A Proposal for Modules in λProlog[*]

Dale Miller

Computer Science Department
University of Pennsylvania
Philadelphia, PA 19104-6389 USA
dale@saul.cis.upenn.edu

Abstract. Higher-order hereditary Harrop formulas, the underlying logical foundation of λProlog [NM88], are more expressive than first-order Horn clauses, the logical foundation of Prolog. In particular, various forms of scoping and abstraction are supported by the logic of higher-order hereditary Harrop formulas while they are not supported by first-order Horn clauses. Various papers have argued that the scoping and abstraction available in this richer logic can be used to provide for modular programming [Mil89b], abstract data types [Mil89a], and state encapsulation [HM90]. None of these papers, however, have dealt with the problems of *programming-in-the-large*, that is, the essentially linguistic problems of putting together various different textual sources of code found, say, in different files on a persistent store into one logic program. In this paper, I propose a module system for λProlog and shall focus mostly on its static semantics.

1 Module syntax should be declarative

Several modern programming languages are built on declarative, formal languages: for example, ML and Scheme are based on the λ-calculus and Prolog is based on Horn clauses. Initial work on developing such languages was first concerned with programming-in-the-small: problems with programming-in-the-large were attacked later. At that point, a second language was often added on top of the initial language. For example, parsing and compiler directives, such as `use`, `import`, `include`, and `local`, were added. This second language generally had little connection with the original declarative foundation of the initial language: it was born out of the necessity to build large programs and its function was expediency. The meaning of the resulting hybrid language is often complex since it loses some of its declarative purity.

Occasionally, programming design is inflicted with what we may call the "recreating the Turing machine" syndrome. Turing machines were important because they were the first formal system that obviously computed and were clearly easy to implement (at least the bounded version of them). They have not been considered seriously as programming languages for several reasons, including the

[*] An earlier version of this paper was first presented at the 1992 Workshop on λProlog. The work presented here is supported in part by grants ONR N00014-88-K-0633, NSF CCR-91-02753, and DARPA N00014-85-K-0018.

difficulty of understanding and reasoning about transition tables. Often the development of modular constructions in programming languages follows a similar path: it is generally easy to develop a language for programming-in-the-large that obviously separates and hides details and for which efficient implementations are possible. Often, however, it is difficult to reason about the meaning of the resulting language.

In order to avoid this syndrome we should ask that any proposal for programming-in-the-large meet several of the following high-level criteria.

1. There should be a non-trivial notion of the equivalence of modules that would guarantee that a module within a larger program can be replaced by an equivalent module without making an impact on the behavior of a larger program. This property is sometimes called *representation independence* (see Section 3). Within logic programming, there might be various versions of this requirement: for example, it may be fine on some occasions to allow equivalent modules to produce the same answers although they may produce them with different multiplicities.
2. Constructs for programming-in-the-large should not complicate the meaning of the underlying, declarative language. For example, in Prolog, a particular challenge is to get modular programming to work smoothly with higher-order programming (uses of `call/1`).
3. Rich forms of abstraction, hiding, and parameterization should be possible. Within logic programming, we can ask for the ability to abstract over individuals, functions, predicates, and collections of program clauses.
4. Modules should support transitions from specification to implementation. In particular, it would be desirable to have a rich calculus of transformations of program clauses and modules, including, for example, partial evaluation, fold/unfold, and even compilation.
5. Important aspects of a module's meaning should be available and verified without examining the module in detail. Notions of interfaces often support this property.
6. The additional syntax for programming-in-the-large should also be readable, natural, and support separate compilation and re-usability.

The success of a proposal for modular programming should not be judged simply on its obviousness or easy of implementation: it should also be judged on its ability to support a large number of properties such as these.

There are some logical systems that can be used as a basis of logic programming and that contain natural notions of scope for program clauses and constants. For example, the logic of *hereditary Harrop formulas*, parts of which were developed independently by Gabbay and Reyle [GR84], McCarty [McC88a, McC88b], and Miller [Mil86, Mil89b, Mil90], allows for a simple stack-based structuring of the runtime program and set of constants. The modal logic of Giordano, Martelli, and Rossi [GMR88] provides an interesting variation on hereditary Harrop formulas that has a different runtime structuring of programs. A recent linear logic refinement of hereditary Harrop formulas by Hodas and Miller

[HM94] modifies the stack-based discipline of programs by allowing some program clauses to be deleted once they are used within a proof. For a recent survey of proposals for modular programming in logic programming, see [BLM94].

