Programming With Higher-Order Encodings

Carsten Schürmann
Yale University, New Haven, CT 06520, USA
carsten@cs.yale.edu

November 2, 2000

Abstract

This paper describes the design of the type theory $\mathcal{M}_2^+$ which supports the formalization of algorithms about deductive systems which are prevalent in the areas of programming languages, type theory, and logics. It is based on the logical framework LF [HHP93] that supports higher-order representation techniques such as higher-order abstract syntax and hypothetical judgments. $\mathcal{M}_2^+$ is implemented in the Twelf system [PS99].

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 this article is the observation, that these extensions are not arbitrary, but that they are regular in structure. Therefore, we replace the the closed world assumption by the *regular world assumption* which allows dynamic but regular extensions of types by new constructors and we develop the type theory $\mathcal{M}_2^+$ that supports the formalization of algorithms about inference systems represented in LF under this new regular world assumption. $\mathcal{M}_2^+$ is a

calculus of partial functions. However, if we restrict $\mathcal{M}_2^+$ to *realizers*, i.e. functions that satisfy termination and progress conditions, $\mathcal{M}_2^+$ can be interpreted as a meta-logic. The types of $\mathcal{M}_2^+$ are propositions *about deductive systems*, and the realizer of a type is its proof.

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 used it, for example, to implement realizers to convert derivations between Hilbert's calculus, Gentzen's natural deduction calculus, and the sequent calculus, respectively. Other experiments include an algorithm to normalize λ-expressions which is based on the proof of the Church-Rosser theorem for the untyped and the simply-typed λ-calculus, and algorithms to transform derivations of a various propositional and first-order sequent calculi with a cut rule to derivations without it. These algorithms implement the proof of cut-elimination theorem. Twelf can construct these functions automatically by merely specifying its type, and the variables cases should be distinguished on.

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 type theory $\mathcal{M}_2^+$ is formally presented in Section 4, its operational semantics in Section 5, and its meta-theory in Section 6. 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 2.1 (Formulas)

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

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

'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 2.2 (Natural Deduction)

Provability judgment: $\Gamma \vdash G$ nd : o $\to$ type

$$\frac{}{\Gamma, u :: G \vdash G}\, u$$

$$\frac{\Gamma, u :: G_1 \vdash G_2}{\Gamma \vdash G_1 \supset G_2} \supset \mathrm{I}^u \qquad \text{impi} : (\mathrm{nd}\ G_1 \to \mathrm{nd}\ G_2) \to \mathrm{nd}\ (G_1\ \mathrm{imp}\ G_2)$$

$$\frac{\Gamma \vdash G_1 \supset G_2 \qquad \Gamma \vdash G_1}{\Gamma \vdash G_2} \supset \mathrm{E} \qquad \text{impe} : \mathrm{nd}\ (G_1\ \mathrm{imp}\ G_2) \to \mathrm{nd}\ G_1 \to \mathrm{nd}\ G_2$$

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 $\ulcorner \Gamma, u :: G_1 \urcorner = \ulcorner \Gamma \urcorner, u : \mathrm{nd}\ \ulcorner G_1 \urcorner$ and consequently, the encoding of the first rule does not give rise to a new LF constant because

$$\ulcorner\ \frac{}{\Gamma, u :: G \vdash G}\, u\ \urcorner = u : \mathrm{nd}\ \ulcorner G \urcorner.$$

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 2.3 (Sequent Calculus)

Provability judgment: $\Gamma \Longrightarrow G$

$$\frac{}{\Gamma, h :: G \Longrightarrow G}\,\text{init}$$

$$\frac{\Gamma \Longrightarrow G_1 \qquad \Gamma, h :: G_1 \Longrightarrow G_2}{\Gamma \Longrightarrow G_2}\,\text{cut}$$

$$\frac{\Gamma, h :: G_1 \Longrightarrow G_2}{\Gamma \Longrightarrow G_1 \supset G_2}\,\supset\text{R}$$

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

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

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 2.2 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 → nd G_2) → 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 3.1 (Natural deduction ⇒ 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: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}' \\ \Gamma, u :: G_1 \vdash G_2\end{array}}{\Gamma \vdash G_1 \supset G_2} \supset \mathrm{I}^u$

$\mathcal{E}'$ 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: $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Gamma \vdash G_1 \supset G_2 & \Gamma \vdash G_1\end{array}}{\Gamma \vdash G_2} \supset \mathrm{E}$

$\mathcal{E}_1$ 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 3.2 (Functional interpretation of *Case 3*)

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

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

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

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

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

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.

