A Meta Logical Framework Based on Realizability

Carsten Schürmann *

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

1 Introduction

The logical framework LF [HHP93] has been designed as a meta-language for representing deductive systems. It allows concise encodings of many inference systems, such as natural deduction, sequent calculi, type systems, operational semantics, compilers, etc. The logical framework derives its expressive power from dependent types together with higher-order representation techniques which directly support common concepts in deductive systems such as variable binding, capture-avoiding substitutions, parametric and hypothetical judgments and substitution properties. The fact that these notions are an integral part of the logical framework would seem to make it an ideal candidate to not only reason within but reason about the various inference systems.

Unfortunately, higher-order representation techniques are incompatible with induction principles [DPS97]. A program in a language which employs higher-order encodings, for example, may need to traverse a λ-binder in order to make a recursive call and it thereby dynamically extends the set of constructors of a given datatype. The *closed world assumption* underlying the inductive interpretation of datatypes does therefore not apply to this setting.

The basis of our meta-logical framework is the observation, that these extensions are not arbitrary, but that they are regular in structure. Therefore, for this work, we make a *regular world assumption* which allows dynamic but regular extensions of types by new constructors.

In this paper, we develop the type theory $\mathcal{M}_2^+$, which is explicitly designed to support automated inductive theorem proving and proof checking for inference systems represented in LF under the regular world assumption. From a functional perspective, $\mathcal{M}_2^+$ is a calculus of partial functions. Under the regular world assumption each of these functions can be analyzed for termination and progress properties and those which satisfy both are called *realizers*. On the other hand, from a logical perspective, the types of $\mathcal{M}_2^+$ are propositions, and the realizer of a type is its proof.

* This work was sponsored by NSF Grant CCR-9619584 and by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050.

The meta-logic $\mathcal{M}_2^+$ is a conservative extension of the the meta-logic $\mathcal{M}_2$ which we have developed in previous work [SP98]. In essence, if we restrict $\mathcal{M}_2^+$ to the closed world assumption we obtain $\mathcal{M}_2$.

$\mathcal{M}_2^+$ is implemented in the Twelf system [PS99]. We have applied it successfully to prove, for example, the equivalence of Hilbert's calculus, natural deduction calculus, and the sequent calculus. Other experiments include the proof of the Church-Rosser theorem for the untyped and the simply-typed λ-calculus, and cut-elimination results for various propositional and first-order sequent calculi. In each case we only specified a sequence of lemmas, the induction variable, and an upper bound for search, the proof was completely automatic in every other respect.

This paper is organized as follows. In Section 2 we revisit higher-order representation techniques, specifically hypothetical judgments, and in Section 3 we motivate the regular world assumption underlying the type theory $\mathcal{M}_2^+$. As running example we use an algorithm for transforming natural deduction derivations to sequent derivations. The functional perspective of $\mathcal{M}_2^+$ is formally presented in Section 4, and its operational semantics in Section 5. We present a logical perspective of $\mathcal{M}_2^+$ in Section 6 before we conclude and assess results in Section 7.

2 Hypothetical Judgments

In this paper, we restrict ourselves to the dependently-typed meta-language λ^Π also known under the name LF. Its formulation is standard and we take $\beta\eta$-conversion as the notion of definitional equality [HHP93,Coq91].

As running examples throughout this paper we use two proof calculi for the implicational fragment of intuitionistic logic: natural deduction and sequent calculus, and their representation in the logical framework. Although we present only one connective, the example scales to full first-order logic. In all examples, we use $\ulcorner \cdot \urcorner$ as the representation function mapping into LF.

Example 1 (Formulas).

$$\textit{Formulas: } G ::= Q \mid G_1 \supset G_2$$

$$\ulcorner G_1 \supset G_2 \urcorner = \ulcorner G_1 \urcorner \;\mathrm{imp}\; \ulcorner G_2 \urcorner \qquad \begin{array}{ll} \mathrm{o} & : \mathrm{type} \\ \mathrm{imp} & : \mathrm{o} \to \mathrm{o} \to \mathrm{o} \end{array}$$

'imp' is used as an infix operator throughout this paper. Furthermore, all our examples are parametric in atomic propositions Q, therefore we do not have to represent them explicitly.

The encoding of the natural deduction calculus [Gen35] illustrates the idea of hypothetical judgments in the $\supset$ I rule which discharges the hypothesis u. We follow the standard approach and make all hypotheses explicit and write them to the left of the '$\vdash$' symbol in form of a context Γ.

$$\Gamma ::= \cdot \mid \Gamma, u :: A$$

We assume that the labels u do not occur repeatedly and that the order in which they occur is irrelevant.

Example 2 (Natural Deduction).

$$
\begin{array}{lll}
\textit{Provability judgment: } \Gamma \vdash G & \text{nd} & : \text{o} \to \text{type} \\[1ex]
\dfrac{}{\Gamma, u :: G \vdash G}\, u & & \\[2ex]
\dfrac{\Gamma, u :: G_1 \vdash G_2}{\Gamma \vdash G_1 \supset G_2} \supset \mathrm{I}^u & \text{impi} & : (\text{nd } G_1 \to \text{nd } G_2) \\
 & & \quad \to \text{nd } (G_1 \text{ imp } G_2) \\[1ex]
\dfrac{\Gamma \vdash G_1 \supset G_2 \qquad \Gamma \vdash G_1}{\Gamma \vdash G_2} \supset \mathrm{E} & \text{impe} & : \text{nd } (G_1 \text{ imp } G_2) \to \text{nd } G_1 \\
 & & \quad \to \text{nd } G_2
\end{array}
$$

Following standard practice [Pfe91], we omit in this presentation all implicit Π-abstractions from types. Moreover we represent the hypotheses Γ directly in the LF context $\ulcorner \Gamma, u :: G_1 \urcorner = \ulcorner \Gamma \urcorner, u : \text{nd } \ulcorner G_1 \urcorner$ and consequently, the encoding of the first rule does not give rise to a new LF constant because

$$
\ulcorner \dfrac{}{\Gamma, u :: G \vdash G}\, u \urcorner = u : \text{nd } \ulcorner G \urcorner .
$$

Sequents are encoded by two separate type families. Hypotheses to the left of the sequent symbol '$\Longrightarrow$' are labeled with h and encoded using 'hyp' whereas the conclusion to the right is encoded by 'conc'. And again, the representation of the sequent calculus in LF employs hypothetical judgments [Pfe95]. The premiss of a derivation which ends in the $\supset$R-rule, for example, is encoded as an LF-function of type 'hyp $G_1 \to$ conc G_2'.

