Recursion for Higher-Order Encodings

Carsten Schürmann *

Yale University
Department of Computer Science
New Haven, CT 06511
carsten@cs.yale.edu

Abstract. This paper describes a calculus of partial recursive functions that range over arbitrary and possibly higher-order objects in LF [HHP93]. Its most novel features include recursion under λ-binders and matching against dynamically introduced parameters.

1 Introduction

Logical frameworks are meta-languages that are designed *to represent* deductive systems [Pfe99], but not all provide the functionality necessary to manipulate those representations. For the class of logical frameworks based on inductive definitions such as in Isabelle/HOL [Pau94] and Coq [DFH$^+$93,PM93] the connection to programming has been thoroughly studied and is well understood. However it is not at all well understood for logical frameworks that support true higher-order encodings such as LF [HHP93]. In this paper we study this particular connection and present a calculus of recursive functions that manipulate higher-order, dependently typed encodings.

LF allows concise and elegant encodings of many inference systems including their side conditions, such as natural deduction, sequent calculi, type systems, operational semantics, compilers, etc. It draws its expressive power from dependent types together with higher-order representation techniques both of 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 also to express functions about various inference systems.

Unfortunately, those higher-order representation techniques generally clash with common programming techniques. The problem arises as a side effect of the higher-order nature of encodings because recursive calls may have to traverse λ-binders. This stands in direct conflict with the central requirement of standard inductive types. The positivity condition [PM93] rules out most datatype definitions that seem natural in LF. Carelessly mixing higher-order representation

* 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.

techniques with function definition by cases and recursion triggers the problem of exotic terms which undermines the adequacy of encodings.

In functional programming languages, arguments to functions are typically closed and therefore functions need not be defined on free parameters. We call this property the *closed world assumption*. In fact, the positivity condition implies the closed world assumption, and is built into all logical frameworks based on inductive definitions such as Coq and Isabelle/HOL. However, when programming with higher-order encodings, the closed world assumption is too restrictive.

Higher-order encodings make frequent use of abstraction as a means to represent variable binding. For example, in a higher-order logical framework such as LF or the simply typed λ-calculus, the identity function '`fn` $x \Rightarrow x$' may be represented as 'lam $(\lambda x.\, x)$' for some constant lam. Under the closed world assumption, however, there is no reasonable scheme for recursion and function definition by cases because recursive calls may have to traverse λ-binders.

In this paper, we propose a solution to the tension between closed worlds and higher-order encodings. We propose to weaken the closed world assumption and allow arguments to be open and depend on parameters. Worlds must not be arbitrary; on the contrary they must be regularly formed in order to allow function extensions by those new cases that match parameters. We call this property the *regular world assumption* and it guides our design of a calculus of partial recursive functions for higher-order encodings which we call $\mathcal{T}_\omega^+$. That $\mathcal{T}_\omega^+$ can be restricted to a meta-logic for LF via a realizability interpretation of proofs is an entirely orthogonal issue and is studied in another forthcoming paper.

This paper is organized as follows. In Section 2 we discuss the technique of higher-order representation, demonstrate its advantages, and compare it to alternative representation techniques. In Section 3 we motivate and analyze the regular world assumption by the means of a function that embeds natural deduction derivations into the sequent calculus. We then describe the type system $\mathcal{T}_\omega^+$ in Section 4, whose elements are partial functions that range over possibly open LF objects. $\mathcal{T}_\omega^+$'s operational semantics is specified in Section 5, and its meta-theory is summarized in Section 6. A sample application is given in Section 7.

2 Higher-Order Encodings

As a motivating example, consider the standard formulation of the calculus of natural deductions [Gen35] for the implicational fragment of propositional logic depicted in Figure 1. The rules define the judgment "*G is true*" ($\Gamma \vdash G$). We take the freedom to assign names to hypotheses $\Gamma ::= \cdot \mid \Gamma, u :: G$. There are many possible solutions of how to represent natural deduction derivations formally and we discuss some of them below. We argue that the representation in a dependently typed, higher-order type theory, such as LF [HHP93] is the most natural. We write $\ulcorner\cdot\urcorner$ for the representation function and begin with the

$$\frac{}{\Gamma, u :: G \vdash G}\, u \qquad \frac{\Gamma, u :: G_1 \vdash G_2}{\Gamma \vdash G_1 \supset G_2} \supset \mathrm{I}^u \qquad \frac{\Gamma \vdash G_1 \supset G_2 \qquad \Gamma \vdash G_1}{\Gamma \vdash G_2} \supset \mathrm{E}$$

Fig. 1. Natural deduction calculus

straightforward representation of formulas.

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

'imp' is used as an infix operator throughout this paper. In LF, judgments are represented as types, and derivations as objects. Following standard practice [Pfe91] we omit all implicit Π-abstractions from types and we take $\beta\eta$-conversion as the notion of definitional equality [Coq91].

