Systems of Polymorphic Type Assignment in LF

Robert Harper

June 1990
CMU-CS-90-144

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Several formulations of type assignment for the Damas-Milner language are studied, with a view toward their formalization in the logical framework LF, and the suitability of these encodings for direct execution by the logic programming language Elf.

This research was supported in part by the Office of Naval Research and in part by the Defense Advanced Research Projects Agency (DOD), monitored by the Office of Naval Research under Contract N00014-84-K-0415, ARPA Order No. 5404.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of ONR, DARPA or the U.S. government.

Keywords: Logical frameworks, type theory, functional programming.

1 Introduction

The logical framework LF is a languge for formally specifying deductive systems in a form suitable for use by generic tools that support a variety of inferential activities. [HHP87, HHP89, AHM87] For the most part, LF has been considered as a vehicle for specifying formal logical systems such as variations on propositional and predicate logic. However, LF is also intended for use in specifying programming languages, in particular operational semantics and type checkers. [BH88] (See also the work of Kahn and his co-workers and of Miller and his students for related efforts. [CDD+85, CDDK86, FM88, HM88]) Here the overall aims are similar, but the intended applications are quite different: rather than focus on machine-assisted proof, one is interested in a direct "operationalization" of specifications of programming languages, yielding, for example, a type checker and evaluator. The programming language Elf [Pfe89] provides just such a vehicle for "executing" LF specifications.

The purpose of this paper is to study the formalization of polymorphic type systems for the fragment of ML introduced by Damas and Milner [DM82]. Our purpose will be two-fold. On the one hand we are interested in the intrinsic difficulty of encoding various systems of type assignment in LF. A number of problems arise along the way, not all of which have an entirely satisfactory solution. On the other hand, we are interested in the operational behavior of our specifications. This will lead us to consider several alternatives, and to study their properties as Elf programs.

The paper is organized as follows. First, we introduce the basic calculus given by Damas and Milner, reformulated to take careful account of the use of type variables, and to make explicit the distinction between types and type schemes. This calculus admits a straightforward encoding in LF, but is unsuitable for direct implementation because of the presence of non-syntax-directed rules. Second, a "normal form" calculus is considered. This calculus may be viewed as generating the derivations in the basic calculus that are in normal form, *i.e.*, for which polymorphic instantiation is applied only to variables, and polymorphic generalization only to let-bound variables at the point at which they are added to the context. The encoding of this calculus in LF requires auxiliary judgements in order to ensure that the normal form property is preserved, but otherwise proceeds along standard lines. This formulation is syntax-directed, but is still insufficiently deterministic to be usable for type checking. We therefore consider a third, "algorithmic" formulation of the calculus, similar to that used in Centaur. This version is a restriction of the normal form calculus that ensures that when a let-bound variable is added to the context, the most general type scheme possible is assigned to the variable.

2 The Damas-Milner Calculus

2.1 Syntax

Let $\mathcal{TV}$ be a denumerable set of *type variables*, ranged over by s and t. Let $\mathcal{BT}$ be a set of *base types*, ranged over by b. The set of *type expressions*, ranged over by τ, is the least set $\mathcal{TE}$ such that $\mathcal{TV} \subseteq \mathcal{TE}$, $\mathcal{BT} \subseteq \mathcal{TE}$, and if $\tau_1 \in \mathcal{TE}$ and $\tau_2 \in \mathcal{TE}$, then $\tau_1 \to \tau_2 \in \mathcal{TE}$. The set of *type schemes*, ranged over by σ, is the least set $\mathcal{TS}$ containing $\Uparrow\tau$ for each $\tau \in \mathcal{TE}$

and $\forall t.\sigma$ for each $t \in \mathcal{TV}$ and $\sigma \in \mathcal{TS}$. In expressions of the form $\forall t.\sigma$, the type variable t is bound in σ; type schemes differing only in the names of bound variables are identified. Free variables and capture-avoiding substitution are defined as usual. The set of free type variables in a type τ or type scheme σ is denoted by $FTV(\tau)$ or $FTV(\sigma)$, respectively.

Let $T \subseteq \mathcal{TV}$ be a finite set of type variables. We write $T \vdash \tau$ to mean that $FTV(\tau) \subseteq T$. Similarly, we write $T \vdash \sigma$ to mean that $FTV(\sigma) \subseteq T$.

Let $\mathcal{V}$ be a denumerable set of *ordinary variables*, ranged over by x and y. Let $\mathcal{C}$ be a set of *ordinary constants*, ranged over by c. The set of *ordinary expressions*, ranged over by e, is the least set $\mathcal{E}$ such that $\mathcal{V} \subseteq \mathcal{E}$, $\mathcal{C} \subseteq \mathcal{E}$, and if $e_1 \in \mathcal{E}$ and $e_2 \in \mathcal{E}$, then so are $e_1\, e_2$, $\lambda x.e_1$, and *let* $x{=}e_1$ *in* e_2. In expressions $\lambda x.e$, the variable x is bound in e; in expressions *let* $x{=}e_1$ *in* e_2, the variable x is bound in e_2. Free variables are defined as usual; expressions differing only in the names of bound variables are identified.

Let $X \subseteq \mathcal{V}$ be a finite set of ordinary variables. We write $X \vdash e$ to indicate that e is an expression such that $FV(e) \subseteq X$.

2.2 Sequent-style Presentation

The standard formulation of a type assignment system is in terms of *typing sequents* consisting of an environment, an expression, and a type. In this subsection we give a sequent-style formulation of the Damas-Milner type assignment system. The presentation differs from that of Damas and Milner in that we include a set of type variables as part of the typing assertion.

A *type environment* is a partial function $E : \mathcal{V} \rightharpoonup \mathcal{TS}$ with finite domain. If E is a type environment and $x \notin \mathrm{dom}(E)$, then $E, x{:}\sigma$ denotes the extension of E by the given binding. Let $FTV(E)$ denote the set $\bigcup_{x \in \mathbf{dom}(E)} FTV(E(x))$.

We write $T \vdash E$ to mean that E is a type environment such that for each $x \in \mathrm{dom}(E)$, $T \vdash E(x)$.

A *typing sequent* is a four-tuple $T; E \gg e : \sigma$ where $T \subseteq \mathcal{TV}$, E is a type environment such that $T \vdash E$, e is an expression such that $\mathrm{dom}(E) \vdash e$, and σ is a type scheme such that $T \vdash \sigma$.

