Using Typed Lambda Calculus to Implement Formal Systems on a Machine

Arnon Avron*Furio Honsell†Ian A. Mason‡and Robert Pollack§

January 15, 1992

1 Introduction

Since the early eighties there has been a growing interest in using computers as an aid for correctly manipulating formal systems. Much research has been devoted in building computer systems for checking proofs (Automath project [9, 26] and the Theory of Constructions [12]) or for developing interactively correct proofs (Edinburgh and Gothenburg LCF [22, 35, 39]) and Nuprl [11] in specific logical systems. However, implementing a proof environment for a specific logical system is both complex and time-consuming, this—together with the proliferation of logics—suggests that a uniform and reliable alternative is desirable. One such alternative is the Edinburgh Logical Framework (LF), developed in the late eighties at the LFCS (Laboratory for Foundations of Computer Science). The LF is a logic-independent tool which, given a specification for a logical system, synthesizes a proof editor and checker for that system. Its specification language is based on a general theory of logics, which enables one to capture uniformities and idiosyncrasies of a large class of logics without sacrificing generality for tractability. Peculiarities (such as side conditions on rule application, variable occurrence or formula formation) are expressed at the level of the specification The goal of Automath was more general than proof-checking and had in fact very similar concerns to those of this paper and the LF. In particular the idea of having an operator T : Prop $\rightarrow$ Type appears already in De Bruijn's earlier work, as does the idea of having several judgements.

The paper [24] describes the basic features of the LF. In this paper we are going to provide a broader illustration of its applicability and discuss to what extent it is successful. The analysis (of the formal presentation) of a system carried out through encoding often illuminates the system itself. This paper will also deal with this phenomenon.

*Department of Computer Science, Tel-Aviv University, Tel-Aviv, Israel

†Dipartimento di Matematica e Informatica , Universita' di Udine, via Zanon, 6, Udine, Italy

‡Computer Science Department, Stanford University, Stanford, California, U.S.A., 94305

§Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, Scotland, EH9 3JZ

Most of the results presented in this paper were obtained by the authors in the years 1986-1988 while working at the LF Project at the LFCS in Edinburgh. An early version of this paper (written by the first three authors) circulated in 1987 as an LFCS Technical Report [6]. A gentle abridged version of that report appears in [5]. At that time the only implementation of the LF was a version written by Timothy Griffin in the Cornell Synthesizer Generator [23]. Since then Robert Pollack implemented the LF in his LEGO system, described by him in the seventh section of this paper.

Since this paper was submitted (in 1987) for publication a great deal of research has been carried out in the area of Computer Aided Formal Reasoning, Interactive proof development environments and Program Extraction from proofs. These recent results and achievements are clearly relevant to the material discussed in this paper. Many of them actually rely heavily on some of the techniques presented here. But due to their great number and diversity of perspectives it is an impossible task to account for all of them in a reasonable amount of space. We therefore limit ourselves to suggest that the interested reader look at the Proceedings of the two Annual Workshops held under the auspices of the ESPRIT Basic Research Action 3245 "Logical Frameworks: Design, Implementation and Experiment". A selection of papers presented in the first Workshop are collected in the book [25]. Among the recent work most closely related to the topics presented here we mention Frank Pfenning [42] and Amy Felty [17, 18, 19].

This paper is organized as follows. In section 2 we introduce the LF specification language (or type theory). In section 3 we discuss the LF paradigm for specifying a logical system. The subsequent sections illustrate this paradigm. Section 4 deals with modal logics. Section 5 deals with various theories of functions, including relevant and linear lambda calculus. Section 6 we discuss program logics. Finally in section 7 we shall discuss the implementations.

The authors would like to express their gratitude to the members of the LFCS that made the two years so productive. In particular: Rod Burstall, Robert Harper, Robin Milner, Gordon Plotkin, and Don Sannella. We would also like to thank the referee for some interesting remarks which have been included in this version of the paper. Support for this work was provided by the Science and Engineering Research Council of the United Kingdom under grant number GR/D 64612 (Computer Assisted Formal Reasoning: Logics and Modularity), by the ESPRIT Basic Research Action grant 3245 (Logical Frameworks: Design, Implementation and Experiment), and by Italian MURST grants.

2 The Edinburgh Logical Framework

The LF specification language is a weak constructive type theory, more specifically a Π-typed λ-calculus, closely related to AUT-PI and AUT-QE [9], to Martin Löf's early type theories and to Meyer and Reinhold's λ^π [34]. The expressive power of this language suffices to specify the language of a logic, its axioms, rules and its proofs.

A typed λ-calculus can be used as a specification language for formal systems because syntax and rules are typically presented schematically. Moreover rules are

usually treated as functional objects, mapping proofs of premisses to proofs of conclusions and proofs of lemmas (or conjectures) to complete proofs. Proof checking reduces to checking the correctness of instantiations of schemas and the application of rules (both basic and derived) to premisses or proofs thereof.

Consequently the LF, whenever possible, reduces: all forms of dependency and parameterization involved in defining and using a formal system to λ-abstractions; all forms of schematic instantiation to λ-application; all forms of substitution to β-reduction. These reductions are carried out in such a way that the correctness of any of the above activities can be enforced through type matching and checking.

The LF type theory is a calculus for establishing the correctness and equivalence (i.e. definitional equality) of certain constructions. These constructions involve four kinds of objects: functions; types, i.e. assertions about functions; type valued functions, i.e. predicates of the assertion language; and kinds, i.e. assertions about typed valued functions. The only type or kind constructor is the dependent product, Π.[1] 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 more compact version of the one given in [24]. The theory we shall deal with is a formal system for deriving assertions of one of the following shapes:

$$\begin{array}{ll} \Gamma \vdash_\Sigma K & K \text{ is a kind} \\ \Gamma \vdash_\Sigma A : K & A \text{ has kind } K \\ \Gamma \vdash_\Sigma M : A & M \text{ has type } A \end{array}$$

where the syntax is specified by the following grammar:

$$\begin{array}{lllll} \textit{Signatures} & \Sigma & ::= & \langle\rangle \mid \Sigma, c : K \mid \Sigma, c : A \\ \textit{Contexts} & \Gamma & ::= & \langle\rangle \mid \Gamma, x : A \\ \textit{Kinds} & K & ::= & \text{Type} \mid \prod_{x:A} . K \\ \textit{Type Families} & A & ::= & c \mid \prod_{x:A} . B \mid \lambda x : A.B \mid AM \\ \textit{Objects} & M & ::= & c \mid x \mid \lambda x : A.M \mid MN \end{array}$$

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 $\prod_{x:A} . B$ when x does not occur free in B. The inference rules of the LF type theory are listed below, according to which of the three forms of assertions they concern, α-conversion is assumed throughout.

1. **Valid Kinds**

$$\frac{}{\vdash \text{Type}} \tag{1}$$

$$\frac{\Gamma \vdash_\Sigma A : \text{Type} \quad \Gamma, x : A \vdash_\Sigma K}{\Gamma \vdash_\Sigma \prod_{x:A} . K} \tag{2}$$

[1] Often the Π is abbreviated to $\to$ when the dependency is trivial.

2. **Valid Elements of a Kind**

$$\frac{\Gamma \vdash_\Sigma \text{Type} \quad c : K \in \Sigma}{\Gamma \vdash_\Sigma c : K} \tag{3}$$

$$\frac{\vdash_\Sigma K \quad c \notin \text{Dom}(\Sigma)}{\vdash_{\Sigma, c:K} \text{Type}} \tag{4}$$

$$\frac{\Gamma \vdash_\Sigma A : \text{Type} \quad \Gamma, x : A \vdash_\Sigma B : \text{Type}}{\Gamma \vdash_\Sigma \prod_{x:A} . \, B : \text{Type}} \tag{5}$$

$$\frac{\Gamma \vdash_\Sigma A : \text{Type} \quad \Gamma, x : A \vdash_\Sigma B : K}{\Gamma \vdash_\Sigma \lambda x : A . B : \prod_{x:A} . \, K} \tag{6}$$

$$\frac{\Gamma \vdash_\Sigma B : \prod_{x:A} . \, K \quad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma BN : [N/x]K} \tag{7}$$

$$\frac{\Gamma \vdash_\Sigma A : K \quad \Gamma \vdash_\Sigma K' \quad \Gamma \vdash_\Sigma K =_{\beta\eta} K'}{\Gamma \vdash_\Sigma A : K'} \tag{8}$$

3. **Valid Elements of a Type**

$$\frac{\vdash_\Sigma A : \text{Type} \quad c \notin \text{Dom}(\Sigma)}{\vdash_{\Sigma, c:A} \text{Type}} \tag{9}$$

$$\frac{\Gamma \vdash_\Sigma A : \text{Type} \quad x \notin \text{Dom}(\Gamma)}{\Gamma, x : A \vdash_\Sigma \text{Type}} \tag{10}$$

$$\frac{\Gamma \vdash_\Sigma \text{Type} \quad M : A \in \Sigma \cup \Gamma}{\Gamma \vdash_\Sigma M : A} \tag{11}$$

$$\frac{\Gamma \vdash_\Sigma A : \text{Type} \quad \Gamma, x : A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma \lambda x : A . M : \prod_{x:A} . \, B} \tag{12}$$

$$\frac{\Gamma \vdash_\Sigma M : \prod_{x:A} . \, B \quad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma MN : [N/x]B} \tag{13}$$

$$\frac{\Gamma \vdash_\Sigma M : A \quad \Gamma \vdash_\Sigma A' : \text{Type} \quad \Gamma \vdash_\Sigma A =_{\beta\eta} A'}{\Gamma \vdash_\Sigma M : A'} \tag{14}$$

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. In rules (8) and (14) we need an auxiliary judgement which expresses beta-eta conversion, $=_{\beta\eta}$, between types of kind Type in the context Γ. This notion of well-typed conversion is defined via a corresponding notion of well typed beta-eta contraction. This, in turn, is defined in the obvious way between objects of the same type in a given context and types of the same kind in a given context. See [49] or [24] for more details. $N \twoheadrightarrow^*_{\beta\eta} P$ for some term P. The following theorem from [24] summarizes the basic theoretical facts about LF (here α ranges over the basic assertions of the type theory):

Theorem 1

1. **Thinning:** *thinning is an admissible rule: if* $\Gamma \vdash_\Sigma \alpha$ *and* $\Gamma, \Gamma' \vdash_{\Sigma,\Sigma'} \text{Type}$*, then* $\Gamma, \Gamma' \vdash_{\Sigma,\Sigma'} \alpha$.

2. **Transitivity:** *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* $M \twoheadrightarrow^*_{\beta\eta} M'$*, then* $\Gamma \vdash_\Sigma M' : A$*, and similarly for types.*

5. **Confluence:** *all well–typed terms are Church–Rosser, while in general this is false.*

6. **Strong Normalization:** *all well–typed terms are strongly normalizing.*

7. **Decidability:** *each of the three relations defined by the inference system of the LF is decidable, as is the property of being well–typed.*

8. **Predicativity:** *if* $\Gamma \vdash_\Sigma M : A$ *then the type free* λ*-term obtained by erasing all type information from* M *can be typed in the Curry type assignment system.*

3 The LF Paradigm for Specifying a Logical System

In the LF a logical system is specified by means of an LF valid signature (i.e. a signature, Σ, in which $\vdash_\Sigma \text{Type}$ can be derived, Σ is of course finite). The syntax (for a given logical system) is encoded, into the signature, by introducing a type for each syntactic category and a constant of appropriate functional type for each expression constructor. Object language variables and schematic variables are then modelled by LF variables of the appropriate type. A schematic expression, of a given syntactic category, in certain schematic variables is expressed as the λ-abstraction

of that expression with respect to those variables. Finally binding operators are modelled as expression constructors with arguments of functional type.

The LF paradigm for specifying and handling rules and proofs is centered on the notion of *judgement*. This notion was introduced by Martin-Löf [31] and corresponds to the notion of *assertion* of a formal system. However the LF does not commit itself to the intuitionistic viewpoint and extends the meaning of this notion. That part of the signature encoding the rules of a logical system is a list of declarations of judgement types of the appropriate kind (corresponding to the assertions of the system) and of constants of the appropriate higher order judgement type (corresponding to the rules and axioms of the system). Rule schemas are modelled by means of λ-abstractions. One of the major benefits of this approach is that proofs of theorems and of derived rules are treated on the same logical level.

An LF type encodes an *open concept*, i.e. no induction principle over the type is available. This implies that the notion of proof actually encoded is not merely the notion of *proof of logical theorem* in a fixed system. Using a judgement J, we encode a *consequence relation* definable in the formal system under consideration. A term of type $J(\phi) \to J(\psi)$ encodes a proof that "J holds of ψ" follows from the assumption that "J holds of ϕ". It does not just encode a function which transfers a proof that ϕ is a logical theorem to a proof that ψ is also a logical theorem. The system may even lack logical theorems altogether (see 3.1). A proof of a hypothetical judgement therefore corresponds to either a rule of derivation or a derivable rule of the system, not simply a rule of proof or an admissible rule. For example, a constant $\wedge I$ of type $T(\phi) \to T(\psi) \to T(\phi \wedge \psi)$ (which we have in the standard internalizations of classical and intuitionistic logics) would be inadequate for relevance or linear logic, despite the fact that $\phi \wedge \psi$ is a theorem of these logics whenever ϕ and ψ are. The reason is that it is not always possible to infer $\phi \wedge \psi$ from the two conjuncts in these logics.

The consequence relations which are directly encoded by the LF's judgements are ordinary single-conclusioned consequence relations [1]. A proof of a sequent $\phi_1, \ldots, \phi_n \vdash \psi$ is encoded by a term of type $J(\phi_1) \to J(\phi_2) \to \cdots J(\phi_n) \to J(\psi)$, where J is a judgement that is induced by $\vdash$. Note, however, that the type structure of the LF makes it possible to formulate and prove also higher-order logical facts about an internalized consequence relation (see example 3.1) or even logical facts relating two or more such relations (see examples 4.1, 4.2 and 5.1).

As was already emphasized above, in the LF we try to achieve as much uniformity as possible. The attempt to translate, whenever possible, the variables of a system with the LF variables is one way in which this attempt manifests itself. Here we have another: whenever possible, rules of arbitrary order are encoded using the single LF $\to$ primitive. For example, the full standard formulation of the elimination rule for $\vee$ (in classical logic) is

$$\frac{\Gamma_1, \phi \vdash \vartheta \qquad \Gamma_2, \psi \vdash \vartheta \qquad \Gamma_3 \vdash \phi \vee \psi}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \vartheta}$$

In this formulation $\vdash$ is the basic consequence relation between *sentences*, while the "fraction" symbol corresponds to a higher order rule concerning *sequents*. In the LF

both are translated using $\rightarrow$ by introducing a constant $\vee$E of type:

$$\prod_{\phi:o}\prod_{\psi:o}\prod_{\vartheta:o}(T(\phi)\rightarrow T(\vartheta))\rightarrow(T(\psi)\rightarrow T(\vartheta))\rightarrow T(\phi\vee\psi)\rightarrow T(\vartheta)$$

It is worth noting that $\vee$E corresponds, in fact, to something stronger than the rule it internalizes, since in the original rule the assumptions in the Γ_i's were intended to be just sentences, while in the LF the application of $\vee E$ is possible under any set of assumptions of arbitrary order.