Example 3 (Sequent Calculus).

$$
\begin{array}{lll}
\textit{Provability judgment: } \Gamma \Longrightarrow G & \text{hyp} & : \text{o} \to \text{type} \\
 & \text{conc} & : \text{o} \to \text{type} \\[1ex]
\dfrac{}{\Gamma, h :: G \Longrightarrow G}\, \text{init} & \text{init} & : \text{hyp } G \to \text{conc } G \\[2ex]
\dfrac{\Gamma \Longrightarrow G_1 \qquad \Gamma, h :: G_1 \Longrightarrow G_2}{\Gamma \Longrightarrow G_2}\, \text{cut} & \text{cut} & : \text{conc } G_1 \\
 & & \quad \to (\text{hyp } G_1 \to \text{conc } G_2) \\
 & & \quad \to \text{conc } G_2 \\[1ex]
\dfrac{\Gamma, h :: G_1 \Longrightarrow G_2}{\Gamma \Longrightarrow G_1 \supset G_2} \supset \mathrm{R} & \text{impR} & : (\text{hyp } G_1 \to \text{conc } G_2) \\
 & & \quad \to \text{conc } (G_1 \text{ imp } G_2)
\end{array}
$$

$$
\dfrac{\Gamma, h :: G_1 \supset G_2 \Longrightarrow G_1 \qquad \Gamma, h :: G_1 \supset G_2, h_2 :: G_2 \Longrightarrow G_3}{\Gamma, h :: G_1 \supset G_2 \Longrightarrow G_3} \supset \mathrm{L}
$$

$$
\begin{array}{ll}
\text{impL} & : \text{conc } G_1 \\
 & \quad \to (\text{hyp } G_2 \to \text{conc } G_3) \\
 & \quad \to (\text{hyp } (G_1 \text{ imp } G_2) \to \text{conc } G_3)
\end{array}
$$

The representations of the natural deduction calculus and the sequent calculus are adequate, which means that canonical forms [HHP93] of type 'nd' are in one-to-one correspondence with natural deduction derivations, and canonical forms of type 'conc' are in one-to-one correspondence with sequent derivations.

3 The Regular World Assumption

The constructors 'impi' and 'impe' for objects of type 'nd' for natural deductions from Example 2 are not the constructors of an *inductive* type, since we have a negative occurrence of 'nd G_1' in the argument type of impi : (nd $G_1 \rightarrow$ nd $G_2) \rightarrow$ nd $(G_1$ imp $G_2)$. This is not just a formal observation, but has practical consequences. Gentzen has shown in [Gen35] that there exists an algorithm which maps natural deductions to sequent deductions and vice versa. Without a more general paradigm for datatypes, we cannot encode this algorithm *directly*, i.e. without using auxiliary constructions, such as, for example, first-order encodings with de Bruijn indices [dB72]. In general, when programming, we want to take full advantage of the expressive power of LF with all its higher-order features and dependent types.

In this paper we do not present an entire programming language. Instead we present a small calculus of functions which is sufficient to express interesting algorithms such as the one above, the proof of the Church-Rosser theorem [Pfe93] and the proof of the cut-elimination theorem [Pfe95]. Alternatively, those algorithms can be encoded as relations in the logic programming language Elf [Pfe94]. As a running example we use Gentzen's result from which we later extract an algorithm.

Lemma 1 (Natural deduction $\Rightarrow$ sequent calculus). *For any set of assumptions Γ and formula G, if $\mathcal{D}$ is a natural deduction derivation of $\Gamma \vdash G$ then there exists a sequent derivation of $\Gamma \Longrightarrow G$.*

Proof. By structural induction on $\mathcal{D}$:

Case 1: $\mathcal{D} = \dfrac{}{\Gamma, u :: G \vdash G}\, u$

Immediate, by init: $\Gamma, h :: G \Longrightarrow G$.

Case 2: $\mathcal{D} = \dfrac{\begin{matrix}\mathcal{D}' \\ \Gamma, u :: G_1 \vdash G_2\end{matrix}}{\Gamma \vdash G_1 \supset G_2} \supset \mathrm{I}^u$

$\mathcal{E}'$ of $\Gamma, h :: G_1 \Longrightarrow G_2$ — by induction hypothesis on $\mathcal{D}'$

$\Gamma \Longrightarrow G_1 \supset G_2$ — by rule $\supset$R on $\mathcal{E}'$

Case 3: $\mathcal{D} = \dfrac{\begin{matrix}\mathcal{D}_1 \\ \Gamma \vdash G_1 \supset G_2\end{matrix} \qquad \begin{matrix}\mathcal{D}_2 \\ \Gamma \vdash G_1\end{matrix}}{\Gamma \vdash G_2} \supset \mathrm{E}$

$\mathcal{E}_1$ of $\Gamma \Longrightarrow G_1 \supset G_2$	by induction hypothesis on $\mathcal{D}_1$
$\mathcal{E}_2$ of $\Gamma \Longrightarrow G_1$	by induction hypothesis on $\mathcal{D}_2$
$\mathcal{E}'_2$ of $\Gamma, h :: G_1 \supset G_2 \Longrightarrow G_1$	by weakening on $\mathcal{E}_2$
$\mathcal{E}_3$ of $\Gamma, h :: G_1 \supset G_2, h_2 :: G_2 \Longrightarrow G_2$	by init
$\mathcal{E}_4$ of $\Gamma, h :: G_1 \supset G_2 \Longrightarrow G_2$	by $\supset$L on $\mathcal{E}'_2$ and $\mathcal{E}_3$
$\Gamma \Longrightarrow G_2$	by cut on $\mathcal{E}_1$ and $\mathcal{E}_4$

□

In essence, this proof contains the recipe we need to transform a given natural deduction derivation $\mathcal{D}$ into a sequent derivation. The proof goes by induction over $\mathcal{D}$, which is functionally speaking a case analysis of $\mathcal{D}$ and allows recursion to model appeals to the induction hypothesis. We name this function **ndseq** and begin its presentation with **Case 3**.

