An Overview of the Edinburgh Logical Framework

Arnon Avron, Furio Honsell and Ian A. Mason
Laboratory for Foundations of Computer Science*

July 18, 1996

1 Introduction

In recent years there has been a growing interest in using computers as an aid for correctly manipulating logical systems, as well as using formal systems for correctly designing computers. 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), currently under development at the LFCS. 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 paper [7] describes the basic features of the LF, while the paper [1] provides a broader illustration of its applicability and discusses to what extent it is successful. This paper serves as an introduction to the LF and summarizes the main points made in [1]. It is organized as follows. In section 2 we provide an outline of the LF specification language. This is done in somewhat more detail than is necessary on first reading. In section 3 we give a simple example of a specification. In section 4 we discuss the general LF paradigm for specifying a logical system. The subsequent sections illustrate this paradigm. Section 5 deals with modal logics, section 6 deals with various lambda calculii and in section 7 we discuss program logics.

2 The LF Specification Language

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 [4], to Martin Löf's early type theories and to Meyer and Reinhold's λ^π [10]. It 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,

*Computer Science Department, Edinburgh University, Edinburgh, EH9 3JZ.

i.e. assertions about typed valued functions. The only type (or kind) constructor is the dependent product, Π. The abstract syntax is given by the following:

Kinds	K	::=	$\text{Type} \mid \prod_{x:A} . K$
Type Families	A	::=	$c \mid \prod_{x:A} . B \mid \lambda x : A.B \mid AM$
Objects	M	::=	$c \mid x \mid \lambda x : A.M \mid MN$

We let M and N range over expressions for objects, A and B for types and families of types, K for kinds, x and y over variables, and c over constants. Types are used to classify objects, while kinds are used to classify types and type families. The kind Type classifies the basic types. The other kinds classify type families, i.e. functions from objects to types. 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.

The theory we shall deal with is a formal system for deriving assertions of one of the following shapes:

$\Gamma \vdash_\Sigma K$	K is a kind
$\Gamma \vdash_\Sigma A : K$	A has kind K
$\Gamma \vdash_\Sigma M : A$	M has type A

where the syntax for Signatures and contexts is specified by the following grammar:

Signatures	Σ	::=	$\langle\rangle \mid \Sigma, c : K \mid \Sigma, c : A$
Contexts	Γ	::=	$\langle\rangle \mid \Gamma, x : A$

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. They are grouped according to which of the three forms of assertions they concern, α-conversion is assumed throughout.

1. **Valid Kinds**

(1) $$\frac{}{\vdash \text{Type}}$$

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

2. **Valid Elements of a Kind**

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

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

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

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

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

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

3. **Valid Elements of a Type**

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

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

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

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

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

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

A term is said to be *well–typed in a signature and context* if it can be shown to either be a kind, have a kind, or have a type in that signature and context. A term is *well–typed* if it is well–typed in some signature and context. The notion of $\beta\eta$–contraction, written $\rightarrow_{\beta\eta}$, can be defined both at the level of objects and at the level of types and families of types in the obvious way. Rules (8) and (14) make use of a relation $=_{\beta\eta}$ between terms which is defined as follows: $M =_{\beta\eta} N$ iff $M \rightarrow^*_{\beta\eta} P$ and $N \rightarrow^*_{\beta\eta} P$ for some term P. The following theorem from [7] 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* $M \rightarrow^*_{\beta\eta} M'$*, 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 A Simple Specification of a Logic

Encoding the classical propositional calculus (with the connectives $\neg$ and $\supset$) provides a simple example of how, in general, a logic is encoded. The set of well formed formulas, o, is represented as a basic type. This is achieved by declaring it to be of kind Type. The connectives then correspond to unary and binary functions on o. This is summarized in the following.

- Syntactic Category of Formulas

$$o \ : \ \text{Type}$$

- Operations

$$\begin{array}{lll} \neg & : & o \to o \\ \supset & : & o \to o \to o \end{array}$$

Propositional atoms (or variables) are then simply constants (or LF variables) of type o. The judgement (or predicate encoding the assertion) that a formula is provable is encoded as a family of types.

- Judgement

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

The set of proofs of a formula Φ is identified, in this encoding, with the basic type $\mathrm{T}(\Phi)$. Consequently showing that a formula Φ is provable reduces to producing a term of type $\mathrm{T}(\Phi)$. Similarly checking whether or not an object M is a proof of Φ reduces to checking whether or not the object M has type $\mathrm{T}(\Phi)$. This style of encoding is refered to as *the judgements as types principle* in the paper [7].