The paradigm outlined above applies to a system when the language is given, roughly, by a context free grammar and when the rules concerning the non-binding constructors are *pure* [1] (i.e not subject to side conditions). Also a number of classical side conditions on rules concerning binding operators can be handled unproblematically. In other cases additional judgements need to be introduced. One aim of this paper is to illustrate, using examples, the major techniques that can be exploited. We remark that the implicit identification the LF utilizes between object language variables and schematic variables (over terms of the language) will often suggest similarities between the LF translation of a system and a denotational model of that system (see for example the internalizations of the various lambda calculii).

It is an *LF thesis* that well-behaved natural-deduction formalisms are those that can be directly encoded, and also that given a formal representation of a consequence relation the notion of a derivable rule (of arbitrary order) is *defined* by the non-emptiness of the type encoding its specification in the corresponding signature.

An LF specification of a formal system is satisfactory only if *adequate*, i.e. if for each syntactic and proof theoretic category of the system there is a *compositional* surjection from the LF type, corresponding to that category, onto the category itself. By compositional we mean that it commutes with substitution.

We finish off this section by giving two examples which exemplify the paradigm, as outlined above.

3.1 Kleene's Three-Valued Logic

3.1.1 The System

Kleene's three-valued logic provides an excellent example to how the LF paradigm works in practice. Syntactically, the propositional fragment of this logic is identical to classical propositional logic (with the connectives $\neg, \vee$ and $\wedge$). Semantically it is based on the truth-values 0, 1 and -1, of which only 1 is taken to be designated. The operations which correspond to the logical connectives are defined as follows:

$$\begin{array}{lcl}\phi\vee\psi & = & \min(\phi,\psi)\\ \phi\wedge\psi & = & \max(\phi,\psi)\\ \neg\phi & = & -\phi.\end{array}$$

What makes this example particularly interesting is the fact that the resulting logic has no theorems. No sentence has the designated value 1 under all possible assignments. Accordingly, this logic is only characterized by the consequence relation which is naturally associated with it:

- $\phi_1, \ldots, \phi_n \vdash_{\mathrm{Kl}} \psi$ if and only if $\nu(\psi) = 1$ for all assignments ν such that $\nu(\phi_1) = \ldots = \nu(\phi_n) = 1$.

The possibility of a good implementation of a logic in the LF always depends on the existence of a reasonable natural deduction representation for its consequence relation. In the case of Kleene's logic such a formal system is described in [8]. Its internalization in the LF is an easy task. The resulting signature is:

3.1.2 The Signature Σ_{Kl}

• Syntactic Categories

o : Type

• Operations

$\neg$: $o \to o$
$\wedge$: $o \to o \to o$
$\vee$: $o \to o \to o$

• Judgement

$\mathrm{T} : o \to \mathrm{Type}$

• Axioms and Rules

$\wedge\mathrm{I}$	:	$\Pi_{\phi,\psi:o}$	$\mathrm{T}(\phi) \to \mathrm{T}(\psi) \to \mathrm{T}(\phi \wedge \psi)$
$\vee\mathrm{I}_l$	:	$\Pi_{\phi,\psi:o}$	$\mathrm{T}(\phi) \to \mathrm{T}(\phi \vee \psi)$
$\vee\mathrm{I}_r$	:	$\Pi_{\phi,\psi:o}$	$\mathrm{T}(\psi) \to \mathrm{T}(\phi \vee \psi)$
$\neg\neg\mathrm{I}$	:	$\Pi_{\phi:o}$	$\mathrm{T}(\phi) \to \mathrm{T}(\neg\neg\phi)$
$\neg\wedge\mathrm{I}_l$	:	$\Pi_{\phi,\psi:o}$	$\mathrm{T}(\neg\phi) \to \mathrm{T}(\neg(\phi \wedge \psi))$
$\neg\wedge\mathrm{I}_r$	:	$\Pi_{\phi,\psi:o}$	$\mathrm{T}(\neg\psi) \to \mathrm{T}(\neg(\phi \wedge \psi))$
$\neg\vee\mathrm{I}$	:	$\Pi_{\phi,\psi:o}$	$\mathrm{T}(\neg\phi) \to \mathrm{T}(\neg\psi) \to \mathrm{T}(\neg(\phi \vee \psi))$
$\neg\mathrm{E}$	:	$\Pi_{\phi,\psi:o}$	$\mathrm{T}(\phi) \to \mathrm{T}(\neg\phi) \to \mathrm{T}(\psi)$
$\wedge\mathrm{E}_l$	:	$\Pi_{\phi,\psi:o}$	$\mathrm{T}(\phi \wedge \psi) \to \mathrm{T}(\phi)$
$\wedge\mathrm{E}_r$	:	$\Pi_{\phi,\psi:o}$	$\mathrm{T}(\phi \wedge \psi) \to \mathrm{T}(\psi)$
$\neg\neg\mathrm{E}$	:	$\Pi_{\phi:o}$	$\mathrm{T}(\neg\neg\phi) \to \mathrm{T}(\phi)$
$\neg\vee\mathrm{E}_l$	:	$\Pi_{\phi,\psi:o}$	$\mathrm{T}(\neg(\phi \vee \psi)) \to \mathrm{T}(\neg\phi)$
$\neg\vee\mathrm{E}_r$	:	$\Pi_{\phi,\psi:o}$	$\mathrm{T}(\neg(\phi \vee \psi)) \to \mathrm{T}(\neg\psi)$
$\vee\mathrm{E}$	:	$\Pi_{\phi,\psi,\vartheta:o}$	$(\mathrm{T}(\phi) \to \mathrm{T}(\vartheta)) \to (\mathrm{T}(\psi) \to \mathrm{T}(\vartheta)) \to (\mathrm{T}(\phi \vee \psi) \to \mathrm{T}(\vartheta))$
$\neg\wedge\mathrm{E}$	:	$\Pi_{\phi,\psi,\vartheta:o}$	$(\mathrm{T}(\neg\phi) \to \mathrm{T}(\vartheta)) \to (\mathrm{T}(\neg\psi) \to \mathrm{T}(\vartheta)) \to (\mathrm{T}(\neg(\phi \wedge \psi)) \to \mathrm{T}(\vartheta))$

We present an example of an important *higher-order* logical fact about $\vdash_{\mathrm{Kl}}$: by adding the excluded-middle as an axiom to the corresponding natural deduction formalism the usual classical introduction rule for negation becomes derivable. Thus such an addition is sufficient for obtaining classical logic.

• Example of a Proof.

Δ : $\Pi_{\phi,\psi:o}$ $\mathrm{T}(\neg\phi \vee \phi) \to (\mathrm{T}(\phi) \to \mathrm{T}(\psi)) \to (\mathrm{T}(\phi) \to \mathrm{T}(\neg\psi)) \to \mathrm{T}(\neg\phi)$

where the term Δ is defined to be the following:

$$\lambda\phi : o.\lambda\psi : o.\lambda X : \mathrm{T}(\neg\phi \vee \phi).\lambda Y : \mathrm{T}(\phi) \to \mathrm{T}(\psi).\lambda Z : \mathrm{T}(\phi) \to \mathrm{T}(\neg\psi).$$
$$\vee E(\neg\phi)(\phi)(\neg\phi)(\lambda v : \mathrm{T}(\neg\phi).v)(\lambda W : \mathrm{T}(\phi).\neg E(\psi)(\neg\phi)(Y(W))(Z(W)))X$$

To illustrate how one goes about constructing such terms (as well as their relationship to the corresponding natural deduction proofs) we treat this first non-trivial example in more detail. Suppose that:

1. P_X is a proof of $\neg\phi\vee\phi$, and X is the corresponding LF term (i.e. $X : \mathrm{T}(\neg\phi\vee\phi)$).
2. P_Y is a proof that ψ follows from the assumption ϕ, and Y is the corresponding LF term (i.e. $Y : \mathrm{T}(\phi) \to \mathrm{T}(\psi)$).
3. P_Z is a proof that $\neg\psi$ follows from the assumption ϕ, and Z is the corresponding LF term (i.e. $Z : \mathrm{T}(\phi) \to \mathrm{T}(\neg\psi)$).

From these we shall construct a proof of $\neg\phi$ and the corresponding LF term of type $\mathrm{T}(\neg\phi)$. The last rule used in this proof will be (an instance of) $\vee$E, and can be represented graphically by:

$$\begin{array}{lccc} & (\neg\phi) & (\phi) & \\ & \vdots & \vdots & \vdots \\ \vee\mathrm{E} & \neg\phi & \neg\phi & \neg\phi \vee \phi \\ \hline & & \neg\phi & \end{array}$$

Let $P_{L_1}, P_{L_2}, P_{L_3}$ denote the three subproofs (or lemmas) and L_1, L_2, L_3 denote the corresponding three LF terms. Specifically: P_{L_1} is the derivation of $\neg\phi$ from the assumption $\neg\phi$; P_{L_2} is the derivation of $\neg\phi$ from the assumption ϕ; while P_{L_3} is the derivation of $\neg\phi \vee \phi$. Now P_{L_1} is a one line proof, which in the LF is represented by the appropriate identity function. So in this case L_1 is the term: $\lambda v : \mathrm{T}(\neg\phi).v$ The proof P_{L_3} may be taken to be P_X and so we may equate L_3 with X. Thus the only non-trivial lemma to establish is P_{L_2}. So assume that ϕ is true and that W is the corresponding element of $\mathrm{T}(\phi)$. Then we use (an instance of) the rule $\neg$E:

$$\begin{array}{lcc} & \vdots & \vdots \\ \neg\mathrm{E} & \psi & \neg\psi \\ \hline & \neg\phi & \end{array}$$

The LF terms corresponding to the subproofs are $Y(W)$ and $Z(W)$ of types $\mathrm{T}(\psi)$ and $\mathrm{T}(\neg\psi)$ respectively, so the term corresponding to the rule application is

$$\neg\mathrm{E}(\psi)(\neg\phi)(Y(W))(Y(Z))$$

which is of type $\mathrm{T}(\neg\phi)$.Thus discharging the assumption concerning the truth of ϕ yields

$$(\lambda W : \mathrm{T}(\phi).\neg\mathrm{E}(\psi)(\neg\phi)(Y(W))(Y(Z))) : \mathrm{T}(\neg\phi).$$

This is the term required to establish the third lemma. The final result, represented by

$$\vee\mathrm{E}(\neg\phi)(\phi)(\neg\phi)(\lambda v : \mathrm{T}(\neg\phi).v)(\lambda W : \mathrm{T}(\phi).\neg\mathrm{E}(\psi)(\neg\phi)(Y(W))(Z(W)))X,$$

may be pictured thus:

$$\dfrac{\neg\phi \qquad \dfrac{\dfrac{\dfrac{(\phi)}{\vdots}}{P_Y}}{\vdots}\ \neg\mathrm{E} \quad \dfrac{\dfrac{(\phi)}{\vdots}}{P_Z}}{\psi \qquad \neg\psi}\ \neg\mathrm{E} \qquad \dfrac{P_X}{\vdots}}{\neg\phi \qquad\qquad \neg\phi \qquad\qquad \neg\phi \vee \phi}\ \vee\mathrm{E}$$

$$\neg\phi$$

The final term Δ is obtained by firstly discharging (using λ) the assumptions X, Y, and Z and then making the result (again using λ) schematic or parametric in the formulas ψ and ϕ.

Adequacy and Faithfulness Property 1 *The following hold*

$$\phi_1, \ldots, \phi_n \vdash_{\mathrm{Kl}} \psi$$

iff there exists a term t such that:

$$p_1 : o, \ldots, p_m : o \vdash_{\Sigma_{\mathrm{Kl}}} t : \mathrm{T}(\phi_1) \to \cdots \to \mathrm{T}(\phi_n) \to \mathrm{T}(\psi)$$

Where $p_1, \ldots, p_m$ are the atomic variables occuring in $\phi_i, 1 \leq i \leq n$ and ψ (we use the same symbol for denoting a formula and the corresponding term in LF). Moreover, there is a compositional bijection between proofs in the natural deduction system and proof terms such as t above.

The above signature is quite similar to that of classical propositional logic (see below).[2] Moreover, the translation process in both cases proceeds according to the same lines. There is one big difference, though. Unlike the classical (or intuitionistic) case, the $\to$ of the LF does not correspond here to any connective of the logic (not even a definable one.). The types of many of the terms of the present signature (including some of its constants) *strictly* correspond, therefore, to higher order sequents and consequence relations. Thus the actual meaning of $\vee E$ is:

$$(\Gamma_1, \phi \vdash^1_{\mathrm{Kl}} \vartheta), (\Gamma_2, \psi \vdash^1_{\mathrm{Kl}} \vartheta), (\Gamma_3 \vdash^1_{\mathrm{Kl}} \phi \vee \psi) \vdash^2_{\mathrm{Kl}} (\Gamma_1, \Gamma_2, \Gamma_3 \vdash^1_{\mathrm{Kl}} \vartheta)$$

The fact that $\to$ can indeed be used to handle both $\vdash^1_{\mathrm{Kl}}$ and $\vdash^2_{\mathrm{Kl}}$ is in a perfect accordance to the LF's paradigm.

3.2 First Order Logic with a Choice Operator

3.2.1 The System

In [24] the signature of first order logic was presented in detail. Rather than duplicate it here we present a version with an additional binding operator. Thus the logic we

[2] The splitting of the of the $\neg$ rules into introduction and elimination schemas, for its combination with other connectives, is a known idea also in the context of classical logic, see e.g. [50], p. 66.

present here illustrates, with three examples, how one handles binding operators in the LF. Apart from the quantifiers $\exists$ and $\forall$ we include ϵ, a version of Hilbert's choice operator. If x is a variable of type i, then $x = x$ is a term of type o. We can bind x by λ–abstraction obtaining an object of type $i \to o$, $\lambda x : i.x = x$. The binding operators applied to this give

1. $\epsilon(\lambda x : i.x = x)$, which represents the first–order term $\epsilon x.x = x$.
2. $\exists(\lambda x : i.x = x)$, which represents the first–order formula $\exists x.x = x$.
3. $\forall(\lambda x : i.x = x)$, which represents the first–order formula $\forall x.x = x$.

The only rule concerning the choice operator is the introduction rule:

$$\frac{\exists x \Phi(x)}{\Phi(\epsilon x \Phi(x))}$$

3.2.2 The Signature $\Sigma_{\epsilon 1^o}$

This signature encodes the consequence relation of *truth* rather than that of *validity*, see [1].