Example 4 (Functional interpretation of **Case 3***).*

$$\begin{array}{l} \textbf{ndseq}\ (\mathrm{impe}\ D_1\ D_2) = \\ \quad (\mathrm{cut}\ (\textbf{ndseq}\ D_1) \\ \qquad (\lambda h_1 : \mathrm{hyp}\ (G_1\ \mathrm{imp}\ G_2).\, \mathrm{impL}\ (\textbf{ndseq}\ D_2)\ (\lambda h_2 : \mathrm{hyp}\ G_2.\, \mathrm{init}\ h_2)\ h_1)) \end{array}$$

The argument to **ndseq** is (impe D_1 D_2), an encoding of the form of $\mathcal{D}$. After two recursive calls (**ndseq** D_1) and (**ndseq** D_2), this program constructs the return value using only LF constants. The construction *does not* apply weakening explicitly, because it is implicitly supported by the logical framework. □

Functional calculi which provide the concept of case analysis and recursion for reasoning about higher-order encodings of deductive systems have already been discussed in the literature in different but somewhat similar settings. In earlier work, for example, we have extended the simply-typed λ-calculus by a concept of primitive recursion via iterators [DPS97] which was recently generalized to the dependently typed setting by Leleu [Lel98]. The central idea in both systems is a modal restriction on types which identifies *closed* objects and renders iteration and case analysis over closed LF objects sound operations. Recently, Hofmann has given a categorical model for this construction [Hof99].

Case 2 of Gentzen's proof introduces new hypotheses, which means, that **ndseq** is indeed a function to range over open objects. The meta logic $\mathcal{M}_2$ [SP98], a direct predecessor of this work, is therefore not general enough to express Gentzen's proof, because it is restricted to reasoning over closed objects only. Unlike ML, where the set of constructors for a given type is fixed in the context (if we exclude the treatment of exceptions), our functional calculus must offer the possibility to dynamically extend it by *new* constructors.

Informally, Gentzen's algorithm maps derivations $\mathcal{D}$ of judgment $u_1 :: G_1, \ldots, u_n :: G_n \vdash G$ to derivations $\mathcal{E}$ of judgment $h_1 :: G_1, \ldots, h_n :: G_n \Longrightarrow G$, but formally things are more complicated: The representation of $\mathcal{D}$, for example, has type

$$u_1 : \mathrm{nd}\ \ulcorner G_1 \urcorner, \ldots, u_n : \mathrm{nd}\ \ulcorner G_n \urcorner \overset{\Pi}{\vdash} \ulcorner \mathcal{D} \urcorner : \mathrm{nd}\ \ulcorner G \urcorner,$$

whereas the expected type of the outcome — the representation of $\mathcal{E}$ — is:

$$h_1 : \text{hyp}\ \ulcorner G_1 \urcorner, \ldots, h_n : \text{hyp}\ \ulcorner G_n \urcorner \overset{\Pi}{\vdash} \ulcorner \mathcal{E} \urcorner : \text{conc}\ \ulcorner G \urcorner$$

The u_i's and the h_i's are all previously dynamically introduced constructors (or parameters), and for the purpose of Gentzen's algorithm, they must both be inserted into the LF context during execution. But note, the algorithm cannot be applied to arbitrary open derivations. It will get stuck if it is applied to a u_i if there is no appropriate h_i! Thus, we stipulate a *regular world assumption*, which loosens the *closed world assumption* underlying $\mathcal{M}_2$; under this new assumption we additionally consider LF objects which are valid in regularly formed LF contexts. The structure of these contexts must be defined a priori.

In our example, during execution, the LF context (or parameter context) must always obey the following syntactical description.

$$\Phi ::= \cdot \mid \Phi, u : \text{nd}\ \ulcorner G \urcorner, h : \text{hyp}\ \ulcorner G \urcorner$$

This definition of Φ is custom-designed to fit Gentzen's algorithm. Of course, implementations of other algorithms require different definitions of Φ. Given an LF object $D : \text{nd}\ F$ in a context Φ of this form, we still can analyze all the cases of D! Note that this definition guarantees that u and h always come in pairs. Once the evaluation of our function **ndseq** encounters a parameter, **Case 1** of Gentzen's algorithm applies. **Case 2** shows how new parameters are introduced into the parameter context during execution.

Example 5 (Functional interpretation of **Case 1***).*

$$\textbf{ndseq}\ \underline{u} = \underline{h}$$

$\underline{u}$ is a variable which can be instantiated with any of the parameters "u" in the LF context, and simultaneously $\underline{h}$ is instantiated with the other parameter "h" — this is always possible because of the regular form of Φ. We underline the variable names to emphasis the different character of this variable concept. □

Example 6 (Functional interpretation of **Case 2***).*

$$\begin{array}{l}
\textbf{ndseq}\ (\text{impi}\ (\lambda u : \text{nd}\ G_1 .\, D'\ u)) = \\
\quad \textbf{let} \\
\quad\quad \textbf{new}\ u : \text{nd}\ G_1, h : \text{hyp}\ G_1 \\
\quad\quad \textbf{val}\ \ \ (E\ h) = \textbf{ndseq}\ (D'\ u) \\
\quad \textbf{in} \\
\quad\quad (\text{impR}\ (\lambda h.\, E\ h)) \\
\quad \textbf{end}
\end{array}$$

The argument to **ndseq** is the representation of $\mathcal{D}$ from the proof of Lemma 1. Since $D' = \ulcorner \mathcal{D}' \urcorner$ is hypothetical, D' has functional type, and we must introduce two new parameters (indicated by the **new**-notation [Ode94]) $u : \text{nd}\ G_1, h : \text{hyp}\ G_1$. The result of the recursive call on $(D'\ u)$ yields a sequent derivation $(E'\ h)$ which is parametric in h and to which 'impR' is applied before it is returned as result. □

