Elf: A Language for Logic Definition and Verified Metaprogramming

Frank Pfenning

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Internet: fp@cs.cmu.edu

Abstract

We describe Elf, a metalanguage for proof manipulation environments that are independent of any particular logical system. Elf is intended for meta-programs such as theorem provers, proof transformers, or type inference programs for programming languages with complex type systems. Elf unifies logic definition (in the style of LF, the Edinburgh Logical Framework) with logic programming (in the style of λProlog). It achieves this unification by giving *types* an operational interpretation, much the same way that Prolog gives certain formulas (Horn-clauses) an operational interpretation. Novel features of Elf include: (1) the Elf search process automatically constructs terms that can represent object-logic proofs, and thus a program need not construct them explicitly, (2) the partial correctness of meta-programs with respect to a given logic can be expressed and proved in Elf itself, and (3) Elf exploits Elliott's unification algorithm for a λ-calculus with dependent types.

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

To appear at the Fourth Annual Symposium on Logic in Computer Science, Asilomar, California, June 5-8, 1989.

1 Introduction

There is a wide variety of deductive systems considered in computer science today (for example Hoare logics, type theories, type deduction systems, operational semantics specifications, first- and higher-order intuitionistic and classical logics). Mechanized support for deduction in a variety of logics and type theories has been the subject of much research (see, for example, Automath [6], LCF [12], HOL [13], Calculus of Constructions [5], Isabelle [25], NuPrl [3]).

In [15], Harper, Honsell, and Plotkin present LF (the Edinburgh Logical Framework) as a general metatheory for the definition of logics. LF provides a uniform way of encoding a logical language, its inference rules and its proofs. In [1], Avron, Honsell, and Mason give a variety of examples for encoding logics in LF. Griffin's EFS (Environment for Formal Systems, see [14]) is an implementation that allows definition of logics and interactive theorem proving in LF. EFS provides a nice syntactic environment, but lacks meta-programming support, this is, it lacks a metalanguage for programming theorem proving, type inference, proof transformation, and similar tasks. We believe that good meta-programming support is essential to obtain adequate theorem proving assistance in an environment that is basically independent of any particular logical system. A signature that defines a logic in LF simply does not contain enough information—it specifies an inference system, but not a useful theorem prover.

In this paper we describe Elf, a metalanguage intended for theorem proving and proof manipulation environments. Its range of applications therefore include the range of applications of such environments (as indicated above). The basic idea behind Elf is to unify logic definition (in the style of LF) with logic pro-

gramming (in the style of λProlog, see [22, 21, 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.

Here are some of the salient characteristics of this unified approach to logic definition and metaprogramming. 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 programmers responsibility to construct object-logic proofs. Secondly, Elf avoids the undesirable operational behavior of meta-programs that sometimes arises from encoding a logic in higher-order logic as done in Isabelle [25] and by Felty and Miller in [11] (see the example in Section 5.3). Finally, the partial correctness of meta-programs with respect to a given logic can be expressed and proved by Elf itself (see the example in Section 5.2). This creates the possibility of deriving verified meta-programs through theorem proving in Elf (see Constable, Knoblock & Bates [4], Knoblock & Constable [19] or Howe [17] for other approaches).

The base language $\lambda_{\Pi\Sigma}$ for Elf is the LF type theory (a simply typed λ-calculus extended to allow dependent function types), enriched with strong sums (which we prefer to call dependent products). Unlike in λProlog, no restriction on goals or programs needs to be made: the completeness theorem for our abstract interpreter guarantees that goal-directed search is complete. Thus, Elf is a language in the spirit of Miller, Nadathur, Pfenning and Scedrov's [21] definition of an abstract logic programming language, though it is not based on *logic* but on the $\lambda_{\Pi\Sigma}$ *type theory*.

The abstract interpreter, formulated as a transition system, is described in Section 3. Transitions correspond either to unification steps or steps in a goal-directed search for a term of the given type. The unification steps are based on an extensions of Elliott's unification algorithm on terms in the LF type theory (see [7] and [8]). The non-deterministic interpreter is made practical by a commitment to depth-first search, a distinction between *open* and *closed* judgments and *dynamic* and *static* constants, and the addition of the cut search directive familiar from Prolog (see Section 4). The language environment includes a notion of module and a term and type-inference algorithm that makes it much more palatable as an implementation language. Experience with λProlog shows that the commitment to depth-first search results in a useful programming language, even though the underlying unification problem is in general only semi-decidable. In Section 5 we give excerpts from some example Elf programs. An implementation of Elf in Common Lisp is currently in progress in the framework of the Ergo project at Carnegie Mellon University.

2 The base language $\lambda_{\Pi\Sigma}$

The base language for Elf, $\lambda_{\Pi\Sigma}$, is the type theory of LF [15], λ_{Π}, enriched by a Σ type constructor. The motivation for extending the LF type theory by Σ are discussed in Section 4.4. We are excluding the λ type family constructor as a matter of convenience—in the formulation of LF in [15], λ at the level of types does not appear in normal forms of types and thus seems essential only for the formulation of type inference algorithms. To simplify the presentation we consider α-convertible terms to be identical and also assume that all constants in a signature and variables in a context are distinct.

2.1 Syntax

There are five syntactic categories, just as in λ_{Π}. In order to be consistent with the notation in [15] we overloaded the symbol Σ to stand for signatures and also be used as a type constructor. It should always be obvious which one is meant. We will use M and N to stand for terms, A and B to stand for types, and K to stand for kinds.

