Modularity in the LF Logical Framework

Robert Harper and Frank Pfenning
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890, U.S.A.
`rwh+@cs.cmu.edu` and `fp+@cs.cmu.edu`

Draft for the Proceedings of the Second Workshop on Logical Frameworks

November 1, 1991

1 Introduction

Formal deductive systems play an important role in computer science, particularly in the areas of logic and semantics of programming languages. They are employed in three different, but obviously related roles. Firstly, they are used to *specify* logics, type systems, operational semantics and other aspects of languages. Secondly, they form the basis for the *implementation* of such deductive systems. Thirdly, they provide an appropriate language for the formulation and proof of *meta-theorems* of programming languages.

The LF Logical Framework [8] has been designed to provide an appropriate language for the high-level specification of deductive systems as they occur in logic and computer science. Its basic principle is often summarized by saying that *judgments* (the basic unit of deductive systems) are represented as *types* and *deductions* as *objects*. The framework was intentionally kept weak (by excluding, for example, polymorphism and impredicative constructs) in order to better support mechanization and to allow a simple meta-theory. This has proved auspicious: algorithms for unification have been developed [5, 21] and the type theory underlying LF has been amenable to an operational interpretation which is realized in the Elf programming language [18, 19]. Furthermore, it also seems possible to express a wide range of meta-theoretic properties of deductive systems within LF, though this line of research is only in its initial stages [15].

We believe that for all three tasks, *specification*, *implementation*, and *meta-theory* of deductive systems, substantial benefits can be derived from explicit structuring mechanisms for the presentation of such systems. In this paper we make a concrete proposal for a module system for the Elf language which attempts to address those three central issues. Various approaches to the static and dynamic semantics of such a module calculus are possible, but beyond the scope of this paper. Here we provide only informal discussions of the meanings of various language constructs and properties. As an extended example throughout the paper we will use two formulations of minimal propositional calculus with implication and conjunction: an axiom system in the style of Hilbert and Gentzen's calculus of natural deduction.

The problem of modularity in the presentation of theories and logical system has been addressed from the semantical [10, 9] and the type-theoretic [3, 4, 25] point of view. Our design has been guided by these ideas and the pragmatic principles of the ML module system [14, 17]. For further discussion of related work, the reader is referred to Section 7.

The remainder of this paper is organized as follows. In Section 2 we review the LF Logical Framework as it is realized within the Elf programming language. As our approach to a module calculus is explicitly stratified (modules do not gain the status of objects, but exist in a different level of language), this core language is not modified in any essential way by the addition of modules. In Section 3 we present a calculus for *signatures* with three basic structuring mechanisms: inclusion, parametrization, and instantiation. As valid objects constructed over a given signatures represent object-language expressions and deductions, this is the centerpiece and most important aspect of the module calculus. In Section 4 we move on to *realizations* which can express logic interpretations through which theorems can be transported from one logic to another. In Section 5 we show how a notion of search, as it underlies the operational semantics of Elf, can naturally be embedded into the module system. In Section 6 we show how limitations of realizations as defined in Section 4 can be circumvented by using relations (rather than functions) between signatures. While such relations do not have the same meta-theoretic force as realizations, they are nonetheless operationally adequate in that they can be executed to implement logic interpretations. We conclude with a brief summary of related work in Section 7 and a recapitulation of the concrete syntax in Appendix A.

2 The Core Language

We briefly review the LF logical framework [8] as realized in Elf [18, 19]. A tutorial introduction to the Elf core language can be found in [15].

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.

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

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).

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

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. Similary, $A \to K$ can stand for $\Pi x{:}A.\ K$ when x does not appear free in K.

The notion of definitional equality we consider here is $\beta\eta$-conversion. Harper *et al.* [8] formulate definitional equality only with β-conversion and conjecture that the system resulting from adding the η-rule would have the properties we list below. This has recently been proved by Coquand [1] and independently by Salvesen [22]. For practical purposes the formulation including the η-rule is superior, since every term has an equivalent canonical form. Thus, for us, $\equiv$ is the least congruence generated from $\beta\eta$-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 Σ 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 [8] and [1].

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 $A \equiv \Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n.\ a\,M_1 \ldots M_m$ for some family constant a and objects $M_1, \ldots, M_m$.
4. (Decidability) All judgments of the LF type system are decidable.

We present concrete syntax for Elf in form of a grammar, where optional constituents are enclosed within $\langle\ \rangle$ and repeated components are shown as $cat_1 \ldots cat_n$. The concrete syntax of the core language is very closely modelled after the abstract syntax presented earlier and is also stratified. We use *term* to refer to an entity which may be from any of the three levels. In the last column we list the corresponding cases in the definition of LF above.

kindexp	::=	`type`	Type
	$\mid$	`{`*id* ⟨`:`*famexp*⟩`}` *kindexp*	$\Pi x{:}A.\ K$
	$\mid$	*famexp* `->` *kindexp*	$A \to K$
	$\mid$	`(`*kindexp*`)`	
famexp	::=	*id*	a
	$\mid$	`{`*id* ⟨`:`*famexp*$_1$⟩`}` *famexp*$_2$	$\Pi x{:}A.\ B$
	$\mid$	`[`*id* ⟨`:`*famexp*$_1$⟩`]` *famexp*$_2$	$\lambda x{:}A.\ B$
	$\mid$	*famexp* *objexp*	$A\,M$
	$\mid$	*famexp*$_1$ `->` *famexp*$_2$	$A \to B$
	$\mid$	*famexp*$_2$ `<-` *famexp*$_1$	$B \leftarrow A$
	$\mid$	`_`	
	$\mid$	*famexp* `:` *kindexp*	
	$\mid$	`(`*famexp*`)`	
objexp	::=	*id*	c or x
	$\mid$	`[`*id* ⟨`:`*famexp*⟩`]` *objexp*	$\lambda x{:}A.\ M$
	$\mid$	*objexp*$_1$ *objexp*$_2$	$M\,N$
	$\mid$	`_`	
	$\mid$	*objexp* `:` *famexp*	
	$\mid$	`(`*objexp*`)`	

id stands either for a bound variable, a free variable, or a constant at the level of families or objects. Bound variables and constants in Elf can be arbitrary identifiers, but free variables in a declaration or query must begin with an uppercase letter (an undeclared, unbound lowercase identifier is flagged as an undeclared constant). Identifiers may contain all characters except `(){}[]:.;%` and whitespace. `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. `->` is right associative, while the left arrow `<-` is left associative. Juxtaposition binds tighter than the arrows and is left associative. The scope of quantifications `{`x `:` A`}` and abstractions `[`x `:` A`]` extends to the next closing parenthesis, bracket, brace or to the end of the term. Term reconstruction fills in the omitted types in quantifications `{`x`}` and abstractions `[`x`]` and omitted types or objects indicated by an underscore `_`. In case of ambiguity a warning or error message results. For a description of Elf's term reconstruction phase see [19].