The following rules define the derivability relation for typing sequents:

$$\frac{T; E \gg e : \forall t.\sigma}{T; E \gg e : [\tau/t]\sigma} \quad (T \vdash \tau) \qquad \text{(S-INST)}$$

$$\frac{T, t; E \gg e : \sigma}{T; E \gg e : \forall t.\sigma} \quad (t \notin T) \qquad \text{(S-GEN)}$$

$$\frac{}{T; E \gg x : \sigma} \quad (E(x) = \sigma) \qquad \text{(S-VAR)}$$

$$\frac{T; E, x{:}\Uparrow \tau_1 \gg e : \Uparrow \tau_2}{T; E \gg \lambda x.e : \Uparrow(\tau_1 \rightarrow \tau_2)} \quad (x \notin \mathrm{dom}(E)) \qquad \text{(S-ABS)}$$

$$\frac{T; E \gg e_1 : \Uparrow(\tau_1 \rightarrow \tau_2) \qquad T; E \gg e_2 : \Uparrow \tau_1}{T; E \gg e_1\, e_2 : \Uparrow \tau_2} \qquad \text{(S-APP)}$$

$$\frac{T; E \gg e_1 : \sigma_1 \qquad T; E, x{:}\sigma_1 \gg e_2 : \sigma_2}{T; E \gg \mathit{let}\ x{=}e_1\ \mathit{in}\ e_2 : \sigma_2} \quad (x \notin \mathrm{dom}(E)) \qquad \text{(S-LET)}$$

The rules S-ABS and S-APP have a somewhat peculiar form due to the explicit inclusion of types into type schemes. A more familiar-looking formulation may be obtained by considering two forms of typing assertion, $T; E \gg e : \sigma$ as before, and $T; E \gg e : \tau$, where τ is a type. This variant is formalized as follows:

$$\frac{T; E \gg e : \forall t.\sigma}{T; E \gg e : [\tau/t]\sigma} \quad (T \vdash \tau) \qquad \text{(S-INST'-1)}$$

$$\frac{T; E \gg e : \Uparrow \tau}{T; E \gg e : \tau} \qquad \text{(S-INST'-2)}$$

$$\frac{T; E \gg e : \tau}{T; E \gg e : \Uparrow \tau} \qquad \text{(S-GEN'-1)}$$

$$\frac{T; E \gg e : \sigma}{T; E \gg e : \forall t.\sigma} \quad (t \notin T) \qquad \text{(S-GEN')}$$

$$\frac{}{T; E \gg x : \sigma} \quad (E(x) = \sigma) \qquad \text{(S-VAR')}$$

$$\frac{T; E, x{:}\Uparrow \tau_1 \gg e : \tau_2}{T; E \gg \lambda x.e : \tau_1 \rightarrow \tau_2} \quad (x \notin \mathrm{dom}(E)) \qquad \text{(S-ABS')}$$

$$\frac{T; E \gg e_1 : \tau_1 \rightarrow \tau_2 \qquad T; E \gg e_2 : \tau_1}{T; E \gg e_1\, e_2 : \tau_2} \qquad \text{(S-APP')}$$

$$\frac{T; E \gg e_1 : \sigma_1 \qquad T; E, x{:}\sigma_1 \gg e_2 : \sigma_2}{T; E \gg \mathit{let}\ x{=}e_1\ \mathit{in}\ e_2 : \sigma_2} \quad (x \notin \mathrm{dom}(E)) \qquad \text{(S-LET')}$$

These two systems are, in an obvious sense, equivalent. We will regard the former as the "official" definition, and write $\mathrm{DM} \vdash T; E \gg e : \sigma$ to indicate that the typing sequent $T; E \gg e : \sigma$ is derivable in accordance with those rules.

2.3 Natural deduction-style Presentation

The Damas-Milner calculus admits a natural deduction-style formulation in which the basic judgement form is the *typing assertion* $e \in \sigma$. The usual notational conventions for discharge of hypotheses are adopted here. (See [Mar82] for a related system of natural deduction.)

The natural deduction-style rules are as follows:

$$\frac{e \in \forall t.\sigma}{e \in [\tau/t]\sigma} \qquad \text{(N-INST)}$$

$$\frac{e \in \sigma}{e \in \forall t.\sigma} \quad (\dagger) \qquad \text{(N-GEN)}$$

$$\frac{}{x \in \sigma} \qquad \text{(N-VAR)}$$

$$\frac{\begin{array}{c}(x \in \Uparrow \tau_1)\\ e \in \Uparrow \tau_2\end{array}}{\lambda x.e \in \Uparrow(\tau_1 \rightarrow \tau_2)} \quad (\ddagger) \qquad \text{(N-ABS)}$$

$$\frac{e_1 \in \Uparrow(\tau_1 \rightarrow \tau_2) \qquad e_2 \in \Uparrow \tau_1}{e_1\, e_2 \in \Uparrow \tau_2} \qquad \text{(N-APP)}$$

$$\frac{e_1 \in \sigma_1 \qquad \begin{array}{c}(x \in \sigma_1)\\ e_2 \in \sigma_2\end{array}}{\mathit{let}\ x{=}e_1\ \mathit{in}\ e_2 \in \sigma_2} \quad (\ddagger) \qquad \text{(N-LET)}$$

where the side conditions indicated above are as follows:

† The type variable t does not occur free in any typing hypothesis on which the derivation of the premise depends.

‡ The variable x does not occur free in any typing hypothesis, other than those discharged by the rule instance, on which the derivation of the premise depends.

Just as with the sequent-based formulation, there is a more natural-looking variant based on two forms of typing judgement, $e \in \sigma$ and $e \in \tau$, related by the rules

$$\frac{e \in \Uparrow \tau}{e \in \tau} \qquad \text{(N-INST')}$$

$$\frac{e \in \tau}{e \in \Uparrow \tau} \qquad \text{(N-GEN')}$$

The uses of $\Uparrow$ in rules N-ABS and N-APP are then eliminated.

We write $T; E \vdash_{\mathrm{DM}} e \in \sigma$ to indicate that $e \in \sigma$ is derivable in the above system from hypotheses E involving only type variables in T. It is worth noting that this definition is

formulated so as to rule out derivations in which a variable is governed by more than one typing hypothesis. Other derivations do arise, but we reject them as "malformed" and do not consider them any further. With this in mind, it is a simple matter to establish the equivalence of the sequent- and natural deduction-style presentations.

Proposition 2.1 $\mathrm{DM} \vdash T; E \gg e : \sigma$ *iff* $T; E \vdash_{\mathrm{DM}} e \in \sigma$.

2.4 Formalization in LF

The Damas-Milner calculus, as presented above, is readily formalizable in LF. The signature, Σ_{DM}, of this system is given as follows:

$$
\begin{array}{rcl}
ty & : & \mathsf{Type} \\
sty & : & \mathsf{Type} \\
tm & : & \mathsf{Type} \\
b & : & ty \quad (b \in \mathcal{BT}) \\
\rightarrow & : & ty \rightarrow ty \rightarrow ty \\
\Uparrow & : & ty \rightarrow sty \\
\forall & : & (ty \rightarrow sty) \rightarrow sty \\
\lambda & : & (tm \rightarrow tm) \rightarrow tm \\
\cdot & : & tm \rightarrow tm \rightarrow tm \\
\mathrm{LET} & : & tm \rightarrow (tm \rightarrow tm) \rightarrow tm \\
\in & : & tm \rightarrow sty \rightarrow \mathsf{Type} \\
\mathrm{INST} & : & \Pi e{:}tm.\Pi\phi{:}ty \rightarrow sty.\Pi\tau{:}ty.e \in \forall(\phi) \rightarrow e \in \phi(\tau) \\
\mathrm{GEN} & : & \Pi e{:}tm.\Pi\phi{:}ty \rightarrow sty.(\Pi t{:}ty.e \in \phi(t)) \rightarrow e \in \forall(\phi) \\
\mathrm{ABS} & : & \Pi f{:}tm \rightarrow tm.\Pi\tau_1{:}ty.\Pi\tau_2{:}ty.(\Pi x{:}tm.x \in \Uparrow\tau_1 \rightarrow fx \in \Uparrow\tau_2) \rightarrow \\
 & & \lambda(f) \in \Uparrow(\tau_1 \rightarrow \tau_2) \\
\mathrm{APP} & : & \Pi e_1{:}tm.\Pi e_2{:}tm.\Pi\tau_1{:}ty.\Pi\tau_2{:}ty.e_1 \in \Uparrow(\tau_1 \rightarrow \tau_2) \rightarrow e_2 \in \Uparrow\tau_1 \rightarrow \\
 & & e_1 \cdot e_2 \in \Uparrow\tau_2 \\
\mathrm{LET} & : & \Pi e_1{:}tm.\Pi f_2{:}tm \rightarrow tm.\Pi\sigma_1{:}sty.\Pi\sigma_2{:}sty.e_1 \in \sigma_1 \rightarrow \\
 & & (\Pi x{:}tm.x \in \sigma_1 \rightarrow f_2(x) \in \sigma_2) \rightarrow \mathrm{LET}(e_1, f_2) \in \sigma_2
\end{array}
$$

