Logic Programming with Monads and Comprehensions

Yves Bekkers * Paul Tarau †

Abstract

We give a logical reconstruction of all-solution predicates in terms of list comprehensions in λProlog's and we describe a variety of logic programming constructs in terms of monads and monad morphisms. Novel monad structures are described for lazy function lists, clause unfoldings and a monad morphism based embedding of Prolog in λProlog is given.

Keywords: computing paradigms, logic programming, monads, *list comprehensions, all solution predicates, λProlog, higher-order unification, lazy function lists.*

1 Introduction

Monads and comprehensions, have been successfully used in functional programming as a convenient generalization of various structurally similar programming constructs starting with simple ones like list processing and ending with intricate ones like CPS transformations and state transformers. We refer to the work of Philip Wadler [16, 17] for a long list of powerful examples and to Moggi [10] for the categorist sources of the concept of monad.

All-solution predicates are the logic programming siblings of list comprehensions in functional programming languages.

Let us point out an interesting (although possibly episodical) parallel between these two programming paradigms. When doubts have been raised by Goguen's thesis [6] on the usefulness of higher order constructs for functional programming, this seemed *déjà vu* for logic programmers familiar with Warren's paper [18]. After emulating some basic λ-calculus constructs in terms of first-order Horn-clause logic (i.e. what is also known as pure Prolog) Warren has shown that most of the expressiveness of higher-order constructs can be emulated in first order logic programming.

All-solution predicates have been recognized by Warren as a notable exception and have been present in logic programming ever since. However, in the

*Université de Rennes I and IRISA/INRIA Rennes, *bekkers@irisa.fr*

†Département d'Informatique, Université de Moncton, *tarau@info.umoncton.ca*

following years they have been quite often disregarded (see comp.lang.prolog 1983-1994) as 'non-logical' and 'non-declarative' by the high (or tight?) standards of the logic programming community.

Interestingly enough, the idea of the intrinsic usefulness of higher-order constructs seems to be back again in the functional programming community after the proof of expressiveness of some essentially high-order monad constructs (used for instance in describing state transformers and constant time array operations) in the papers of Philip Wadler.

Post-Prolog generation logic programming languages like λProlog make essential use of high-order unification and quantification to help the programmer specify the intended meaning of complex operations.

We will propose in this paper a 'monad-based' approach to the definition of all-solution predicates in λProlog [12]. As a follow-up, we obtain a clarification of Prolog's high-order operators and their similarities and differences with monad comprehensions in functional languages.

More generally, we will see various program transformations as morphisms between suitably defined monads and hint to some of their practical uses in compilation.

Novel monad structures are described for lazy function-lists, clause unfoldings and a monad morphism based embedding of an Prolog in λProlog is given.

In the examples that follow we will not limit ourself to Prolog but refer most of the time to its more powerful *superset* λProlog [7, 8, 14, 13] (in its λProlog-Mali incarnation [5, 2], which has been used to run all the programming examples).

The paper is organized as follows: we start by describing some monads for elementary data structures the (lists, lazy function-lists etc.) and monad morphisms. After that, we describe how to implement list comprehensions with monads, how to express monad operations in terms of all-solution predicates and the reverse process of defining list comprehensions in terms of `bagof`. Finally we deal with metaprogramming with monads; after describing programs as monads of clauses, we give an embedding of Prolog in λProlog. We conclude with the usual future work and conclusion sections.

2 Monads for elementary data structures

2.1 The list-monad in λProlog

We will start with a definition of the list monad in λProlog. Note that λProlog. is a language with polymorphic types, and with lambda terms manipulated through high-order unification. Let us stress from the start that terms are mostly used as data structures and that λProlog is not a mixture of orthogonal functional and logic programming languages but a natural extension of Horn clause logic based on the concept of *uniform proof* [9].