3 Signatures

In a typical application of the LF methodology of representing deductive systems, an object language and its rules of deduction are defined through a signature. Well-typed objects, constructed from constants over a fixed signature, represent well-formed object language expressions and deductions. In the LF core calculus, such signatures consist of a flat enumeration of constants with their type or kind. In this section we will develop a structuring discipline for signatures.

As an example we will consider various formulations and properties of minimal propositional logic, with implication, conjunction, and truth as logical constants.

At the top-level, we can define a signature as a list of declarations. The simplest declare constants at the level of families (`fam`) or objects (`obj`).

top	::=	`signature` *sigid* `=` *sigexp*	signature binding
	|	top_1 `;` top_2	composition
sigexp	::=	`sig` *decl* `end`	encapsulation
decl	::=	`fam` *id* `:` *famexp*	family
	|	`obj` *id* `:` *objexp*	object
	|	$decl_1$ $decl_2$	composition

We begin with a signature `MPC_LANG` which defines the language of the minimal propositional calculus we consider here. `%` begins a comment which extends to the end of the line.

```
signature MPC_LANG =
sig
  fam o  : type          % Propositions

  obj => : o -> o -> o   % Implication
  obj &  : o -> o -> o   % Conjunction
  obj tt : o             % Truth
end;
```

Propositional variables are represented as LF variables and are therefore not explicitly mentioned in the signature.

There are two basic structuring devices at the level of signatures: a signature can be parameterized by a declaration, and a signature can be included within another signature. We first consider explicit parametrization. This is a more verbose, but more basic, alternative to sharing constraints as they are available within the ML module system [14]. We are considering the possibility of adding sharing constraints to the language in a future version of the module calculus.

Typically, a signature will be parameterized by a realization of another signature. One can think of a realization as providing a definition for each of the constants in the signature ascribed to it. Constants which are defined in such a realization (and therefore declared in the ascribed signature) can be referred to by a prefix, separated from the constant identifier by a period, as in the declaration of `|-` below. The idiom `let open` *realid* `in` *decl* `end` allows the omission of the prefix *realid* within *decl*. In its general form, `let` allows arbitrary local definitions within signatures. These are discussed in more detail in Section 4.

```
signature HILBERT (realizor L : MPC_LANG) =
sig
  % Hilbert deductions
  fam |- : L.o -> type

  let  open L  in
  % Axioms
  obj K     : |- (=> A (=> B A))
  obj S     : |- (=> (=> A (=> B C)) (=> (=> A B) (=> A C)))
  obj ONE   : |- tt
  obj PAIR  : |- (=> A (=> B (& A B)))
  obj LEFT  : |- (=> (& A B) A)
  obj RIGHT : |- (=> (& A B) B)

  % Inference Rule
  obj MP    : |- (=> A B) -> |- A -> |- B

  end  % let open L
end;  % signature HILBERT
```

Variables which are free in a `fam` or `obj` declaration are implicitly quantified within this declaration. Thus the full form of the declaration of the `K` proof constructor would be

$$K : \Pi A{:}o.\, \Pi B{:}o.\ \vdash (\Rightarrow\ A\ (\Rightarrow\ B\ A))$$

in LF, or

```
  obj K     : {A:o} {B:o} |- (=> A (=> B A))
```

in Elf's concrete syntax. Such implicit quantifiers are tied to a form of argument synthesis (as used in systems such as LEGO [20] or the Calculus of Constructions [11]) in that the constant `K` has two implicit arguments which are determined through term reconstruction.[1]

Parameterized signatures can be *instantiated* by providing a definition for the parameter. In the example below this has the form `realizor` *realid* = *realexp* which simply instantiates *realid* to *realexp*. Such instantiations are checked for type-correctness through *signature matching.* This ensures that every constant required by the signature ascribed to a formal parameter *realid* is in

[1]See [19] for a more complete discussion of this aspect of the Elf front-end.

fact provided by the realizor *realexp*. If *realexp* provides additional definitions, those are no longer accessible; that is, signature matching is *coercive* as in ML. We will see more general forms of definition in Section 4.

`include` *sigexp* textually substitutes the signature represented by *sigexp* in place, providing, in effect, a form of signature extension. In the example below we use it with the intent of defining a conservative extension of the signature `HILBERT` by a theorem and a derived rule of inference.

```
signature HILBERT+ (realizor L : MPC_LANG) =
sig
  include HILBERT (realizor L = L)
  obj ID     : |- (L.=> A A)
  obj COMM_& : |- (L.& A B) -> |- (L.& B A)
end;
```

The representation of a proof that this is in fact a conservative extension of the `HILBERT` signature will occupy us in the next section. We conclude here with the syntax for the language of declarations and signatures as it presents itself so far. We defer definitions *defn* to the next section.

top	::=	`signature` *sigid* $\langle \mathit{parm}_1 \ldots \mathit{parm}_n \rangle$ `=` *sigexp*	signature binding
	$\mid$	top_1 `;` top_2	composition
parm	::=	`(`*decl*`)`	parameter declaration
sigexp	::=	`sig` *decl* `end`	encapsulation
	$\mid$	*sigid* $\langle \mathit{inst}_1 \ldots \mathit{inst}_n \rangle$	signature instantiation
inst	::=	`(`*defn*`)`	parameter instantiation
decl	::=	`fam` *id* `:` *kindexp*	family
	$\mid$	`obj` *id* `:` *famexp*	object
	$\mid$	`realizor` *realid* `:` *sigexp*	realizor
	$\mid$	decl_1 decl_2	composition
	$\mid$	`let` *defn* `in` *decl* `end`	local definition
	$\mid$	`include` *sigexp*	inclusion