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

This definition of Φ is custom-designed to fit Gentzen's algorithm. Of course, implementations of other algorithms require different definitions of Φ. Given a canonical LF object $D : \mathrm{nd}\ F$ in a context Φ of this form, we can analyze cases over D since there are three and only three possibilities:

1. $D = \underline{u}$
2. $D = \text{impi}\ (\lambda u : \text{nd}\ G_1.\ D'\ u)$
3. $D = \text{impe}\ D_1\ D_2$

Once the evaluation of **ndseq** encounters that D is bound to $\underline{u}$, **Case 1** of Gentzen's algorithm applies, and because of the regular form of Φ, the existence of a parameter $\underline{h}$ in the context is guaranteed.

Example 3.3 (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.

New parameter pairs are introduced into the context once Gentzen's algorithm is applied to a natural deduction derivation encoded as "impi $(\lambda u : \text{nd}\ G_1.\ D'\ u)$". The appeal to the induction hypothesis in **Case 2**, for example, requires Gentzen's algorithm to assume a natural derivation of $u :: G_1$ which implicitly corresponds to assuming a sequent derivation of $h :: G_1$. Therefore, **ndseq** relies on additional functionality, not present in typical functional programming languages, namely the ability to introduce and to discharge parameters during runtime.

Example 3.4 (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 3.1. Since $D' = \ulcorner \mathcal{D}' \urcorner$ is hypothetical, D' has functional type, and therefore **ndseq** cannot be applied recursively, unless two new parameters (indicated by the **new**-notation [Ode94]) $u : \text{nd}\ G_1, h : \text{hyp}\ G_1$ are introduced. 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. Note, that E' expects only h as argument and not on u because E' is a hypothetical sequent derivation which by definition cannot depend on any natural deduction derivations.

```
Let Φ ::= · | Φ, u : nd ⌜G⌝, h : hyp ⌜G⌝
fun ndseq u = (init h)
  | ndseq (impi D') =
      let
        new u : nd G, h : hyp G
        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
```

Figure 1: An implementation of Gentzen's algorithm

Example 3.5 (An implementation of Gentzen's algorithm) Figure 1 shows an implementation of the proof of Lemma 3.1 in ML-like notation, summarizing results from Example 3.2–3.4. 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 powerful enough to express partial functions like **ndseq** from the previous section. If we restrict $\mathcal{M}_2^+$ to realizers, i.e. functions that satisfy termination and progress properties, $\mathcal{M}_2^+$ can be interpreted as a meta logic. The reader is invited to consult [Sch00] for a detailed presentation of this interpretation of $\mathcal{M}_2^+$.

$\mathcal{M}_2^+$'s design includes two explicit levels. The logical framework 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 $\Phi \stackrel{\Pi}{\vdash} D : \text{nd}\ G$, there exists an LF-object E which satisfies $\Phi \stackrel{\Pi}{\vdash} E : \text{conc}\ G$. For the purpose of this paper, we leave the specification of Φ to the module system, as we already 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.

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

We call these constructs types, although the notation is clearly borrowed from logic. Indeed, when looking at $\mathcal{M}_2^+$ as a system of realizers, the type system can be interpreted as a meta-logic for specifying and verifying properties about deductive systems. The body of the impe-case in Figure 1, 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. Unlike variables that are bound by the universal quantifier $\forall$, parameter variables are place holders for and only for parameters whose structure must not be analyzed. Therefore, $\mathcal{M}_2^+$ provides a new quantifier $\mathbf{\Pi}$ on the meta level that binds exclusively blocks of parameter variables. These blocks conform to the block structure of Φ.

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.

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

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

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

With the new quantifier at hand, a type can be assigned to the first case of **ndseq**.

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

For all parameter blocks $(\underline{u}, \underline{v})$, there exists a derivation of type 'conc G'.

Parameter variables are an important part of the type system $\mathcal{M}_2^+$, and the standard notion of contexts and typing rules have to be extended and generalized. Differently from the standard notion of context, *generalized contexts* declare in addition variable blocks, which stipulate the existence of a specific parameter block in the context during runtime.