Example 1 (Representation of Figure 1 in LF).

$$\begin{array}{ll} \mathrm{nd} & : \mathrm{o} \to \mathrm{type} \\ \mathrm{impi} & : (\mathrm{nd}\; G_1 \to \mathrm{nd}\; G_2) \to \mathrm{nd}\; (G_1 \;\mathrm{imp}\; G_2) \\ \mathrm{impe} & : \mathrm{nd}\; (G_1 \;\mathrm{imp}\; G_2) \to \mathrm{nd}\; G_1 \to \mathrm{nd}\; G_2 \end{array}$$

$\Gamma \vdash A$ is a hypothetical judgment since the premiss to '$\supset$ I' in Figure 1 discharges the hypothesis u. A good choice for the representation of Γ is the LF context itself, because then all (admissible) structural rules including 'u', weakening, contraction, and exchange, are inherited from LF and need not to be encoded individually. Consequently, hypothetical judgments are represented as functions, and thus the premiss of $\supset$ I corresponds to an object of type (nd $G_1 \to$ nd G_2). The substitution operation for hypotheses may be encoded as a simple LF β-redex. The representation of natural deduction derivations in LF by 'impi' and 'impe' is adequate, i.e. they are in one-to-one correspondence with their canonical forms [HHP93].

Despite its elegance, the encoding of the natural deduction calculus in LF has one fundamental drawback. It is not inductive in the sense of [PM93]. 'nd G_1' in the type of 'impi' violates the positivity condition, and thus recursive functions cannot be defined by cases on an argument of type 'nd G'. It is the main contribution of this paper to resolve this seeming contradiction.

There are many other logical frameworks that represent natural deductions while supporting function definition by cases, however many of these encodings are less immediate. For example, the encoding from Example 1 can be turned into an inductive type by a technique proposed by Despeyroux et al. [DFH95]. This technique enforces the positivity condition by replacing the negative occurrence 'nd G_1' in 'impi' by a new parameter type 'var G_1' at the expense of the advantages that come with higher-order abstract syntax. Structural rules and substitution are no longer directly supported.

The modal λ-calculus [DPS97,DL99,Hof99] supports true higher-order encodings and function definition by iteration. Its version of iteration however is quite rigid and sometimes poses problems when applying weakening and substitution lemmas or using dependent types.

The programming language FreshML [GP99,PG00] imports higher-order abstract syntax into a functional programming setting and takes care of α-renaming internally. However, neither dependent types nor substitutions are directly supported by FreshML, and must therefore be programmed explicitly.

The meta-logic $\mathcal{M}_2$ is a direct predecessor of this work [SP98]. It supports higher-order encodings, but it does not allow recursion to traverse λ-binders. Consequently, $\mathcal{M}_2$'s expressiveness is limited and those functions that motivate this work are simply not expressible.

The calculi of partial inductive definitions [Hal87] and definitional reflection [SH93] are purely logical systems. Even when interpreted functionally, their proofs require all arguments to be closed and recursion must not traverse λ-binders. Figure 1 can also be encoded in a logical framework implemented in the meta-logic $FO\lambda^{\Delta I\!N}$ [MM97]. $FO\lambda^{\Delta I\!N}$, however, is a sequent calculus, and does not support functions to range over those encodings.

Miller [Mil90] suggests to extend ML datatypes by a parameter mechanism to support higher-order abstract syntax. New parameters can be dynamically generated and (recursive) function dynamically extended by new cases, however dependent types are not supported.

The representation of natural derivations in Example 1 is concise and elegant and it leverages off properties inherent to LF. In this paper we design a type system $\mathcal{T}_\omega^+$ of partial recursive functions that range over higher-order encodings. The functions are definable by case analysis and recursion.

3 The Regular World Assumption

As running example we use a mapping of natural deduction derivations into derivations of the sequent calculus. Although we present only one connective, the example scales to full first-order logic. The judgment for sequent derivations is $\Gamma \Longrightarrow G$, its rules and its adequate encoding are depicted in Figure 2 [Pfe95]. Hypotheses to the left of the sequent symbol '$\Longrightarrow$' are encoded using 'hyp' in order to distinguish them from the conclusion to the right which is encoded using 'conc'. Only by representing them as two separate type families we can guarantee the adequacy of the encoding which also relies on the fact that the hard-wired structural properties of hypotheses such as weakening and contraction are implicit in the encoding in LF.

Gentzen has shown in [Gen35] that such a mapping from natural deductions $\mathcal{D}$ into sequent derivations $\mathcal{E}$ exists. This algorithm is depicted in the left column of Figure 3, and we refer to it as Gentzen's algorithm. Because natural deductions and sequent derivations are represented differently, we let $\Gamma_u = u_1 :: G_1, \ldots, u_n :: G_n$ be the natural deduction context, and $\Gamma_h = h_1 :: G_1, \ldots, h_n :: G_n$ be the sequent context.

Intuitively, Gentzen's algorithm should be expressible as a recursive function that maps natural deduction derivations $\ulcorner \mathcal{D} \urcorner$ to sequent derivations $\ulcorner \mathcal{E} \urcorner$ where

$$\frac{}{\Gamma, h :: G \Longrightarrow G}\,\text{init} \qquad \frac{\Gamma \Longrightarrow G_1 \qquad \Gamma, h :: G_1 \Longrightarrow G_2}{\Gamma \Longrightarrow G_2}\,\text{cut}^h \qquad \frac{\Gamma, h :: G_1 \Longrightarrow G_2}{\Gamma \Longrightarrow G_1 \supset G_2}\supset\!\text{R}^h$$

$$\frac{\Gamma, h :: G_1 \supset G_2 \Longrightarrow G_1 \qquad \Gamma, h :: G_1 \supset G_2, h_2 :: G_2 \Longrightarrow G_3}{\Gamma, h :: G_1 \supset G_2 \Longrightarrow G_3}\supset\!\text{L}^{h_2}$$

