LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

A Framework for Defining Logics

by

Robert Harper Furio Honsell Gordon Plotkin

LFCS Report Series **ECS-LFCS-87-23**
(Also published as CSR-228-87)

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

March 1987
Update September 1987

A Framework for Defining Logics*

Robert Harper Furio Honsell Gordon Plotkin
Laboratory for Foundations of Computer Science
University of Edinburgh

Abstract

The Logical Framework (LF) is a system for defining a wide class of logics. It is based on a general treatment of syntax, rules, and proofs in terms of a typed λ–calculus with dependent types. Syntax is treated in a style similar to, but more general than, Martin-Löf's system of arities. The treatment of rules and proofs focuses on the notion of a *judgement*. Logics are encoded in the LF via a new principle, the *judgements as types* principle, whereby each judgement is identified with the type of its proofs. This allows for a smooth treatment of discharge and variable occurrence conditions and leads to a uniform treatment of rules and proofs whereby rules are viewed as proofs of higher–order judgements and proof checking is reduced to type checking. An important benefit of our treatment of formal systems is that logic–independent tools such as proof editors and proof checkers can be constructed.

1 Introduction

Much work has been devoted to building systems for checking and building formal proofs in various logical systems. The AUTOMATH project of deBruijn [4] considered first proof checking. The problem of interactive proof construction was first considered by Milner, *et. al.* in the LCF system [11]. The LCF system was adapted to type theory by Petersson [19]. The work of Huet and Coquand on the Calculus of Constructions [7,8] extends the AUTOMATH and LCF work to a more powerful logic. Paulson's work on Isabelle [18] is a general approach to proof construction based on higher–order resolution. The NuPRL system of Constable [6] is a display–based interactive proof development environment for type theory that includes facilities for notation extension, library management, and automated proof search.

There are a great many logics of interest (equational, first–order, higher–order, temporal, type and set theories, type assignment systems and operational semantics for

*This is a slightly edited version of a paper delivered at the Second Symposium on Logic in Computer Science, Ithaca, NY, June, 1987. Any citations should refer to the proceedings of that conference.

programming languages). Implementing an interactive proof development environment for any style of presentation of any of these logics is a daunting task. For example, one must implement a parser, term manipulation operations (such as substitution and α–conversion), definitions and notation extension, inference rules, proofs, tactics, and tacticals. Thus it is desirable to find a general theory of inference systems that captures the uniformities of a large class of logics so that much of this effort can be expended once and for all.

The Logical Framework is an attempt to provide such a general theory of logics. It is based on a weak type theory that is closely related to AUT–PI and AUT–QE [9], to Martin-Löf's early type theory [14], to Huet and Coquand's Calculus of Constructions [8], and to Meyer and Reinhold's λ^{π} [16]. It is able to specify both the language of a logic, its axiom and rule schemes, and its proofs. The language of a logic is defined in a general theory of expressions that exploits the λ–calculus structure to provide binding operators, substitution, capture, α–conversion, and schematic abstraction and instantiation.

The treatment of rules and proofs is based on the notion of a *judgement* [15], the unit of assertion in a logical system. (See also Schroeder–Heister [22] for a related point of view.) Each logic is a system for asserting basic judgements. The set of judgements is then closed under two higher–order judgement forms that are used to specify inference rule schemes and to model discharge and variable occurrence conditions such as arise in Hilbert systems or systems of natural deduction. Rules are viewed as the proofs of (possibly higher–order) judgements that specify them. There is no distinction between primitive and derived rules. The extension to higher–order judgements allows for a natural presentation of many logical systems that avoids side conditions on rules. Judgements, rules, and proofs are represented in the LF type theory by applying what we call the *judgements as types* principle whereby each judgement is identified with the type of its proofs. Each basic judgement is represented by a base type of the LF type theory, and the higher–order judgements are represented in a logic–independent way by functional types. Proofs, and hence rules, are represented as terms of the LF type theory, thereby reducing proof checking to type checking.

In Section 2 we present the type theory of the LF, along with some of its important proof–theoretic properties. In Section 3, we introduce the LF's theory of expressions, and consider predicate calculus and Church's higher–order logic as examples. In Section 4 we consider the treatment of judgements, rules, and proofs in the LF. The method is illustrated for predicate calculus and higher–order logic. In Section 5 we compare our work with other systems for defining logics, and in Section 6 we suggest directions for future research.

We gratefully acknowledge the influence of Per Martin-Löf, particularly the lectures delivered in Edinburgh in the spring of 1986. We are grateful to Tim Griffin of Cornell University for his efforts in building a prototype implementation of the LF. We also thank the other members of the Computer Assisted Formal Reasoning project at Edinburgh, especially Arnon Avron and Ian Mason, for their comments and criticisms. Support for this research was provided by the Science and Engineering Research Council.

2 The LF Type Theory

The type theory of the LF is closely related to the Π–fragment of AUT–PI, a language belonging to the AUTOMATH family. The LF type theory is a language with entities of three levels: (1) objects, (2) types and families of types, and (3) kinds. Objects are classified by types, types and families of types by kinds. The kind Type classifies the types; the other kinds classify functions f which yield a type $f(x_1)\ldots(x_n)$ when applied to objects $x_1,\ldots,x_n$ of certain types determined by the kind of f. Any function definable in the system has a type as domain, while its range can either be a type, if it is an object, or a kind, if it is a family of types. The LF type theory is therefore predicative.