Signatures	$\Sigma ::= \langle\rangle \mid \Sigma, c{:}K \mid \Sigma, c{:}A$
Contexts	$\Gamma ::= \langle\rangle \mid \Gamma, x{:}A$
Kinds	$K ::= \mathrm{Type} \mid \Pi x{:}A\,.\,K$
Type Families	$A ::= c \mid A\,M \mid \Pi x{:}A\,.\,B \mid \Sigma x{:}A\,.\,B$
Terms	$M ::= c \mid x \mid \lambda x{:}A\,.\,M \mid M\,N$
	$\mid (M, N) \mid \mathrm{fst}\,M \mid \mathrm{snd}\,M$

We will use the abbreviations $A \to B$ for $\Pi x{:}A\,.\,B$ when x is not free in B, and $A \times B$ for $\Sigma x{:}A\,.\,B$ when x is not free in B. $[M/x]N$, $[M/x]A$, and $[M/x]K$ are our notation for substitution, renaming bound variables if necessary to avoid name clashes. Atomic types C are types that begin with neither a Π nor a Σ.

2.2 Typing rules

We adopt the presentation from [15] and add the following rules for dependent products (strong sums):

$$\frac{\Gamma \vdash_\Sigma A \in \text{Type} \qquad \Gamma, x{:}A \vdash_\Sigma B \in \text{Type}}{\Gamma \vdash_\Sigma \Sigma x{:}A \,.\, B \in \text{Type}}$$

$$\frac{\Gamma \vdash_\Sigma M \in A \qquad \Gamma \vdash_\Sigma N \in [M/x]B}{\Gamma \vdash_\Sigma (M, N) \in \Sigma x{:}A \,.\, B}$$

$$\frac{\Gamma \vdash_\Sigma M \in \Sigma x{:}A \,.\, B}{\Gamma \vdash_\Sigma \text{fst}\, M \in A} \qquad \frac{\Gamma \vdash_\Sigma M \in \Sigma x{:}A \,.\, B}{\Gamma \vdash_\Sigma \text{snd}\, M \in [\text{fst}\, M/x]B}$$

2.3 Conversion rules

When choosing a notion of conversion for $\lambda_{\Pi\Sigma}$ there is an unfortunate tradeoff. From a practical point of view (both for the encoding of inference systems in $\lambda_{\Pi\Sigma}$ and for the implementation of Elf) one would like $\beta\eta$-reduction and the rules for surjective pairing. Unfortunately, the Church-Rosser property for well-typed terms under this notion of reduction is still open, though we conjecture that it holds. One can weaken the notion of conversion and obtain the Church-Rosser property by omitting η and π, but a complete abstract interpreter for Elf using this weaker notion would have to be unduly complex. For λ_Π (without pairing) these tradeoffs already exist and are discussed at some length in [15]. Here are the reduction rules in question:

$$\begin{array}{lll}
\beta & (\lambda x{:}A \,.\, M)\, N \longrightarrow_\beta [N/x]M & \\
\pi^1 & \text{fst}\,(M, N) \longrightarrow_{\pi^1} M & \\
\pi^2 & \text{snd}\,(M, N) \longrightarrow_{\pi^2} N & \\
\eta & (\lambda x{:}A \,.\, M\, x) \longrightarrow_\eta M & \text{if } x \text{ not free in } M \\
\pi & (\text{fst}\, M, \text{snd}\, M) \longrightarrow_\pi M &
\end{array}$$

A term is *normal form* if none of the reduction rules above apply, and a term is *strongly normalizing* if every sequence of reductions terminates. We use $\approx$ for *strong conversion*, that is, all reductions may be used an arbitrary number of times in either direction at any location in a term, type, or kind. *Weak conversion* is generated the same way, but only from β, π^1, and π^2. In either case, a reinterpretation of the type and kind conversion rules in the presentation of λ_Π is required to encompass a larger set of conversions.

2.4 Some properties of $\lambda_{\Pi\Sigma}$

$\lambda_{\Pi\Sigma}$ inherits some, but not all of its properties from LF.

Theorem 1 (Basic properties of $\lambda_{\Pi\Sigma}$ under weak conversion.)

1. *All terms are Church-Rosser.*
2. *All well-typed terms are strongly normalizing.*
3. *Type-checking and kind-checking is decidable.*

Proof sketch: The proof is an extension of the one in [15]. Church-Rosser for this weak notion of conversion can be proved as in [15], since it holds for all, and not only for well-typed terms. To prove strong normalization, we translate both types and terms from $\lambda_{\Pi\Sigma}$ into terms in a simply typed λ-calculus with products and explicit types. It follows from a the strong normalization theorem by Troelstra [28] for a simply typed λ-calculus (even including η and surjective pairing) that such terms are strongly normalizing which in turn implies this for the original terms with dependent types. □

Under strong conversion, only the proof of strong normalization for well-typed terms can be extended in a straightforward way, as indicated in the proof sketch above. The Church-Rosser property now fails in general, but we conjecture that it holds for well-typed terms. For the remainder of this paper we will use the notion of strong conversion, since it is desirable in practice and the basis for our implementation. The theorems are therefore qualified by an assumption about the Church-Rosser property for $\lambda_{\Pi\Sigma}$. The abstract interpreters and theorems could be modified to suit the notion of weak reduction, but the additional complexity introduced seems unwarranted.

One of the properties that does *not* hold for $\lambda_{\Pi\Sigma}$ due to the presence of products is uniqueness of types, that is, a given well-typed term M may have many different types. This may seem like a basic flaw, but it does not lead to problems in Elf where computation originates from the structure of types rather than the structure of terms.