Let $\Phi ::= \cdot \mid \Phi, u : \text{nd}\ \ulcorner G \urcorner, h : \text{hyp}\ \ulcorner G \urcorner$

$$
\begin{array}{l}
\textbf{fun ndseq}\ \underline{u} = (\text{init}\ \underline{h}) \\
\quad \mid \textbf{ndseq}\ (\text{impi}\ D') = \\
\qquad \textbf{let} \\
\qquad\quad \textbf{new}\ u : \text{nd}\ G, h : \text{hyp}\ G \\
\qquad\quad \textbf{val}\ \ \ (E\ h) = \textbf{ndseq}\ (D'\ u) \\
\qquad \textbf{in} \\
\qquad\quad (\text{impR}\ (\lambda h.\, E\ h)) \\
\qquad \textbf{end} \\
\quad \mid \textbf{ndseq}\ (\text{impe}\ D_1\ D_2) = \\
\qquad \textbf{let} \\
\qquad\quad \textbf{val}\ (E_1) = \textbf{ndseq}\ D_1 \\
\qquad\quad \textbf{val}\ (E_2) = \textbf{ndseq}\ D_2 \\
\qquad \textbf{in} \\
\qquad\quad (\text{cut}\ E_1\ (\lambda h_1.\, \text{impL}\ E_2\ (\lambda h_2.\, \text{init}\ h_2)\ h_1)) \\
\qquad \textbf{end}
\end{array}
$$

Fig. 1. An implementation of Gentzen's algorithm

Example 7 (An implementation of Gentzen's algorithm). Figure 1 shows an implementation of the proof of Lemma 1 in ML-like notation, summarizing results from Example 4–6. It employs a let-construct in order to cleanly separate between the LF and the type system level. The function owes its simplicity and elegance to the expressive power of the logical framework LF. □

The idea of introducing formal parameters during the execution of a program goes back to Miller [Mil90]. In his paper he suggests to extend ML by a mechanism to support parameters in order to allow datatypes to use higher-order abstract syntax but without dependencies.

4 Type System $\mathcal{M}_2^+$

We begin now with the formal presentation of the type system $\mathcal{M}_2^+$ which is general enough to represent partial functions like **ndseq**. Throughout this entire section and the next (which defines $\mathcal{M}_2^+$'s operational semantics) we view $\mathcal{M}_2^+$ from a purely functional perspective. In Section 6 we change our point of view and interpret propositions-as-types presupposing that we can restrict $\mathcal{M}_2^+$ to a calculus of total functions. Those functions are called realizers and can be interpreted as proofs. The overall goal of this work is to use the meta-logic $\mathcal{M}_2^+$ as formal basis for an automated theorem prover. The reader can find a detailed presentation of $\mathcal{M}_2^+$ and its meta-theory in the forthcoming thesis [Sch00].

$\mathcal{M}_2^+$ is a two level system. LF on the lower level provides the data which the partial functions on the $\mathcal{M}_2^+$ level manipulate. $\mathcal{M}_2^+$ inherits not only LF's expressive power, but also the properties which are associated with hypothetical judgments, namely, substitutions, weakening, contraction, and exchange.

We start with the notion of types. Informally, **ndseq**'s type must express that, for any parameter context Φ of the appropriate form, for any formula G : o, and for any LF-object D which satisfies $\Phi \vdash^{\Pi} D : \text{nd}\ G$, there exists an LF-object E which satisfies $\Phi \vdash^{\Pi} E : \text{conc}\ G$. For the purpose of this paper, we leave the specification of Φ to the module system, as we do, for example, with the specification of the LF signature. We use '$\in$' as typing symbol for $\mathcal{M}_2^+$ in order to clearly distinguish it from the ':' for LF. Moreover, because $\mathcal{M}_2^+$ inherits dependencies from LF, we cannot write '$o \Rightarrow \text{nd}\ G \Rightarrow \text{conc}\ G'$' for **ndseq**'s type, because G occurs unbound. Instead, we have adopted a notation borrowed from logic.

$$\textbf{ndseq} \in \forall G : \text{o}. \forall D : \text{nd}\ G. \exists E : \text{conc}\ G. \top$$

Again, for the purpose of this section, we call these constructs types, but in the next section we will view them as propositions. Other example of types result from examining subterms from the program in Figure 1. The body of the impe-case, for example, uses two new variables D_1, and D_2 (including implicit arguments), and therefore it has type $\exists E : \text{conc}\ G_1. \top$ given that $D_1 : \text{nd}\ G_2 \supset G_1$ and $D_2 : \text{nd}\ G_2$.

A new variable concept becomes necessary if we try to assign a type to the body of the first case: **ndseq** $\underline{u}$ = (init $\underline{h}$). $\underline{u}$ and $\underline{h}$ are variables which will be instantiated with parameters — we therefore call them parameter variables — and thus we cannot use the universal quantifier to bind them. Universal quantifiers are designed to bind variables whose structure can be analyzed by case distinction. In our example, there exists only one basic building block, in the general case however, Φ can be described by several different blocks [Sch00]. We distinguish each of them by labels, and assign the label L to the one block in our example.

$$\Phi ::= \cdot \mid \Phi, (\underline{u} : \text{nd}\ \ulcorner G \urcorner, \underline{h} : \text{hyp}\ \ulcorner G \urcorner)^L$$

Blocks of well-typed parameter variables ranging over those blocks are called *variable blocks*,

$$\textit{Variable block: } \rho ::= \cdot \mid \rho, \underline{x} : A$$

and they are being bound by a new quantifier $\mathbf{\Pi}$. Therefore the type for the first case of **ndseq** is

$$\mathbf{\Pi}(\underline{u} : \text{nd}\ G, \underline{h} : \text{hyp}\ G)^L. \exists E : \text{conc}\ G. \top.$$

Generalized contexts summarize not only the set of free variables, but also the form of the parameter context.

$$\textit{Generalized Context: } \Psi ::= \cdot \mid \Psi, x : A \mid \Psi, \rho^L$$

A generalized context of the form $D : \text{nd}\ G_1, (\underline{u} : \text{nd}\ G_2, \underline{h} : \text{nd}\ G_2)^L$, for example, expresses that a variable D is declared, and that the parameter context contains at least one block. Variable blocks can only be instantiated by other variable blocks, or by parameters. In general, the validity judgments for generalized contexts extend the standard definition for well-typed LF contexts in a straightforward way.

$$\mathcal{M}_2^+ \textit{ types: } F ::= \forall x : A.\, F \mid \mathbf{\Pi}\rho^L.\, F \mid \exists x : A.\, F \mid \top$$

The $\mathcal{M}_2^+$ type system can be easily extended by product types.