4 Realizations

Definitions play several roles within the Elf module system. We have already seen two uses of definitions in the previous section: definitions instantiate parameterized signatures, and they can be used to introduce local abbreviations within a signature. Their concrete syntax is defined by the following grammar (to be generalized later).

defn	::=	`fam` *id* $\langle$`:`*kindexp*$\rangle$ `=` *famexp*	family
	$\mid$	`obj` *id* $\langle$`:`*famexp*$\rangle$ `=` *objexp*	object
	$\mid$	`realizor` *realid* $\langle$:*sigexp*$\rangle$ `=` *realexp*	realizor
	$\mid$	defn_1 defn_2	composition
	$\mid$	`let` defn_1 `in` defn_2 `end`	local definition
	$\mid$	`open` *longrealid* $\langle$: *sigexp*$\rangle$	inclusion

Semantically most complex is `open` R, where R may be a qualified realization identifier (explained below). By an invariant of the module system, there will always be a signature expression which has been ascribed to R. For each declaration of the form `fam` *id* `:` *kindexp*, `obj` *id* `:` *famexp*, or `realizor` *realid* `:` *sigexp* in this ascribed signature, `open` creates a definition of the form `fam` *id* `=` R`.`*id*, `obj` *id* `=` R`.`*id*, or `realizor` *realid* `=` R`.`*realid*, respectively. This splices all definitions in R into the current context and the prefix "R`.`" can be omitted in the scope of the `open` definition.

Local definitions (created with `let` *defn*) are transparent: when leaving the scope of a `let`, they are simply expanded by substitution.

Besides providing for instantiation and local abbreviation, definitions are also crucial for the representation of signature morphisms or logic interpretations, here called *realizations.* A realization of a signature S given a realization of a signature T must provide a definition of every constant declared in S, using the constants provided by the realization of T.

As a first example, we show a realization from a signature to an extension. Since it must provide definitions for all constants in the extended signature, such a realization witnesses a conservative (definitional) extension theorem for a logic, if the family representing proofs is realized by itself.

```
realizor Hilbert+
  (realizor L : MPC_LANG)
  (realizor H : HILBERT (realizor L = L))
  : HILBERT+ (realizor L = L) =
real
  open H
  obj ID = MP (MP S K) K
  obj COMM_& = [x : |- (L.& A B)] MP (MP PAIR (MP RIGHT x)) (MP LEFT x)
end;
```

Here, `open H` has the same semantic effect as the definitions

```
  fam |- = H.|-
  obj K  = H.K
  obj S  = H.S
  % ... etc ...
```

Elf's type-checking process will check that each constant declared in the signature `HILBERT+ (realizor L = L)` is given a definition of the appropriate type or kind in the body of the realizor (enclosed in `real`...`end`). An example of a property which must be checked in this case is

```
  MP (MP S K) K  :  |- (L.=> A A).
```

As some arguments are synthesized, internally this requires checking that in a context where `A:o`,

```
  MP A A (MP A A (S A A A) (K A A)) (K A A) : |- (L.=> A A).
```

where all the implicit arguments to `MP`, `S`, and `K` have been determined by term reconstruction. This mechanism is reminiscent of the implicit instantiation of type variables during signature matching in the ML module system. There, type schemas (with implicit universal quantifiers) may be instantiated by types during signature matching; here the Π-quantifiers which remained implicit in the declaration of a constant may be instantiated by objects.

As in ML, signature matching is *coercive*: if we ascribe a type A to a an object definition only this type will be visible externally. Similarly, all definitions within a realizor which do not appear in an ascribed signature are treated as local definitions within a realizor and are not visible at the outside.

As one can see from the example, a realization may also be parameterized by several declarations. The syntax for realizations is otherwise straightforward.

top	::=	`realizor` *realid* $\langle parm_1 \ldots parm_n \rangle$ `=` *realexp*	realizor binding
	|	...	
realexp	::=	`real` *defn* `end`	encapsulation
	|	*longrealid* $\langle inst_1 \ldots inst_n \rangle$	instance

We also introduce two new categories which account for the use of identifiers qualified by a path.

longrealid	::=	$\langle$*longrealid*`.`$\rangle$*realid*	qualified realization identifier
longid	::=	$\langle$*longrealid*`.`$\rangle$*id*	qualified term identifier

The syntax of terms is modified in the obvious fashion in allowing non-binding occurrences of *id* to be *longid*.

In the remainder of this section we give another example of an interpretation, which makes the symmetry of conjunction explicit. Similar duality interpretations have been investigated and used in the IMPS system [6]. Such an interpretation is characterized by the fact that the language under consideration is interpreted in itself in a non-trivial fashion. Here, we interpret `& A B` as `& B A`.

```
realizor SYMM_&_LANG (realizor L : MPC_LANG) : MPC_LANG =
real
  fam o  = L.o
  obj => = L.=>
  obj &  = [A:o] [B:o] L.& B A
  obj tt = L.tt
end;
```

The realization which shows that this interpretation transforms theorems into theorems is non-trivial only in one case: we have to show that the translation of the `PAIR` axiom is a theorem. The proof we reproduce below is perhaps not the most direct Hilbert deduction of this theorem—see Section 5 how such proof objects may be constructed automatically, taking advantage of Elf's dynamic semantics. Note that type-checking guarantees that the proofs supplied below are correct.

```
realizor SYMM_& (realizor L : MPC_LANG)
                (realizor H : HILBERT (realizor L = L))
                : HILBERT (realizor L = SYMM_&_LANG (realizor L = L)) =
real
  let  open L  open H  in

  fam |-    = |-

  obj K     = K
  obj S     = S
  obj ONE   = ONE
  obj PAIR  : |- (=> A (=> B (& B A)))
            = MP (MP S (MP K (MP S (MP (MP S (MP K PAIR)) (MP (MP S K) K)))))
                 (MP (MP S (MP K K)) (MP (MP S K) K))
  obj LEFT  : |- (=> (& B A) A) = RIGHT
  obj RIGHT : |- (=> (& B A) B) = LEFT

  % Inference Rule
  obj MP    = MP

  end  % let open
end;  % realizor SYMM_&
```

5 Search

The Elf core language has an operational interpretation which resembles Prolog's operational interpretation of Horn clauses. We will only sketch it here by means of an example—details and further discussion can be found in [18, 19].

