SUBMITTED TO ICLP '99

Los Cruces New Mex

# Algebra of Logic Programming (2)

Silvija Seres Michael Spivey Tony Hoare

*Oxford University Computing Laboratory*
*Wolfson Building, Parks Road, Oxford OX1 3QD, U.K.*

**Abstract**

A declarative programming language has two kinds of semantics. The more abstract helps in reasoning about specifications and correctness, while an operational semantics determines the manner of program execution. A correct program should reconcile its abstract meaning with its concrete interpretation.

To help in this, we present a kind of algebraic semantics for logic programming. It lists only those laws that are equally valid for predicate calculus and for the standard depth-first strategy of Prolog. An alternative strategy is breadth-first search, which shares many of the same laws. Both strategies are shown to be special cases of the most general strategy, that for tree searching. The three strategies are defined in the lazy functional language Haskell, so that each law can be proved by standard algebraic reasoning. The laws are an enrichment of the familiar categorical concept of a monad, and the links between such monads are explored.

## 1 Introduction

In an earlier paper [5] we have proposed a simple and direct embedding of the main logical constructs of Prolog in a lazy functional language. Its use of lazy lists gives rise to a natural implementation of the possibly infinite search-space and the depth-first search strategy of Prolog. We call this the *stream model* of logic programming. We have described in [5] how the embedding can be changed to implement breadth-first search by lifting the operations to streams of lists, where each list has one higher cost that the predecessor; we call this the *matrix model.* In this paper we offer a third, more flexible model based on lists of tree which accommodates *both* search strategies; we call this the *forest model* of logic programming.

We therefore have three different implementations of a logic programming language. In comparing their logical and computational properties we concentrate on those algebraic laws which are shared in all three execution models, and in this sense the three models prove to be strongly consistent with each other and with the declarative reading. We use category theory to specify how the three

models are related as ***semi-distributive monads.*** We define three such monads – extensions of the stream, matrix and forest monads – that satisfy certain laws and that capture each of the computation models. Then we claim the existence of unique mappings between the monad corresponding to the most general model and the other two monads. The two mappings correspond exactly to the depth-first and breadth-first traversal of the search tree of a logic program.

The embedding of logic into functional programming is achieved by translating each primitive to a function or an operator. In this manner we can separately implement the logic operators and the search-control operators, so this is a realisation of Kowalski's slogan that programs are logic plus control. Our implementation is arguably the simplest possible formalisation of different operational semantics of logic programming and can be thought of as an executable operational semantics for logic programming.

We describe the concrete implementations in the functional language Haskell, for three reasons: it is rewarding to have a concrete, working prototype to experiment with; the functional languages are easy to interpret in terms of the category theory; and the proofs of the proposed algebraic laws only use the standard algebra of functions. As an alternative to Haskell any lazy language (with the Hindley-Milner type system) and $\lambda$-abstractions would do. Our implementation shows that any such functional language contains much of the expressive power of functional logic languages, but to achieve the full power of these languages further extensions (e.g. typed unification) are needed.

In section 2 we describe the ideas behind our translation of logic programs to functional ones, and in sections 3, 4 and 5 we outline the implementations of three different execution models of logic programs: one using depth-first search, one using breadth-first search and one model accommodating both search strategies. In section 6 we show how these three models can be understood in terms of monads in category theory, and in section 7 we prove the existence of shape-preserving morphisms between the forest model and the other two. Except in these two sections, we do not assume any knowledge of category theory, but some basic knowledge of functional programming is needed.

# 2 Functional Interpretation of Logic Programs

In our embedding of logic programs into a functional language, we aim to give rules that allow any pure Prolog program to be translated into a functional program with the same meaning. To this end, we introduce two data types *Term* and *Predicate* into our functional language, together with the following four operations:

$$\begin{aligned}
\&, \| &: \quad Predicate \to Predicate \to Predicate,\\
\doteq &: \quad Term \to Term \to Predicate,\\
\exists &: \quad (Term \to Predicate) \to Predicate.
\end{aligned}$$

The intention is that the operators & and || denote conjunction and disjunction of predicates, the operator $\doteq$ forms a predicate expressing the equality of two terms, and the operation $\exists$ expresses existential quantification. In terms of logic programs, we will use & to join literals of a clause, || to join clauses, $\doteq$ to express the primitive unification operation, and $\exists$ to introduce fresh local variables in a clause.