• Syntactic Categories

i : Type
o : Type

• Operations

$=$: $i \to i \to o$
$\neg$: $o \to o$
$\supset$: $o \to o \to o$
ϵ : $(i \to o) \to i$
$\exists$: $(i \to o) \to o$
$\forall$: $(i \to o) \to o$

• Judgement

T : $o \to$ Type

- Axioms and Rules

$$
\begin{array}{lcll}
\mathrm{E}_0 & : & \Pi_{x:i} & \mathrm{T}(x = x) \\
\mathrm{E}_1 & : & \Pi_{\substack{x,y:i \\ t:i\to i}} & \mathrm{T}(x = y) \to \mathrm{T}(t(x) = t(y)) \\
\mathrm{E}_2 & : & \Pi_{\substack{x,y:i \\ \Phi:i\to o}} & \mathrm{T}(x = y) \to \mathrm{T}(\Phi(x)) \to \mathrm{T}(\Phi(y)) \\
\neg\mathrm{I} & : & \Pi_{\phi,\psi:o} & (\mathrm{T}(\phi) \to \mathrm{T}(\psi)) \to (\mathrm{T}(\phi) \to \mathrm{T}(\neg\psi)) \to \mathrm{T}(\neg\phi) \\
\wedge\mathrm{I} & : & \Pi_{\phi,\psi:o} & \mathrm{T}(\phi) \to \mathrm{T}(\psi) \to \mathrm{T}(\phi \wedge \psi) \\
\vee\mathrm{I}_l & : & \Pi_{\phi,\psi:o} & \mathrm{T}(\phi) \to \mathrm{T}(\phi \vee \psi) \\
\vee\mathrm{I}_r & : & \Pi_{\phi,\psi:o} & \mathrm{T}(\psi) \to \mathrm{T}(\phi \vee \psi) \\
\wedge\mathrm{E}_l & : & \Pi_{\phi,\psi:o} & \mathrm{T}(\phi \wedge \psi) \to \mathrm{T}(\phi) \\
\wedge\mathrm{E}_r & : & \Pi_{\phi,\psi:o} & \mathrm{T}(\phi \wedge \psi) \to \mathrm{T}(\psi) \\
\neg\neg\mathrm{E} & : & \Pi_{\phi:o} & \mathrm{T}(\neg\neg\phi) \to \mathrm{T}(\phi) \\
\vee\mathrm{E} & : & \Pi_{\phi,\psi,\vartheta:o} & (\mathrm{T}(\phi) \to \mathrm{T}(\vartheta)) \to (\mathrm{T}(\psi) \to \mathrm{T}(\vartheta)) \to (\mathrm{T}(\phi \vee \psi) \to \mathrm{T}(\vartheta)) \\
\supset\mathrm{E} & : & \Pi_{\phi_1,\phi_2:o} & \mathrm{T}(\phi_1 \supset \phi_2) \to \mathrm{T}(\phi_1) \to \mathrm{T}(\phi_2) \\
\supset\mathrm{I} & : & \Pi_{\phi,\psi:o} & (\mathrm{T}(\phi) \to \mathrm{T}(\psi)) \to \mathrm{T}(\phi \supset \psi) \\
\epsilon\mathrm{I} & : & \Pi_{\Phi:i\to o} & \mathrm{T}(\exists(\Phi)) \to \mathrm{T}(\Phi(\epsilon(\Phi))) \\
\exists\mathrm{E} & : & \Pi_{\substack{\Phi:i\to o \\ \phi:o}} & \mathrm{T}(\exists(\Phi)) \to (\Pi_{x:i}\mathrm{T}(\Phi(x)) \to \mathrm{T}(\phi)) \to \mathrm{T}(\phi) \\
\exists\mathrm{I} & : & \Pi_{\substack{\Phi:i\to o \\ t:i}} & \mathrm{T}(\Phi(t)) \to \mathrm{T}(\exists(\Phi)) \\
\forall\mathrm{E} & : & \Pi_{\substack{\Phi:i\to o \\ t:i}} & \mathrm{T}(\forall(\Phi)) \to \Phi(t) \\
\forall\mathrm{I} & : & \Pi_{\Phi:i\to o} & (\Pi_{t:i}\mathrm{T}(\Phi(t))) \to \mathrm{T}(\forall(\Phi))
\end{array}
$$

- Example of a Proof

$$\Delta \quad : \quad \Pi_{\Phi:i\to o} \quad \mathrm{T}(\forall(\Phi)) \to \mathrm{T}(\exists(\Phi))$$

In the example above Δ is the following term

$$\lambda\Phi : i \to o \;\; . \;\; \lambda\rho : \mathrm{T}(\forall(\Phi)) \;\; . \;\; \exists\mathrm{I}(\Phi)(\epsilon(\Phi))(\forall\mathrm{E}(\Phi)(\epsilon(\Phi))(\rho))$$

We should point out that if we removed the choice operator from the signature this example would no longer be provable, unless one explicitly added a constant of type i.

Adequacy and Faithfulness Property 2 *We shall state the adequacy only for a monadic language, the general case follows exactly the same pattern. Letting*

$$\Gamma = \{x_1 : i, \ldots x_n : i, X_1 : i \to o, \ldots X_m : i \to o\}$$

the following hold:

1. $\Gamma \vdash M : i$ *iff* $\Phi_\Gamma(M)$ *is a well formed term of first order logic with a choice operator whose only free individual variables are among* $x_1, \ldots, x_n$ *and whose unary relations are among the* $X_1, \ldots, X_m$.

2. $\Gamma \vdash M : o$ *iff* $\Phi_\Gamma(M)$ *is a well formed formula of first order logic with a choice operator whose only free individual variables are among* $x_1, \ldots, x_n$ *and whose unary relations are among the* $X_1, \ldots, X_m$.

3. $\Gamma \cup \{y_1 : \mathrm{True}(\phi_1), \ldots, y_k : \mathrm{True}(\phi_k)\} \vdash M : \mathrm{True}(\phi)$

iff

$$\Phi_\Gamma(\phi_1), \ldots, \Phi_\Gamma(\phi_k) \vdash \Phi_\Gamma(\phi).$$

where Φ_Γ *is a bijective function*

$$\Phi_\Gamma : \Xi_\Gamma(i) \cup \Xi_\Gamma(o) \to \epsilon 1^\circ$$

to be defined shortly. $\epsilon 1^\circ$ *denotes the collection of terms and formulas of first order logic with a choice operator whose only free individual variables are among* $x_1, \ldots, x_n$ *and whose unary relations are among the* $X_1, \ldots, X_m$. $\Xi_\Gamma(\tau)$ *is the set of long* $\beta\eta$ *normal forms of type* τ *in the context* Γ. *Finally*

$$\Phi_\Gamma(M) = \begin{cases} x & \text{if } M \equiv x \\ \Phi_\Gamma(M') = \Phi_\Gamma(N) & \text{if } M \equiv (M' = N) \\ \epsilon(\Phi_{\Gamma,x:i}(P[x])) & \text{if } M \equiv \epsilon(\lambda x : i.P[x]) \\ \neg\Phi_\Gamma(M') & \text{if } M \equiv \neg M' \\ \Phi_\Gamma(M') \supset \Phi_\Gamma(N) & \text{if } M \equiv M' \supset N \\ \forall x.\Phi_{\Gamma,x:i}(M'[x]) & \text{if } M \equiv \forall(\lambda x : i.M'[x]) \\ \exists x.\Phi_{\Gamma,x:i}(M'[x]) & \text{if } M \equiv \exists(\lambda x : i.M'[x]) \\ X(\Phi_\Gamma(M')) & \text{if } M \equiv X(M'). \end{cases}$$

Throughout this paper we will identify terms up to α-equivalence, and assume that in notations such as $\forall x.\Phi_{\Gamma,x:i}(M'[x])$ we have $x \notin \mathrm{dom}(\Gamma)$.

4 Modal Logics

Modal logics are obtained from classical logic by adding to its language a new unary operator, $\Box$, together with certain axioms and rules controlling ts use. They may be regarded as the simplest representatives of a larger class of logics that are of interest in computer science. This class includes temporal logics [27] and generalized modal logics [51]. Encoding these logics poses serious difficulties. These are overcome, essentially, by taking full advantage of the ability within the LF to employ, simultaneously, several different consequence relations. In this section we present two examples of encoding modal logics and the problems involved.

4.1 Hilbert Style Modal Logics

4.1.1 The System

This case presents a new kind of problem: Standard Hilbert systems for modal logics usually have, apart from axiom schemes, two rules of inference: M.P. and the necessitation rule: from ϕ infer $\Box\phi$. The second one, however, is taken to be only a *rule of proof.* This means that its application to a sentence is permitted only if that sentence was shown to be a logical theorem of the system, but not if it depends on

assumptions. It is not the case, therefore, that $\Box\phi$ *follows* from ϕ in such systems. It would not be sound, accordingly, to internalize the necessitation rule by the standard method of introducing a constant Nec (say) of type $\prod_{\phi:o} \mathrm{T}(\phi) \to \mathrm{T}(\Box\phi)$ (where T corresponds to the intended consequence relation), since such a constant would force the deducibility of $\Box\phi$ from any set of assumptions which entail ϕ.

The solution to this problem is an example of the power gained by the LF ability to employ different judgements. The solution, in this case, is to introduce *two* judgements, "True" and "Valid". The first corresponds to the original consequence relation which we have in mind (in which necessitation is only a rule of proof). The second to the one which is obtained by taking both rules as pure rules of derivation. Following [1] we shall denote the first by $\vdash_t$ and the second by $\vdash_v$.

Before we proceed we should emphasize that $\vdash_t$ and $\vdash_v$ are two different consequence relations associated with one (Hilbert-type) formal system. Their definitions are independent of the LF and depend only on two ways of defining a deduction in the given system. In particular the way one of its two rules can be used in such deductions. In the present case (of modal logic) both relations have been investigated previously, [1, 16]. They have a clear semantical interpretation (see below). In the LF, however, the less standard one arises in a natural way, and the LF has the power to handle them simultaneously. Moreover, even if we are interested in only one of them, it is useful (from the LF point of view) to introduce the other.

4.1.2 The Signature $\Sigma_{\mathrm{H}\Box}$

The resulting signature in the particular case of S4 is:

• Syntactic Categories

o : Type

• Operations

$\neg$: $o \to o$

$\supset$: $o \to o \to o$

$\Box$: $o \to o$

• Judgements

True : $o \to$ Type

Valid : $o \to$ Type

• Axioms and Rules

C	:	$\Pi_{\phi:o}$	$\mathrm{Valid}(\phi) \to \mathrm{True}(\phi)$
A_1	:	$\Pi_{\phi_1,\phi_2:o}$	$\mathrm{Valid}(\phi_1 \supset (\phi_2 \supset \phi_1))$
A_2	:	$\Pi_{\phi_1,\phi_2,\phi_3:o}$	$\mathrm{Valid}(\phi_1 \supset (\phi_2 \supset \phi_3) \supset (\phi_1 \supset \phi_2) \supset (\phi_1 \supset \phi_3))$
A_3	:	$\Pi_{\phi_1,\phi_2:o}$	$\mathrm{Valid}((\neg\phi_1 \supset \neg\phi_2) \supset (\phi_2 \supset \phi_1))$
A_4	:	$\Pi_{\phi:o}$	$\mathrm{Valid}(\Box\phi \supset \phi)$
A_5	:	$\Pi_{\phi_1,\phi_2:o}$	$\mathrm{Valid}(\Box(\phi_1 \supset \phi_2) \supset (\Box\phi_1 \supset \Box\phi_2))$
A_6	:	$\Pi_{\phi:o}$	$\mathrm{Valid}(\Box\phi \supset \Box\Box\phi)$
MP_T	:	$\Pi_{\phi_1,\phi_2:o}$	$\mathrm{True}(\phi_1 \supset \phi_2) \to \mathrm{True}(\phi_1) \to \mathrm{True}(\phi_2)$
MP_V	:	$\Pi_{\phi_1,\phi_2:o}$	$\mathrm{Valid}(\phi_1 \supset \phi_2) \to \mathrm{Valid}(\phi_1) \to \mathrm{Valid}(\phi_2)$
Nec	:	$\Pi_{\phi:o}$	$\mathrm{Valid}(\phi) \to \mathrm{Valid}(\Box\phi)$

• Example of a Proof

$\Delta \quad : \quad \Pi_{\phi:o} \quad \mathrm{True}(\Box(\Box\phi \supset \phi) \supset \Box\phi) \to \mathrm{True}(\phi)$

Where Δ is the following term

$$\begin{array}{l} \lambda\phi : o.\lambda x : \mathrm{True}(\Box(\Box\phi \supset \phi) \supset \Box\phi). \\ \quad \mathrm{MP}_T(\Box\phi)(\phi)(\mathrm{C}(\mathrm{A}_4(\phi))) \\ \quad\quad (\mathrm{MP}_T(\Box(\Box\phi \supset \phi))(\Box\phi)(x) \\ \quad\quad\quad (\mathrm{C}(\Box(\Box\phi \supset \phi))(\mathrm{Nec}(\Box\phi \supset \phi)(\mathrm{A}_4(\phi))))) \end{array}$$

Note: $\Box(\Box\phi \supset \phi) \supset \Box\phi$ is the characteristic axiom of GL— the famous modal system for provability in PA. The above is a proof that in S4 any ϕ-instance of this formula actually entails ϕ.

Adequacy and Faithfulness Property 3 *The following hold:*

1. *There is a compositional surjection between proofs in the Hilbert-type system of S4 that $\phi_1, \ldots, \phi_n \vdash_t \psi$ and terms t such that:*

   $p_1 : o, \ldots, p_m : o \vdash_{\Sigma_{H\Box}} t : \mathrm{True}(\phi_1) \to \cdots \to \mathrm{True}(\phi_n) \to \mathrm{True}(\psi)$

   Where $p_1, \ldots, p_m$ are the atomic variables of ϕ.

2. *There is a compositional bijection between proofs in the Hilbert-type system of S4 that $\phi_1, \ldots, \phi_n \vdash_v \psi$ and terms t such that:*

   $p_1 : o, \ldots, p_m : o \vdash_{\Sigma_{H\Box}} t : \mathrm{Valid}(\phi_1) \to \cdots \to \mathrm{Valid}(\phi_n) \to \mathrm{Valid}(\psi)$

   Where $p_1, \ldots, p_m$ are the atomic variables of ϕ.

The above can be strengthened as follows:

- *There is a compositional surjection between LF-terms of type:*

  $(\dagger) \quad \mathrm{Valid}(\phi_1) \to \cdots \to \mathrm{Valid}(\phi_n) \to \mathrm{True}(\psi_1) \to \cdots \mathrm{True}(\psi_m) \to \mathrm{True}(\vartheta)$

  and proofs that $\psi_1, \ldots, \psi_m \vdash_t \vartheta$ in the Hilbert-type system which is obtained by adding $\phi_1, \ldots, \phi_n$ as axioms to S4.

The above method for handling rules of proof is not specific to modal logic. It is, in fact, rather general and the reader is referred to 4.5 for a further example. But in the realm of modal logic the corresponding consequence relations have a natural semantical interpretation in terms of Kripke models:

- $\phi_1, \ldots, \phi_n \vdash_v \psi$ iff ψ is *valid* in any frame (of the relevant class) in which $\phi_1, \ldots, \phi_n$ are all valid (i.e. true in all worlds).
- $\phi_1, \ldots, \phi_n \vdash_t \psi$ iff ψ is *true* in every world (of a frame from the appropriate class) in which $\phi_1, \ldots, \phi_n$ are all true.

It is important to note, finally, that the LF enables us to express and prove logical facts concerning both internalized consequence relations. For example a term of type $\dagger$ encodes a proof that ϑ is true in any world in which the $\bar{\psi}$ are all true, provided this world belongs to a frame in which the $\bar{\phi}$ are all valid.