The approach we shall take in this paper to developing a declarative modular programming language is to reduce programming-in-the-large to programming-in-the-small in such a way that modular programming can be explained completely in terms on the logical connectives of the underlying language. That is, a linked collection of modules would be mapped to a (possibly large) collection of (possibly large) formulas. Furthermore, we would like the combinators for building modules to correspond closely to logical connectives. The *static semantics* of a collection of modules is specified by describing how such modules denote a collection of constants and program clauses. The *dynamic semantics* of a collection of modules is specified by describing the collection of goal formulas that can be proved from them. Given the richness of hereditary Harrop formulas and their variants, the main challenge in specifying the static semantics of modules appears to be determining the scope and types of constants.

2 A specific module proposal

We shall now turn to a specific proposal for modules for λProlog. Since the underlying logic of λProlog is that of the intuitionistic (actually minimal) theory of hereditary Harrop formulas, we shall consider how modules can be mapped into such formulas. It would be interesting to consider a similar mapping into either the modal or linear logic variants of these formulas mentioned above. We shall not, however, consider these other variations here.

2.1 General comments

λProlog extends first-order Horn clauses by allowing higher-order quantification and by allowing richer collections of logical connectives. As it turns out, the main scoping primitives for the module facility proposed here do not come from the higher-order quantification: in fact, the propositional logic fragment of λProlog supports the stack-based treatment of programming clauses. Higher-order quantification does play a role, however, in providing scope for predicate and function symbols as well as in providing for higher-order programming (an important abstraction separate from the module proposal here).

Both the proof theoretic and model theoretic treatments of λProlog's foundation treat a program as a pair containing a signature and a set of clauses. For example, the proof theoretic treatment of λProlog given in [Mil90] uses sequents of the form $\Sigma;\mathcal{P} \longrightarrow G$, where Σ is a signature (a collection of typed constants) and $\mathcal{P}$ is a set of Σ-formulas (closed formulas all of whose non-logical constants are contained in Σ). Similarly, a canonical model for a large fragment of the logic underlying λProlog can be given as a Kripke model where possible worlds are pairs $\langle\Sigma,\mathcal{P}\rangle$, where Σ is a signature and $\mathcal{P}$ is a set of Σ-formulas [Mil92]. Thus it will not be surprising that the module proposal presented will be based also on pairing signatures with program clauses. Even if λProlog was not a typed language signatures would be important since the set of constants available to a computation changes, and describing how that set of constants changes would make use of a notion of signature similar to that used here. Gunter [Gun91] also makes use of signatures in developing a module calculus for λProlog.

What follows is just a draft of a proposal: much of it has not yet been debated by those currently using and building implementations of λProlog. Also, most experience with λProlog has been with small programs. Few people have yet had experience with large λProlog programs. This proposal is hopefully another step in determining a viable solution to programming-in-the-large in the logic programming setting. This proposal should also be relevant to modular programming for logic programming based only on first-order Horn clauses: several aspects of this proposal can make sense in that logically weaker setting.

2.2 Persistent store

A persistent store, such as the Unix file system, is (currently) outside the scope of the logical core of λProlog and Prolog. Thus some non-logical predicates are required at the core of our module facility to deal with the persistent store. In particular, the predicate

```
type load  string -> o
```

performs a side-effect: it is used to reflect some of the persistent store into the space of meaningful λProlog objects. As edits are done on files, new calls to `load` are needed to update these objects. An attempt to prove the atom `load name` takes the string `name` as a reference to an actual file. The resolution of this string into a file can be done in possibly many ways. The method used in LP2.7 [MN88] is to maintain a list of Unix path names and to search in them for a file whose name is `name` augmented with ".mod". If such a file is found, then it is parsed and type checked. Other methods to resolve the string `name` with a file are possible.

2.3 Kinds and types

λProlog allows type constructors in order to build compound types. One such type constructor `->` is for building "function space" and it is the only one that is built into λProlog. Other type constructors can be declared via the **kind** declaration. For example,

```
kind bool type.
kind list type -> type.
kind pair type -> type -> type.
```

As this example show, kind expressions are simple expressions involving only `type` and `->`, with the later nesting only to the right (only "first-order kinds" are allowed). Qualifying a type constructor with a non-negative integer (0 instead of `type`, 1 instead of `type -> type`, etc.) could also have worked here.

