# Logic Programming in the LF Logical Framework

Frank Pfenning
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890
Internet: fp@cs.cmu.edu

Revised draft for the Proceedings of the First Workshop on Logical Frameworks

February 1991

## Contents

# 1 Introduction

In [12], Harper, Honsell, and Plotkin present LF (the Logical Framework) as a general framework for the definition of logics. LF provides a uniform way of encoding a logical language, its inference rules and its proofs. In [2], Avron, Honsell, and Mason give a variety of examples for encoding logics in LF. In this paper we describe Elf, a meta-language intended for environments dealing with deductive systems represented in LF.

While this paper is intended to include a full description of the Elf core language, we only state, but do not prove here the most important theorems regarding the basic building blocks of Elf. These proofs are left to a future paper. A preliminary account of Elf can be found in [26]. The range of applications of Elf includes theorem proving and proof transformation in various logics, definition and execution of structured operational and natural semantics for programming languages, type checking and type inference, *etc.* The basic idea behind Elf is to unify logic definition (in the style of LF) with logic programming (in the style of $\lambda$Prolog, see [22, 24]). It achieves this unification by giving *types* an operational interpretation, much the same way that Prolog gives certain formulas (Horn-clauses) an operational interpretation. An alternative approach to logic programming in LF has been developed independently by Pym [28].

Here are some of the salient characteristics of our unified approach to logic definition and meta-programming. First of all, the Elf search process automatically constructs terms that can represent object-logic proofs, and thus a program need not construct them explicitly. This is in contrast to logic programming languages where executing a logic program corresponds to theorem proving in a meta-logic, but a meta-proof is never constructed or used and it is solely the programmer's responsibility to construct object-logic proofs where they are needed. Secondly, the partial correctness of many meta-programs with respect to a given logic can be expressed and proved by Elf itself (see the example in Section 5). This creates the possibility of deriving verified meta-programs through theorem proving in Elf (see Knoblock & Constable [18] or Allen *et al.* [14] for other approaches).

Elf is quite different in look and feel to the standard meta-programming methodology of writing tactics and tacticals in ML [11]. On the positive side, Elf programs tend to be more declarative and easier to understand. Often one can take what authors bill as an "algorithmic" version of an inference system and implement it in Elf with very little additional work. Moreover, it is possible to implement tactics and tacticals in Elf along the lines proposed by Felty [8]. Such tactics are also often easier to write and understand than tactics written in a functional style, since they inherit a notion of meta-variable (the *logic variable*, in logic programming terminology), a notion of unification, and nondeterminism and backtracking in a uniform way from the underlying logic programming language. The Isabelle system [25] also provides support for meta-variables and higher-order unification in tactics, but they are generally not as accessible as in Elf. On the negative side we encounter problems with efficiency when manipulating larger objects, something which we hope to address in future work with compilation techniques from logic programming adapted to this setting. Also, on occasion, it is difficult to express the operations we would like to perform as a pure logic program. For example, neither the cut operator ! familiar from logic programming nor a notion of negation-by-failure are available in Elf. For some initial ideas to address these deficiencies see Section 6.

We conclude this introduction with an overview of the remainder of this paper. After a brief review of the LF Logical Framework, we begin with an exposition of unification as used in Elf. The general unification problem for the LF type theory is undecidable and non-deterministic, which immediately calls into question the whole enterprise of designing a logic programming language based on LF, given that unification is such a central operation. However, inspired by Miller's

work on $L_\lambda$ [21], we design an algorithm which solves the "easy" unification problems (without branching, for example) and postpones all other equalities which may arise as *constraints*. Since dependent types and explicit Π-quantification further complicate unification, we found it necessary to develop a new view of unification as theorem proving in a very simple logic (the *unification logic*). A unification algorithm is then described as a set of transformations on formulas in that logic, reminiscent of the description of first-order unification by transformations on a set of equations. In Section 3 we develop this view by first presenting the main ideas for an untyped, first-order language and then generalizing it to the LF type theory.

Besides unification, execution of a logic program also requires back-chaining search. In Elf, this search takes the form of finding a term of a given type (possibly containing logic variables) over a given signature. This necessitates a form of resolution, which we express through an extension of the unification logic by a notion of *immediate implication*. When restricted to ordinary first-order logic programming, we would say that a clause $H \leftarrow B_1, \ldots, B_n$ immediately implies an atomic goal $G$, if $G$ unifies with $H$ under substitution $\theta$ and the remaining subgoals $(\theta B_i)$ are all provable. A formalization of this concept in this more general setting can be found in Section 4.

Back-chaining search and unification describe a non-deterministic interpreter. In order to make this useful as a programming language, search control must be added. This takes two forms. First we distinguish those types which are subject to search (and thus play the role of goals) from those types whose elements are subject only to unification (and thus play the role of ordinary logic variables). We call these *closed* and *open* type families, respectively. Second we make a commitment to depth-first search. These are described in Sections 4.4 and 4.5 which conclude the (partly informal) definition of the operational semantics of Elf.

In Section 5 we then introduce the concrete syntax for Elf and present examples which illustrate some of its unique features and common patterns of usage. The main example is an implementation of a decision procedure for term equality in the simply-typed $\lambda$-calculus. We also describe some aspects of the implementation Elf in this Section. In particular, we sketch our method of type reconstruction, since, we believe, it has independent interest.

We conclude the paper with speculation about future work.

## 2 The LF Logical Framework

We review here only the basic definitions and properties of the LF Logical Framework. For more details, the reader is referred to [12]. A number of examples of representations of logical systems in LF can be found in [2].

The LF calculus is a three-level calculus for *objects*, *families*, and *kinds*. Families are classified by kinds, and objects are classified by *types*, that is, families of kind Type.

$$\begin{array}{llll} \textit{Kinds} & K & ::= & \text{Type} \mid \Pi x{:}A.K \\ \textit{Families} & A & ::= & a \mid \Pi x{:}A.B \mid \lambda x{:}A.B \mid A\,M \\ \textit{Objects} & M & ::= & c \mid x \mid \lambda x{:}A.M \mid M\,N \end{array}$$

We use $K$ to range over kinds, $A, B$ to range over families, $M, N$ to range over objects. $a$ stands for constants at the level of families, and $c$ for constants at the level of objects. In order to describe the basic judgments we consider contexts (assigning types to variables) and signatures (assigning kinds and types to constants at the level of families and objects, respectively).

$$\begin{array}{llll} \textit{Signatures} & \Sigma & ::= & \langle\,\rangle \mid \Sigma, a{:}K \mid \Sigma, c{:}A \\ \textit{Contexts} & \Gamma & ::= & \langle\,\rangle \mid \Gamma, x{:}A \end{array}$$

Since families (and thus types) may be indexed by objects, it is important that signatures and contexts be ordered lists, rather than sets. We stipulate that constants can appear only once in signatures and variables only once in contexts. This can always be achieved through renaming. $[M/x]N$ is our notation for the result of substituting $M$ for $x$ in $N$, renaming variables as necessary to avoid name clashes. We also use the customary abbreviation $A \to B$ and sometimes $B \leftarrow A$ for $\Pi x{:}A.B$ when $x$ does not appear free in $B$.

The notion of definitional equality we consider here is $\beta\eta$-conversion. Harper *et al.* [12] formulate definitional equality only with $\beta$-conversion and conjecture that the system resulting from adding the $\eta$-rule would have the properties we list below. This has recently been proved by Coquand [3] and independently by Salvesen [30]. For practical purposes the formulation including the $\eta$-rule is superior, since every term has an equivalent canonical form. Thus, for us, $\equiv$ is the least congruence generated by $\beta\eta$-conversions in the usual manner. The basic judgments are $\Gamma \vdash_\Sigma M : A$ and $M \equiv N$ and analogous judgments at the levels of families and kinds. We assume that a well-formed signature $\Sigma$ is given, but omit the signature subscript of the various judgments in our presentation. As examples, we show the rules for abstraction, application, and type-conversion at the level of objects.

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B} \qquad \frac{\Gamma \vdash M : \Pi x{:}A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\,N : [N/x]B}$$

$$\frac{\Gamma \vdash M : A \qquad A \equiv A' \qquad \Gamma \vdash A' : \mathrm{Type}}{\Gamma \vdash M : A'}$$

We state a selection of the crucial properties of the LF type theory as given and proven in [12] and [3].

1. (Unicity of Types) If $\Gamma \vdash M : A$ and $\Gamma \vdash M : A'$ then $A \equiv A'$.
2. (Strong Normalization) If $\Gamma \vdash M : A$ then $M$ is strongly normalizing.
3. (Canonical Forms for Types) If $\Gamma \vdash A : \mathrm{Type}$ then $A \equiv \Pi u_1{:}A_1 \ldots \Pi u_n{:}A_n.a\,M_1 \ldots M_n$ for some family $a$ and objects $M_1, \ldots, M_n$.
4. (Decidability) All judgments of the LF type system are decidable.

The existence of canonical forms for types will be used tacitly in the remainder of this paper. For example, the phrase "*in the case that* $M$ *has the form* $\lambda x : \Pi x{:}A.B\ .\ M'$" is to be interpreted as "*in the case that* $M$ *has the form* $\lambda x : A'\ .\ M'$ *where* $A' \equiv \Pi x{:}A.B$".