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

$$\frac{\Psi, x : A \vdash F \ \text{formula}}{\Psi \vdash \forall x : A.\, F \ \text{formula}}\,\mathsf{V}\forall \qquad \frac{\Psi, \rho^L \vdash F \ \text{formula}}{\Psi \vdash \mathbf{\Pi}\rho^L.\, F \ \text{formula}}\,\mathsf{V}\Pi$$

$$\frac{\Psi, x : A \vdash F \ \text{formula}}{\Psi \vdash \exists x : A.\, F \ \text{formula}}\,\mathsf{V}\exists \qquad \frac{\vdash \Psi \ \text{gctx}}{\Psi \vdash \top \ \text{formula}}\,\mathsf{V}\mathsf{true}$$

Figure 2: Well-formed formulas

A generalized context of the form D : nd $G_1, (\underline{u}$: nd $G_2, \underline{h}$: nd $G_2)^L$, for example, expresses that during runtime, $\underline{u}$ and $\underline{h}$ are instantiated by genuine parameters, and that D is instantiated by a possibly open but well-typed object of appropriate type. The validity judgments for generalized contexts extend the standard definition for well-typed LF contexts in a straightforward way. As judgment we write $\vdash \Psi$ gctx for valid generalized context whose formal definition we omit from this presentation. By omitting labels and parentheses, any generalized context can be easily turned into an LF context. In this paper, we will take advantage of this property at several occasions without being explicit.

In conclusion, the $\mathcal{M}_2^+$ type system has the following form.

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

$\top$ is the unit type. This type system can be easily extended by product types. We write $\Psi \vdash F$ formula for well-formed $\mathcal{M}_2^+$ types which is defined in Figure 2.

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

The syntax we have presented in Figure 1 contains syntactic sugar in order to make this material accessible. The desugared version however is less intuitive and consists of several mutually dependent categories such as programs and declarations. 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 third kind of variable that can bind program code. In summary, we write x for LF-variables, $\underline{x}$ for parameter variables, and $\mathbf{x}$ for program variables. 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.

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

$\mathcal{M}_2^+$'s programs are defined of declarations, and declarations are defined in terms of programs.

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

Functional abstraction on the program level is provided by $\Lambda x : A.\, P$ that binds the variable x in P and by $\lambda\rho^L.\, P$ that binds an entire variable block ρ^L. $\langle M, P\rangle$ is a program of existential type with the witnessing (canonical) LF object M as its first component. And finally, $\langle\rangle$ is a program of unit type $\top$. The let-expression **let** D **in** P **end** binds a list of declarations.

The regular world assumption is reflected in $\mathcal{M}_2^+$ through the ν-declaration that extends the current world by new parameters during runtime and discharges them thereafter. All other declarations are dual to programs. $(\mathbf{x} \in F = P\ M, D)$, for example, stands for the application of a program P to a canonical LF object M where the residual program is being bound to $\mathbf{x}$ before D is executed, and similarly, $(\mathbf{x} \in F = P\ \rho, D)$ stands for the application of a program P to a variable block ρ. $(\langle x : A, \mathbf{y} \in F\rangle = P, D)$ binds the witness object M of an existential to x, and the residual program to $\mathbf{y}$.

Example 4.1 (Introduction of hypotheses) 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.

$$
\begin{array}{l}
\textbf{ndseq}\ (\text{impi}\ D') = \\
\quad \textbf{let} \\
\quad\quad \nu\ (\underline{u} : \text{nd}\ G_1, \underline{h} : \text{hyp}\ G_1)^L\ . \\
\quad\quad \mathbf{x}_0 \in \exists E : \text{conc}\ G_2.\ \top = \textbf{ndseq}\ (D'\ u), \\
\quad\quad \langle E : \text{conc}\ G_2, \mathbf{x}_1 \in \top\rangle = \mathbf{x}_0 \\
\quad \textbf{in} \\
\quad\quad (\text{impR}\ (\lambda h : \text{hyp}\ G_1.\, E\ h)) \\
\quad \textbf{end}
\end{array}
$$

Intuitively, the execution of the ν-declaration extends the natural deduction calculus and the sequent calculus by two *new* hypothesis labeled $\underline{u}$ and $\underline{h}$, which causes $\mathbf{x}_0$, E, and $\mathbf{x}_1$ to be computed in an extended world. Upon return, $\underline{u}$: nd G_1 and $\underline{h}$: hyp G_1 are discharged, which is reflected in the shift of E's type (in LF) from "conc G_2" to "hyp $G_1 \to$ conc G_2". In fact, $\mathbf{x}_0$'s and $\mathbf{x}_1$'s types (in $\mathcal{M}_2^+$) are also shifted using $\mathbf{\Pi}$-abstraction. The justification of the soundness of this shifting operation follows directly from the one-to-one correspondence of sequent derivations and their image in LF [Sch00].