A number of different presentations of this system can be given. We shall describe a version which trades off conciseness against readability. The theory we shall deal with is a formal system for deriving assertions of one of the following shapes:

$\vdash \Sigma$ sig	Σ is a signature
$\vdash_\Sigma \Gamma$ context	Γ is a context
$\Gamma \vdash_\Sigma K$ kind	K is a kind
$\Gamma \vdash_\Sigma A : K$	A has kind K
$\Gamma \vdash_\Sigma M : A$	M has type A

where the syntax is specified by the following grammar:

Signatures	Σ	::=	$\langle\rangle \mid \Sigma, c{:}K \mid \Sigma, c{:}A$
Contexts	Γ	::=	$\langle\rangle \mid \Gamma, x{:}A$
Kinds	K	::=	$\mathrm{Type} \mid \Pi x{:}A.K$
Type Families	A	::=	$c \mid \Pi x{:}A.B \mid \lambda x{:}A.B \mid AM$
Objects	M	::=	$c \mid x \mid \lambda x{:}A.M \mid MN$

We let M and N range over expressions for objects, A and B for types and families of types, K for kinds, x and y over variables, and c over constants. We write $A \to B$ for $\Pi x{:}A.B$ when x does not occur free in B.

The inference rules of the LF type theory appear in Table 1. A term is said to be *well–typed in a signature and context* if it can be shown to either be a kind, have a kind, or have a type in that signature and context. A term is *well–typed* if it is well–typed in some signature and context. The notion of $\beta\eta$–contraction, written $\to_{\beta\eta}$, can be defined both at the level of objects and at the level of types and families of types in the obvious way. Rules (12) and (17) make use of a relation $=_{\beta\eta}$ between terms which is defined as follows: $M =_{\beta\eta} N$ iff $M \to^*_{\beta\eta} P$ and $N \to^*_{\beta\eta} P$ for some term P. We conjecture that variants of rules (12) and (17) obtained by taking $=_{\beta\eta}$ to be $\beta\eta$–conversion are admissible rules of the theory.

Since the notion of $\beta\eta$–conversion over $K \cup A \cup M$ is not Church–Rosser, the order of technical priority in which the basic metatheoretical results are proved is essential. The following theorem summarizes these results in a convenient order (here α ranges over the basic assertions of the type theory):

Valid Signature

$$\frac{}{\vdash \langle\rangle \text{ sig}} \tag{1}$$

$$\frac{\vdash \Sigma \text{ sig} \quad \vdash_\Sigma K \text{ kind} \quad c \notin \text{dom}(\Sigma)}{\vdash \Sigma, c{:}\, K \text{ sig}} \tag{2}$$

$$\frac{\vdash \Sigma \text{ sig} \quad \vdash_\Sigma A : \text{Type} \quad c \notin \text{dom}(\Sigma)}{\vdash \Sigma, c{:}\, A \text{ sig}} \tag{3}$$

Valid Context

$$\frac{\vdash \Sigma \text{ sig}}{\vdash_\Sigma \langle\rangle \text{ context}} \tag{4}$$

$$\frac{\vdash_\Sigma \Gamma \text{ context} \quad \Gamma \vdash_\Sigma A : \text{Type} \quad x \notin \text{dom}(\Gamma)}{\vdash_\Sigma \Gamma, x{:}\, A \text{ context}} \tag{5}$$

Valid Kinds

$$\frac{\vdash_\Sigma \Gamma \text{ context}}{\Gamma \vdash_\Sigma \text{Type kind}} \tag{6}$$

$$\frac{\Gamma \vdash_\Sigma A : \text{Type} \quad \Gamma, x{:}\, A \vdash_\Sigma K \text{ kind}}{\Gamma \vdash_\Sigma \Pi x{:}\, A.K \text{ kind}} \tag{7}$$

Valid Elements of a Kind

$$\frac{\vdash_\Sigma \Gamma \text{ context} \quad c{:}\, K \in \Sigma}{\Gamma \vdash_\Sigma c : K} \tag{8}$$

$$\frac{\Gamma \vdash_\Sigma A : \text{Type} \quad \Gamma, x{:}\, A \vdash_\Sigma B : \text{Type}}{\Gamma \vdash_\Sigma \Pi x{:}\, A.B : \text{Type}} \tag{9}$$

$$\frac{\Gamma \vdash_\Sigma A : \text{Type} \quad \Gamma, x{:}\, A \vdash_\Sigma B : K}{\Gamma \vdash_\Sigma \lambda x{:}\, A.B : \Pi x{:}\, A.K} \tag{10}$$

$$\frac{\Gamma \vdash_\Sigma B : \Pi x{:}\, A.K \quad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma BN : [N/x]K} \tag{11}$$

$$\frac{\Gamma \vdash_\Sigma A : K \quad \Gamma \vdash_\Sigma K' \text{ kind} \quad K =_{\beta\eta} K'}{\Gamma \vdash_\Sigma A : K'} \tag{12}$$

Valid Elements of a Type

$$\frac{\vdash_\Sigma \Gamma \text{ context} \quad c{:}\, A \in \Sigma}{\Gamma \vdash_\Sigma c : A} \tag{13}$$

$$\frac{\vdash_\Sigma \Gamma \text{ context} \quad x{:}\, A \in \Gamma}{\Gamma \vdash_\Sigma x : A} \tag{14}$$

$$\frac{\Gamma \vdash_\Sigma A : \text{Type} \quad \Gamma, x{:}\, A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma \lambda x{:}\, A.M : \Pi x{:}\, A.B} \tag{15}$$

$$\frac{\Gamma \vdash_\Sigma M : \Pi x{:}\, A.B \quad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma MN : [N/x]B} \tag{16}$$

$$\frac{\Gamma \vdash_\Sigma M : A \quad \Gamma \vdash_\Sigma A' : \text{Type} \quad A =_{\beta\eta} A'}{\Gamma \vdash_\Sigma M : A'} \tag{17}$$