3 An abstract interpreter

Before turning to the formal definition of the abstract interpreter, let us outline why Elf is more than just a theorem prover, but a programming language, similar in many respects to logic programming languages.

The basic idea behind turning a logic into a logic programming language is to identify two sets of formulas: legal goals and legal programs. Many factors may influence the choice of these sets, but we would like to single out particularly important criterion (as argued in [21]): a (non-deterministic) abstract interpreter should be able to perform goal-directed search in such a way that every legal goal that is a theorem in the underlying logic, will succeed. It is interesting to note that this condition is independent of any notion

of unification and thus encompasses *constraint logic programming* (see Jaffar & Lassez [18]).

Here we are in a similar situation, turning a type theory into a programming language. The basic idea is to give *types* an operational interpretation much in the same way that formulas are given an operational interpretation in a logic programming language.

Informally, this operational interpretation is as follows. A goal $z \in \Sigma x{:}A\ .\ B$ should succeed, if the goals $x \in A$ and $y \in B$ both succeed, and z is the pair (x, y). Note that x may occur in B, and that therefore the two subgoals may not be independent. Thus x may serve as a "logical variable" except that it may also range over *proofs* constructed by the interpreter, something not possible in logic programming languages.

A goal $z \in \Pi x{:}A\ .\ B$ should succeed, if the goal $y \in B$ succeeds under the assumption that x has type A, and z is the abstraction $\lambda x{:}A\ .\ y$ (where x may occur free in y, since x may occur free in B). This does not correspond to any construct in a Horn-clause logic, but the Π type construction serves the role of the $\forall$ and $\supset$ connectives in a hereditary Harrop logic (see [24]).

The natural criterion for the choice of legal goal and program types is slightly stronger here, since we would also like to ensure that *all* terms of the given type can be found using goal-directed search, interpreting Σ and Π as outlined above. Actually we can only require that for any term M of the given type A, we can find a term N such that $N \approx M$. Surprisingly, this criterion is satisfied if we admit *all* types of $\lambda_{\Pi\Sigma}$ as goals and programs. We will not formalize and prove this fact as a separate theorem, since it follows rather directly from the completeness of the abstract interpreter. Intuitively, this is due to the presence of strong sums, while a (logical) existential quantifier is only a weak sum.

3.1 A state logic

The inference system for type deduction in $\lambda_{\Pi\Sigma}$ defines a number of judgments, such as convertibility, or $\Gamma \vdash_\Sigma M \in A$. However, the structure of conclusions on the right-hand of $\vdash_\Sigma$ is not expressive enough to describe the states of an abstract interpreter for Elf. In order to gain this expressive power, we generalize $\lambda_{\Pi\Sigma}$ by introducing a new judgment $\Gamma \Vdash_\Sigma F$ with a much richer language for conclusions F. We refer to the conclusions F as *formulas*. We use the letter C to stand for atomic types, which in $\lambda_{\Pi\Sigma}$ have the form $c\, M_1 \ldots M_n$.

$$\begin{array}{rcl} F & ::= & M \doteq N \in A \mid M \in A \mid N \in A \supset M \in C \\ & & \mid T \mid F_1 \wedge F_2 \mid \forall x{:}A\ .\ F \mid \exists x{:}A\ .\ F \end{array}$$

The first line contains the formulas that are considered atomic. Except for atomic $\doteq$ and $\supset$ formulas, this is very close to the unification logic introduced in [26], and it is used in a very similar fashion. The restricted form of implication is used to describe the backchaining in the abstract interpreter. The inference system in Figure 1 defines the judgment $\Vdash_\Sigma$. Note that these inference rules do not define a search process or strategy, merely a judgment—it is the abstract interpreter in Section 3.2 which defines a complete (non-deterministic) search procedure.

The basic property of the state logic is summarized in the following theorem. Of course, the completeness with respect to the atomic formulas that are also judgments in $\lambda_{\Pi\Sigma}$ is obvious.

Theorem 2 (Soundness of state logic)

1. *If* $\Gamma \Vdash_\Sigma M \in A$ *then* $\Gamma \vdash_\Sigma M \in A$.
2. *If* $\Gamma \Vdash_\Sigma M \doteq N \in A$ *then* $\Gamma \vdash_\Sigma M \in A$, $M \approx N$, *and* $\Gamma \vdash_\Sigma N \in A$.

The proof is by straightforward inductions on the form of deductions of $\Gamma \Vdash_\Sigma F$.

3.2 A first non-deterministic abstract interpreter

We now present the non-deterministic transition system on formulas in the state logic that defines our first abstract interpreter. The rewrites may be applied at any occurrence in the state formula. Given implicitly is a signature Σ. The transition system is organized into classes of transitions, each class dealing with different atomic formulas. Since some information is needed by different components of the abstract interpreter, the state formula F will contain some seemingly redundant information.

Goal transitions G**.** These four transitions (see G_Σ, G_Π, G^1_{Atom}, and G^2_{Atom} in Figure 2) analyze formulas of the form $M \in A$. Note that x may appear free in the type B, and remember that C stands for an atomic type.