4.2 Natural Deduction Style S4

4.2.1 The System

We now give an example of a known natural deduction formalism which is not "well-behaved" according to the LF standards. It is Prawitz's system for S4 [45]. This system is obtained from the usual natural deduction formulation of classical propositional calculus by the addition of the following two rules:

$$\frac{\phi}{\Box\phi} \qquad \frac{\Box\phi}{\phi}$$

The crucial point about this formalism is that there is a side condition on the application of the first rule (the introduction rule for $\Box$). Prawitz gives several possible versions of this side condition. In the first version all assumptions on which ϕ depends should be modal (i.e. the main connective is $\Box$). In the second these assumptions may be what he calls essentially modal, i.e., formulas which are obtained from modal formulas by arbitrary combinations of disjunction, conjunction and double-negation. In both versions the side condition makes this rule *impure* [1]. This generally means that the coherence which the LF paradigm expects between the formulation of the rules of a system and the consequence relation represented by it (see section 2) is broken. In the present case, for example the rule has the form

$$\frac{\phi}{\Box\phi},$$

but it is not the case that $\phi \vdash \Box\phi$ (in the sense that there is a proof of $\Box\phi$ from ϕ in the system).

The solution to the problem which is caused by the impurity of the rules for the $\Box$ is solved in the present case by separating the rules of the system into two groups and introducing two corresponding judgements. The rules of the first group are just the rules of classical logic. These rules induce the usual consequence relation of classical propositional logic. We denote by Taut the corresponding judgement in the LF internalization of the system. The second group consists of the special rules for $\Box$. We denote by Valid the judgement which corresponds to the whole system, including these two special rules. The constants of the internalization then fall into three groups: those corresponding to pure tautological inferences, those corresponding to the modal rules and those which relate the two sorts of inferences. The resulting signature is:

4.2.2 The Signature $\Sigma_{\mathrm{ND}\Box}$

• Syntactic Categories

o : Type

• Operations

$\bot$: o
$\supset$: $o \to o \to o$
$\Box$: $o \to o$

• Judgements

$$\begin{array}{lcl} \text{Taut} & : & o \to \text{Type} \\ \text{Valid} & : & o \to \text{Type} \end{array}$$

• Axioms and Rules

$$\begin{array}{lcll} \text{C} & : & \Pi_{\phi:o} & \text{Taut}(\phi) \to \text{Valid}(\phi) \\ \text{R} & : & \prod_{\phi_1,\phi_2:o} & (\text{Taut}(\phi_1) \to \text{Valid}(\phi_2)) \to (\text{Valid}(\phi_1) \to \text{Valid}(\phi_2)) \\ \supset\text{I}_V & : & \Pi_{\phi_1,\phi_2:o} & (\text{Valid}(\Box\phi_1) \to \text{Valid}(\phi_2)) \to \text{Valid}(\Box\phi_1 \supset \phi_2) \\ \bot\ \text{E} & : & \prod_{\phi:o} & \text{Taut}(\bot) \to \text{Taut}(\phi) \\ \neg\neg\text{E} & : & \prod_{\phi:o} & \text{Taut}((\phi \supset \bot) \supset \bot) \to \text{Taut}(\phi) \\ \supset\text{I}_T & : & \Pi_{\phi_1,\phi_2:o} & (\text{Taut}(\phi_1) \to \text{Taut}(\phi_2)) \to \text{Taut}(\phi_1 \supset \phi_2) \\ \supset\text{E}_T & : & \Pi_{\phi_1,\phi_2:o} & \text{Taut}(\phi_1 \supset \phi_2) \to \text{Taut}(\phi_1) \to \text{Taut}(\phi_2) \\ \Box\text{I} & : & \Pi_{\phi:o} & \text{Valid}(\phi) \to \text{Valid}(\Box\phi) \\ \Box\text{E} & : & \Pi_{\phi:o} & \text{Valid}(\Box\phi) \to \text{Valid}(\phi) \end{array}$$

• Example of a Proof

$$\begin{array}{lcll} \supset\text{E}_V & : & \Pi_{\phi_1,\phi_2:o} & \text{Valid}(\phi_1 \supset \phi_2) \to \text{Valid}(\phi_1) \to \text{Valid}(\phi_2) \\ \Delta & : & \prod_{\phi_0:o}\prod_{\phi_1:o} & \text{Valid}(\Box(\phi_0 \supset \phi_1) \supset (\Box\phi_0 \supset \Box\phi_1)) \end{array}$$

It is straightforward but tedious exercise to construct in the above signature the term $\supset \text{E}_V$ and we omit mentioning it explicitly. In the second example we use this term to internalize the natural deduction proof of axiom A_5 of the corresponding Hilbert-type system. Δ is defined to be the following term:

$$\begin{array}{l} \lambda\phi_0 : o.\lambda\phi_1 : o.\ \supset \text{I}_V(\phi_0 \supset \phi_1)(\Box\phi_0 \supset \Box\phi_1) \\ \quad (\lambda x : \text{Valid}(\Box(\phi_0 \supset \phi_1)). \\ \quad\quad (\supset \text{I}_V(\phi_0)(\Box\phi_1)(\lambda y : \text{Valid}(\Box\phi_0). \\ \quad\quad\quad \Box\text{I}(\phi_1)(\supset \text{E}_V(\phi_0)(\phi_1)(\Box\text{E}(\phi_0 \supset \phi_1)(x))(\Box\text{E}(\phi_0)(y)))))) \end{array}$$

Adequacy and Faithfulness Property 4 *Given a proof of a formula ϕ in Prawitz's system, there is an uniform way of constructing a corresponding term t of the LF (in the above signature) of type* $\text{Valid}(\phi)$, *in which the only free variables are those corresponding to the atomic variables of ϕ. Moreover, this construction provides a compositional surjection between assumptions free proofs in Prawitz and this sort of LF terms.*

Some explanations about the role and the meaning of the first three constants of the above signature, are in order:

1. the first two constants, C and R, together enable us to transfer any purely truth-functional proof of a sequent to a corresponding proof in the internalized system. Formally: using them we can uniformly construct, for each n, a term R_n of type:

$$\prod_{\phi_1:o} \cdots \prod_{\phi_n:o} (\text{Taut}(\phi_1) \to \cdots \to \text{Taut}(\phi_n)) \to (\text{Valid}(\phi_1) \to \cdots \to \text{Valid}(\phi_n))$$

The term $\supset E_V$ mentioned above is, for example, immediately obtained from $\supset E_T$ using R_3. Conversely, C and $\supset E_T$ suffices for obtaining all the R_n's (which in turn suffices for the adequacy theorem above, but this is done in a roundabout way and so it is a less faithful method [3]).

The *meaning* of C is obvious. The meaning of R is that if we have a uniform method for extending any proof of ϕ in classical logic to a proof of ψ in Prawitz' system then by the same method we can extend *any* proof of ϕ in that system to a proof of ψ (since applicability of rules in a proof does not depends on what rules were applied above — only on the structure of the formulas involved and the assumptions). R encodes therefore something which is almost an identity function.

2. The role of the remaining constant, $\supset I_V$, is to solve the following problem: applications of the introduction rule for implication can be made in Prawitz's system after applications of the $\Box$-rules. Hence $\supset I_T$ alone is not sufficient for achieving the full power of this rule. $\supset I_V$ directly solves the problem in the basic case in which the application of $\supset$-Introduction is made immediately after an application of $\Box$-introduction. In fact since in this case the discharged formula is necessarily modal, $\supset I_V$ applies. It is possible to prove that this solution of the basic case suffices in general (note that axiom A_4 of the Hilbert system is easily derived using $\Box E$ and $\supset I_V$ and so this rule poses no extra problem![3]). The above proof of A_5 is a good example how this method works.

   Intuitively, $\supset E_V$ corresponds to a deduction theorem about $\vdash_v$ that can be proved either syntactically or by using Kripke models [3].

The explanations concerning $\supset I_V$ should make it obvious how to internalize the second version of Prawitz' system for S4. All one needs to do is to introduce first a new *syntactic* judgement EM (corresponding to the syntactic category of essentially modal formulas) with the obvious characterizing constants. The remaining step is to replace $\supset I_V$ with a constant $\supset I'_V$ of type:

$$\prod_{\phi_1,\phi_2:o} \mathrm{EM}(\phi_1) \to (\mathrm{Valid}(\phi_1) \to \mathrm{Valid}(\phi_2)) \to \mathrm{Valid}(\phi_1 \supset \phi_2)$$

Our final note is concerned with the consequence relations which are defined by Prawitz' system and the above signature. The adequacy theorem above applied to proofs of *theorems* in Prawitz' system. It does not applies to proofs of sequents. As a matter of fact, the "Valid" judgement corresponds exactly to the $\vdash_v$ of the last section: We have that $\phi_1, \ldots, \phi_n \vdash_v \psi$ iff there exist (in the appropriate context) a term t of type $\mathrm{Valid}(\phi_1) \to \cdots \to \mathrm{Valid}(\phi_n) \to \mathrm{Valid}(\psi)$. The consequence relation which is directly defined by Prawitz' system is, on the other hand, $\vdash_t$. In order to fully and faithfully internalize also partial proofs in Prawitz we need to introduce a third judgement: True. The reader may find more details in [3].

[3]It is possible, in fact, to use Taut rather then Valid for internalizing $\Box E$, leaving Valid to handle just the single impure rule of Prawitz' system.

5 Theories of Functions

We now discuss the main issues which arise in encoding functional calculii, such as λ-calculus, call-by-value-λ-calculus, λ-I-calculus and linear λ-calculus. While all these systems are of interest from the point of view of functional programming, the latter two are interesting also from a purely logical point of view. Systems such as relevance and linear logic have consequence relations with weaker structural rules than those implicit in the LF type theory, at least when the constructor $\rightarrow$ is used to encode the $\vdash$. For example, in the case of relevance logic the implication introduction is sound only for λ_I-abstraction. Therefore if we do not introduce in the LF new primitive abstraction operators, then we essentially have to implement this calculus prior to encoding the logic.

5.1 The Classical Lambda Calculus

5.1.1 The System

We begin by discussing the case of the classical λ calculus, we assume the reader familiar with the standard definitions and notations in [7]. To this end we define a basic LF type, o, encoding the set of λ-terms, which we will denote by Λ, together with a judgement, $M = N$, intended to encode the assertion that the term M is $\alpha - \beta$ equal to the term N, this in [7] is denoted by $\vdash_\lambda$. In order to encode the β-reduction rule it is convenient to encode the λ-constructor as $\underline{\lambda} : (o \rightarrow o) \rightarrow o$. In doing so we take care of, at the level of the metalanguage, the operation of capture avoiding substitution which is normally used in formulating the β-rule. Finally we introduce the constant $\mathrm{App} : o \rightarrow o \rightarrow o$ encoding application. Encoding the β and congruence rules is now routine. We also encode the ξ rule which is classically formulated as

$$\frac{M = N}{\lambda x.M = \lambda x.N}$$

in the following.

5.1.2 The Signature Σ_Λ

• Syntactic Category

o : Type

• Operations

$\underline{\lambda}$: $(o \rightarrow o) \rightarrow o$
App : $o \rightarrow o \rightarrow o$

• Judgements

$=$: $o \rightarrow o \rightarrow$ Type

• Axioms and Rules

$$\begin{array}{lcll}
\mathrm{E}_0 & : & \Pi_{x:o} & x = x \\
\mathrm{E}_1 & : & \Pi_{x,y:o} & x = y \to y = x \\
\mathrm{E}_2 & : & \Pi_{x,y,z:o} & x = y \to y = z \to x = z \\
\mathrm{E}_3 & : & \Pi_{x,y,x',y':o} & x = y \to x' = y' \to \mathrm{App}(x, x') = \mathrm{App}(y, y') \\
\beta & : & \Pi_{\substack{x:o \to o \\ y:o}} & \mathrm{App}(\underline{\lambda}(x), y) = xy \\
\xi & : & \Pi_{x,y:o \to o} & (\Pi_{z:o} xz = yz) \to \underline{\lambda}(x) = \underline{\lambda}(y)
\end{array}$$

Notice that there is no counterpart to α-conversion in the above signature. Also note that here, and elsewhere, we shall abbreviate $f(x_1)(x_2)\ldots(x_n)$ by $f(x_1, x_2, \ldots, x_n)$ as is standard. The fact that we have encoded the classical λ-calculus is the expressed by the following theorem.

Adequacy and Faithfulness Property 5 *The following hold:*

1. $x_1 : o, \ldots, x_n : o \vdash_{\Sigma_\Lambda} M : o$ iff $\Phi_\Gamma(M) \in \Lambda$
2. *there is a term* P *such that* $\eta_1 : M_1 = N_1, \ldots, \eta_t : M_t = N_t \vdash_{\Sigma_\Lambda} P : M = N$ *if and only if* $\Phi(M_1) = \Phi(N_1), \ldots, \Phi(M_t) = \Phi(N_t) \vdash_\lambda \Phi(M) = \Phi(N)$

where $M \in \Xi_\Gamma(o)$, $\Xi_\Gamma(o)$ *is the set of normal forms of type* o *in the context* Γ,

$$\Gamma = x_1 : o, \ldots, x_n : o$$

and

$$\Phi_\Gamma : \Xi_\Gamma(o) \longrightarrow \Lambda[x_1, \ldots, x_n]$$

is a bijective function defined as follows

$$\Phi_\Gamma(M) = \begin{cases} x & \text{if } M = x \\ \Phi_\Gamma(M')\Phi_\Gamma(N) & \text{if } M = \mathrm{App}(M', N) \\ \lambda x.\Phi_{\Gamma, x:o}(M'[x]) & \text{if } M = \underline{\lambda}(\lambda x.M'[x]) \end{cases}$$

Notice that the adequacy theorem above has been stated only for closed terms. What can we say if we consider open terms in assumptions? Some remarks are in order here. For the sake of simplicity we do not formalize the relationship between terms in LF, like the term P above, and proofs in the object calculus. One could also in this case, however, extend Φ to a compositional surjection over proof terms. The consequence relation $\vdash_\lambda$ as used in [7] is what is traditionally called a *validity consequence relation*. Assumptions involving open terms are assumed to be true for all instances of the free variables occurring in them. For example, in the classical λ-calculus one can show that

$$x(\Delta\Delta) = x(\Delta\Delta\Delta) \vdash_\lambda \lambda x.x(\Delta\Delta) = \lambda x.x(\Delta\Delta\Delta)$$

where Δ is the term $\lambda z.zz$. But this is not what we have implicitly if we consider the inhabitability of types like

$$M_1 = N_1 \to \ldots \to M_t = N_t \to M = N.$$

What we implicitly encode is in fact a so called *truth consequence relation.* We cannot take assumptions involving open terms to hold for all instances. An explicit LF quantifier is necessary to enforce this, as is done in the encoding of the ξ-rule or the example below. The version of the ξ-rule that we encode has implicitly buried in, the side condition that the variable which we abstract on, does not occur free in any assumption. There is no such side condition in the ξ-rule as presented in [7]. If we were to encode faithfully $\vdash_\lambda$ problems similar to those encountered in Hoare's logic (see section 5.6) would arise. We could tackle this problem differently by giving a more involved statement of the adequacy theorem where assumptions involving free variables are encoded as judgements suitably prefixed with a sufficient number of Π abstractions to make them closed. Interestingly enough Barendregt uses $\vdash_\lambda$ essentially with no assumptions or with closed assumptions, and the validity and the truth versions of $\vdash_\lambda$ coincide for assertions involving closed terms. In view of the discussions carried out in the previous sections we make a final remark which can illuminate on the theory of lambda algebras. The truth consequence relation that we define implicitly, can be viewed also as the consequence relation obtained taking the ξ-rule only as a rule of proof. This in turn amounts to defining the consequence relation corresponding to lambda algebras. In this case truth and validity coincide since no rules involving open assumptions are used for presenting the theory of lambda algebras.