However we will not use in this case any of the quantification related features of λProlog so that the code will run with minor modifications on any Prolog system having the `call/N` primitive introduced by [11].

```
% primitive operations

% maps a single element of A to an object of the monad
type unitList A -> (list A) -> o.
unitList X [X].

% applies to a list an operation from elements to lists
% to obtain a new list
type bindList (list A)-> (A ->(list B)->o)-> (list B)-> o.
bindList [] _K [].
bindList [X|Xs] K R :-
  K X Ys,
  bindList Xs K Zs,
  append Ys Zs R.
```

```
% derived operations

% concatenates the elements of a list of lists
type joinList (list (list A)) -> (list A) -> o.
joinList Xss Xs :- bindList Xss id Xs.

% applies an operation to each element of a list
type mapList (A->B->o) -> (list A) -> (list B) -> o.
mapList F Xs Ys :- bindList Xs (compose F unitList) Ys.
```

```
%tools

type id A -> A ->o.
id X X.

% composes two predicates which implement functions
type compose (A->B->o) -> (B->C->o) -> (A->C->o).
compose F G  X Y :- F X Z, G Z Y.

type dupList A -> (list A) -> o.
dupList X [X,X].
```

For instance, the goal `bindList [1,2,3] dupList R.` will instantiate `R` to `[1,1,2,2,3,3]`.

It is easy to verify (and actually programmable in λProlog) that all the monad axioms and properties of [16, 17] hold for our definitions.

2.2 The monad of lazy function-lists in λProlog

Lazy function-lists [4] are a surprising but natural 'difference' data-structure which among other properties allows 'linear' performance on apparently quadratic predicates like the well known 'naive reverse' Prolog benchmark [4]. Basically this is achieved by a lazy implementation of β reduction. We refer to [3, 4] for the details of their operational semantics.

A simple example (which can be seen as a `unit` operator) is `x\z\[x|z]` which applied to `1` gives `\z[1|z]`.

The following program[1] implements the monad of lazy function-lists.

```
% tools
#define flistT(A) ((list A) -> (list A))
#define NIL x\x
#define CONS x\l\u\ [x | (l u)]
#define CONC l1\l2\x\ (l1 (l2 x))
```

```
% primitive monad operations
type unitFlist A -> flistT(A) -> o.
unitFlist X (CONS X NIL).

type bindFlist flistT(A)->(A->flistT(B)->o)->flistT(B) -> o.
bindFlist NIL _K NIL.
bindFlist (CONS X Xs) K (CONC Ys Zs) :-
  K X Ys,
  bindFlist Xs K Zs.
```

```
% derived monad operations
type joinFlist flistT(flistT(A)) -> flistT(A) -> o.
joinFlist Xss Xs :- bindFlist Xss id Xs.

type mapFlist (A->B->o) -> flistT(A) -> flistT(B) -> o.
mapFlist F Xs Ys :- bindFlist Xs (compose F unitFlist) Ys.
```

[1] using the concrete syntax of λProlog-Mali, with parametric types obtained through macro-expansion

2.3 Other elementary monads

2.3.1 Constructor sequence monads

By replacing the list constructor `'.'/2` and terminator `[]/0` by another pair we can generalize the list monad to a more general 'constructor sequence' monad. Two widely used instances are:

- the conjunction monad for `<',',true>`
- the disjunction monad for `<';',fail>`

2.3.2 The monad of difference lists

The implementation of this monad is similar to that of the monads of lists and lazy function-lists. However, its correctness in the absence of occur check and the need for extra logical tests to detect 'the end' of such lists make it less interesting especially when a concept of lazy function-list is available as it is the case with λProlog-Mali.

2.4 Monad morphisms

A monad morphism [16] is an application which preserves the monad operations.

Intuitively they correspond to programs which exhibit a similar structure inside different syntactic wrappers.

Morphisms of monads are a convenient way to implement equivalence preserving program transformations.

Due to space constraints we will point out here only a less obvious morphism pair between the monad of lists and the monad of lazy function lists.

```
type list2flist (list A) -> flistT(A) -> o.
list2flist L FL :- pi nil\ (append L nil (FL nil)).

type flist2list flistT(A) -> (list A) -> o.
flist2list F (F []).
```

`List2flist/2` is similar to a list-to-difference list converter except for the use of the universal quantifier `pi` to end the lazy function list.