These four operations suffice to translate any pure Prolog program into a functional program. As an example, we take the well-known program for append:

```
append([], Ys, Ys) :- .
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

As a first step, we remove any patterns and repeated variables from the head of each clause, replacing them by explicit equations written at the start of the body. The head of each clause then contains only a list of distinct variables. By renaming, we can ensure that the list of variables is the same in each clause. We complete the translation to Haskell by joining the clause bodies with the || operation, the literals in each clause by the & operator, and existentially quantifying any variables that appear in the body but not in the head of a clause:

$$\begin{aligned}&append(Ps, Qs, Rs) = \\ &\qquad (Ps \doteq nil \mathbin{\&} Qs \doteq Rs) \parallel \\ &\qquad (\exists X, Xs, Ys \rightarrow Ps \doteq cons(X, Xs) \mathbin{\&} Rs \doteq cons(X, Ys) \mathbin{\&} \\ &\qquad\qquad\qquad append(Xs, Qs, Ys)).\end{aligned}$$

Here *nil* is used for the value of type *Term* representing the empty list, and *cons* is written for the function on terms corresponding to the Prolog constructor `[|]`. The function *append* defined by this recursive equation has type:

$$append : (Term, Term, Term) \rightarrow Predicate.$$

This translation to a functional program obviously respects the *declarative* semantics of the original logic program; the operators &, ||, $\doteq$ and $\exists$ have as their main role to make the declarative semantics of the logic program explicit. We will define and implement these four basic operators such that the translation *also* respects the *execution* semantics of the logic program. In fact, we show that this translation can be adapted to three different execution models.

The four operators can be divided into two groups: we call & and || the *structuring* operators, and $\doteq$ and $\exists$ the *unification* operators. The choice of a concrete scheduling strategy affects mainly the basic type of predicates and the structuring operators, so the focus of this paper will be on the implementation and analysis of these. A closer semantical study of the unification operators is a task we pursue in another paper; they have interesting monotonicity properties and are important in our study of typed unification, program transformation, functional logic programming and other topics.

The implementation details of all the four operators for both the depth-first and breadth-first computation models are described in [5]. We now give an outline of the differences between these two computation models, and in section 5 we describe a model that accommodates both search strategies.

## 3 The Depth-First Search Strategy

The key idea of all the three implementations is that each predicate is a function taking an 'answer' (that represents the state of knowledge about the values of variables at the time the predicate is solved), and producing a *collection* of answers, where each answer corresponds to a solution of the predicate that is consistent with the input. An answer is in principle just a substitution, i.e. an associative list mapping each value to the term that is to be substituted for it.

The representation, or type, of the collections of answers is the main difference between the three models. In the depth-first model, the set of answers is represented by a lazy list, or stream:[1]

$$\mathbf{type}\ \mathit{Predicate} = \mathit{Answer} \rightarrow \mathit{Stream\ Answer}.$$

The $\|$ operator computes the answers to the disjunction of two predicates by concatenating the streams of answers returned by its two operands. The & operator computes the answers to the conjunction of two predicates by a pairwise unification of the answers from the two streams computed by its two arguments. It combines all the answers from its two arguments by first applying the left-hand predicate to the incoming answer, followed by applying the right-hand predicate to each of the answers in the resulting stream. Finally, to preserve the types, concatenation of the resulting stream of streams into a single stream is needed. The definitions of $\|$ and & are thus:

$$(p \parallel q)\ x = q\ x +\!\!+\ q\ x, \qquad (1)$$

$$p\ \&\ q = \mathit{concat} \cdot \mathit{map}\ q \cdot p. \qquad (2)$$

We also define constant predicates *true* and *false*, one corresponding to immediate success and the other to immediate failure:

$$\mathit{true} :: \mathit{Predicate} \qquad \mathit{false} :: \mathit{Predicate}$$

$$\mathit{true}\ x = [x], \qquad \mathit{false}\ x = [\,].$$

A series of interesting algebraic properties can be proved for the operators & and $\|$. The proofs are based on equational reasoning using the definitions of the operators, the associativity property of functional composition $(\cdot)$ and the

[1] For clarity, we use the type constructor *Stream* to denote infinite streams, and *List* to denote finite lists. In a lazy functional language, these two concepts share the same implementation.

definitions of the standard functions *map*, *concat* and their following well-known properties (see [2]):

$$map\ (f \cdot g) = (map\ f) \cdot (map\ g), \tag{3}$$

$$map\ f \cdot concat = concat \cdot map\ (map\ f), \tag{4}$$