We begin by defining a system of natural deduction for the minimal propositional calculus we have considered so far. We have to model the natural deduction rules reproduced below.

$$\dfrac{\begin{array}{c}\overline{A}^{\,x}\\ \vdots \\ B\end{array}}{A \Rightarrow B} \Rightarrow \mathrm{I}^x \qquad\qquad \dfrac{A \Rightarrow B \qquad A}{B} \Rightarrow \mathrm{E}$$

$$\dfrac{A \qquad B}{A \mathbin{\&} B} \mathbin{\&}\mathrm{I} \qquad \dfrac{A \mathbin{\&} B}{A} \mathbin{\&}\mathrm{E_L} \qquad \dfrac{A \mathbin{\&} B}{B} \mathbin{\&}\mathrm{E_R} \qquad \dfrac{}{\top} \top\mathrm{I}$$

The rule of implication introduction cancels all assumptions of the formula A which have been labelled with x. The transcription of these into Elf follows the standard LF methodology and is discussed in [8]. Note the implicit quantification and the representation of the deduction of B from assumption A as a function which transforms a deduction of A into a deduction of B.

```
signature NATDED (realizor L : MPC_LANG) =
sig
  let  open L  in

  % Natural deductions
  fam !   : o -> type

  % Inference rules
  obj =>I : (! A -> ! B) -> ! (=> A B)
  obj =>E : ! (=> A B) -> ! A -> ! B
  obj &I  : ! A -> ! B -> ! (& A B)
  obj &EL : ! (& A B) -> ! A
  obj &ER : ! (& A B) -> ! B
  obj ttI : ! tt

  end
end;
```

Next we program a very simple theorem prover for this logic. The intended operational use of this is to search for natural deductions in normal form up to a given depth bound. This does not directly construct natural deductions as they are defined above, but bounded normal deductions can be interpreted as natural deductions, as we will show later.

```
signature NAT =
sig
  fam nat : type
  obj z   : nat
  obj s   : nat -> nat
end;

signature BDD_NATDED (realizor L : MPC_LANG) (realizor Nat : NAT) =
sig
  let open L
      obj s = Nat.s
  in

  % Bounded, normal deductions
  % Searching backwards from the conclusion,
  %  using only introduction rules
  fam !<  : Nat.nat -> o -> type

  % Searching forwards from the assumptions,
  %  using only elimination rules
  fam !>  : Nat.nat -> o -> type

  % Inference rules
  obj =>I<  : (({M:Nat.nat} !> (s M) A) -> !< N B) -> !< (s N) (=> A B)
  obj =>E>  : !> N (=> A B) -> !< N A -> !> (s N) B
  obj &I<   : !< N A -> !< N B -> !< (s N) (& A B)
  obj &EL>  : !> N (& A B) -> !> (s N) A
  obj &ER>  : !> N (& A B) -> !> (s N) B
  obj ttI<  : !< (s N) tt
  obj close : !> N A -> !< N A

  end  % let open L ...
end;  % signature BDD_NATDED
```

Within the core language implementation, an interactive top-level similar to that of Prolog is provided in order to pose queries. A query in this context consists of a type, possibly with some free variables. This represents the goal of finding a closed object of the given type by searching through a signature in a depth-first way for appropriate constants which could construct an object of the given type. As in Prolog, this employs a unification algorithm (which postpones certain equations as constraints) and back-chaining.

This only describes the search behavior to a first approximation. In order to make this operational model feasible, the programmer has a certain amount of control over how the search is performed by distinguishing *open* and *closed* families. Intuitively, a proof object and answer substitution may contain free variables of open type, but no free variables of closed type. Put another way: a logic variable of closed type represents a goal (which is instantiated through search); a logic variable of open type on the other hand is instantiated only through unification and thus represents ordinary logic variables in the sense of Prolog. Within the module system, we prefer to call a closed family *dynamic*.

As an example, consider the problem of finding a bounded normal deduction for the proposition $A \Rightarrow A$, where A is a propositional variable. Assuming the families `!<` and `!>` are dynamic, the following query (with a rather arbitrary bound of 3) finds such a deduction.

```
?- {A:o} !< (s (s (s z))) (=> A A).
solved

Query = [A:o] =>I< ([p:{M:nat} !> (s M) A] close (p (s z))).
```

Within the module system, execution of a query is viewed as the process of finding a definition of an object with a declared type. There is an additional mechanism for search control in that the realizations which may be used for search have to be explicitly introduced as dynamic. Since, *a priori*, there are no constants available at the top-level it is convenient to introduce top-level constructs which allow us to declare constants and initial realizations. These come in two flavors: `dynamic` declarations are used for search, while `static` declarations are used only for type-checking. A properly scoped version to introduce dynamic declarations inside of realizors (with keyword `using`) is introduced later in this section.

top	::=	`static` *decl*	static declaration
	|	`dynamic` *decl*	dynamic declaration
	|	...	

The following top-level definitions will pose the same query as the one shown above.

```
static realizor L : MPC_LANG;
dynamic realizor BN : BDD_NATDED (realizor L = L);
open L;
open BN;
solve obj Query : {A:o} !< (s (s (s z))) (=> A A);
```

This example also introduces `solve`, a new form of definition.

defn	::=	`solve` *decl*	initiate search
	|	...	