- Example of a Proof

$$\begin{array}{lll}\Delta & : & (\Pi_{M:o} \quad (\underline{\lambda}(\lambda x : o.\mathrm{App}(M,x)) = M) \rightarrow \\ & & \qquad (\Pi_{M,N:o}((\Pi_{x:o}\mathrm{App}(M,x) = \mathrm{App}(N,x)) \rightarrow M = N)))\end{array}$$

The example describes a proof in the LF that assuming the rule η (i.e. that $\lambda x.Mx = M$ provided $x \notin M$) the rule ext

$$\frac{Mx = Nx}{M = N}$$

can be derived, the side condition on the rule is that x should not occur free in any assumption. The term Δ is the following LF term.

$$\begin{array}{l}\lambda y : \prod_{M:o} . \ (\underline{\lambda}(\lambda x : o.\mathrm{App}(M,x)) = M).\lambda M : o.\lambda N : o. \\ \quad \lambda z : \prod_{x:o} . \ \mathrm{App}(M,x) = \mathrm{App}(N,x).\mathrm{E}_2(M)(\underline{\lambda}(\lambda x : o.\mathrm{App}(M,x)))(N) \\ \quad\quad (\mathrm{E}_1(\underline{\lambda}(\lambda x : o.\mathrm{App}(M,x)))(M)(y(M)))(\mathrm{E}_2(\underline{\lambda}(\lambda x : o.\mathrm{App}(M,x))) \\ \quad\quad\quad (\underline{\lambda}(\lambda x : o.\mathrm{App}(N,x)))(N)(\xi(\lambda x : o.\mathrm{App}(M,x)) \\ \quad\quad\quad\quad (\lambda x : o.\mathrm{App}(N,x))(z))(y(N)))\end{array}$$

5.2 Call-By-Value Lambda Calculus

5.2.1 The System

The call-by-value λ-calculus, usually referred to as the λ_v-calculus, was first introduced and studied in Plotkin [43]. In the usual formulation its syntax is identical to that of the traditional λ-calculus. The crucial difference manifests itself in the proof system. In particular in the formulation of the β-reduction rule.

- $\beta_v \quad (\lambda x \ . \ M)N = M[x := N]$
  - provided that N is a value, i.e. either a variable or an abstraction.

The immediate difficulty in encoding this formal system in the LF is how to express the syntactic notion of being a variable. This is compounded by the desire within the LF to have the variables of the LF stand also for schematic variables ranging over λ_v-terms. The solution to this problem, that is presented here, was inspired by the denotational semantics of the calculus [20]. Let D be a domain such that

$$D \rhd \underset{i}{\overset{j}{\rightleftarrows}} [D \to_\perp D],$$

where $i \circ j \not\in [D \to_\perp D]$ while $j \circ i = Id_{[D \to_\perp D]}$, The set $D \to_\perp D$ consists of all strict continuous functions from D to D. Such a D can easily be turned into a model of the λ_v-calculus provided environments are restricted to range only over $D - \{\perp\}$, the subset of intended values. The LF encoding of the λ_v-calculus uses two syntactic categories, v for values (corresponding to objects whose denotation is in $D - \{\perp\}$) and o for expressions (corresponding to objects whose denotation is in the whole of D). The only bindable type (i.e. a type with a binding operator defined on it) is v. The map $! : v \to o$ can be interpreted as the injection

$$! : D - \{\perp\} \to D.$$

This illustrates the general technique for handling subcategories in LF.

5.2.2 The Signature Σ_{Λ_v}

- Syntactic Categories

o : Type
v : Type

- Operations

$!$: $v \to o$
$\underline{\lambda}_v$: $(v \to o) \to v$
App : $o \to o \to o$

- Judgement

$=$: $o \to o \to$ Type

- Axioms and Rules

E_0	:	$\Pi_{x:o}$	$x = x$
E_1	:	$\Pi_{x,y:o}$	$x = y \to y = x$
E_2	:	$\Pi_{x,y,z:o}$	$x = y \to y = z \to x = z$
E_3	:	$\Pi_{x,y,x',y':o}$	$x = y \to x' = y' \to \mathrm{App}(x, x') = \mathrm{App}(y, y')$
β_v	:	$\Pi_{\substack{x:v \to o \\ y:v}}$	$\mathrm{App}(\underline{\lambda}_v(x)!, y!) = xy$
ξ_v	:	$\Pi_{x,y:v \to o}$	$(\Pi_{z:v} xz = yz) \to \underline{\lambda}_v(x)! = \underline{\lambda}_v(y)!$
η_v	:	$\Pi_{x:v}$	$\underline{\lambda}_v(\lambda y : v.\mathrm{App}(x!, y!) = x!$

• Example of a Proof

$$\lambda x : v.\beta_v(!)(x) \quad : \quad \Pi_{x:v}\mathrm{App}(\underline{\lambda}_v(!)!, x!) = x!$$

The example of a proof included above demonstrates that

$$\underline{\lambda}_v(!)!$$

behaves like the identity (with respect to App) on the image of v, via !, in o.

It is worth noting that in this setting the correct version of the η-rule suggests itself more naturally than in the original presentation. The symbol $\vdash$ used in the following theorem denotes the *truth consequence relation* determined by provability in the call-by-value lambda calculus

Adequacy and Faithfulness Property 6 *The following hold:*

1. $x_1 : v, \ldots, x_n : v \vdash_{\Sigma_{\Lambda_v}} M : o$ iff $\Phi_\Gamma(M) \in \Lambda_v$
2. *there is a term* P *such that* $x_1 : v, \ldots, x_n : v, \eta_1 : M_1 = N_1, \ldots, \eta_m : M_m = N_m \vdash_{\Sigma_{\Lambda_v}} P : M = N$

   iff

   $$\Phi_\Gamma(M_1) = \Phi_\Gamma(N_1), \ldots, \Phi_\Gamma(M_m) = \Phi_\Gamma(N_m) \vdash \Phi_\Gamma(M) = \Phi_\Gamma(N)$$

where $M \in \Xi_\Gamma(o)$, $\Xi_\Gamma(o)$ *is the set of normal forms of type* o *in the context* Γ,

$$\Gamma = x_1 : v, \ldots, x_n : v$$

and

$$\Phi_\Gamma : \Xi_\Gamma(o) \longrightarrow \Lambda_v[x_1, \ldots, x_n]$$

is a bijective function defined as follows

$$\Phi_\Gamma(M) = \begin{cases} x & \text{if } M = x! \\ \Phi_\Gamma(M')\Phi_\Gamma(N) & \text{if } M = \mathrm{App}(M', N) \\ \lambda x.\Phi_{\Gamma,x:v}(M'[x]) & \text{if } M = \underline{\lambda}_v(\lambda x.M'[x])! \end{cases}$$

As before the function Φ_Γ could (but won't be) be extended to proof terms.

5.3 The Lambda I Calculus

5.3.1 The System

In encoding the λ_I-calculus we must deal with another syntactic idiosyncrasy, namely the restriction on λ-abstraction. The set, Λ_I, of terms of the λ_I-calculus is defined as in the classical calculus, except for the abstraction clause. In this case the rule states:

$$M \in \Lambda_I \quad \wedge \quad x \in M \quad \Rightarrow \quad \lambda x \ . \ M \in \Lambda_I$$

The immediate difficulty in formalizing this system in the LF is how to enforce the binding constructor λ_I to be defined only on *relevant* schemes. For encoding this

system we will use an approach similar to the one taken in the case of the λ_v-calculus. In other words we will give a solution inspired by the denotational semantics of the calculus. A typical model, in this case, can be constructed from a domain D such that

$$D \triangleright \underset{i}{\overset{j}{\rightleftarrows}} [D \to_\perp D].$$

with $i \circ j \in [D \to_\perp D]$ and $j \circ i = Id_{[D \to_\perp D]}$, see [20].

A new constant $\perp$: o is introduced together with rules governing its behavior. The predicate *being a relevant function* is encoded as:

$\mathrm{Rel}_1 \equiv \lambda x : o \to o.x(\perp) = \perp$.

The λ_I-constructor is $\underline{\lambda}_I : \prod_{x:o \to o} \mathrm{Rel}_1(x) \to o$. Informally we can say that a schema is relevant only if it is strict. It is interesting to notice the role of judgements in the definition of the syntax. In particular proofs of $x(\perp) = \perp$ appear in the construction of terms. Of course these proofs are irrelevant in the sense that their use is only in expressing the fact that we are indeed manipulating $\lambda - I$-terms. This fact is reflected in the adequacy theorem in that the usual correspondence function Φ from LF terms to ordinary terms is only surjective and not injective as in the previous examples. This phenomenon appears also in our treatment of the linear lambda calculus in the following section. The referee pointed out that De Bruijn had already noticed this phenomenon, which he termed *proof-irrelevance*, in a different context.

5.3.2 The Signature Σ_{Λ_I}

• Syntactic Categories

o : Type

• Operations

$\perp$: o
$\underline{\lambda}_I$: $\Pi_{x:o \to o}$ $\quad x(\perp) = \perp \to o$
App : $o \to o \to o$

• Judgement

$=$: $o \to o \to$ Type

• Axioms and Rules

E_0 : $\Pi_{x:o}$ $\quad x = x$
E_1 : $\Pi_{x,y:o}$ $\quad x = y \to y = x$
E_2 : $\Pi_{x,y,z:o}$ $\quad x = y \to y = z \to x = z$
E_3 : $\Pi_{x,y,x',y':o}$ $\quad x = y \to x' = y' \to \mathrm{App}(x,x') = \mathrm{App}(y,y')$
$\perp_r$: $\Pi_{x:o}$ $\quad \mathrm{App}(x,\perp) = \perp$
$\perp_l$: $\Pi_{x:o}$ $\quad \mathrm{App}(\perp,x) = \perp$
$\perp_{\underline{\lambda}}$: $\quad \perp = \underline{\lambda}_I(\lambda x : o\ .\ \perp, \mathrm{E}_0(\perp))$
β_I : $\Pi_{\substack{x:o \to o \\ y:o}}$ $\quad \Pi_{t:x(\perp)=\perp} \mathrm{App}(\underline{\lambda}_I(x,t),y) = xy$
ξ_I : $\Pi_{x,y:o \to o}$ $\quad \Pi_{\substack{t_1:x(\perp)=\perp \\ t_2:y(\perp)=\perp}} (\Pi_{z:o} x(z) = y(z)) \to \underline{\lambda}_I(x,t_1) = \underline{\lambda}_I(y,t_2)$

• Example of a Proof

$$\Delta \ : \ \Pi_{x:o\to o}\Pi_{t_1,t_2:x(\bot)=\bot} \quad \underline{\lambda}_I(x,t_1) = \underline{\lambda}_I(x,t_2)$$

Where the term Δ is the following:

$$\lambda x : o \to o \ . \ \lambda t_1 : x(\bot) = \bot \ . \ \lambda t_2 : x(\bot) = \bot \ . \xi_i(x)(x)(t_1)(t_2)(\lambda z : o \ . \ \mathrm{E}_0(x(z))).$$

The symbol $\vdash$ used in the following theorem denotes the *truth consequence relation* determined by provability in Λ_I-calculus

Adequacy and Faithfulness Property 7 *The following hold:*

1. $x_1 : o, \ldots, x_n : o \vdash_{\Sigma_{\Lambda_I}} M : o$ iff $\Phi_\Gamma(M) \in \Lambda_I$
2. *there is a term* P *such that* $x_1 : o, \ldots, x_n : o, \eta_1 : M_1 = N_1, \ldots, \eta_m : M_m = N_m \vdash_{\Sigma_{\Lambda_I}} P : M = N$

   iff

   $$\Phi_\Gamma(M_1) = \Phi_\Gamma(N_1), \ldots, \Phi_\Gamma(M_m) = \Phi_\Gamma(N_m) \vdash \Phi_\Gamma(M) = \Phi_\Gamma(N)$$

where $M \in \Xi_\Gamma(o)$, $\Xi_\Gamma(o)$ *is the set of normal forms of type* o *in the context* Γ, *in which* $\bot$ *occurs only within the second argument to the* Λ_I*-operator.*

$$\Gamma = x_1 : o, \ldots, x_n : o$$

and

$$\Phi_\Gamma : \Xi_\Gamma(o) \longrightarrow \Lambda_I[x_1, \ldots, x_n]$$

is a surjective function defined as follows

$$\Phi_\Gamma(M) = \begin{cases} x & \text{if } M = x \\ \Phi_\Gamma(M')\Phi_\Gamma(N) & \text{if } M = \mathrm{App}(M', N) \\ \lambda x.\Phi_{\Gamma,x:o}(M'[x]) & \text{if } M = \underline{\lambda}_I(\lambda x : o.M'[x], t) \end{cases}$$

For the sake of simplicity we do not formalize the relationship between terms in LF, like the term P above, and proofs in the object calculus. One could also in this case, however, extend Φ to a compositional surjection over proof terms. A second approach can be given, it is a generalization of the previous one. No $\bot$ constant is needed. The idea is to axiomatize the predicate $x \in M$ by introducing a new judgement $\in$: $o \to o \to$ Type and appropriate rules. A representative example of these rules is provided by the constant $\in_{\underline{\lambda}}$.

$$\in_{\underline{\lambda}}: \prod_{\substack{x:o \\ M:o\to o \\ t:\mathrm{Rel}_2(M)}} (\Pi_{y:o} x \in My) \to x \in \underline{\lambda}_I(M,t)$$

The predicate *being a relevant function* is encoded as:

$$\mathrm{Rel}_2 \equiv \lambda x : o \to o. \prod_{z,y:o} z \in y \to z \in xy$$

The λ_I constructor is encoded as follows: $\underline{\lambda}_I : \prod_{x:o\to o}.\mathrm{Rel}_2(x) \to o$.

5.4 Linear Lambda Calculus

5.4.1 The System

The set Λ_L of terms of the linear lambda calculus is defined as follows:

1. $x \in \Lambda_L$
2. If $M \in \Lambda_L$ and $x \in FV(M)$ then $\lambda x.M \in \Lambda_L$
3. If $M, N \in \Lambda_L$ and $FV(M) \cap FV(N) = \emptyset$ then $MN \in \Lambda_L$

In order to encode this system into the LF one makes use of the idea of actually encoding the notion of linearity of a function. A function $f : X \to Y$, where X and Y are upper semilattices with a least element, is linear iff

1. $f(\bot) = \bot$
2. $f(sup(a, b)) = sup(f(a), f(b))$

We will in fact end up encoding a related calculus Λ_L^* whose syntax is defined as follows:

1. $x \in \Lambda_L^*$
2. If $M \in \Lambda_L^*$ and x occurs free in M exactly once then $\lambda x.M \in \Lambda_L^*$
3. If $M, N \in \Lambda_L^*$ then $MN \in \Lambda_L^*$

This example is based on an idea of Gordon Plotkin and we thank him for allowing us to include it here.