Not every assumption to be discharged leaves a footprint in the shifted type. For example, E's new type need not reflect the discharge of $\underline{u}$: nd G_1, since natural deduction derivations can never occur as subderivations to sequent derivations. Formally, we say that that LF type 'nd G_1' *is not subordinate to* LF type 'conc G_2' (written as 'nd $G_1 \not\prec$ conc G_2'), but type family hyp *is subordinate to* type family conc (written as 'hyp $G_1 \prec$ conc G_2'). The subordination relation is sometimes called dependency relation and can be effectively computed from the LF signature [Vir99]. In the example above, we write 'shift $(\underline{u} : \text{nd}\ G_1, \underline{h} : \text{hyp}\ G_1)$ (conc G_2)' for the new type of E.

Definition 4.2 (Shifting for types)

$$\begin{array}{rcl} \mathit{shift} \cdot A & = & A \\ \mathit{shift}\ (\rho, \underline{x} : A_1)\ A_2 & = & \begin{cases} \mathit{shift}\ \rho\ (\Pi x : A_1.\ A_2[x/\underline{x}]) & \text{if } A_1 \prec A_2 \\ \mathit{shift}\ \rho\ A_2 & \text{if } A_1 \not\prec A_2 \end{cases} \end{array}$$

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 a generalized context Ψ, and a meta context Δ, whereas for declarations we write $\Psi; \Delta \vdash D \in \Psi'; \Delta'$. The pair $\Psi; \Delta$ is valid, if Ψ is a valid generalized context, and Δ is valid meta-context, possibly referring to variables and parameters declared in Ψ. While D is computed, it computes new LF-objects for declarations in Ψ' and new partially applied programs for declarations in Δ'.

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

The typing rules for these judgments, which are presented in Figure 3, are mutually dependent and resemble a sequent calculus [Gal93]. The shifting operation that defines the side condition of rule Lnew shifts the types and formulas in $\Psi'; \Delta'$ with respect to ρ, respectively. We write "$\Psi'; \Delta' = \text{shift}\ \rho\ (\Psi; \Delta)$" for this straightforward generalization of the type shifting operation [Sch00] from Definition 4.2 to contexts $(\Psi; \Delta)$. The premiss $\Psi \overset{\Pi}{\vdash} M : A$ of the rules $\mathsf{R}\exists$ and $\mathsf{L}\forall$ is the standard LF typing judgment. It allows M to depend on all free variables and parameters declared in Ψ. Functions of type $\mathbf{\Pi}\rho^L.\ F$ can only be applied to other variable blocks given that the types of the individual variables are componentwise convertible. This property is expressed by premiss $\rho' \equiv \rho$ in rule $\mathsf{L}\Pi$.

4.2 Case Analysis

Unlike ML pattern-matching, where valid patterns only involve linear, atomic, and individually independent variables, case analysis in $\mathcal{M}_2^+$ must account for non-linear, functional, and even dependent variables due to higher-order abstract syntax and hypothetical judgments. For example, in Gentzen's algorithm matching 'impi $(\lambda u : \text{nd}\ G_1.\ D)$' against the declaration D of type 'nd G' must make two pieces of information explicit. First, a function D' can now be assumed to exist, and second G's shape is in fact $G = G_1 \supset G_2$ for some G_2. The pattern matching techniques proposed in this paper significantly and elegantly extend the ones typically used in functional programming languages.

Instead of matching only one expression at the time, case subjects in $\mathcal{M}_2^+$ must be entire lists of expressions — or better, substitutions — that follow the structure of the generalized context the case expression is valid in. In the example above, consider the generalized context $\Psi = G : \text{o}, D : \text{nd}\ G$ in which we distinguish cases over D. In the impi-case we can summarize the information obtained by the case split as a generalized context $\Psi' = G_1 : \text{o}, G_2 : \text{o}, D' :$

nd $G_1 \to$ nd G_2. The relationship between Ψ and Ψ' is best expressed by a substitution ψ, which states how G is refined into G_1 and G_2, and D into D'.

$$\psi = (G_1 \text{ imp } G_2/G, \text{impi } (\lambda u : \text{nd } G_1.\, D'\ u)/D)$$

Because the typing judgment is defined with respect to a context pair $\Psi; \Delta$, we take the freedom to extend case subjects to a pair $\psi; \delta$, where δ is defined as a substitution from programs to program variables. Each case is therefore determined by the new domain of the generalized context Ψ', the substitution ψ, and a program P, which is to be executed when the particular case applies and after all declarations in Ψ' are appropriately instantiated.