When processing `solve` *decl* the interpreter tries to find definitions for all the constants declared by *decl*, using objects which are dynamically available. Operationally, logic variables will be created for all free variables and declared constants in *decl* and the Elf interpreter will then search for appropriate instances for these variables in the order of declaration. Upon success (it only looks for first simultaneous solution), the bindings of the logic variables are used to extract appropriate definitions of the declared constants by implicitly quantifying over all remaining logic variables (which must have static type). The `solve` construct cannot be applied to declarations of families: it only searches for definitions of objects.

A top-level declaration of the form `dynamic fam` *id* `:` *kindexp* means that the declared family will be treated as dynamic. That is, search within the scope of the declaration (initiated by `solve`) will make sure that all free variables of dynamic type will be solved. A declaration of the form `dynamic obj` *id* `:` *famexp* means the the declared constant *id* will be available for backchaining search. A dynamic declaration of a realizor applies hereditarily to its constituent declarations (similarly for composition and inclusion).

To illustrate the concept of dynamic declarations and `solve`, we consider a few simple examples.

```
static fam nat : type;
static obj z : nat;
static obj s : nat -> nat;
solve n : nat;
```

will succeed, elaborating the last line to the definition

```
obj n = N : nat;
```

which, in full notation, would be

```
obj n = [N:nat] N : nat -> nat;
```

where the argument to `n` is implicit. In fact, using `n` anywhere subsequently will have the effect of replacing it by a free (anonymous) variable, since it will be expanded to `n _`, which is equal to `_` by the definition of `n`. This is generally the case if the type of a variable to be solved is not dynamic.

```
dynamic fam nat : type;
static obj z : nat;
static obj s : nat -> nat;
solve n : nat;
```

This will fail, since there are no dynamic objects available to construct a term of type `nat`. Since `nat` is dynamic, no free variables of type `nat` are tolerated in the substitution for `n`, and search will fail.

```
dynamic fam nat : type;
dynamic obj z : nat;
dynamic obj s : nat -> nat;
solve n : nat;
```

This will succeed and bind `n` to `z`.

```
dynamic fam nat : type;
static obj z : nat;
dynamic obj s : nat -> nat;
solve n : nat;
```

In an abstract sense, this would fail since there is no closed term of type `nat` which can be found by search over the dynamic objects. In the concrete interpreter, this would loop while constructing incomplete answers of the form `s (s ... (s` N`)))`, for logic variables N of type `nat`.

We now would like to make the relationship between bounded normal deductions and natural deductions explicit. Not surprisingly, this takes the form of a realization of the signature `BDD_NATDED` from the signature `NATDED`, both over the same language `L`. There is no particular difficulty in defining this realization: intuitively, both backwards and forwards provability judgments are interpreted as provability by simply ignoring the bounds.

```
realizor BDD_ND (realizor L : MPC_LANG)
               (realizor Nat : NAT)
               (realizor ND : NATDED (realizor L = L))
               : BDD_NATDED (realizor L = L) (realizor Nat = Nat) =
real
  fam !< N A = ND.! A
  fam !> N A = ND.! A

  obj =>I< P   = ND.=>I ([x:ND.! A] P ([n:Nat.nat] x))
  obj =>E> P Q = ND.=>E P Q
  obj &I<  P Q = ND.&I P Q
  obj &EL> P   = ND.&EL P
  obj &ER> P   = ND.&ER P
  obj ttI<     = ND.ttI
  obj close P  = P
end;  % realizor BDD_ND
```

Next we can combine the search for bounded normal deductions with the interpretation above to construct an interpretation showing that natural deductions are complete with respect to Hilbert deductions. The following realization shows that all Hilbert axioms are provable within our natural deduction calculus, and that Modus Ponens can be realized as a rule of inference. The construction of this realization involves bounded search (with a bound of 5).

```
realizor Natded_Hilbert
  (realizor L : MPC_LANG)
  (realizor Nat : NAT)
  (realizor ND : NATDED (realizor L = L))
  : HILBERT (realizor L = L) =
real
  let
       obj five = Nat.s (Nat.s (Nat.s (Nat.s (Nat.s Nat.z))))
       open L
       open ND
  in

    using realizor BD = BDD_ND (realizor L = L)
                               (realizor Nat = Nat)
                               (realizor ND = ND)
    in
```

```
      solve obj K' : {A:o} {B:o} BD.!< five (=> A (=> B A))
      solve obj S' : {A:o} {B:o} {C:o}
                       BD.!< five (=> (=> A (=> B C)) (=> (=> A B) (=> A C)))
    end

    obj |-    =  !
    obj K     =  K' _ _
    obj S     =  S' _ _ _
    obj ONE   =  ttI
    obj PAIR  =  =>I [x] =>I [y] &I x y
    obj LEFT  =  =>I [x] &EL x
    obj RIGHT =  =>I [x] &ER x

    obj MP    =  =>E

  end  % let solve ...
end;  % realizor Natded_Hilbert
```

New here is the construct `using` with syntax

defn	::=	`using` $\textit{defn}_1$ `in` $\textit{defn}_2$ `end`	use dynamically
	|	...	

For each definition in $\textit{defn}_1$ a corresponding dynamic declaration will be introduced. The constants thus declared are not defined within $\textit{defn}_2$, but upon leaving the scope of the `using` statement, the constants are replaced by the definition given to them in $\textit{defn}_1$. Thus, in contrast to `let`, `using` introduces definitions which are opaque (abstract) within its scope. This yields exactly the right behavior here: while we solve for objects `K'` and `S'`, we use the signature containing `!<` and `!>` for back-chaining search. Once we have constructed the desired objects (representing bounded normal deductions) and leave the scope of `using`, the definitions are applied, interpreting the bounded normal deductions as natural deductions. Those natural deductions are then used directly to provide definitions for axioms `K` and `S` in the Hilbert calculus.

When processing `using` $\textit{defn}_1$ `in` $\textit{defn}_2$ `end`, realizations within $\textit{defn}_1$ are hereditarily marked as dynamic. This is convenient in most circumstances, but it does require that declarations intended for dynamic use and those intended for static use are separated. Normally, the definition of the syntax of an object language will be static, while signatures intended to perform search over the expressions of the object language are explicitly parameterized over a realization of the object language.