$$concat \cdot map\ concat = concat \cdot concat. \tag{5}$$

The laws are listed below. The operator & is associative (6) with unit element *true* (7). The predicate *false* is a left zero for & (8), but the & operator is strict in its left argument, so *false* is not a right zero. The || operator is associative (9) and has *false* as its unit element (10). The & operator distributes through || from the right (11):

$$(p \mathbin{\&} q) \mathbin{\&} r = p \mathbin{\&} (q \mathbin{\&} r), \tag{6}$$

$$p \mathbin{\&} true = true \mathbin{\&} p = p, \tag{7}$$

$$false \mathbin{\&} p = false, \tag{8}$$

$$(p \parallel q) \parallel r = p \parallel (q \parallel r), \tag{9}$$

$$p \parallel false = false \parallel p = p, \tag{10}$$

$$(p \parallel q) \mathbin{\&} r = (p \mathbin{\&} r) \parallel (q \mathbin{\&} r). \tag{11}$$

Other identities that are satisfied by the connectives of propositional logic are not shared by our operators because in our stream-based implementation, answers are produced in a definite order and with definite multiplicity. This behaviour mirrors the operational behaviour of Prolog. For example, neither || nor & are idempotent because the number of answers to $p \parallel p$ and to $p \mathbin{\&} p$ is not the same as the the number of answers to $p$ except in the trivial cases.

Some of the missing laws could be reestablished if bags or sets were used to collect the answers. For example, if we used bags instead of streams, the order of answers would not matter. In that case both & and || would become commutative, and & could distribute through || also from the left. If sets were used instead, & and || would in addition become idempotent. The problem with using bags or sets is that in the infinite case their equality cannot be algorithmically defined. Furthermore, some other standard laws from propositional logic, for example the distributivity of || through &, or *false* being a right zero for ||, could be established as inequalities if an ordering on predicates was used.

We can also define the predicate operators *not* and *cut*:

$$
\begin{array}{ll}
not :: Predicate \to Predicate & cut :: Predicate \to Predicate \\
not\ p = true\ \mathit{if}\ p == false, & cut\ p\ x = [\,]\ \mathit{if}\ p\ x == [\,] \\
 & \qquad\qquad \mid\ [head\ (p\ x)]\ otherwise.
\end{array}
$$

The *cut* operator defined here does not exactly correspond to the cut operator in Prolog, because we have made it into an operator on predicates rather than a single predicate like in Prolog. This possibly influences the way cut behaves

with respect to backtracking. As compensation for our lack of faith to Prolog, we can define a set of algebraic properties, useful for optimising purposes among others:

$$cut\ (cut\ p) = cut\ p, \tag{12}$$
$$cut\ (false) = false, \tag{13}$$
$$cut\ (p\ \&\ q) = cut\ (p\ \&\ (cut\ q)), \tag{14}$$
$$cut\ (p \parallel q) = cut\ ((cut\ p) \parallel q), \tag{15}$$
$$not\ (not\ (not\ p)) = not\ p, \tag{16}$$
$$not\ (p \parallel q) = (not\ p)\ \&\ (not\ q). \tag{17}$$

The operator *cut* is idempotent (12), and has *false* as its zero element (13). It does not distribute through & nor ||, but it does satisfy the equations (14) and (15). The operator *not* is not idempotent, but does satisfy (16). Only one of the De Morgan laws holds (17).

# 4 The Breadth-First Search Strategy

In a model that allows breadth-first search, we need to maintain the information about the computational cost for each answer. The cost of an answer is measured by the number of resolution steps required in its computation. Therefore, the predicates in our breadth-first model return a stream of bags[2], or a *matrix*, of answers. Each bag represents the finite number of answers reached at the same depth, or level, of the search tree. All such bags are finite because there are only a finite number of branches in each node in the search tree, so the bag equality in this case is always computable. Intuitively, each successive bag of answers in the stream contains the answers with the same computational 'cost'. The type of *Predicate* is thus:

$$Predicate :: Answer \rightarrow Matrix\ Answer,$$
$$\textbf{type}\ Matrix\ a = Stream\ Bag\ a.$$

The bookkeeping of the resolution costs for each of the answers to a predicate is implemented by the function *step*, with type *Predicate* → *Predicate*, and the definition of each predicate implementation needs to be changed to perform a call to *step* on the outermost level. In the depth-first model, *step* is the identity function on predicates, because the cost of answers is irrelevant. In the breadth-first model it increases the cost of computation of the predicate by one. It does this by shifting all the bags of answers one position to the right in the main

[2]If lists were used instead of bags, the definite ordering of answers would imply the loss of associativity for the & operator in this model. The underlying implementation of bags and lists is same in a functional language, but we use the *Bag* type constructor to stress the semantical difference, and we use bag equality for all our algebraic laws.

stream:

$$step\ p\ x = [\,] : (p\ x).$$