Types will be used to qualify constants. Types are any first-order term structure built from type variables and type constructors. The presence of type variables provides λProlog with a degree of polymorphism. Type variables are tokens within type expressions that have an initial uppercase letter. The following are some declarations.

```
type nil     list A.
type ::      A -> list A -> list A.
infixr ::    4.
type append  list A -> list A -> list A -> o.
type memb    A -> list A -> o.
```

λProlog has numerous built-in types, including type `o`, the type of λProlog formulas. Lists are given polymorphic type (as in ML) and are built from the empty list `nil` and the (infix) list constructor `::`. The term `(1::2::3::nil)` is of type `(list int)` and would be written in Prolog as `[1,2,3]`.

The subsumption relation on types is that familiar from first order logic: a type is subsumed by another type if the first is a substitution instance of the second.

2.4 Static semantics for types and terms

We shall assume that types are properly formed (they respect kind declarations) and that formulas and terms are well typed. See [NP92] for a fuller discussion of this aspect of static semantics.

2.5 Signatures

Signatures are lists of tokens assigned kinds and types, and are denoted by the syntactic variable Σ. The same token can be given a type and a kind. Infix declarations are also stored as members of signatures. The following is an example of a signature.

```
kind   list          type -> type.
type   ::            A -> list A -> list A.
infixr ::            4.
type   nil           list A.
type   memb, member  A -> list A -> o.
type   append, join  list A -> list A -> list A -> o.
```

The two keywords `infixl` and `infixr` declare that the following token is to be parsed and printed as an infix symbol, with default to either the left or the right. The number following the token must be in the range 0 to 9, and denotes the grouping priority of that symbol. A formula is a Σ-formula if it is a correctly typed, closed formula all of whose non-logical constants are from

Σ. Since modules are collections of formulas, we shall use signatures to qualify (type) modules.

It will be useful to have *signature descriptions* to represent possibly long signatures. For this, we shall use the keywords `signature`, `type`, `kind`, `infixl`, `infixr`, `accumulate`, `local`, and `localkind`. The keyword `signature` is used to name a signature and the keywords `type`, `kind`, `infixl`, and `infixr` are used simply to enumerate the members of a signature. For example, below is a simple signature.

```
signature lists.
kind   list          type -> type.
type   ::            A -> list A -> list A.
infixr ::            4.
type   nil           list A.
type   memb, member  A -> list A -> o.
type   append, join  list A -> list A -> list A -> o.
```

Three other keywords are also allowed to describe signatures. The keyword `accumulate` takes a list of signature names: its intended meaning is to insert the listed signatures. The two keywords `local` and `localkind` are used to limit the scope of types and kinds so that they are actually not part of this signature. The `local` keyword can take a type declaration as an optional third argument; similarly with `localkind`. The following is another example of a signature (assuming that the one listed above has been defined.)

```
signature rev.
accumulate lists.
type reverse list A -> list A -> o.
local revaux list A -> list A -> list A -> o.
local join.
```

Constants can be given multiple types within the same module or within accumulating chains of modules. It is an error if these types are not comparable via subsumption. Otherwise, the type assumed is the least general of those types.

Signature descriptions are *elaborated* into signatures using the following rules. First, eliminate all `accumulate` keywords by replacing them with the signatures they name. In doing this, if a constant is given two infix declarations, then it is an error if those two declarations are not identical. Second, `local` can be dropped by deleting it and any constant of the same name in the accumulated signature. If `localkind` is present, then first check to see if there are constants in the signature that have a type containing this type constructor. If so, produce an error. Otherwise, simply drop this declaration.

The notion of *signature containment* is given simply as follows: Σ_1 is contained in Σ_2 if

– for every constant in Σ_1 given a kind, that constant is given the same kind in Σ_2,

– for every constant in Σ_1 given a type τ, that constant is given a type in Σ_2 that subsumes τ, and
– for every constant in Σ_1 given an infix declaration, that constant is given the identical infix declaration in Σ_2.

This notion of signature containment will be needed for defining equal signatures and for a certain kind of dynamic qualification of modules (see subsection 2.10).