4.1 The Core of $\mathcal{M}_2^+$

Similar to ML, the syntax we have presented in Figure 1 contains some syntactic sugar, in order to make this material accessible. The desugared version consists of several mutually dependent categories: programs and declarations among others. The type of each program and of each declaration is uniquely defined by inference rules. In spirit very similar to Standard ML, $\mathcal{M}_2^+$ provides a notion of program variables which can act as subroutines. Altogether we have three different variable concepts: LF-variables x, parameter variables $\underline{x}$, and program variables $\mathbf{x}$ which can only be instantiated by programs. Because of the two-level design of $\mathcal{M}_2^+$, $\mathbf{x}$ can never occur in any object bound to x or $\underline{x}$ or their types.

$$\textit{Meta-contexts: } \Delta ::= \cdot \mid \Delta, \mathbf{x} \in F$$

Programs resemble somewhat the set of proof terms of an intuitionistic sequent calculus [Gal93] with the important addition of the $\nu\rho^L.\, D$ declaration which, operationally speaking, introduces new parameters into the LF context before it evaluates D.

$$\begin{array}{ll}
\textit{Programs:} & P ::= \mathbf{x} \mid \Lambda x : A.\, P \mid \lambda\rho^L.\, P \mid \langle M, P\rangle \mid \langle\rangle \mid \mathbf{let}\ D\ \mathbf{in}\ P \\
\textit{Declarations:} & D ::= \cdot \mid \nu\ \rho^L.\, D \mid \mathbf{x} \in F = P\ M, D \mid \mathbf{x} \in F = P\ \rho, D \\
& \qquad \mid \langle x : A, \mathbf{y} \in F\rangle = P, D
\end{array}$$

Example 8 (Desugared program code). The partially desugared version of the body of the impi-case (it still omits implicit arguments) in Gentzen's algorithm in Figure 1 has in $\mathcal{M}_2^+$ the following form.

$$\begin{array}{l}
\mathbf{ndseq}\ (\mathrm{impi}\ D') = \\
\quad \mathbf{let} \\
\quad\quad \nu\ (\underline{u} : \mathrm{nd}\ F, \underline{h} : \mathrm{hyp}\ F)^L\,. \\
\quad\quad \mathbf{x}_0 \in \exists E : \mathrm{conc}\ G_2.\, \top = \mathbf{ndseq}\ (D'\ u), \\
\quad\quad \langle E : \mathrm{conc}\ G_2, \mathbf{x}_1 \in \top\rangle = \mathbf{x}_0 \\
\quad \mathbf{in} \\
\quad\quad (\mathrm{impR}\ (\lambda h : \mathrm{hyp}\ F.\, E\ h))
\end{array}$$

The newly introduced parameters in $\underline{u}$ and $\underline{h}$ are externally bound in the result of the ν declaration, indicated by the shift of the type of E. □

Programs and declarations give rise to the definition of two typing judgments. We write $\Psi; \Delta \vdash P \in F$ if program P is of type F, in an generalized context Ψ, and a meta context Δ, whereas for declarations we write $\Psi; \Delta \vdash D \in \Psi'; \Delta'$.

$$\frac{(\mathbf{x} \in F) \text{ in } \Delta}{\Psi; \Delta \vdash \mathbf{x} \in F}\,\mathsf{axvar} \qquad \frac{}{\Psi; \Delta \vdash \langle\rangle \in \top}\,\mathsf{R}\top$$

$$\frac{\Psi, x : A; \Delta \vdash P \in F}{\Psi; \Delta \vdash \Lambda x : A.\, P \in \forall x : A.\, F}\,\mathsf{R}\forall \qquad \frac{\Psi, \rho^L; \Delta \vdash P \in F}{\Psi; \Delta \vdash \lambda\rho^L.\, P \in \mathbf{\Pi}\rho^L.\, F}\,\mathsf{R}\Pi$$

$$\frac{\Psi \vdash M : A \qquad \Psi; \Delta \vdash P \in F[M/x]}{\Psi; \Delta \vdash \langle M, P\rangle \in \exists x : A.\, F}\,\mathsf{R}\exists$$

$$\frac{\Psi; \Delta \vdash D \in \Psi'; \Delta' \qquad \Psi, \Psi'; \Delta, \Delta' \vdash P \in F}{\Psi; \Delta \vdash \mathbf{let}\ D\ \mathbf{in}\ P \in F}\,\mathsf{sel}$$

$$\frac{}{\Psi; \Delta \vdash \cdot \in \cdot; \cdot}\,\mathsf{Ldone}$$

$$\frac{\Psi \vdash \rho^L \text{ variable block} \qquad \Psi, \rho^L; \Delta \vdash D \in \Psi'; \Delta'}{\Psi; \Delta \vdash \nu\, \rho^L.\, D \in \Psi''; \Delta''}\,\mathsf{Lnew}$$
$$\text{where } \Psi''; \Delta'' = \text{abstract } \rho^L.\, (\Psi'; \Delta')$$

$$\frac{\Psi; \Delta \vdash P \in \forall x : A.\, F \qquad \Psi \vdash M : A \qquad \Psi; \Delta, \mathbf{y} \in F[M/x] \vdash D \in \Psi'; \Delta'}{\Psi; \Delta \vdash \mathbf{y} \in F[M/x] = P\ M, D \in \Psi'; \mathbf{y} \in F[M/x], \Delta'}\,\mathsf{L}\forall$$

$$\frac{\Psi; \Delta \vdash P \in \mathbf{\Pi}\rho^L.\, F \qquad \rho'^L \in \Psi \qquad \rho' \equiv \rho \qquad \Psi; \Delta, \mathbf{y} \in F[\rho'/\rho] \vdash D \in \Psi'; \Delta'}{\Psi; \Delta \vdash \mathbf{y} \in F[\rho'/\rho] = P\ \rho', D \in \Psi'; \mathbf{y} \in F[\rho'/\rho], \Delta'}\,\mathsf{L}\Pi$$

$$\frac{\Psi; \Delta \vdash P \in \exists x : A.\, F \qquad \Psi, x : A; \Delta, \mathbf{y} \in F \vdash D \in \Psi'; \Delta'}{\Psi; \Delta \vdash \langle x : A, \mathbf{y} \in F\rangle = P, D \in x : A, \Psi'; \mathbf{y} \in F, \Delta'}\,\mathsf{L}\exists$$

Fig. 2. Typing Rules of the Core of $\mathcal{M}_2^+$

While D is computed, it possibly calculates new LF-objects which are returned in Ψ' and new partially applied programs which are returned in Δ'.