Backchaining transitions D**.** The final two transitions in the previous group create implications which are now further analyzed by the transitions D_Π, D^1_Σ, D^2_Σ, and D_{Atom} in Figure 2). In a Horn-clause logic

$$\Gamma \Vdash_\Sigma T \qquad \frac{\Gamma \Vdash_\Sigma F_1 \qquad \Gamma \Vdash_\Sigma F_2}{\Gamma \Vdash_\Sigma F_1 \wedge F_2}$$

$$\frac{\Gamma \vdash_\Sigma M \in A}{\Gamma \Vdash_\Sigma M \in A} \qquad \frac{\Gamma \vdash_\Sigma M \in A \qquad M \approx N \qquad \Gamma \vdash_\Sigma N \in A}{\Gamma \Vdash_\Sigma M \doteq N \in A}$$

$$\Gamma \Vdash_\Sigma M \in C \supset M \in C \qquad \frac{\Gamma \Vdash_\Sigma N\, N_0 \in [N_0/x]B \supset M \in C \qquad \Gamma \Vdash_\Sigma N_0 \in A}{\Gamma \Vdash_\Sigma N \in \Pi x{:}A\,.\,B \supset M \in C}$$

$$\frac{\Gamma \Vdash_\Sigma \mathrm{fst}\, N \in A \supset M \in C}{\Gamma \Vdash_\Sigma N \in \Sigma x{:}A\,.\,B \supset M \in C} \qquad \frac{\Gamma \Vdash_\Sigma \mathrm{snd}\, N \in [\mathrm{fst}\, N/x]B \supset M \in C}{\Gamma \Vdash_\Sigma N \in \Sigma x{:}A\,.\,B \supset M \in C}$$

$$\frac{\Gamma, x{:}A \Vdash_\Sigma F}{\Gamma \Vdash_\Sigma \forall x{:}A\,.\,F} \qquad \frac{\Gamma \Vdash_\Sigma [M/x]F \qquad \Gamma \Vdash_\Sigma M \in A}{\Gamma \Vdash_\Sigma \exists x{:}A\,.\,F}$$

Figure 1: Deduction rules for the state logic

they can be formulated more easily, since the necessary subgoals are immediately available in the body of a clause—here subgoals have to be constructed. Of course, the actual implementation can be more efficient. In D_{Atom}, the types $A_1, \ldots, A_n$ and A are determined from the kind of c in the signature. Note that A_i may contain N_j for $j < i$, and A may contain all terms N_i.

Unification transitions U**.** Unfortunately, space does not permit to include a presentation of the unification transitions, which are discussed in [7]. There are two extensions to Elliott's algorithm required here, both of which have been described for the simply typed λ-calculus and carry over to $\lambda_{\Pi\Sigma}$ in a straightforward way: (1) the dependency of universal and existential quantifiers must be taken into account without Skolemization (see Miller [23]), and (2) the algorithm must deal with products (see Elliott [8]).

We write $\Longrightarrow^*$ for the reflexive and transitive closure of the transition relation $\Longrightarrow$. At this point we are ready to formulate a first preliminary soundness and completeness theorems for the abstract interpreter with respect to the state logic. We will later refine $\Longrightarrow^*$, since the interpreter as stated so far is still too non-deterministic. We are omitting here soundness and completeness theorems for this first abstract interpreter, since they are subsumed by Theorem 4.

3.3 Open and closed judgments

We now introduce the important concepts of *open* and *closed* judgments. Judgments are represented in LF (and Elf) as type families, so they are corresponding notions of open and closed types. This step towards a practical programming language is still fully justified by the underlying type theory and does not introduce any incompleteness. Intuitively, we are willing to tolerate free variables of open type in proofs, but no free variables of closed type. In an encoding of first-order logic as in [15], $\phi\,\mathrm{true}$ would be a closed judgment, while i (representing the domain of individuals) would be an open judgment. It is the programmer's responsibility to annotate constants in the signature as `open` or `closed`, but a convenient defaulting mechanism is provided.

Definition 3 *A state F is* solved *iff*

1. *there are no implicational atomic subformulas in F,*
2. *every atomic subformula $M \doteq N \in A$ is in* solved form[1]*, and*
3. *for every atomic subformula $M \in A$, M is an existentially quantified variable and A is open.*

We now restrict our abstract interpreter to account for open and closed judgments by placing some of the burden for completeness of the unification transitions. Let $\Longrightarrow_U$ be the restriction of $\Longrightarrow$ by restricting uses of transitions rules G^1_{Atom} and G^2_{Atom} to the case where c is a closed type family. Let $\Longrightarrow^*_U$ be the reflexive and transitive closure of $\Longrightarrow_U$.

Theorem 4 *Given a signature Σ with open type constants $\mathcal{O}$ and a type A with free variables $y_1{:}A_1, \ldots, y_n{:}A_n$. Under the assumption of the weak*

[1] Pairs in solved form are guaranteed to have solutions. Elliott's unification algorithm uses the criterion that both M and N are "flexible" (see [7]).

$$
\begin{array}{rlcl}
G_\Sigma: & M \in \Sigma x{:}A\,.\,B & \Longrightarrow & \exists x{:}A\,\exists y{:}B\,.\,M \doteq (x,y) \in \Sigma x{:}A\,.\,B \wedge x \in A \wedge y \in B \\
G_\Pi: & M \in \Pi x{:}A\,.\,B & \Longrightarrow & \forall x{:}A\,\exists y{:}B\,.\,M\,x \doteq y \in B \wedge y \in B \\
G^1_{\mathrm{Atom}}: & M \in C & \Longrightarrow & x \in A \supset M \in C \quad \text{where } M \in C \text{ is in the scope of } \forall x{:}A. \\
G^2_{\mathrm{Atom}}: & M \in C & \Longrightarrow & c_0 \in A \supset M \in C \quad \text{where } c_0{:}A \text{ in } \Sigma.
\end{array}
$$

$$
\begin{array}{rlcl}
D_\Pi: & N \in \Pi x{:}A\,.\,B \supset M \in C & \Longrightarrow & \exists x{:}A\,.\,(N\,x \in B \supset M \in C) \wedge x \in A \\
D^1_\Sigma: & N \in \Sigma x{:}A\,.\,B \supset M \in C & \Longrightarrow & \mathrm{fst}\,N \in A \supset M \in C \\
D^2_\Sigma: & N \in \Sigma x{:}A\,.\,B \supset M \in C & \Longrightarrow & \mathrm{snd}\,N \in [\mathrm{fst}\,N/x]B \supset M \in C \\
D_{\mathrm{Atom}}: & N \in c\,N_1 \ldots N_n \supset M \in c\,M_1 \ldots M_n & & \\
 & \Longrightarrow N_1 \doteq M_1 \in A_1 \wedge \ldots \wedge N_n \doteq M_n \in A_n \wedge N \doteq M \in A & &
\end{array}
$$