Notice that `flist2list/2` means simply applying the lazy function list to the list terminator `[]`.

Let us just hint to an application: the programmer who works with monads is free to move from one representation to another. For instance he can debug a simpler 'plain' list program and then chose a more efficient representation following a monad morphism.

3 List comprehensions with monads

3.1 Concrete notation for list comprehension in λProlog

Let us consider two examples of set comprehensions :

$$B_1 = \{x | x \in L \ \& \ \text{odd}\ x\}$$
$$B_2 = \{[x, y] | x \in L_1 \ \& \ y \in L_2\}$$

B_1 is the set of odd elements in the set L, and B_2 is the set of pairs of values taken from the sets L_1 for the first value in the pair and L_2 for the second value in the pair.

The corresponding list comprehensions in λProlog are defined as follows :

```
B1 <== (all x\ (x : (x <-- L, 1 is x mod 2)))
B2 <== (all y\ (all x\ ([x, y] : (x <-- L1, y <-- L2))))
```

The syntax for list comprehension is :

lcT_τ	::=	(`all` *variable*\ lcT_τ) | (*term*$_\tau$ `:` *qualifiers*)
qualifiers	::=	*qualifier* | *qualifier* , *qualifiers* | (*qualifiers*)
qualifier	::=	*generator* | *filter*
generator	::=	*variable* `<--` *list*

A *filter* is a Prolog goal like `1 is x mod 2`. Infix operators are used in λProlog with the following correspondence :

Set comp.	*List comp.*
$\in$	`<--`
|	`:`
&	`,`
=	`<==`

Variables in list comprehension are explicitly quantified with the quantifier `all`. The constructors `all` and `:` define the *List comprehension data type* which we call `lcT`.

Data Type for list comprehensions Here are the λProlog type declarations for implementing list comprehensions :

```
kind lcT type -> type.
type all (_B -> (lcT A)) -> (lcT A).
type : A -> o -> (lcT A).
```

`lcT` is a type constructor, and `all` and `:` and data constructors for that type.

The generator `<--` is a 2-argument predicate, its type in λProlog is :

```
% -X <-- +Xs : X is an element of the list Xs
type <-- X -> (list X) -> o.
```

The translation from list comprehensions to lists is made with the predicate `<==`. Its type is :

```
type <== (list A) -> (lcT A) -> o.
```

All operators have been declared infix (xfx) with priority 700.

Translating list comprehensions to lists with the list-monad P. Wadler [16] gives a translation of list comprehensions into *list-monad* functions as follows :

`[t|x<--L]` $= (map\ \lambda x \rightarrow t\ L)$
`[t|(p,q)]` $= (join\ [[t|q]|p])$
`[t|true]` $= (unit\ t)$
`[t|b]` $= (if\ b\ then\ [t]\ else\ [])$

Due to space limitations we will not give translation of the two last equivalence rules into the two last clauses in figure 1. Note that `if` is expresed with conditional `(P ? A ; B)` of λProlog.

The rule for composing qualifiers is also easily translated. First, a list of list comprehensions is built and stored in `Xs`. Then the list comprehensions are translated applying the predicate `<==` to `Xs` with the `map` predicate. The resulting list is finally flattened with the `join` predicate.