5.4.2 The Signature Σ_{Λ_L}

In the following presentation of the signature we make, for typographical reasons, the following definition. We let

$$L = \lambda x : o \to o.\ \prod_{z,w:o} .\ x(z \vee w) = x(z) \vee x(w).$$

• Syntactic Categories

o : Type

• Operations

$\bot$	:		o
$\vee$	:		$o \to o \to o$
$\underline{\lambda}_L$	:	$\Pi_{x:o\to o}$	$x(\bot) = \bot \to L(x) \to o$
App	:		$o \to o \to o$

• Judgement

$=$: $o \to o \to$ Type

• Axioms and Rules

$$
\begin{array}{lcll}
\mathrm{E}_0 & : & \Pi_{x:o} & x = x \\
\mathrm{E}_1 & : & \Pi_{x,y:o} & x = y \to y = x \\
\mathrm{E}_2 & : & \Pi_{x,y,z:o} & x = y \to y = z \to x = z \\
\mathrm{E}_3 & : & \Pi_{x,y,x',y':o} & x = y \to x' = y' \to \mathrm{App}(x,x') = \mathrm{App}(y,y') \\
\bot_r & : & \Pi_{x:o} & \mathrm{App}(x,\bot) = \bot \\
\bot_l & : & \Pi_{x:o} & \mathrm{App}(\bot,x) = \bot \\
\bot_{\underline{\lambda}} & : & & \bot = \underline{\lambda}_L(\lambda x : o.\ \bot, \mathrm{E}_0(\bot), \lambda y : o.\lambda z : o.\ \vee_{\mathrm{id}}(\bot)) \\
\vee_{\mathrm{id}} & : & \Pi_{x:o} & x = x \vee x \\
\vee_{\bot} & : & \Pi_{x:o} & x \ \vee \bot = x \\
\vee_{\mathrm{sym}} & : & \Pi_{x,y:o} & x \vee y = y \vee x \\
\vee_{\mathrm{assoc}} & : & \Pi_{x,y,z:o} & (x \vee y) \vee z = x \vee (y \vee z) \\
\vee_{=} & : & \Pi_{x,y,z:o} & x = y \to (x \vee z) = (y \vee z) \\
\vee_{l} & : & \Pi_{x,y,z:o} & \mathrm{App}(x, y \vee z) = \mathrm{App}(x,y) \vee \mathrm{App}(x,z) \\
\vee_{r} & : & \Pi_{x,y,z:o} & \mathrm{App}(x \vee y, z) = \mathrm{App}(x,z) \vee \mathrm{App}(y,z) \\
\vee_{\underline{\lambda}} & : & \Pi \begin{smallmatrix} x,y:o\to o \\ t_1:x(\bot)=\bot \\ t_2:y(\bot)=\bot \\ t_3:L(x) \\ t_4:L(y) \\ t_5:y(\bot)\vee z(\bot)=\bot \\ t_6:L(\lambda z:o.x(z)\vee y(z)) \end{smallmatrix} & \underline{\lambda}_L(x,t_1,t_3) \vee \underline{\lambda}_L(y,t_2,t_4) = \underline{\lambda}_L(\lambda z : o.x(z) \vee y(z), t_5, t_6) \\
\beta_L & : & \Pi \begin{smallmatrix} x:o\to o \\ y:o \\ t_1:x(\bot)=\bot \\ t_2:L(x) \end{smallmatrix} & \mathrm{App}(\underline{\lambda}_L(x,t_1,t_2), y) = xy \\
\xi_L & : & \Pi \begin{smallmatrix} x,y:o\to o \\ t_1:x(\bot)=\bot \\ t_2:y(\bot)=\bot \\ t_3:L(x) \\ t_4:L(y) \end{smallmatrix} & (\Pi_{z:o} x(z) = y(z)) \to \underline{\lambda}_L(x,t_1,t_3) = \underline{\lambda}_L(y,t_2,t_4)
\end{array}
$$

• Example of a Proof

$$\Delta \ : \ \Pi \begin{smallmatrix} x:o\to o \\ t_1,t_2:x(\bot)=\bot \\ t_3,t_4:L(x) \end{smallmatrix} \quad \underline{\lambda}_L(x,t_1,t_3) = \underline{\lambda}_L(x,t_2,t4)$$

Where the term Δ is the following:

$$\lambda x : o \to o.\lambda t_1 : x(\bot) = \bot\ .\lambda t_2 : x(\bot) = \bot\ .\lambda t_3 : L(x).\lambda t_4 : L(x)$$
$$\xi_L(x)(x)(t_1)(t_2)(t_3)(t_4)(\lambda z : o.\mathrm{E}_0(x(z))).$$

The symbol $\vdash$ used in the following theorem denotes the *truth consequence relation* determined by provability in the linear λ-calculus.

Adequacy and Faithfulness Property 8 *The following hold:*

1. $\vdash_{\Lambda_L} M : o \quad$ iff $\quad \Phi_\emptyset(M) \in \Lambda_L^\circ$
2. *there is a term* P *such that* $\eta_1 : M_1 = N_1, \ldots, \eta_m : M_m = N_m \vdash_{\Lambda_L} P : M = N$

iff

$$\Phi_\emptyset(M_1) = \Phi_\emptyset(N_1), \ldots, \Phi_\emptyset(M_m) = \Phi_\emptyset(N_m) \vdash \Phi_\emptyset(M) = \Phi_\emptyset(N)$$

where $M \in \Xi_\Gamma(o)$, $\Xi_\Gamma(o)$ *is the set of normal forms of type* o *in the context* Γ, *in which* $\perp$ *occurs only within the second argument and* $\vee$ *occurs only in the third argument to the* $\underline{\lambda}_L$*-operator.*

$$\Gamma = x_1 : o, \ldots, x_n : o$$

and

$$\Phi_\Gamma : \Xi_\Gamma(o) \longrightarrow \Lambda^*_L[x_1, \ldots, x_n]$$

is a surjective function defined as follows

$$\Phi_\Gamma(M) = \begin{cases} x & \text{if } M = x \\ \Phi_\Gamma(M')\Phi_\Gamma(N) & \text{if } M = \mathrm{App}(M', N) \\ \lambda x.\Phi_{\Gamma,x:o}(M'[x]) & \text{if } M = \underline{\lambda}_L(\lambda x : o.M'[x], t, s) \end{cases}$$

For the sake of simplicity we do not formalize the relationship between terms in LF, like the term P above, and proofs in the object calculus. One could also in this case, however, extend Φ to a compositional surjection over proof terms. The above signature can be taken as a basis for the LF encoding of the external consequence relation of the minimal fragment of linear logic [2] [21].

6 Program Logics

Program logics such as Hoare's logic and dynamic logic exhibit an unusual overloading of variables. In both these logics variables play two roles, behaving in some instances as *logical variables* ranging over the data domain, and in other instances as assignable *identifiers* or *locations*. A typical example, from dynamic logic, is

$\forall x > 0[\mathrm{while}(x > 0, x := x - 1)]x = 0.$

It not only illustrates the dual nature of variables but also the difficulties in defining the notion of a free and bound variable. The occurrence of x in the while test is, in a sense, bound by both the quantifier and the assignment. Nevertheless even in the somewhat simpler case of Hoare's logic for a simple assignment language (whose only control primitives are assignment and sequencing), problems arise. In this section we outline these problems and offer two different solutions.

6.1 Hoare's Logic

In this example we give a brief exposition of Hoare's Logic in the language of integers $\{+, 0, 1\}$.

6.1.1 The System

Let L denote the first order language of the integers with equality, the meta-variables x, y, z denote or range over the variables of L, the meta-variables s, t denote or range over the terms or *expressions* of L, the meta-variable e is used to denote a quantifier-free formula or *boolean expression* of L, and, finally, p, q, r denote or range over the formulas or *assertions* of L. Let W denote the least class of while programs satisfying

1. for every variable x and expression t, $x := t \in W$; and

2. if $S_1, S_2 \in W$ then $S_1; S_2 \in W$, and for every boolean expression e of L, we have that $\texttt{if}(e, S_1, S_2) \in W$ and $\texttt{while}(e, S_1) \in W$.

The basic formulas of Hoare's logic are objects of the form $\{p\}S\{q\}$ where p, q are assertions and S is a while program. The intuitive meaning of an *asserted program,*

$$\{p\}S\{q\},$$

is as follows: whenever p holds before execution of S and S terminates, then q holds after execution of S. Hoare's logic is a system of formal reasoning about these asserted programs. Its axioms and proof rules are the following.

Axiom 1: Assignment Axiom

$$\{p[t/x]\}x := t\{p\}.$$

Rule 2: Composition Rule

$$\frac{\{p\}S_1\{r\}, \quad \{r\}S_2\{q\}}{\{p\}S_1; S_2\{q\}}$$

Rule 3: If Rule

$$\frac{\{p \wedge e\}S_1\{q\}, \quad \{p \wedge \neg e\}S_2\{q\}}{\{p\}\texttt{if}(e, S_1, S_2)\{q\}}$$

Rule 4: While Rule

$$\frac{\{p \wedge e\}S\{p\}}{\{p\}\texttt{while}(e, S)\{p \wedge \neg e\}}$$

The final rule involves some notion of a consequence relation, [1], for the assertion language.

Rule 5: Consequence Rule

$$\frac{p \Rightarrow p_1, \quad \{p_1\}S\{q_1\}, \quad q_1 \Rightarrow q}{\{p\}S\{q\}}$$

Here $p \Rightarrow p_1$ and $q_1 \Rightarrow q$ are assumed to follow from the background first order theory using the proof system for the assertion language. As usual, $p[t/x]$ stands for the result of substituting t for the free occurrences of x in p.

There are, at least, three complications one must deal with in encoding this logic. Firstly we must distinguish between the variables of the first order logic and the variables of the programming language. We cannot model := as an object of type $i \to i \to w$ since this would allow expressions like $0 := 1$. A new type l, corresponding to locations, is introduced together with a function $! : l \to o$, called bang, which takes a location to its contents. Secondly, since the LF does not have subtypes we must

distinguish between the boolean expressions and the first order formulas. We do this by introducing a new judgment (a syntactic one), $\mathtt{QF}$, on o. Thus the if and while constructs not only take an element of o as an argument but also a proof that they are quantifier free. Thirdly, note that $:=$ is a binding operator. In the assignment axiom free occurrences of x in p are bound by the assignment operator $x := t$. This is not true of those occurrences in t either in $p[t/x]$ or in the assignment. One could even claim that it is an example of a binding operator which does not α-convert. α-conversion does not appear to be in the spirit of Hoare's logic, since one wants to reason about the identifier x not some α-conversion of it. This has the consequence that simply modelling the assignment axiom by

$$\mathrm{Ass} : \Pi_{\substack{x:l,t:i \\ p:i\to o}} \vdash_h \{p(t)\}x := t\{p(x!)\}$$

would be incorrect, e.g.

$$\mathrm{Ass}(y)(1)(\lambda u.\neg(y! = u)) : \quad \vdash_h \{\neg(y! = 1)\}y := 1\{\neg(y! = y!)\}.$$

The problem, intuitively, is that $\{p(t)\}x := t\{p(x!)\}$ can be false because the assignment $x := t$ can alter the meaning of the predicate $\lambda z : i.p(z)$. One solution to this problem is to incorporate syntactic notions explicitly into the theory. We do this by adding three new judgements, $\sharp_l, \sharp_i$ and $\sharp_o$, concerning non-interference along the lines of [48], $\sharp_x$ is of type $l \to (x \to \mathtt{TYPE})$. The intuitive meaning of the judgements can be explained, using infix notation, as follows: the judgement $x\sharp_l y$ is interpreted as meaning that x and y denote distinct identifiers or locations. the judgement $x\sharp_i t$ is interpreted as meaning that no assignment to the location denoted by x effects the value of the term denoted by t. This of course is equivalent to saying that the location or identifier denoted by x does not occur in the term denoted by t. the judgement $x\sharp_o e$ is interpreted as meaning that no assignment to the location denoted by x effects the value or meaning of the formula denoted by e. Again this is equivalent to saying that the location or identifier denoted by x does not occur freely in the formula denoted by e (note that it cannot occur bound). The corrected version of the assignment axiom may be written as follows.

$$\mathrm{Ass} : \Pi_{\substack{x:l,t:i \\ \Phi:i\to o}} x\sharp_o \forall\Phi \to (\vdash_h \{\Phi(t)\}x := t\{\Phi(x!)\})$$

This solution, see [32], takes the notion of a free variable as primitive, another solution is to encode *substituting a term for all free occurrences of a banged location in terms and formulas.* This would involve introducing two new operations (rather than the two judgements $\sharp_i$ and $\sharp_o$) sub_i and sub_o, where sub_x is of type $i \to l \to x \to x$, and $sub_x(t, y, z)$ represents the result of substituting the term t for all free occurrences of $y!$ in z. To axiomatize these operations, in particular the base case, one must still retain the judgement $\sharp_l$, and so in some sense the two solutions are dual. There is little reason, on the face of it, to choose one over the other. We should point out, however, that to correctly formalize more complex versions of Hoare's logic, for example one in which recursive procedure calls were allowed, it would be necessary to incorporate the notion on non-interference anyway. Thus in the long run the first solution seems most suited to Hoare's logic. On the other hand in dynamic logic the substitution approach may be more natural, since there is no clear notion of free and bound variables in that logic.