## 3 A Meta-Logic for Unification

The foundation of the logic programming paradigm is goal-directed search and unification. Due to the nature of LF, both of these differ significantly from first-order logic programming languages. In particular, proof search and unification become intertwined and unification is no longer a simple subroutine. This phenomenon is already familiar from constraint logic programming [17, 24], but Elf has at least one additional complication: goals are identified with logic variables (see Section 4).

This set of circumstances calls for a new approach to describe the operational semantics of Elf. The key idea is to develop an explicit meta-logic for LF, not to prove properties about LF, but to

describe the operational behavior of Elf. This meta-logic was called *state logic* in [26], since it is used to describe the states of an abstract interpreter for Elf. An alternative approach was taken by Pym & Wallen [29] who give a sequent presentation for LF which allows free meta-variables to appear in the sequents.

We begin with the discussion of unification which will be extended to a logic strong enough to describe the complete state of the interpreter in Section 4. One important property of the unification logic is that it is separated from the formalism of LF and has its own, independent judgments. It is possible to construct such a meta-logic for unification over a number of different term languages and type theories, but a further exploration of this possibility is beyond the scope of this paper.

Miller's *mixed prefixes* [20] and the *existential variables* considered by Dowek [4] perform a function similar to our unification logic. Though Dowek deals with a more expressive logical framework (the Calculus of Constructions) we believe that our meta-logic is more expressive, since we can represent not only a formula in a context with some existential variables, but also several formulas simultaneously which share some context. This allows the natural specification and manipulation of unsolved equations as constraints.

### 3.1 A First-Order Unification Logic with Quantifier Dependencies

The unification logic arises most naturally from a generalization of the usual view of unification as transformations of a set of equations [19, 31]. There we are given set of equations with some free variables. This set is *unifiable* if there is a substitution for the free variables such that all the equations become true. A unification algorithm is described as a set of transformations which can be used to transform the original set of equations into a *solved form* from which a satisfying substitution can be extracted easily.

We transform this view, first by replacing the notion *set of equations* by the notion *conjunction of equations*. The second step is to existentially quantify explicitly over the free variables. Thus the logical content of a unification problem becomes apparent.

Now the problem of determining unifiability becomes one of establishing the truth of a certain closed formula. When function symbols are uninterpreted, this view gives rise to a deductive system in which provable formulas correspond to unifiable equations. We further generalize the usual unification by allowing explicit universal quantification. In first-order unification, this generalization is not necessary, since one can use Skolemization, though even there it may have some interest for efficient implementation. In the higher-order case Skolemization is more problematic and thus explicit variable dependency is desirable not only from the point of view of implementation, but also in order to simplify the theory.

In summary, the unification logic is built upon the following fragment of first-order logic, defined inductively by a BNF grammar.

$$\textit{Formulas} \quad F ::= u \doteq v \mid \top \mid F \wedge G \mid \exists x.F \mid \forall y.F$$

We use $x$ and $y$ to stand for variables, $u$ and $v$ to stand for terms, and $F$ and $G$ to stand for formulas. The basic judgment is $\Vdash F$ ($F$ is provable) is defined by the following inference rules:

$$\frac{}{\Vdash u \doteq u} \qquad \frac{}{\Vdash \top} \qquad \frac{\Vdash F \qquad \Vdash G}{\Vdash F \wedge G}$$

$$\frac{\Vdash [u/x]F}{\Vdash \exists x.F} \qquad\qquad \frac{\Vdash F}{\Vdash \forall y.F}$$

Here $[u/x]F$ is the result of substituting $u$ for $x$ in $F$, renaming bound variables if necessary to avoid capturing free variables in $u$.

### 3.2 Transformations for First-Order Unification

In the representation of unification problems as sets, one might have the following set of transformations

$$\begin{array}{rcll} \{u = u\} \cup S & \Longrightarrow & S & \textit{identity} \\ \{u = v\} \cup S & \Longrightarrow & \{v = u\} \cup S & \textit{exchange} \\ \{f(u_1, \ldots, u_n) = f(v_1, \ldots, v_n)\} \cup S & \Longrightarrow & \{u_1 = v_1, \ldots, u_n = v_n\} \cup S & \textit{term decomposition} \\ \{x = v\} \cup S & \Longrightarrow & \{x = v\} \cup [v/x]S & \textit{variable elimination} \end{array}$$

when $x$ is not free in $v$. A pair $x = u$ in $S$ is in *solved form* if $x$ does not occur in $u$ or elsewhere in $S$, and $S$ is in solved form if every member of $S$ is in solved form. A set of equations $S$ is said to unify iff there is a sequence of transformations $S \Longrightarrow \cdots \Longrightarrow S'$ such that $S'$ is in solved form.

Here, we take an analogous approach. Instead of writing "$\ldots \cup S$" we stipulate that our transformations can be applied to an arbitrary subformula occurrence matching the left-hand side.

$$\begin{array}{rcll} u \doteq u & \longrightarrow & \top & \textit{identity} \\ u \doteq v & \longrightarrow & v \doteq u & \textit{exchange} \\ f(u_1, \ldots, u_n) \doteq f(v_1, \ldots, v_n) & \longrightarrow & u_1 \doteq v_1 \wedge \ldots \wedge u_n \doteq v_n & \textit{term decomposition} \\ \exists x.F[x \doteq t] & \longrightarrow & \exists x.x \doteq t \wedge [t/x](F[x \doteq t]) & \textit{variable elimination} \end{array}$$

where $x$ is not bound in $F$, no free variable in $t$ is bound in $F$, and $x$ does not occur in $t$. Here $F[G]$ is our notation for a formula $F$ with a subformula occurrence $G$. However, these rules are not sufficient to perform unification: we also need some structural rules which allow us to exchange quantifiers (with the obvious provisos regarding variable occurences in the rules dealing with conjunction):

$$\begin{array}{rcl} \exists y.\exists x.F & \longrightarrow & \exists x.\exists y.F \\ F \wedge (\exists x.G) & \longrightarrow & \exists x.F \wedge G \\ (\exists x.F) \wedge G & \longrightarrow & \exists x.F \wedge G \\ \forall y.\exists x.F & \longrightarrow & \exists x.\forall y.F \end{array}$$

A formula is in *solved form* if it has the form $S$ defined inductively by

$$S ::= \top \mid S \wedge S' \mid \exists x.x \doteq u \wedge S \mid \exists x.S \mid \forall y.S$$

where $x$ is not free in $u$.

It is important that the opposite of the last transformation, namely $\exists x.\forall y.F \longrightarrow \forall y.\exists x.F$ is *not* valid. The reason why the quantifier exchange rules are required is somewhat subtle. Consider an example of the form

$$\exists x.\forall y.\exists z.\ x \doteq f(z) \wedge F.$$

$z$ appears in the "substitution term" $f(z)$ for $x$, and it would thus be illegal to instantiate $z$ to a term containing $y$, since this would violate the condition on variable elimination for $x$ ("no free

variable in $t$ may be bound in $F$"). Thus, in order to put the example formula into solved form, we will have to move the quantifier on $z$ outward past the quantifiers on $y$ and $x$, yielding

$$\exists z.\exists x.\forall y.\ x \doteq f(z) \wedge F.$$

Now variable elimination can be applied, yielding

$$\exists z.\exists x.\ x \doteq f(z) \wedge \forall y.\ f(z) \doteq f(z) \wedge [f(z)/x]F.$$

It is now obvious that $z$ can no longer depend on $y$.

These rules are sound and complete for unification as specified by the inference rules for the $\Vdash$ judgment, and, with some additional control structure, can be guaranteed to terminate. We will not formulate these theorems here, as our real interest is in the generalization of this idea to the LF type theory.

### 3.3 A Unification Logic for LF

LF poses two new problems over the first-order setting discussed above: the presence of types and a non-trivial notion of equality between terms. In the specification of the unification logic, this is a rather simple change: in order to prove an equality, we have to show that the terms are $\beta\eta$-convertible, and the rule for the existential quantifier must check that the substituted term is of the correct type. The generalized class of formulas is defined inductively by

$$F \quad ::= \quad M \doteq N \mid \top \mid F \wedge G \mid \exists x{:}A.F \mid \forall y{:}A.F$$

We will use $M$, $N$, $A$, $B$, as in LF, and $F$ and $G$ will again range over formulas. The generalization of the basic provability judgment $\Vdash$ now requires a context $\Gamma$ assigning types to variables. As we will see later, the unification transformations do not need to deal with this context — it maintains its own notion of context and quantifier dependencies through universal and existential quantification. Throughout we make the simplifying assumption that all variable names bound by $\exists$, $\forall$, or $\lambda$ in a formula are distinct. We can then say that $y$ is *quantified outside* of $x$ if the (unique) quantifier on $x$ is in the scope of the quantifier on $y$. The defining inference rules for this unification logic exhibit the connection between the equality formula $M \doteq N$ and $\beta\eta$-convertibility ($M \equiv N$).

$$\frac{\Gamma \vdash M : A \qquad M \equiv N \qquad \Gamma \vdash N : A}{\Gamma \Vdash M \doteq N} \qquad \frac{}{\Gamma \Vdash \top} \qquad \frac{\Gamma \Vdash F \qquad \Gamma \Vdash G}{\Gamma \Vdash F \wedge G}$$

$$\frac{\Gamma \Vdash [M/x]F \qquad \Gamma \vdash M : A}{\Gamma \Vdash \exists x{:}A.F} \qquad \frac{\Gamma, x{:}A \Vdash F}{\Gamma \Vdash \forall x{:}A.F}$$

Substitution at the level of formulas must, of course, substitute into the types attached to the meta-quantifiers and, as before, rename variables when necessary.