$$\begin{array}{ll}
\text{hyp} & : \text{o} \rightarrow \text{type} \\
\text{conc} & : \text{o} \rightarrow \text{type} \\
\\
\text{init} & : \text{hyp } G \rightarrow \text{conc } G \\
\text{cut} & : \text{conc } G_1 \rightarrow (\text{hyp } G_1 \rightarrow \text{conc } G_2) \rightarrow \text{conc } G_2 \\
\text{impR} & : (\text{hyp } G_1 \rightarrow \text{conc } G_2) \rightarrow \text{conc } (G_1 \text{ imp } G_2) \\
\text{impL} & : \text{conc } G_1 \rightarrow (\text{hyp } G_2 \rightarrow \text{conc } G_3) \rightarrow (\text{hyp } (G_1 \text{ imp } G_2) \rightarrow \text{conc } G_3)
\end{array}$$

Fig. 2. Sequent Calculus and its Representation in LF

$\ulcorner \mathcal{D} \urcorner$ and $\ulcorner \mathcal{E} \urcorner$ satisfy the invariant:

$$u_1 : \text{nd} \ulcorner G_1 \urcorner, \ldots, u_n : \text{nd} \ulcorner G_n \urcorner \vdash^{\Pi} \ulcorner \mathcal{D} \urcorner : \text{nd} \ulcorner G \urcorner$$
$$h_1 : \text{hyp} \ulcorner G_1 \urcorner, \ldots, h_n : \text{hyp} \ulcorner G_n \urcorner \vdash^{\Pi} \ulcorner \mathcal{E} \urcorner : \text{conc} \ulcorner G \urcorner$$

Here, the function must accommodate the fact that $\ulcorner \mathcal{D} \urcorner$ and $\ulcorner \mathcal{E} \urcorner$ are open objects since the LF contexts to the left of the $\vdash^{\Pi}$ symbols are not empty, and therefore violate the so called *closed world assumption*. Non-standard is also the recursive call in **Case 2** which extends the natural deduction context to $\Gamma_u, u :: G_1$. This extension corresponds to the traversal of the λ-binder '$u : \ulcorner G_1 \urcorner$' in LF.

The main contribution of this paper is a characterization of partial recursive functions such as the one described above. To address this situation we propose to change the status quo of the closed world assumption and generalize it to the *regular world assumption*. If we accept this new assumption, arguments to recursive functions do not need to be closed. They may be open in a parameter context which conforms to an a priori specified *regular* grammar, and it is this restriction which makes the construction work.

The form of the regular world for Gentzen's algorithm is best explained by interpreting the invariant associated with the algorithm. It says, that Γ_u and Γ_h may only differ in names, but not in length. Thus, the grammar that defines the regular world for our example results from interleaving Γ_u and Γ_h as follows:

$$u_1 : \text{nd} \ulcorner G_1 \urcorner, h_1 : \text{hyp} \ulcorner G_1 \urcorner, \ldots, u_n : \text{nd} \ulcorner G_n \urcorner, h_n : \text{hyp} \ulcorner G_n \urcorner$$

Now the u_i's and the h_i's always come in pairs and therefore, any valid world can be expressed by the following grammar that defines the *regular world* Φ.

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

A world Φ is a finite LF context that results from repeatedly unfolding the definition of Φ. With each unfolding operation, a new *parameter block* is added

Input: A derivation $\mathcal{D}$ of $\Gamma \vdash G$

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

Compute $\mathcal{E}$ of $\Gamma_h, h :: G \Longrightarrow G$ by init
Return $\mathcal{E}$.

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

Compute $\mathcal{E}'$ of $\Gamma_h, h :: G_1 \Longrightarrow G_2$ by recursion on $\mathcal{D}'$
Compute $\mathcal{E}$ of $\Gamma_h \Longrightarrow G_1 \supset G_2$ by rule $\supset$R on $\mathcal{E}'$
Return $\mathcal{E}$

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

Compute $\mathcal{E}_1$ of $\Gamma_h \Longrightarrow G_1 \supset G_2$ by recursion on $\mathcal{D}_1$
Compute $\mathcal{E}_2$ of $\Gamma_h \Longrightarrow G_1$ by recursion on $\mathcal{D}_2$
Compute $\mathcal{E}_2'$ of $\Gamma_h, h :: G_1 \supset G_2 \Longrightarrow G_1$
by weakening on $\mathcal{E}_2$
Compute $\mathcal{E}_3$ of $\Gamma_h, h :: G_1 \supset G_2, h_2 :: G_2 \Longrightarrow G_2$
by init
Compute $\mathcal{E}_4$ of $\Gamma_h, h :: G_1 \supset G_2 \Longrightarrow G_2$
by $\supset$L on $\mathcal{E}_2'$ and $\mathcal{E}_3$
Compute $\mathcal{E}$ of $\Gamma_h \Longrightarrow G_2$ by cut on $\mathcal{E}_1$ and $\mathcal{E}_4$
Return $\mathcal{E}$.

fun ndseq $\underline{u}$ = (init $\underline{h}$)
  | **ndseq** (impi D') =
    let
      new u : nd G_1, h : hyp G_1
      val $(E\ h) =$ **ndseq** $(D'\ u)$
    in
      (impR $(\lambda h.\, E\ h)$)
    end
  | **ndseq** (impe $D_1\ D_2$) =
    let
      val $(E_1) =$ **ndseq** D_1
      val $(E_2) =$ **ndseq** D_2
    in
      (cut E_1 $(\lambda h_1.\,$impL E_2
          $(\lambda h_2.\,$init $h_2)\ h_1))
    end

Fig. 3. Gentzen's algorithm.

to the world. This particular definition of Φ is custom-designed to fit Gentzen's algorithm, recursive functions from other problem domains however will require different Φ's. The results presented in this paper are not linked to one particular definition of Φ, they are valid for any Φ. Another example is given in Section 7.