There are several systems for constructing proofs in the propositional calculus. Two well known examples, which we shall treat in this section, are Hilbert style and natural deduction style systems. Encoding a Hilbert style system involves introducing a constant for each axiom schema together with a constant corresponding to modus ponens. The following is one such system.

- Hilbert Style Axioms and Rules

$$\begin{array}{lll} \mathrm{A}_1 & : & \prod_{\Phi_1,\Phi_2:o} \quad \mathrm{T}(\Phi_1 \supset (\Phi_2 \supset \Phi_1)) \\ \mathrm{A}_2 & : & \prod_{\Phi_1,\Phi_2,\Phi_3:o} \quad \mathrm{T}(\Phi_1 \supset (\Phi_2 \supset \Phi_3) \supset (\Phi_1 \supset \Phi_2) \supset (\Phi_1 \supset \Phi_3)) \\ \mathrm{A}_3 & : & \prod_{\Phi_1,\Phi_2:o} \quad \mathrm{T}((\neg\Phi_1 \supset \neg\Phi_2) \supset (\Phi_2 \supset \Phi_1)) \\ \mathrm{MP} & : & \prod_{\Phi_1,\Phi_2:o} \quad \mathrm{T}(\Phi_1 \supset \Phi_2) \to \mathrm{T}(\Phi_1) \to \mathrm{T}(\Phi_2) \end{array}$$

Encoding a natural deduction style system, on the other hand, involves introducing constants corresponding to each introduction and elimination rule. In this case the following declarations suffice.

- Natural Deduction Style Rules

$$\begin{array}{lll} \supset\mathrm{E} & : & \Pi_{\Phi_1,\Phi_2:o} \quad \mathrm{T}(\Phi_1 \supset \Phi_2) \to \mathrm{T}(\Phi_1) \to \mathrm{T}(\Phi_2) \\ \supset\mathrm{I} & : & \Pi_{\Phi,\Psi:o} \quad (\mathrm{T}(\Phi) \to \mathrm{T}(\Psi)) \to \mathrm{T}(\Phi \supset \Psi) \\ \neg\mathrm{I} & : & \Pi_{\Phi,\Psi:o} \quad (\mathrm{T}(\Phi) \to \mathrm{T}(\Psi)) \to (\mathrm{T}(\Phi) \to \mathrm{T}(\neg\Psi)) \to \mathrm{T}(\neg\Phi) \\ \neg\mathrm{E} & : & \Pi_{\Phi:o} \quad \mathrm{T}(\neg\neg\Phi) \to \mathrm{T}(\Phi) \end{array}$$

In both versions schemata are encoded as functions mapping formulas to the instatiated axioms and rules. Instantiation of a schema is thus simply application. For example, if $\Phi : o$ is a formula, then $\supset\mathrm{I}(\Phi)(\Phi)$ inhabits the type $(\mathrm{T}(\Phi) \to \mathrm{T}(\Phi)) \to \mathrm{T}(\Phi\supset\Phi)$. Hence to show that $\Phi\supset\Phi$ is provable, it suffices to show that $\mathrm{T}(\Phi\supset\Phi)$ is inhabited, which reduces to producing an element of type $\mathrm{T}(\Phi) \to \mathrm{T}(\Phi)$. The identity function $\lambda x : \mathrm{T}(\Phi).x$ satisfies this requirement. Thus

$$\supset\mathrm{I}(\Phi)(\Phi)(\lambda x : \mathrm{T}(\Phi).x) : \mathrm{T}(\Phi\supset\Phi).$$

The $\supset$I rule also provides an example of how assumption discharge is handled in the LF. To prove $\Phi\supset\Psi$ it suffices to assume that Φ is true, by introducing a variable $x : \mathrm{T}(\Phi)$, and producing a proof of Ψ, i.e. an object $B(x) : \mathrm{T}(\Psi)$. The initial assumption that Φ is true is discharged by forming $\lambda x : \mathrm{T}(\Phi).B(x)$ and supplying it to $\supset$I. Thus both assumption discharge and the schematic nature of rules and axioms is handled by lambda abstraction.