### 3.4 A Generalization of $L_\lambda$ Unification to LF

The general problem of higher-order unification is undecidable even for the second-order simply-typed $\lambda$-calculus with only one binary function constant [10]. This result notwithstanding, a complete pre-unification algorithm for the simply-typed $\lambda$-calculus with generally good operational

behavior has been devised by Huet [15]. Extensions to LF have been developed independently by Elliott [5, 7] and Pym [28]. "Pre-unification" here refers to the fact that the algorithm will not enumerate unifiers, but simply reduce the original problem to a satisfiable set of constraints (so-called *flex-flex* pairs) whose unifiers are difficult to enumerate.

While this unification algorithm has proven quite useful in the context of automated theorem proving [1, 13], as the basis for a a logic programming language it has some drawbacks. In particular, the potentially high branching factor and the possibility of non-termination make it difficult to exploit the full power of Huet's algorithm in a safe and predictable way.

Observing the actual practice of programming, both in λProlog and Elf, it is noticable that almost all defensible uses of unification are deterministic. Based on this observation, Miller designed a syntactically restricted logic programming language $L_\lambda$ [21] in which it is guaranteed that only deterministic unification problems arise during program execution. However, the restricted subset disallows important LF representation techniques. For example, the natural rule for universal elimination in an encoding of first-order logic (see [12])

$$\forall E : \Pi F{:}i \to o \;.\; \Pi x{:}i \;.\; \mathit{true}(\forall\, F) \to \mathit{true}(F\, x)$$

would not satisfy Miller's restriction once generalized to LF from the simply-typed λ-calculus, since the variable $x$ (which is subject to instantiation during Elf search) appears in an argument to $F$, which is also subject to instantiation.

Thus, unlike Miller, we make no static restriction of the language. Unification problems which arise during Elf program execution and are not in the decidable subset are simply postponed as constraints. The disadvantage of this approach is that, unlike in Huet's algorithm, constraints cannot be guaranteed to have solutions. For example, given two distinct constants $c : i$ and $c' : i$, the formula $\exists x{:}i \to i \;.\; \exists z{:}i \;.\; x\, z \doteq c \;\wedge\; x\, z \doteq c'$ has no proof and none of the transformations we list are applicable, since the left-hand side of both equations, $x\, z$, is not in the form of a generalized variable (see below). On the other hand the formula $\exists x{:}i \to i \;.\; \exists z{:}i \;.\; x\, z \doteq c$ has many proofs, but no transitions are applicable either. We say that a formula from which no transformations apply is in *reduced form*.

In Elf the fact that reduced forms do not always have solutions has not shown itself to be a practical problem. On the contrary: some programs (such as a program for type reconstruction for the polymorphic λ-calculus) now show markedly improved performance and behavior.

We briefly review the ideas behind the unification in $L_\lambda$ through some special cases. The central concept is that of a *generalized variable* which in turn depends on the notion of a *partial permutation*.

Given $n$ and $p$, a *partial permutation* $\phi$ from $n$ into $p$ is an injective mapping from $\{1, \ldots, n\}$ into $\{1, \ldots, p\}$, that is, $\phi(i) = \phi(i')$ implies $i = i'$.

Assume we are considering a formula of the form

$$\exists x_1 \ldots \exists x_q \;.\; \forall y_1 \ldots \forall y_p \;.\; N \doteq M$$

(omitting types for the moment). In this special case, $N$ will be a *generalized variable* or *Gvar* if it has the form $x_j\, y_{\phi(1)} \ldots y_{\phi(n)}$ for some partial permutation $\phi$ from $n$ into $p$ and $1 \leq j \leq q$. The following examples should provide some intuition about the way generalized variables are unified with terms. Restore the types of all universal variables as some base type, say $i$, and the remaining types so as to lead to well-typed equations at type $i$. Without formally defining substitutions, the annotations shown below should give an indication of the substitution term used for the existential variable in the proof of $\Vdash F$, where $F$ is the formula on the left.

| | | |
|---|---|---|
| $(i)$ | $\exists x.\forall y_1.\forall y_2 \,.\, x\, y_1\, y_2 \doteq y_1$ | $[(\lambda u_1.\lambda u_2.u_1) \,/\, x]$ |
| $(ii)$ | $\exists x.\forall y_1.\forall y_2 \,.\, x\, y_1\, y_2 \doteq g\, y_2$ | $[(\lambda u_1.\lambda u_2.g\, u_2) \,/\, x]$ |
| $(iii)$ | $\exists x.\forall y_1.\forall y_2.\forall y_3 \,.\, x\, y_2\, y_3\, y_1 \doteq x\, y_1\, y_3\, y_2$ | $[(\lambda u_1.\lambda u_2.\lambda u_3.x'\, u_2) \,/\, x]$ |
| | where $x'$ is a new existential variable. | |
| $(iv)$ | $\exists x_1.\exists x_2.\forall y_1.\forall y_2.\forall y_3 \,.\, x_2\, y_3\, y_1 \doteq x_1\, y_2\, y_3$ | $[(\lambda v_1.\lambda v_2.x_3\, v_2) \,/\, x_1]$ |
| | | $[(\lambda u_1.\lambda u_2.x_3\, u_1) \,/\, x_2]$ |
| | where $x_3$ is a new existential variable. | |

The remaining are counterexamples to illustrate branching when restrictions on generalized variables are violated. Each of these have two most general solutions.

| | |
|---|---|
| $\forall y_1.\exists x \,.\, x\, y_1 \doteq y_1$ | $[(\lambda u_1.u_1) \,/\, x]$ |
| | or $[(\lambda u_1.y_1) \,/\, x]$ |
| $\exists x.\forall y_1 \,.\, x\, y_1\, y_1 \doteq y_1$ | $[(\lambda u_1.\lambda u_2.u_1) \,/\, x]$ |
| | or $[(\lambda u_1.\lambda u_2.u_2) \,/\, x]$ |
| $\forall y.\exists x_1.\exists x_2 \,.\, x_1\, x_2 \doteq y$ | $[(\lambda u_1.u_1) \,/\, x_1]\ [y \,/\, x_2]$ |
| | or $[(\lambda u_1.y) \,/\, x_1]$ |

Now we state the transformations, beginning with those common to $L_\lambda$ and Elliott's algorithm and then introduce the notion of a generalized variable. We omit some types in the presentation below—they can be inferred easily from the context. As before, transformations may be applied at any subformula occurrence which matches the left-hand side.

$$
\begin{array}{rcll}
\lambda x{:}A.M \doteq \lambda x{:}A.N & \longrightarrow & \forall x{:}A \,.\, M \doteq N & \text{Lam-Lam} \\
\lambda x{:}A.M \doteq N & \longrightarrow & \forall x{:}A \,.\, M \doteq N\, x & \text{Lam-Any} \\
M \doteq \lambda x{:}A.N & \longrightarrow & \forall x{:}A \,.\, M\, x \doteq N & \text{Any-Lam} \\
 & & & \\
(\lambda x{:}A.M_0)\, M_1\, M_2 \ldots M_n \doteq N & \longrightarrow & ([M_1/x]M_0)\, M_2 \ldots M_n \doteq N & \text{Beta-Any} \\
M \doteq (\lambda x{:}A.N_0)\, N_1\, N_2 \ldots N_n & \longrightarrow & M \doteq ([N_1/x]N_0)\, N_2 \ldots N_n & \text{Any-Beta} \\
 & & & \\
c\, M_1 \ldots M_n \doteq c\, N_1 \ldots N_n & \longrightarrow & M_1 \doteq N_1 \wedge \ldots \wedge M_n \doteq N_n & \text{Const-Const} \\
\forall y.F[y\, M_1 \ldots M_n \doteq y\, N_1 \ldots N_n] & \longrightarrow & \forall y.F[M_1 \doteq N_1 \wedge \ldots \wedge M_n \doteq N_n] & \text{Uvar-Uvar} \\
 & & & \\
\forall y.\exists x.F & \longrightarrow & \exists x.\forall y.F & \text{Forall-Exists} \\
\exists y.\exists x.F & \longrightarrow & \exists x.\exists y.F & \text{Exists-Exists} \\
F \wedge (\exists x.G) & \longrightarrow & \exists x.F \wedge G & \text{And-Exists} \\
(\exists x.F) \wedge G & \longrightarrow & \exists x.F \wedge G & \text{Exists-And}
\end{array}
$$

Our assumption that no variable name is bound twice entails, for example, that $x$ cannot be free in $F$ in the And-Exists transformation. The term *Uvar* in the names of the rules stands for *universal variable.*

In first-order unification, unifying a variable $x$ with itself or another variable is of course trivial—here possible arguments add complications. Huet [15] has shown for the simply-typed $\lambda$-calculus that such equations (Huet calls them *flex-flex*) can always be unified, but that enumeration of all unifiers of such problems is very undirected. This analysis has been extended to LF by Elliott [7]. Here some flexible-flexible pairs can be solved completely, but other unification problems for which Elliott's algorithm would have enumerated solutions or failed, will be postponed. Thus Huet's algorithm and extended $L_\lambda$ unification as presented here are in some sense incomparable: each will

postpone some equations as constraints which could have been solved by the other algorithm. In the eLP implementation of λProlog [6] a combination of the two algorithms is used.