In this world Φ, there exists a recursive function **ndseq** which is depicted (in informal syntax) in the right column of Figure 3. Parallel to the informal case analysis performed on $\mathcal{D}$ in Gentzen's algorithm, **ndseq** is defined by cases.

The first case of **ndseq** formalizes **Case 1**. It applies in a situation where $\ulcorner\mathcal{D}\urcorner$ is bound to some parameter u in Φ. Automatically, by invariant, there will also be a parameter h (the sibling of u). We use $\underline{u}$ and $\underline{h}$ as variables that range exclusively over parameters from the regular world Φ.

The second case of **ndseq** formalizes **Case 2**. The argument $D' = \ulcorner\mathcal{D}'\urcorner$ is of functional type 'nd $G_1 \to$ nd G_2', and therefore **ndseq** must traverse the λ-binder before it recurses. It does so by extending the regular world by a new parameter block (indicated by the **new**-notation) u : nd G_1, h : hyp G_1. The result of the recursive call on $(D'\ u)$ yields a sequent derivation $(E'\ h)$ which is parametric in h. One more application of 'impR' forms the return object.

The third case formalizes **Case 3** and simply returns an appropriate LF object after recursing on D_1 and D_2. Note, that in constructing this result object all appeals to weakening lemmas are automatic since they are inherited from LF. In other examples, appeals to substitution lemmas are formalized equally elegantly.

The regular world assumption provides many new concepts and advantages. It generalizes recursive functions to higher-order encodings *without* an explicit positivity condition [PM93]. It permits arguments passed to recursive functions to be open. It gives recursive calls the freedom to traverse λ-binders. It allows recursive function to define cases for parameters. It is elegant in that recursive functions can be formulated concisely and directly which are not so easy to write in modal λ-calculi [DPS97,DL99,Hof99]. And finally, it is compatible with dependent types.

4 Type System $\mathcal{T}_\omega^+$

We begin now with the formal presentation of the type system $\mathcal{T}_\omega^+$ which supports function definition by recursion as used above for the definition of Gentzen's algorithm **ndseq**. A detailed presentation of $\mathcal{T}_\omega^+$ and its meta-theory can be found in the author's thesis [Sch00]. $\mathcal{T}_\omega^+$'s recursive functions range over higher-order LF objects and can therefore take advantage of all properties which are associated with hypothetical judgments, such as, substitutions, weakening, contraction, and exchange.

$\mathcal{T}_\omega^+$'s type system is simple. It provides a dependent function type $\forall$, a dependent product type $\exists$, and a unit type $\top$. $\mathcal{T}_\omega^+$'s function types can contain ω many alternations of $\forall$ and $\exists$ types. The type constructors of $\mathcal{T}_\omega^+$ are inspired by logic, however, for the purpose of this paper we view them solely as type constructors. **ndseq**, for example, has type $\forall G : \mathrm{o}. \forall D : \mathrm{nd}\ G. \exists E : \mathrm{conc}\ G. \top$.

That **ndseq** in Figure 3 expects only one argument is due to the fact that the first argument can be inferred from the second, and is therefore omitted when using informal syntax. **ndseq**'s type states that in any regular world Φ, for every valid formula $G : \mathrm{o}$ and for every valid natural deduction derivation D ($\Phi \overset{\Pi}{\vdash} D : \mathrm{nd}\ G$), the execution of **ndseq** applied to G and D might or might not terminate normally. However, if it terminates normally, then **ndseq** has computed an instance of type $\exists E : \mathrm{conc}\ G. \top$ and the left projection yields a valid LF object representing a sequent derivation in Φ ($\Phi \overset{\Pi}{\vdash} E : \mathrm{conc}\ G$). For the purpose of this paper, we assume the specification of Φ to be fixed, as we do, for example, with the specification of the LF signature. $\forall$ and $\exists$ types are designed to bind variables whose structure can be analyzed by case analysis.

$$\mathcal{T}_\omega^+ \textit{ types: } F ::= \forall x : A. F \mid \exists x : A. F \mid \top$$

Here, A ranges over LF types [HHP93]. The $\mathcal{T}_\omega^+$ type system can be easily extended by product types $F_1 \wedge F_2$ which we omit because of space constraints.

4.1 Subordination

Recursive functions in $\mathcal{T}_\omega^+$ are executed in regular worlds. During execution, they may extend the current world by new parameters. Thus, the world grows with each traversal of a λ-binder before a recursive call and it shrinks upon return when parameters are discharged. In general, a returned object may need not depend on all but some of the parameters to be discharged. In the second case of **ndseq**, for example, executing $(D'\ u)$ yields an E which may only depend on h but not on u.

