Lolli: An Extension of λProlog with Linear Logic Context Management *

Joshua S. Hodas
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6839 USA
hodas@saul.cis.upenn.edu

May 7, 1992

Introduction

The announcement for this workshop begins with a passage about the utility of higher-order hereditary Harrop formulas for many applications, and the very existence of the workshop is a partial correctness proof of the passage. Nevertheless, there are applications for which the intuitionistic management of proof contexts (or, concretely, program databases) provided by λProlog has been unable to provide natural, logical solutions. Many such problems, such as how to program the Prolog `bag_of` predicate — which would require a way of augmenting the database such that the changes survive a failure — seem unlikely to yield to logical analysis in any system related to hereditary Harrop formulas. Others, however, can be addressed by relatively simple modifications of the logic underlying λProlog.

In 1990 two problems motivated Dale Miller and me to examine the possibility of designing a logic programming language based on a fragment of Girard's linear logic [2] similar to the hereditary Harrop formula fragment of intuitionistic logic.

The first problem involved representing the notion of mutable object state within logic programming [4]. While it is simple to use representative predicates to store the state of an object in the database (or proof context), it is not possible to model the modification of state, since the only change to the database allowed in λProlog is that of stack-like augmentation through the use of implications in goals. Thus, if the state of a switch is stored using the predicates *off* and *on*, and the program Γ includes the (slightly) higher-order clauses:

$$\Gamma = \begin{cases} \forall G.[toggle(G) \subset (on \wedge (off \supset G))] \\ \forall G.[toggle(G) \subset (off \wedge (on \supset G))] \end{cases}$$

*This paperappears in the proceedings of the 1992 Workshop on the λProlog Programming Language. The entire proceedings is available electronically at `http://www.cis.upenn.edu/~dale/lProlog/workshop92.html`.

then the proof of the goal $\mathit{off} \multimap \mathit{toggle}(G)$ might proceed as follows:

$$\dfrac{\dfrac{\dfrac{}{\Gamma, \mathit{off} \longrightarrow \mathit{off}} \quad \dfrac{\vdots}{\Gamma, \mathit{off}, \mathit{on} \longrightarrow G}}{\Gamma, \mathit{off} \longrightarrow \mathit{off} \wedge (\mathit{on} \supset G)}}{\dfrac{\Gamma, \mathit{off} \longrightarrow \mathit{toggle}(G)}{\Gamma \longrightarrow \mathit{off} \supset \mathit{toggle}(G)}}$$

So, rather than being toggled, the switch has indeterminate state during the proof of G. The problem is the implicit use of the contraction rule of intuitionistic logic which allows the original state of the switch to be copied to both sides of the proof tree.

By considering linear management of proof contexts, in which the use of contraction and weakening is restricted to formulas marked with the ! operator, this and several other similar problems can be properly modeled. For instance, if the horn clauses above are replaced with the following linear logic formulas:

$$\Gamma = \begin{cases} !\{\forall G.[\mathit{toggle}(G) \circ\!\!- (\mathit{on} \otimes (\mathit{off} \multimap G))]\} \\ !\{\forall G.[\mathit{toggle}(G) \circ\!\!- (\mathit{off} \otimes (\mathit{on} \multimap G))]\} \end{cases}$$

then the proof of the equivalent goal, $\mathit{off} \supset \mathit{toggle}(G)$ proceeds as:

$$\dfrac{\dfrac{\dfrac{}{\Gamma, \mathit{off} \longrightarrow \mathit{off}} \quad \dfrac{\vdots}{\Gamma, \mathit{on} \longrightarrow G}}{\Gamma, \mathit{off} \longrightarrow \mathit{off} \otimes (\mathit{on} \multimap G)}}{\dfrac{\Gamma, \mathit{off} \longrightarrow \mathit{toggle}(G)}{\Gamma \longrightarrow \mathit{off} \multimap \mathit{toggle}(G)}}$$

with the desired result that the switch is in the toggled position during the proof of G.