One of the positive features of the LF is that proofs of theorems, derivable rules, axioms and basic rules are treated uniformly, they are all simply LF objects of a particular type. So for example, in the natural deduction style system we could introduce a new connective encoding conjunction together with constants encoding its introduction and elimination rules as follows:

- Conjunction

$$\wedge \quad : \quad o \to o \to o$$

- Introduction and Elimination Rules for Conjuction

$$\begin{array}{lcl} \wedge\mathrm{I} & : & \Pi_{\Phi,\Psi:o}\ \ \mathrm{T}(\Phi) \to \mathrm{T}(\Psi) \to \mathrm{T}(\Phi \wedge \Psi) \\ \wedge\mathrm{E_l} & : & \Pi_{\Phi,\Psi:o}\ \ \mathrm{T}(\Phi \wedge \Psi) \to \mathrm{T}(\Phi) \\ \wedge\mathrm{E_r} & : & \Pi_{\Phi,\Psi:o}\ \ \mathrm{T}(\Phi \wedge \Psi) \to \mathrm{T}(\Psi) \end{array}$$

A more frugal, but equivalent approach, would be to let $\wedge$ denote the LF term

$$\lambda x : o.\lambda y : o.\neg(x\supset\neg y).$$

Thus $\wedge$ is of type $o \to o \to o$. One could then construct LF terms, out of the constants governing $\neg$ and $\supset$, which inhabited the same type as the rules for conjunction above. For example, suppose that $x : \mathrm{T}(\Phi)$ and $y : \mathrm{T}(\Psi)$. Then letting

$$\begin{array}{ll} \Delta_\Psi & = \lambda z : \mathrm{T}(\Phi\supset\neg\Psi).y \\ \Delta_{\neg\Psi} & = \lambda z : \mathrm{T}(\Phi\supset\neg\Psi)\supset\mathrm{E}(\Phi)(\neg\Psi)(z)(x) \end{array}$$

we have that

$$\begin{array}{ll} \Delta_\Psi & : \mathrm{T}(\Phi\supset\neg\Psi) \to \mathrm{T}(\Psi) \\ \Delta_{\neg\Psi} & : \mathrm{T}(\Phi\supset\neg\Psi) \to \mathrm{T}(\neg\Psi). \end{array}$$

Applying $\neg$I yields

$$\Delta \quad = \neg\mathrm{I}(\Phi\supset\neg\Psi)(\Psi)(\Delta_\Psi)(\Delta_{\neg\Psi}),$$

which is a term of type $\mathrm{T}(\neg(\Phi\supset\neg\Psi))$, which by definition is the same as $\Delta : \mathrm{T}(\Phi\wedge\Psi)$. Discharging the assumptions regarding the truth of Φ and Ψ gives

$$\lambda x : \mathrm{T}(\Phi).\lambda y : \mathrm{T}(\Psi).\Delta \quad : \quad \mathrm{T}(\Phi) \to \mathrm{T}(\Psi) \to \mathrm{T}(\Phi \wedge \Psi),$$

which is the same type as the constant $\wedge$I above.

Thus derived rules and basic rules are treated as equal in the LF. The case is somewhat different for admissible rules however. Both the encoded natural deduction system and the encoded Hilbert system are equivalent in the sense that

$$\Phi_1, \ldots, \Phi_n \vdash \Psi$$

in the classical propositional calculus iff there exists an LF term Δ such that, assuming $p_1 : o, \ldots, p_m : o$ (the $p_1, \ldots, p_m$ being the atomic variables occurring in $\Phi_1, \ldots \Phi_n$ and Ψ), we can show that

$$\Delta : \mathrm{T}(\Phi_1) \to \cdots \to \mathrm{T}(\Phi_n) \to \mathrm{T}(\Psi)$$

in either of the above LF encodings (we use the same symbol for denoting a formula and the corresponding term in LF).

This does not mean they have that same *higher order rules*. For example in the Hilbert system the type

$$\Pi_{\Phi,\Psi:o}(\mathrm{T}(\Phi) \to \mathrm{T}(\Psi)) \to \mathrm{T}(\Phi \supset \Psi)$$

corresponds to the non-trivial direction of the deduction theorem. The theorem is proved by induction on the complexity of proofs. Since no such proof procedure is available in the LF presentation, we cannot produce a term which inhabits this type. In the natural deduction system the type is inhabited by the basic rule $\supset$I.

We can extend either of the above specifications to encompass full predicate logic. In this presentation we will be content with extending the natural deduction style system. We begin by adding a type, i, encoding the set of individuals (over which the quantifiers will range) together with the equality predicate on them.

- Additional Syntactic Category

$$i \ : \ \text{Type}$$

- Additional Operation

$$= \ : \ i \to i \to o$$

The rules governing equality are then easily stated.