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

Case subjects expose some of the properties that explicit substitutions possess. For example, if a substitution is applied to a case, it is simply absorbed by substitution composition (which we will not define here). Consequently, the definition of substitution application $P[\psi; \delta]$, $D[\psi; \delta]$ whose definition we omit from this paper need not to traverse the bodies of the defining cases in Ω recursively.

$$(\text{case } (\psi; \delta) \text{ of } \Omega)[\psi'; \delta'] = \text{case } (\psi; \delta) \circ (\psi'; \delta') \text{ of } \Omega$$

Each individual case is therefore closed in that it defines its own generalized context and a corresponding mediating generalized substitution ψ. Formally, the substitution pair $(\psi; \delta)$ is defined to substitute LF objects for LF variables, variable blocks for variable blocks, and programs for program variables.

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

We write $\Psi' \vdash \psi \in \Psi$ if ψ is a valid substitution mapping objects valid in Ψ' into variables in Ψ, and $\Psi'; \Delta' \vdash \psi; \delta \in \Psi; \Delta$ for well-typed substitution pairs.

This seems to conclude the exposition about case analysis in $\mathcal{M}_2^+$, however, as of yet we cannot guarantee that the operational semantics is decidable. In fact, it is not. Consider the example from above again, this time with a weakened Ψ' by the additional (underlined) assumption E

$$G_1 : \text{o}, G_2 : \text{o}, D' : \text{nd } G_1 \to \text{nd } G_2, \underline{E : \text{nd } G_1}$$

but the same valid mediating substitution ψ. Since E is not even mentioned in ψ, it can in general not be decided how to instantiated it during runtime. We henceforth require that for any case $\Psi' \triangleright \psi \mapsto P$ it must be uniquely possible to determine how to instantiate the variables declared in Ψ' by matching against ψ. We say, that Ψ' is strict in ψ, written as $\vdash \Psi'$ strict ψ. A decidable algorithm for strictness is discussed in detail in [Sch00].

Going back to the example above, it is easy to derive the contexts and mediating substitutions for the remaining first case

$$\begin{array}{l} G : \text{o}, (\underline{u} : \text{nd } G, \underline{h} : \text{hyp } G)^L \\ \quad \vdash (G/G, \underline{u}/D) : (G : \text{o}, D : \text{nd } G) \end{array}$$

and for the last case:

$$G_1 : \mathrm{o}, G_2 : \mathrm{o}, D_1 : \mathrm{nd}\ (G_1\ \mathrm{imp}\ G_2), D_2 : \mathrm{nd}\ G_1 \\ \vdash (G_2/G, \mathrm{impe}\ D_1\ D_2/D) : (G : \mathrm{o}, D : \mathrm{nd}\ G)$$

Putting all parts together yields the main case distinction of Gentzen's algorithm from Figure 1. It is sketched in $\mathcal{M}_2^+$ as the program below. How to fill the ... with programs is postponed until Section 4.3.

Example 4.3 (Case analysis)

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

The static semantics of $\mathcal{M}_2^+$ is expressed in terms of the validity of programs discussed above and the validity of cases which is expressed by the following judgment.

$$\textit{Typing cases:} \quad \Psi; \Delta \vdash \Omega \in F$$

The defining rules are given in Figure 4. The premiss of the case rule ensures that the subject of the case program is well-typed.

4.3 Recursion

ndseq cannot be defined without iteration and recursion. In the second and the third case of **ndseq**, for example, recursion is necessary to convert the premisses of a natural deduction derivation into sequent derivations. Therefore, $\mathcal{M}_2^+$ provides a recursion operator μ with the typical associated meaning.

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

During execution, the program variable $\mathbf{x}$ is replaced by $\mu \mathbf{x} \in F.\, P$. Recall that each recursive call corresponds to an appeal of the induction hypothesis in the proof of Lemma 3.1. The one rule that defines recursion can be found in Figure 5. This completes the set of typing rules for $\mathcal{M}_2^+$.

Theorem 4.4 (Decidability) *For any closed program P, it is decidable if there exists a formula F, such that $\cdot; \cdot \vdash P \in F$.*

Proof: By inspection of the typing rules. □

Gentzen's algorithm as given in Figure 1 can be written as program in $\mathcal{M}_2^+$. The underlying induction principle is expressed by case analysis and recursion. For the sake of conciseness, we omit the implicit arguments to impi, impe, impR, and impR.