6 Signature Relations

The interpretations which can be represented using realizors within the language we have presented so far are limited by the functions which are expressible in the core language. The core language provides λ-abstraction as its only mechanism for the formation of functions. This is no accident—generalizations of the core language to admit, for example, some forms of recursion either render LF type-checking undecidable, or destroy the adequacy of encodings by admitting too many functions. Consider, for example, the rule of implication introduction in the representation of natural deductions:

$$\dfrac{\dfrac{}{A}\,x \quad \vdots \quad B}{A \Rightarrow B} \Rightarrow \mathrm{I}^{x}$$

The premiss of this rule is represented as a function which transforms a deduction of A into a deduction of B, that is

```
  obj =>I : (! A -> ! B) -> ! (=> A B)
```

In order for this to lead to an adequate encoding of natural deductions we must make sure that every term of the form `=>I` P does in fact represent a deduction of B from the assumption A. If λ-abstraction is the only way to form functions this is guaranteed, since P must be equivalent to a term of the form `[`x `: !` A`]` P', where P' is schematic in x. Under generalization, for example, to admit primitive recursion, this is no longer the case.

We call the kind of realizations which can be directly represented as a `realizor` a *uniform realization.* There are a number of reasons why certain interpretations may not be uniform. A realization may have to be defined by general or primitive recursion on the structure of an object. Or a realization may require recursion on the structure of a type. As an example, consider the following attempt to prove the deduction theorem for the Hilbert formulation of minimal propositional calculus.

```
signature DEDTHM (realizor L : MPC_LANG) =
sig
  include HILBERT (realizor L = L)
  obj dedthm : (|- A -> |- B) -> |- (=> A B)
end;
```

We cannot *uniformly* construct a deduction of `|- => A B` from a deduction of `|- B` under the assumption `|- A`. Instead, we must distinguish cases, depending on the form of the deduction of `|- B`. In a hypothetical language extension by a form of primitive recursion, this might be written as following "realizor".

```
"realizor" DedThm (realizor L : MPC_LANG)
                  (realizor H : HILBERT (realizor L = L))
                  : DEDTHM (realizor L = L) =
real
  let  open L  open H  in
  obj dedthm ([x] x)      = MP (MP S K) K
    | dedthm ([x] ONE)    = MP K ONE
    | dedthm ([x] PAIR)   = MP K PAIR
    | dedthm ([x] LEFT)   = MP K LEFT
    | dedthm ([x] RIGHT) = MP K RIGHT
    | dedthm ([x] K)      = MP K K
    | dedthm ([x] S)      = MP K S
    | dedthm ([x] MP (P x) (Q x)) = MP (MP S (dedthm P)) (dedthm Q)
  end
end;
```

Does the fact that this is not a legal `realizor` mean that we cannot use the deduction theorem for the construction of deductions? Fortunately not! While we do not have a way of internally verifying through a realization that the deduction theorem is an admissible rule of inference, we can still implement the computational content of the realization above as a relation between deductions.

```
signature DEDTHM (realizor L : MPC_LANG)
                 (realizor H : HILBERT (realizor L = L)) =
sig
  let  open L  open H  in

  fam ded : (|- A -> |- B) -> |- (=> A B) -> type

  obj ded_ID    : ded ([x] x)     (MP (MP S K) K)
  obj ded_K     : ded ([x] K)     (MP K K)
  obj ded_S     : ded ([x] S)     (MP K S)
  obj ded_ONE   : ded ([x] ONE)   (MP K ONE)
  obj ded_PAIR  : ded ([x] PAIR)  (MP K PAIR)
  obj ded_LEFT  : ded ([x] LEFT)  (MP K LEFT)
  obj ded_RIGHT : ded ([x] RIGHT) (MP K RIGHT)
  obj ded_MP    : ded ([x] MP (P x) (Q x)) (MP (MP S P') Q')
                     <- ded P P' <- ded Q Q'
  end
end;
```

We have written the final clause using the left-arrow notation in order to emphasize the operational reading of this signature. More efficient versions of this algorithm (known as bracket abstraction) can easily be implemented. For example, `ded_K` through `ded_RIGHT` can be combined into one clause. Type-checking of the above signature guarantees that if a query of the form `solve` c `:` `ded` P Q succeeds, then Q will indeed be a deduction of the appropriate implication. The fact that such a query will *always* succeed whenever P is closed and Q is a free variable is easily checked by hand, but not internally representable.

As a related example, consider the implementation of a translation from natural deductions into Hilbert deductions. This relation, too, defines a (non-uniform) realization and takes advantage of the deduction theorem.

```
signature HILBERT_NATDED
  (realizor L : MPC_LANG)
  (realizor H : HILBERT (realizor L = L))
  (realizor N : NATDED (realizor L = L)) =
sig
  let  open L  open H  open N  in

  realizor D : DEDTHM (realizor L = L) (realizor H = H)

  fam ndh       : ! A -> |- A -> type

  obj ndh_=>I   : ndh (=>I PP) Q
                    <- ({x}{y} ndh x y
                              -> D.ded ([z] y) (MP K y)
                              -> ndh (PP x) (PP' y))
                   <- D.ded PP' Q
  obj ndh_=>E    : ndh (=>E P Q) (MP P' Q') <- ndh P P' <- ndh Q Q'
  obj ndh_&I     : ndh (&I P Q) (MP (MP PAIR P') Q') <- ndh P P' <- ndh Q Q'
  obj ndh_&EL    : ndh (&EL P) (MP LEFT P') <- ndh P P'
  obj ndh_&ER    : ndh (&ER P) (MP RIGHT P') <- ndh P P'
  obj ndh_ttI    : ndh ttI ONE

  end
end;
```

Note how we used `D.ded` as an auxiliary judgment in the implication introduction rule. According to our signature specifying inference rules for natural deduction, `PP` has type `! A -> ! B` for some `A` and `B`. To translate `PP` we make the local assumption `! A` labelled `x` and then translate `PP x` (of type `! B`). The corresponding Hilbert deduction will use a corresponding hypothesis `y : |- A`. We make this correspondence explicit through the assumption `ndh x y`. This translation yields a deduction `PP' : |- A -> |- B`, but we need to construct a deduction of `|- (=> A B)`. This is where we apply bracket abstraction (the computational content of the deduction theorem), which is done by calling `D.ded PP' Q`. It remains to explain the second local assumption in this rule, namely `D.ded ([z] y) (MP K y)`. Recall, that bracket abstraction as implemented in the signature `DEDTHM` applies only to deductions from exactly one hypothesis. Here, however, we introduce a new, temporary hypothesis `y` which is not accounted for in the signature `DEDTHM`. Therefore we have to specify the behavior of the bracket abstraction algorithm on `y`. The obvious course of action is the same as for the axioms: simply return `MP K y`.