- Equality Rules

$$\begin{array}{lcll} \mathrm{E}_0 & : & \Pi_{x:i} & \mathrm{T}(x = x) \\ \mathrm{E}_1 & : & \Pi_{\substack{x,y:i \\ t:i\to i}} & \mathrm{T}(x = y) \to \mathrm{T}(t(x) = t(y)) \\ \mathrm{E}_2 & : & \Pi_{\substack{x,y:i \\ \Phi:i\to o}} & \mathrm{T}(x = y) \to \mathrm{T}(\Phi(x)) \to \mathrm{T}(\Phi(y)) \end{array}$$

Note that the LF encoding of substitution is carried out by lambda abstraction and application. Thus the LF encoding of the rule

$$\frac{x = y}{\Phi[x/z] = \Phi[y/z]}$$

is schematic not in $\Phi : o$ but rather in $\Psi = \lambda z.\Phi$ of type $i \to o$. Thus $\Phi[x/z]$ is simply encoded as $\Psi(x)$. Binding operators are handled similarly. We provide three examples of 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. The only rule concerning the choice operator is the introduction rule:

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

The binding operators are encoded as follows.

- Binding Operators

$$\begin{array}{lcl} \epsilon & : & (i \to o) \to i \\ \exists & : & (i \to o) \to o \\ \forall & : & (i \to o) \to o \end{array}$$

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 $\forall x.x = x$.

Representing binding operators as constructors of higher order type allows for α-conversion and substitution to be taken care of by the LF, rather than axiomatized by the encoding. Another consequence of this method of encoding is that we can identify the variables of the object logic with those of the LF (of type i). It also allows for a smooth representation of instantiating a quantified formula (and generalizing and instantiated one) as the constants encoding the introduction and elimination rules below indicate.

- Introduction and Elimination Rules

$$\begin{array}{lcll} \epsilon\mathrm{I} & : & \Pi_{\Phi:i\to o} & \mathrm{T}(\exists(\Phi)) \to \mathrm{T}(\Phi(\epsilon(\Phi))) \\ \exists\mathrm{E} & : & \Pi_{\substack{\Phi:i\to o \\ \Psi:o}} & \mathrm{T}(\exists(\Phi)) \to (\Pi_{x:i}\mathrm{T}(\Phi(x)) \to \mathrm{T}(\Psi)) \to \mathrm{T}(\Psi) \\ \exists\mathrm{I} & : & \Pi_{\substack{\Phi:i\to o \\ t:i}} & \mathrm{T}(\Phi(t)) \to \mathrm{T}(\exists(\Phi)) \\ \forall\mathrm{E} & : & \Pi_{\substack{\Phi:i\to o \\ t:i}} & \mathrm{T}(\forall(\Phi)) \to \Phi(t) \\ \forall\mathrm{I} & : & \Pi_{\Phi:i\to o} & (\Pi_{t:i}\mathrm{T}(\Phi(t))) \to \mathrm{T}(\forall(\Phi)) \end{array}$$

Note that side conditions on variable occurrence are handled by the scoping conventions of the underlying lambda calculus. For example, the introduction rule for $\forall$ is usually stated as

$$\frac{\Phi(x)}{\forall x \Phi(x)}$$

with the side condition that the variable x does not occur free in any assumption that Φ depends on. The side condition is encoded in the LF by requiring a parametric proof, which instantiated at any variable $x : i$ yields a proof of $\Phi(x)$. This slight rewording of the rule is easily seen to be equivalent to the classical formulation.

An example of a proof in this system is

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

where 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. The adequacy of this representation is expressed in the following theorem.

Theorem 2 *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. $(\exists M)(\Gamma \cup \{y_1 : \mathrm{True}(\Psi_1), \ldots, y_k : \mathrm{True}(\Psi_k)\} \vdash M : \mathrm{True}(\Psi))$

   iff

   $$\Phi_\Gamma(\Psi_1), \ldots, \Phi_\Gamma(\Psi_k) \vdash \Phi_\Gamma(\Psi).$$

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*

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

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

4 The LF Paradigm for Specifying a Logical System

In this section we outline the LF paradigm for specifying a logic and elaborate on some of the points made in the previous section. A typed λ-calculus is used as the 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.

In the LF language a logical system is specified by a finite list of typed constants, called a *signature*. 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 centred on the notion of *judgement*. This notion was introduced by Martin-Löf [8] 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. 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.

The consequence relations which are directly encoded by the LF's judgements are ordinary single-conclusioned consequence relations [2]. 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 or even logical facts relating two or more such relations, e.g Modal logics.

A number of classical side conditions on rules concerning binding operators and connectives can be handled unproblematically. In other cases additional judgements need to be introduced. 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 ($\beta - \eta$ equivalence classes of an) LF type, corresponding to that category, onto the category itself.

