The Algebra of Searching

Mike Spivey and Silvija Seres

December 1999

1 Introduction

What is the logic of Prolog? The usual answer is that it is Horn-clause logic, but this answer does justice only to the 'declarative' reading of programs, and ignores the implicit control information that is contained in a 'procedural' reading. Whilst we may regard two logical specifications as equivalent if they have the same meaning in Horn-clause logic, it is certainly true that logically equivalent programs can have very different operational behaviours. In this paper, we seek to develop a logic for logic programs that takes into account this procedural aspect of their behaviour under different search strategies, emphasizing algebraic properties that are common to all search strategies.

It is perhaps unsurprising that the concept of the search tree of a program gives a universal search strategy, in the sense that any other search strategy can be simulated by building the search tree, then traversing it in an appropriate way. What is perhaps more surprising is that it is possible to present this property of search trees in the language of category theory by forming a category in which the objects are search strategies, the morphisms recover the output of one strategy from the output of another, and search trees are an initial object.

In this paper, we address only the facet of logic programming that has to do with search, and ignore the facet that contains 'logical variables' and unification. We have written elsewhere [2, 4] about this other facet, and how the two may be combined to make a 'shallow embedding', a set of combinators that allow any (pure) Prolog program to be rewritten succinctly as a lazy functional program, in a way that is independent of the search strategy used.

We begin with the familiar idea of implementing depth-first search using lists of successes, then show how the same structure can support combinators that produce first search trees and then breadth-first search. All these search strategies are presented in a compositional way, so that logic programs can be composed semantically, with combinators $\wedge$ and $\vee$ that act on the meanings of programs, not just on pieces of abstract syntax; this is the main advantage of our shallow embedding of Prolog in lazy functional programming over a 'deep embedding', i.e., an interpreter that treats logic programs as syntactic objects.

2 Depth-first search

The use of 'lists of successes' to model backtracking search was suggested by Wadler [5] and has become part of the folklore of functional programming. As a reminder, a relation that takes an argument of type α and may return several results of type β – or none – is represented by a function of type $\alpha \to \mathit{Stream}\ \beta$, where $\mathit{Stream}\ \beta$ is the type of lazy streams over β.[1] Following Richard Bird [1], if $f :: \alpha \to \beta$, we write $f* :: \mathit{List}\ \alpha \to \mathit{List}\ \beta$ for the corresponding mapping function on lists.

Given two relations of the same type, we can form their disjunction by concatenating the streams of answers they return, defining an operator $\vee$ by

$$(p \vee q)\ x = p\ x +\!\!+ q\ x.$$

This operation is associative, and has a unit element false, defined by

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

As in Prolog, the answers from p precede those from q, so that if p returns an infinite stream of answers, or if p diverges, then no answers from q will be seen in the output.

As well as the disjunction of two relations, we may also form their conjunction. More precisely, if $p :: \alpha \to \mathit{Stream}\ \beta$ and $q :: \beta \to \mathit{Stream}\ \gamma$ are two relations, then there is a relation $p \wedge q :: \alpha \to \mathit{Stream}\ \gamma$, defined by

$$p \wedge q = \mathit{concat} \cdot q* \cdot p.$$

The operation $\wedge$ also has a unit element true, defined by $\mathit{true}\ x = [x]$.

Readers who know some category theory will immediately recognise our operation $\wedge$ as the composition operator of the Kleisli category for the monad,

$$\langle \mathit{Stream}, +\!\!+, \mathit{true} \rangle,$$

in which true is the unit. Given this observation, standard arguments show that the $\wedge$ operation is associative and has true as its unit element.

The signs and names we have used for our operations suggest that they should have further algebraic properties. Indeed, we find that false, the unit element for $\vee$, is a zero element on the left for $\wedge$, in the sense that $(\mathit{false} \wedge p) = \mathit{false}$ for any relation p. But we will be disappointed if we push this analogy too far, for false is not a right zero for $\wedge$: if the relation p diverges or returns an infinite stream of answers, then the relation $p \wedge \mathit{false}$ itself diverges. Likewise, though it is easy to show that $\wedge$ distributes over $\vee$ from the right, we find that it does not do so from the left. This behaviour reflects what we find in Prolog, for if $r(X)$ is the predicate that is satisfied when $X = 1$ or $X = 2$, then for

$$r(X) \wedge (Y = 5 \vee Y = 6)$$

[1] In this paper, we shall later want to make a clear distinction between potentially infinite, lazy streams and strictly finite lists, so we distinguish the two types $\mathit{Stream}\ \alpha$ and $\mathit{List}\ \alpha$, though both have the same representation in a lazy functional programming language.

we obtain the four answers $\langle X = 1, Y = 5\rangle$, $\langle X = 1, Y = 6\rangle$, $\langle X = 2, Y = 5\rangle$, $\langle X = 2, Y = 6\rangle$, in that order, while for

$$(r(X) \wedge Y = 5) \vee (r(X) \wedge Y = 6),$$

the second and third answers come in the opposite order. In this finite example, the order and multiplicity of the solutions may be of small importance. With larger examples however, and particularly those involving recursion, it can make the difference between a program that returns an infinite stream of answers and one that returns no answers at all, because the depth-first search has become stuck in an infinite but fruitless branch of the program.