The remaining transitions we consider have a left-hand side of the form

$$\exists x : \Pi u_1{:}A_1 \ldots \Pi u_n{:}A_n.A \;.\\ F[\forall y_1{:}A'_1 \;.\; G_1[\ldots \forall y_p{:}A'_p \;.\; G_p[x\, y_{\phi(1)} \ldots y_{\phi(n)} \doteq M] \ldots]]$$

for some partial permutation $\phi$ from $n$ into $p$. We refer to $x\, y_{\phi(1)} \ldots y_{\phi(n)}$ in the context above as a *generalized variable*. Depending on the form of $M$, we consider various subcases. For some of the transitions there is a symmetric one ($M$ is on the left-hand side) which we will not state explicitly.

The formulation below does not carry along the substitutions made for existential variables, that is, instead of $\exists x.F \longrightarrow \exists x.x \doteq L \wedge [L/x]F$ the transitions have the form $\exists x.F \longrightarrow [L/x]F$. This simplifies the presentation and no essential properties are lost. The substitutions for the original variables can be recovered from the sequence of transformations.

**Gvar-Const** $M$ has the form $c\, M_1 \ldots M_m$ for a constant $c : \Pi v_1{:}B_1 \ldots \Pi v_m{:}B_m.B$. In this case we perform an *imitation* [15]. Let $L = \lambda u_1{:}A_1 \ldots \lambda u_n{:}A_n \;.\; c\,(x_1 u_1 \ldots u_n) \ldots (x_m u_1 \ldots u_n)$ and we make the transition to

$$\exists x_1 : \Pi u_1{:}A_1 \ldots \Pi u_n{:}A_n.B_1 \ldots \\ \exists x_m : \Pi u_1{:}A_1 \ldots \Pi u_n{:}A_n.[x_{m-1}\, u_1 \ldots u_n / v_{m-1}] \ldots [x_1\, u_1 \ldots u_n / v_1]B_m \;.\; [L/x]F$$

Solution of example (*ii*) above would begin with this step.

**Gvar-Uvar-Outside** $M$ has the form $y\, M_1 \ldots M_m$ for a $y$ universally quantified outside of $x$. Here an analogous transition applies (replace $c$ by $y$ in Gvar-Const).

**Gvar-Uvar-Inside** $M$ has the form $y_{\phi(i)}\, M_1 \ldots M_m$ for $1 \leq i \leq n$. In this case we perform a *projection* [15]. Let $L = \lambda u_1{:}A_1 \ldots \lambda u_n{:}A_n \;.\; u_i\,(x_1\, u_1 \ldots u_n) \ldots (x_m\, u_1 \ldots u_n)$ and then perform the same transition as in Gvar-Const where $B_1, \ldots, B_m$ and $B$ are determined (up to conversion) by $A_i \equiv \Pi v_1{:}B_1 \ldots \Pi v_m{:}B_m.B$.

The solution of example (*i*) above would be generated by this transformation.

**Gvar-Identity** $M$ is identical to $x\, y_{\phi(1)} \ldots y_{\phi(n)}$. In this case we simply replace the equation by $\top$.

**Gvar-Gvar-Same** $M$ has the form $x\, y_{\psi(1)} \ldots y_{\psi(n)}$. In this case, pick a partial permutation $\rho$ satisfying $\phi(i) = \psi(i)$ iff there is a $k$ such that $\rho(k) = \phi(i)$. Such a partial permutation $\rho$ always exists and is unique up to a permutation: it simply collects those indices for which the corresponding argument positions in $x\, y_{\phi(1)} \ldots y_{\phi(n)}$ and $x\, y_{\psi(1)} \ldots y_{\psi(n)}$ are identical. Let $L = \lambda u_1{:}A_1 \ldots \lambda u_n{:}A_n \;.\; x'\, u_{\rho(1)} \ldots u_{\rho(l)}$ and make the transition to

$$\exists x' : \Pi u_1{:}A_{\rho(1)} \ldots \Pi u_l{:}A_{\rho(l)} \;.\; [L/x]F$$

Example (*iii*) above illustrates this case.

**Gvar-Gvar-Diff** $M$ has the form $z\, y_{\psi(1)} \ldots y_{\psi(m)}$ for some existentially quantified variable $z$ distinct from $x$ and partial permutation $\psi$. In this case we only apply a transition if we have the following situation:

$$\exists z : \Pi v_1{:}B_1 \ldots \Pi v_m{:}B_m.B \;.\; \exists x : \Pi u_1{:}A_1 \ldots \Pi u_n{:}A_n.A \;.\\ F[\forall y_1{:}A'_1 \;.\; G_1[\ldots \forall y_p{:}A'_p \;.\; G_p[x\, y_{\phi(1)} \ldots y_{\phi(n)} \doteq z\, y_{\psi(1)} \ldots y_{\psi(m)}] \ldots]]$$

for partial permutations $\phi$ and $\psi$, that is, the quantifiers and $z$ and $x$ are consecutive, with $z$ outside of $x$.

In this case, pick two new partial permutations $\phi'$ and $\psi'$ such that $\phi(i) = \psi(j)$ iff there is a $k$ such that $\phi'(k) = i$ and $\psi'(k) = j$. $\phi'$ and $\psi'$ always exist and are unique up to a permutation. Given such partial permutations, we define

$$\begin{array}{rcl} L & = & \lambda u_1{:}A_1 \ldots \lambda u_n{:}A_n.x'\, u_{\phi'(1)} \ldots u_{\phi'(l)} \\ L' & = & \lambda v_1{:}B_1 \ldots \lambda v_m{:}B_m.x'\, v_{\psi'(1)} \ldots v_{\psi'(l)} \end{array}$$

and transform the initial formula to

$$\exists x' : \Pi u_1{:}A_{\phi'(1)} \ldots \Pi u_l{:}A_{\phi'(l)} \,.\, [L'/z][L/x]F$$

An example of this transformation is given by $(iv)$ above.

The last case might seem overly restrictive, but one can use the quantifier exchange rules to transform an equation of two generalized variables into one where Gvar-Gvar-Diff applies. For example

$$\begin{array}{lll} & \exists z.\forall y_1.\exists x.\forall y_2 \,.\, z\, y_1\, y_2 \doteq x\, y_2 & \\ \longrightarrow & \exists z.\exists x.\forall y_1.\forall y_2 \,.\, z\, y_1\, y_2 \doteq x\, y_2 & \text{by Forall-Exists} \\ \longrightarrow & \exists x'.\forall y_1.\forall y_2 \,.\, (\lambda u_1.\lambda u_2 \,.\, x'\, u_2)\, y_1\, y_2 \doteq (\lambda v_1.x'\, v_1)\, y_2 & \text{by Gvar-Gvar-Diff} \\ \longrightarrow^* & \exists x'.\forall y_1.\forall y_2 \,.\, x'\, y_2 \doteq x'\, y_2 & \text{by Beta-Anys and Any-Beta} \\ \longrightarrow & \exists x'.\forall y_1.\forall y_2 \,.\, \top & \text{by Gvar-Identity} \end{array}$$

If we make no restrictions, the transformations stated are not sound. Consider, for example, $\forall y{:}a.\exists x{:}b \,.\, x \doteq y$. This is clearly not provable, since $y$ and $x$ have different types. On the other hand, using Gvar-Uvar-Outside, we can make a transition to $\forall y{:}a.y \doteq y$ which is provable. In the case of the simply-typed $\lambda$-calculus, it is enough to require that any equation $M \doteq N$ in a formula is well-typed (both $M$ and $N$ have the same type in the appropriate context). Here, it is not possible to maintain such a strong invariant due to the dependent types. Instead, we maintain the rather technical invariant that $F$ is *acceptable*. The Definitions 4.37 and 4.38 in [7] can be transcribed into this setting. Following the ideas of Elliott it is then possible to show that if $F \longrightarrow F'$ and $F$ is acceptable, then $F'$ is acceptable. Initially, acceptability is established by type-checking. Definition of these concepts and the proof of the soundness theorem below are beyond the scope of this paper.

**Theorem** (Soundness of Unification) *If $F$ is acceptable, $F \longrightarrow F'$, and $\Gamma \Vdash F'$ then $\Gamma \Vdash F$.*

### 3.5 Precompleteness of the Transformations for Unification

The set of transformations given above is weak in the sense that many unification problems cannot be transformed, regardless of whether they have a solution or not (see the earlier examples).

On the other hand, there are some desirable properties of this restricted form of unification which can be stated once we have fixed an algorithm by imposing a control structure on the rules which explicitly allows for the possibility of failure (indicated by an additional atomic formula $\bot$ which has no proof). The resulting algorithm is deterministic (up to some equivalence, as in the case of first-order unification) and preserves provability, that is if $\Gamma \Vdash F$ and $F \Longrightarrow F'$ then $\Gamma \Vdash F'$. Thus there is no branching in the unification algorithm, and logic program execution can never fail due to incompleteness in unification.

The basis for the more committed formulation of the transformation rules (written as $F \Longrightarrow F'$) is formed by the transformations defining the $\longrightarrow$ relation. We restrict the quantifier exchange rules (they can lead to incompleteness and non-termination) as described below. We also add explicit rules for failure as shown below.

$$\begin{array}{rcll} c\,M_1 \ldots M_n \doteq c'\,N_1 \ldots N_m & \Longrightarrow & \bot & \text{if } c \neq c' \quad \text{Const-Clash} \\ \forall y.F[\forall y'.F'[y\,M_1 \ldots M_n \doteq y'\,N_1 \ldots N_n] & \Longrightarrow & \bot & \quad\quad\quad\quad\quad\quad \text{Uvar-Clash} \end{array}$$

There are also two circumstances under which unification of a generalized variable with a term must fail. Consider as examples:

$$\exists x.\forall y_1.\forall y_2\ .\ x\,y_1 \doteq y_2$$
$$\exists x\ .\ x \doteq g\,x$$

Then, when unifying a generalized variable with a term (where the equation in question has the form $x\,y_{\phi(1)} \ldots y_{\phi(n)} \doteq M$) we may apply:

**Gvar-Uvar-Depend** $M$ has the form $y_i\,M_1 \ldots M_m$ such that $i$ is not in the range of $\phi$. In this case we make a transition to $\bot$.