The `rev` signature above thus elaborates to the following signature:

```
signature rev.
kind   list          type -> type.
type   ::            A -> list A -> list A.
infixr ::            4.
type   nil           list A.
type   memb, member  A -> list A -> o.
type   append        list A -> list A -> list A -> o.
type   reverse       list A -> list A -> o.
```

We shall assume that there is a special system signature that contains declarations for all logical and built-in constants of a given λProlog system. For example, it would contain the fact that the reverse implication symbol `:-` is an infix symbol associating to the left and that it has type `o -> o -> o`.

2.6 Module syntax

Modules will be built from kinds, types, and program clauses using the following keywords: `type`, `kind`, `infixr`, `infixl`, `local`, `localkind`, `module`, `accumulate`, and `import`. The meaning of `type`, `kind`, `infixr`, and `infixl`, are as they were for signature descriptions. The keyword `module` names a module (similar to the keyword `signature`). The keywords `local` and `localkind` provide scope to constants and type constructors (respectively) within a module: the dynamic semantics of `local` will be interpreted as an existential quantifier, as described in [Mil89a]. The keywords `accumulate` and `import` will be described further below. It is possible for a formula to explicitly mention the name of a module. This is done using the "module implication" construction. For example, the expression `mod ==> G` denotes the formula `E => G`, where the module `mod` elaborates to the formula `E`. There is also a syntax for "qualified module implications" written as `===> mod sig G` where `mod` is a module name and `sig` is a signature name and `G` is a goal formula. This syntax is used in the case that `mod` is not known at parse-time: the module that eventually finds it way into this position must have a signature that is contained in the signature named by `sig` – the latter must be known at parse-time. Thus, it is possible at this point to have a run-time signature error, which would force computation to quit. Module implications will be described more below.

Although only the keyword `module` must appear at the front of a module, for the convenience of parsing and reading modules, we assume that it is an error if a declaration of a constant appears after the first occurrence using that constant.

All declarations are global within a module. Figure 1 contains two examples of modules.

```
module lists.

kind list          type -> type.
type ::            A -> list A -> list A.
type nil           list A.
type memb, member  A -> list A -> o.
type append, join  list A -> list A -> list A -> o.
infixr  ::  4.

memb X (X::L).
memb X (Y::L) :- memb X L.
member X (X::L) :- !.
member X (Y::L) :- member X L.
append nil K K.
append (X::L) K (X::M) :- append L K M.
join nil K K.
join (X::L) K M :- memb X K, !, join L K M.
join (X::L) K (X::M) :- join L K M.


module rev.

accumulate lists.
type  reverse  list A -> list A -> o.
local rev      list A -> list A -> list A -> o.
local join.

reverse L K :- rev L K nil.
rev nil K K.
rev (X::L) K (X::Acc) :- rev L K Acc.
```

Fig. 1. The `lists` and `rev` modules.

2.7 Static semantics for modules

The static semantics of modules is used to determine which signature and program clauses are intended by a given module name or module description. Since we are attempting to reduce modules to formulas, recursion between modules is not allowed: that is, if `mod1` imports or accumulates `mod2` then `mod2` can not import or accumulate, either directly or indirectly, module `mod1`.

A signature description is built from a module as follows.

1. Keep all `type`, `kind`, `local`, and `localkind` declarations as they are.
2. All module names used with `import` and `accumulate` keywords, as well as all module names used in module implications have their signatures accumulated. As far as signatures are concerned, importing is the same as accumulation. The difference between these two declarations appears in their effect on the construction of program clauses.
3. If the "qualified module implication" `(===> mod sig) G` is used, then the signature named by `sig` is accumulated instead of the module `mod`.

Notice that it is possible for `local` and `localkind` to provide scope to constants that are brought into a module via `import`, `accumulate`, `==>`, and `===>` (as in Figure 1). If `import` or `accumulate` is used in a module and there is no corresponding module with the correct name, then a signature with that name is used. Thus modules without clauses can simply be written as signatures.

The static semantics of the `import` keyword construction is a bit more involved than that for `accumulate` and follows closely the description given in [Mil89b] and implemented in LP2.7 and eLP [EP89]. If a module `mod1` contains the line

```
import mod2 mod3.
```

then the modules `mod2` and `mod3` are made available (via implications) during the search for proofs of the body of clauses listed in `mod1`. Thus, if the formulas E_2 and E_3 are associated with `mod2` and `mod3`, then a clause $G \supset A$ listed in `mod1` is elaborated to the clause $((E_2 \wedge E_3) \supset G) \supset A$. The fact that this gives a sensible dynamic semantics is explained below.