The adequacy of this signature is stated as follows:

Proposition 2.2 *Let* Σ_{DM} *be given as above.*

1. *For every finite set* T *of type variables, there is a compositional bijection* $(-)^*$ *between types* τ *(resp., type schemes* σ*) such that* $T \vdash \tau$ *(resp.,* $T \vdash \sigma$*) and canonical LF terms of type* ty *(resp.,* sty*) in context* $\Gamma(T)$*. Here* $\Gamma(T)$ *is the context consisting of one declaration* $t : ty$ *for each* $t \in T$*, written in some standard order.*

2. *For every finite set* X *of ordinary variables, there is a compositional bijection* $(-)^*$ *between terms* e *such that* $X \vdash e$ *and canonical LF terms of type* tm *in context* $\Gamma(X)$*. The context* $\Gamma(X)$ *is defined similarly to* $\Gamma(T)$*.*

3. *There is a compositional bijection between derivations of* $T; E \vdash e \in \sigma$ *and canonical LF terms of type* $e^* \in \sigma^*$ *in context* $\Gamma(T), \Gamma(E)$*, where* $\Gamma(E)$ *is the context consisting*

of the pairs of declarations $x : tm, x' : \tau^*$ *where* $\Gamma(x) = \tau$ *and* x' *is distinct from any ordinary variable.*

From an operational point of view, this signature is undesirable for direct execution in Elf since the presence of the INST and GEN rules leads to undirected search. However, it is clear that these rules need be used only in rather restricted contexts. To make this precise, we turn to the question of normalization of derivations in the Damas-Milner calculus.

2.5 Proof Normalization

A normal form derivation in the natural deduction calculus are those for which no occurrence of N-GEN is immediately followed by an occurrence of N-INST. When this is the case we may collapse the adjacent occurrences, replacing the derivation

$$\dfrac{\dfrac{\begin{array}{c}\delta\\ \vdots\\ e \in \sigma\end{array}}{e \in \forall t.\sigma}}{e \in [\tau/t]\sigma}$$

from premises $T, t; E$, where $t \notin T$, by the derivation

$$\begin{array}{c}[\tau/t]\delta\\ \vdots\\ e \in [\tau/t]\sigma\end{array}$$

from premises $T; E$. (This is essentially just type-β-reduction of derivations.)

Taking this as the primitive notion of reduction on derivations, we may obtain the following Prawitz-style result:

Proposition 2.3 *Every derivation has a unique normal form.*

Normal forms can be further refined by considering "commuting conversions" whereby rules N-INST and N-GEN commute with rule N-LET. By orienting these commutation conditions to force occurrences of N-INST and N-GEN "upward", we obtain an extended notion of reduction for which we conjecture the existence of unique normal forms.

One reason for interest in normalization properties of derivations is that they provide useful information for a type checking algorithm: such an algorithm may, without loss of generality, attempt to construct only normal form derivations. It is easy to see that normal form derivations (in the second, extended, sense) have the following important structural properties:

1. The premise of any occurrence of rule N-INST is either the conclusion of an N-VAR occurrence or of another N-INST occurrence.

2. The conclusion of any occurrence of rule N-GEN is either the last step of the derivation, or the premise of another N-GEN occurrence, or the first premise of an N-LET occurrence.

Thus all uses of N-INST and N-GEN occur in "maximal segments." For N-INST, all such segments begin with N-VAR and end with the application of some other rule, and for N-GEN, all such segments begin with some other rule and end with an occurrence of N-LET (or terminate the derivation.) By considering an n-ary form of quantification, all such segments may be collapsed into a single use of the appropriate rule. (We will not pursue this refinement any further.)

3 The Normal-Form Calculus

As remarked above, normal-form derivations in a type assignment calculus are particularly relevant for type checking. It is therefore interesting to consider a direct presentation of the normal form derivations, and to assess the completeness of an algorithm against this formulation. The purpose of this section is to explore several possible formulations of a normal form calculus, and to consider their formalization in LF.

3.1 Sequent-style Presentation

There are two sequent-style presentations. The simplest formulation makes use of one non-trivial meta-level operation (polymorphic instantiation), and infinitely-many rule schemes for LET. Alternatively, we may achieve a finitary formulation that makes use of two auxiliary judgements.

The first sequent-style presentation uses typing sequents of the form $T; E \gg e : \tau$, where T and E are as before, and the right-hand side is a type, rather than a type scheme. The typing rules make use of the relation $T \vdash \sigma \geq \tau$ as defined by Damas and Milner, with the restriction that the free type variables in σ and τ are limited to those occurring in T. The derivation rules are as follows:

$$\frac{}{T; E \gg x : \tau} \quad (T \vdash \sigma \geq \tau, E(x) = \sigma) \qquad \text{(S-VAR-NF)}$$

$$\frac{T; E, x{:}\Uparrow \tau_1 \gg e : \tau_2}{T; E \gg \lambda x.e : \tau_1 \rightarrow \tau_2} \quad (x \notin \mathrm{dom}(E)) \qquad \text{(S-ABS-NF)}$$

$$\frac{T; E \gg e_1 : \tau_1 \rightarrow \tau_2 \qquad T; E \gg e_2 : \tau_1}{T; E \gg e_1\, e_2 : \tau_2} \qquad \text{(S-APP-NF)}$$

$$\frac{T, t_1, \ldots, t_n; E \gg e_1 : \tau_1 \qquad T; E, x{:}\forall t_1 \ldots t_n.\tau_1 \gg e_2 : \tau_2}{T; E \gg \mathit{let}\ x{=}e_1\ \mathit{in}\ e_2 : \tau_2} \quad (t_1, \ldots, t_n \notin T,\ x \notin \mathrm{dom}(E))$$

(S-LET-NF)

The relation $T \vdash \sigma \geq \tau$ is defined similarly to Damas-Milner, as follows: $T \vdash \Uparrow\tau \geq \tau$, and $T \vdash \forall t.\sigma \geq [\tau/t]\sigma$, provided that $T \vdash \tau$.