5 Modal Logics

Standard presentations of modal logics are problematic even in the simplest case, Hilbert systems. In these systems one usually has, apart from axiom schemes, two rules of inference: Modus Ponens and necessitation (from Φ infer $\Box\Phi$). The latter, however, is taken to be only a *rule of proof*: its application to a premise is permitted only if the premise does not depend on any assumptions (i.e. is a theorem). It is not the case, therefore, that $\Box\Phi$ *follows* from Φ in such systems. Thus it would not be sound to encode the necessitation rule by simply introducing a constant, Nec, of type $\prod_{\Phi:o} \mathrm{T}(\Phi) \to \mathrm{T}(\Box\Phi)$ (where T corresponds to the intended consequence relation).

The solution to this problem illustrates the power gained by simultaneously employing different judgements. In the case of S4 we introduce *two* judgements: True and Valid. The first corresponds to the intended consequence relation, $\vdash_t$, in which necessitation is only a rule of proof. The second one corresponds to the consequence relation, $\vdash_v$, obtained by taking both rules as pure rules of derivation. The complete signature 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

$$\begin{array}{lcll}
\mathrm{C} & : & \prod_{\Phi:o} & \mathrm{Valid}(\Phi) \to \mathrm{True}(\Phi) \\
\mathrm{A}_1 & : & \prod_{\Phi_1,\Phi_2:o} & \mathrm{Valid}(\Phi_1 \supset (\Phi_2 \supset \Phi_1)) \\
\mathrm{A}_2 & : & \prod_{\Phi_1,\Phi_2,\Phi_3:o} & \mathrm{Valid}(\Phi_1 \supset (\Phi_2 \supset \Phi_3) \supset (\Phi_1 \supset \Phi_2) \supset (\Phi_1 \supset \Phi_3)) \\
\mathrm{A}_3 & : & \prod_{\Phi_1,\Phi_2:o} & \mathrm{Valid}((\neg\Phi_1 \supset \neg\Phi_2) \supset (\Phi_2 \supset \Phi_1)) \\
\mathrm{A}_4 & : & \prod_{\Phi:o} & \mathrm{Valid}(\Box\Phi \supset \Phi) \\
\mathrm{A}_5 & : & \prod_{\Phi_1,\Phi_2:o} & \mathrm{Valid}(\Box(\Phi_1 \supset \Phi_2) \supset (\Box\Phi_1 \supset \Box\Phi_2)) \\
\mathrm{A}_6 & : & \prod_{\Phi:o} & \mathrm{Valid}(\Box\Phi \supset \Box\Box\Phi) \\
\mathrm{MP}_T & : & \prod_{\Phi_1,\Phi_2:o} & \mathrm{True}(\Phi_1 \supset \Phi_2) \to \mathrm{True}(\Phi_1) \to \mathrm{True}(\Phi_2) \\
\mathrm{MP}_V & : & \prod_{\Phi_1,\Phi_2:o} & \mathrm{Valid}(\Phi_1 \supset \Phi_2) \to \mathrm{Valid}(\Phi_1) \to \mathrm{Valid}(\Phi_2) \\
\mathrm{Nec} & : & \prod_{\Phi:o} & \mathrm{Valid}(\Phi) \to \mathrm{Valid}(\Box\Phi)
\end{array}$$

An example of a proof is $\Delta : \prod_{\Phi:o} \mathrm{True}(\Box(\Box\Phi \supset \Phi) \supset \Box\Phi) \to \mathrm{True}(\Phi)$, where Δ is the following:

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

$\Box(\Box\Phi \supset \Phi) \supset \Box\Phi$ is the characteristic axiom of GL— the famous modal system for provability in Peano arithmetic. The above is a proof that in S4 any Φ-instance of this formula actually entails Φ.

Theorem 3 *In the Hilbert-type system which is obtained by adding $\Phi_1, \ldots, \Phi_n$ as axioms to S4,*

$$\Psi_1, \ldots, \Psi_m \vdash \vartheta$$

if and only if there exists a term Δ of type

$$(\dagger) \quad \mathrm{Valid}(\Phi_1) \to \cdots \to \mathrm{Valid}(\Phi_n) \to \mathrm{True}(\Psi_1) \to \cdots \mathrm{True}(\Psi_m) \to \mathrm{True}(\vartheta)$$

in the context $p_1 : o, \ldots, p_m : o$ and the above signature. Where $p_1, \ldots, p_m$ are the atomic variables occuring in $\Phi_1, \ldots \Phi_n, \Psi_1, \ldots \Psi_m$ and Ψ.