That E does not depend on u is justified by the way sequent derivations are constructed. They are simply not defined in terms of natural deduction derivations. Virga [Vir99] proposed a technique that decides if a parameter can possibly occur in an object of a given type or not. His procedure inspects the LF signature and derives the so called "subordination relation". A type A is said to be subordinate to A' ($A \prec A'$), if objects of type A may occur within objects of type A'.

Lemma 1 (Strengthen). *If* $\Gamma, x : A \overset{\Pi}{\vdash} M : A'$ *and* $A \not\prec A'$ *then* $\Gamma \overset{\Pi}{\vdash} M : A'$.

Indeed, because 'hyp' $\prec$ 'conc', and 'nd' $\not\prec$ 'conc' it follows from Lemma 1 that E cannot depend on $u : \mathrm{nd}\ G_1$, and thus $E : \mathrm{hyp}\ G_1 \rightarrow \mathrm{conc}\ G_2$. In general, if a return object has type A_2 before discharging parameters, then it has type '$\mathrm{abs}\ \Gamma.\, A_2$' after the discharge takes place.

Definition 1 (Abstraction).

$$\mathrm{abs}\ \Gamma.\, A_2 = \begin{cases} A_2 & \textit{if } \Gamma = \cdot \\ \mathrm{abs}\ \Gamma'.\, A_2 & \textit{if } \Gamma = \Gamma', x : A_1 \textit{ and } A_1 \not\prec A_2 \\ \Pi x : A_1.\, (\mathrm{abs}\ \Gamma'.\, A_2) & \textit{if } \Gamma = \Gamma', x : A_1 \textit{ and if } A_1 \prec A_2 \end{cases}$$

In the example '$\mathrm{hyp}\ G_1 \rightarrow \mathrm{conc}\ G_2 = \mathrm{abs}\ (u : \mathrm{nd}\ G_1, h : \mathrm{hyp}\ G_1).\, (\mathrm{conc}\ G_2)$' is the type of E after parameter discharge. Lemma 1 guarantees that no parameters escape their scope. Similarly, on the object level, abstraction must bind each parameter it discharges if there is any chance that the parameter occurs free in the object.

Definition 2 (Raising). *Let* M *be of type* A_2.

$$\mathrm{raise}\ \Gamma.\, M = \begin{cases} M & \textit{if } \Gamma = \cdot \\ \mathrm{raise}\ \Gamma'.\, M & \textit{if } \Gamma = \Gamma', x : A_1 \textit{ and } A_1 \not\prec A_2 \\ \lambda x : A_1.\, (\mathrm{raise}\ \Gamma'.\, M) & \textit{if } \Gamma = \Gamma', x : A_1 \textit{ and } A_1 \prec A_2 \end{cases}$$

By a simple inductive argument we can show that this design is sound.

Lemma 2 (Soundness).

1. *If* $\Gamma, \Gamma' \overset{\Pi}{\vdash} A : \textit{type}$ *then* $\Gamma \overset{\Pi}{\vdash} \mathrm{abs}\ \Gamma'.\, A : \textit{type}$.
2. *If* $\Gamma, \Gamma' \overset{\Pi}{\vdash} M : A$ *then* $\Gamma \overset{\Pi}{\vdash} \mathrm{raise}\ \Gamma'.\, M : \mathrm{abs}\ \Gamma'.\, A$.

4.2 The core of $\mathcal{T}_\omega^+$

$\mathcal{T}_\omega^+$'s functions use two kinds of variable binding, and their use is demonstrated by **ndseq** in Figure 3. First, there are variables x that bind arbitrary LF objects M valid in any well-formed regular world Φ. These variables are used, for example, in the patterns of the second and third case. In general, they are bound by the $\forall$ and $\exists$ quantifiers and any instantiation may be analyzed by cases.

Second, there are *parameter variables* $\underline{u}$ and $\underline{h}$ as used in the first case of **ndseq**. They are designed exclusively to range over parameter blocks. In this sense, they behave more like constructors than variables, and yet, they are variables because they bind concrete parameter occurrences declared in Φ at runtime. The fact that parameters are always organized in blocks is also reflected in the formal system of $\mathcal{T}_\omega^+$. Parameter variables never occur alone, they only occur as compound *variable blocks* $\rho = (\underline{u} : \text{nd } G, \underline{h} : \text{hyp } G)$. Variable blocks can only range over parameter blocks declared in Φ, and binding is defined componentwise. Although our informal syntax of $\mathcal{T}_\omega^+$ functions used in Figure 3 hides variable blocks, they are important in the formal exposition.

In our example, Φ is defined in terms of one block, but in the general case it can be defined in terms of several [Sch00]. We distinguish each of them by labels, and assign the label L to the one block in our example.

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

Variable blocks ρ^L are indexed by the appropriate label L. Ψ is a *context* that denotes not only the set of free LF-variables, but also what is know about the regular world.

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

The context $D : \text{nd } G_1, (\underline{u} : \text{nd } G_2, \underline{h} : \text{hyp } G_2)^L$, for example, expresses that D may be instantiated with an object of type 'nd G_1', valid in a regular world Φ and that $(\underline{u} : \text{nd } G_2, \underline{h} : \text{hyp } G_2)^L$ is bound to one parameter block in this world. The validity judgments for contexts extend the standard definition of valid LF contexts in a straightforward way.