In two recent papers Miller and I have discussed at length the design of a logic programming language based on such formulas [6, 5]. Inference rules for the operators of the language are given in Figure 1. While these rules are not the standard ones of linear logic, they are equivalent to a fragment of linear logic. In this system a proof context consists of two parts: the intuitionistic part (on the left of the semi-colon), in which arbitrary implicit contraction and weakening are allowed, and the linear part (on the right of the semi-colon), in which those rules are barred.

Concrete Syntax and the Relationship with λProlog

An important aspect of the Lolli project was the hope that the language could be designed as a modular refinement of λProlog. That is, any purely λProlog program should run 'unmodified' within Lolli[1] and behave in the expected way.

Since the logical operators of the two languages are different, this embedding requires defining a mapping of formulas of intuitionistic logic into the new system. Girard gave such a mapping in

[1]The current implementation of Lolli is an essentially first-order language (ie., while it allows quantification over predicates, formulas, and terms, it does not implement λ-terms or higher-order unification), so this section should be read as referring to the similar fragment of λProlog.

$$\frac{}{\Gamma; A \longrightarrow A}\ \text{identity} \qquad \frac{}{\Gamma; \Delta \longrightarrow \top}\ \top_R \qquad \frac{}{\Gamma; \emptyset \longrightarrow \mathbf{1}}\ \mathbf{1}_R \qquad \frac{\Gamma, B; \Delta, B \longrightarrow C}{\Gamma, B; \Delta \longrightarrow C}\ absorb$$

$$\frac{\Gamma; \Delta, B_i \longrightarrow C}{\Gamma; \Delta, B_1 \,\&\, B_2 \longrightarrow C}\ \&_{Li} \qquad \frac{\Gamma; \Delta \longrightarrow B \qquad \Gamma; \Delta \longrightarrow C}{\Gamma; \Delta \longrightarrow B \,\&\, C}\ \&_R$$

$$\frac{\Gamma; \Delta_1 \longrightarrow B \qquad \Gamma; \Delta_2, C \longrightarrow E}{\Gamma; \Delta_1, \Delta_2, B \multimap C \longrightarrow E}\ \multimap_L \qquad \frac{\Gamma; \Delta, B \longrightarrow C}{\Gamma; \Delta \longrightarrow B \multimap C}\ \multimap_R$$

$$\frac{\Gamma; \emptyset \longrightarrow B \qquad \Gamma; \Delta, C \longrightarrow E}{\Gamma; \Delta, B \Rightarrow C \longrightarrow E}\ \Rightarrow_L \qquad \frac{\Gamma, B; \Delta \longrightarrow C}{\Gamma; \Delta \longrightarrow B \Rightarrow C}\ \Rightarrow_R$$

$$\frac{\Gamma; \Delta_1 \longrightarrow B \qquad \Gamma; \Delta_2 \longrightarrow C}{\Gamma; \Delta_1, \Delta_2 \longrightarrow B \otimes C}\ \otimes_R \qquad \frac{\Gamma; \emptyset \longrightarrow G}{\Gamma; \emptyset \longrightarrow !G}\ !G$$

$$\frac{\Gamma; \Delta \longrightarrow B[t/x]}{\Gamma; \Delta \longrightarrow \exists x.B}\ \exists_R \qquad \frac{\Gamma; \Delta \longrightarrow B_i}{\Gamma; \Delta \longrightarrow B_1 \oplus B_2}\ \oplus_{Ri}$$

$$\frac{\Gamma; \Delta, B[t/x] \longrightarrow C}{\Gamma; \Delta, \forall x.B \longrightarrow C}\ \forall_L \qquad \frac{\Gamma; \Delta \longrightarrow B[y/x]}{\Gamma; \Delta \longrightarrow \forall x.B}\ \forall_R$$

provided that y is not free in the lower sequent.