Typing programs: $\Psi; \Delta \vdash P \in F$
Typing declarations: $\Psi; \Delta \vdash D \in \Psi'; \Delta'$

The typing rules for these judgments are mutually dependent and resemble a sequent calculus (see Figure 2). The abstraction operation that defines the side condition of rule Lnew raises the types and formulas in $\Psi'; \Delta'$ with respect to ρ^L, respectively. The premiss $\Psi \vdash M : A$ of the rules $\mathsf{R}\exists$ and $\mathsf{L}\forall$ is the standard LF typing judgments. It allows M to depend on free variables and parameters declared in Ψ.

4.2 Case Analysis and Recursion

Unlike in ML where case subjects may involve only some variables and where patterns are linear, we take in $\mathcal{M}_2^+$ always the entire (generalized) context into account. Because of dependencies, patterns are in general non-linear, and when analyzing different cases of one variable, one may implicitly distinguish cases over another that occurs in an index to a type family. In Gentzen's algorithm, for example, matching against the declaration D determines the form of the formula G in the 'impi'-case. Thus, our design employs explicit substitutions $(\psi; \delta)$ as case subjects. Another positive aspect is that substitution application (whose definition we omit here) is absorbed by simple substitution composition therefore protecting the body of each case. Each individual case is therefore closed in that it defines its own generalized context and a corresponding mediating generalized substitution ψ.

$$\begin{array}{ll} \textit{Programs:} & P ::= \ldots \mid \text{case } (\psi; \delta) \text{ of } \Omega \\ \textit{Cases} & \Omega ::= \cdot \mid \Omega, (\Psi \triangleright \psi \mapsto P) \end{array}$$

A generalized substitution ψ is defined with respect to generalized contexts, in particular it substitutes LF objects for LF-variables and variable blocks for variable blocks. The explicit substitution $(\psi; \delta)$ in the case-program consists to one part of a generalized substitution and to the other of a meta substitution δ, which replaces program variables by programs.

$$\begin{array}{ll} \textit{Generalized substitutions:} & \psi ::= \cdot \mid \psi, M/x \mid \psi, \rho'/\rho \\ \textit{Meta substitutions:} & \delta ::= \cdot \mid \delta, P/\mathbf{x} \end{array}$$

We write $\Psi' \vdash \psi \in \Psi$ if ψ is valid and $\Psi'; \Delta' \vdash \psi; \delta \in \Psi; \Delta$ for well-typed meta/generalized substitution pairs.

Example 9 (Case analysis). The main case distinction of Gentzen's algorithm from Figure 1 is expressed in $\mathcal{M}_2^+$ by the program below. Each case is equipped with its own generalized context and a mediating generalized substitution whose domain is $G : \text{o}, D : \text{nd } G$. For the sake of brevity, we only show its skeleton.

$$\begin{array}{l} \lambda G : \text{o}.\, \lambda D : \text{nd } G. \\ \quad \text{case } (G/G, D/D; \cdot) \\ \quad\ \text{of } (G : \text{o}, \underline{u} : \text{nd } G, \underline{h} : \text{hyp } G \\ \qquad\quad \triangleright (G/G, \underline{u}/D) \mapsto \ldots) \\ \quad \mid\ (G_1 : \text{o}, G_2 : \text{o}, D' : \text{nd } G_1 \to \text{nd } G_2 \\ \qquad\quad \triangleright (G_1 \text{ imp } G_2/G, \text{impi } (\lambda u : \text{nd } G_1.\, D' u)/D) \mapsto \ldots) \\ \quad \mid\ (G_1 : \text{o}, G_2 : \text{o}, D_1 : \text{nd } (G_1 \text{ imp } G_2), D_2 : \text{nd } G_1 \\ \qquad\quad \triangleright (G_2/G, \text{impe } D_1\ D_2/D) \mapsto \ldots) \\ \in \forall G : \text{o}.\, \forall D : \text{nd } G.\, \exists E : \text{conc } G.\, \top \end{array}$$

□

$$\frac{\Psi; \Delta, \mathbf{x} \in F \vdash P \in F}{\Psi; \Delta \vdash \mu\mathbf{x} \in F.\, P \in F}\,\mathsf{Rctx} \qquad \frac{\Psi; \Delta \vdash \psi; \delta \in \Psi'; \Delta' \qquad \Psi'; \Delta' \vdash \Omega \in F}{\Psi; \Delta \vdash \text{case } (\psi; \delta) \text{ of } \Omega \in [\psi]F}\,\mathsf{case}$$

$$\frac{}{\Psi; \Delta \vdash \cdot \in F}\,\mathsf{base} \qquad \frac{\Psi' \vdash \psi \in \Psi \qquad \Psi; \Delta \vdash \Omega \in F \qquad \Psi'; [\psi]\Delta \vdash P \in [\psi]F}{\Psi; \Delta \vdash \Omega, (\Psi' \triangleright \psi \mapsto P) \in F}\,\mathsf{alt}$$

Fig. 3. Case Analysis and Recursion in $\mathcal{M}_2^+$.

$\mathcal{M}_2^+$ also defines a recursion operator μ since functions such a Gentzen's algorithm are recursive; each recursive call corresponds to an appeal of the induction hypothesis (see the proof of Lemma 1).

$$\textit{Programs: } P ::= \ldots \mid \mu\mathbf{x} \in F.\, P$$

The rules defining case analysis and recursion can be found in Figure 3. The premiss of the case rule ensures that the subject of the case program is well-typed. For the sake of brevity, we have omitted a strictness side condition on alt, that guarantees that matching against ψ always instantiates all variables declared in Ψ'.

5 Operational Semantics of $\mathcal{M}_2^+$

The rules defining the operational semantics are constructed in such a way, that once evaluation is invoked in a regularly formed parameter context Φ its result is well-defined in the same context. During evaluation, new parameters can be dynamically introduced into the parameter context.

The operational semantics is call-by-value and it relates programs to be executed with the result of their computation. Altogether, there are three evaluations judgments for programs, declarations, and cases, respectively. Each judgment is indexed by the parameter context Φ.