Notice that a module denotes both a set of program clauses and a signature. The signature that is inferred from a module can be used as an interface: when parsing and compiling modules, only the signature of an accumulated or imported module need be read.

2.8 Environment support

The process of parsing a module will also be accompanied by type checking and type inference. In particular, a file containing a module need not explicitly attribute a type to all constants and variables. In this case, the programming environment must be able to infer a reasonable type for the undeclared constants. Type inference can be done much as it is in ML: see [NP92] for more discussion on possible type inference procedures for λProlog.

Signature checking and inference will also need to be done by the environment. Checking involves making certain that when modules are accumulated and imported, constants are not given incomparable types and declarations in two different signatures.

2.9 Dynamic semantics for modules

I shall assume that the reader is already familiar with the operational (dynamic) semantics of hereditary Harrop formulas, in particular, with the meaning of implications and universal quantifiers in goals. For a description of this aspect of these formulas, see any one of the following papers: [BLM94, Mil90, MNPS91, NM88].

Although the meaning of the `accumulate` keyword is simple, it is not present in either LP2.7 or eLP. It is similar to the `use` directive of Prolog/Mali. If a module `mod1` contains the line

```
accumulate mod2 mod3.
```

then it is intended that the program clauses in `mod2` and `mod3` are available at the end of the list of program clauses listed explicitly in `mod1`. Thus, when selecting clauses in this module for backchaining over, accumulated clauses are selected after those listed in the module.

A description of the `import` keyword benefits from some recent work on provability in intuitionistic logic. For example, both Hudelmaier [Hud89] and Dyckhoff [Dyc92] have demonstrated that the implication-left rule (of Gentzen's formulation of sequent calculus [Gen69]) can be refined with respect to effective proof search. For example, the implication-left rule can be split into several cases depending on the form of the implication. The following is one of these rules:

$$\frac{\Sigma; \mathcal{P}, E, G \supset D \longrightarrow G \qquad \Sigma; \mathcal{P}, D \longrightarrow G'}{\Sigma; \mathcal{P}, (E \supset G) \supset D \longrightarrow G'}.$$

Consider the case when the formulas D and G' are the same atomic formula A (as is the case with backchaining):

$$\frac{\Sigma; \mathcal{P}, E, G \supset A \longrightarrow G}{\Sigma; \mathcal{P}, (E \supset G) \supset A \longrightarrow A}.$$

Notice that the formula $(E \supset G) \supset A$ could be the result of importing a module E into a module containing the clause $G \supset A$. Notice that backchaining on a clause in this module provides an operational reading of importing: the imported module is added to the current clauses along with the un-elaborated clauses from the initial module.

A generalization of this inference rule would be the following:

$$\frac{\Sigma; \mathcal{P}, E, \wedge_{i=1}^{n}(G_i \supset A_i) \longrightarrow G_j}{\Sigma; \mathcal{P}, \wedge_{i=1}^{n}((E \supset G_i) \supset A_i) \longrightarrow A}$$

where A_j is equal to A, for some $j = 1, \ldots, n$. The completeness of this rule can be found in [KNW93].

In the above inference rule, assume that the formula E is of the form $\exists \bar{x}.D$ where the list of typed, bound variables $\bar{x}$ are not in the signature Σ. This inference rule could then be modified to be

$$\frac{\Sigma, \bar{x}; \mathcal{P}, D, \wedge_{i=1}^{n}(G_i \supset A_i) \longrightarrow G_j}{\Sigma; \mathcal{P}, \wedge_{i=1}^{n}((E \supset G_i) \supset A_i) \longrightarrow A}.$$

Thus, backchaining using a clause contained in a module that imports a second module containing local constants causes the imported module's local constants (here the variables $\bar{x}$) and clauses (here the formulas D) to be loaded to the current signature and program.