Table 1: The LF Type System

Theorem 2.1

1. *Thinning is an admissible rule: if* $\Gamma \vdash_\Sigma \alpha$ *and* $\vdash_{\Sigma,\Sigma'} \Gamma, \Gamma'$ context, *then* $\Gamma, \Gamma' \vdash_{\Sigma,\Sigma'} \alpha$.
2. *Transitivity is an admissible rule: if* $\Gamma \vdash_\Sigma M : A$ *and* $\Gamma, x{:}A, \Delta \vdash_\Sigma \alpha$, *then* $\Gamma, [M/x]\Delta \vdash_\Sigma [M/x]\alpha$.
3. *Uniqueness of types and kinds: if* $\Gamma \vdash_\Sigma M : A$ *and* $\Gamma \vdash_\Sigma M : A'$, *then* $A =_{\beta\eta} A'$, *and similarly for kinds.*
4. *Subject reduction: if* $\Gamma \vdash_\Sigma M : A$ *and* $M \to^*_{\beta\eta} M'$, *then* $\Gamma \vdash_\Sigma M' : A$, *and similarly for types.*
5. *All well–typed terms are strongly normalizing.*
6. *All well–typed terms are Church–Rosser.*
7. *Each of the five relations defined by the inference system of Table 1 is decidable, as is the property of being well–typed.*

The proof of this theorem is surprisingly complicated. However, the methods developed by van Daalen in his thesis [9] can be adapted to this type theory. It is important to note that the Church–Rosser property, strong normalization, and the presence of type labels are essential in the proof of decidability of the type theory. In particular, given a signature and a context and any object (type) expression, it is decidable if the object (type) expression is well–typed; if so, a type (kind) can be computed for it.

We shall outline only the proof of strong normalization, since, unlike the systems dealt with by van Daalen, it can be proved independently of the other metatheoretic results. Moreover, it yields a corollary that is useful for characterizing the terms that are definable in the LF type theory:

Corollary 2.2 *If* $\Gamma \vdash_\Sigma M : A$, *then* Erase$(M)$ *can be typed in Curry's type assignment system, where* Erase(M) *denotes the term obtained from* M *by removing the type labels from the* λ *abstractions.*

We start by defining a translation τ of the LF types and kinds into simple types with a base type α, and a translation $\tilde{}$ of LF objects and types into the untyped λ terms with a constant π. These translations are extended to signatures and contexts in the obvious way.

Definition 2.3

$$
\begin{array}{lcl}
\tau(\text{Type}) & = & \alpha \\
\tau(\Pi x{:}A.K) & = & \tau(A) \to \tau(K) \\
\tau(c) & = & \alpha \\
\tau(\lambda x{:}A.B) & = & \tau(B) \\
\tau(AM) & = & \tau(A) \\
\tau(\Pi x{:}A.B) & = & \tau(A) \to \tau(B) \\
\tilde{c} & = & c \\
\tilde{x} & = & x \\
\widetilde{AM} & = & \tilde{A}\tilde{M} \\
\widetilde{MN} & = & \tilde{M}\tilde{N} \\
\Pi x{:}\widetilde{A}.B & = & \pi\tilde{A}\tilde{B} \\
\lambda x{:}\widetilde{A}.M & = & (\lambda y.\lambda x.\tilde{M})\tilde{A} \quad (y \notin \mathrm{FV}(\lambda x.\tilde{M})) \\
\lambda x{:}\widetilde{A}.B & = & (\lambda y.\lambda x.\tilde{B})\tilde{A} \quad (y \notin \mathrm{FV}(\lambda x.\tilde{B}))
\end{array}
$$

The precise sense in which this definition is consistent is stated in the following lemma:

Lemma 2.4

1. *If* $\Gamma \vdash_\Sigma A : K$, *then* $\tau(\Gamma) \vdash_{\tau(\Sigma)} \tilde{A} : \tau(K)$;
2. *If* $\Gamma \vdash_\Sigma M : A$, *then* $\tau(\Gamma) \vdash_{\tau(\Sigma),\pi:\alpha\to\alpha\to\alpha} \tilde{M} : \tau(A)$.

Proof. *By induction on the structure of the proof of* $\Gamma \vdash_\Sigma A : K$ *and* $\Gamma \vdash_\Sigma M : A$. *We have only to notice that any derivation of* $\vdash_\Sigma \Gamma, x : A$ context *contains as a subderivation a derivation of* $\Gamma \vdash_\Sigma A : \text{Type}$. □

The translation has been carried out in such a way that the extra combinatorial complexity in the LF terms due to the presence of type labels is not lost. We then have the following result:

Theorem 2.5

1. *If* $A \to_{\beta\eta} A'$, *then* $\tilde{A} \to^*_{\beta\eta} \tilde{A}'$;
2. *If* $M \to_{\beta\eta} M'$, *then* $\tilde{M} \to^*_{\beta\eta} \tilde{M}'$.

Proof. *By induction on the derivation of* $A \to A'$ *and* $M \to M'$. □

Now since the Curry typable terms are strongly normalizing, so too are the terms of the LF. Moreover, it can be easily seen that $\tilde{A} \to^*_{\beta\eta} \text{Erase}(A)$, and $\tilde{M} \to^*_{\beta\eta} \text{Erase}(M)$.

A few remarks about the choice of the type theory are in order. It is essential that the type theory be decidable, at least theoretically, for, as we shall see below, proof checking is reduced to type checking. The use of conversion as the only counterpart of definitional equality is due to the fact that at this stage the LF does not deal with the issue of proof reduction and equality in the sense of Prawitz. We use η–conversion in order to make the encoding of syntax more transparent.

3 Theory of Expressions

The approach to formalizing the syntax of a language is inspired by Church [5] and Martin-Löf system of arities [17]. Each syntactic category of the logic is represented by a type. The expressions of each category are built up using expression–forming constructors, which are formalized by suitable constants of the LF. Variable binding operators are represented by constants whose domain is a function type, so that binding is handled by the λ–calculus of the LF. The type theory of the LF being richer than simple type theory, our approach to syntax is more general than Martin-Löf's. Explicit use of this extra generality is made in the formalization of Church's higher–order logic.

To illustrate the formalization of syntax within the LF, we consider two examples: the language of Peano arithmetic (as defined in Schoenfield [21]) and the language of higher–order logic [5]. The presentation of the syntax of Peano arithmetic will form a part of its signature, Σ_{PA}, and similarly for the signature of higher–order logic, Σ_{HOL}.