In the breadth-first model, the implementations of the operators || and & need to be adapted to preserve the cost information that is embedded in the input matrices. The || operator simply zips the two matrices into a single one, using the function *zipwith* which concatenates all the bags of answers with the same cost and returns a single stream of these new bags. The & operator has to add the costs of its arguments; the idea is first to compute all the answers to $p$, then map $q$ on each answer in the resulting matrix by the matrix-map function *mmap*, and then to use function *shuffle* to flatten the resulting matrix of matrices to a single matrix, according to the cost:

$$(p \parallel q)\ x = zipwith\ (+\!\!+)\ (p\ x)\ (q\ x), \tag{18}$$

$$p \,\&\, q = shuffle \cdot mmap\ q \cdot p. \tag{19}$$

The function *mmap* is simply a composition of *map* with itself, and the function *zipwith* is a generalisation of the standard function *zipwith* such that it does not stop when it reaches the end of the shortest of its two argument streams. The implementation of *shuffle* is too technical to be included here, please see [5] for details. From the definitions of these functions it can be proved by structural induction on the matrices that they enjoy the following algebraic properties:

$$mmap\ (f \cdot g) = (mmap\ f) \cdot (mmap\ g), \tag{20}$$

$$mmap\ f \cdot shuffle = shuffle \cdot mmap\ (mmap\ f), \tag{21}$$

$$shuffle \cdot mmap\ shuffle = shuffle \cdot shuffle, \tag{22}$$

$$zipwith\ f\ (zipwith\ f\ l_1\ l_2)\ l_3 = zipwith\ f\ l_1\ (zipwith\ f\ l_2\ l_3), \tag{23}$$

$$mmap\ f \cdot zipwith\ g\ l_1\ l_2 = zipwith\ g\ (mmap\ f\ l_1)\ (mmap\ f\ l_2), \tag{24}$$

$$shuffle \cdot zipwith\ (+\!\!+)\ l_1\ l_2 = zipwith\ (+\!\!+)\ (shuffle\ l_1)\ (shuffle\ l_2). \tag{25}$$

The law (23) requires that $f$ is associative, and the law (24) holds if $f$ and $g$ commute. For these equalities to hold it is necessary to interpret the equality sign in the laws as equality of *streams of bags* rather than a stream equality.

The predicate *false* in this model has the same implementation as in the stream model: it has no answers at any cost level so it is simply the empty stream [ ]. The predicate *true* has to be lifted to matrices, where it returns its input answer as its only answer at level 0, and has no other answers. The predicate operator *not* stays the same as in the stream model, it returns true if its input predicate equals *false*, while *cut* needs to be lifted to return the matrix containing only the first element in the first non-empty bag:

$$true\ x = [\,[x]\,],$$

$$cut\ p\ x = [\,[\,head \cdot concat \cdot map\ first\ (p\ x)\,]\,],$$

where *first* returns the singleton list containing the first element of its input list if it is non-empty, or an empty list otheriwse.

All the algebraic laws for *true*, *false*, & and || listed in the previous section hold in this model too, and all the laws for *not* and *cut* hold except for (14-15). The proofs of these laws are again based on equational reasoning, using the definitions of the operators, the associativity of functional composition, and the properties (20-25). For example, in the proof of the associativity of ||, we use the associativity property of *zipwith* (23), and in the proof of the right-distributivity of & through || we use the distributivity properties of *mmap* and *shuffle* through *zipwith* (24-25). As a concrete example we show the proof of the associativity of & in this model:

$$\begin{aligned}
&(p \mathbin{\&} q) \mathbin{\&} r \\
&= \mathit{shuffle} \cdot \mathit{mmap}\ r \cdot \mathit{shuffle} \cdot \mathit{mmap}\ q \cdot p && \text{by (19)} \\
&= \mathit{shuffle} \cdot \mathit{shuffle} \cdot (\mathit{mmap}\ \mathit{mmap}\ r) \cdot \mathit{mmap}\ q \cdot p && \text{by (21)} \\
&= \mathit{shuffle} \cdot \mathit{mmap}\ \mathit{shuffle} \cdot (\mathit{mmap}\ \mathit{mmap}\ r) \cdot \mathit{mmapq} \cdot p && \text{by (22)} \\
&= \mathit{shuffle} \cdot \mathit{mmap}\ (\mathit{shuffle} \cdot \mathit{mmap}\ r \cdot q) \cdot p && \text{by (20)} \\
&= p \mathbin{\&} (q \mathbin{\&} r) && \text{by (19)}
\end{aligned}$$

As in the previous model, the additional properties of the propositional logic operators could be established as equalities on bags or sets (rather than equalities on streams of bags), or as inequalities using the subsumption ordering on predicates. For the same constructive reasons as before, we choose not to use these.

# 5 The General Model

In the model that allows the use of *both* depth-first and breadth-first search the predicates can be modelled by functions returning lists of trees of answers, or *forests* of answers. The type of *Answer* is same as before. Each inner node in a tree can have an arbitrary number of children; this can be implemented by collecting all the children nodes in a new forest:

$$\begin{aligned}
&\textbf{type}\ \mathit{Predicate} = \mathit{Answer} \to \mathit{Forest}\ \mathit{Answer}, \\
&\textbf{type}\ \mathit{Forest}\ a = \mathit{List}\ \mathit{Tree}\ a, \\
&\textbf{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ a \mid \mathit{Fork}\ (\mathit{Forest}\ a).
\end{aligned}$$

The cost of an answer corresponds to its depth in the search tree. Consequently, the function *step* pushes all the computed answers one level down the tree by adding a new parent node as a root. It forms a tree from the input forest of answers and for type correctness converts this tree to a singleton forest:

$$\mathit{step}\ p\ x = [\,\mathit{Fork}\ (p\ x)\,].$$