This example illustrates the use of declared realizations within a signature compared with explicit parametrization or inclusion. The signature `HILBERT_NATDED` intrisically contains the realization of the deduction theorem. As a consequence, whenever `HILBERT_NATDED` is used dynamically, the judgment `D.ded` will also be available dynamically. At the same time, unlike when `include` is employed, the name space of the realization `D` remains separate.

The computational contents of relations such as the one above is sufficient to aid in the construction of other realizations. We now return to an earlier example, showing how the realization establishing the symmetry of conjunction could have been established by translating a natural deduction.

```
realizor SYMM_&' (realizor L : MPC_LANG)
                 (realizor H : HILBERT (realizor L = L))
                 (realizor N : NATDED (realizor L = L))
                 (realizor HN : HILBERT_NATDED (realizor L = L)
                                   (realizor H = H)
                                   (realizor N = N))
                 : HILBERT (realizor L = SYMM_&_LANG (realizor L = L)) =
real
  let  open L  open H  in

  fam |-    = |-
  obj K     = K

  obj S     = S
  obj ONE   = ONE

  using realizor HN = HN
  in
    solve obj H   : |- (=> A (=> B (& B A)))
          obj NDH : HN.ndh (N.=>I [x] N.=>I [y] (N.&I y x)) H
  end

  obj PAIR  = H

  obj LEFT  : |- (=> (& B A) A) = RIGHT
  obj RIGHT : |- (=> (& B A) B) = LEFT

  % Inference Rule
  obj MP    = MP

  end
end;  % realizor SYMM_&'
```

The two objects to be solved for, `H` and `NDH`, are solved simultaneously. Since `|-` is not dynamic, it is instantiated only through the process of solving for `NDH`, whose type is dynamic. However, we need to declare `H` here so we can use its binding as the definition for `PAIR`.

The natural deduction we give explicitly here could also have been generated through search for a bounded normal deduction and its interpretation as a natural deduction. We leave it to the reader to write out the appropriate `realizor`.

7 Related and Future Work

The design of the module system owes a great deal to the ML module system [14, 17]. We have replaced *sharing equations* by explicit parametrization, at the cost of some verbosity, but with the gain of semantic simplicity. While the similarities to the ML module system are striking in some respects, emphasis has shifted significantly. In ML, definitions within *structures* (*realizations*, in our terminology) give rise to computation, while in our setting of logic programming the signatures themselves are given an operational interpretation. Realizations only interpret the result of computations, which consist of a form of search over signatures.

The problem of modularity in logical frameworks has previously been addressed in various typed λ-calculi. De Bruijn's *telescopic mappings* [3], for example, provide for a first-class notion of contexts. Along similar lines, a type-theoretic calculus with explicit contexts called DEVA has been developed and applied to a number of interesting examples by de Groote [4] and Weber [25]. Our module calculus can be seen as a higher-level language which could be compiled into a lower-level type theory such as DEVA. The main difficulty in this compilation process is to account for constant names (which are *not* subject to α-conversion, in contrast to bound variables names) and the implicit coercions which take place when signatures or realizations are instantiated.

Much more powerful, impredicative type theories have been investigated by Luo [12] and are implemented within the LEGO system [13]. The basis for this work is the Calculus of Constructions [2] which is not intended as a logical framework, but a type theory in which constructive mathematics could be directly formalized and reasoned about internally. Explicit theory structuring is achieved through Σ-types. Again, this provides no real name-space management—it has more the flavor of an "abstract syntax" for a module calculus.

None of the calculi mentioned above give an integral treatment to search control, which is provided within our framework. In the context of more traditional logic programming, this has been in addressed by Miller [16], but he does not provide any form of name-space management besides local declarations through existential quantification. Sannella and Burstall [23] introduced a notion of modular presentation for LCF theories, with an associated search procedure that takes advantage of the structure of a theory presentation to cut down the search space. These methods were generalized to an arbitrary logic, considered as an abstract family of consequence relations, by Harper, Sannella, and Tarlecki [10, 9], where their behavior under representation in a logical framework is also considered. Local declarations introduce problems of adequacy that are rectified by *ad hoc* techniques that segregate types that represent judgements from other types. Subsequently, Gardner [7] introduced a more refined notion of framework that enforces such a segregation; it seems plausible that in this setting the aforementioned problems of adequacy do not arise. The possibility of introducing local declarations in the present setting requires further investigation.

Sannella and Wallen's proposal [24] for a module system for Prolog bears certain similarities to ours as both have been inspired by ML. In their system, signatures provide declarations of arities for predicate and function symbols and structures contain clauses.

While this is ongoing research, we feel confident that we can provide a natural type-theoretic account of the module system for Elf as presented here. There are a number of additional features for which this is less clear, and which we have omitted at present. We will reconsider them in the light of practical experience with a planned prototype implementation. The principal features we have in mind are sharing equations, higher-order realizations, local declarations, inclusion of individual declarations, and declaration of infix operators. Finally, we are considering whether the language of realizations could be strengthened by some form of primitive recursion in order to allow non-uniform realizations without destroying the basic properties of the LF type theory. We feel that the natural separation of the module level from the object level provides us with an opportunity for such an extension which would not be available in the core calculus, the LF type theory.

