A Proposal for Modules in λProlog: Preliminary Draft

Dale Miller [1]
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 [20], 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 [15], abstract data types [14], and state encapsulation [7]. 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 it static semantics. The dynamic aspects are covered in various other papers: in particular, see the paper by Kwon, Nadathur, and Wilson [10] in these proceedings.

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 attached 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: its 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. They have not been considered seriously as programming languages for several reasons, including the difficulty of understanding and

[1]Supported in part by ONR N00014-88-K-0633, NSF CCR-91-02753, and DARPA N00014-85-K-0018.

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 have several high-level principles. For example, we should ask for such proposals to support several of the following properties.

- There should be a non-trivial notion of the equivalence of modules that would guarantee that a module can be replaced by an equivalent module with little to no impact on the behavior of a larger program. This property is sometimes called *representation independence* (see Section 3).
- Constructs for programming-in-the-large should not complicate the meaning of the underlying, declarative language.
- Modules should support transitions from specification to implementation.
- Modular programming should work smoothly with higher-order programming. In Prolog, a particular challenge is getting the semantics of the `call/1` predicate correct.
- Rich forms of abstraction, hiding, and parametrization should be possible.
- Modules should allow a rich calculus of transformations. These should include partial evaluation, fold/unfold, and even compilation.
- 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.
- 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.

One approach: map module syntax directly to logic 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 [4], McCarty [11, 12], and Miller [13, 15, 16], allows for a simple stack-based structuring of the runtime program and set of constants. The modal logic of Giordano, Martelli, and Rossi [5] provides an interesting variation on the simple "visibility rules" effecting logic programs based on the intuitionistic theory of hereditary Harrop formulas. A recent linear logic refinement of hereditary Harrop formulas by Hodas and Miller [8] modifies the stacked-based discipline of programs by allowing some program clauses to be deleted once they are used within a proof.

One approach to developing a principled modular programming language is to reduce programming-in-the-large to programming-in-the-small is 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 in several ways. As it turns out, much of the scoping primitives for the module facility proposed here do not come from the higher-order quantification available in λProlog. In fact, the propositional logic fragment of λProlog supports the stacked-based treatment of programming clauses. Higher-order quantification is important, 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 treats a program as a pair containing a signature and set of clauses. For example, the proof theoretic treatment of λProlog given in [16] 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 [17]. Thus it will not be surprising that the module proposal presented here will make extensive use of signatures. 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 change would make use of a notion of signature similar to that used here. Gunter [6] also makes use of signatures in developing a module calculus for λProlog.

Finally, it is important to say that what follows is just the draft of a proposal. Much of what follows has not be debated by those currently using 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 this logic programming setting.

2.2 Persistent store

Interacting with a persistent store, such as the Unix file system, is problematic within our logic programming setting: some non-logical predicates are required at the core of our module facility. In particular, the predicate

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

predicate 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 [18] was 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

In order to allow useful types, we admit type constructors. There is only one of these built into λProlog, namely the infix "function space" constructor `->`. Other type constructors can be declared via the `KIND` declaration. (Keywords will be capitalized for readability: in most implementations of λProlog, keywords appear in lowercase letters.) For example,

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

As this example show, the only kind that can be associated with a type constructor is any "first-order kind" involving only `type` and `->`. 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 types variables will provide λProlog with a degree of polymorphism. Type variables are tokens within type expressions that have an initial uppercase letter. The following are some type declarations.

```
TYPE nil     list A.
TYPE ::      A -> list A -> list A.
TYPE append  list A -> list A -> list A -> o.
TYPE memb    A -> list A -> o.
```

λProlog has numerous build-in types, including type `o`, the type of λProlog formulas.

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

I will assume that types are property formed (they respect kind declarations) and that formulas and terms are well typed. See [21] 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. Op-declarations are also stored as members of signatures. The following is an example of a signature.

```
OP   150 :: xfy.
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.
```

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 lists of constants. For this, we shall use the keywords `SIGNATURE`, `TYPE`, `KIND`, `OP`, `ACCUMULATE`, `LOCAL`, and `LOCALKIND`. The keyword `SIGNATURE` is used to name a signature and the keywords `TYPE`, `KIND`, and `OP` are used simply to enumerate the members of a signature. `ACCUMULATE` takes a list of signatures: its intended meaning is to merge in 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 are two signature descriptions.

```
SIGNATURE lists.
OP   150 :: xfy.
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.

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 `ACCUMULATE`ing 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 op-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 op-declaration, that constant is given the identical op-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).

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.

2.6 Module syntax

Modules will be built from kinds, types, and program clauses using the following keywords: `TYPE`, `KIND`, `OP`, `LOCAL`, `LOCALKIND`, `MODULE`, `ACCUMULATE`, and `IMPORT`. The meaning of `TYPE`, `KIND`, and `OP` 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 within a module: the dynamic semantics of `LOCAL` will be interpreted as an existential quantifier, as described in [14]. The keywords `ACCUMULATE` and `IMPORT` will be described further 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 of that constant. All declarations are global in a module. Figure 1 contains two examples of modules.

2.7 Static semantics for modules

The static semantics of modules is used to determine which signature and formulas are intended by the module. 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 `mod1`.

A signature description is built from a module as follows.

```
MODULE lists.

OP 150 :: xfy.
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.

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.

reverse L K :- rev L K nil.

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

Figure 1: The `lists` and `rev` modules.

- `TYPE` and `KIND` declarations stay `TYPE` and `KIND` declarations.
- All `IMPORT`ed, `ACCUMULATE`d, and module implication (`==>`) modules have their signatures `ACCUMULATE`d.
- If the qualified module importing `(===> mod sig) G` is used, then the signature `sig` is `ACCUMULATE`d (see Section 2.10 for a description of `===>`).
- `LOCAL` and `LOCALKIND` become `LOCAL` and `LOCALKIND`.