Program evaluation	$\Phi \vdash P \hookrightarrow V$
Declarations evaluation	$\Phi \vdash D \hookrightarrow (\eta; \delta)$
Selection	$\Phi \vdash (\psi; \delta) \sim \Omega \hookrightarrow V$

The first and third judgment return values, whereas the second returns a list of values in form of a substitution providing (canonical) LF-objects for Ψ' and values for Δ' as used in the typing judgment of declarations.

$$\textit{Values } V ::= \langle\rangle \mid \langle M, V\rangle \mid \Lambda x : A.\, P \mid \lambda\rho^L.\, P$$

The rules defining the operational semantics are given in Figure 4. We assume that only well-typed programs, declarations, or cases are executed and we can show that the execution is type preserving.

$$\frac{}{\Phi \vdash \Lambda x : A.\, P \hookrightarrow \Lambda x : A.\, P}\ \mathsf{ev\_Lam} \qquad \frac{}{\Phi \vdash \lambda \rho^L.\, P \hookrightarrow \lambda \rho^L.\, P}\ \mathsf{ev\_lam}$$

$$\frac{\Phi \vdash P \hookrightarrow V}{\Phi \vdash \langle M, P \rangle \hookrightarrow \langle M, V \rangle}\ \mathsf{ev\_inx} \qquad \frac{}{\Phi \vdash \langle\rangle \hookrightarrow \langle\rangle}\ \mathsf{ev\_unit}$$

$$\frac{\Phi \vdash D \hookrightarrow \psi; \delta \qquad \Phi \vdash P[\psi; \delta] \hookrightarrow V}{\Phi \vdash \mathbf{let}\ D\ \mathbf{in}\ P \hookrightarrow V}\ \mathsf{ev\_let}$$

$$\frac{\Phi \vdash P[\mu \mathbf{x} \in F.\, P/\mathbf{x}] \hookrightarrow V}{\Phi \vdash \mu \mathbf{x} \in F.\, P \hookrightarrow V}\ \mathsf{ev\_rec} \qquad \frac{\Phi \vdash (\psi; \delta) \sim \Omega \hookrightarrow V}{\Phi \vdash \text{case } (\psi; \delta) \text{ of } \Omega \hookrightarrow V}\ \mathsf{ev\_case}$$

$$\frac{}{\Phi \vdash \cdot \hookrightarrow \cdot; \cdot}\ \mathsf{ev\_empty}$$

$$\frac{\Phi \vdash P \hookrightarrow \langle M, V \rangle \qquad \Phi \vdash D[M/x; V/\mathbf{y}] \hookrightarrow \psi'; \delta'}{\Phi \vdash \langle x : A, \mathbf{y} \in F \rangle = P, D \hookrightarrow M/x, \psi'; V/\mathbf{y}, \delta'}\ \mathsf{ev\_split}$$

$$\frac{\Phi \vdash P \hookrightarrow \Lambda x : A.\, P' \qquad \Phi \vdash P'[M/x] \hookrightarrow V \qquad \Phi \vdash D[V/\mathbf{y}] \hookrightarrow \psi'; \delta'}{\Phi \vdash \mathbf{y} \in F = P\ M, D \hookrightarrow \psi'; V/\mathbf{y}, \delta'}\ \mathsf{ev\_App}$$

$$\frac{\Phi \vdash P \hookrightarrow \lambda \rho'^L.\, P' \qquad \Phi \vdash P'[\rho/\rho'] \hookrightarrow V \qquad \Phi \vdash D[V/\mathbf{y}] \hookrightarrow \psi'; \delta'}{\Phi \vdash \mathbf{y} \in F = P\ \rho, D \hookrightarrow \psi'; V/\mathbf{y}, \delta'}\ \mathsf{ev\_app}$$

$$\frac{\Phi, \rho^L \vdash D \hookrightarrow \psi'; \delta'}{\Phi \vdash \nu\ \rho^L.\, D \hookrightarrow \lambda \rho.\, \psi'; \lambda \rho^L.\, \delta'}\ \mathsf{ev\_new}$$

$$\frac{\Phi \vdash [\psi''; \delta] P \hookrightarrow V}{\Phi \vdash (\psi; \delta) \sim (\Omega, (\Psi \triangleright \psi' \mapsto P)) \hookrightarrow V}\ \mathsf{ev\_yes}$$

if there exists a ψ'' s.t. $(\psi'; \mathrm{id}_\Delta) \circ (\psi''; \delta) = (\psi; \delta)$

$$\frac{\Phi \vdash (\psi; \delta) \sim \Omega \hookrightarrow V}{\Phi \vdash (\psi; \delta) \sim (\Omega, (\Psi \triangleright \psi' \mapsto P)) \hookrightarrow V}\ \mathsf{ev\_no}$$

Fig. 4. Operational semantics of $\mathcal{M}_2^+$

Theorem 1 (Type-preservation).

1. *If* $\Phi \vdash P \hookrightarrow V$ *and* $\Phi; \cdot \vdash P \in F$ *then* $\Phi; \cdot \vdash V \in F$
2. *If* $\Phi \vdash D \hookrightarrow \psi; \delta$ *and* $\Phi; \cdot \vdash D \in \Psi; \Delta$ *then* $\Phi; \cdot \vdash (id_\Phi, \psi; \delta) : (\Phi, \Psi; \Delta)$
3. *If* $\Phi \vdash (\psi; \delta) \sim \Omega \hookrightarrow V$ *and* $\mathcal{F} :: \Phi; \cdot \vdash (\psi; \delta) : (\Psi; \Delta)$ *and* $\Psi; \Delta \vdash \Omega \in F$ *then* $\Phi; \cdot \vdash V \in F[\psi]$

The proof goes by induction over the different evaluation derivations, applying various substitution properties. See [Sch00] for details.

6 Meta Logic $\mathcal{M}_2^+$

In this section we discuss $\mathcal{M}_2^+$ from a logical perspective. So far, $\mathcal{M}_2^+$ is a system of partial functions, but in order to interpret them as proofs, we must ensure that they are realizers, i.e. they are total functions. For the purpose of this section we restrict ourselves to the Π_2-fragment of $\mathcal{M}_2^+$, and require that all formulas are of the form $\forall x_1 : A_1 \ldots \forall x_n : A_n. \exists y_1 : A'_1. \exists y_m : A'_m. \top$.