6.1.2 The Signature Σ_{HL}

• Syntactic Categories

$$
\begin{array}{lcl}
l & : & \mathrm{Type} \\
i & : & \mathrm{Type} \\
o & : & \mathrm{Type} \\
w & : & \mathrm{Type} \\
h & : & \mathrm{Type}
\end{array}
$$

• Operations

$$
\begin{array}{lcl}
! & : & l \to i \\
0 & : & i \\
1 & : & i \\
+ & : & i \to i \to i \\
= & : & i \to i \to o \\
\neg & : & o \to o \\
\supset & : & o \to o \to o \\
\forall & : & (i \to o) \to o \\
\ldots := \ldots & : & l \to i \to w \\
\ldots ; \ldots & : & w \to w \to w \\
\mathrm{if} & : & \prod_{e:o} . \, \mathtt{QF}(e) \to w \to w \to w \\
\mathrm{while} & : & \prod_{e:o} . \, \mathtt{QF}(e) \to w \to w \\
\{\ldots\} \ldots \{\ldots\} & : & o \to w \to o \to h
\end{array}
$$

• Judgements

$$
\begin{array}{lcl}
\vdash_o & : & o \to \mathrm{Type} \\
\vdash_h & : & h \to \mathrm{Type} \\
\neq & : & l \to l \to \mathrm{Type} \\
\sharp_i & : & l \to i \to \mathrm{Type} \\
\sharp_o & : & l \to o \to \mathrm{Type} \\
\mathtt{QF} & : & o \to \mathrm{Type}
\end{array}
$$

• Axioms and Rules

$\sharp_0$	:	$\Pi_{x:l}$	$x\sharp_i 0$
$\sharp_1$	:	$\Pi_{x,y:l}$	$x \neq y \rightarrow x\sharp_i y!$
$\sharp_2$	:	$\Pi_{\substack{x:l \\ t_1,t_2:i}}$	$x\sharp_i t_1 \rightarrow x\sharp_i t_2 \rightarrow x\sharp_i(t_1 + t_2)$
$\sharp_3$	:	$\Pi_{\substack{x:l \\ t_1,t_2:i}}$	$x\sharp_i t_1 \rightarrow x\sharp_i t_2 \rightarrow x\sharp_o(t_1 = t_2)$
$\sharp_4$	:	$\Pi_{\substack{x:l \\ \phi:o}}$	$x\sharp_o\phi \rightarrow x\sharp_o\neg\phi$
$\sharp_5$	:	$\Pi_{\substack{x:l \\ \phi_1,\phi_2:o}}$	$x\sharp_o\phi_1 \rightarrow x\sharp_o\phi_2 \rightarrow x\sharp_o(\phi_1 \supset \phi_2)$
$\sharp_6$	:	$\Pi_{\substack{\Phi:i\rightarrow o \\ x:l}}$	$(\Pi_{y:l}(x \neq y \rightarrow x\sharp_o\Phi(y!))) \rightarrow x\sharp_o\forall\Phi$
$\mathtt{QF}_0$	:	$\Pi_{t_1,t2:i}$	$\mathtt{QF}(t_1 = t_2)$
$\mathtt{QF}_2$	:	$\Pi_{\phi_1:o}$	$\mathtt{QF}(\phi_1) \rightarrow \mathtt{QF}(\neg\phi_1)$
$\mathtt{QF}_3$	:	$\Pi_{\phi_1,\phi_2:o}$	$\mathtt{QF}(\phi_1) \rightarrow \mathtt{QF}(\phi_2) \rightarrow \mathrm{QF}(\phi_1 \supset \phi_2)$
Ass	:	$\Pi_{\substack{x:l \\ t:i \\ \Phi:i\rightarrow o}}$	$x\sharp_o\forall\Phi \rightarrow (\vdash_h \{\Phi(t)\}x := t\{\Phi(x!)\})$
Seq	:	$\Pi_{\substack{\phi_0,\phi_1,\phi_2:o \\ w_1,w_2:w}}$	$\vdash_h \{\phi_0\}w_1\{\phi_1\} \rightarrow$ $\vdash_h \{\phi_1\}w_2\{\phi_2\} \rightarrow$ $\vdash_h \{\phi_0\}w_1; w_2\{\phi_2\}$
If	:	$\Pi_{\substack{\phi,\phi_1,\phi_2:o \\ w_1,w_2:w \\ p:\mathtt{QF}(\phi)}}$	$\vdash_h \{\phi_1 \wedge \phi\}w_1\{\phi_2\} \rightarrow$ $\vdash_h \{\phi_1 \wedge \neg\phi\}w_2\{\phi_2\} \rightarrow$ $\vdash_h \{\phi_1\}\mathrm{if}(\phi, p, w_1, w_2)\{\phi_2\}$
While	:	$\Pi_{\substack{\phi,\phi_1:o \\ w_1:w \\ p:\mathtt{QF}(\phi)}}$	$\vdash_h \{\phi_1 \wedge \phi\}w_1\{\phi_1\} \rightarrow \quad \vdash_h \{\phi_1\}\mathrm{while}(\phi, p, w_1)\{\neg\phi_1\}$
Con	:	$\Pi_{\substack{\phi_1,\phi_1',\phi_2,\phi_2':o \\ w_1:w}}$	$\vdash_o \phi_1 \Rightarrow \phi_1' \rightarrow$ $\vdash_o \phi_2' \Rightarrow \phi_2 \rightarrow$ $\vdash_h \{\phi_1'\}w_1\{\phi_2'\} \rightarrow \quad \vdash_h \{\phi_1\}w_1\{\phi_2\}$

Adequacy and Faithfulness Property 9 *Let Γ_n^m be the following context, for m, n integers:*

$$\Gamma_n^m = \{y_0 : i, \ldots, y_m : i, x_0 : l, \ldots, x_n : l, z_{i,j} : x_i \neq x_j, z_{i,j}^* : x_j \neq x_i\}_{0 \leq i \leq j \leq n}.$$

In the above LF signature and in the context Γ_n^m we have the following facts concerning syntax:

- *l : All well formed long $\beta\eta$–normal forms of type l are LF variables of type l, and hence are among the $x_0, \ldots, x_n$.*
- *There are compositional bijections τ_K, for each syntactic category K, which map long $\beta\eta$–normal forms of type K to:*
  - *well formed terms of the assertion language built up from the set of identifiers $\bar{x}$ and the logical variables $\bar{y}$, in the case when $K = i$.*

- *formulas of the assertion language built up from the set of identifiers $\bar{x}$ (which if they occur must occur free) and the logical variables $\bar{y}$, in the case when $K = o$.*
- *while programs of τ built up from the set of identifiers $\bar{x}$ and the logical variables $\bar{y}$ (which do not occur in the left hand side of any assignment statement), in the case when $K = w$.*
- *asserted programs (i.e. Hoare triples) built up from the set of identifiers $\bar{x}$ and the logical variables $\bar{y}$ (here again no variable from $\bar{y}$ can occur in the left hand side of any assignment statement), in the case of $K = h$.*

- *There is a compositional bijection between proofs of a Hoare triple $\{p\}S\{q\}$ from assumptions $r_0, \ldots, r_s$ (in the assertion language) and assumptions*

$$\{p_0\}S_0\{q_0\}, \ldots, \{p_t\}S_t\{q_t\}$$

(concerning asserted programs) and well formed $\beta\eta$–normal forms of type

$$\vdash_h \{p\}S\{q\}$$

in the above signature and in the context Γ where

$$\Gamma = \Gamma_n^m \cup \{w_j : (\vdash_o r_j), v_i : (\vdash_h \{p_i\}S_i\{q_i\})\}_{0 \leq j \leq s, 0 \leq i \leq t},$$

and Γ_n^m is adequate for the syntax of objects involved.

6.2 Two- Register Machine Hoare's Logic

6.2.1 The System

Another approach is not to reason about Hoare triples directly but rather deal primarily with functions from state to triples. Explicitly we deal with objects obtained from triples by abstracting the program locations. Thus we must restrict our attention to assertions concerning programs built up from a fixed finite number of such locations. In the abridged signature we present below this number is two, the judgement $\vdash$ is therefore of type $(l \to l \to h) \to$ Type.

6.2.2 The Signature Σ_{HL^2}

• Syntactic Categories

l : Type
i : Type
o : Type
w : Type
h : Type

• Operations

$$
\begin{array}{ll}
! & : \; l \to i \\
0 & : \; i \\
+ & : \; i \to i \to i \\
= & : \; i \to i \to o \\
\neg & : \; o \to o \\
\supset & : \; o \to o \to o \\
\forall & : \; (i \to o) \to o \\
\ldots := \ldots & : \; l \to i \to w \\
\ldots ; \ldots & : \; w \to w \to w \\
\{\ldots\}\ldots\{\ldots\} & : \; o \to w \to o \to h
\end{array}
$$

• Judgements

$$\vdash \; : \; (l \to l \to h) \to \text{Type}$$

• Axioms and Rules

$$
\begin{array}{lll}
\text{Ass}_1 & : \; \Pi_{\substack{t:l\to l\to i \\ \Phi:i\to i\to o}} & \vdash \lambda x : l.\lambda y : l.\{\Phi(t(x,y),y!)\}x := t(x,y)\{\Phi(x!,y!)\} \\
\text{Ass}_2 & : \; \Pi_{\substack{t:l\to l\to i \\ \Phi:i\to i\to o}} & \vdash \lambda y : l.\lambda x : l.\{\Phi(t(x,y),y!)\}x := t(x,y)\{\Phi(x!,y!)\} \\
\text{Seq} & : \; \Pi_{\substack{\phi_0,\phi_1,\phi_2:l\to l\to o \\ w_1,w_2:l\to l\to w}} & \\
 & & (\vdash \lambda x : l.\lambda y : l.\{\phi_0(x,y)\}w_1(x,y)\{\phi_1(x,y)\}) \to \\
 & & \quad (\vdash \lambda x : l.\lambda y : l.\{\phi_1(x,y)\}w_2(x,y)\{\phi_2(x,y)\}) \to \\
 & & \quad\quad (\vdash \lambda x : l.\lambda y. : l.\{\phi_0(x,y)\}w_1(x,y); w_2(x,y)\{\phi_2(x,y)\})
\end{array}
$$

We shall not state an adequacy and faithfulness property for the above signature. It follows the usual pattern.

The question "Which solution is best?" is rather a philosophical one, and the reply depends somewhat on the aims of the answerer. We only point out that the syntactic judgements in the first solution are axiomatizable in such a way as to ensure that if they can be proved, then such a proof is unique. In other words the search space for these subsystems is linear, and so extremely suitable for automation, perhaps behind the naive users back.

7 A Machine Implementation of LF

LEGO [29, 44, 52] is a system for mechanically checked formal mathematics. It supports a number of type theories, from a variant of LF to an extension of the Generalized Calculus of Constructions with strong sums [28]. These theories are all extensions of LF (both language and derivable judgements), and have some formal properties in common, including strong normalization of all well-typed terms, a decidable conversion relation, and effective type synthesis (i.e. given Γ and M we can find A such that $\Gamma \vdash M : A$, or fail if none exists). LEGO provides for the interactive construction of valid contexts, and for refinement style proof by resolution, uniformly for all the type theories supported. LEGO uses ideas from LCF [22], Nuprl [11],

A	::=	`Type`	
	|	`x`	variable
	|	`[x:A]B` | `[x|A]B`	λ binding; 'hidden' λ binding
	|	`{x:A}B` | `{x|A}B`	Π binding; 'hidden' Π binding
	|	`A->B`	`{x:A}B` when `x` doesn't occur free in `B`
	|	`A B`	application
	|	`[x=A]B`	local definition; i.e. 'let'
	|	`M:A`	type cast
Γ	::=	$\langle\rangle$	empty context
	|	Γ`[x:A]`	declaration
	|	Γ`[x=M]`	global definition

Table 1: Basic Syntax of LEGO

EFS [23], Isabelle [37, 38], and especially from INRIA implementations of the Calculus of Constructions [14, 15, 13]. LEGO is coded in CAML, a version of ML from INRIA [53].

7.1 LF in LEGO

LEGO implements a variant of LF, LEGO-LF, with some pragmatic features not included in pure LF. LEGO-LF has no separate signature, but instead merges the notions of signature and context by allowing the context to contain declarations `c:K` where `K` is a kind[4]. Since the rules of LEGO-LF have conditions that prevent discharging such declarations, this doesn't essentially change LF.

The basic syntax of terms and contexts is given in Table 1. LEGO has the usual syntax conventions: application associates to the left, `->` to the right, the scope of binders goes as far to the right as possible, and parentheses are used to be explicit about grouping.

LEGO-LF allows *definitions*, both globally (in the context) and locally (i.e. 'let' in terms). Any well-typed term, `M`, may be assigned a name, `x`, by the definition `[x=M]`. All instances of `x` in the scope of this definition behave as if they were instances of `M`. Definitions are explained as adding δ-reductions to the reduction relation of LF. This is a conservative extension of LF: any judgement derivable using the definition facility, becomes derivable without using it when all its definitions are expanded. LEGO takes definitions seriously: a name instance is expanded to it's value only when this is requested by the user, or when necessary for unification to succeed in a refinement proof step.

Definitions serve two purposes in LEGO: as notational extension and to capture the notion of 'proved theorem', or more generally, 'derivable rule'. This latter use is justified by the judgements as types principle. If Γ is a LEGO-LF context representing some formal system, `A` is a judgement of that sytem, and $\Gamma \vdash$ `M:A` for some term `M`,

[4]When the type theory is presented in this way it is easy to see that the Pure Calculus of Constructions is obtained by extending LF with rules for abstraction and generalization indexed by kinds [44].

then extending Γ with the definition `[x=M]` is like assuming `[x:A]` in the system, while also expressing the intention that all uses of that assumption can be normalized away (in the metatheory). This can be made even more explicit using the type-casting notation in a definition `[x=M:A]`.

The 'hidden' forms of binders require some explanation. They mark positions subject to *argument synthesis*[5]: these arguments are redundant, and need not appear explicitly in a term because LEGO can synthesize them by unification with other parts of the term. For example, consider the 'polymorphic identity function' I = `[t:Type][x:t]x`. In pure LF, I must be applied to both a type and an object of that type, say `M`. But given `M`, it is always possible to synthesize its type, so a user need not actually supply it. The LEGO syntax `[t|Type][x:t]x` indicates that the variable `t` is to be unified with the type synthesised for the actual argument instantiating `x`, and after this unification the type of `t` must be `Type`. Furthermore, `[t|Type][x:t]x` has type `{t|Type}{x:t}t`. In the general case, this idea requires unification of arbitrary LF terms, hence is undecidable; LEGO uses a simple unification algorithm, and asks the user for more type information if necessary. In practice this feature works very well; for example, the proof term constructed in the example below would be three times larger without this supression of redundant arguments.

While we could say that the assertions of LEGO-LF are of the form $\Gamma \vdash$ `M:A`, the point is that LEGO allows both the context, Γ, and the inhabiting term, `M`, to be constructed and checked incrementally. The equivalence between LEGO-LF (including argument synthesis, definitions, and other features) and LF as described in Section 2 is formally explained in [44].

7.2 An Example: Call-By-Value Lambda Calculus

In this example we use LEGO to mechanically check that the presentation of the call-by-value lambda calculus of Section 5.2 is well formed, and construct a proof in that system.

The LEGO user interacts with the top-level of ML, and LEGO commands are ML functions, but there is a YACC generated parser, so the LEGO terms and commands that the user enters are not limited to ML syntax. It is intended that LEGO be used interactively through an Emacs editor. In the transcript below, `#` is the ML prompt, and user input appears on lines beginning with `#`. This input is converted into ML objects and function calls by the YACC generated parser. LEGO's responses are shown, slightly edited for clarity, on lines not beginning with `#`.

7.2.1 Presenting the Object System

First the type theory LF is selected, and initialized to the empty context:

```
# Init LF ;
LF: Initial State!
```

The syntactic categories of the call-by-value lambda calculus are declared by:

[5]This feature is formally explained in [44]. See also [40]

```
# [ o,v:Type ] ;
declare  o v
```

This means 'extend the current context with bindings `o:Type` and `v:Type` if it is valid to do so in LF'. (Notice that `[o,v:Type]` is an abbreviation for `[o:Type][v:Type]`.) In this case LEGO checks that `Type` is a kind, so the context extension is legal. Now we similarly declare the rest of the object language syntax:

```
# [ shriek: v->o ]
# [ lam: (v->o)->v ]
# [ app: o->o->o ]
# [ eq: o->o->Type ] ;
declare  shriek lam app eq
```

On each declaration LEGO checks that the context extension is valid. The congruence rules are entered in the same way:

```
# [ E0: {x:o}(eq x x) ]
# [ E1: {x,y|o}(eq x y)->(eq y x) ]
# [ E2: {x,y,z|o}(eq x y)->(eq y z)->(eq x z) ]
# [ E3: {x,y,x',y'|o}(eq x y)->(eq x' y')->
#                     (eq (app x x') (app y y')) ] ;
declare  E0 E1 E2 E3
```

Notice that these rules have some 'hidden' bindings, annotations to perform argument synthesis.