The implementations of || and & operators in this model are similar to the implementations in the stream model. The || operator actually stays the same,

it simply concatenates the two forests of answers, and the costs do not change. The & in the forest model is the lifting of the original & to the forest type. The left-hand argument to & returns a list of trees. The right-hand argument is then applied to all the answers in the resulting list – which are simply all the leafs of each tree in the list – by the function *fmap*. This results in a forest of answers at each leaf, and these are grafted into the tree by the function *fgraft*:

$$(p \parallel q)\ x = p\ x +\!\!+\ q\ x, \tag{26}$$

$$p\ \&\ q = \mathit{fgraft} \cdot \mathit{fmap}\ q \cdot p. \tag{27}$$

The motivation for choosing forests rather than just trees for the type of answers is that $\parallel$ and & cannot be cost-preserving on trees. If simple trees were used, $p \parallel q$ would have to combine their trees of answers by inserting them under a new parent node in a new tree, but that would increase the cost of each answer to $p \parallel q$ by one. For example, the answers to $p \parallel \mathit{no}$ would in the tree model cost more than the answers to $p$, which would be wrong – the number of resolution steps performed is the same. Also, in the tree model the $\parallel$ operation would not be associative.

The following definitions of the auxiliary functions *graft*, *graft2*, *fmap* and *tmap* are needed in the proofs in later sections:

$$\mathit{fgraft} = \mathit{concat} \cdot \mathit{map}\ \mathit{graft2}, \tag{28}$$

$$\mathit{graft2}\ (\mathit{Leaf}\ \mathit{xf}) = \mathit{xf}, \tag{29}$$

$$\mathit{graft2}\ (\mathit{Fork}\ \mathit{xff}) = [\,\mathit{Fork}\ (\mathit{fgraft}\ \mathit{xff})\,], \tag{30}$$

$$\mathit{fmap}\ f = \mathit{map}\ (\mathit{tmap}\ f), \tag{31}$$

$$\mathit{tmap}\ f\ (\mathit{Leaf}\ x) = \mathit{Leaf}\ (f\ x), \tag{32}$$

$$\mathit{tmap}\ f\ (\mathit{Fork}\ \mathit{xf}) = \mathit{Fork}\ (\mathit{fmap}\ f\ \mathit{xf}). \tag{33}$$

From these definitions it can be proved by structural induction that *fmap* and *fgraft* both distribute through $+\!\!+$ and that they share the standard properties of the functions *map*, *concat* and functional composition:

$$\mathit{fmap}\ (f \cdot g) = (\mathit{fmap}\ f) \cdot (\mathit{fmap}\ g), \tag{34}$$

$$\mathit{fmap}\ f \cdot \mathit{fgraft} = \mathit{fgraft} \cdot \mathit{fmap}\ (\mathit{fmap}\ f), \tag{35}$$

$$\mathit{fgraft} \cdot \mathit{fgraft} = \mathit{fgraft} \cdot \mathit{fmap}\ \mathit{fgraft}. \tag{36}$$

As an example we give the proof of (36). The function *fgraft* is defined through indirect recursion with the function *graft2*, so a proof of (36) requires a simultaneous inductive proof of the equation (37):

$$\mathit{fgraft}\ \cdot \mathit{graft2} = \mathit{graft2} \cdot \mathit{tmap}\ \mathit{fgraft}. \tag{37}$$

Assuming that (37) holds, we prove (36):

$$
\begin{aligned}
& \mathit{fgraft} \cdot \mathit{fgraft} \\
&= \mathit{concat} \cdot \mathit{map}\ \mathit{graft2} \cdot \mathit{concat} \cdot \mathit{map}\ \mathit{graft2} && \text{by (28)} \\
&= \mathit{concat} \cdot \mathit{concat} \cdot \mathit{map}\ (\mathit{map}\ \mathit{graft2}) \cdot \mathit{map}\ \mathit{graft2} && \text{by (4)} \\
&= \mathit{concat} \cdot \mathit{map}\ \mathit{concat} \cdot \mathit{map}\ (\mathit{map}\ \mathit{graft2}) \cdot \mathit{map}\ \mathit{graft2} && \text{by (5)} \\
&= \mathit{concat} \cdot \mathit{map}\ (\mathit{concat} \cdot \mathit{map}\ \mathit{graft2} \cdot \mathit{graft2}) && \text{by (3)} \\
&= \mathit{concat} \cdot \mathit{map}\ (\mathit{fgraft} \cdot \mathit{graft2}) && \text{by (28)} \\
&= \mathit{concat} \cdot \mathit{map}\ (\mathit{graft2} \cdot \mathit{tmap}\ \mathit{graft}) && \text{by (37)} \\
&= \mathit{concat} \cdot \mathit{map}\ \mathit{graft2} \cdot \mathit{map}\ (\mathit{tmap}\ \mathit{graft}) && \text{by (3)} \\
&= \mathit{graft} \cdot \mathit{fmap}\ \mathit{graft} && \text{by (28,31)}
\end{aligned}
$$

To prove (37), we need to look at both inductive cases. The proof of the base case, $\mathit{fgraft}\ (\mathit{graft2}\ \mathit{Leaf}\ \mathit{xf}) = \mathit{graft2}\ (\mathit{tmap}\ \mathit{fgraft}\ \mathit{xft})$, follows trivialy from the definitions (29) and (32). In the induction case, if $\mathit{xft} = \mathit{Fork}\ \mathit{xf}$ and the induction hypothesis (36) holds of $\mathit{xf}$, we find:

$$
\begin{aligned}
& \mathit{fgraft} \cdot \mathit{graft2}\ (\mathit{Fork}\ \mathit{xf}) \\
&= \mathit{fgraft}\ [\,\mathit{Fork}\ (\mathit{fgraft}\ \mathit{xf})\,] && \text{by (30)} \\
&= \mathit{concat}\ (\mathit{map}\ \mathit{graft2}\ [\,\mathit{Fork}\ (\mathit{fgraft}\ \mathit{xf})\,]) && \text{by (28)} \\
&= \mathit{concat}\ [\,\mathit{graft2}\ (\mathit{Fork}\ (\mathit{fgraft}\ \mathit{xf}))\,] && \text{by } (\mathit{map}) \\
&= [\,\mathit{graft2}\ (\mathit{Fork}\ (\mathit{fgraft}\ \mathit{xf}))\,] && \text{by } (\mathit{concat}) \\
&= [\,\mathit{Fork}\ (\mathit{fgraft}\ (\mathit{fgraft}\ \mathit{xf}))\,] && \text{by (30)} \\
&= [\,\mathit{Fork}\ (\mathit{fgraft}\ (\mathit{fmap}\ \mathit{fgraft}\ \mathit{xf}))\,] && \text{by (36)} \\
&= \mathit{graft2}\ (\mathit{Fork}\ (\mathit{fmap}\ \mathit{fgraft}\ \mathit{xf})) && \text{by (30)} \\
&= \mathit{graft2} \cdot \mathit{tmap}\ \mathit{fgraft}\ (\mathit{Fork}\ \mathit{xf}) && \text{by (33)}