A variable $x$ is said to *occur rigidly in* $M$ iff *(i)* $M$ is $x\,M_1 \ldots M_n$, or *(ii)* $M$ is $c\,M_1 \ldots M_m$ or $y\,M_1 \ldots M_m$ for a universally quantified $y$, and $x$ occurs rigidly in at least one $M_j$, or *(iii)* $M$ is $\lambda u{:}A.M'$ and $x$ occurs rigidly in $A$ or $M'$ (the definition of rigid occurrence in a type $A$ is analogous). The next rule takes precedence over the Gvar-Const and Gvar-Uvar rules.

**Occurs-Check** The Gvar-Gvar-Same rule does not apply and $x$ occurs rigidly in $M$. In this case we make the transition to $\bot$.

The final case to consider is the case where we unify two distinct generalized variables, but the condition on the transition Gvar-Gvar-Diff is not satisfied. In this case we pick the inner variable, say $z$ and move its quantifier outwards using the quantifier exchange rules until the condition holds.

**Theorem** (Precompleteness of Unification) *If* $\Gamma \Vdash F$ *and* $F \Longrightarrow F'$ *then* $\Gamma \Vdash F'$.

A brief aside: in the implementation, the rules dealing with generalized variables are combined into a form of *generalized occurs-check* which performs three functions: the usual occurs-check along rigid paths [15], the dependency check, which also might lead to failure, and finally it generates constraints (equations which cannot be reduced further) from flexible subformulas which are not generalized variables.

If we also maintain an approximate well-typedness condition [5] we can show termination of unification. Approximate well-typedness is necessary in order to guarantee termination of successive applications of the $\beta$-reduction transformations. This is beyond the scope of this paper, but both approximate well-typing and occurs-check are realized in the implementation.

# 4 Adding Proof Search

Besides the basic notion of unification, the interpreter must be able to perform back-chaining search. This search is modelled after the behavior of ordinary logic programming interpreters, though there are a number of complicating factors. For example, due to the presence of constraints, back-chaining search through a program and unification cannot be completely decoupled. The other

complication is that the logic program (represented as a signature) does not remain static, but changes during execution. In order to describe the form of search necessary here, we extend the unification logic introduced in the previous section to a logic expressive enough to describe the full state of the interpreter. The interpreter is then described in two steps analogous to our presentation of unification: first non-deterministic transformations are presented, and then a control structure is imposed.

### 4.1 A State Logic for the Interpreter

We add two new atomic formulas to the unification logic: $M \in A$ which represents the goal of finding an $M$ of type $A$, and *immediate implication* $M \in A \gg N \in C$, where $C$ must be atomic, that is, of the form $a\, M_1 \ldots M_n$ for a family $a$. The former is used to represent *goals*, the latter to describe *back-chaining*. Formulas now follow the grammar

$$F \quad ::= \quad M \doteq N \mid M \in A \mid N \in A \gg M \in C \mid \top \mid F \wedge G \mid \exists x{:}A.F \mid \forall y{:}A.F$$

It may be possible to formulate this logic without the formulas of the form $M \in A$, since the existential quantifier is typed and thus also imposes a typing constraint (albeit only on existential variables, not arbitrary terms). Economizing the system along these lines would significantly complicate the description of the operational semantics of Elf, since typing information available to unification and typing information necessary for search are not separated.

The meaning of the two new kinds of formulas is defined by the following inference rules.

$$\frac{M \equiv M' \qquad C \equiv C'}{\Gamma \Vdash M \in C \gg M' \in C'} \qquad\qquad \frac{\Gamma \vdash M : A}{\Gamma \Vdash M \in A}$$

$$\frac{\Gamma \Vdash N\, N' \in [N'/x]B \gg M \in C \qquad \Gamma \Vdash N' \in A}{\Gamma \Vdash N \in \Pi x{:}A.B \gg M \in C}$$

The following lemma illustrates the significance of immediate implication and formulas of the form $M \in C$:

**Lemma** (Immediate Implication)

1. *If* $\Gamma \Vdash M \in A$ *then* $\Gamma \vdash M : A$.
2. *If* $\Gamma \Vdash M \in A$ *and* $\Gamma \Vdash M \in A \gg N \in C$ *then* $\Gamma \Vdash N \in C$.

From the inference rule one can see that $(N \in \Pi x_1{:}A_1 \ldots x_n{:}A_n.C) \gg M \in C'$ iff there are appropriately typed terms $N_1, \ldots, N_n$ such that $[N_1/x_1] \ldots [N_n/x_n]C \equiv C'$ and $N\, N_1 \ldots N_n \equiv M$. If there are no dependencies and $N$ is a constant $c$, we can think of $C$ as "the head of the clause named $c$". Then we have that $c \in (A_1 \to \ldots \to A_n \to C) \gg M \in D$ iff $C \equiv D$ and, for proofs $N_i : A_i$, $M \equiv c\, N_1 \ldots N_n$. Now the relation to the back-chaining step in Prolog should become clear: the difference here is that we also have to maintain proof objects. Moreover, the implicit universal quantifier is replaced by $\Pi$, and conjunction in the body of a clause is replaced by nested implication. Thus, for example, the Prolog clause

```
p(X) :- q(X,Y), r(X).
```

would be expressed as the constant declaration

$$c : \Pi x{:}i \,.\, \Pi y{:}i \,.\, r\, x \to q\, x\, y \to p\, x$$

where $c$ can be thought of as the name of the clause, and $i$ : Type is the type of first-order terms. In order to improve readibility of the clauses, we introduce the notation $B \leftarrow A$ to stand for $A \to B$. The $\leftarrow$ operator is right associative. Moreover, type reconstruction will add implicit quantifiers and their types, so in Elf's concrete syntax the clause above would actually be written as

```
c : p X <- q X Y <- r X.
```

In this manner pure Prolog programs can be transcribed into Elf programs.

### 4.2 Transformations for Proof Search

The operational meaning of $M \in A$ and immediate implication is given by the following transformations (recall that $C$ stands for an atomic type of the form $a\, N_1 \ldots N_n$ for some family $a$).

$$\begin{array}{rll}
G_\Pi : & M \in \Pi x{:}A.B & \longrightarrow \quad \forall x{:}A \,.\, \exists y{:}B \,.\, y \doteq M\, x \wedge y \in B \\
G^1_{\mathrm{Atom}} : & \forall x{:}A \,.\, F[M \in C] & \longrightarrow \quad \forall x{:}A \,.\, F[x \in A \gg M \in C] \\
G^2_{\mathrm{Atom}} : & M \in C & \longrightarrow \quad c_0 \in A \gg M \in C \qquad \text{where } c_0{:}A \text{ in } \Sigma. \\
 & & \\
D_\Pi : & N \in \Pi x{:}A.B \gg M \in C & \longrightarrow \quad \exists x{:}A.(N\, x \in B \gg M \in C) \wedge x \in A \\
D_{\mathrm{Atom}} : & N \in a\, N_1 \ldots N_n \gg M \in a\, M_1 \ldots M_n & \longrightarrow \quad N_1 \doteq M_1 \wedge \ldots \wedge N_n \doteq M_n \wedge N \doteq M
\end{array}$$

The soundness theorem below is the crucial theorem in the context of logic programming. It has been argued elsewhere [22] that non-deterministic completeness is also an important criterion to consider. Here, completeness fails (even non-deterministically), due to the incompleteness of unification. On the other hand, there is an analogue to the precompleteness theorem in Section 3.5 which is beyond the scope of this paper. But the practical importance of such (pre)completeness theorems is not clear: an early version of Elf as described in [26] based on Elliott's unification algorithm was non-deterministically complete, but in practice less useful than the version we describe here.

**Theorem** (Soundness of Search) *If $F$ is acceptable, $F \longrightarrow F'$, and $\Gamma \Vdash F'$, then $\Gamma \Vdash F$.*

### 4.3 Search Control

What we have described so far could be considered a non-deterministic proof search procedure for LF. However, as the basis of a proof procedure it has some serious drawbacks, such as incompleteness of unification and a very high branching factor in search.

The transitions for unification and search we gave are more amenable to an interpretation as a method for *goal reduction*: rather than completely solve an original goal (given as a formula), we reduce it to another goal (also represented as a formula). In practice, the final reduced goal will often be in a solved form with obvious solutions.

To turn this view of the transformations as goal reductions into a useful programming language, we need mechanisms to control applications of the transformations. The basic ideas for the control mechanisms come from logic programming. This is in contrast to the approach taken in many current proof development systems where *tactics* and *tacticals* are used to describe when inference rules should be applied. Elf gives meta-programs a much more declarative flavor, and programs tend to be easier to read and have more predictable behavior than tactics. Moreover, tactics can easily be defined within Elf (similarly to Felty's formulation [8]). Finally, by programming directly

in a language with dependent types, static typechecking can make stronger correctness guarantees than functional meta-languages without dependent types (such as ML, for example).