Using quantifications and higher order unification The first rule in figure 1 interprets the quantifier `all`. A quantified variable is represented with the λ-variable of an abstraction such as `F`. For each quantified variable, a new universal constant `x` is created (using λProlog logical connector `pi`). The λ-variable is then substituted by a simple *application* `(F x)`. λProlog β-reduction insures the (lazy) copying of the term. Each new universal constant `x` is associated with a freshly created existentially quantified logical variable `_X`. This association is memorized with λProlog logical connector `=>`. The dynamic assertion `variable x _X` extends the program context. Notice that, according to λProlog quantification rules, the quantifiers on the two variables `x` and `_X` are ordered as :

$\exists$ `_X` $\forall$ `x`

The intuitionistic interpretation of such a declaration means that the signature of the logical variable `_X` does not contain the universal constant `x`. Hence, λProlog system, subsequently enforces that the substitution values for `_X` do not contain the universal constant `x`. This is fundamental for the rest of the interpretation, see further the interpretation of the generators `<--`.

```
% X <== Lc : X is the simple list corresponding to
%           the comprehension list Lc
Ys <== (all F) :- !,
   pi x\ (variable x _X => (Ys <== (F x))).
Ys <== ((F U) : (U <-- Xs)) :-
   \+ (\+ (variable U F)), !,
   mapList x\ y\ (y = (F x)) Xs Ys.
Zs <== (T : (P , Q)) :- !,
   Xs <== ((T : Q) : P),
   mapList x\ y\ (x <== y) Ys Xs,
   joinList Ys Zs.
Ys <== (T : true) :- !,
   unitList T Ys.
Ys <== (T : P) :-
   (P ? unitList T Ys ; Ys = []).
```

Figure 1: From list comprehensions to lists

The predicate `variable` is a dynamic predicate declared as follows :

```
type variable A -> (A -> B) -> o.
dynamic variable.
```

λProlog higher order unification is used for interpreting the rule for the generators `<--`. Higher order unification unifies the left term of the comprehension list with the term `(F U)`, and finds substitution values for the logical variable `F`. There may be several such substitutions, including some containing the universal constant `U`. The solutions containing the universal constant `U` are discarded with the interpretation of the goal `\+ (\+ (variable U F))`. The substitution values for `F` will be filtered because the second argument for `variable` is a logical variable which does not contain the universal constant `U` in its signature.

3.1.1 Using list comprehensions

Here are four examples of the use of list comprehensions :

1. Generating the odd numbers of a list `L`
2. Generating the pairs `[x,y]` with the values of `x` taken from a list `L1`, and the values of `y` taken from a list `L2`.
3. Generating the pairs `[x,y]` with the values of `x` taken from a list `L1`, and the values of `y` taken from a list `L2`, such that $y \neq x$.

4. Generating the values `[a,b,c]` such that $a^2 + b^2 = c^2$.

```
Odds <== (all x\ (x : (x <-- L, 1 is x mod 2)))
Pairs <== (all y\ (v x\ ([x, y] : (x <-- L1, y <-- L2))))
Pairs <== (all y\ (v x\
   ([x, y] : (x <-- L1, y <-- L2, y =\= x))))
R <== (all c\ (all b\ (all a\
   ([a,b,c] : (a <-- L, b <-- L, a < b, c <-- L;
                a*a + b*b =:= c*c)))))
```

Here is a quick sort program using list comprehensions.

```
% sort X Y : list Y is a sorted version of list X
type sort (list A) -> (list A) -> o.
sort [] [].
sort [X | L] S :-
   Low <== (all y\ (y : (y <-- L , y < X))),
   sort Low SLow,
   Up <== (all y\ (y : (y <-- L , y > X))),
   sort Up SUp,
   append SLow [X | SUp] S.
```

4 Monad-based comprehensions vs. all-solution predicates

4.1 Expressing monad operations in terms of all-solution predicates

The differences between comprehensions if functional languages and logic programming come mostly from the presence of nondeterminism, which will tend to return instances with renamed logical variables unless instructed otherwise with appropriate quantifiers.

In λProlog-Mali, `bagof` is typed as follows:

```
type bagof (A->o) -> (list A) -> o.
mode (setof +AbstractGoal ?SetOfSols)
```

When executed it picks am unknown (logical variable) `U` and unifies the non-empty list of solutions for `U` in `(AbstractGoal U)` with `SetOfSols`. As in any Prolog, if there are unknowns in AbstractGoal (global unknowns), `bagof` will backtrack on alternative solutions corresponding to different instantiations for global unknowns.