Figure 1: A proof system for the connectives $\top$, $\mathbf{1}$, $\&$, $\multimap$, $\Rightarrow$, $!$, $\otimes$, $\oplus$, $\forall$, and $\exists$.

the first paper on linear logic [2]. However, given that we are working in the restricted setting of hereditary Harrop formulas it is possible to define a more parsimonious, albeit more complicated, one. This translation, which was introduced in [5], is in the form of two mutually recursive functions, one applied to formulas in negative positions (ie. program clauses), and the other to formulas in positive positions (ie. queries).

$$\begin{gathered}
(A)^+ = (A)^- = A, \text{ where } A \text{ is atomic} \\
(true)^+ = \mathbf{1} \\
(true)^- = \top \\
(B_1 \wedge B_2)^+ = (B_1)^+ \otimes (B_2)^+ \\
(B_1 \wedge B_2)^- = (B_1)^- \,\&\, (B_2)^- \\
(B_1 \supset B_2)^+ = (B_1)^- \Rightarrow (B_2)^+ \\
(B_1 \supset B_2)^- = (B_1)^+ \multimap (B_2)^- \\
(\forall x.B)^+ = \forall x.(B)^+ \\
(\forall x.B)^- = \forall x.(B)^- \\
(B_1 \vee B_2)^+ = (B_1)^+ \oplus (B_2)^+ \\
(\exists x.B)^+ = \exists x.(B)^+
\end{gathered}$$

The intuitionistic sequent (over just these operators) $\Gamma \longrightarrow G$ is then mapped to the sequent $\Gamma^-; \emptyset \longrightarrow G^+$, which has a proof if and only if the original sequent did.

Given the λProlog syntax for hereditary Harrop formula programs, this mapping suggests a concrete syntax for the operators of the language, which is given in the table on the next page.

Operator	Parity	Syntax
$\top$	+	`erase`
$\mathbf{1}$	+	`true`
$\&$	+	`&`
	-	`&`
$\otimes$	+	`,`
$\oplus$	+	`;`
$\multimap$	+	`-o`
	-	`:-`
$\Rightarrow$	+	`=>`
	-	`<=`
!	+	`{...}`
$\forall x.B$	+	`forall x\B`[2]
	-	`forall x\B`*
$\exists x.B$	+	`exists x\B`*

As with λProlog, terms and atoms are written in a curried form and the standard quantifier assumptions are made. It is straightforward to confirm that existing Prolog and λProlog programs are written, and run, as expected. For instance, the λProlog query:

```
pi X\ pi Y\
   (memb X (X::Y))  =>
pi X\ pi Y\ pi Z\
   (memb X (Y::Z) :- neq X Y, memb X Z) =>
memb G (a::b::nil).
```

represents the formula:

$$\exists G.[(\forall X.\forall Y.memb(X, X :: Y)) \supset (\forall X.\forall Y.\forall Z.(memb(X, Y :: Z) \subset (neq(X, Y) \wedge memb(X, Z)))) \supset memb(G, a :: b :: nil)]$$

which, when translated into the new system becomes:

$$\exists G.[(\forall X.\forall Y.memb(X, X :: Y)) \Rightarrow (\forall X.\forall Y.\forall Z.(memb(X, Y :: Z) \multimap (neq(X, Y) \otimes memb(X, Z)))) \Rightarrow memb(G, a :: b :: nil)]$$

which has the concrete syntax:

```
forall X\ forall Y\
   (memb X (X::Y))  =>
forall X\ forall Y\ forall Z\
   (memb X (Y::Z) :- neq X Y, memb X Z) =>
memb G (a::b::nil).
```

[2]The use of `forall` and `exists` as syntax for the explicit quantifiers represents a personal preference of this author.

And, when run, this query will have the same execution profile as the original λProlog query.