An important aspect of λProlog's operational semantics is given by what is called the AUGMENT search rule: to prove the goal $D \supset G$ from the signature Σ and program $\mathcal{P}$, attempt to prove G from the signature Σ and the augmented program $\mathcal{P} \cup \{D\}$. An important variation of the AUGMENT search rule is presented in [KNW93] where the AUGMENT search rule is modified to be the AUGMENT' search rule. This new rule is used only when the formula D is a module name and not for more general formulas: thus an operational (but not declarative) distinction between programming-in-the-large and small arises. The AUGMENT' rule essentially says that if the current program space already contains a module, that module should not be added again to the current program: that is, there should be at most one copy of a module in the current program space at a time. The goal `mod ==> mod ==> G` is operationally the same as `mod ==> G`. Such an optimization is unlikely at the level of formulas because of the cost of checking duplicates and because of the following example. Consider a goal of the form `(p a) => (p X) => G`, where `X` is a logical variable. If we were to check to see if `(p X)` is in the current context, we must decide whether or not to allow the unification of `X` with `a`. If such unification is allowed, then a new source of incompleteness is introduced: for example, if `G` above is `(p b)`, then that query would not be proved although it is provable. If unification is not done, then it is possible for two copies of a clause to appear in the program space (let `G` be `X = a`), the very thing that we were attempting to avoid. Thus, it seems sensible to avoid dealing with this situation and apply AUGMENT' only to module names.

2.10 Questions and additional features

Below are some questions and possible additional features that could be incorporated in the module system sketched above.

Parametric modules. When a module is defined using the `module` keyword, it might be possible to add to it a signature over which that module is parametric. An example could be given as follows.

```
module {quicksort kind Aty     type.
                  type Order Aty -> Aty -> o}.

type      qsort  list Aty -> list Aty -> o.
local     split  Aty -> list Aty -> list Aty -> list Aty -> o.
import    lists.

qsort nil nil.
qsort (X::L) K :- split X L Low High, qsort Low R,
                  qsort High S, append R (X::S) K.
```

```
split X (Y::L) (Y::K) M :- Order X Y, !, split X L K M.
split X (Y::L) K (Y::M) :- split X L K M.
```

The argument signature is described using only the `kind` and `type` keywords and the order in which items are listed in this signature is important. The first occurrence of `Aty` is a binding occurrence for both the type of `Order` as well as the types for `qsort` and `split`. The corresponding signature for `quicksort` should probably be written as

```
signature {quicksort kind Aty    type.
                     type Order  Aty -> Aty -> o}.

type         qsort  list Atype -> list Atype -> o.
accumulate   lists.
```

A use of such a module can be given as

```
?- {quicksort int <} ==> qsort (2::3::4::nil) L.
```

Parsing this module implication `==>` is a bit different from parsing other terms: in particular, the subexpression `{quicksort int <}` should be treated by the parser as a subterm over the signature

```
type qsort int list -> int list -> o.
accumulate   lists.
```

plus the signature items in the system module, where `int` and `<` is are given declarations. Since uses of parametric modules may contain terms that can be open, parametric modules would need to be processed using the AUGMENT and not AUGMENT' search rule.

In this example, it is only the predicate variable `Order` that is abstracted and not the clauses that may define the predicate that eventually instantiates it. Thus, if `Order` were instantiated with the predicate constant `lessthan`, clauses for `lessthan` could come from the current environment as well as from the module `quicksort` itself (if such clauses appeared there).

Using constants to denote modules and signatures. The names for modules and signatures should be converted to constants that are given types, say `modname` and `signame`, and declarations for these names need to be added (destructively) to the system module. In this way, the names of loaded modules will be available globally in a way that mirrors the persistent store on which the textual description of modules reside. Thus, `==>` and `===>` would have the types

```
kind modname, signame    type.
type ==>  modname -> o -> o.
type ===> modname -> signame -> o -> o.
```

For the purposes of compilation and parsing, once a module is parsed and checked, the signature file for that module should be placed on the persistent store. It should only be this second file that is needed during parsing and compiling of other modules. The `aux` files generated by Prolog/Mali [BR92] are essentially signatures that parallel modules.

Quantification over module names. It may be possible to permit variables to range over modules if we are willing to admit runtime signature checking of modules. For example, consider a goal of the form `(===> mod sig G)`. Here `mod` is a module whose signature is contained in that given by `sig`: this check would be done when this goal is attempted. Thus, in determining the static properties of a goal with this syntax, simply use the signature `sig` instead of attempting to determine the signature for `mod`, which may be a variable. Thus, the goal

```
?- memb M (mod1::mod2::mod3::nil), ===> M sig G.
```

would search for a module that can be used to establish the goal `G`. If all the modules `mod1`, `mod2`, and `mod3` have a signature contained in the signature `sig`, then no runtime error is generated by this goal. The syntax `(===> M sig G)` is essentially the same as `(M ==> G)` except that `M` must be restricted by the signature `sig`. Notice that it will not be possible to quantify over signature names.