Example 4.5 (Desugared version of ndseq)

$$
\begin{array}{l}
\mu\mathbf{ndseq} \in \forall G : \mathrm{o}.\, \forall D : \mathrm{nd}\ G.\, \exists E : \mathrm{conc}\ G.\, \top. \\
\quad \Lambda G : \mathrm{o}.\, \Lambda D : \mathrm{nd}\ G. \\
\quad\quad \mathrm{case}\ (G/G, D/D; \cdot) \\
\quad\quad\ \mathrm{of} \quad (G : \mathrm{o}, (\underline{u} : \mathrm{nd}\ G, \underline{h} : \mathrm{hyp}\ G)^L \\
\quad\quad\quad\quad \triangleright\ (G/G, \underline{u}/D) \mapsto \underline{h}) \\
\quad\quad\quad \| \quad (G_1 : \mathrm{o}, G_2 : \mathrm{o}, D' : \mathrm{nd}\ G_1 \to \mathrm{nd}\ G_2 \\
\quad\quad\quad\quad \triangleright (G_1\ \mathrm{imp}\ G_2/G, \mathrm{impi}\ (\lambda u : \mathrm{nd}\ G_1.\, D' u)/D) \\
\quad\quad\quad\quad \mapsto \mathbf{let} \\
\quad\quad\quad\quad\quad\quad \nu\ (\underline{u} : \mathrm{nd}\ G_1, \underline{h} : \mathrm{hyp}\ G_1)^L\,. \\
\quad\quad\quad\quad\quad\quad \mathbf{x}_0 \in \exists E : \mathrm{conc}\ G_2.\, \top = \mathbf{ndseq}\ (D'\ \underline{u}), \\
\quad\quad\quad\quad\quad\quad \langle E : \mathrm{conc}\ G_2, \mathbf{x}_1 \in \top \rangle = \mathbf{x}_0 \\
\quad\quad\quad\quad\quad \mathbf{in} \\
\quad\quad\quad\quad\quad\quad (\mathrm{impR}\ (\lambda h : \mathrm{hyp}\ G_1.\, E\ h)) \\
\quad\quad\quad\quad\quad \mathbf{end}) \\
\quad\quad\quad \| \quad (G_1 : \mathrm{o}, G_2 : \mathrm{o}, D_1 : \mathrm{nd}\ (G_1\ \mathrm{imp}\ G_2), D_2 : \mathrm{nd}\ G_1 \\
\quad\quad\quad\quad \triangleright\ (G_2/G, \mathrm{impe}\ D_1\ D_2/D) \\
\quad\quad\quad\quad \mapsto \mathbf{let} \\
\quad\quad\quad\quad\quad\quad \mathbf{x}_0 \in \exists E_1 : \mathrm{conc}\ (G_1\ \mathrm{imp}\ G_2).\, \top = \mathbf{ndseq}\ D_1, \\
\quad\quad\quad\quad\quad\quad \langle E_1 : \mathrm{conc}\ (G_1\ \mathrm{imp}\ G_2), \mathbf{x}_1 \in \top \rangle = \mathbf{x}_0 \\
\quad\quad\quad\quad\quad\quad \mathbf{x}_2 \in \exists E_2 : \mathrm{conc}\ G_1.\, \top = \mathbf{ndseq}\ D_2, \\
\quad\quad\quad\quad\quad\quad \langle E_2 : \mathrm{conc}\ G_1, \mathbf{x}_3 \in \top \rangle = \mathbf{x}_2 \\
\quad\quad\quad\quad\quad \mathbf{in} \\
\quad\quad\quad\quad\quad\quad (\mathrm{cut}\ E_1\ (\lambda h_1.\, \mathrm{impL}\ E_2\ (\lambda h_2.\, \mathrm{init}\ h_2)\ h_1)) \\
\quad\quad\quad\quad\quad \mathbf{end}) \\
\quad \in \forall G : \mathrm{o}.\, \forall D : \mathrm{nd}\ G.\, \exists E : \mathrm{conc}\ G.\, \top
\end{array}
$$

$\mathcal{M}_2^+$'s type system provides the necessary mechanisms to define functions by cases that range over arbitrary, possibly higher-order LF objects. Functions of this kind are not yet common in functional programming, but they are essential for the formalization of the meta-theory of deductive systems, such as soundness and completeness properties of logics, type theories, and programming languages.