Note that there is one rule scheme S-LET-NF for each choice of $n \in \omega$, reflecting the "degree of polymorphism" of the type scheme assigned to the *let*-bound expression. Note that n is not required to be maximal: any degree of polymorphism sufficient to carry out a typing derivation is legitimate. (This is in contrast to the "algorithmic" formulation studied below in which we are required to choose the largest degree of polymorphism compatible with the typing rules.)

There is also a finitary formulation that avoids the use of the type scheme instantiation relation, and requires only one rule scheme for LET. However, it makes use of two auxiliary typing sequents, $T; E \gg_G e : \sigma$, and $T; E \gg_I e : \sigma$. The intention is that the $\gg_G$ rules govern generalization, and the $\gg_I$ rules govern instantiation of type schemes. The main typing rules are as above, with the following two rules replacing S-VAR-NF and S-LET-NF:

$$\frac{}{T; E \gg_I x : \sigma} \quad (E(x) = \sigma) \qquad \text{(S-VAR-NF')}$$

$$\frac{T; E \gg_G e_1 : \sigma_1 \qquad T; E, x{:}\sigma_1 \gg e_2 : \tau_2}{T; E \gg \mathit{let}\ x{=}e_1\ \mathit{in}\ e_2 : \tau_2} \qquad \text{(S-LET-NF')}$$

The rules governing $\gg_I$ are:

$$\frac{T; E \gg_I x : \Uparrow\tau}{T; E \gg x : \tau} \qquad \text{(S-INST-NF-1)}$$

$$\frac{T; E \gg_I x : \forall t.\sigma}{T; E \gg_I x : [\tau/t]\sigma} \quad (T \vdash \tau) \qquad \text{(S-INST-NF-2)}$$

The rules governing $\gg_G$ are:

$$\frac{T; E \gg e : \tau}{T; E \gg_G e : \Uparrow\tau} \qquad \text{(S-GEN-NF-1)}$$

$$\frac{T, t; E \gg_G e : \sigma}{T; E \gg_G \forall t.\sigma} \quad (t \notin T) \qquad \text{(S-GEN-NF-2)}$$

Proposition 3.1 *There exists a compositional bijection between normal form derivations in the DM calculus and derivations in the second "normal form" calculus.*

The details of the equivalence are tedious to write down, but amount to proving the structural properties of normal form derivations in the DM calculus discussed in the previous section.

3.2 Natural deduction-style Presentation

A natural deduction-style presentation of the infinitary sequent-style formulation makes use of two typing judgements, $e \in \tau$ and $e \in_X \sigma$. The latter judgement form is used in the encoding to record the types of variables, reflecting the fact that in the normal form calculi, variables are given polymorphic types, whereas phrases are given monotypes. The natural deduction formulation is as follows:

$$\frac{x \in_X \sigma}{x \in \tau} \quad (\sigma \geq \tau) \qquad \text{(N-INST-NF)}$$

$$\frac{\begin{array}{c}(x \in_X \Uparrow(\tau_1)) \\ e \in \tau_2\end{array}}{\lambda x.e \in \tau_1 \rightarrow \tau_2} \quad (\ddagger) \qquad \text{(N-ABS-NF)}$$

$$\frac{e_1 \in \tau_1 \rightarrow \tau_2 \qquad e_2 \in \tau_1}{e_1\, e_2 \in \tau_2} \qquad \text{(N-APP-NF)}$$

$$\frac{e_1 \in \tau_1 \qquad \begin{array}{c}(x \in_X \forall t_1 \ldots t_n.\tau_1) \\ e_2 \in \tau_2\end{array}}{let\ x{=}e_1\ in\ e_2 \in \tau_2} \quad (\dagger\dagger) \qquad \text{(N-LET-NF)}$$

where the ($\ddagger$) side condition is as before, and the ($\dagger\dagger$) condition requires as well that $t_1, \ldots, t_n$ be a sequence of type variables not occurring free in any typing hypothesis on which the derivation of the first premise depends.

The finitary formulation also has a natural deduction analogue, employing three forms of typing assertion, $e \in \tau$, $e \in_I \sigma$, and $e \in_G \sigma$, corresponding to each of the three forms of typing sequent. The rules are as follows:

$$\frac{}{x \in_I \sigma} \qquad \text{(N-VAR-NF')}$$

$$\frac{\begin{array}{c}(x \in_I \Uparrow(\tau_1)) \\ e \in \tau_2\end{array}}{\lambda x.e \in \tau_1 \rightarrow \tau_2} \quad (\ddagger) \qquad \text{(N-ABS-NF')}$$

$$\frac{e_1 \in \tau_1 \rightarrow \tau_2 \qquad e_2 \in \tau_1}{e_1\, e_2 \in \tau_2} \qquad \text{(N-APP-NF')}$$

$$\frac{e_1 \in_G \sigma_1 \qquad \begin{array}{c}(x \in_I \sigma_1) \\ e_2 \in \tau_2\end{array}}{let\ x{=}e_1\ in\ e_2 \in \tau_2} \quad (\ddagger) \qquad \text{(N-LET-NF')}$$

where the ($\ddagger$) side condition is as before.

The rules governing $\in_I$ are:

$$\frac{x \in_I \Uparrow(\tau)}{x \in \tau} \qquad \text{(N-INST-NF-1)}$$

$$\frac{x \in_I \forall t.\sigma}{x \in_I [\tau/t]\sigma} \qquad \text{(N-INST-NF-2)}$$

The rules governing $\in_G$ are:

$$\frac{e \in \tau}{e \in_G \Uparrow \tau} \qquad \text{(N-GEN-NF-1)}$$

$$\frac{e \in_G \sigma}{e \in_G \forall t.\sigma} \quad (\dagger) \qquad \text{(N-GEN-NF-2)}$$

where the condition $(\dagger)$ is as before.

A correspondence between derivations in these calculi and normal-form derivations in the basic natural deduction calculus of the previous section may also be obtained by proceeding along the same lines as for the sequent presentations. The details are omitted here.

3.3 Formalization in LF

The infinitary natural deduction formulation may be encoded in LF using relatively standard methods, provided that we are willing to admit a formalization that is not uniform in the choice of parameter $n \in \omega$ to the *let* rule. In other words, we present an *infinitary* signature Σ_{NFDM1} that contains one *let* rule for each choice of parameter n. Although the pattern is entirely clear, the formulation is not uniform. To achieve a completely uniform encoding appears to require methods similar to those used in Mason's encoding of Hoare logic. [Mas87, AHM87]