With appropriate quantification (as available in λProlog) `bagof` can be used therefore as a practical way to implement for instance `join` and `map` as folows:

```
type join (list (list T)) -> (list T) -> o.
join Xss Ys :-
   bagof X\(sigma Xs\(member Xs Xss, member X Xs)) Ys.

type map (A -> B -> o) -> (list A) -> (list B) -> o.
map F Xs Ys :-
   bagof Y\(sigma X\(member X Xs, F X Y)) Ys.
```

Except for reversibility and behaviour on non-ground `F` map will fonction as if defined by:

```
map _ [] [].
map F [X|Xs] [Y|Ys] :- F X Y, map F Xs Y
```

i.e. a goal like

```
Xs=[A,B,B,A], map unit Xs Zs.
```

will return `Zs=[[A],[B],[B],[A]]`.

Notice that in λProlog-Mali `bagof` (and also `findall`) allows to specify its intended use through explicit quantification. However, by replacing `bagof` with `findall` *sharing* is not preserved, i.e. something like `Zs=[[_A],[_B],[_C],[_D]]`. is returned.

4.2 Defining list comprehensions with `bagof`

The following is a translation of our list comprehensions with the builtin predicate `bagof`. This translation first converts the list comprehension into an abstraction wich is given as an argument to the predicate `bagof`. The conversion is made with the predicate `get_list`. Notice the terminal case which builds the abstraction by merely tranforming a list comprehension `(T : P)` into the abstraction `x\ (x = T, P)`. Notice also the definition of the predicate `<--` which is directly mapped onto the non-deterministic predicate `member`. In our previous definition of list comprehension, the predicate `<--` did not need to be defined as it was meta-interpreted.

```
% get_list +Lc -Goal
type get_list (lcT A) -> (A -> o) -> o.
get_list (all F) y\ (sigma x\ (R x y)) :- !,
   pi x\ (get_list (F x) y\ (R x y)).
get_list (T : P) x\ (x = T, P).

X <-- Xs :- member X Xs.

Ys <== P :-
   get_list P G,
   bagof G Ys.
```

5 Metaprogramming with monads

5.1 Programs as monads of clauses

Let us start from simple (algebraic) clause unfolding operation:

Definition 1 *Let* $\mathtt{A_0}\mathtt{:-}\mathtt{A_1},\mathtt{A_2},\ldots,\mathtt{A_n}$ *and* $\mathtt{B_0}\mathtt{:-}\mathtt{B_1},\ldots,\mathtt{B_m}$ *be two clauses (suppose* $n > 0, m \geq 0$*). We define*

$$(\mathtt{A_0}\mathtt{:-}\mathtt{A_1},\mathtt{A_2},\ldots,\mathtt{A_n}) \oplus (\mathtt{B_0}\mathtt{:-}\mathtt{B_1},\ldots,\mathtt{B_m}) = (\mathtt{A_0}\mathtt{:-}\mathtt{B_1},\ldots,\mathtt{B_m},\mathtt{A_2},\ldots,\mathtt{A_n})\theta$$

with $\theta = mgu(\mathtt{A_1},\mathtt{B_0})$*. If the atoms* $\mathtt{A_1}$ *and* $\mathtt{B_0}$ *do not unify, the result of the composition is denoted as* $\perp$*.*

Furthermore, as usual, we consider $\mathtt{A_0}\mathtt{:-true},\mathtt{A_2},\ldots,\mathtt{A_n}$ *to be equivalent to* $\mathtt{A_0}\mathtt{:-}\mathtt{A_2},\ldots,\mathtt{A_n}$*, and for any clause* C, $\perp \oplus \mathtt{C} = \mathtt{C} \oplus \perp = \perp$ *. We suppose that at least one operand has been renamed apart to a variant with fresh variables.*

Prolog programs and their evaluation is expressed naturally in terms of the monad of clause unfoldings, as described by the folowing Prolog meta-program.

```
unitClause(Cs,G,[A]):-A=(G:-[G]).

joinClause(Cs,Gs,NewGs):-
   findall(NewG,expandClause(Cs,Gs,NewG),NewGs).

expandClause(Cs,Gs,NewG):-
   member(G,Gs), member(C,Cs),
   composeClause(G,C,NewG).

composeClause((H:-[]),_,(H:-[])).
composeClause((H:-[B|Bs]),(B:-NewBs),(H:-Gs)):-
   append(NewBs,Bs,Gs).
```

Notice that monad operations are parameterized with respect to a fixed set of clauses `Cs` (i.e. a given 'program').