The way dependent types can be used to impose constraints on logic variables is one of the attractive features of Elf. The feature is missing if one follows the proposal by Felty & Miller [9] to interpret LF signatures in Hereditary Harrop logic. While their encoding is adequate on the declarative level, it is inadequate on the operational level, since typing constraints are expressed as predicates which are checked after the execution of a goal, thus potentially leading to much backtracking and generally undesirable operational behavior.

### 4.4 Open and Closed Type Families

Since we now would like to think operationally, we speak of a formula in the state logic as a *goal*. The first control mechanism we introduce is to distinguish goals we would like to be fully solved from those we would like to postpone if possible. This is done by declaring families to be either *open* or *closed*. If a family $a$ has been declared open, then any atomic type of the form $a\,M_1 \ldots M_n$ is defined to be open, if a family $a$ has been declared closed, then any atomic type of the form above is also defined to be closed. Every family-level constant must be declared as open or closed, but the syntactic form of this declaration depends on Elf's module system whose design is still in progress, and thus we do not address such syntactic issues here.

Intuitively, if $A$ is open, then a goal $\exists x{:}A.F[x \in A]$ should be postponed—otherwise, the current context of $x$ will be searched for entries which could construct a term of type $A$. The type of $x$ then acts purely as a restriction on the instantiations for $x$ which might be made by unification. Thus *open* and *closed* declarations function purely as search control declarations—they do not affect the soundness of the interpreter. In Section 4.5 we explain in more detail what we mean by "postponement" here.

In most examples, types which classify syntactic entities will be declared as open. To illustrate this, consider the following simple signature.

$$\begin{array}{rcl} \mathit{nat} & : & \text{Type} \\ \mathit{zero} & : & \mathit{nat} \\ \mathit{succ} & : & \mathit{nat} \to \mathit{nat} \\ \\ \mathit{eqnat} & : & \mathit{nat} \to \mathit{nat} \to \text{Type} \\ \mathit{refl} & : & \Pi n{:}\mathit{nat}\ .\ \mathit{eqnat}\, n\, n \end{array}$$

There are four possibilities of open/closed declarations. We consider each of them in turn. For the sake of brevity, we omit the types of the quantified variables.

1. If *nat* is open and *eqnat* closed, then the formula

   $$\exists N.\exists M.\exists Q\ .\ N \in \mathit{nat} \wedge M \in \mathit{nat} \wedge Q \in \mathit{eqnat}\,(\mathit{succ}\,N)\,(\mathit{succ}\,M)$$

   will be reduced to $\exists N.N \in \mathit{nat}$ (with some added $\top$ conjuncts). During this transformation, the substitution for $Q$ will be $\mathit{refl}\,N$. In the actual top-level interaction with Elf, the answer substitutions would be `M = N` and `Q = refl N`. This is the typical case and the desired behavior.

2. If *nat* and *eqnat* are both closed the goal above will be fully solved (transformed to a conjunction of $\top$'s). Both $N$ and $M$ are instantiated to *zero* for the first solution. Upon backtracking, $N$ and $M$ will be instantiated to $\mathit{succ}\,\mathit{zero}$, *etc.* The problem with this scheme of "eager"

solution is that solving any free variable of type *nat* may be an overcommittment, leading to a potentially large amount of backtracking. In most applications it is better to leave types such as *nat* open—variables of such type will then only be instantiated by unfication.

3. If *nat* and *eqnat* are both open, then the original goal is already in reduced form and no reductions will be applied. This may lead to very undesirable behavior. For example, the formula $\exists Q.Q \in \mathit{eqnat}\ \mathit{zero}\ (\mathit{succ}\ \mathit{zero})$ is also in reduced form! What the interpreter establishes in such a case is that

$$\lambda Q.Q : \ \mathit{eqnat}\ \mathit{zero}\ (\mathit{succ}\ \mathit{zero}) \to \mathit{eqnat}\ \mathit{zero}\ (\mathit{succ}\ \mathit{zero})$$

   which is a valid LF typing judgment, given the signature above, but not very useful.

4. If *nat* is closed and *eqnat* is open, then the formula

$$\exists N.\exists M.\exists Q\ .\ Q \in \mathit{eqnat}\ N\ (\mathit{succ}\ M) \wedge N \in \mathit{nat} \wedge M \in \mathit{nat}$$

   would be reduced with the substitution of *zero* for $M$ and *zero* for $N$ ($Q$ remains uninstantiated). Upon backtracking, $M$ will be increased to *succ zero*, *etc.* Clearly, this is very undesirable behavior, as there are no solutions to $Q$ in most of these cases.

### 4.5 Depth-First Search

In the spirit of logic programming, search is committed to be depth-first. The enables an efficient implementation without overly constraining the programmer. Of course, this means that search will be incomplete, and the programmer will have to take account of this when formulating programs. Let us make the point again: Elf is a *programming language* and not a theorem prover. Given a signature defining a logic, in Elf one will generally have to *program* a theorem prover—the signature alone will usually not be sufficient.

The operational semantics of Elf is given by imposing a control structure on the application of the transformation rules in the previous sections, that is, the transitions for unification and the transitions for search. The state of the interpreter is completely described by a formula $G$ without free variables. The interpreter traverses this formula $G$ from left to right until it encounters an atomic formula $F$. Depending on the structure of $F$, it takes one of the following actions.

1. If $F$ has the form $M \in \Pi x{:}A.B$ the transformation $G_\Pi$ is applied.

2. If $F$ is of the form $M \in C$ for atomic and closed $C$, it applies $G^1_{\mathrm{Atom}}$ to the innermost quantifier $\forall x{:}A$ such that $A$ is closed and $M \in C$ is in its scope. On backtracking, further universal quantifiers are considered (from the inside out). Finally the signature $\Sigma$ is scanned from left to right, applying $G^2_{\mathrm{Atom}}$ to declarations $c_0 \in A$ for closed $A$. We backtrack when the entire signature has been scanned.

3. If $F$ is an immediate implication, we apply rule $D_\Pi$ if it matches. Finally we apply $D_{\mathrm{Atom}}$ if both atomic types begin with the same family-level constant. Otherwise we backtrack over previous choices.

4. If $F$ is an equality not in reduced form, we repeatedly apply unification transformations to all of $G$ until all equalities are in reduced form or unification fails. In the latter case we backtrack.

5. If $F$ is $\top$, an equality in reduced form (no transformation applies), or $M \in C$ for open $C$, we pass over it, looking for the next atomic formula in $G$ (still searching from left to right). Thus, equalities in reduced form are postponed as constraints which are reexamined whenever unification transitions are applied. If we have scanned all of $G$ and none of cases 1 through 4 apply, the formula is in reduced form and we "succeed". In this case the problem of proving the initial formula has been reduced to proving the current formula (in the extended unification logic).

# 5 An Extended Example

We now illustrate Elf and its operation through an extended example. In order to closely match the implementation and describe some of its features, we use concrete syntax in these examples.

## 5.1 Elf Concrete Syntax

In the syntax, the level of kinds, families, and objects are not distinguished, but they can be determined by type reconstruction. We use *expression* to refer to an entity which may be from any of the three levels. In the last columns we list the corresponding cases in the definition of LF in Section 2.

| | | | | |
|---|---|---|---|---|
| *Expressions* | $e$ | ::= | $\mid c \mid x$ | $a$ or $c$ or $x$ |
| | | | $\mid$ `{`$x$`:`$e$`}`$e$ | $\Pi x{:}A.B$ or $\Pi x{:}A.K$ |
| | | | $\mid$ `[`$x$`:`$e$`]`$e$ | $\lambda x{:}A.M$ or $\lambda x{:}A.B$ |
| | | | $\mid e\, e$ | $A\, M$ or $M\, N$ |
| | | | $\mid$ `type` | Type |
| | | | $\mid e$ `->` $e \mid e$ `<-` $e$ | |
| | | | $\mid$ `{`$x$`}`$e \mid$ `[`$x$`]`$e \mid$ `_` $\mid e{:}e \mid$ `(`$e$`)` | |
| *Signatures* | *sig* | ::= | *empty* $\mid c$ `:` $e$`.` *sig* | |

Here $c$ stands for a constant at the level of families or objects. `A -> B` and `B <- A` both stand for $A \to B$. The later is reminiscent of Prolog's "backwards" implication and improves the readability of some Elf programs. Type reconstruction fills in the omitted types in quantifications `{`$x$`}` and abstractions `[`$x$`]`. Omitted expressions (indicated by an underscore `_`) will also be filled in by type or term reconstruction, though in case of ambiguity a warning or error message results (see Section 5.5). Bound variables and constants in Elf can be either uppercase or lowercase, but free variables in a clause or query must be in uppercase (an undeclared, unbound lowercase identifier is flagged as an undeclared constant).

Because of the different roles of signature entries we sometimes refer to the declaration of a constant of closed type as a *clause*, and a constant of open type as a *constructor*.