Figure 2: Transition of non-deterministic abstract interpreter

Church-Rosser property for $\lambda_{\Pi\Sigma}$ under strong conversion, $\Vdash_\Sigma \exists y_1{:}A_1 \ldots \exists y_n{:}A_n\ \exists x{:}A\ .\ y_1 \in A_1 \wedge \ldots \wedge y_n \in A_n \wedge x \in A \Longrightarrow^*_U \Vdash_\Sigma F$ for some solved F iff there are $N_1, \ldots, N_n$ and M such that $\vdash_\Sigma M \in [N_1/y_1] \ldots [N_n/y_n]A$ and any free variable in $N_1, \ldots, N_n$ and M has open type.[2]

The proof is constructive, that is, gives explicit transformations of transition sequences to deductions in the state logic and vice versa. It also requires the completeness of higher-order unification, since open types are not analyzed as goals with respect to the signature.

As discussed earlier, a different version of this theorem for a modified interpreter can be given for $\lambda_{\Pi\Sigma}$ under weak conversion, but this modified theorem is not practically motivated. Note that only completeness depends on the Church-Rosser property, not soundness.

4 The Elf language and interpreter

We now proceed to turn the abstract interpreter into a practical interpreter, following the ideas underlying λProlog. These commitments and extensions are motivated by the experience with Prolog and λProlog, and completeness is lost. This step leads to Elf as a true *programming language* in which one can write theorem proving programs, rather than a logic-independent theorem prover (which we believe to be a problem too difficult for a general, complete solution).

[2]The abstract interpreter satisfies a stronger condition: it characterizes *all* terms $M \in A$. Stating a theorem to this effect would require a discussion of higher-order preunification, which is beyond the scope of this paper.

4.1 Depth-first search

Search through the program is committed to be depth-first. This means that the interpreter goes through the state formula from left to right until it encounters an atomic formula F.

1. If F matches the left-hand side of G_Σ or G_Π, that rule is applied.

2. If F is of the form $M \in C$ for atomic and closed C, it applies G^1_{Atom} to the innermost quantifier $\forall x{:}A$ such that A is closed and $M \in C$ is in its scope. On backtracking, the next further universal quantifier is considered, etc., until all have been considered. Finally the *current signature* Σ (see Section 4.2) is scanned from left to right, applying G^2_{Atom} to declarations $c_0 \in A$ for closed A.

3. If F is an implication, we apply rule D_Π if it matches. If D^1_Σ applies, we use it, and use D^2_Σ on backtracking. Finally we apply D_{Atom} if both atomic types begin with the same constant. Otherwise we backtrack over previous choices.

4. If F is T, an equality in solved form, or $M \in C$ for open C, we pass over it, looking for the next atomic formula. Thus, equalities in solved form are constraints in the sense of Jaffar & Lassez [18].

5. If F is an equality not in solved form, we call the unification algorithm on the whole state. Unification may fail (upon which we backtrack), not terminate, or replace the given equality by a conjunction of solved equalities, with T representing the empty conjunction. On backtracking, the unifier will enumerate more solutions, which is nec-

essary since unique most general unifiers do not exist for $\lambda_{\Pi\Sigma}$ in general.

In the remainder of the paper, we will refer to the program defined by cases 1 through 4 as the *goal interpreter*, case 5 defines the *unifier*.

4.2 Dynamic and static constants

In the interpreter as given above we have used the notion of *current signature*. Signatures are the basic unit of programs, and they serve two purposes. They are necessary for unification (which includes term- and type-checking and inference, see Section 4.3) and for the goal-directed search performed by the goal interpreter. If all constants were visible to the goal interpreter, this would lead to very undesirable behavior. For example $\supset\mathrm{E} : \Pi A{:}o\,.\,\Pi B{:}o\,.\,\vdash A \supset B \to \vdash A \to \vdash B$ would apply to any goal of the the form $\vdash C$ and lead to very undirected search. However, the type of the constant $\supset$E must be available to the unifier. Therefore term constants (such as $\supset$E) may be declared as *dynamic* or *static*. A dynamic constant will be used by the interpreter when visible according to the module visibility rules [3]. A static constant will never be used by the goal interpreter. The type of both dynamic and static constants will be visible to the unifier.

4.3 Term and type inference

Using the LF type theory or $\lambda_{\Pi\Sigma}$ without term and type inference can be extremely cumbersome, since much information would have to be given that could be inferred. The basic mechanism for term inference in λ_{Π} is described in [7]. It is extremely important to note that term inference, as defined by Elliott, does *not* require general theorem proving, since it leaves free variables of closed type uninstantiated, even though there may not be any terms without free variables of such a type. Instead, it relies entirely on unification on terms in $\lambda_{\Pi\Sigma}$. Unfortunately, unification on $\lambda_{\Pi\Sigma}$ (and λ_{Π}) is only semi-decidable, and the term-inference problem is only semi-decidable as well. Therefore, a resource bound is put on term inference, and it may return three answers (yes, no, or maybe). In case of a "maybe" the user can add more type information to his program. Experience with higher-order unification suggests that $\lambda_{\Pi\Sigma}$-unification should be able to handle most practical examples of term inference rather easily, so a small resource bound should suffice. The additional complexity of types over terms is small and term-inference can be extended easily to type inference. A more serious practical problem is that of ambiguity: omitted types and terms can often be restored in a number of incompatible ways. Currently, we require more information from the user in such a case.