A Prolog resolution step (depth first search with leftmost selection rule) can be conveniently described as expandClause. Multiple parallel goal rewriting is naturally expressed as joinClause.

A more efficient version is obtained by following the morphism from the monad of lists to the monad of difference lists.

5.2 An embedding of Prolog in λProlog

Using the natural monad structure induced by the Prolog style *clause unfolding* operation $\oplus$ we will give a straightforward embedding of Prolog in λProlog.

We can map an arbitrary (untyped) Prolog term to a universal type in λProlog as follows:

```
kind prologT type.

type pINT int -> prologT.
type pFLOAT float -> prologT.
type pSTRUCT (list int) -> (list prologT) -> prologT.
```

It is easy to verify that by mapping variables to variables on both sides, the mapping will commute with unfoldings. Therefore it will also commute with finite sequences of resolution steps. This allows us to write down a basic Prolog to λProlog compiler in a few dozen lines.

With this straightforward scheme, the compilation of

```
app([],Ys,Ys).
app([A|Xs],Ys,[A|Zs]) :- app(Xs,Ys,Zs).
```

becomes:

```
type demo prologT -> o.

demo (pSTRUCT "app" [(pSTRUCT "[]" []),A,A]).
demo (pSTRUCT "app" [(pSTRUCT "." [A,B]),C,
                     (pSTRUCT "." [A,D])]) :-
   demo (pSTRUCT "app" [B,C,D]).
```

In practice, the scheme can be refined by having the mapping act as **id** on the set of builtins and Prolog's CUT operation. By mapping predicates to predicates and generating trivial type declarations for the arguments we obtain a practical embedding of Prolog in λProlog which is useful to run existing untyped Prolog applications.

6 Related work

Despite the proven expressiveness of monadic style in functional programming [16, 17] and denotational semantics [10] no systematic exploration of the concept has been done for logic programming languages. However, ideas alike in substance are present in Peter VanRoy's Extended DCGs [15] and their Wild-Life [1] counterparts, which encapsulate a concept of state similar to the state transformers in functional programming [16] and some of the morfisms we have described are 'implicitely' known to good Prolog programmers. This situation largely motivates our effort as both major declarative programming paradigms share the reasons for adopting monads in every-day programming.

7 Future work

The following are some of the most attempting follow-ups to the experiments described in this paper:

- state transformers in logic programming
- an implementation of arbitrary monad comprehensions
- the monad of the answers of a (nondeterministic) logic program
- Fold/Unfold transformations and monad operations
- transforming meta-interpreters with monad operations
- a monadic view of continuation passing and binarization in logic programming

For instance, meta-interpreters for logic programming languages can be described in terms of the monad of clause unfoldings. All-solution predicates can be seen as morphisms from a suitably organized monad of answers to the monad of lists. Knowing their monad properties allows to transform (i.e. 'partially evaluate') all-solution predicates in a rigorous way to their more efficient first-order equivalents.

8 Conclusion

We have shown that monads are as useful to describe the basic data structures of logic programming languages as they are in functional programming. In particular we have organized function lists as a monad and studied some interesting monad morphisms. We have introduced a monad-based concept of lists comprehensions for λProlog. We have described Prolog's unfolding operation in a monadic style, with a practical compilation scheme from untyped Prolog

to λProlog. The use of an intrinsically high order logic programming language (λProlog) has been shown particularly useful, although most of our techniques will also work in ordinary Prolog systems. Although transposing an elegant *declarative* concept from one declarative programming paradigm to another is not difficult, their elegant implementation was not always obvious. Moreover, the monadic view of lazy function lists and meta-programming is novel and it looks like a good starting point for a uniform description for the semantics of logic programs and for program transformations.