$\mathcal{T}_\omega^+$ is a two level system, where x and $\underline{x}$ are variables ranging over LF objects and parameters, respectively. Variables on the meta-level are separate, they are called *meta-variables* $\mathbf{x}$ and range over $\mathcal{T}_\omega^+$-objects of type F. The $\mathbf{x}$'s form a *meta-context* Δ on the meta-level.

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

The syntax we have used to present **ndseq** in Figure 3 contains syntactic sugar in order to make this material accessible. Its desugared version (given in the right column of Figure 6) however is less intuitive and requires several mutually dependent syntactic categories including programs and declarations. These programs and declarations resemble the proof terms of an intuitionistic sequent calculus with the important addition of the $\nu\rho^L. D$ declaration, which — operationally speaking — extends the world by new parameters before evaluating D.

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

The syntactic category of programs is extended below to accommodate case analysis and recursion. For the sake of brevity we omit the straightforward desugaring algorithm that maps our informal syntax into $\mathcal{T}_\omega^+$.

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 and for declarations we write

$$\frac{(\mathbf{x} \in F) \text{ in } \Delta}{\Psi; \Delta \vdash \mathbf{x} \in F}\,\mathsf{axvar} \qquad \frac{}{\Psi; \Delta \vdash \langle\rangle \in \top}\,\mathsf{P}\top \qquad \frac{\Psi; \Delta \vdash D \in \Psi' \qquad \Psi, \Psi'; \Delta \vdash P \in F}{\Psi; \Delta \vdash \mathbf{let}\ D\ \mathbf{in}\ P \in F}\,\mathsf{sel}$$

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

$$\frac{}{\Psi; \Delta \vdash \cdot \in \cdot}\,\mathsf{D}\cdot \qquad \frac{\Psi, \rho^L; \Delta \vdash D \in \Psi' \qquad \text{abs } \rho.\, \Psi' = \Psi''}{\Psi; \Delta \vdash \nu\ \rho^L.\, D \in \Psi''}\,\mathsf{D}\nu$$

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

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

Fig. 4. Typing Rules of the Core of $\mathcal{T}_\omega^+$

$\Psi; \Delta \vdash D \in \Psi'$. From an operational point of view, all variable declarations in Ψ' will be instantiated and returned after D is computed. We use '$\in$' as typing symbol for $\mathcal{T}_\omega^+$ in order to clearly distinguish it from the ':' for LF.

$$\begin{array}{ll} \textit{Typing programs:} & \Psi; \Delta \vdash P \in F \\ \textit{Typing declarations:} & \Psi; \Delta \vdash D \in \Psi' \end{array}$$

The typing rules for these judgments are mutually dependent and are given in Figure 4. We omit the validity judgment for Ψ from the rules axvar, $\mathsf{P}\top$, and $\mathsf{D}\cdot$. The premiss $\Psi \vdash M : A$ of the rules $\mathsf{P}\exists$ and $\mathsf{D}\forall$ is the standard LF typing judgment. It allows M to depend on free variables and parameters declared in Ψ. The $\mathsf{D}\nu$ rule specifies how new parameter blocks are inserted into the regular world and how they are discharged thereafter. The premiss 'abs $\rho.\Psi' = \Psi''$' of the $\mathsf{D}\nu$ rule shifts all types in Ψ' componentwise (see Definition 1).

4.3 Case Analysis and Recursion

The design of $\mathcal{T}_\omega^+$'s case analysis mechanism is non-standard. Because of dependencies, variables can occur non-linearly within the types of other variables. Thus case analysis is not a local operation that effects only one variable but can simultaneously effect others as well. In Gentzen's algorithm (Figure 3), for example, matching against the declaration D determines the form of the formula G in the 'impi'-case. Therefore, case analysis must simultaneously distinguish cases on *all* variables declared in Ψ.

To this effect, our design employs a substitution (or environment) η as case subject. It enables us to consider cases over all objects stored in η simultaneously. Similarly, patterns ψ that trigger each case are also substitutions, and thus the matching problem "if ψ matches η" reduces to the question "if ψ is a *more general* substitution than η". We define that ψ *matches* η iff there exists an η', such that $\eta = \psi \circ \eta'$.

$$\frac{\Psi; \Delta, \mathbf{x} \in F \vdash P \in F}{\Psi; \Delta \vdash \mu\mathbf{x} \in F.\, P \in F}\,\mathsf{fix} \qquad \frac{\Psi \vdash \eta \in \Psi' \qquad \Psi'; \Delta' \vdash \Omega \in F}{\Psi; \Delta'[\eta] \vdash \mathrm{case}\ \eta\ \mathrm{of}\ \Omega \in F[\eta]}\,\mathsf{case}$$

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

Fig. 5. Case Analysis and Recursion in $\mathcal{T}_\omega^+$.