Elf also offers convenient syntax to specify omitted terms at the place where a constant is declared, rather than where it is used (where one would use "_". For example, one would declare $\supset\mathrm{I}'' : \Pi A{:}o\,.\,\Pi B{:}o\,.\,(\vdash A \to \vdash B) \to \vdash A \supset B$. Then every occurrence of $\supset$I will be replaced by $\supset\mathrm{I}''\,\_\,\_$ when the program is read. If, for some reason, one would like to be more explicit about the first two arguments to $\supset$I, one can still use constants $\supset\mathrm{I}'$ and $\supset\mathrm{I}''$. An additional note on inferred quantifiers. In logic programming languages, it is convenient to be able to omit explicit quantifiers over the free variables in a clause. A new problem arises here if we try to do the same: the order of the quantifiers may depend on the term- and type-inference algorithm. In such a case, the constant will *always* be replaced by an application to omitted terms: the programmer loses the ability to specify the inferred arguments explicitly.

4.4 Σ-types and queries

Let us return to the motivation for including dependent products in the language. Firstly, Σ-types and the ability to form pairs of terms are a matter of convenience, in the same way conjunction in logic programming is a matter of convenience, though not strictly necessary (one could use nested implications). Secondly, products combined with polymorphism significantly strengthen Elf as a representation and metaprogramming language to handle the common case of a object languages with binding constructs of variable arity (see [27]). For example, a natural encoding of Hoare logic in LF as given in Section 4.10 of [1] must be restricted to a fixed number of registers—a restriction that can be dropped in $\lambda_{\Pi\Sigma}$ with polymorphism. Thirdly, Σ-types can introduce "constants" that are local to a module as Σ-quantified variables, something not possible in λProlog. Finally, products are important in queries, where they can be used to hide information (thinking of Σ as an existential quantifier) and to express several goals to be satisfied simultaneously.

5 Examples

We give excerpts from some programs that highlight some of the unique features of Elf that set it apart from related languages such as λProlog. Some of the types given in these examples could be inferred. A feature

[3] similar to the ones in λProlog, see [20]. Another related approach to structuring of theories may be found in [16].

used in the examples that has not yet been discussed is a very weak form of equality in signatures, $c = M$, which is used exclusively for term and type inference.

5.1 A module defining a first-order logic

We give a condensed signature for a first-order logic in the style of Elf below. It is very close to the LF encoding in [15], with the exception of some annotations. We are now using A and B to stand for formulas, and $\vdash A$ for the judgment that A is true.

```
static Module fol
```

$$
\begin{array}{rcl}
\texttt{open}\ i & : & \text{Type} \quad \texttt{\%} \text{ type of terms.} \\
\texttt{open}\ o & : & \text{Type} \quad \texttt{\%} \text{ type of propositions.} \\
\texttt{closed}\ \vdash & : & o \to \text{Type} \quad \texttt{\%} \text{ type of proofs.} \\
\\
\bot & : & o \\
\neg & : & o \to o \\
\wedge, \vee, \supset & : & o \to o \to o \\
\forall, \exists & : & (i \to o) \to o \\
\\
\bot_I{}' & : & \Pi C \,.\, \vdash \bot \to \vdash C \\
\neg\mathrm{I}' & : & \Pi A \,.\, (\vdash A \to \vdash \bot) \to \vdash \neg A \\
\neg\mathrm{E}' & : & \Pi A \,.\, \vdash \neg A \to \vdash A \to \vdash \bot \\
\wedge\mathrm{I}'' & : & \Pi A \,.\, \Pi B \,.\, \vdash A \to \vdash B \to \vdash A \wedge B \\
\wedge\mathrm{E}''_l & : & \Pi A \,.\, \Pi B \,.\, \vdash A \wedge B \to \vdash A \\
\wedge\mathrm{E}''_r & : & \Pi A \,.\, \Pi B \,.\, \vdash A \wedge B \to \vdash B \\
\vee\mathrm{I}'_l & : & \Pi A \,.\, \Pi B \,.\, \vdash A \to \vdash A \vee B \\
\vee\mathrm{I}'_r & : & \Pi A \,.\, \Pi B \,.\, \vdash A \to \vdash B \vee A \\
\vee\mathrm{E}''' & : & \Pi C \,.\, \Pi A \,.\, \Pi B \,.\, \vdash A \vee B \to \\
 & & \quad (\vdash A \to \vdash C) \to (\vdash A \to \vdash C) \to \vdash C \\
\supset\mathrm{I}'' & : & \Pi A \,.\, \Pi B \,.\, (\vdash A \to \vdash B) \to \vdash A \supset B \\
\supset\mathrm{E}'' & : & \Pi A \,.\, \Pi B \,.\, \vdash A \supset B \to \vdash A \to \vdash B \\
\forall\mathrm{I}' & : & \Pi A{:}i \to o \,.\, (\Pi x{:}i \,.\, \vdash A\, x) \to \vdash \forall\, A \\
\forall\mathrm{E}' & : & \Pi A{:}i \to o \,.\, \vdash \forall A \to (\Pi x{:}i \,.\, \vdash A\, x) \\
\exists\mathrm{I}' & : & \Pi A{:}i \to o \,.\, \Pi x{:}i \,.\, \vdash A\, x \to \vdash \exists A \\
\exists\mathrm{E}'' & : & \Pi C \,.\, \Pi A{:}i \to o \,.\, (\Pi x{:}i \,.\, \vdash A\, x \to \vdash C) \\
 & & \quad \to \vdash \exists A \to \vdash C
\end{array}
$$