In a first–order language there are two syntactic categories: the *individual expressions*, which stand for individuals (objects in the domain of quantification), and the *formulas*, which stand for propositions. These are represented in the LF by the type ι of individuals, and the type o of propositions. Thus Σ_{PA} begins with:

$$\begin{array}{lcl} \iota & : & \mathrm{Type} \\ o & : & \mathrm{Type} \end{array}$$

The individual expression constructors of Peano arithmetic are formalized in Σ_{PA} by:

$$\begin{array}{lcl} 0 & : & \iota \\ \mathrm{succ} & : & \iota \to \iota \\ + & : & \iota \to \iota \to \iota \\ \times & : & \iota \to \iota \to \iota \end{array}$$

Terms of type ι in Σ_{PA} represent the individual expressions of Peano arithmetic. There are no declarations for the variables of first–order logic. The variables of the object language are identified with the variables of the LF, so that an open term of type ι in Σ_{PA}, all of whose free variables are of type ι, represents an open individual expression. For example, in a context containing $x{:}\iota$, the term $\mathrm{succ}(x)$ has type ι as well. This representation is defined compositionally by: $x^\circ = x$, $0^\circ = 0$, $\mathrm{succ}(t)^\circ = \mathrm{succ}(t^\circ)$, $t + u^\circ = +(t^\circ)(u^\circ)$, and $t \times u^\circ = \times(t^\circ)(u^\circ)$,

Theorem 3.1 (Adequacy for Syntax, I) *The correspondence $^\circ$ is a bijection between the expressions of Peano arithmetic and the normal forms of type ι in Σ_{PA} with all free variables of type ι.*

Proof sketch. *The translation is evidently well–defined and one–one. Surjectivity is proved by induction on the structure of the normal forms.* □

The following are the other constant declarations for the formulas of Peano arithmetic:

$$\begin{array}{lcllcl} = & : & \iota \to \iota \to o & < & : & \iota \to \iota \to o \\ \neg & : & o \to o & \wedge & : & o \to o \to o \\ \vee & : & o \to o \to o & \supset & : & o \to o \to o \\ \forall & : & (\iota \to o) \to o & \exists & : & (\iota \to o) \to o \end{array}$$

The formula $\psi = \forall x.\phi[x]$ is represented by the term $\psi^\circ = \forall(\lambda x{:}\,\iota.\phi^\circ)$. This approach allows α–conversion and capture–avoiding substitution to be factored out of the definition of each individual logic, leaving it to be implemented once and for all by the framework. This treatment of binding operators relies on the variables of the first–order language being identified with the variables of the LF type theory. For example, if x is a variable of type ι, then $x = x$ is a term of type o.[1] We can bind x by λ–abstraction, obtaining $\lambda x{:}\,\iota.x = x$, and universally quantify it by applying it to the constant $\forall$, obtaining $\forall(\lambda x{:}\,\iota.x = x)$, which represents the first–order formula $\forall x.x = x$.

In this way, each formula ϕ of Peano arithmetic is represented by a term ϕ° of type o in Σ_{PA}, all of whose free variables are of type ι; sentences are represented by the closed terms of type o. An open term M of type o is an *incomplete formula*. Its λ–abstraction is a *formula scheme*. For example, the formula scheme

$$M = \lambda\phi{:}\,o.\lambda\Phi{:}\,\iota \to o.\phi \supset \exists(\Phi)$$

can be instantiated by application to a formula ϕ and a matrix Φ, so that

$$M(\forall(\lambda x{:}\,\iota.x = x))(\lambda x{:}\,\iota.x = x)$$

represents the first–order formula

$$\forall x.x = x \supset \exists x.x = x.$$

Theorem 3.2 (Adequacy for Syntax, II) *The compositional translation* $^\circ$ *is a bijection between the formulas of Peano arithmetic and the long* $\beta\eta$ *normal forms of type* o *in* Σ_{PA} *with all free variables of type* ι.

Proof sketch. *Similar to Theorem 3.1.* □

The role of η–conversion in the above proof is mainly to ensure that the well–typed terms of type o in Σ_{PA} are exactly the formulas of Peano arithmetic, up to the notion of definitional equality built in to the system. There is no intrinsic difference between $\forall(<(0))$ and $\forall(\lambda x{:}\,\iota.<(0)(x))$.

The formalization of the syntax of higher–order logic makes use of the dependent function type of the LF. Quantification in higher–order logic is over a type drawn from the hierarchy of simple functional types with two base types ι (of individuals) and o (of propositions). In order to avoid confusion with the types of the LF, we call the types of

[1] We freely use infix and postfix application in accordance with custom and readability considerations.

higher–order logic "sorts," and we shall write $\sigma \Rightarrow \tau$ for the sort of functions from sort σ to sort τ. In the formalization of higher–order logic the collection of sorts is represented as a type with members ι and o, closed under $\Rightarrow$. The signature Σ_{HOL} thus begins as follows:

$$\begin{array}{lcl} \text{sorts} & : & \text{Type} \\ \iota & : & \text{sorts} \\ o & : & \text{sorts} \\ \Rightarrow & : & \text{sorts} \rightarrow \text{sorts} \rightarrow \text{sorts} \end{array}$$

To each sort is associated the type of objects of that sort. The objects of sort ι are, for the present purposes, the natural numbers. The objects of sort o are the propositions of higher–order logic. The quantifiers range over an arbitrary sort, rather than the fixed sort of individuals as in first–order logic. The objects of functional sort are given by typed λ terms (which we write with a capital λ to avoid confusion), and there is a form for expressing application.