The above method for handling rules of proof is not specific to modal logic. In the above case the consequence relations have natural semantic interpretations in terms of Kripke models: $\bar{\Phi} \vdash_v \Psi$ iff Ψ is *valid* in any frame in which all the $\bar{\Phi}$ are valid (i.e. true in all worlds); $\bar{\Phi} \vdash_t \Psi$ iff Ψ is *true* in every world in which all the $\bar{\Phi}$ are true. Moreover, in the LF we can express and prove logical facts concerning *both* internalized consequence relations. For example a term of type † 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.

We turn now to the natural deduction formulation of S4, presented by Prawitz in [12]. It is obtained from the usual natural deduction formulation of classical propositional calculus by the addition of the following two rules:

$$(\Box \text{ Intro}) \quad \frac{\Phi}{\Box\Phi} \qquad\qquad \frac{\Box\Phi}{\Phi} \quad (\Box \text{ Elim})$$

The first rule has a side condition on its application. Prawitz gives several possible versions of this side condition. In the first one, for example, all assumptions on which Φ depends should be modal (i.e. the main connective is $\Box$). In all versions the side condition makes this rule *impure*. This impurity is of the *second degree* [2]. Thus we lack the coherence, which the LF paradigm expects, between the formulation of

the rules of a system and the consequence relation represented by it. In the Hilbert style presentation we can ignore the intended consequence relation of truth and still encode all proofs of theorems using only one judgement, validity. This is not possible here since the introduction rule for implication is not sound for validity.

A compact solution to this problem is to encode proofs of theorems using *two* judgements and model implication introduction in a more elaborate way. The two judgements are Taut and Valid , both of type $o \to$ Type. Taut encodes the usual consequence relation of classical propositional logic. Valid encodes the consequence relation of validity. The constants of the specification 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 has the same syntactic categories and operations as the previous example. We omit two groups of rules. The first group simply states that Taut behaves like truth in the usual natural deduction presentation of classical propositional calculus. The second group states that Valid(Φ) is equivalent to Valid($\Box\Phi$). The crucial rules are:

$$\begin{array}{lcll} \mathrm{C} & : & \prod_{\Phi:o} & \mathrm{Taut}(\Phi) \to \mathrm{Valid}(\Phi) \\ \mathrm{R} & : & \prod_{\Phi_1,\Phi_2:o} & (\mathrm{Taut}(\Phi_1) \to \mathrm{Valid}(\Phi_2)) \to (\mathrm{Valid}(\Phi_1) \to \mathrm{Valid}(\Phi_2)) \\ \supset\mathrm{I}_V & : & \prod_{\Phi_1,\Phi_2:o} & (\mathrm{Valid}(\Box\Phi_1) \to \mathrm{Valid}(\Phi_2)) \to \mathrm{Valid}(\Box\Phi_1 \supset \Phi_2) \end{array}$$

6 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 $\to$ 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.

We begin by discussing the case of the classical λ-calculus. To this end we define a basic LF type, o, encoding the set of λ-terms together with a judgement, $M = N$, intended to encode the assertion that the term M is $\alpha - \beta$-equal to the term N. In order to encode the β-reduction rule it is convenient to encode the λ-constructor as $\Lambda : (o \to o) \to o$. 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 App : $o \to o \to o$ encoding application. All this is summarized in the following.

- Syntactic Category

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

- Operations

  $$\begin{array}{lcl} \Lambda & : & (o \to o) \to o \\ \mathrm{App} & : & o \to o \to o \end{array}$$

- Judgements

$$= \quad : \quad o \to o \to \text{Type}$$

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.

- Axioms and Rules

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

Notice that there is no counterpart to α-conversion in the above signature. The fact that we have encoded the classical λ-calculus is the expressed by the following theorem.

Theorem 4 *The following hold:*

1. $x_1 : o, \ldots, x_n : o \vdash_{\Sigma_\Lambda} M : o \quad \text{iff} \quad \Phi_\Gamma(M) \in \Lambda$
2. $(\exists P)(x_1 : o, \ldots, x_n : o \vdash_{\Sigma_\Lambda} P : M = N) \quad \text{iff} \quad \vdash_\lambda \Phi_\Gamma(M) = \Phi_\Gamma(N)$

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

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

and

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

is a bijective function defined as follows

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

It is interesting to consider the possibility of extending $\vdash_\lambda$ from a unary to a binary consequence relation, and hence extend the above theorem to proofs from assumptions. However, the consequence relation that is encoded by considering the inhabitability of types like

$$M_1 = N_1 \to \ldots \to M_n = N_n \to M = N$$

is not exactly as one would hope. The way the ξ rule has been encoded is responsible for the discrepancy. 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 $\lambda z.zz$. In contrast to this there is no way of showing in the above signature that

$$\Phi_{\{x:o\}}(x(\Delta\Delta)) = \Phi_{\{x:o\}}(x(\Delta\Delta\Delta)) \rightarrow \Phi_{\emptyset}(\lambda x.x(\Delta\Delta)) = \Phi_{\emptyset}(\lambda x.x(\Delta\Delta\Delta)).$$