The signature Σ_{NFDM1} is defined as follows. The syntax part is the same as for Σ_{DM}, and is omitted. The remainder is as follows:

$$\begin{array}{rcl}
\in & : & tm \to ty \to \mathsf{Type} \\
\in_X & : & tm \to sty \to \mathsf{Type} \\
\geq & : & sty \to ty \to \mathsf{Type} \\
\text{INST1} & : & \Pi\tau : ty.\Pi\phi : ty \to sty.\forall(\phi) \geq \phi(\tau) \\
\text{INST2} & : & \Pi\tau : ty.\Uparrow\tau \geq \tau \\
\text{INSTNF} & : & \Pi e : tm.\Pi\sigma : sty.\Pi\tau : ty.(e \in_X \sigma) \to (\sigma \geq \tau) \to (e \in \tau) \\
\text{APPNF} & : & \Pi e_1 : tm.\Pi e_2 : tm.\Pi\tau_1 : ty.\Pi\tau_2 : ty. \\
 & & (e_1 \in \tau_1 \to \tau_2) \to (e_2 \in \tau_1) \to (e_1 \cdot e_2 \in \tau_2) \\
\text{ABSNF} & : & \Pi f : tm \to tm.\Pi\tau_1 : ty.\Pi\tau_2 : ty. \\
 & & (\Pi x : tm.(x \in_X \tau_1) \to (fx \in \tau_2)) \to (\lambda(f) \in \tau_1 \to \tau_2) \\
\text{LETNF1} & : & \Pi e_1 : tm.\Pi f_2 : tm \to tm.\Pi\phi : ty \to sty.\Pi\tau_2 : ty. \\
 & & (\Pi t : ty.e_1 \in \phi(t)) \to (\Pi x : tm.(x \in_X \forall(\phi)) \to (f_2(x) \in \tau_2)) \to \\
 & & \text{LET}(e_1, f_2) \in \tau_2 \\
\text{LETNF2} & : & \Pi e_1 : tm.\Pi f_2 : tm \to tm.\Pi\phi : ty \to ty \to sty.\Pi\tau_2 : ty. \\
 & & (\Pi t_1 : ty.\Pi t_2 : ty.e_1 \in \phi(t_1)(t_2)) \to \\
 & & (\Pi x : tm.(x \in_X \forall(\lambda t_1 : ty.\forall(\lambda t_2 : ty.\phi(t_1)(t_2)))) \to (f_2(x) \in \tau_2)) \to \\
 & & \text{LET}(e_1, f_2) \in \tau_2 \\
 & \vdots &
\end{array}$$

The adequacy theorem is similar to that of the encoding of the basic system, except that it is stated once for each choice of parameter $n \in \omega$. That is, we establish a compositional bijection between derivations in the infinitary calculus with maximum "degree of polymorphism" n and canonical LF terms of appropriate type in the prefix of the above signature including *let* rules up to degree n.

The finitary natural deduction formulation of the normal form calculus may be directly transcribed into LF by proceeding along standard lines, as follows:

$$
\begin{array}{rcl}
\in & : & tm \to ty \to \mathsf{Type} \\
\in_I & : & tm \to sty \to \mathsf{Type} \\
\in_G & : & tm \to sty \to \mathsf{Type} \\
\textsc{absnf} & : & \Pi\tau_1 : ty.\Pi\tau_2 : ty.\Pi f : tm \to tm. \\
 & & (\Pi x : tm.x \in_I \Uparrow \tau_1 \to fx \in \tau_2) \to \lambda(f) \in \tau_1 \to \tau_2 \\
\textsc{appnf} & : & \Pi e_1 : tm.\Pi e_2 : tm.\Pi\tau_1 : ty.\Pi\tau_2 : ty. \\
 & & (e_1 \in \tau_1 \to \tau_2) \to (e_2 \in \tau_1) \to (e_1 \cdot e_2 \in \tau_2) \\
\textsc{letnf} & : & \Pi e_1 : tm.\Pi f_2 : tm \to tm.\Pi\sigma_1 : sty.\Pi\tau_2 : ty. \\
 & & e_1 \in_G \sigma_1 \to (\Pi x : tm.(x \in_I \sigma_1 \to (f_2(x) \in \tau_2)) \to \\
 & & \textsc{let}(e_1, f_2) \in \tau_2 \\
\textsc{instnf1} & : & \Pi e : tm.\Pi\tau : ty.e \in_I \Uparrow \tau \to e \in \tau \\
\textsc{instnf2} & : & \Pi e : tm.\Pi\tau : ty.\Pi\phi : ty \to sty.e \in_I \forall(\phi) \to e \in_I \phi(\tau) \\
\textsc{gennf1} & : & \Pi e : tm.\Pi\tau : ty.e \in \tau \to e \in_G \Uparrow \tau \\
\textsc{gennf2} & : & \Pi e : tm.\Pi\phi : ty \to sty.(\Pi t : ty.e \in_G \phi(t)) \to e \in_G \forall(\phi)
\end{array}
$$

The adequacy of this signature may be stated as follows.

Proposition 3.2 *For any context Γ representing a typing context $T; E$, there is a compositional bijection between proofs of $e \in \tau$ (resp., $e \in_I \sigma$, $e \in_G \sigma$) in the finitary, natural deduction presentation of the normal form calculus and canonical LF terms of type $e^* \in \tau^*$ (resp., $e^* \in_I \sigma^*$, $e^* \in_G \sigma^*$) in context Γ.*

The use of multiple judgements in this encoding ensures that derivations have a restricted form determined by the syntax of the expression. However, it still suffers from the deficiency that the rules do not constrain the "degree of polymorphism" to choose for a *let*-bound expression. In particular, it is legitimate to choose an excessively precise type for the identity function, and this can lead to failure of type checking. From an operational point of view, this means that the Elf interpreter would be forced to backtrack until all choices are exhausted. This leads us to consider a more restricted version of the normal form calculus that ensures that principal types are chosen for *let*-bound expressions.

4 The Algorithmic Calculus

One application of normal-form calculi is in the proof of syntactic completeness of a type checking algorithm: since the algorithm constructs only normal form derivations, it ought to be relatively straightforward to prove that it is complete with respect to the normal form calculi. However, this is not so since the algorithm constructs derivations of an even more

restricted form than the normal forms of the last section. For example, in the *let* rule there is no requirement that the type of the *let*-bound expression be taken as general as possible, whereas the algorithm will clearly do so. Thus, there is not a complete correspondence between the normal form calculus and the derivations built by the algorithm. To achieve this correspondence we consider a further refinement of the normal form calculus, called the *algorithmic* calculus. Such algorithmic systems are employed in the definition of Standard ML [MTH90]. They have the advantage that they specify quite precisely the compile-time elaboration without getting involved in the details of a type checking algorithm.

4.1 Sequent-style Presentation

There are two sequent-style presentations, corresponding to the infinitary and finitary variants of the normal form sequent systems.

The first is obtained by replacing the rule S-LET-NF by the following rule:

$$\frac{T, t_1, \ldots, t_n; E \gg e_1 : \tau_1 \qquad T; E, x{:}\forall t_1 \ldots t_n.\tau_1 \gg e_2 : \tau_2}{T; E \gg \mathit{let}\ x{=}e_1\ \mathit{in}\ e_2 : \tau_2} \quad (\ddagger\ddagger) \qquad \text{(S-LET-A)}$$

where the side condition $(\ddagger\ddagger)$ is

$$t_1, \ldots, t_n \notin T,\ x \notin \mathrm{dom}(E),\ T \subseteq FTV(E)$$

The condition $T \subseteq FTV(E)$ ensures that the universal closure is maximal, since we know in general that $FTV(E) \subseteq T$, and hence no type variable in T is dischargeable.