$$\begin{array}{lcl} \text{obj} & : & \text{sorts} \rightarrow \text{Type} \\ 0 & : & \text{obj}(\iota) \\ \text{succ} & : & \text{obj}(\iota \Rightarrow \iota) \\ = & : & \Pi\sigma\text{:sorts.}\,\text{obj}(\sigma \Rightarrow \sigma \Rightarrow o) \\ \neg & : & \text{obj}(o \Rightarrow o) \\ \wedge & : & \text{obj}(o \Rightarrow o \Rightarrow o) \\ \vee & : & \text{obj}(o \Rightarrow o \Rightarrow o) \\ \supset & : & \text{obj}(o \Rightarrow o \Rightarrow o) \\ \forall & : & \Pi\sigma\text{:sorts.}\,\text{obj}((\sigma \Rightarrow o) \Rightarrow o) \\ \exists & : & \Pi\sigma\text{:sorts.}\,\text{obj}((\sigma \Rightarrow o) \Rightarrow o) \\ \Lambda & : & \Pi\sigma\text{:sorts.}\Pi\tau\text{:sorts.}(\text{obj}(\sigma) \rightarrow \text{obj}(\tau)) \rightarrow \text{obj}(\sigma \Rightarrow \tau) \\ \text{ap} & : & \Pi\sigma\text{:sorts.}\Pi\tau\text{:sorts.}\,\text{obj}(\sigma \Rightarrow \tau) \rightarrow \text{obj}(\sigma) \rightarrow \text{obj}(\tau) \end{array}$$

The representation of equality and the quantifiers makes use of the dependent function types of the LF. For each sort σ, the equality relation for objects of sort σ is written $=_\sigma$; it is an object of sort $\sigma \Rightarrow \sigma \Rightarrow o$. Similarly, the quantifiers ranging over sort σ are written $\forall_\sigma$ and $\exists_\sigma$; they are objects of sort $(\sigma \Rightarrow o) \Rightarrow o$, just as in Church's formulation. The Λ and ap forms must similarly be tagged with types, which we write as subscripts. The Λ form must be tagged with both the domain and range types, unlike in Church's definition. The difference is minor, and we expect that analogs of Theorems 3.1 and 3.2 can be proved.

4 Theory of Rules and Proofs

The treatment of inference rules and proofs lies at the heart of the LF. The approach is organized around the notion of a *judgement* [15], the unit of inference of a logic. Each logic comes with a set of *basic* judgements. In first–order logic there is only one form of

basic judgement, the assertion ϕ true that a formula ϕ is (logically) true (usually written as $\vdash \phi$ or just ϕ). Sequent calculi also have one basic judgement, written $\Gamma \Rightarrow \Delta$, the assertion that some formula in Δ is a logical consequence of all the formulas in Γ. In Martin-Löf's system of type theory, there are four basic judgements, A type, $A = B$, $a \in A$, and $a = b \in A$.

In traditional logical systems the inference rules determine the set of proofs of basic judgements, and thereby also determine the set of "correct" or "evident" [15] basic judgements, namely those that have proofs. There are several approaches to the definition of a proof in a formal system. [21,20] Proofs are sometimes viewed as sequences of formulas that satisfy the condition that each formula is obtained from previous formulas by application of a rule. Another view is that proofs are trees satisfying certain conditions. In any case the *notion* of a proof is independent of the particular rules of inference.

We extend the notion of proof to include our view of rules as proofs of *higher–order judgements*. There are two forms: the *hypothetical* and the *schematic* (or *general*). The hypothetical judgement $J_1 \vdash J_2$ is the assertion that J_2 is a logical consequence of J_1, according to the rules of the logic. It is proved by a function mapping proofs of J_1 to proofs of J_2. The schematic judgement $\bigwedge_{x:A} J(x)$ represents the idea of generality: the judgement $J(x)$ is evident for any object x of type A. It is proved by a function mapping objects x of type A to proofs of $J(x)$.

Rules and proofs are represented as terms of the LF type theory. The basic rules are presented as constants in the signature of the logic, and the derived rules are complex proofs that are λ–abstracted with respect to their premises. Since rules are functions, complex proofs are built by applying (in the sense of the λ–calculus) rules to the proof(s) of their premise(s). Rule schemes are represented as proofs of schematic judgements. Schematic variables are identified with the variables of the LF, so that schematization is achieved by λ abstraction, and schematic instantiation by application.

If rules and proofs are terms, what are to be their types? Since a proof is viewed as evidence for a judgement, it seems natural to identify judgements with the type of their proofs: a judgement is evident iff it has a proof iff there is a term of that type (in the signature of the logic). We call this the *judgements as types* principle, by analogy with the propositions as types principle of Curry, deBruijn, and Howard. Here we are making no commitment to the semantics of a logic. Instead we are merely formalizing the idea that to make an assertion in a logical system, one must have a proof of it.

The type of proofs of a basic judgement is determined by the inference rules of the logic. The types of proofs of the higher–order judgement forms are defined by the LF. We define $J_1 \vdash J_2$ to be $J_1 \to J_2$, the type of functions mapping J_1 to J_2. This definition is motivated by the meaning of the hypothetical judgement and the judgements as types principle. Similarly, we define $\bigwedge_{x:A} J(x)$ to be $\Pi x{:}\, A.J(x)$, the type of functions mapping objects x of type A to $J(x)$. We write $J_1, \ldots, J_m \vdash_{x_1:A_1,\ldots,x_n:A_n} J$ for $\bigwedge_{x_1:A_1} \cdots \bigwedge_{x_n:A_n} J_1 \vdash \cdots (J_m \vdash J)$. This incorporates and generalizes Martin-Löf's hypothetico–general judgements [15].

$$(\neg\neg\text{E})\quad \frac{\neg\neg\phi}{\phi} \qquad (\supset\text{I})\quad \frac{\begin{array}{c}(\phi)\\ \psi\end{array}}{\phi \supset \psi}$$

$$(\forall\text{I}^*)\quad \frac{\phi[x]}{\forall x.\phi[x]} \qquad (\forall\text{E})\quad \frac{\forall x.\phi[x]}{\phi[t/x]}$$

(x not free in any assumption on which ϕ depends.)*

$$(\exists\text{E}^*)\quad \frac{\exists x.\phi[x] \quad \begin{array}{c}(\phi)\\ \psi\end{array}}{\psi}$$

(x not free in ψ or any assumptions on which ψ depends.)*

$$(\text{IND}^*)\quad \frac{\phi(0) \quad \begin{array}{c}(\phi(x))\\ \phi(\text{succ}(x))\end{array}}{\phi(x)}$$

(x not free in any assumption, other than $\phi(x)$, in which $\phi(\text{succ}(x))$ depends.)*

Table 2: Some Rules of Peano Arithmetic

We take incomplete proofs to be open terms of judgement type J. They can be completed by substitution or by λ–abstraction, yielding proofs of schematic judgements. Abstraction on judgement type variables not occurring in J yields proofs of hypothetical judgements.