We can deal with the bi-directional character of Prolog programs by making both the input and output of our relation be an *answer*, that is, essentially a substitution. The input answer represents what is known about the values of logical variables before the relation is considered, and the output answer contains also the additional information contributed by the relation itself, if this information is consistent with the input. In our embedding of Prolog in a lazy functional language, a predicate such as *append* has type

$$\mathit{append} :: \mathit{Term} \times \mathit{Term} \times \mathit{Term} \to \mathit{Answer} \to \mathit{Stream}\ \mathit{Answer},$$

so that if x, y and z are terms, then $\mathit{append}\ (x, y, z)$ is a relation mapping input answers (which may give values to some of the variables in x, y and z) to multiple output answers, which may give values to more of those variables. For details see [4] and [2].

3 Search trees

Search strategies for logic programs are often described in terms of the search tree of the program, a tree in which the outcomes of each step of computation are shown as the children of that step. In order to deliver the results of a relation as a tree, we must introduce into our model the idea of a *step of computation*. One possibility would be to incorporate it as part of the $\vee$ operation, so that a step was equated with a choice between alternatives. We prefer to introduce a separate operation, so that it is possible that a single step of computation achieves a choice from among many possibilities, or indeed involves no choice at all, perhaps just the unfolding of a definition. For depth-first search, the idea of computation steps is irrelevant, so in that model, the new operation can be implemented as the identity function.

The search trees we shall use are finitary rose trees – that is, trees in which each node has a *finite list* of children. The type of such trees is defined by

$$\textbf{data}\ \mathit{Tree}\ \alpha = \mathit{Tip}\ \alpha \mid \mathit{Fork}\ (\mathit{Forest}\ \alpha),$$

where the type $\mathit{Forest}\ \alpha$ is defined by

$$\textbf{type}\ \mathit{Forest}\ \alpha = \mathit{List}\ (\mathit{Tree}\ \alpha),$$

with $List\,\alpha$ being the type of finite lists over α. Actually, we shall take the result of a logic program to be a forest, not a single tree, because that will allow us to define an associative disjunction operator easily. We use the notation $\dagger$ for the mapping operator that turns a function $f :: \alpha \to \beta$ into a function $f\dagger :: Forest\,\alpha \to Forest\,\beta$.

The analogue of the function *concat* that we used to define conjunction in the depth-first model is the function

$$graft :: Forest(Forest\,\alpha) \to Forest\,\alpha$$

defined by the following equations:

$$\begin{aligned} graft\,[\,] &= [\,] \\ graft\,[Tip\,xf] &= xf \\ graft\,[Fork\,xff] &= [Fork\,(graft\,xff)] \\ graft\,(xf \mathbin{+\!\!+} yf) &= graft\,xf \mathbin{+\!\!+} graft\,yf \end{aligned}$$

(This kind of definition of a function on lists has a pleasing symmetry that is lost by the usual *nil*/*cons* form of definition. The function is well-defined provided that the right-hand sides describe a monoid in the appropriate way.) This polymorphic function is the multiplication of a monad, and the unit is the function $twig :: \alpha \to Forest\,\alpha$ defined by

$$twig\,x = [Tip\,x].$$

The analogues of the operation $+\!\!+$ on streams (used to define the $\vee$ combinator) and the constant $[\,]$ (used to define *false*) are concatenation of (finite) forests and the empty forest; again these form a monoid. Finally, a step of computation is modelled by applying the function $branch :: Forest\,\alpha \to Forest\,\alpha$ defined by

$$branch\,xf = [Fork\,xf].$$

This function wraps up its argument, a forest, as a single tree, pushing everything down one level.

Following the programme that we used with depth-first search, we can now define logical connectives $\wedge$, $\vee$, *true*, *false* that operate on relations of type $\alpha \to Forest\,\beta$, and a function *step* such that $step\,p$ is a relation that computes the same answers as p, but with a computation cost that is one unit greater than that of p. The definitions are as follows:

$$\begin{aligned} p \wedge q &= graft \cdot q\dagger \cdot p \\ (p \vee q)\,x &= p\,x \mathbin{+\!\!+} q\,x \\ true\,x &= twig\,x \\ false\,x &= [\,] \\ step\,p &= branch \cdot p. \end{aligned}$$

Our family of operations *graft*, $+\!\!+$, *twig*, $[\,]$, *branch* gives the framework for a *compositional* interpretation of these logical connectives. By a compositional

interpretation, we mean one where both $p \wedge q$ and $p \vee q$ have a meaning that is a function of the meanings of p and q.