It is easy to see that this rule is essentially equivalent to the more familiar

$$\frac{T; E \gg e_1 : \tau_1 \qquad T; E, x{:}\, Cl_{T;E}(\tau_1) \gg e_2 : \tau_2}{T; E \gg \mathit{let}\ x{=}e_1\ \mathit{in}\ e_2 : \tau_2} \quad (x \notin \mathrm{dom}(E)) \qquad \text{(S-LET-A')}$$

where $Cl_{T;E}(\tau)$ is the usual *polymorphic closure* operation yielding the most general generalization of τ_1 compatible with T and E. (The only difference is in the handling of the variable sets.)

There are two formulations based on the use of auxiliary typing judgements. The first is a more-or-less direct expression of the side condition on T used above:

$$\frac{FTV(E); E \gg_G e_1 : \sigma_1 \quad T; E, x : \sigma_1 \gg e_2 : \tau_2}{T; E \gg \mathit{let}\ x{=}e_1\ \mathit{in}\ e_2 : \tau_2} \qquad \text{(S-LET-A'''')}$$

The restriction of the available type variables in the first derivation to those occurring free in E ensures that σ_1 is most general, for otherwise σ_1 would involve a free type variable not bound in E, contradicting the property that all such type variables occur in the type variable set.

This formulation leans heavily on the sequent-style presentation of the calculus in which we regard the type set and the type environment to be an "input parameter" of the type system. This approach does not lend itself to formalization in LF, and we therefore consider an alternative approach employing a "*post-hoc*" auxiliary judgement, $T; E \gg \sigma$ *witnessed*:

$$\frac{T;E \gg_G e_1 : \sigma_1 \qquad T;E \gg \sigma_1 \ witnessed \qquad T;E,x{:}\sigma_1 \gg e_2 : \tau_2}{T;E \gg let\ x{=}e_1\ in\ e_2 : \tau_2} \qquad \text{(S-LET-A'')}$$

The intention of the second premise is to ensure that every free type variable in *sigma* is "witnessed" by some typing assumption in E (*i.e.*, occurs in the type scheme assigned to some variable by E). This judgement may be formalized as follows:[1]

$$\frac{T;E \gg_X x : \sigma}{T;E \gg \sigma \ witnessed} \qquad \text{(S-VAR-WIT)}$$

$$\frac{T;E \gg \Uparrow \tau \ witnessed}{T;E \gg \tau \ witnessed} \qquad \text{(S-LIFT-WIT)}$$

$$\frac{T;E \gg \forall t.\sigma \ witnessed \quad T;E \gg \tau \ witnessed}{T;E \gg [\tau/t]\sigma \ witnessed} \qquad \text{(S-ALL-WIT)}$$

$$\frac{T;E \gg \tau_1 \rightarrow \tau_2 \ witnessed}{T;E \gg \tau_1 \ witnessed} \qquad \text{(S-ARR-WIT-L)}$$

$$\frac{T;E \gg \tau_1 \rightarrow \tau_2 \ witnessed}{T;E \gg \tau_2 \ witnessed} \qquad \text{(S-ARR-WIT-R)}$$

$$\frac{}{T;E \gg b \ witnessed} \qquad \text{(S-WIT-B)}$$

$$\frac{T;E \gg \tau_1 \ witnessed \qquad T;E \gg \tau_2 \ witnessed}{T;E \gg \tau_1 \rightarrow \tau_2 \ witnessed} \qquad \text{(S-WIT-ARR)}$$

$$\frac{T;E \gg \tau \ witnessed}{T;E \gg \Uparrow \tau \ witnessed} \qquad \text{(S-WIT-LIFT)}$$

$$\frac{T,t;E,x{:}t \gg \sigma \ witnessed}{T;E \gg \forall t.\sigma \ witnessed} \quad (t \notin T,\ x \notin \mathrm{dom}(E)) \qquad \text{(S-WIT-ALL)}$$

In the rule S-WIT-ALL the purpose of the "spurious" variable declaration $x{:}t$ is to ensure that t is note regarded as "close-able" in the subderivation.

The rule S-VAR-WIT makes use of a sequent $T;E \gg_X x : \sigma$ which is axiomatized by the single rule

$$\frac{}{T;E \gg_X x : \sigma} \quad (E(x) = \sigma) \qquad \text{(S-LOOKUP)}$$

We may *not* replace $\gg_X$ with either $\gg_I$ or $\gg_G$ since it is imperative that σ be *precisely* the type scheme bound to the variable x by E.

[1] Strictly speaking, there are two "witnessed" judgements, one for types and the other for type schemes.

The purpose of rules S-VAR-WIT through S-ARR-WIT-R is to allow for a type expression assigned to a variable to be decomposed into component parts, which may then be reassembled using rules S-WIT-B through S-WIT-ALL. For example, if x is assigned the type scheme $\Uparrow(t \rightarrow t)$ for some type variable t, then we should be able to determine, on this basis, that t is witnessed, and that $t \rightarrow t \rightarrow t$ is witnessed as well. If we are willing to make use of a meta-level operation $FTV(\sigma)$, then we may replace rules S-VAR-WIT through S-ARR-WIT-R with the rule

$$\frac{T; E \gg_X x : \sigma}{T; E \gg t \ witnessed} \qquad (t \in FTV(\sigma) \qquad \text{(S-VAR-WIT')}$$

The more detailed axiomatization is preferable since it avoids use of such a non-trvial operation.

4.2 Encoding in LF

The algorithmic formulation based on "witnessed" judgements may be readily adapted to a natural deduction setting, and hence may be directly encoded in LF. We omit presentation of the natural deduction calculus in favor of the LF encoding. The signature, Σ_{DMALG} is a modification of the signature of the normal form calculus, and is defined as follows:

$$\begin{array}{rcl}
\in & : & tm \rightarrow ty \rightarrow \mathsf{Type} \\
\in_I & : & tm \rightarrow sty \rightarrow \mathsf{Type} \\
\in_G & : & tm \rightarrow sty \rightarrow \mathsf{Type} \\
\in_X & : & tm \rightarrow sty \rightarrow \mathsf{Type} \\
witnessed_s & : & sty \rightarrow \mathsf{Type} \\
witnessed_t & : & ty \rightarrow \mathsf{Type} \\
\text{APP} & : & \Pi e_1, e_2 : tm.\Pi \tau_1, \tau_2 : tm.e_1 \in \tau_1 \rightarrow \tau_2 \rightarrow e_2 \in \tau_2 \rightarrow e_1 \cdot e_2 \in \tau_2 \\
\text{ABS} & : & \Pi f : tm \rightarrow tm.\Pi \tau_1, \tau_2 : ty.(\Pi x : tm.x \in_X \Uparrow \tau \rightarrow fx \in \tau_2) \rightarrow \lambda(f) \in \tau_1 \rightarrow \tau_2 \\
\text{LET} & : & \Pi e : tm.\Pi f : tm \rightarrow tm.\Pi \sigma : sty.\Pi \tau : ty. \\
 & & e \in_G \sigma \rightarrow \sigma \ witnessed_s \rightarrow (\Pi x : tm.x \in_X \sigma \rightarrow fx \in \tau) \rightarrow \text{LET}(e, f) \in \tau \\
\text{LOOKUP} & : & \Pi e : tm.\Pi \sigma : sty.e \in_X \sigma \rightarrow e \in_I \sigma \\
\text{INST1} & : & \Pi e : tm.\Pi \tau : ty.e \in_I \Uparrow \tau \rightarrow e \in \tau \\
\text{INST2} & : & \Pi e : tm.\Pi \phi : ty \rightarrow sty.\Pi \tau : ty.e \in_I \forall(\phi) \in e \in_I \phi(\tau) \\
\text{GEN1} & : & \Pi e : tm.\Pi \tau : ty.e \in \tau \rightarrow e \in_G \Uparrow \tau \\
\text{GEN2} & : & \Pi e : tm.\Pi \phi : ty \rightarrow sty.(\Pi t : ty.e \in_G \phi(t)) \rightarrow e \in_G \forall(\phi) \\
\text{VAR} - \text{WIT} & : & \Pi e : tm.\Pi \sigma : sty.e \in_X \sigma \rightarrow \sigma \ witnessed_s \\
\text{LIFT} - \text{WIT} & : & \Pi \tau : ty.\Uparrow \tau \ witnessed_s \rightarrow \tau \ witnessed_t \\
\text{ALL} - \text{WIT} & : & \Pi \phi : ty \rightarrow sty.\Pi \tau : ty.\forall(\phi) \ witnessed_s \rightarrow \tau \ witnessed_t \rightarrow \phi(\tau) \ witnessed_s \\
\text{ARR} - \text{WIT} - \text{L} & : & \Pi \tau_1, \tau_2 : ty.\tau_1 \rightarrow \tau_2 \ witnessed_t \rightarrow \tau_1 \ witnessed_t \\
\text{ARR} - \text{WIT} - \text{R} & : & \Pi \tau_1, \tau_2 : ty.\tau_1 \rightarrow \tau_2 \ witnessed_t \rightarrow \tau_2 \ witnessed_t \\
\text{WIT} - \text{B} & : & b \ witnessed_t \quad (b \in \mathcal{BT}) \\
\text{WIT} - \text{ARR} & : & \Pi \tau_1, \tau_2 : ty.\tau_1 \ witnessed_t \rightarrow \tau_2 \ witnessed_t \rightarrow \tau_1 \rightarrow \tau_2 \ witnessed_t \\
\text{WIT} - \text{LIFT} & : & \Pi \tau : ty.\tau \ witnessed_t \rightarrow \Uparrow \tau \ witnessed_s \\
\text{WIT} - \text{ALL} & : & \Pi \phi : ty \rightarrow sty.(\Pi t : ty.t \ witnessed_t \rightarrow \phi(t) \ witnessed_s) \rightarrow \forall(\phi) \ witnessed_s
\end{array}$$

The statement of the adequacy of this encoding is similar to those given above. We have only to note that we consider canonical LF terms of judgement type in contexts of the form

$$t_1{:}ty, \ldots, t_n{:}ty, x_1{:}tm, x_1'{:}x_1 \in_X \sigma_1^*, \ldots, x_k{:}tm, x_k'{:}x_k \in_X \sigma_k^*$$

in the above signature. Note that in the case of the *let* rule the typing context of the first two premises is the *same*, ensuring that we have correctly captured the notion of "witnessed" expressed in the sequent formulation. In essence, the adequacy theorem is stated for a sequent formulation of LF, and hence we can achieve the degree of hypothesis control required.

It is worth noting that canonical LF terms of judgement type in the above signature and suitable context determine canonical LF terms of corresponding type in the signature of the normal form calculus and corresponding context: we simply have to "forget" the terms encoding proofs that certain type schemes are "witnessed." Thus we may regard the algorithmic calculus as a kind of "meta-calculus" for the normal form calculus, limiting derivations to those that would be constructed by a type checking algorithm.

Unfortunately, this encoding does not itself determine a reasonable type checking algorithm (in the sense of Elf), since search would be wildly undirected. For example, to type check a *let* expression by a direct operationalization of the above rules would entail "guessing" a type scheme for the *let*-bound expression, and checking both that it is a legal type scheme for the expression, and that it is fully general. This is clearly unreasonable, and may be taken as evidence that the overall approach is in doubt. For here we have a completely deterministic formulation of the calculus whose only known encoding in LF incurs an excessive overhead from the point of view of an Elf-like interpreter.

One possible solution to this difficulty is to ignore the "declarative" formulation of the algorithmic calculus in favor of a direct operational extension to Elf that allows us to control the search for a normal-form derivation in such a way that only "algorithmic" derivations may be constructed. Pfenning has considered such an extension, which may be summarized as follows. The crucial idea is to recall that Elf employs higher-order unification to construct derivations in the LF λ-calculus. Since the unifier enumerates only maximally general substitutions, we are guaranteed that if Elf constructs a derivation of $e \in \tau$, then τ is not over-committed: it will contain "logic variables" of type *ty*, corresponding to the generic type variables of ML. These logic variables may then be discharged (by repeated use of Π-introduction) to achieve an LF type with all free variables of type *ty* bound by the outermost sequence of Π's. Given this, we may then apply the rule GEN2 repeatedly to "convert" these Π's to $\forall$'s, and achieve a most general typing (in the sense of ML). Finally, we may continue typing the body of the *let* expression, taking the result of the foregoing process as a new hypothesis. This sequence of operations is expressed in Elf using a special control construct, called "*resolve*", that allows for the specification of such generalization and forward-chaining operations. Although this extension is sound (in that only correct derivations can be constructed) it appears to have no logical basis, much as the "cut" operator of Prolog has no logical correlate.

4.3 A Modal Alternative

The encoding of the algorithmic formulation of the Damas-Milner calculus is based on requiring a certain "maximality" criterion to be fulfilled, namely that the typing derivation

of a *let*-bound expression be "maximally discharged" with respect to type variables. This pattern seems to come up in several different formal systems. For example, one reading of the rule of necessitation in a Hilbert-style formulation of S4 modal logic is that the proof of the premise is required to be "maximally discharged" with respect to non-logical axioms. That is, if we have a proof of ϕ from the non-logical assumption ψ, we may *not* infer $\Box\phi$. The best that we can do is to "discharge" the assumption ψ (by an application of the deduction theorem) to achieve a pure proof of $\psi \supset \phi$, from which we may conclude $\Box(\psi \supset \phi)$. Similar maximality criteria arise in other settings as well. It therefore makes sense to consider to what extent it might be possible to enrich the LF type theory so as to admit specification of such conditions. We might also hope to provide a logical basis for the non-logical control construct described at the end of the previous section. It should be made clear at the outset, however, that the proposed extension is extremely speculative: we mention it here only as a indication of a possible future extension to LF.

As a first approximation, we consider a "modal" type constructor $\mathsf{W}_T(A)$, where T is a type. The rough idea is that $\mathsf{W}_T(A)$ is inhabited by $\mathsf{w}_T(M)$ only if A is inhabited by M and every free variable of type T in M is "witnessed" in the sense that it occurs free in the context. Using this modality, we may express the algorithmic version of the *let* rule as follows:

$$\begin{aligned} \mathit{let} \;:\; & \Pi e_1 : \mathit{tm}.\Pi f_2 : \mathit{tm}\to\mathit{tm}.\Pi\sigma_1 : \mathit{sty}.\Pi\tau_2 : \mathit{ty}. \\ & \mathsf{W}_{\mathit{ty}}(e_1 \in_G \sigma_1) \to (\Pi x : \mathit{tm}.x \in_I \sigma_1 \to fx \in \tau_2) \to \mathit{let}(e_1, f_2) \in \tau_2 \end{aligned}$$

Bearing in mind the shape of the contexts considered in the adequacy theorems above, it is easy to see that the modality correctly enforces the "maximality" condition on the typing of e_1.