## 5.2 Equality in the Simply-Typed $\lambda$-Calculus

We begin with a representation of the simply typed $\lambda$-calculus ($\lambda^{\to}$) in LF. In this formulation, the types are "intrinsic" to the representation: we can only represent well-typed terms. `tp` is the syntactic category of types of $\lambda^{\to}$. We include a single base type `tp` and the `arrow` type constructor to form function types. Both families are *open*: they will only be instantiated via unification, not through a search of the signature.

```
tp    : type.
arrow : tp -> tp -> tp.
nat   : tp.
```

The representation of a *term* of $\lambda^{\rightarrow}$ is indexed by its type. The constructors of terms represent 0, successor, $\lambda$-abstraction, and application. In order to obtain the maximal benefits of the expressiveness of the meta-language (LF), the variables of the object language $\lambda^{\rightarrow}$ are represented as variables in the meta-language.

```
term : tp -> type.
z    : term nat.
s    : term (arrow nat nat).
lam  : (term A -> term B) -> term (arrow A B).
app  : term (arrow A B) -> term A -> term B.
```

Note the free variables `A` and `B` in the declaration for `lam`. In a pure LF signature, one would have to specify

$$lam : \Pi A{:}tp.\Pi B{:}tp\ .\ (term(A) \rightarrow term(B)) \rightarrow term(arrow\ A\ B).$$

In Elf, the quantifiers on `A` and `B` are inferred, including the type of the variables. The omitted quantifier also has another role: wherever `lam` is encountered subsequently it is replaced by `lam _ _`, where `_` stands for an (LF) object or type to be reconstructed (see Section 5.5 for further discussion).

In this representation, the simply-typed term $(\lambda x{:}nat \rightarrow nat.x)\ z$ is encoded as the following term:

```
app nat nat (lam nat nat [x:term nat] x) z.
```

In concrete syntax this will be accepted and printed as

```
app (lam [x] x) z.
```

The next part of the signature defines $\beta\eta$-equality between terms in $\lambda^{\rightarrow}$. Before showing it, we give the usual two-dimensional representation of three of the inference rules which are discussed below.

$$\frac{}{(\lambda x{:}A.M)\ N \approx [N/x]M}\ \beta \qquad \frac{}{(\lambda x{:}A.M\ x) \approx M}\ \eta \qquad \frac{M \approx M'}{\lambda x{:}A.M \approx \lambda x{:}A.M'}\ \lambda$$

with the proviso that $x$ is not free in $M$ in the $\eta$-rule. The `equiv` judgment is declared as *closed*, since we hardly would want to accept free variables ranging over equivalence proofs in an answer to a query. On the other hand the signature below should never be used for search, as it very quickly gets into infinite loops. In the implementation this is handled by allowing the programmer to specify which signatures will be used in search. The concrete syntax for these language features are likely to change in the near future as the design of the module system for Elf matures, and thus not described here.

```
equiv : term A -> term A -> type.

e_beta : equiv (app (lam M) N)     (M N).
e_eta  : equiv (lam [x] (app M x)) M.
```

```
e_app : equiv (app M N) (app M' N') <- equiv M M' <- equiv N N'.
e_lam : equiv (lam M)   (lam M')    <- {x} equiv (M x) (M' x).

e_refl  : equiv M M.
e_sym   : equiv M N <- equiv N M.
e_trans : equiv M N <- equiv M R <- equiv R N.
```

A number of common techniques are illustrated in this signature. The formulation of the $\beta$-rule takes advantage of LF-level application in order to represent substitution. The usual side-condition on $\eta$-conversion is also implicit: the quantifier on `M` (which we omitted) is on the outside. If we tried to instantiate this quantifier with a term containing `x` free, substitution would actually rename the bound variable `x` in order to avoid a name clash. The rule `e_lam` illustrates the way we descend into abstractions by using the LF-level context.

This signature is never used for search, only for type-checking purposes. One can easily see why: depth-first search using this signature, or any other naive control structure will almost inevitably lead to non-termination. Instead we take the approach of explicitly giving an *algorithmic* formulation of equivalence in terms of a signature which *can* be used for search. The soundness of the algorithmic formulation with respect to the definition can then be expressed in Elf (see Section 5.4).

### 5.3 An Algorithmic Formulation

The algorithmic version of equality requires three separate judgments: $M$ and $N$ are equal at type $A$ (the main judgment), $M$ weakly head reduces to $N$, and $M$ and $N$ have the same head and equal arguments. All three judgments are declared to be *closed*.

We begin with weak head reduction, `whr`. As an inference system, the above might have been written as

$$\frac{}{(\lambda x{:}A.M)\,N \longrightarrow_{\mathrm{whr}} [N/x]M}\,\mathrm{redex} \qquad\qquad \frac{M \longrightarrow_{\mathrm{whr}} M'}{M\,N \longrightarrow_{\mathrm{whr}} M'\,N}\,\mathrm{left}$$

The Elf formulation is not much more verbose than the formulation through inference rules. Moreover, it expresses and verifies directly that weak head reduction relates terms of the same type! The possibility of statically verifying this property during type reconstruction stems from the use of dependent types in the declaration of `whr`: we explicitly state that both arguments to `whr` have the same type `A` (in $\lambda^{\rightarrow}$).

```
whr : term A -> term A -> type.

whr_redex : whr (app (lam M) N) (M N).
whr_left  : whr (app M N)       (app M' N) <- whr M M'.
```

Next we introduce the main judgments, `eq` and `eq'`, which are mutually recursive. `eq` reduces equality at function types to equality at base type by using extensionality, and `eq'` checks whether two terms have the same head and correspondingly `eq` arguments. The type argument to `eq` is made explicit, since the program unifies against this type. This is only a stylistic decision: if the explicit quantifier over `A` were omitted, we could, for example, formulate the rule `eq_base` as `eq (M:nat) N <- eq' M N`.

```
eq  : {A:tp} term A -> term A -> type.
eq' : term A -> term A -> type.

eq_arrow : eq (arrow A B) M N
             <- ({x:term A} eq' x x -> eq B (app M x) (app N x)).

eq_base : eq nat M N <- eq' M N.

eq_whrl : eq nat M N <- whr M M' <- eq nat M' N.
eq_whrr : eq nat M N <- whr N N' <- eq nat M N'.

eq'_z   : eq' z z.
eq'_s   : eq' s s.
eq'_app : eq' (app M N) (app M' N') <- eq' M M' <- eq _ N N'.
```

From the formulation of the inference rules for `eq'` it may seem that there is no case for variables. However, when a variable is added to the context (in the right-hand side of the `eq_arrow` rule), we also add to the context a proof that this variable is equal (`eq'`) to itself—another common programming technique in Elf.

The use of `_` in the last clause indicates that the omitted type should be inferrable from the context. We have no explicit name for the type here, since it appears only in the synthesized and thus suppressed arguments to `eq'` and `app`. We could have easily created such a name through an explicit type annotation, as in

```
eq'_app : eq' (app M (N:term A)) (app M' N') <- eq' M M' <- eq A N N'.
```

To see the practical utility of implicit syntax and implicit quantification as realized in Elf, consider the explicit version of the `eq'_app` rule:

```
eq'_app : {A:tp} {N:term A} {N':term A}
            {A':tp} {M:term (arrow A A')} {M':term (arrow A A')}
              eq A N N' -> eq' (arrow A A') M M'
                -> eq' A' (app A A' M N) (app A A' M' N').
```

In particular deductions where the 6 implicit arguments are shown are completely unreadable.

Here is a sample query as given to the interactive top-level loop of Elf which is modeled after Prolog.

```
?- eq A (app (lam [x] x) s) M.

M <- s ,
A <- arrow nat nat .

Query <- eq_arrow ([x:term nat] [p:eq' x x] eq_whrl
          (eq_base (eq'_app (eq_base p) eq'_s)) (whr_left whr_redex)) .
```

Here the substitution for `A` is determined during type reconstruction, the substitutions for `M` and `Query` are constructed by search and unification.

### 5.4 Soundness of the Algorithmic Formulation

We would now like to show that the program for `eq` given above is sound with respect to the definition of `equiv`—so far we have made no formal connection. The most immediate one is to interpret `whr`, `eq`, and `eq'` all as establishing equivalence between two terms, and then show that the rules for those judgments are derived rules. In an extension of Elf currently under development this interpretation will be in the form of a functor (in the terminology of the Standard ML module system). Here we simply show how the definitions as they might appear in the body of such a functor.

```
def whr = equiv.

def whr_redex  = e_beta.
def whr_left P = e_app e_refl P.

def eq A = equiv.
def eq'  = equiv.

def eq_arrow P = e_trans (e_trans e_eta (e_lam P)) (e_sym e_eta).
def eq_base P  = P.

def eq_whrl Q P = e_trans P Q.
def eq_whrr Q P = e_trans (e_sym Q) P.

def eq'_z       = e_refl.
def eq'_s       = e_refl.
def eq'_app Q P = e_app P Q.
```

The fact that these definitions are type-correct is enough to guarantee the soundness of our algorithm for deciding equality for $\lambda^{\rightarrow}$, since it allows us to interpret the "trace" (algorithmic deduction) of `eq A M N` as a proof of `equiv M N`. Of course, such a direct interpretation will not always be possible. This is because the LF type theory has no recursion operator, as the addition of operators for recursion would render many encodings as not adequate. In such cases the soundness of an operational formulation can only be expressed as a relation. In our example, the following signature relates algorithmic deductions to proofs of equivalence.

```
treq  : eq A M N -> equiv M N -> type.
treq' : eq' M N -> equiv M N -> type.
trwhr : whr M N -> equiv M N -> type.

tr_redex  : trwhr whr_redex     e_beta.
tr_redexl : trwhr (whr_left Tr) (e_app e_refl P) <- trwhr Tr P.

tr_z   : treq' eq'_z e_refl.
tr_s   : treq' eq'_s e_refl.
tr_app : treq' (eq'_app Tr Tr') (e_app P P') <- treq Tr P <- treq' Tr' P'.

tr_base  : treq (eq_base Tr') P <- treq' Tr' P.
```

```
tr_arrow : treq (eq_arrow F) (e_trans (e_trans e_eta (e_lam P)) (e_sym e_eta))
             <- {x} {tr'} treq' tr' e_refl -> treq (F x tr') (P x).

tr_whrl : treq (eq_whrl Tr TrW) (e_trans P Q)
            <- treq Tr P <- trwhr TrW Q.
tr_whrr : treq (eq_whrr Tr TrW) (e_trans (e_sym Q) P)
           <- treq Tr P <- trwhr TrW Q.
```

Thus while the method of direct interpretation allows a form of compile-time soundness proof through the type checker, the second method requires that we explicitly transform a proof `Tr` of `eq M N` into a proof `P` of `equiv M N` by calling `treq Tr P`. We are guaranteed that `P` is an equivalence proof for `M` and `N` only when this translation succeeds.