An important consequence of the judgements as types principle is that we are able to reduce proof checking to type checking. A term M is a proof of a judgement J iff M has type J in the signature of the logic. This reduction is the most important reason for insisting that the type theory of the LF be decidable, for otherwise one could not construct a mechanical proof checker for a logic.

To illustrate these ideas, we formalize an illustrative selection of rules from first–order and higher–order logic formalized as systems of natural deduction. Returning to Σ_{PA}, we represent the single judgement form ϕ true by introducing a family of types indexed by the propositions:

$$\text{true} \quad : \quad o \rightarrow \text{Type}$$

We write "ϕ true" for "true(ϕ)." For any formula ϕ (*i.e.*, any term of type o in Σ_{PA}), the type ϕ true is the type of proofs of ϕ.

Each rule in Table 2 is represented by a constant whose type is the specification of the rule, a higher–order judgement. For instance, the double negation elimination rule

is given by:

$$\neg\neg\mathrm{E} \quad : \quad \neg\neg\phi \text{ true} \vdash_{\phi:o} \phi \text{ true}$$

The judgement is schematic in propositions ϕ and hypothetical in proofs of $\neg\neg\phi$, so if ϕ is a formula and M is a proof of $\neg\neg\phi$ true, then $\neg\neg\mathrm{E}(\phi)(M)$ is a proof of ϕ true.

The implication introduction rule makes use of the hypothetical judgement form to model discharge. The formulation of $\supset$I in Table 2 takes a *hypothetical proof of* ϕ as premise, and discharges the hypothesis. We instead treat $\supset$I as taking a *proof of a hypothetical judgement.* The general intention is that a sufficient condition for establishing the truth of $\phi \supset \psi$ is to establish that ψ is a logical consequence of ϕ. The formalization of $\supset$I, which is schematic in ϕ and ψ, is:

$$\supset\mathrm{I} \quad : \quad (\phi \text{ true} \vdash \psi \text{ true}) \vdash_{\phi:o,\psi:o} \phi \supset \psi \text{ true}$$

So, for example, $\supset\mathrm{I}(\phi)(\phi)(\lambda x{:}\phi \text{ true}.x)$ is a proof of $\phi \supset \phi$ true.

Universal elimination is given by:

$$\forall\mathrm{E} \quad : \quad \forall(\Phi) \text{ true} \vdash_{\Phi:\iota\to o,a:\iota} \Phi(a) \text{ true}$$

The rule is schematic in Φ, the matrix of the universally quantified formula, and a, the instance. Given such and a proof M of $\forall(\Phi)$ true, $\forall\mathrm{E}(\Phi)(a)(M)$ is a proof of $\Phi(a)$ true. Substitution is modelled by applying the matrix to the instance.

Universal introduction is formalized like implication introduction. A condition for the truth of $\forall(\Phi)$ is that $\Phi(x)$ is true for arbitrary x. In Table 2 variable occurrence conditions are used for a *schematic proof of the judgement* $\Phi(x)$. We instead use a *proof of the schematic judgement* $\bigwedge_x{:}\iota.\Phi(x)$. The rule is given by:

$$\forall\mathrm{I} \quad : \quad (\textstyle\bigwedge_{x:\iota} \Phi(x) \text{ true}) \vdash_{\Phi:\iota\to o} \forall(\Phi) \text{ true}$$

Existential elimination uses both hypothetical and schematic judgements, and makes use of scoping to avoid side conditions:

$$\exists\mathrm{E} \quad : \quad \exists(\Phi) \text{ true}, (\Phi(x) \text{ true} \vdash_{x:\iota} \psi \text{ true}) \vdash_{\Phi:\iota\to o,\psi:o} \psi \text{ true}$$

Since ψ is bound outermost, there is no possibility that the x of the schematic judgement form occur free in an instance.

Induction makes use of scoping and higher–order judgements:

$$\mathrm{IND} \quad : \quad \Phi(0) \text{ true}, (\Phi(x) \text{ true} \vdash_{x:\iota} \Phi(\mathrm{succ}(x)) \text{ true}) \vdash_{\Phi:\iota\to o,x:\iota} \Phi(x) \text{ true}$$

The correctness of the formalization is expressed by the following theorem:

Theorem 4.1 (Adequacy for Theorems) *There is a (compositionally–defined) bijection between first–order natural deduction proofs of a formula ϕ of Peano arithmetic from assumptions $\phi_1, \ldots, \phi_n$ and normal forms M of type ϕ° true in Σ_{PA}, all of whose free variables are of type ι and ϕ_i° true $(1 \leq i \leq n)$.*

Proof. *It is straightforward to prove by induction on the length of derivations, that if* $A_1, \ldots, A_n \vdash_{\mathrm{PA}} A$ *is derivable, then*

$$\Gamma, x_1{:}A_1 \text{ true}, \ldots, x_n{:}A_n \text{ true} \vdash_{\Sigma_{\mathrm{PA}}} M : A_{n+1} \text{ true}$$

is derivable, where Γ *contains assignments of the form* $x{:}\iota$ *for the free object variables* x *occurring in the* A_i*'s and in* M*, and where* M *faithfully encodes instantiation and application of rules. Surjectivity can be proved by induction on the structure of the normal forms of type* ϕ true *(for* $\phi{:}o$*), keeping in mind the uniqueness of types and the Church–Rosser property.* □

Note that it is possible for M in the above proof to have free variables of type ι, even when $n = 0$ (*i.e.*, when there are no assumptions) and when ϕ has no free variables. This is true, for example, in a proof of

$$\forall x.\phi \supset \exists x.\phi(x).$$

It is a peculiarity of first–order logic that the assumption that the domain of quantification is non–empty is not made explicit in proofs.