Notice that it is possible for `LOCAL` and `LOCALKIND` to provide scope to a constant that is `IMPORT`ed or `ACCUMULATE`d. If `IMPORT` or `ACCUMULATE` is used in a module and there is no corresponding module with the correct name, then look for a signature with that name. Thus modules without clauses can simply be written as signatures.

The static semantics of the `IMPORT` keyword construction is a bit complicated, although it does follow closely the lines described in [15] and implemented in LP2.7 and eLP [3]. 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 $((E_2 \wedge E_3) \supset G) \supset A$.

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, it should only be necessary for the signature of an accumulated or imported module to be read.

2.8 Environment support

The process of parsing a module will also be accompanied with type checking and type inference. In particular, a file containing a module may not attribute a type to all constants. 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 [21] for more discussion on type inference 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.

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.

The ACCUMULATE keyword. Although the meaning of this 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 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`.

The IMPORT keyword. Proof search based on clauses obtained by importing a module into another module can benefit from some recent work on provability in intuitionistic logic. For example, both Hudelmaier [9] and Dyckhoff [2] have demonstrated that the implication-left rule can be improved (with respect to proof search). For example, the implication-left rule can be split into several cases depending of 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.

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

Notice that the formula $(E \supset G) \supset A$ could be the result of importing a module E into a module listing 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 $j = 1, \ldots, n$. An argument for the completeness for this rule can be found in [10].

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 into a module which imports a module containing local constants essentially loads its local constants into the current signature and loads it's code (the formula D) into the current program.

Another important aspect of the dynamic semantics of modules is presented in [10] where the AUGMENT search rule is modified to be the AUGMENT' search rule. This new rule is used only for modules and not formulas thus forcing an operational (but not declarative) distinction between

programming-in-the-large and small. The AUGMENT' rule essentially says that if the current program space already contains a module, that module should not be assumed again: 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 following example. Consider a goal of the form `(p a) => (p X) => G`, where `X` is a logical variable. If we checked to see if `(p X)` in the context, it would seem that we should allow the unification of `X` with `a`. It would be easy to construct examples where the order of instantiating variables would yield two different answers to this computation, an undesirable effect.

2.10 Questions and additional features

I list below 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 also add to it a signature over which that module is parametric. An example could be given as follows.

```
MODULE {quicksort KIND Atype    type.
                  TYPE Order    Atype -> Atype -> o}.

TYPE      qsort  list Atype -> list Atype -> o.
LOCAL     split  Atype -> list Atype -> list Atype -> list Atype -> 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 corresponding signature should probably be written as

```
SIGNATURE {quicksort KIND Atype    type.
                     TYPE Order    Atype -> Atype -> 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

```
KIND int   type.
TYPE >     int -> int -> o.
TYPE qsort int list -> int list -> o.
```

plus the signature items in `lists` (and the system module, where `<` is given an op-declaration).

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, they will be available globally. Thus, `==>` and `===>` (this second arrow is described below) would have the types

```
KIND modname, signame    type.
TYPE ==>  modname -> o -> o.
TYPE ===> modname -> signame -> o -> o.
```

The current convention in LP2.7 and eLP is that there is one module per file and that the file's name is built from the module's name. This approach has the advantage that by mentioning a module name in one of these interpreters, it is possible for the system to find the file containing that module. It may be an advantage, however, to drop this linkage, in which case, files, possibly containing a number of modules and signatures, are loaded by using entire path names. For the purposes of compilation and parsing, once a file is parsed and checked, a second, parallel file containing only signatures might be generated from the one that is just parsed. It should only be this second file that is needed during parsing and compiling of other modules. The `aux` files generated by Prolog/Mali [1] 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 one for `mod`, which may be a variable. Thus, a goal of the form

```
?- 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 `op` might also be allowed. 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 must also be considered carefully.

3 Formal aspects of this proposal

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. There are many reasons for this emphasis on logic: the resulting language remains declarative and programs can be given meaning using such deep meta-theoretic properties as cut-elimination and model theoretic semantics. Thus, analyzing *programming-in-the-small* within "pure" λProlog can be attached using these deep principles. We can hope that the language for describing modules will also have such principles.

As an example of such principles, consider the problem of *representation independence* for abstract data types. If we follow the line of argument given in [14] (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.)

```
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.

Since the higher-order theory of hereditary Harrop formulas has been worked out in [19], there should be little problem getting this module facility to work smoothly with higher-order programming. Numerous other formal aspects of this module proposal must also be explored.

4 Conclusion

I have described a possible approach to programming-in-the-large for λProlog. This proposal is designed to ensure that the module constructions are declarative and this was done by making certain that the module syntax can be replaced in a very natural way by logical connectives.

This proposal is just a draft: many details have been left out. A subsequent version of this proposal will hopefully correct this shortcoming.

References

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

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

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

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

[5] 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.

[6] 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.

[7] 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.

[8] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Journal of Information and Computation*, 1992. Invited to a special issue of papers from the 1991 LICS conference.

[9] Jörg Hudelmaier. *Bounds for cut elimination in intuitionistic propositional logic.* PhD thesis, University of Tübingen, Tübingen, 1989.

[10] Keehang Kwon, Gopalan Nadathur, and Debra Sue Wilson. Implementing a notion of modules in the logic programming langauage λprolog. In *Proceedings of the 1992 λProlog Workshop*, 1992.

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

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

[13] 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.

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

[15] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79 – 108, 1989.

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

[17] 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. Also available as technical report MS-CIS-91-72, UPenn.

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

[19] 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.

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

[21] 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.