Environments η and patterns ψ are substitutions with domain Ψ. They substitute objects M for variables x and variable blocks ρ' for variable blocks ρ. We write $\Psi' \vdash \psi \in \Psi$ if ψ is valid.

$$\textit{Object-substitutions: } \psi ::= \cdot \mid \psi, M/x \mid \psi, \rho'/\rho$$

The choice of a substitution η as case subject has another advantage namely that each branch remains invariant under substitution application. If substitution σ is applied to a case program, it is absorbed by simple substitution composition. The new case subject is $\eta \circ \sigma$. Each individual case is therefore closed and defined in terms of the pattern ψ, its co-domain Ψ, and a body P. $\mathcal{T}_\omega^+$ also provides a recursion operator μ with the standard static semantics.

$$\begin{array}{ll} \textit{Programs: } P & ::= \ldots \mid \mathrm{case}\ \eta\ \mathrm{of}\ \Omega \mid \mu\mathbf{x} \in F.\, P \\ \textit{Cases: } \quad \Omega & ::= \cdot \mid (\Omega \mid (\Psi \triangleright \psi \mapsto P)) \end{array}$$

The rules defining case analysis and recursion can be found in Figure 5. The premisses of the case rule ensure that the subject of the case program is well-typed as well as Ω. The latter requires a new judgment: $\Psi; \Delta \vdash \Omega \in F$. For the sake of brevity, we have omitted a strictness side condition on alt, that guarantees that matching against ψ instantiates all variables declared in Ψ'.

Figure 6 shows the definition of **ndseq** (from Figure 3) on the left in informal syntax and on the right in $\mathcal{T}_\omega^+$ syntax. For improved readability we omit the type annotations from declarations.

5 Operational Semantics of $\mathcal{T}_\omega^+$

The rules defining the operational semantics are constructed in such a way, that once evaluation is invoked in a regular world Φ its result is well-defined in exactly the same world. During evaluation, new parameters can be dynamically introduced and discharged. 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 indexed by the world Φ.

$$\begin{array}{ll} \text{Program evaluation:} & \Phi \vdash P \hookrightarrow V \\ \text{Declarations evaluation:} & \Phi \vdash D \hookrightarrow \eta \\ \text{Selection:} & \Phi \vdash \eta \sim \Omega \hookrightarrow V \end{array}$$

```
fun ndseq u = (init h)
  | ndseq (impi D') =
      let
        new u : nd G1, h : hyp G1
        val   (E h) = ndseq (D' u)
      in
        (impR (λh. E h))
      end
  | ndseq (impe D1 D2) =
      let
        val (E1) = ndseq D1
        val (E2) = ndseq D2
      in
        (cut E1 (λh1. impL E2
               (λh2. init h2) h1))
      end
```

```
μndseq. ΛG : o. ΛD : nd G.
    case (G/G, D/D)
     of (G : o, (u : nd G, h : hyp G)^L)
            ▷ (G/G, u/D) ↦ (init h)
      | (G1 : o, G2 : o, D' : nd G1 → nd G2)
            ▷(G1 imp G2/G,
                 impi (λu : nd G1. D' u)/D)
            ↦ let
                 ν (u : nd G1, h : hyp G1)^L.
                 x0 = ndseq G2,
                 x1 = x0 (D' u),
                 ⟨E, _⟩ = x1, ·
                in (impR (λh. E h))
      | (G1 : o, G2 : o, D1 : nd (G1 imp G2),
          D2 : nd G1)
            ▷ (G2/G, impe D1 D2/D)
            ↦ let
                 x0 = ndseq (G1 imp G2),
                 x1 = x0 D1,
                 ⟨E1, _⟩ = x1,
                 x2 = ndseq (G1),
                 x3 = x2 D2,
                 ⟨E2, _⟩ = x3, ·
                in (cut E1 (λh1. impL E2
                          (λh2. init h2) h1)))
∈ ∀G : o. ∀D : nd G. ∃E : conc G. ⊤
```

Fig. 6. Gentzen's algorithm in $\mathcal{T}_\omega^+$.

The first judgment relates P with the result of its evaluation. The second returns a list of values in the form of an environment instantiating all variables in Ψ' (the type of D). And the third judgment selects a case from Ω with pattern ψ where η matches ψ and evaluates the respective body to V.

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

The rules defining the operational semantics are given in Figure 7. We comment only on the non-standard rule ev_new since it requires all objects summarized in η' to be abstracted to their new abstract types. This is established by a straightforward generalization of abstraction to substitutions 'raise $\rho.\,\eta'$'.

6 Meta Theory

$\mathcal{T}_\omega^+$ is a type system of partial recursive functions. It is defined with respect to the regular world assumption. If we assume that only well-typed programs, declarations, or cases are executed, we can show that the operational semantics is type preserving. See [Sch00] for a proof.

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* $\Phi \vdash D \hookrightarrow \eta;$ *and* $\Phi; \cdot \vdash D \in \Psi$ *then* $\Phi \vdash \cdot, \eta : \Phi, \Psi$
3. *If* $\Phi \vdash \eta \sim \Omega \hookrightarrow V$ *and* $\mathcal{F} :: \Phi \vdash \eta : \Psi$ *and* $\Psi; \cdot \vdash \Omega \in F$ *then* $\Phi; \cdot \vdash V \in F[\eta]$

Recursive functions in $\mathcal{T}_\omega^+$ are partial. Their execution does not always terminate nor does it always make progress. For example, the execution of '$\mu\mathbf{x} \in F.\, \mathbf{x}$' is non-terminating, the execution of 'case η of $\cdot$' gets always stuck for any environment η.

$$\frac{}{\Phi \vdash \Lambda x : A.\, P \hookrightarrow \Lambda x : A.\, P}\,\mathsf{ev\_lam} \qquad \frac{\Phi \vdash D \hookrightarrow \eta \qquad \Phi \vdash P[\eta] \hookrightarrow V}{\Phi \vdash \mathbf{let}\ D\ \mathbf{in}\ P \hookrightarrow V}\,\mathsf{ev\_let}$$