The above proof illustrates the fact that judgements in the LF actually encode consequence relations that satisfy, in view of Theorem 2.1, weak forms of thinning, transitivity, and contraction.

We shall give two examples. In the first we present a proof of $\phi \supset (\psi \supset \phi)$ true as a well–typed term in the signature Σ_{PA}. Let x have type ϕ true, and let y have type ψ true. Abstracting the incomplete proof x with respect to y, we obtain a proof $\lambda y{:}\psi \text{ true}.x$ of ψ true $\vdash \phi$ true. Applying $\supset$I to this proof, we obtain the (incomplete) proof $\supset\mathrm{I}(\psi)(\phi)(\lambda y{:}\psi \text{ true}.x)$ of $\psi \supset \phi$ true. Abstracting with respect to x, and applying $\supset$I again, we obtain the complete proof

$$\supset\mathrm{I}(\phi)(\psi \supset \phi)(\lambda x{:}\phi \text{ true}.\supset\mathrm{I}(\psi)(\phi)(\lambda y{:}\psi \text{ true}.x))$$

of $\phi \supset (\psi \supset \phi)$ true.

In the second example we give evidence for the claim that the traditional notion of a derivable rule has a formal counterpart in the LF. We show that in the signature Σ_{PA} the elimination rule for the universal quantifier in Schroeder–Heister's style can be derived. The Schroeder–Heister rule is specified as follows:

$$\forall\mathrm{E}_{SH} \quad : \quad \forall(\Phi) \text{ true}, ((\Lambda_{x:\iota}\, \Phi(x) \text{ true}) \vdash \psi \text{ true}) \text{ true} \vdash \psi \text{ true}) \vdash_{\Phi:\iota\to o, \psi:o} \psi \text{ true}.$$

It can be easily verified that the term

$$\lambda\Phi{:}\iota \to o.\lambda\psi{:}o.\lambda p{:}\forall(\Phi) \text{ true}.\lambda q{:}((\bigwedge_{x:\iota} \Phi(x) \text{ true}) \vdash \psi \text{ true}).q(\lambda x{:}\iota.\forall\mathrm{E}(\Phi)(x)(p))$$

has the above type.

With regard to derived rules, it is interesting to point out that in view of the fact that thinning is an admissible rule of the LF type theory, judgements are "open" concepts. This precludes an induction principle on proofs. Therefore typical admissible rules for a given logic $\mathcal{L}$, or meta rules such as the deduction theorem for a Hilbert–style presentation of first–order logic, are not directly derivable in certain adequate signatures for $\mathcal{L}$.

Turning to the formalization of higher–order logic, we formalize the inference rules in a manner quite similar to that of first–order logic. There is one judgement form, the assertion that ϕ is true, for ϕ an object of sort o.

$$\text{true} \quad : \quad \text{obj}(o) \to \text{Type}$$

The rules of β and η–conversion for the λ–calculus appear as axioms about equality. They are schematic in the domain and range sorts of the functions, and in the terms themselves:

$$\beta \quad : \quad \bigwedge_{\sigma:\text{sorts},\tau:\text{sorts},f:\text{obj}(\sigma)\to\text{obj}(\tau),a:\text{obj}(\sigma)} \text{ap}_{\sigma,\tau}(\Lambda_{\sigma,\tau}(f), a) =_\tau f(a) \text{ true}$$

$$\eta \quad : \quad \bigwedge_{\sigma:\text{sorts},\tau:\text{sorts},f:\text{obj}(\sigma\Rightarrow\tau)} \Lambda_{\sigma,\tau}(\lambda x{:}\,\text{obj}(\sigma).\,\text{ap}_{\sigma,\tau}(f, x)) =_{\sigma\Rightarrow\tau} f \text{ true}$$

Strictly speaking, the equations in the above axioms should be written using ap, for the type of $=_\tau$ is $\text{obj}(\tau \Rightarrow \tau \Rightarrow o)$.

The formalization of the logical rules is similar to that of first–order logic. The universal introduction and elimination rules are formalized as follows:

$$\forall\text{I} \quad : \quad (\bigwedge_{x:\text{obj}(\sigma)} \text{ap}_{\sigma,o}(f, x) \text{ true}) \vdash_{\sigma:\text{sorts},f:\text{obj}(\sigma\Rightarrow o)} \forall_\sigma(f) \text{ true}$$

$$\forall\text{E} \quad : \quad \forall_\sigma(f) \text{ true} \vdash_{\sigma:\text{sorts},f:\text{obj}(\sigma\Rightarrow o),a:\text{obj}(\sigma)} \text{ap}_{\sigma,o}(f, a) \text{ true}$$

The adequacy of this representation of higher–order logic can be established by means similar to that for Peano arithmetic.

5 Comparison with Related Work

Work in the area of proof checking began with the AUTOMATH project [4]. They sought to build a framework for expressing arbitrary mathematical texts in a formal way, and developed many examples, notably the formalization of Landau's textbook on Analysis by Jutting [12]. In contrast to the LF approach they work directly within the type theory, using the propositions as types principle. They do not seem to have isolated any general principles about the formalization of logic. Our work can be seen as a development of the AUTOMATH ideas by providing a framework that keeps the meta– and object level clearly separated. We are also concerned with supporting interactive proof development, particularly automated proof assistance, an area that was never considered by the AUTOMATH project.