\end{aligned}
$$

The predicate operator *not* stays the same as in the stream model, it returns true if its input predicate equals *false*. Since the search strategy is not specified for this model, there can be several different definitions of the *cut* operator such that it is idempotent, monotonous and returns the first answer relative to some search strategy. Once again, the predicate *false* in this model has the same implementation as in the stream model; it has no sub-trees, so it always returns the empty stream [ ]. The predicate *true* has to be lifted to forests, where it returns its input answer as its only answer at level 0 in the first subtree, and has no other answers:

$$\mathit{true}\ x = [\,\mathit{Leaf}\ x\,].$$

The same algebraic laws for *true*, *false*, & and || hold of this model as of the previous two, and the equalities in this case need to be interpreted as equalities on streams of trees. The proofs are based on equational reasoning: the laws regarding *true* and *false* follow directly from the definitions of the operators,

the associativity of $\|$ follows from the associativity of $+\!\!+$, the associativity of $\&$ has a similar proof as for matrices and uses (34-36), and the proof of the distributivity of $\&$ through $\|$ from the left uses the distributivity of *fgraft* and *fmap* through $+\!\!+$.

# 6 The Three Monads

We have so far described implementations of three scheduling strategies for logic programming, and we have seen that the same set of algebraic laws holds for the structuring operators of each model. The aim of this section is to present the mathematical framework which will help us explore and express the relationships between our three models.

Phil Wadler has shown in [6, 7] that many aspects of functional programming, for example laziness or eagerness of evaluation, non-determinism and handling of input and output, can be captured by the monad construction from category theory. Here we show in a similar fashion how our models of logic programming relate to concepts from category theory.

A monad $T$ is a triple $(map_T, unit_T, join_T)$, where $T$ is a type constructor $T$ with an associated function $map_T$, and $unit_T$ and $join_T$ are polymorphic functions, with types:

$$\begin{aligned}
&map_T :: (a \to b) \to T\ a \to T\ b,\\
&unit_T :: a \to T\ a,\\
&join_T :: T\ (T\ a) \to T\ a.
\end{aligned}$$

In addition, we use $id_T$ for the identity function on each $T$. For such a triple to qualify as a monad, the following equalities must be satisfied:

$$map_T\ id_T = id_T, \tag{38}$$

$$map_T\ (f \cdot g) = map_T\ f \cdot map_T\ g, \tag{39}$$

$$map_T\ f \cdot unit_T = unit_T \cdot f, \tag{40}$$

$$map_T\ f \cdot join_T = join_T \cdot map_T\ (map_T\ f), \tag{41}$$

$$join_T \cdot unit_T = id_T, \tag{42}$$

$$join_T \cdot map_T\ unit_T = id_T, \tag{43}$$

$$join_T \cdot map_T\ join_T = join_T \cdot join_T. \tag{44}$$

Taking *Stream* for $T$, the standard stream function *map* for $map_T$, the list unit constructor $[-]$ for $unit_T$, and *concat* for $join_T$, one can easily verify that $(map, [-], concat)$ is a monad. The equations (39), (41) and (44) correspond to the standard laws for list operators (3-5); the rest of the equations follow from the definitions of *map* and *concat*.

The *Matrix* monad results from taking *mmap* for $map_T$, the matrix unit constructor $[[-]]$ for $unit_T$ and *shuffle* for $join_T$. The equations (39), (41) and (44)

for this monad correspond to the equations (20-22), and the remaining equations can be proved from the definitions of the matrix functions *mmap* and *shuffle*.

Finally, the *Forest* monad results from taking the function *fmap* for $map_T$, the forest unit constructor [*Leaf* −] for $unit_T$ and *fgraft* for $join_T$. Again, the equations (39), (41) and (44) for this monad are the same as the equations (34-36) described in section 5, and the remaining ones can be proved from the definitions of the forest functions *fmap* and *fgraft*.

There is an alternative definition of a monad, where the function $join_T$ is replaced by the operator $\star_T$, also called the Kleisli composition, defined as:

$$(\star) :: (a \to T\ b) \to (b \to T\ c) \to (a \to T\ c), \tag{45}$$

$$p \star_T q = join_T \cdot map_T\ q \cdot p. \tag{46}$$

The triple $(map_T, unit_T, \star_T)$ is a monad if the equations (47-49) given below are satisfied. These equations are implied by the equations (38-44):

$$(unit_T\ a) \star k = k\ a, \tag{47}$$

$$m \star unit_T = m, \tag{48}$$

$$m \star (p \star q) = (m \star p) \star q. \tag{49}$$

Conversely, the functions $map_T$ and $join_T$ can be defined in terms of $\star_T$, and the equations (47-49) imply the original monad equations (38-44), so the two alternative definitions of a monad are equivalent.

This second definition of a monad is particularly convenient for our purposes, since the operator $\star_T$ corresponds exactly to the definition of the operator & in each of the models, and $unit_T$ corresponds to our function *true* in each model. We have already seen that in all three models & is associative with unit element *true*, so the laws (47-49) are satisfied in the *Stream*, *Matrix* and *Forest* monads. In that sense we can say that they capture the algebraic semantics of the operator & and predicate *true* in our three models of logic programming. The remaining structural parts of each model are || and *false*. We now formulate the right notion of the extended monad that captures these and their properties.