Other declarations. Other declarations besides those for kind, type, and infix might also be allowed for constants and predicates. For example, certain types could be specified as being open or closed and certain predicates could have declarations describing how atomic goals could be suspended if certain argument positions are unbound.

Relationship to other aspects of an interpreter. The interaction of the module system with input/output and with the top-level of an interpreter (the read-prove-print loop) must also be considered carefully. A quite sensible approach to avoiding problems here is to take the approach of many programming languages (although, not taken by Prolog) of only printing out and reading in numbers and strings. Since these are globally available (declared implicitly in the "system" module) problematic interactions between modules and input/output can be averted. If a program wishes to read or print items of data types other than numbers and strings, parsers and printers must be written.

3 Formal aspects of this proposal

Connecting modular and higher-order programming. The interaction between modular programming and higher-order programming can be complex. In the setting here, it is possible for predicate names to appear within terms and for the code that describes such predicates to appear and disappear from the current program. Thus, the clauses defining a predicate may not be present when the predicate inside a term is finally called. In a recent draft of the evolving Prolog standard, the resolution of such conflicts was based on rather ad hoc considerations. With the proposal here, we are fortunate to have access to a higher-order theory of hereditary Harrop formulas [MNPS91] to help us resolve problems about this interaction. Having explicit signatures, explicit quantification, and unification support for local constants helps to give more structure to this problem.

Representation independence. The design of λProlog has been motivated in part by the desire to make logic play as large a role as possible in efforts to extend the expressiveness of logic programming. Of the many reasons for pursuing declarative languages, one is the fact that such deep meta-theoretic properties as cut-elimination and model theoretic semantics can be used to reason about the text of programs directly. Thus, analyzing *programming-in-the-small* within "pure" λProlog can be attacked using these deep principles. We can hope that the collections of modules can also be studied using these same principles.

As an example of such reasoning, consider the problem of *representation independence* for abstract data types. If we follow the line of argument given in [Mil89a] (and above) for coding abstract data types, representation independence follows directly. For example, consider the following two existentially quantified formulas, E_1 and E_2, which provide different implementations of queues. (I shall use the syntactic variable E to range over possibly existentially quantified definite formulas.) Here, we use λProlog's `sigma` to denote existential quantifiers: informally, we can think of the quantifier expression `sigma qu\(sigma f\(` as the declaration `local qu, f` at the top of a module.

```
sigma qu\(sigma f\(
      pi L\       ( empty (qu L L) ),
  pi X\(pi L\(pi K\( enter  X (qu L (f X K)) (qu L K) ))),
  pi X\(pi L\(pi K\( remove X (qu (f X L) K) (qu L K) ))) )).


sigma emp\(sigma g\(
                  ( empty emp ),
  pi X\(pi L\      ( enter  X L (g X L) )),
  pi X\           ( remove X (g X emp) emp ),
  pi X\(pi L\(pi K\( remove X (g Y L) (g Y K) :-
                                        remove X L K ))) )).
```

Let $\vdash$ be intuitionistic provability and let $\vdash^+$ be an enrichment of $\vdash$ that is conservative over $\vdash$ and that also makes it possible to reason about data structures (that is, induction must be incorporated). Then if we show that E_1 and E_2 are equivalent in $\vdash^+$, that is, $E_1 \vdash^+ E_2$ and $E_2 \vdash^+ E_1$, then the following argument is immediate: if $\Gamma, E_1 \vdash G$ then $\Gamma, E_1 \vdash^+ G$ since $\vdash^+$ enriches $\vdash$; by cut-elimination (assumed also for $\vdash^{+}$), $\Gamma, E_2 \vdash^{+} G$; finally, by conservative extension, $\Gamma, E_2 \vdash G$. Thus, if a goal G is provable using E_1, it is provable using E_2 (the converse is similar). The fact that abstractions are based on logic made this argument particularly direct.