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

The formal system defining the operational semantics is constructed in such a way, that once a computation is invoked in a regularly formed world its result is well-defined in the same world. During evaluation, new parameters can be dynamically introduced but they are discharged when the subcomputation terminates.

The operational semantics is designed to be 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 a generalized 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.

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

The discharge of parameters which is the final step during the execution of a ν-declaration has not only the effect that the types in $\Psi'; \Delta'$ are being shifted (see Section 4.1), but that the substitution $(\psi; \delta)$ is being appropriately parameterized by functional abstraction. The subordination embodies the information which parameter may occur in an object. Depending on if a parameter may occur in an object in ψ or not, the object is either abstracted or left alone. This process is called raising.

Definition 5.1 (Raising) *Let ρ be a variable block, and M an LF-object of type A_2. We write $(\mathit{raise} \cdot M)$ for the result of raising M in ρ.*

$$\begin{array}{rcl} \mathit{raise} \cdot M & = & M \\ \mathit{raise}\ (\rho, \underline{x} : A_1)\ M & = & \begin{cases} \mathit{raise}\ \rho\ (\lambda x : A_1.\, M[x/\underline{x}]) & \text{if } A_1 \prec A_2 \\ \mathit{raise}\ \rho\ M & \text{if } A_1 \not\prec A_2 \end{cases} \end{array}$$

The raising operation is then componentwise extended from individual objects to generalized substitutions. For meta-substitutions however, it is enough to replace programs P by $\lambda\rho^L.\, P$. We write 'raise $\rho\ (\psi; \delta) = (\psi'; \delta')$' for this raising operation. The rules defining the operational semantics are given in Figure 6 where raising is used in the side condition to ev_new.

6 Meta-Theory

The operational semantics of $\mathcal{M}_2^+$ satisfies the usual weakening and substitution properties. To simplify notation we write $(\Psi; \Delta) \leq (\Psi'; \Delta')$ if $(\Psi; \Delta)$ is an initial subcontext of $(\Psi'; \Delta')$.

Lemma 6.1 (Weakening) *1. If $\mathcal{D} :: \Psi; \Delta \vdash P \in F$ and $\Psi; \Delta \leq \Psi'; \Delta'$ then $\Psi'; \Delta' \vdash P \in F$.*

2. If $\mathcal{D} :: \Psi; \Delta \vdash D \in \Psi''; \Delta''$ and $\Psi; \Delta \leq \Psi'; \Delta'$ then $\Psi'; \Delta' \vdash D \in \Psi''; \Delta''$.

3. If $\mathcal{D} :: \Psi'; \Delta' \vdash \psi; \delta \in \Psi; \Delta$ and $\Psi'; \Delta' \leq \Psi''; \Delta''$ then $\Psi''; \Delta'' \vdash \psi; \delta \in \Psi; \Delta$.

Proof: By structural induction on $\mathcal{D}(1)$, $\mathcal{D}(2)$, and $\mathcal{D}(3)$. □

Without defining substitution application to formulas $F[\psi]$, and to contexts $[\psi](\Psi;\Delta)$ we give the following formulation of the substitution lemma.

Lemma 6.2 (Substitution) *1. If* $\mathcal{D} :: \Psi;\Delta \vdash P \in F$ *and* $\Psi';\Delta' \vdash \psi;\delta \in \Psi;\Delta$ *then* $\Psi';\Delta' \vdash P[\psi;\delta] \in F[\psi]$.

2. If $\mathcal{D} :: \Psi;\Delta \vdash D \in \Psi'';\Delta''$ *and* $\Psi';\Delta' \vdash \psi;\delta \in \Psi;\Delta$ *then* $\Psi';\Delta' \vdash D[\psi;\delta] \in [\psi](\Psi'';\Delta'')$.

3. If $\mathcal{D}_1 :: \Psi_2;\Delta_2 \vdash \psi_1;\delta_1 \in \Psi_1;\Delta_1$ *and* $\Psi_3;\Delta_3 \vdash \psi_2;\delta_2 \in \Psi_2;\Delta_2$ *then* $\Psi_3;\Delta_3 \vdash (\psi_1;\delta_1)\circ(\psi_2;\delta_2) \in \Psi_1;\Delta_1$ *and* $(\psi_1;\delta_1)\circ(\psi_2;\delta_2) = (\psi_1\circ\psi_2,\delta')$ *for some effectively computable meta-substitution* δ'.