Paulson's Isabelle system, as presented in [18], is a generalization of LCF to an arbitrary logic. He is primarily concerned with theorem proving, rather than proof checking and proof editing. Consequently his approach is quite different from ours, particularly in the treatment of rules and proofs. Isabelle avoids the construction of proof trees by viewing proof search as a process of building derived rules of inference. His representation of rules is based on a direct encoding of rules and their side conditions, using a clever algorithm due to Lincoln Wallen to enforce variable occurrence conditions.

6 Directions for Future Research

The LF system is a first step towards developing a general theory of interactive proof checking and proof construction. Much more work remains to be done. At present we do not treat definitions and abbreviations for an object logic. There appear to be at least two ways in which the LF type theory might be extended to include an account of definitional equality. One way is to add axioms for δ reductions [4,23]. We have not yet conducted a thorough analysis of such an extension. Another approach is to formalize LF type theory as an equational theory, with a set of equations representing definitions being included as part of the signature of the logic. These equations may be directed to obtain a reduction relation suitable for use by the type checker, but in general this relation will not be Church–Rosser or normalizing, and so decidability of type checking is lost.

It would be interesting to develop a characterization of the class of logics that can be formalized within the LF. It is clear from recent results of Avron and Mason [1,2] that one can exploit multiple judgement forms to encode logics that ordinarily have complex side conditions on their rules. While it appears that almost any formal system can be represented in the LF, some representations seem more natural than others. Is there a precise characterization of naturality in this sense? If so, what logics admit natural representations?

In a natural representation of a logic, the variables of the object language are represented by variables of the metalanguage. This means that, for the case of first–order logic, that the type ι can be viewed as the domain of quantification in a given model. Thus a satisfactory account of our treatment of variables seems to involve a notion of model for the LF. We have defined a class of models for which the type theory is complete. It is interesting to consider the possibility of connections between the LF and Burstall and Goguen's institutions [10] and Barwise's abstract model theory [3].

A general treatment of tactics is clearly desirable. The terms representing proofs in an LF encoding of a logic can be viewed as validations (in the sense of LCF [11].) Since the proof terms are total functions, a tactic that is validated by a proof term has the property that any proofs of the subgoals are guaranteed to lead to a proof of the goal (such tactics are called *valid* by Milner). Griffin's implementation of the LF demonstrates that this property can be checked automatically for a small class of tactics

called *refinement rules*. Constable and Knoblock [13] carry this idea even further by considering the possibility of proving the validity of tactics for type theory in the type theory itself. It would be interesting to adapt these ideas to the more general setting of the LF. In another direction we have defined a logic–independent search space that generalizes Paulson's higher–order resolution [18].

References

[1] Arnon Avron. *Simple Consequence Relations.* Technical Report, Laboratory for the Foundations of Computer Science, Edinburgh University, 1987. In preparation.

[2] Arnon Avron and Ian Mason. *Case Studies in the Edinburgh Logical Framework.* Technical Report, Laboratory for the Foundations of Computer Science, Edinburgh University, 1987. In preparation.

[3] J. Barwise and S. Feferman, editors. *Model–Theoretic Logics. Perspectives in Mathematical Logic*, Springer-Verlag, 1985.

[4] Nicolas G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606, Academic Press, 1980.

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

[6] Robert L. Constable, *et. al. Implementing Mathematics with the NuPRL Proof Development System.* Prentice–Hall, Englewood Cliffs, NJ, 1986.

[7] Thierry Coquand. *Une théorie des constructions.* Thèse de Troisième Cycle, Université Paris VII, January 1985.

[8] Thierry Coquand and Gérard Huet. Constructions: a higher–order proof system for mechanizing mathematics. In B. Buchberger, editor, *EUROCAL '85: European Conference on Computer Algebra*, pages 151–184, Springer-Verlag, 1985.

[9] Diedrik T. van Daalen. *The Language Theory of AUTOMATH.* PhD thesis, Technical University of Eindhoven, Eindhoven, Netherlands, 1980.

[10] Joseph Goguen and Rod Burstall. Introducing Institutions. In E. Clarke and D. Kozen, editors, *Logics of Programs*, pages 221–256, Springer-Verlag, 1984.

[11] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation.* Volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, 1979.

[12] L. S. Jutting. *Checking Landau's Grundlagen in the AUTOMATH System.* PhD thesis, Eindhoven University, The Netherlands, 1977.

[13] Todd B. Knoblock and Robert L. Constable. Formalized metareasoning in type theory. In *Proceedings of the Symposium on Logic in Computer Science*, pages 237–248, 1986.

[14] Per Martin-Löf. An intuitionistic theory of types: predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium, '73*, pages 73–118, North-Holland, Amsterdam, 1973.

[15] Per Martin-Löf. *On the Meanings of the Logical Constants and the Justifications of the Logical Laws.* Technical Report 2, Scuola di Specializzazione in Logica Matematica, Dipartimento di Matematica, Università di Siena, 1985.

[16] Albert Meyer and Mark Reinhold. 'Type' is not a type: preliminary report. In *Proceedings of the 13th ACM Symposium on the Principles of Programming Languages*, 1986.

[17] Bengt Nordström, Kent Petersson, and Jan Smith. *An Introduction to Martin-Löf's Type Theory.* University of Göteborg, Göteborg, Sweden, 1986. Preprint.

[18] Lawrence Paulson. Natural deduction proof as higher–order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[19] Kent Petersson. *A Programming System for Type Theory.* Technical Report 21, Programming Methodology Group, University of Göteborg/Chalmers Institute of Technology, March 1982.

[20] Dag Prawitz. *Natural Deduction: A Proof-Theoretical Study.* Almquist & Wiksell, Stockholm, 1965.

[21] Joseph R. Schoenfield. *Mathematical Logic.* Addison–Wesley, Reading, Massachusetts, 1967.

[22] Peter Schroeder–Heister. A natural extension of natural deduction. *Journal of Symbolic Logic*, 49(4), December 1984.

[23] Sören Stenlund. *Combinators, λ-terms, and Proof Theory.* D. Reidel, Dordrecht, Holland, 1972.