In an application to our models of logic programming, we define an *extended* monad $T^+$ as a five-tuple, where $T$ is one of *Stream*, *Matrix* and *Forest* monads, $true_T$ is the unit, and $\&_T$ is the Kleisli composition in each of the monads:

$$T^+ = (map_T,\ true_T,\ false_T,\ \|_T,\ \&_T),$$

such that the laws for || and *false* listed in section 3 hold. The extended monads *Stream*$^+$, *Matrix*$^+$ and *Forest*$^+$ capture the algebraic semantics of the three different scheduling strategies for logic programming.

A *morphism* between two extended monads is a mapping which preserves the structure of the monads, i.e. which maps *true* in one monad to *true* in the other and so on with *false*, $\|_T$ and $\&_T$. We now proceed to show the existence of monad morphisms between the third, most general, extended monad and the other two.

# 7 Relationships between the Monads

In the most general model, each query to a logic program returns a forest corresponding to the search tree of the query. This forest can be converted to a stream of answers, by traversing the trees in either a depth-first or breadth-first manner. The functions *dfs* and *bfs* below, with type $\mathit{Forest}\ a \to \mathit{Stream}\ a$, implement these two search strategies.

The *dfs* function applies the auxiliary *dfs2* function to each tree in the list and concatenates the resulting lists. The function *dfs2* returns the leaf nodes of each tree in a depth-first manner, by recursively calling *dfs*:

$$\mathit{dfs} = \mathit{concat} \cdot \mathit{map}\ \mathit{dfs2}, \tag{50}$$
$$\mathit{dfs2}\ (\mathit{Leaf}\ x) = [x], \tag{51}$$
$$\mathit{dfs2}\ (\mathit{Fork}\ \mathit{xf}) = \mathit{dfs}\ \mathit{xf}. \tag{52}$$

The *bfs* function needs to take account of the cost of the answers. It does this by collecting all the answers from a same level in all the search trees of the input forest in a same list, and by returning the lists in an increasing order of level. The function *bfs2* performs the sorting of input answers with respect to their cost. The function *levels* sorts the leafs of a input tree in lists by increasing cost, and the function *bfs2* lifts it to forests. The auxiliary function *combine* takes a list of lists, and reshuffles it by making the first list from all the first elements in each list, the next list from the second elements etc.

$$\mathit{bfs} = \mathit{concat} \cdot \mathit{bfs2}, \tag{53}$$
$$\mathit{bfs2}\ [\,] = [\,], \tag{54}$$
$$\mathit{bfs2}\ \mathit{xf} = \mathit{combine}\ (\mathit{map}\ \mathit{levels}\ \mathit{xf}), \tag{55}$$
$$\mathit{levels}\ (\mathit{Leaf}\ x) = [\,[x]\,], \tag{56}$$
$$\mathit{levels}\ (\mathit{Fork}\ \mathit{xf}) = [\,] : \mathit{bfs2}\ \mathit{xf}, \tag{57}$$
$$\mathit{combine} = \mathit{foldr}\ (\mathit{zipwith}\ (+\!+))\ [\,]. \tag{58}$$

We now show that any query results in the same stream of depth-first sorted answers regardless whether one computes the answers in the stream model, or one computes the answers by the forest model and then applies *dfs* to this forest. Also, one gets the same matrix of breadth-first sorted answers, either by computing queries directly in the matrix model or by applying *bfs2* to the forest resulting from the most general model. Categorically speaking, we show that there exist *morphisms* between the three monads, and that they are exactly the functions *dfs* and *bfs2*.

The polymorphic function *dfs* is a morphism between the $\mathit{Forest}^+$ and $\mathit{Stream}^+$ extended monads if it maps the predicates $\mathit{true}_F$ and $\mathit{false}_F$ to their counterparts in the stream model and if it preserves the behaviour of the *map* functions of the two monads and of the operators & and ||. In other words, *dfs* is a

$Forest^+ \Longrightarrow Stream^+$ morphism if it satisfies:

$$dfs \cdot true_F = true_S, \tag{59}$$
$$dfs \cdot false_F = false_S, \tag{60}$$
$$dfs \cdot fmap\ f = map\ f \cdot dfs, \tag{61}$$
$$dfs \cdot (p \mathbin{\&_F} q) = (dfs \cdot p) \mathbin{\&_S} (dfs \cdot q), \tag{62}$$
$$dfs \cdot (p \parallel_F q) = (dfs \cdot p) \parallel_S (dfs \cdot q). \tag{63}$$

The equations (59) and (60) follow directly from the definitions of *dfs* and the predicates *true* and *false*. Equation (61) can be proved by simultaneous induction for *dfs* and *dfs2*, similarly to the proof of (36). To prove (62), we need a following lemma:

$$dfs \cdot fgraft = concat \cdot (dfs * dfs). \tag{64}$$

Here $dfs * dfs$ denotes a categorical construction called the *horizontal composition* of *dfs* with itself; it is expressed as either:

$$dfs * dfs = dfs \cdot fmap\ dfs, \tag{65}$$
$$dfs * dfs = map\ dfs \cdot dfs. \tag{66}$$

These equations express the fact that it does not matter whether one first does depth-first search on the sub-forests or the main forest. The proof of (64) is done by simultaneous induction on *dfs* and *dfs2*, similarly to the proof of (36).

The proof of (62) is then:

$$\begin{aligned}
& dfs \cdot (p \mathbin{\&_F} q) \\
&= dfs \cdot fgraft \cdot fmap\ q \cdot p && \text{by (27)} \\
&= concat \cdot map\ dfs \cdot dfs \cdot fmap\ q \cdot p && \text{by (64,66)} \\