A Syntax

Identifiers

id			term identifier
sigid			signature identifier
realid			realizations identifier
longrealid	$::=$	$\langle$*longrealid*.$\rangle$*realid*	qualified realization identifier
longid	$::=$	$\langle$*longrealid*.$\rangle$*id*	qualified term identifier

Core Language

kindexp	::=	`type`	Type
	|	`{`*id* ⟨`:`*famexp*⟩`}` *kindexp*	$\Pi x{:}A.\ K$
	|	*famexp* `->` *kindexp*	$A \to K$
	|	`(`*kindexp*`)`	
famexp	::=	*longid*	a
	|	`{`*id* ⟨`:`$famexp_1$⟩`}` $famexp_2$	$\Pi x{:}A.\ B$
	|	`[`*id* ⟨`:`$famexp_1$⟩`]` $famexp_2$	$\lambda x{:}A.\ B$
	|	*famexp objexp*	$A\,M$
	|	$famexp_1$ `->` $famexp_2$	$A \to B$
	|	$famexp_2$ `<-` $famexp_1$	$B \leftarrow A$
	|	`_`	
	|	*famexp* `:` *kindexp*	
	|	`(`*famexp*`)`	
objexp	::=	*longid*	c or x
	|	`[`*id* ⟨`:`*famexp*⟩`]` *objexp*	$\lambda x{:}A.\ M$
	|	$objexp_1$ $objexp_2$	$M\,N$
	|	`_`	
	|	*objexp* `:` *famexp*	
	|	`(`*objexp*`)`	

Declarations and Signatures

sigexp	::=	`sig` *decl* `end`	encapsulation
	|	*sigid* ⟨$inst_1 \ldots inst_n$⟩	signature instantiation
parm	::=	`(`*decl*`)`	parameter declaration
decl	::=	`fam` *id* `:` *kindexp*	family
	|	`obj` *id* `:` *famexp*	object
	|	`realizor` *realid* `:` *sigexp*	realizor
	|	$decl_1$ $decl_2$	composition
	|	`let` *defn* `in` *decl* `end`	local definition
	|	`include` *sigexp*	inclusion

Definitions and Realizations

realexp	::=	`real` *defn* `end`	encapsulation
	|	*longrealid* $\langle inst_1 \ldots inst_n\rangle$	instance
inst	::=	`(`*defn*`)`	parameter instantiation
defn	::=	`fam` *id* $\langle$`:` *kindexp*$\rangle$ `=` *famexp*	family
	|	`obj` *id* $\langle$`:` *famexp*$\rangle$ `=` *objexp*	object
	|	`realizor` *realid* $\langle$:*sigexp*$\rangle$ `=` *realexp*	realizor
	|	$defn_1$ $defn_2$	composition
	|	`let` $defn_1$ `in` $defn_2$ `end`	local definition
	|	`open` *longrealid* $\langle$:*sigexp*$\rangle$	inclusion
	|	`using` $defn_1$ `in` $defn_2$ `end`	use dynamically
	|	`solve` *decl*	initiate search

Top-Level

top	::=	`signature` *sigid* $\langle parm_1 \ldots parm_n\rangle$ `=` *sigexp*	signature binding
	|	`realizor` *realid* $\langle parm_1 \ldots parm_n\rangle$ `=` *realexp*	realizor binding
	|	`static` *decl*	static declaration
	|	`dynamic` *decl*	dynamic declaration
	|	*defn*	pervasive definition
	|	top_1 `;` top_2	composition

References

[1] Thierry Coquand. An algorithm for testing conversion in type theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, 1991.

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

[3] N. G. de Bruijn. Telescopic mapping in typed lambda calculus. *Information and Computation*, To appear.

[4] Philippe de Groote. Nederpelt's calculus extended with a notion of context as a logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 69–86. Cambridge University Press, 1991.

[5] 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.

[6] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: An interactive mathematical proof system. In M. E. Stickel, editor, *10th International Conference on Automated Deductions*, pages 653–654, 1990.

[7] Phillippa Gardner. Lecture given at the 1991 Workshop on Logical Frameworks. Edinburgh. May, 1991.

[8] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, To appear. A preliminary version appeared in *Symposium on Logic in Computer Science*, pages 194–204, June 1987.

[9] Robert Harper, Donald Sannella, and Andrzej Tarlecki. Logic representation. In *Proceedings of the Workshop on Category Theory and Computer Science*. Springer-Verlag, September 1989.

[10] Robert Harper, Donald Sannella, and Andrzej Tarlecki. Structure and representation in LF. In *Fourth Annual Symposium on Logic in Computer Science*, pages 226–237. IEEE, June 1989.

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

[12] Zhaohui Luo. ECC, an extended Calculus of Constructions. In *Fourth Annual Symposium on Logic in Computer Science*, pages 386–395. IEEE Computer Society Press, June 1989.

[13] Zhaohui Luo, Robert Pollack, and Paul Taylor. How to use LEGO. Technical Report LFCS–TN–27, Laboratory for Foundations of Computer Science, University of Edinburgh, 1989.

[14] David MacQueen. Using dependent types to express modular structure. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, pages 277–286. ACM SIGPLAN/SIGACT, 1986.

[15] Spiro Michaylov and Frank Pfenning. Natural semantics and some of its meta-theory in Elf. Technical Report MPI–I–91–211, Max-Planck-Institute for Computer Science, Saarbrücken, Germany, August 1991.

[16] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6(1-2):57–77, January 1989.

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

[18] 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. Also available as Ergo Report 89–067, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[19] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[20] Randy Pollack. Implicit syntax. In G. Huet and G. Plotkin, editors, *Proceedings of the First Workshop on Logical Frameworks, Antibes*, pages 421–434. Preliminary Version, May 1990.

[21] 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.

[22] 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.

[23] D. Sannella and R. Burstall. Structured theories in LCF. Technical Report CSR-129-83, University of Edinburgh, 1983.

[24] D. T. Sannella and L. A. Wallen. A calculus for the construction of modular Prolog programs. *Journal of Logic Programming*, To appear. A preliminary version appears in the Proceedings of the 4th Symposium on Logic Programming, San Francisco, September 1987.

[25] Matthias Weber. *A Meta-Calculus for Formal System Development.* R. Oldenbourg Verlag, München/Wien, 1991.