Proof: by induction on $\mathcal{D}(1), \mathcal{D}(2), \mathcal{D}_1(3)$ using the substitution Lemma for LF [HHP93]. A detailed proof can be found in [Sch00], Appendix C. □

The main theoretical contribution of this article is that the operational semantics presented in Section 5 is type preserving. Well-typed programs, declarations, and cases will always compute a result of the same type. We write id_Φ for the identity substitution, which satisfies $\Phi \vdash \mathrm{id}_\Phi : \Phi$.

Theorem 6.3 (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* $\Phi;\cdot \vdash V \in F[\psi]$

Proof: by simultaneous induction over $\mathcal{D}(1), \mathcal{D}(2), \mathcal{D}(3)$ using Lemma 6.2, Lemma 6.1. A detailed proof can be found in [Sch00], Appendix C. □

7 Conclusion

$\mathcal{M}_2^+$ is a type theory that serves the formalization of algorithms about higher-order encodings of deductive systems prevalent in the areas of logics and programming languages. It draws its expressive power from a combination of the regular world assumption and elegant higher-order encodings supported by the logical framework LF.

The regular world assumption permits regular extensions of the set of constructors for a particular datatype during runtime which in turn makes it possible for recursive calls to be executed even under λ-binders. Arguments of functional type are quite common for encodings in LF because higher-order abstract syntax and hypothetical judgments are directly supported by LF. Moreover, $\mathcal{M}_2^+$ inherits all properties that are associated with contexts from LF, such

as weakening, contraction, exchange, the definition of substitution application, and all related substitution lemmas.

The Twelf system [PS99] implements a type-checker and an theorem prover for $\mathcal{M}_2^+$, that search for functions of given type. Besides the type, the theorem prover requires as input a list of arguments on which cases should be distinguished on, it is fully automatic in any other respect.

For example, $\mathcal{M}_2^+$ has been used to formalize and automatically construct algorithms to compute normal forms in various λ-calculi exploiting the proof of the Church-Rosser theorem [Pfe93], procedures to eliminate the cut rule from sequent derivations in first-order classical and intuitionistic logic based on the proof of cut-elimination [Pfe95], and conversions from Cartesian Closed Categories to the simply typed λ-calculus. See [Sch00] for a more complete list of experiments conducted with $\mathcal{M}_2^+$.

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 allows to formulate algorithms about resource oriented logics and programming languages with state in a natural way. We are convinced that it is feasible because McDowell and Miller [MM97] successfully encoded fragments of second-order linear logic into their meta-logic $FO\lambda^{\Delta IN}$.

Acknowledgment: I want to thank my advisor Frank Pfenning for his guidance and advice during my thesis 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.

[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.

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

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

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

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

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

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

$$\frac{\Psi, \rho^L; \Delta \vdash D \in \Psi'; \Delta'}{\Psi; \Delta \vdash \nu\ \rho^L.\, D \in \Psi''; \Delta''}\,\mathsf{Lnew}$$

where $\Psi''; \Delta'' = \text{shift } \rho\ (\Psi'; \Delta')$

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

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

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

Figure 3: Typing Rules of the Core of $\mathcal{M}_2^+$

$$\frac{\Psi; \Delta \vdash \psi; \delta \in \Psi'; \Delta' \qquad \Psi'; \Delta' \vdash \Omega \in F}{\Psi; \Delta \vdash \text{case } (\psi; \delta) \text{ of } \Omega \in [\psi]F}\ \mathsf{case}$$

$$\frac{}{\Psi; \Delta \vdash \cdot \in F}\ \mathsf{base}$$

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

Figure 4: Case Analysis in $\mathcal{M}_2^+$.

$$\frac{\Psi; \Delta, \mathbf{x} \in F \vdash P \in F}{\Psi; \Delta \vdash \mu \mathbf{x} \in F.\, P \in F}\ \mathsf{Rctx}$$

Figure 5: Recursion in $\mathcal{M}_2^+$.

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

$$\frac{\Phi \vdash P \hookrightarrow V}{\Phi \vdash \langle M, P\rangle \hookrightarrow \langle M, V\rangle}\ \mathsf{ev_inx} \qquad \frac{}{\Phi \vdash \langle\rangle \hookrightarrow \langle\rangle}\ \mathsf{ev_unit}$$

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

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

$$\frac{}{\Phi \vdash \cdot \hookrightarrow \cdot; \cdot}\ \mathsf{ev_empty}$$

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

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

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

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

where $(\psi'; \delta') = \text{raise } \rho\ (\psi; \delta)$

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

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

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

Figure 6: Operational semantics of $\mathcal{M}_2^+$