In order to transform $\mathcal{M}_2^+$ into a system of realizers, we augment the typing system from Section 4 with two side conditions. Because of space limitations we cannot give the corresponding syntactic criteria in this paper therefore we must stay informal. The first side condition enforces termination of a proof term P by ensuring that the argument vector to each occurrence of $\mathbf{x}$ in P decreases according to some well-founded ordering. Our termination criterion is very similar to the one already proposed by [RP96]. The second side condition enforces that any Ω occurring anywhere in a proof term covers all possible cases. More precisely, for any instantiation $(\eta; \delta)$ of $(\Psi'; \Delta')$ $(\Phi; \cdot \vdash \eta; \delta \in \Psi'; \Delta')$ there exists at least one case $(\Psi'' \triangleright \psi \mapsto P)$ in Ω and a new substitution $(\eta'; \delta')$ s.t.

$$\Phi; \cdot \vdash (\psi'; \mathrm{id}_\Delta) \circ (\eta'; \delta') = (\eta; \delta) \in \Psi'; \Delta'$$

The coverage criterion employs techniques such as most general unifiers inspired by ideas in the formulation of the case rule in [SP98].

The proof that the corresponding syntactic criteria suffice to guarantee that all functions are realizers is based on a refinement of the operational semantics from Section 5. In [Sch00] we introduce a trace-based small-step semantics, for which we show that the execution of any function satisfying the syntactic criteria always makes progress until it eventually terminates. Thus, $\mathcal{M}_2^+$ is a calculus of realizers.

Theorem 2 (Totality). *For every program* P *that satisfies* $\Phi; \cdot \vdash P \in F$ *there exists a value* V *such that* $\Phi \vdash P \hookrightarrow V$ *and* $\Phi; \cdot \vdash V \in F$.

Moreover, by ascribing the following natural meaning to formulas in $\mathcal{M}_2^+$, we can view $\mathcal{M}_2^+$ as a logic.

$$\begin{array}{ll} \Phi \models \forall x : A.\, F \text{ iff } \Phi \models F[M/x] & \text{for any } M, \text{ s.t. } \Phi \vdash M : A \\ \Phi \models \mathbf{\Pi}\rho^L.\, F \text{ iff } \Phi \models F[\rho'/\rho] & \text{for some } \rho'^L \in \Phi, \text{ s.t. } \Phi \vdash \rho' \equiv \rho \end{array}$$

$$\Phi \models \exists x : A.\, F \text{ iff } \Phi \models F[M/x] \qquad \text{for some } M, \text{ s.t. } \Phi \vdash M : A$$
$$\Phi \models F_1 \wedge F_2 \text{ iff } \Phi \models F_1 \text{ and } \Phi \models F_2$$
$$\Phi \models \top$$

Every function in this version of $\mathcal{M}_2^+$ is a realizer, and therefore we conclude that under the regular world assumption its interpretation as logic is sound. The proof goes by induction over the structure of formulas and uses Theorem 2.

Theorem 3 (Soundness of $\mathcal{M}_2^+$).
If $\Phi; \cdot \vdash V \in F$ and V is a value then $\Phi \models F$.

7 Conclusion

We have presented the meta logic $\mathcal{M}_2^+$, which takes full advantage of the regular world assumption. Using the appropriate side conditions on the typing rules, $\mathcal{M}_2^+$ is a type system for realizers, its functions range over arbitrary LF types. It inherits its expressive power from LF, especially from properties associated with hypothetical judgments, such as substitution, weakening, contraction, and exchange. The theorem prover implemented in the Twelf system [PS99] is based on a proof search algorithm for realizers in $\mathcal{M}_2^+$.

From an experimental point of view, Twelf is automatic, and it proves Lemma 1 in less than 1 sec. Among many other examples, Twelf proves the Church-Rosser theorem and various cut-elimination theorems for first-order classical and intuitionistic logic. Since $\mathcal{M}_2^+$ is a conservative extension over $\mathcal{M}_2$, it continues to prove all examples from [SP98].

In future work, we will investigate extensions of $\mathcal{M}_2^+$ to other logical frameworks supporting subtyping and linearity. We believe that this is a worthwhile endeavor because it permits reasoning about resource oriented logics and programming languages with state, and we are convinced that it is feasible because McDowell and Miller [MM97] successfully encoded fragments of second-order linear logic into their meta-logic $FO\lambda^{\Delta\mathbb{N}}$.

Acknowledgment: I want to thank my advisor Frank Pfenning for all his contributions to this work.

References

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

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

[DPS97] Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In R. Hindley, editor, *Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, pages 147–163, Nancy, France, April 1997. Springer-Verlag LNCS. An extended version is available as Technical Report CMU-CS-96-172, Carnegie Mellon University.

[Gal93] Jean Gallier. On the correspondence between proofs and λ-terms. Cahiers du centre de logique, Université Catholique de Louvain, January 1993.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935.

[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.

[Hof99] Martin Hofmann. Semantical analysis for higher-order abstract syntax. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 204–213, Trento, Italy, July 1999. IEEE Computer Society Press.

[Lel98] Pierre Leleu. *Induction et Syntaxe Abstraite d'Ordre Supérieur dans les Théories Typées*. PhD thesis, Ecole Nationale des Ponts et Chaussees, Marne-la-Vallee, France, December 1998.

[Mil90] Dale Miller. An extension to ML to handle bound variables in data structures: Preliminary report. In *Proceedings of the Logical Frameworks BRA Workshop*, Nice, France, May 1990.

[MM97] Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax: An extended abstract. In Glynn Winskel, editor, *Proceedings of the Twelfth Annual Symposium on Logic in Computer Science*, pages 434–445, Warsaw, Poland, June 1997.

[Ode94] Martin Odersky. A functional theory of local names. In *Proceedings of 21st Annual Symposium on Principles of Programming Languages*, pages 48–59, Portland, Oregon, January 1994. ACM Press.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 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.

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

[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.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

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

[Sch00] Carsten Schürmann. *Automating the Meta-Theory of Deductive systems*. PhD thesis, Carnegie-Mellon University, 2000. forthcoming.

[SP98] Carsten Schürmann and Frank Pfenning. Automated theorem proving in a simple meta-logic for LF. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction (CADE-15)*, pages 286–300, Lindau, Germany, July 1998. Springer-Verlag LNCS 1421.