In contrast, programs which take advantage of the linear features of the system will of necessity make use of the new elements of the syntax. So, for instance, the ill-performing intuitionistic formulas defining the *toggle* predicate would be written (in λProlog and Lolli) as:

```
toggle G :- on, off => G.
toggle G :- off, on => G.
```

while the well-performing linear logic formulas would be written as:

```
toggle G :- on, off -o G.
toggle G :- off, on -o G.
```

In order for existing programs to work properly, it is assumed that the clauses in a module are loaded into the unbounded (intuitionistic) portion of the proof context. The programmer can override this assumption by preceding individual clauses with the `LINEAR` declaration. Thus, it is possible to specify an initial setting for the switch within the program file, as in:

```
LINEAR on.
```

Note that the use of all uppercase for `LINEAR`, is not optional. Since the system uses curried notation, this is the only way (short of ruling out its use in other forms) of recognizing that it is a declaration, and not a predicate name. For consistency, and improved readability, this restriction is also applied to the `LOCAL` and `MODULE` declarations described below.

Modules

Lolli programs are divided into modules in the same way as λProlog programs. By convention, enforced by the interpreter, files carry the extension '`.ll`', and are loaded using the operator '`--o`', or at the top level, with '`load modulename`', which is equivalent to '`modulename --o top`'.

A module may begin with a list of local constant declarations, such as:

```
LOCAL a B c.
LOCAL d.
```

with multiple constants separated by spaces, or listed in separate declarations. Because Lolli is essentially first-order, types and kinds, and their declarations, are not needed or supported. A future release of Lolli may support L_λ-unification, but will likely still be type-free. Note that since constants are untyped, predicate names may be reused at different arities, as in ordinary Prolog.

The λProlog module system has been extended to allow for parameterized modules. That is, the module declaration is of the form:

```
MODULE modname param_1 ... param_n.
```

where `modname` matches the root of the file name, and the parameters are variables to be unified placewise with the terms in the loading goal. Note that while the formal parameters are variables, they are generally intended to be viewed as constants within the module, and as such may begin with lowercase characters if the programmer so chooses. Thus, if the module is declared:

```
MODULE foo a B.
```

and is loaded with '`foo c d --o top`', then the clauses in `foo.ll` are loaded with all instances of `a` and `B` instantiated to `c` and `d` respectively.

The logical status of the module system can be summarized as follows:

`MODULE mod` $x_1 \ldots x_n$.
`LOCAL` $y_1 \ldots y_m$.

$H_1 x_1 \ldots x_n y_1 \ldots y_m$.
$\vdots \qquad \vdots$
`LINEAR` $H_i x_1 \ldots x_n y_1 \ldots y_m$.
$\vdots \qquad \vdots$
$H_p x_1 \ldots x_n y_1 \ldots y_m$.

associates to `mod` the parameters $x_1 \ldots x_n$, the local constants $y_1 \ldots y_m$, and the clauses $H_1 \ldots H_p$, which may contain free occurrences of the variables $x_1 \ldots x_n$ and constants $y_1 \ldots y_m$. When the module is loaded within a goal formula, using the syntax `mod` $t_1 \ldots t_n$`--o` B, that goal is considered only as short-hand for the goal

`forall` $y_1 \backslash \ldots$`forall` $y_m \backslash$
$[(H_1 t_1 \ldots t_n y_1 \ldots y_m) => \ldots (H_i t_1 \ldots t_n y_1 \ldots y_m)$ `-o` $\ldots (H_p t_1 \ldots t_n y_1 \ldots y_m)$ `=>` $B]$.

Here, we overload the symbols $y_1, \ldots, y_m$ to be constants in the `LOCAL` declaration and bound variables in the displayed formula above. In general, this overloading should not cause problems. Also, in this example, it is assumed that the formula B and the terms $t_1, \ldots, t_n$ do not contain occurrences of $y_1, \ldots, y_m$.