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

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

$$\frac{}{\Phi \vdash \cdot \hookrightarrow \cdot}\,\mathsf{ev\_empty} \qquad \frac{\Phi, \rho^L \vdash D \hookrightarrow \eta' \qquad \text{raise } \rho.\, \eta' = \eta''}{\Phi \vdash \nu\, \rho^L.\, D \hookrightarrow \eta''}\,\mathsf{ev\_new}$$

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

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

$$\frac{\Phi \vdash P[\eta'] \hookrightarrow V \qquad \psi \circ \eta' = \eta}{\Phi \vdash \eta \sim (\Omega, (\Psi \triangleright \psi \mapsto P)) \hookrightarrow V}\,\mathsf{ev\_yes} \qquad \frac{\Phi \vdash \eta \sim \Omega \hookrightarrow V}{\Phi \vdash \eta \sim (\Omega, (\Psi \triangleright \psi \mapsto P)) \hookrightarrow V}\,\mathsf{ev\_no}$$

Fig. 7. Operational semantics of $\mathcal{T}_\omega^+$

7 Applications

In our experience many algorithms related to programming languages and logics can be directly represented in $\mathcal{T}_\omega^+$. In the setting of first-order intuitionistic logic, for example, we have encoded an algorithm that transforms sequent derivations with cut into sequent derivations without cut [Pfe95]. In the interest of space, we can only give the definition of the regular world and the types of the two recursive $\mathcal{T}_\omega^+$ functions. The sequent calculus for full first-order intuitionistic logic extends Figure 2 with additional left and right rules for the other connectives. Let 'i' be the LF type for terms, and Φ a regular world of the form

$$\Phi ::= \cdot \mid \Phi, (u : \text{hyp } G)^{L_1} \mid \Phi, (a : \text{i})^{L_2} \mid \Phi, (p : \text{o})^{L_3}.$$

The block labeled L_1 covers extensions of the world by new hypotheses, L_2 by new term parameters, and L_3 by new formulas. We write 'conc$^-$' for the cut-free sequent calculus. **ca** and **ce**, which implement the admissibility and the elimination of cut

$$\begin{array}{l} \mathbf{ca} \in \forall G : \text{o}.\, \forall H : \text{o}.\, \forall D_1 : \text{conc}^-\ H.\, \forall D_2 : \text{hyp } H \to \text{conc}^-\ G.\, \exists D' : \text{conc}^-\ G.\, \top \\ \mathbf{ce} \in \forall G : \text{o}.\, \forall D : \text{conc } G.\, \exists D' : \text{conc}^-\ G.\, \top \end{array}$$

have direct and elegant implementations in $\mathcal{T}_\omega^+$.

8 Conclusion

We have presented a type system $\mathcal{T}_\omega^+$ whose partial recursive functions range over (possibly higher-order) objects and are defined by cases and recursion. Most importantly, $\mathcal{T}_\omega^+$ allows recursive calls to traverse λ-binders, and function definitions to match parameters. In addition $\mathcal{T}_\omega^+$ supports dependent types and its functions can take direct advantage of all properties that are associated with the representation in LF, e.g. weakening, contraction, and substitution. These features make $\mathcal{T}_\omega^+$ an excellent candidate for the formalization of algorithms over deductive systems that are encoded in the logical framework LF [HHP93] via higher-order abstract syntax and hypothetical judgments.

$\mathcal{T}_\omega^+$'s concept of partial recursive functions is justified on the basis of the regular world assumption. It successfully resolves the tension between higher-order encodings and function definition by cases and recursion. Once specified, the form of regular worlds limits the ways that parameters are introduced during execution, and accounts for the exact form of a parameter block. Conversion procedures from one logic to another, bracket abstraction, and normalization via parallel reduction are only few of the examples that can be directly and cleanly expressed as recursive functions in $\mathcal{T}_\omega^+$.

Acknowledgment: I want to thank my advisor Frank Pfenning for all his contributions to this work, and comments on earlier versions of this draft.

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.

[DFH+93] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.

[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.

[DL99] Joelle Despeyroux and Pierre Leleu. Primitive recursion for higher-order abstract syntax with dependent types. In *Proceedings of the Workshop on Intuitionistic Modal Logics and Applications (IMPL'99)*, Trento, Italy, July 1999.

[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.

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

[GP99] Murdoch Gabbay and Andrew Pitts. A new approach to abstract syntax involving binders. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 214–224, Trento, Italy, July 1999. IEEE Computer Society Press.

[Hal87] Lars Hallnäs. A note on the logic of a logic program. In *Proceedings of the Workshop on Programming Logic*. University of Göteborg and Chalmers University of Technology, Report PMG-R37, 1987.

[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.

[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.

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

[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.

[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.

[Pfe99] Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science Publishers, 1999. In preparation.

[PG00] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction, MPC2000, Proceedings, Ponte de Lima, Portugal, July 2000*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, Heidelberg, 2000.

[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, TLCA'93*, pages 328–345, Utrecht, The Netherlands, March 1993. Springer-Verlag LNCS 664.

[Sch00] Carsten Schürmann. *Automating the Meta-Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, 2000. CMU-CS-00-146.

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

[SP98] Carsten Schürmann and Frank Pfenning. Automated theorem proving in a simple meta-logic for LF. 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.

[Vir99] Roberto Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, 1999. Forthcoming.