&= concat \cdot map\ dfs \cdot map\ q \cdot dfs \cdot p && \text{by (61)} \\
&= concat \cdot map\ (dfs \cdot q) \cdot dfs \cdot p && \text{by (3)} \\
&= (dfs \cdot p) \mathbin{\&_S} (dfs \cdot q) && \text{by (2)}
\end{aligned}$$

The proof of (63) is a simple consequence of the distributivity of *concat* and *map* through ++.

Similarly, to prove that also *bfs2* is a monad morphism, we need to show that $true_F$ and $false_F$ are correctly mapped by *bfs2* and that:

$$mmap\ f \cdot bfs2 = bfs2 \cdot fmap\ f, \tag{67}$$
$$bfs2 \cdot (p \mathbin{\&_F} q) = (bfs2 \cdot p) \mathbin{\&_M} (bfs2 \cdot q), \tag{68}$$
$$bfs2 \cdot (p \parallel_F q) = (bfs2 \cdot p) \parallel_M (bfs2 \cdot q). \tag{69}$$

The proofs of (67) and (69) go by structural induction on forests. The proof of (68) requires the following two lemmas which also can be proved by structural

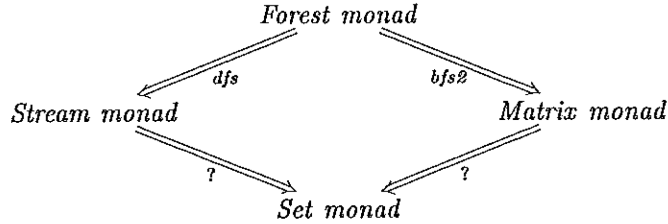
induction on forests:

$$bfs2 \cdot fgraft = shuffle \cdot (bfs2 * bfs2), \tag{70}$$
$$bfs2 * bfs2 = bfs2 \cdot fmap\ bfs2 = mmap\ bfs2 \cdot bfs2, \tag{71}$$

and we get:

$$\begin{aligned}
& bfs2 \cdot (p \,\&_F\, q) \\
&= bfs2 \cdot fgraft \cdot fmap\ q \cdot p && \text{by (27)} \\
&= shuffle \cdot mmap\ bfs2 \cdot bfs2 \cdot fmap\ q \cdot p && \text{by (70,71)} \\
&= shuffle \cdot mmap\ bfs2 \cdot mmap\ q \cdot bfs2 \cdot p && \text{by (67)} \\
&= shuffle \cdot mmap\ (bfs2 \cdot q) \cdot bfs2 \cdot p && \text{by (20)} \\
&= (bfs2 \cdot p) \,\&_M\, (bfs2 \cdot q) && \text{by (19)}
\end{aligned}$$

Further work on this topic is to show that the monad $Forest^+$ is an *initial object* in the 'category of monads that describe logic programming'. This captures the fact that the *dfs* and *bfs2* arrows in the diagram below are unique. Another interesting monad for further algebraic survey of logic programming is the monad where the answers are returned as sets:




The definitions of the operators in the set monad would necessarily have to be less operational then in the other three monads; it would be an algebraic formulation of the least Herbrand models semantics of logic programs. The existence of morphisms between the set monad and the stream and matrix monads would show that the set of answers in both stream and matrix monads is the same, i.e. it would serve as a formal proof that our implementation is correct not only with regards to the operational semantics, but also to the formal declarative semantics of logic programs. These morphisms would correspond to a forgetful functor, i.e. one that "forgets" the information about the ordering and the multiplicity of answers.

## 8 Conclusion and Related Work

Declarative programming, with its mathematical underpinning, was aimed to simplify mathematical reasoning about programs. Both logic and functional programming paradigms facilitate writing of mathematically clear programs and

both paradigms admit variation in execution strategy (lazy or eager, breadth-first or depth-first). So far, only functional programming has allowed easy reasoning about the equality of the clear program with the corresponding efficient program. The proofs take advantage of a suite of algebraic laws for equational inductive reasoning about functions over lists. We propose a corresponding algebraic approach to reasoning about logic programs.

Such algebraic laws can be exploited at all stages of program development. They have had a significant influence on the development of the functional programs so far: designers use them to device correct algorithms, programmers use them to make efficient programs, and language implementors use them to build a language that is suited for optimisation by both programmers and compilers.

Our idea to use algebraic laws to describe and calculate logic programs is motivated by several sources: [1] uses an algebraic approach to functional programming both to derive individual programs and to study programming principles (such as algorithm design) in general, while [4] uses algebraic description to classify and study the different programming paradigms. This paper is an attempt to carry these ideas over to logic programming, emphasising the similarity between its declarative and procedural readings.

Here we have concentrated on the study of the scheduling strategies, but the algebraic approach has many other interesting applications. One interesting topic is a semantical study of functional-logic programming, with more execution details than the least complete Herbrand model in [3]. We hope that this work can stimulate other applications of algebraic reasoning to logic programs.

# References


[1] R. Bird and O. de Moor. *Algebra of Programming.* Prentice Hall, 1997.

[2] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[3] M. Hamana. *Semantics for Interactive Higher-order Functional-logic Programming.* PhD thesis, University of Tsukuba, 1998.

[4] C.A.R. Hoare and H. Jifeng. *Unifying Theories of Programming.* Prentice Hall, 1998.

[5] J.M. Spivey and S. Seres. Embedding Prolog in Haskell. To appear in 1999. http://www.comlab.ox.ac.uk/oucl/users/silvija.seres/Papers/ehp.ps.gz.

[6] P. Wadler. The essence of functional programming. In *19'th Annual Symposium on Principles of Programming Languages,* January 1992.

[7] P. Wadler. Monads for functional programming. In *Advanced Functional Programming,* volume 925 of *LNCS.* Springer Verlag, 1995.