The implementation of parameterized modules was driven by the need to be able to handle the object-oriented programming examples from an earlier paper [4], where they were used to pass initialization information to objects. Nevertheless they have proved useful in a number of instances. For example, the following module defines the shell of a multiset rewriting system, along the lines of the example given in [6, 5]. The rewrite rules themselves, however, are in a separate module, whose name is passed to this one as a parameter when this module is loaded. In order to ensure the soundness of the rewriter, a local predicate name is used to store the multiset in the database. That name is, in turn, passed to the rules module when it is loaded. The shell is given by:

```
MODULE rewrite rulemodule.

LOCAL hyp.

collect nil.
collect (X::L) :- hyp X, collect L.

unpack nil G :- G.
unpack (X::L) G :- hyp X -o unpack L G.

rewrite L K :- unpack L ((rulemodule hyp) --o (rewrite  (collect K))).
```

while a rule module might be of the form:

```
MODULE rules1 hyp.

rewrite G :- G.

rewrite G :- hyp 4, ((hyp 2, hyp 2) -o rewrite G).
rewrite G :- hyp 4, ((hyp 3, hyp 1) -o rewrite G).
rewrite G :- hyp 3, ((hyp 2, hyp 1) -o rewrite G).
rewrite G :- hyp 2, ((hyp 1, hyp 1) -o rewrite G).
```

and a sample query would be:

```
?- rewrite rules1 --o rewrite (3::nil) L.

?L674 <- (3 :: nil) .;
?L674 <- (2 :: 1 :: nil) .;
?L674 <- (1 :: 2 :: nil) .;
?L674 <- (1 :: 1 :: 1 :: 1 :: nil) .;
?L674 <- (1 :: 1 :: 1 :: 1 :: nil) .;
?L674 <- (1 :: 1 :: 1 :: 1 :: nil) .;
?L674 <- (1 :: 1 :: 1 :: 1 :: nil) .;
?L674 <- (1 :: 1 :: 1 :: 1 :: nil) .;
?L674 <- (1 :: 1 :: 1 :: 1 :: nil) .;
```

Implementation

Lolli is currently available in two implementations. The first is a simple Prolog meta-interpreter given in [6, 5] and reproduced in Figure 2. The code as given implements only the propositional fragment of the language (with a few differences from the concrete syntax described above), but is useful for experimenting with the core of the underlying logic. The meta-interpreter could be trivially extended to the first-order language by re-implementing it in λProlog. Other than the change of syntax, that system would differ only in the addition of two clauses to handle quantification. Unfortunately, the lack of `op` declarations in λProlog would make the system a little more unwieldy.