Next, extend the context with a definition and the remaining rules:

```
# [ shr_lam = [x:v->o]shriek (lam x) ]
# [ beta: {x|v->o}{y:v}eq (app (shr_lam x) (shriek y)) (x y) ]
# [ xi: {x,y|v->o}({z:v}eq (x z) (y z))->
#                       (eq (shr_lam x) (shr_lam y)) ]
# [ eta: {x:v}eq (shr_lam [y:v]app (shriek x) (shriek y))
#                (shriek x)) ] ;
define   shr_lam : (v->o)->o
declare  beta xi eta
```

The definition of `shr_lam` is used for notational convenience, and makes statement of the rules shorter and more uniform than a direct translation from Section 5.2.

7.2.2 Example Proof

We show that the call-by-value lambda calculus has a form of extensionality (compare this example with the similar proof in Section 5.1). First, for notational convenience, define a version of `app` restricted to values:

```
# [ app_shr = [x:v]app (shriek x) ] ;
define app_shr : v->o->o
```

The command `Goal` starts a refinement proof by making a conjecture into a refinement goal:

```
# Goal {M,N:v}({x:o}eq (app_shr M x) (app_shr N x))->
#                              (eq (shriek M) (shriek N)) ;
?0 : {M,N:v}({x:o}eq (app_shr M x) (app_shr N x))->
                              (eq (shriek M) (shriek N))
```

LEGO's response shows the conjecture has become a goal, in this case goal `?0`. We may think of `?0` as a meta-variable; i.e. a totally uninstantiated term whose type is the conjecture. Finding an instantiation of `?0` having this type is what we mean by 'proving the conjecture'.

The first step in the proof is to use the Π-introduction rules (i.e rules 6 or 12 of Section 2; the rules are used 'backwards' because we are constructing a proof by refinement) to extend the context with the hypotheses of our conjecture:

```
# Intros M N h ;
M : v
N : v
h : {x:o}eq (app_shr M x) (app_shr N x)
?1 : eq (shriek M) (shriek N)
```

`Intros` is a simple tactic that maps the basic Π-introduction rule across a list of identifiers that become the names of the new declarations. LEGO shows a new goal in an extended context.

In order to see what to do next, remember that we are building a proof by refinement, and ask what rules of the call-by-value lambda calculus have *conclusions* that match the current goal? Only `E1` (symmetry of `eq`) and `E2` (transitivity of `eq`) can be used. Symmetry gets us nowhere, so we use transitivity, `E2`, to separate the sides of the equation. To understand how this works, we need to consider the refinement process more closely.

Resolution in LEGO The basic refinement step of LEGO is resolution: use some rule of the object theory (or a derived rule, or theorem) like a Prolog clause, unifying its 'head' with some goal, and returning its 'body', instantiated with the unifying substitution, as new goals. Two questions are raised:

1. What is the 'head' and what is the 'body' of an arbitrary type? For example, should `A->B->C` be read as the 'clause' `C:- A,B.`, or `(B->C):- A.`, or even `A->B->C.`? All of these are possible readings, and LEGO's built-in resolution tactic searches through them all, starting with the last one (which is the best match, because it produces no new subgoals), working back to the first, until finding a reading whose head unifies with the goal.

2. Which variables in a 'clause' are unification variables, and which must be treated as constants by unification? Consider the clause `A:- B`. All variable occurrences in A bound in B are free for unification, all others are constants for unification. For example, if

```
{a:A}{b:(B a)}(C a b)
```

is read as the clause

```
{b:(B a)}(C a b):- a:A,
```

then `a` is a unification variable, but `b` is a unification constant. (Actually, LEGO substitutes new meta-variables, `?n`, for the unification variables, so this example will look like `{b:(B ?n)}(C ?n b):- ?n:A` for some `n`.)

There is technical point that this informal comparison with Prolog fails to address. In Prolog, all variables are globally bound by implicit existential quantifiers, but in LEGO, as in Isabelle [38], there can be quantifier alternation, and the unification algorithm must account for this if resolution is to be sound. See [33, 38, 44, 47] for more discussion of this point.

Coming back to the example, we use the built-in resolution tactic, `Refine`, to resolve goal `?1` by the transitivity rule `E2`.

```
# Refine E2
  ?3 : o
  ?5 : eq (shriek M) ?3
  ?6 : eq ?3 (shriek N)
```

`Refine` (after some search) reads `E2` as the clause

```
(eq ?2 ?4) :- ?2:o, ?3:o, ?4:o, ?5:(eq ?2 ?3), ?6:(eq ?3 ?4),
```

unifies `(eq ?2 ?4)` with goal `?1:(eq (shriek M) (shriek N))` thereby instantiating `?2` and `?4`, and returns the remaining new goals.

Now rule `eta` applies to goal `?6`. By default `Refine` resolves with the first goal in the list, but we can say explicitly which goal to refine:

```
# Refine 6 eta ;
?5 : eq (shriek M) (shr_lam [y:v]app (shriek N) (shriek y))
```

Notice that the unification with goal `?6` also solved goal `?3`, and substituted its solution for the occurrence of `?3` in goal `?5`. Now it's clear we can use similar steps to apply `eta` to an equation involving `shriek M`.

```
# Refine E2 ;
?9 : o
?11 : eq (shriek M) ?9
?12 : eq ?9 (shr_lam [y:v]app (shriek N) (shriek y))

# Refine 11 E1 ;
?13 : o
?15 : eq ?13 (shriek M)
?12 : eq ?9 (shr_lam [y:v]app (shriek N) (shriek y))

# Refine 15 eta ;
?12 : eq (shr_lam [y:v]app (shriek M) (shriek y))
                 (shr_lam [y:v]app (shriek N) (shriek y))
```

Now rule `xi` can be used, which is what we've been working toward.

```
# Refine xi ;
?19 : {z:v}eq (app (shriek M) (shriek z))
              (app (shriek N) (shriek z))
```

The proof can be finished using the hypothesis introduced in the context as `h`.

```
# Intros z ;
z : v
?20 : eq (app (shriek M) (shriek z))
         (app (shriek N) (shriek z))

# Refine h ;
Discharge..  z
Discharge..  h N M
*** QED ***
```

Finding no remaining goals in the current context, LEGO discharges local declarations and assumptions, `z`, `h`, `N`, and `M`. Finding no remaining goals in the discharged context, and no further discharges to do, LEGO prints `*** QED ***`.

We have finished the proof without actually entering a compound term, letting unification construct the proof term for us. In practice it is often convenient to do some proof steps 'bottom up', i.e. just type in a term that satisfies a goal. LEGO supports this, allowing refinement by any term well-typed in the current context. To see an example, we undo the last two steps

```
# Undo 2
?19 : {z:v}eq (app (shriek M) (shriek z))
              (app (shriek N) (shriek z))
```

and solve this goal explicitly

```
# Refine [z:v]h (shriek z) ;
Discharge..  h N M
*** QED ***
```

We can save the actual proof term as a definition in the context:

```
#Save funny_Ext ;
funny_Ext  saved
```

and examine it's value and type, as for any other well-typed term, by 'evaluating' it at the top level.

```
# funny_Ext ;
value = [M,N:v]
        [h:{x:o}eq (app_shr M x) (app_shr N x)]
        E2 (E2 (E1 (eta M))
               (xi [z:v]h (shriek z)))
           (eta N)
type  = {M,N:v}
        ({x:o}eq (app_shr M x) (app_shr N x))->
                               (eq (shriek M) (shriek N))
```

The underlying pure LF term represented by this pretty-printed syntax is very much bigger; the annotations for argument synthesis allow this abbreviated form to be printed, parsed, and type-checked. Unlike systems with very general syntax, LEGO will correctly parse and type-check any concrete term that it prints.

Now `funny_Ext` can be used as a lemma in proofs just as if it had been declared as a constant of the object system.

7.3 Further issues in LEGO

Tactics In the example above, we used only two tactics, `Refine` and `Intros`, both very general. For an LF based mechanization of a formal system to be comparable with a specially coded implementation, we believe a library of logic-specific tactics will be needed. For an implementation of LF to meet the goal of a general logic theorem prover mentioned in Section 1, it should supply a library of 'meta-tactics' applicable to classes of logics, and useful as building blocks for logic-specific tactics. While LEGO supports user tactics written in CAML, no large body of tactics, either general or logic specific, has yet been coded.

Theories Mechanization of even small mathematical examples suggests the need for some modular, parametric notion of *theory*. For example, both the additive and multiplicative aspects of a ring have monoid structure. We don't want to re-prove monoid theorems about every mathematical structure containing a monoid; but rather build specialized theories (e.g. `Ring`) out of more general ones (e.g. `Monoid`) in such a way that general theorems are inherited. LEGO has a primitive notion of theory.

Examples Among the examples formalized in LEGO are Lagrange's Theorem for groups, the Binomial Theorem for rings [52], a proof of bisimulation (using maximal fixpoint induction) [52], some elementary topology and domain theory, development of several sorting algorithms (bubble sort, exchange sort, quicksort), and work on operational semantics of programming languages [36].

References

[1] Arnon Avron. Simple Consequence Relations. *Information and Computation* 92:105–139, 1991

[2] Arnon Avron. The semantics and Proof Theory of Linear Logic. *Theoretical Computer Science* 57:161–184, 1988.

[3] Arnon Avron. Modal Logics in the Edinburgh LF in [4].

[4] A. Avron, R. Harper, F. Honsell, I. Mason, and G. Plotkin (Editors) *Workshop on General Logic — Edinburgh 1987.* Technical Report, Laboratory for the Foundations of Computer Science, Edinburgh University, 1988. ECS-LFCS-88-52

[5] A.Avron, F.Honsell, and Mason, I. *An Overview of the Edinburgh Logical Framework.* in: Current Trends in Hardware Verifications and Automated Theorem Proving., Edited by G.Birtwistle and P.A.Subramanyam, Springer-Verlag, 1989.

[6] A.Avron and F.Honsell and Mason, I.A., Using Typed Lambda Calculus to Implement Formal Systems on a Machine. Technical Report, Laboratory for Foundations of Computer Science, University of Edinburgh, ECS-LFCS-87-31.

[7] H. Barendregt, *The Lambda Calculus - Its syntax and semantics, revised edition.* North Holland, 1984.

[8] H. Barringer, J.H. Cheng and C.B. Jones. A Logic Covering Undefinedness in Program Proofs. *Acta Informatica*, 21:251–269, 1984.

[9] 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.

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

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

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

[13] Thierry Coquand, Gilles Dowek, Gérard Huet, and Christine Paulin-Mohring. *The Calculus of Constructions; Documentation and user's guide.* Projet Formel, INRIA-ENS, July 1989.

[14] 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.

[15] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Control*, 76:95–120, 1988.

[16] Fagin R., Halpern J.Y., and Vardi M. What Is an Inference Rule? To appear in the Journal of Symbolic Logic.

[17] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language* PhD thesis, University of Pennsylvania, August 1989.

[18] Amy Felty., and Dale Miller. Specifying Theorem Provers in a Higher-Order Logic Programming Language. in *Ninth International Conference on Automated Deduction.* Argonne, Il, May 1988.

[19] Amy Felty., and Dale Miller. Encoding a dependent-type λ-calculus in a logic programming language. in *Tenth International Conference on Automated Deduction.* Kaiserslautern, Germany, July 1990.

[20] Mariangiola Dezani, Furio Honsell and Simonetta Ronchi della Rocca. Models for Theories of Functions Strictly Depending on all their Arguments. *Journal of Symbolic Logic* 51:3, 1986. Abstract.

[21] Jean-Yves Girard. Linear Logic. *Theoretical Computer Science* 50:1, 1987.

[22] 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.

[23] Timothy Griffin. *An Environment for Formal Systems.* Technical Report, Laboratory for the Foundations of Computer Science, Edinburgh University, 1987. ECS-LFCS-87-34.

[24] Robert Harper, Furio Honsell, Gordon Plotkin. A Framework for Defining Logics. *Proceedings of the Second Annual Symposium on Logic in Computer Science*, Cornell, 1987. (The full version will appear in Journal of the ACM)

[25] G. Huet, and G. Plotkin. (Editors) Logical Frameworks, Cambridge University Press, 1991.

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

[27] Fred Kröger. *Temporal Logic of Programs.* Volume 8 of *EATCS Monographs on Theoretical Computer Science*, Springer-Verlag, 1987.

[28] Zhaohui Luo. ECC, an Extended Calculus of Constructions. *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, Pacific Grove, 1989.

[29] Zhaolui Luo, Robert Pollack, and Paul Taylor. How to Use Lego; A preliminary user's manual. Laboratory for the Foundations of Computer Science, Edinburgh University, April 1989.

[30] 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.

[31] 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.

[32] Ian A. Mason. *Hoare's Logic in the LF.* Technical Report, Laboratory for the Foundations of Computer Science, Edinburgh University, 1987. ECS-LFCS-87-32

[33] Dale Miller. *Solutions to λ-Term Equations Under a Mixed Prefix.* Unpublished draft, Department of Computer and Information Sciences, University of Pennsylvania, January 1989.

[34] 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.

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

[36] Christian-Emil Ore. *On Natural Deduction Style Semantics, Environments and Stores.* Technical Report, Laboratory for the Foundations of Computer Science, Edinburgh University, 1989. ECS-LFCS-89-88.

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

[38] Lawrence Paulson. *The Foundation of a Generic Theorem Prover.* Technical Report 130, University of Cambridge Computer Laboratory, March 1988.

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

[40] Frank Pfenning. Partial Polymorphic Type Inference and Higher-Order Unification. *Proceedings of the 1988 ACM Lisp and Functional Programming Conference.*

[41] Frank Pfenning. Elf: A Language for Logic Definition and Verified Metaprogramming. *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, Pacific Grove, 1989.

[42] Frank Pfenning. Logic Programming in the LF logical framework. in [25]

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

[44] Robert Pollack. *The Theory of LEGO.* Technical Report, Laboratory for the Foundations of Computer Science, Edinburgh University, 1989. In preparation.

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

[46] David Pym. *Proofs, Search and Computation in General Logic.* Ph.D. Dissertation, University of Edinburgh. In preparation, 1989.

[47] David Pym and Lincoln Wallen. *Effective Search for the Logical Framework.* Technical Report, Laboratory for the Foundations of Computer Science, Edinburgh University. In preparation, 1989.

[48] Reynolds, J.C. Syntactic Control of Interference. *Conference Record of the Fifth Annual Symposium on Principles of Programming Languages*, Tucson, 1978.

[49] Anne Salvesen, A proof of the Church-Rosser property for the Edinburgh LF with eta-conversion Proceedings of the first Workshop on Logical Frameworks, Sophia Antipolis 1990.

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

[51] Colin Stirling. Modal Logics for Communicating Systems. *Theoretical Computer Science*, 49:311–347, 1987.

[52] Paul Taylor. *Playing with LEGO: Some Examples of Developing Mathematics in the Calculus of Constructions.* Technical Report, Laboratory for the Foundations of Computer Science, Edinburgh University, 1989. ECS-LFCS-89-89.

[53] Pierre Weis, et al. *The CAML Reference Manual, Version 2.6.* Projet Formel, INRIA-ENS, March 1989.

Contents