4 Conclusion

A proposal for programming-in-the-large is given for λProlog. It is based on introducing keywords and declarations into the basic syntax of formulas and types in such a way that their declarative meaning can be reduced to the declarative principles of programming-in-the-small. Given that the latter language here is based on hereditary Harrop formulas and that these formulas have various structuring and abstraction mechanisms, programming-in-the-large is capable of exploiting notions of modules, local declarations, importing, and parametric modules.

References

[BLM94] M. Bugliesi, E. Lamma, and P. Mello. Modularity in logic programming. *Journal of Logic Programming*, 1994. To appear.

[BR92] Pascal Brisset and Olivier Ridoux. The architecture of an implementation of λProlog: Prolog/Mali. In Dale Miller, editor, *Proceedings of the 1992 λProlog Workshop*, 1992.

[Dyc92] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57(3), September 1992.

[EP89] Conal Elliott and Frank Pfenning. eLP, a Common Lisp Implementation of λProlog. Implemented as part of the CMU ERGO project, May 1989.

[Gen69] Gerhard Gentzen. Investigations into logical deductions, 1935. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland Publishing Co., Amsterdam, 1969.

[GMR88] L. Giordano, A. Martelli, and G. F. Rossi. Local definitions with static scope rules in logic languages. In *Proceedings of the FGCS International Conference, Tokyo*, 1988.

[GR84] D. M. Gabbay and U. Reyle. N-Prolog: An extension of Prolog with hypothetical implications. I. *Journal of Logic Programming*, 1:319–355, 1984.

[Gun91] Elsa L. Gunter. Extensions to logic programming motivated by the construction of a generic theorem prover. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming: International Workshop, Tübingen FRG, December 1989*, volume 475 of *Lecture Notes in Artificial Intelligence*, pages 223–244. Springer-Verlag, 1991.

[HM90] Joshua Hodas and Dale Miller. Representing objects in a logic programming language with scoping constructs. In David H. D. Warren and Peter Szeredi, editors, *1990 International Conference in Logic Programming*, pages 511–526. MIT Press, June 1990.

[HM94] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Journal of Information and Computation*, 1994. (To appear). Available from ftp.cis.upenn.edu, pub/papers/miller/ic92.dvi.Z.

[Hud89] Jörg Hudelmaier. *Bounds for cut elimination in intuitionistic propositional logic*. PhD thesis, University of Tübingen, Tübingen, 1989. To appear in Archive of Mathematical Logic.

[KNW93] Keehang Kwon, Gopalan Nadathur, and Debra Sue Wilson. Implementing a notion of modules in the logic programming language λprolog. In E. Lamma and P. Mello, editors, *Proceedings of the 1992 Workshop on Extensions to Logic Programming*, number 660 in Lecture Notes in Computer Science. Springer-Verlag, 1993.

[McC88a] L. T. McCarty. Clausal intuitionistic logic I. fixed point semantics. *Journal of Logic Programming*, 5:1–31, 1988.

[McC88b] L. T. McCarty. Clausal intuitionistic logic II. tableau proof procedure. *Journal of Logic Programming*, 5:93–132, 1988.

[Mil86] Dale Miller. A theory of modules for logic programming. In Robert M. Keller, editor, *Third Annual IEEE Symposium on Logic Programming*, pages 106–114, Salt Lake City, Utah, September 1986.

[Mil89a] Dale Miller. Lexical scoping as universal quantification. In *Sixth International Logic Programming Conference*, pages 268–283, Lisbon, Portugal, June 1989. MIT Press.

[Mil89b] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6(1-2):79–108, January 1989.

[Mil90] Dale Miller. Abstractions in logic programming. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 329–359. Academic Press, 1990.

[Mil92] Dale Miller. Abstract syntax and logic programming. In *Logic Programming: Proceedings of the First and Second Russian Conferences on Logic Programming*, number 592 in Lecture Notes in Artificial Intelligence, pages 322–337. Springer-Verlag, 1992.

[MN88] Dale Miller and Gopalan Nadathur. λProlog Version 2.7. Distribution in C-Prolog and Quintus sources, July 1988.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[NM88] Gopalan Nadathur and Dale Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.

[NP92] Gopalan Nadathur and Frank Pfenning. The type system of a higher-order logic programming language. In Frank Pfenning, editor, *Types in Logic Programming*, pages 245–283. MIT Press, 1992.