5.2 Miniscoping

Minimizing the scope of quantifiers is an common preprocessing step in theorem provers. This example illustrates how a "clause" (a declaration `dynamic` $c : A$) in Elf can be proved correct by giving a closed term of type A. In practice, one would obtain the proof of $\forall\text{push}\wedge$ by theorem proving in a programming environment (for example, in the style of Isabelle [25]), an issue beyond the scope of this paper. The excerpt contains two clauses, one that pushes a universal quantifier over a conjunction, and one that pushes a universal quantifier over a disjunction, if the bound variable does not appear free in the right disjunct.

$$
\begin{array}{rcl}
\forall\text{push}\wedge'' & : & \Pi A\ \Pi B \,.\, \vdash\forall\,(\lambda x \,.\, A\, x \wedge B\, x) \leftarrow \vdash\forall A \wedge \forall B \\
\forall\text{push}\wedge'' & = & \lambda A \,.\, \lambda B \,.\, \lambda z{:}\vdash\forall A \wedge \forall B \,.\, \forall\mathrm{I}'\,\_\,(\lambda x{:}i \,. \\
 & & \quad \wedge\mathrm{I}''\,(A\, x)\,(B\, x)\,(\forall\mathrm{E}'\,\_\,(\wedge\mathrm{E}'_l\,\_\,z)\, x)\,(\forall\mathrm{E}'\,\_\,(\wedge\mathrm{E}'_r\,\_\,z)\, x)) \\
\forall\text{push}\vee_l & : & \vdash\forall\,(\lambda x \,.\, A\, x \vee B) \leftarrow \vdash\forall A \vee B \\
\forall\text{push}\vee_l & = & \lambda z{:}\vdash\forall A \vee B \,.\, \forall\mathrm{I}\,(\lambda x{:}i \,.\, \vee\mathrm{E}\, z \\
 & & \quad (\vee\mathrm{I}_l\,\_\,(\lambda y_1 \,.\, \forall\mathrm{E}\, y_1\, x))\,(\vee\mathrm{I}_r\,\_\,(\forall A)))
\end{array}
$$

5.3 Proof reduction and normalization

This next example is the implementation of proof reductions for proofs in a first-order intuitionistic logic. Felty showed in [9] how to implement the reductions in λProlog, but at a very heavy price. Each of her clauses contains a proof-checking subgoal that may be very expensive to execute. If her reductions are combined into a normalization procedure, each reduction step must do proof-checking, something that can be avoided in Elf. This is because Elliott's unification algorithm eliminates redundant type-checking (which is proof-checking in an encoded logic) in many cases. The following reductions rules are verified since the type of reduce$'$ guarantees that its second and third argument are proofs of the same theorem. The constants in this example are anonymous and their names will be generated by Elf.

$$
\begin{array}{rcl}
\text{reduce}' & : & \Pi A \,.\, \vdash A \to \vdash A \to \text{Type} \\
 & & \text{reduce}\ (\neg\mathrm{E}\,(\neg\mathrm{I}\, M)\, N)\ (M\, N) \\
 & & \text{reduce}\ (\wedge\mathrm{E}_l\,(\wedge\mathrm{I}\, M\, N))\ M \\
 & & \text{reduce}\ (\wedge\mathrm{E}_r\,(\wedge\mathrm{I}\, M\, N))\ N \\
 & & \text{reduce}\ (\vee\mathrm{E}\,(\vee\mathrm{I}_l\,\_\, M)\, N_1\, N_2)\ (N_1\, M) \\
 & & \text{reduce}\ (\vee\mathrm{E}\,(\vee\mathrm{I}_r\,\_\, M)\, N_1\, N_2)\ (N_2\, M) \\
 & & \text{reduce}\ (\supset\mathrm{E}\,(\supset\mathrm{I}\, M) N)\ (M\, N) \\
 & & \text{reduce}\ (\forall\mathrm{E}\,(\forall\mathrm{I}\, M)\, T)\ (M\, T) \\
 & & \text{reduce}\ (\exists\mathrm{E}\, N\,(\exists\mathrm{I}\, T\, M))\ (N\, T\, M)
\end{array}
$$

The commutative reductions can be formulated very generally (and non-deterministically). The declaration as `static` ensures that this is not used by the goal interpreter, only by the type-checker.

$$
\begin{array}{l}
\texttt{static}\ \text{cred}''' : \Pi A\ \Pi B\ \Pi C\ \Pi F{:}(\vdash C \to \vdash C) \,. \\
\quad \Pi M{:}\vdash A \vee B \,.\, \Pi N_1 \,.\, \Pi N_2 \,. \\
\qquad \text{reduce}'\ (\vdash C)\ (F\,(\vee\mathrm{E}\, M\, N_1\, N_2)) \\
\qquad\quad (\vee\mathrm{E}\, M\ (\lambda x \,.\, F\,(N_1\, x))\,(\lambda y \,.\, F\,(N_2\, y)))
\end{array}
$$

