment of LF. Technical Report CMU-CS-95-218, Carnegie Mellon University, SCS, December 1995.

[Sch97] Carsten Schürmann. Automating the meta theory of deductive systems. Technical Report CMU-CS-97-???, Carnegie Mellon University, 1997. forthcoming.

[SH93a] Peter Schroeder-Heister. Definitional reflection and the completion. In R. Dyckhoff, editor, *Proceedings of the 4th International Workshop on Extensions of Logic Programming*, pages 333–347. Springer-Verlag LNAI 798, 1993.

[SH93b] 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.

[Sny91] Wayne Snyder. *A proof Theory for general Unification*. Birkhäuser, 1991.

[Sza69] M.E. Szabo, editor. *The collected papers of Gerhard Gentzen*. Amsterdam, North-Holland Pub. Co., 1969.

[Vir96] Roberto Virga. Higher-order superposition for dependent types. In *Proceedings of 7th International Conference on Rewriting Techniques and Applications, New Brunswick, USA*. Springer, 1996.