Acknowledgment

Paul Tarau thanks for support from the Canadian National Science and Engineering Research Council (grant OGP0107411), the FESR of the Université de Moncton and for the fellowships from Université de Rennes and the I.R.I.S.A Rennes. Discussions with members of the LANDE group and especially Pascal Brisset, Pascal Fradet, Daniel LeMetayer and Olivier Ridoux have been very helpful in clarifying our ideas on the subject.

References

[1] H. Aït-Kaci and A. Podelski. Towards a meaning of LIFE. In J. Maluszyński and M. Wirsing, editors, *Proceedings of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming, PLILP91, Passau, Germany*, number 528. Springer Verlag, Aug. 1991.

[2] Y. Bekkers, B. Canet, O. Ridoux, and L. Ungaro. MALI: A memory with a real-time garbage collector for implementing logic programming languages. In *3rd Symp. Logic Programming*, Salt Lake City, UT, USA, 1986. IEEE.

[3] P. Brisset. Compilation de λProlog. Thèse, Université de Rennes I, 1992.

[4] P. Brisset and O. Ridoux. Naïve reverse can be linear. In K. Furukawa, editor, *8th Int. Conf. Logic Programming*, pages 857–870, Paris, France, 1991. MIT Press. ftp: //ftp.irisa.fr/local/lande.

[5] P. Brisset and O. Ridoux. The architecture of an implementation of λProlog: Prolog/Mali. In *Workshop on λProlog*, Philadelphia, PA, USA, 1992. ftp: //ftp.irisa.fr/local/lande.

[6] J. Goguen. Higher order functions considered unnecessary for higher order programming. Technical report, SRI International, Jan. 1989. Technical report SRI-CSL-88-1.

[7] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Logic and Computation*, 1(4):497–536, 1991.

[8] D. Miller. Unification of simply typed lambda-terms as logic programming. In K. Furukawa, editor, *8th Int. Conf. Logic Programming*, pages 255–269, Paris, France, 1991. MIT Press.

[9] D. Miller, G. Nadathur, and A. Scedrov. Hereditary Harrop formulas and uniform proof systems. In D. Gries, editor, *2nd Symp. Logic in Computer Science*, pages 98–105, Ithaca, New York, USA, 1987.

[10] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[11] A. Mycroft and R. A. O'Keefe. A polimorphic type system for prolog. *Artificial Intelligence*, (23):295–307, 1984.

[12] G. Nadathur and D. Miller. An overview of λProlog. In K. Bowen and R. Kowalski, editors, *Symp. Logic Programming*, pages 810–827, Seattle, Washington, USA, 1988.

[13] G. Nadathur and D. Miller. Higher-order Horn clauses. *JACM*, 37(4):777–814, 1990.

[14] G. Nadathur and D. Wilson. A representation of lambda terms suitable for operations on their intensions. In *ACM Conf. Lisp and Functional Programming*, pages 341–348, Nice, France, 1990. ACM Press.

[15] P. V. Roy. A useful extension to Prolog's Definite Clause Grammar notation. *SIGPLAN notices*, 24(11):132–134, Nov. 1989.

[16] P. Wadler. Comprehending monads. In *ACM Conf. Lisp and Functional Programming*, pages 61–78, Nice, France, 1990. ACM Press.

[17] P. Wadler. The essence of functional programming. In *ACM Symposium POPL'92*, pages 1–15. ACM Press, 1992.

[18] D. H. D. Warren. Higher-order extensions to Prolog – are they needed? In D. Michie, J. Hayes, and Y. H. Pao, editors, *Machine Intelligence 10*. Ellis Horwood, 1981.

Word count:

```
dvi2tty -l -w132 art | wc
     943    4969   55597
```