However we can show that

$$\prod_{x:o}(\Phi_{\{x:o\}}(x(\Delta\Delta)) = \Phi_{\{x:o\}}(x(\Delta\Delta\Delta))) \rightarrow \Phi_{\emptyset}(\lambda x.x(\Delta\Delta)) = \Phi_{\emptyset}(\lambda x.x(\Delta\Delta\Delta)).$$

The underlying reason for this discrepancy is that traditionally free variables in assumptions are implicitly taken as universally quantified. The consequence relation defined by such a convention is often referred to as the consequence relation of validity. On the contrary in the LF encoding we are forced to indicate explicitly if our assumptions are universally quantified, as in the case for the ξ-rule. We are in fact encoding the consequence relation of truth. If we were to encode the consequence relation of validity, problems similar to those encountered in Hoare's logic (see section 7) would arise. To summarize, if universal quantification in assumptions is not made explicit by means of Π, then the ξ-rule can only be utilized as a rule of proof. This situation corresponds to the encoding of the theory of λ-algebras. It is important to note that the consequence relations of truth and validity either for λ-algebras or for the λ-calculus all coincide if only closed assumptions are considered. In the remainder of this section we shall only discuss the truth-consequence relations.

The call-by-value λ-calculus (λ_v-calculus) [11] differs from the traditional λ-calculus in the formulation of the β-reduction rule:

$$(\lambda x \ . \ M)N = M[x := N]$$

provided that N is a value, i.e. either a variable or an abstraction.

The immediate problem in encoding the call-by-value λ-calculus is expressing the syntactic notion of being a variable. The solution is inspired by a denotational model for the calculus [5] where the functions are strict and the variables range only over $D - \{\perp\}$. The syntax is modelled using two syntactic categories, v for values and o for expressions, together with a mapping $! : v \rightarrow o$. This illustrates the general technique for handling subcategories in LF. The only bindable type is v, with binding operator $\Lambda_v : (v \rightarrow o) \rightarrow v$. The full signature is:

- Syntactic Categories

  o : Type
  v : Type

- Operations

  $!$: $v \rightarrow o$
  Λ_v : $(v \rightarrow o) \rightarrow v$
  App : $o \rightarrow o \rightarrow o$

- Judgement

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

- Axioms and Rules

$$\begin{array}{lcll} \mathrm{E}_0 & : & \Pi_{x:o} & x = x \\ \mathrm{E}_1 & : & \Pi_{x,y:o} & x = y \to y = x \\ \mathrm{E}_2 & : & \Pi_{x,y,z:o} & x = y \to y = z \to x = z \\ \mathrm{E}_3 & : & \Pi_{x,y,x',y':o} & x = y \to x' = y' \to \mathrm{App}(x,x') = \mathrm{App}(y,y') \\ \beta_v & : & \Pi^{x:v\to o}_{y:v} & \mathrm{App}(\Lambda_v(x)!, y!) = xy \\ \xi_v & : & \Pi_{x,y:v\to o} & (\Pi_{z:v} xz = yz) \to \Lambda_v(x)! = \Lambda_v(y)! \\ \eta_v & : & \Pi_{x:v} & \Lambda_v(\lambda y : v.\mathrm{App}(x!, y!) = x! \end{array}$$

- Example of a Proof

$$\beta_v(!) \quad : \quad \Pi_{x:v}\mathrm{App}(\Lambda_v(!)!, x!) = x!$$

The example of a proof included above demonstrates that

$$\Lambda_v(!)!$$

behaves like the identity (with respect to App) over values. 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 set, Λ_I, of terms of the λ_I-calculus is defined as in the classical calculus, except for the abstraction clause: if $M \in \Lambda_I$ and $x \in M$, then $\lambda x.M \in \Lambda_I$.

The problem of encoding the λ_I-calculus is enforcing the binding constructor λ_I to be defined only on *relevant* schemes. Two solutions can be given, again inspired by denotational models of the calculus. The first follows quite closely the model presented in [5]. A new constant $\bot$: o is introduced together with rules governing its behaviour. The predicate *being a relevant function* is encoded as:

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

The λ_I-constructor is $\Lambda_I : \prod_{x:o\to o} \mathrm{Rel}_1(x) \to o$. Constants appearing in the rule, other than the ones mentioned explicitly, are as in the previous signature. It is interesting to notice the role of judgements in the definition of the syntax.