We can then use specializations of this general rule dynamically in the normalization program. The two specializations listed below apply in the case of a $\supset$E directly preceded by an $\vee$E, either in the left or right premise. The types of cr_1 and cr_2 are inferred by the type inference algorithm. One can write similar specializations for $\vee$E followed by other elimination rules that form a maximal segment. Note the conciseness of

this formulation over the many rules one must state in λProlog, and all of which are instances of the general transformation cred.

$$\begin{array}{lrcl} \texttt{dynamic} & \mathrm{cr}_1 & = & \mathrm{cred}\,(\lambda x\,.\,(\supset \mathrm{E}\,x\,\_)) \\ \texttt{dynamic} & \mathrm{cr}_2 & = & \mathrm{cred}\,(\lambda x\,.\,(\supset \mathrm{E}\,\_\,x)) \end{array}$$

5.4 Other examples

Another important application is to mechanize type-checking for programming languages with complex type systems. The property of a program to be well-typed in such a language can be formalized in Elf as an inference system in the style of LF. This does not immediately lead to a type-checking algorithm, unless there is also a theorem prover for some of the more complex relations (like subtypes). Often the search space for such relations is linear and a decision procedure can be given immediately, that is, the signature itself may be used dynamically.

Natural deduction theorem provers such as Gentzen (see Beeson [2]) or those proposed by Felty and Miller [11] can also be expressed very naturally in Elf. The extraction of programs from proofs is another example of the kind of algorithm that can easily be implemented in Elf. In many of these examples, the implementations are related to λProlog implementations of a similar flavor. The main added advantages of Elf are (1) the program does not need to keep track of *proofs* explicitly—that is done by the Elf interpreter itself (without performance penalty, when proofs are not used), (2) the partial correctness of many programs can be guaranteed by Elf, and (3) the operational behavior of Elf is much better when explicit type-checking or proof-checking would be required in λProlog (as is frequently necessary in the programs obtained by translating LF signatures into λProlog programs as outlined by Felty in [11]).

6 Implementation and further work

An implementation of Elf in Common Lisp is in progress in the framework of the Ergo project at Carnegie Mellon University. Among the important optimizations not mentioned above are (1) signatures are transformed and stored in a hash-table indexed by the type families they define which allows fast backchaining in the style of Prolog, and (2) term construction can often be avoided if the constructed term would not be used (which is frequently the case when a theorem prover is called, since proofs are often irrelevant). The implementation also contains a few extra-logical primitives such as cut, `read` and `write`, and a module system, similar to the one in λProlog.

Extensions we are considering concern polymorphism (which is included in the implementation, but treated in an incomplete way), a stronger notion of definitional equality including δ-reductions, and the embedding of Elf in a general proof development and transformation environment. We are also considering a sublanguage of Elf along the lines of Felty and Miller's L_λ [10] for which unification would be decidable.

Acknowledgments

I would like to thank Ken Cline, Conal Elliott, Amy Felty, and Dale Miller for helpful discussions concerning the subject of this paper.

References

References

[1] 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.

[2] M. Beeson. Some applications of Gentzen's proof theory in automated deduction. 1988. Submitted.

[3] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[4] Robert L. Constable, Todd Knoblock, and Joseph L. Bates. Writing programs that construct proofs. *Journal of Automated Reasoning*, 1(3):285–326, 1984.

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

[6] N. G. de Bruijn. A survey of the project Automath. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, 1980.

[7] Conal Elliott. Higher-order unification with dependent types. In *Rewriting Techniques and Applications*, Springer-Verlag LNCS, April 1989. To appear.

[8] Conal Elliott. *Some Extensions and Applications of Higher-order Unification: A Thesis Proposal.* Ergo Report 88–061, Carnegie Mellon University, Pittsburgh, June 1988. Thesis to appear June 1989.

[9] Amy Felty. *Implementing Theorem Provers in Logic Programming.* Technical Report MS-CIS-87-109, University of Pennsylvania, Philadelphia, December 1987.

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

[11] Amy Felty and Dale A. Miller. Specifying theorem provers in a higher-order logic programming language. In Ewing Lusk and Ross Overbeek, editors, *9th International Conference on Automated Deduction, Argonne, Illinois*, pages 61–80, Springer-Verlag LNCS 310, Berlin, May 1988.

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

[13] Mike Gordon. *HOL: A Machine Oriented Formulation of Higher-order Logic.* Technical Report 68, University of Cambridge, Computer Laboratory, July 1985.

[14] Timothy G. Griffin. *An Environment for Formal Systems.* Technical Report 87-846, Department of Computer Science, Cornell University, Ithaca, New York, June 1987.

[15] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. January 1989. Submitted to JACM. A preliminary version appeared in *Symposium on Logic in Computer Science*, pages 194–204, June 1987.

[16] Robert Harper, Donald Sannella, and Andrzej Tarlecki. Structure and representation in the Edinburgh logical framework. This volume.

[17] 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, Springer-Verlag LNCS 310, Berlin, May 1988.

[18] 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.

[19] 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.

[20] Dale Miller. A logical analysis of modules for logic programming. *Journal of Logic Programming*, 1988. To appear.

[21] 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.

[22] 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.

[23] Dale A. Miller. Unification under mixed prefixes. 1987. Unpublished manuscript.

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

[25] Lawrence C. Paulson. *The Representation of Logics in Higher-Order Logic.* Technical Report 113, University of Cambridge, Cambridge, England, August 1987.

[26] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, ACM Press, July 1988.

[27] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, ACM Press, June 1988. Available as Ergo Report 88–036.

[28] Anne S. Troelstra. Strong normalization for typed terms with surjective pairing. *Notre Dame Journal of Formal Logic*, 27(4):547–550, October 1986.