### 5.5 Type Reconstruction for Elf

The method of type reconstruction for Elf is different from what is used in LEGO [27] or the implementation of the Calculus of Constructions at INRIA [16] in that (1) argument synthesis and type and term reconstruction are completely decoupled, and (2) there is no restriction on which types and terms can be omitted in the input. We only sketch the algorithm here.

If a constant declaration has implicit quantifiers, these arguments are assumed to be implicit and are inserted as underscores during parsing (see the earlier example). In the second phase, the type reconstruction algorithm performs unification as described in Section 3 augmented with straightforward transitions to deal with variables at the level of types. Underscores are converted into logic variables during this phase, and their value may depend on all bound variables it is in the scope of. At present, we do not attempt to infer types for undeclared constants, though underscores may be inserted in any place where a term or type would be legal.

Thus we presuppose no particular flow of information, except that the analysis is done one constant declaration at a time. This avoids some unnatural problems which sometimes arise in other systems. For example, in many synthesis algorithms polymorphic constants with no arguments (such as `nil`) or constructors with only implicit arguments (such as the inference rule `e_beta` in the example above) must be supplied with arguments, even though the context often determines unambiguously what these arguments should be.

This algorithm has surprisingly good operational behavior: it is very efficient, gives good error messages most of the time, and one rarely has to disambiguate by giving additional information. If a type has been reconstructed without any remaining constraints it is a principal type, since unification as described by the transitions in Section 3 does not branch. Moreover, type reconstruction will not fail if there is a valid typing, due to the precompleteness of unification. However, in some, in practice rare cases, constraints remain after type reconstruction. These may or may not be satisfiable and the programmer will be advised to further annotate his program.

Elf has no explicit way to specify that an argument should be synthesized, or to explicitly override synthesis. In the case where the latter would be necessary, one can use the general `:` operator to annotate arbitrary subexpressions and achieve the same effect as the `|` annotation provides in LEGO, though somewhat more verbosely.

## 6 Conclusion and Future Work

To summarize, we have presented the design and preliminary implementation of the logic programming language Elf based on the LF logical framework. As in classical first-order logic programming,

Elf proof search is sound, but not complete with respect to a given LF signature. Unlike Prolog, the interpreter constructs proof terms during the solution of a query, and such proof terms can be used later. In this framework logic variables and goals are identified, a distinction which is replaced by a similar distinction between open and closed families—a distinction made purely for reasons of search control.

Dependent types and can be used as expressive constraints on logic variables. Moreover, dependent types can internally (through Elf type checking) give correctness guarantees which are not possible in a system with only simple types. Elf has been used to specify and write meta-programs in the domains of theorem proving, type checking and inference, natural semantics, program extraction and proof transformation. Some of these are partially verified internally.

The primary deficiencies of Elf as it currently stands are the lack of a module system (under development), the absence of notational definition and polymorphism (both of which are actually provided in the implementation, but for which the theoretical consequences have not been fully investigated), and the inefficiency of the implementation on large problems. We hope to address the latter through a number of improvements, such as dependency analysis to avoid redundant unification, elimination of the construction of proofs whenever it can be shown statically that they will be discarded, and the compilation of signatures to take advantage of indexing techniques and to optimize unification. In particular, the costly (generalized) occurs-check can be avoided in many cases without loss of soundness.

Finally, Elf lacks some control constructs such as cut (`!`), `assert`, `var`, *etc.* that are familiar from logic programming, and occasionally this presents some difficulties. Our basic design principle in extending the language has to be to preserve the declarative reading of a signature at all costs, that is, the operational behavior of a language extension will have to be sound with respect to its declarative interpretation (though usually incomplete). This is more or less forced, as Elf, unlike Prolog, will deliver *proofs*, not just substitutions. On the other hand, we try to be circumspect and only incorporate extensions which in some sense appear necessary. The addition of $\Sigma$-types (strong sums) as first envisioned in [26] has not been necessary as anticipated and has, for the moment at least, been abandoned.

It is possible, however, to include some search control operators in Elf programs without destroying the basic promise to deliver proofs for the original query. An example of such an operator is *once*, where we define that $\Gamma \vdash M : \mathit{once}A$ iff $\Gamma \vdash M : A$, though search for proofs of $\mathit{once}A$ will behave differently from search for proofs of $A$ (operationally, by delivering only the first proof $M$ but deliver no more on backtracking). The theoretical and practical properties of such extensions have not yet been fully investigated.

## References

[1] Peter B. Andrews. On connections and higher-order logic. *Journal of Automated Reasoning*, 5:257–291, 1989.

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

[3] Thierry Coquand. An algorithm for testing conversion in Type Theory In Gérard Huet and Gordon D. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991. To appear.

[4] Gilles Dowek. A proof synthesis algorithm for a mathematical vernacular in a restriction of the Calculus of Constructions. Unpublished manuscript, January 1991.

[5] Conal Elliott. Higher-order unification with dependent types. In *Rewriting Techniques and Applications*, pages 121–136. Springer-Verlag LNCS 355, April 1989.

[6] Conal Elliott and Frank Pfenning. eLP: A Common Lisp implementation of λProlog in the Ergo Support System. Available via ftp over the Internet, October 1989. Send mail to elp-request@cs.cmu.edu on the Internet for further information.

[7] Conal M. Elliott. *Extensions and Applications of Higher-Order Unification*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1990. Available as Technical Report CMU-CS-90-134.

[8] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, July 1989.

[9] Amy Felty and Dale Miller. Encoding a dependent-type λ-calculus in a logic programming language. In M.E. Stickel, editor, *10th International Conference on Automated Deduction, Kaiserslautern, Germany*, pages 221–235. Springer-Verlag LNCS 449, July 1990.

[10] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[11] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Springer-Verlag LNCS 78, 1979.

[12] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204. IEEE, June 1987. An extended and revised version is available as Technical Report CMU-CS-89-173, School of Computer Science, Carnegie Mellon University.

[13] Leen Helmink and René Ahn. Goal directed proof construction in type theory. In Gérard Huet and Gordon D. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991. To appear.

[14] Douglas J. Howe. Computational metatheory in Nuprl. In Ewing Lusk and Ross Overbeek, editors, *9th International Conference on Automated Deduction, Argonne, Illinois*, pages 238–257, Berlin, May 1988. Springer-Verlag LNCS 310.

[15] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[16] Gérard Huet. The calculus of constructions, documentation and user's guide. Rapport technique 110, INRIA, Rocquencourt, France, 1989.

[17] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich*, pages 111–119. ACM, January 1987.

[18] Todd B. Knoblock and Robert L. Constable. Formalized metareasoning in type theory. In *First Annual Symposium on Logic in Computer Science, Cambridge, Massachusetts*, pages 237–248. IEEE Computer Society Press, June 1986.

[19] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[20] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 1991. To appear.

[21] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming*, pages 153–281. Springer-Verlag LNCS 475, 1991.

[22] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Journal of Pure and Applied Logic*, 1988. To appear. Available as Ergo Report 88–055, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[23] Dale Miller, Gopalan Nadathur, and Andre Scedrov. Hereditary Harrop formulas and uniform proof systems. In *Second Annual Symposium on Logic in Computer Science*, pages 98–105. IEEE, June 1987.

[24] Gopalan Nadathur and Dale Miller. An overview of $\lambda$Prolog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, pages 810–827, Cambridge, Massachusetts, August 1988. MIT Press.

[25] Lawrence C. Paulson and Tobias Nipkow. Isabelle tutorial and user's manual. Technical Report 189, Computer Laboratory, University of Cambridge, January 1990.

[26] Frank Pfenning. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322. IEEE, June 1989.

[27] Randy Pollack. Implicit syntax. Unpublished notes to a talk given at the First Workshop on Logical Frameworks in Antibes, May 1990.

[28] David Pym. *Proofs, Search and Computation in General Logic*. PhD thesis, University of Edinburgh, 1990. Available as CST-69-90, also published as ECS-LFCS-90-125.

[29] David Pym and Lincoln Wallen. Investigations into proof-search in a system of first-order dependent function types. In M.E. Stickel, editor, *10th International Conference on Automated Deduction, Kaiserslautern, Germany*, pages 236–250. Springer-Verlag LNCS 449, July 1990.

[30] Anne Salvesen. The Church-Rosser theorem for LF with $\beta\eta$-reduction. Unpublished notes to a talk given at the First Workshop on Logical Frameworks in Antibes, May 1990.

[31] Wayne Snyder and Jean H. Gallier. Higher-order unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, 8:101–140, 1989.