The second approach 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 $\in$ and appropriate rules. 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: $\Lambda_I : \prod_{x:o\to o} .\mathrm{Rel}_2(x) \to o$.

The set Λ_L of terms of the linear-λ-calculus is inductively defined as follows:

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

Encoding the linear λ-calculus exploits the notion of a function being linear. A function $f : X \to Y$, where X and Y are upper semi-lattices with a least element, is linear iff its strict and distributive. This solution is based on an idea of Gordon Plotkin. The predicate *being a distributive function* is encoded as

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

the λ-linear constructor is then encoded by $\Lambda_L : \prod_{x:o\to o} x(\bot) = \bot \to L(x) \to o$. Of course the full signature includes enough rules to axiomatize the notion of upper semi-lattice with least element.

7 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

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

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. The assignment axiom for this system is $\{p[t/x]\}x := t\{p\}$. As usual, $p[t/x]$ stands for the result of substituting t for the free occurrences of x in p.

There are, at least, two 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, 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

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

would be incorrect, e.g. $\text{Ass}(y)(1)(\lambda u.\neg(y! = u)) : \ \vdash_h \{\neg(y! = 1)\}y := 1\{\neg(y! = y!)\}$.

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 [13], $\sharp_x$ is of type $l \to (x \to \texttt{TYPE})$. The intuitive meaning of the judgements can be explained, using infix notation, as follows: $x \sharp_l y$ is interpreted as meaning that

x and y denote distinct identifiers or locations. $x\sharp_i t$ 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. $x\sharp_o e$ 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 [9], 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.

Presentations of these systems in the literature are not uniform and often important syntactic decisions are not entirely motivated. Such systems may even benefit from the analysis required to encode them.

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 case we present here this number is two, the judgement $\vdash$ is therefore o f type $(l \to l \to h) \to$ Type, and the sequencing and assignment axioms are:

$$\begin{array}{lcl}
\mathrm{Ass}_1 & : & \Pi_{\substack{t:l\to l\to i \\ \Phi:i\to i\to o}} \vdash \lambda x : l \ . \ \lambda y : l \ . \ \{\Phi(t(x,y),y!)\}x := t(x,y)\{\Phi(x!,y!)\} \\
\mathrm{Ass}_2 & : & \Pi_{\substack{t:l\to l\to i \\ \Phi:i\to i\to o}} \vdash \lambda y : l \ . \ \lambda x : l \ . \ \{\Phi(t(x,y),y!)\}x := t(x,y)\{\Phi(x!,y!)\} \\
\mathrm{Seq} & : & \Pi_{\substack{\Phi_0,\Phi_1,\Phi_2:l\to l\to o \\ w_1,w_2:l\to l\to w}} \\
 & & \quad (\vdash \lambda x : l \ . \ \lambda y : l.\{\Phi_0(x,y)\}w_1(x,y)\{\Phi_1(x,y)\}) \to \\
 & & \quad\quad (\vdash \lambda x : l \ . \ \lambda y : l.\{\Phi_1(x,y)\}w_2(x,y)\{\Phi_2(x,y)\}) \to \\
 & & \quad\quad\quad (\vdash \lambda x : l \ . \ \lambda y. : l.\{\Phi_0(x,y)\}w_1(x,y); w_2(x,y)\{\Phi_2(x,y)\})
\end{array}$$

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.

References

[1] Arnon Avron, Furio Honsell and Ian A. Mason. *Using Typed Lambda Calculus to Implement Formal Systems on a Machine.* Technical Report, Laboratory for the Foundations of Computer Science, Edinburgh University, 1987. ECS-LFCS-87-31.

[2] Arnon Avron. *Simple Consequence Relations.* Technical Report, Laboratory for the Foundations of Computer Science, Edinburgh University, 1987. ECS-LFCS-87-30.

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

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

[5] 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.

[6] Jean-Yves Girard. *Linear Logic.* Theoretical Computer Science. volume 50, 1987, pp 1-102.

[7] Robert Harper, Furio Honsell, Gordon Plotkin. *A Framework for Defining Logics.* Proceedings of the Second Annual Conference on Logic in Computer Science, Cornell, 1987.

[8] 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.

[9] 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.

[10] 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.

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

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

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

Contents

1 Introduction 1

2 The LF Specification Language 1

3 A Simple Specification of a Logic 4

4 The LF Paradigm for Specifying a Logical System 9

5 Modal Logics 10

6 Theories of Functions 12

7 Program Logics 16