The author has also developed a relatively rich implementation of Lolli in Standard ML of New Jersey (which should port to any ML which can handle MLYACC and MLLEX). That implementation supports the full language as described here, in addition to a reasonable selection of evaluable predicates and one extra-logical control structure (guard expressions). That implementation was inspired by (and built on a core of code from) Elliott and Pfenning's article on implementing λProlog-like languages in a functional setting [1]. The full implementation of Lolli, with documentation and copies of [6, 5] is available by anonymous ftp from *add directions when this is a reality later (by mid-summer)*

```
%  The logic being interpreted contains the following logical connectives:
%    true/0             a constant (empty tensor, written as 1 in the logic)
%    erase/0            a constant (erasure, written as Top in the logic)
%    bang/1              the modal, written as {} in the paper.

:- op(145,xfy,->).  %    linear implication, written as -o in the paper
:- op(145,xfy,=>).  %    intuitionistic implication
:- op(140,xfy,x ).  %    multiplicative conjunction (tensor)
:- op(140,xfy,& ).  %    additive conjunction
:- op(150,xfy,::).  %    non-empty list constructor

   interp(G) :- prove(nil, nil, G).

   isG(true).                              isR(erase).
   isG(erase).                             isR(B)        :- isA(B).
   isG(B)        :- isA(B).                isR(B1 & B2)  :- isR(B1), isR(B2).
   isG(B1 -> B2) :- isR(B1), isG(B2).      isR(B1 -> B2) :- isG(B1), isR(B2).
   isG(B1 => B2) :- isR(B1), isG(B2).      isR(B1 => B2) :- isG(B1), isR(B2).
   isG(B1 &  B2) :- isG(B1), isG(B2).
   isG(B1 x  B2) :- isG(B1), isG(B2).
   isG(bang(B))  :- isG(B).

   prove(I,I, true).
   prove(I,O, erase)   :- subcontext(O,I).
   prove(I,O, G1 & G2) :- prove(I,O,G1), prove(I,O,G2).
   prove(I,O, R -> G)  :- prove(R :: I, del :: O,G).
   prove(I,O, R => G)  :- prove(bang(R) :: I, bang(R) :: O,G).
   prove(I,O, G1 x G2) :- prove(I,M,G1), prove(M,O,G2).
   prove(I,I, bang(G)) :- prove(I,I,G).
   prove(I,O, A)       :- isA(A), pickR(I,M,R), bc(M,O,A,R).

   bc(I,I,A, A).
   bc(I,O,A, G -> R)  :- bc(I,M,A,R), prove(M,O,G).
   bc(I,O,A, G => R)  :- bc(I,O,A,R), prove(O,O,G).
   bc(I,O,A, R1 & R2) :- bc(I,O,A,R1); bc(I,O,A,R2).

   pickR(bang(R)::I, bang(R)::I, R).                    subcontext(del::O, R ::I) :- isR(R), subcontext(O,I).
   pickR(R::I,       del::I,     R) :- isR(R).          subcontext(S::O,   S::I)  :- subcontext(O,I).
   pickR(S::I,       S::O,       R) :- pickR(I,O,R).    subcontext(nil,    nil).

% The following code provides the hooks into application programs.
:- op(150,yfx,<-). %     the converse of the linear implication

% Applications using this interpreter are specified using the <-/2 functor (denoting the converse
% of linear implication).  We shall assume that clauses so specified are implicitly banged (belong
% to the unbounded part of the initial context) and that the first argument to -> is atomic.  The
% following clause is the hook to clauses specified using <-.

prove(I,O, A) :- isA(A), A <- G, prove(I,O,G).

% A few input/output non-logicals.

prove(I,I, write(X)) :- write(X).         prove(I,I, read(X)) :- read(X).         prove(I,I, nl) :- nl.

% The following is a flexible specification of isA/1
notA(write(_)).  notA(read(_)).   notA(nl).   notA(erase).  notA(true).  notA(del).
notA(_ & _).  notA(_ x _).  notA(_ -> _).  notA(_ => _).  notA(bang(_)).
isA(A) :- \+(notA(A)).
```

Figure 2: A Prolog implementation of Lolli

Conclusion

The Lolli project is an ongoing one, and the language is by no means frozen. On the other hand, the collection of program examples is growing [6, 5, 3], and this shows that the logic fragment chosen represents a useful extension of the traditional hereditary Harrop formulas of λProlog.

Acknowledgements

The author is grateful to Dale Miller, for his partnership in this work, and to Jean-Marc Andreoli, Gianluigi Bellin, Jawahar Chirimar, Remo Pareschi, Pat Lincoln, Andre Scedrov, James Harland, Jean-Yves Girard and Fernando Pereira for conversations (with the author and with Dale Miller) about aspects of the design and theory of Lolli. He is also grateful to Frank Pfenning and Conal Elliott for providing such a strong base to work with in implementation. Finally, to Elizabeth Hodas for helpful editorial comments.

The author has been funded by ONR N00014-88-K-0633, NSF CCR-91-02753, and DARPA N00014-85-K-0018 through the University of Pennsylvania.

References

[1] Conal Elliott and Frank Pfenning. A semi-functional implementation of a higher-order logic programming language. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 289 – 325. MIT Press, 1991.

[2] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[3] Joshua S. Hodas. Implementing Gap Threading Parsers in a Linear-Logic Programming Language. Submitted to the 1992 Joint International Conference and Symposium on Logic Programming.

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

[5] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Journal of Information and Computation*, 1992. To appear.

[6] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic: Extended abstract. In G. Kahn, editor, *Sixth Annual Symposium on Logic in Computer Science*, pages 32 – 42, Amsterdam, July 1991.