Similarly, the rule of necessitation might be expressed as follows (using an extended form of W in which the subscript is allowed to be an arbitrary family of types):

$$\mathit{Nec} \;:\; \Pi\phi : o.\, \mathsf{W}_{\mathit{true}}(\mathit{true}(\phi)) \to \mathit{true}(\Box(\phi))$$

The adequacy of this ecoding is proved by considering contexts containing assumptions of the form $x : \mathit{true}(\phi)$, for non-logical axioms, and $y : \mathit{valid}(\phi)$, for logical axioms. In such a context, the modal operator correctly limits derivations to those in which all no use is made of *any* non-logical axiom. The side condition on the necessitation rule is therefore met, and the adequacy of the encoding follows.

Before pursuing the question of the status of the putative modal extension of LF, we ought to consider reasons for considering it. After all, in both the polymorphic type assignment system example and in the modal logic example, encodings into "straight" LF are known. The most obvious answer is that the whole point of LF is to try to isolate the commonalities of a variety of formal systems so that they may be implemented once and for all. Since there are two examples, perhaps there are more .... In a slightly different direction, it might be hoped that by considering such an extension to LF we might provide a logical basis for constraining the search space associated with the signature so as to ensure that the type checking algorithm is well-behaved.

Unfortunately, this does not seem to be the case, as we now explain. The side condition on the applicability of the introductory rule for the modality $\mathsf{W}_T(A)$ requires that we first

find a term of type A, and then check that it satisfy the condition that every variable on which it depends be witnessed. If implemented naïvely, this entails as much overhead as is implied by the explicit formulation of the "witnessed" judgement of the previous section. (In fact, we might hope for a reductive explanation of W in terms of pure LF by generating suitable "witnessed" judgements that must be fulfilled whenever the W modality is used.) A slightly more sophisticated implementation would proceed as follows. To find an element of type $\mathsf{W}_T(A)$ in a context Γ, find an element of A in the context $\Gamma\backslash T$, the context obtained from Γ by striking out all declarations of "unwitnessed" variables of type T.[2] Should the subgoal succeed, the proof is guaranteed to satisfy the side condition on the introductory rule for $\mathsf{W}_T(A)$. Thus, the *post-hoc* verification of the condition is avoided, but we are nonetheless left with an undirected search space since the interpreter would still have to "guess" the degree of polymorphism appropriate for typing the first premise of the *let* rule. In the end, the modal approach is no better than the encoding given in the previous section.

5 Conclusion

We have presented three main variations on the Damas-Milner type assignment system, and considered their formulation in LF. Although each admits a relatively natural encoding in LF, none of these encodings is suitable for direct execution as an Elf program. This lends credence to the belief that even in such a restricted setting, meta-programming is unavoidable. In Elf this takes the form of employing non-logical constructs to guide search. In a tactic-based implementation such as Lego [LPT89], it is necessary to write ML programs to specify both the search strategy and the algorithm for matching inference rules with terms.

Two rather different approaches are worth mentioning. One very natural approach is to avoid altogether the formalization of type assignment, and instead consider a language with explicit type information attached to programs, relying on general "argument synthesis" mechanisms [Hue86, CH88, LPT89, Ell89, Pfe89] to help overcome the excessive verbosity of explicitly-typed terms. Although it appears that one cannot completely recover the ML type checking algorithm in this way, there is empirical evidence to suggest that this approach suffices in practice. Moreover, this general strategy is much more widely applicable, since most type systems do not appear to admit simple, complete inference algorithms. Another, particularly clever, appoach was proposed by Miller and Hannan: rather than encode type schemes and polymorphic generalization and instantiation, they suggest an encoding of the typing rule for *let* that amounts to replacing *let*-bound variables with their definitions, but without incurring the cost of performing the replacement. The idea is to type check the body of a *let* expression under the assumption that the bound variable takes on whatever types the expression bound to it may take. In LF notation this is expressed as follows:

$$\begin{aligned}\textsc{letmh} \;:\; & \Pi e_1 : tm.\Pi f_2 : tm \to tm.\Pi \tau_1 : ty.\Pi \tau_2 : ty. \\ & e_1 \in \tau_1 \to (\Pi x : tm.(\Pi \tau : ty.e_1 \in \tau \to x \in \tau) \to f_2\, x \in \tau_2) \to \\ & \textsc{let}(e_1, f_2) \in \tau_2\end{aligned}$$

[2] The fact that $\Gamma\backslash T$ is well-formed is tantamount to the strengthening property of typing whose status in this system is unclear.

Operationally, this amounts to re-type-checking the expression e_1 for each occurrence of x_1, allowing a distinct type to be chosen for each occurrence. In this way polymorphism is retained, but without introducing type schemes and the associated problems. The typing premise for e_1 is included only to ensure that e_1 is well-typed, even if the *let*-bound variable is never used.

References

[AHM87] Arnon Avron, Furio Honsell, and Ian Mason. Using typed lambda calculus to implement formal systems on a machine. Technical Report ECS–LFCS–87–31, Laboratory for the Foundations of Computer Science, Edinburgh University, June 1987.

[BH88] Rod Burstall and Furio Honsell. A natural deduction treatment of operational semantics. Technical Report ECS–LFCS–88–69, Laboratory for the Foundations of Computer Science, Edinburgh University, November 1988.

[CDD+85] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, Laurent Hascoet, and Gilles Kahn. Natural semantics on the computer. Technical Report RR 416, INRIA, Sophia–Antipolis, France, June 1985.

[CDDK86] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *1986 Symposium on LISP and Functional Programming*, 1986.

[CH88] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[DM82] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Ninth ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[Ell89] Conal Elliott. Higher-order unification with dependent function types. In *Proceedings of Rewriting Techniques and Applications*, Chapel Hill, NC, April 1989. (To appear).

[FM88] Amy Felty and Dale Miller. A metalanguage for type checking and inference. Manuscript, November 1988.

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Second Symposium on Logic in Computer Science*, pages 194–204, Ithaca, New York, June 1987.

[HHP89] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. Technical Report CMU–CS–89–173, School of Computer Science, Carnegie Mellon University, January 1989. Revised and expanded version of [HHP87], submitted for publication.

[HM88] John Hannan and Dale Miller. Uses of higher-order unification for implementing program transformers. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 2*, pages 942–959, Cambridge, Massachusetts, August 1988. MIT Press.

[Hue86] Gérard Huet. Formal structures for computation and deduction. Lecture notes for a graduate course at Carnegie Mellon University, May 1986.

[LPT89] Zhaolui Luo, Robert Pollack, and Paul Taylor. How to use lego: A preliminary user's manual. Technical report, Laboratory for the Foundations of Computer Science, Edinburgh University, April 1989.

[Mar82] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175. North-Holland, 1982.

[Mas87] Ian Mason. Hoare's logic in the LF. Technical Report ECS–LFCS–87–32, lfcs, June 1987.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Pfe89] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Fourth Symposium on Logic in Computer Science*, June 1989.