4 The category of strategies

We now have two models of search that are defined along the same lines, so we may ask what is the relationship between them. Category theory provides the language in which to answer this question; given an appropriate definition of morphisms between different families of operations, we can form a category of search strategies, and we shall find that search trees form an initial object in this category.

Let us use the term *semi-distributive monad* (SDM) for a type constructor and collection of polymorphic combinators that form a search strategy. Specifically, an SDM is a tuple $\langle T, \mathit{join}, \oplus, \mathit{unit}, \mathit{zero}, \mathit{wrap}\rangle$, where T is a functor (type constructor) with associated mapping operator $\triangleright$, and the other elements are natural transformations (polymorphic functions) with the following types:

$$\begin{array}{rcl} \mathit{join} & :: & T\,(T\,\alpha) \to T\,\alpha \\ (\oplus) & :: & T\,\alpha \times T\,\alpha \to T\,\alpha \\ \mathit{unit} & :: & \alpha \to T\,\alpha \\ \mathit{zero} & :: & T\,\alpha \\ \mathit{wrap} & :: & T\,\alpha \to T\,\alpha. \end{array}$$

Any SDM must satisfy certain algebraic laws:

- $\langle T, \mathit{join}, \mathit{unit}\rangle$ is a monad,
- $\langle T\,\alpha, \oplus, \mathit{zero}\rangle$ is a monoid for each type α, and
- the following 'one-sided distributive laws' hold:

$$\begin{array}{rcll} \mathit{join}\,(\mathit{xt} \oplus \mathit{yt}) & = & \mathit{join}\,\mathit{xt} \oplus \mathit{join}\,\mathit{yt} & (1) \\ \mathit{join}\,\mathit{zero} & = & \mathit{zero} & (2) \\ \mathit{join} \cdot \mathit{wrap} & = & \mathit{wrap} \cdot \mathit{join}. & (3) \end{array}$$

These last three laws translate into algebraic properties of our higher-level connectives:

$$\begin{array}{rcl} (p \vee q) \wedge r & = & (p \wedge r) \vee (q \wedge r) \\ \mathit{false} \wedge p & = & \mathit{false} \\ \mathit{step}\,(p \wedge q) & = & (\mathit{step}\,p) \wedge q \end{array}$$

The last of these seems to express the fact that the first step in solving $p \wedge q$ is devoted to the solution of p.

We can also define the notion of a *morphism of search strategies*. A morphism

$$h :: \langle T, \mathit{join}, \oplus, \mathit{unit}, \mathit{zero}, \mathit{wrap}\rangle \to \langle T', \mathit{join}', \oplus', \mathit{unit}', \mathit{zero}', \mathit{wrap}'\rangle$$

is a natural transformation $h :: T\ \alpha \to T'\ \alpha$ that respects the other components of the SDM. Specifically,

$$\begin{aligned} h \cdot \mathit{join} &= \mathit{join}' \cdot (h \star h) \\ h\ (x \oplus y) &= h\ x \oplus' h\ y \\ h \cdot \mathit{unit} &= \mathit{unit}' \\ h\ \mathit{zero} &= \mathit{zero}' \\ h \cdot \mathit{wrap} &= \mathit{wrap}' \cdot h. \end{aligned}$$

(We use the notation $h \star h = h \cdot h\triangleright = h\triangleright' \cdot h$ for the *horizontal composition* of h with itself.)

These definitions make semi-distributive monads into a category, with both our examples of search strategies as objects. It turns out that the tree-based strategy is an initial object in this category. Given any SDM

$$\langle T, \mathit{join}, \oplus, \mathit{unit}, \mathit{zero}, \mathit{wrap}\rangle,$$

we can define a morphism

$$h :: \langle \mathit{Forest}, \mathit{graft}, +\!\!+, \mathit{twig}, [\,], \mathit{branch}\rangle \to \langle T, \mathit{join}, \oplus, \mathit{unit}, \mathit{zero}, \mathit{wrap}\rangle$$

by the recursive equations,

$$\begin{aligned} h\ [\,] &= \mathit{zero} \\ h\ [\mathit{Tip}\ x] &= \mathit{unit}\ x \\ h\ [\mathit{Fork}\ xf] &= \mathit{wrap}\ (h\ xf) \\ h\ (xf +\!\!+ yf) &= h\ xf \oplus h\ yf. \end{aligned}$$

We need to make several observations about this definition. The first is that the function h is well-defined by the equations, because $\oplus$ is associative with unit *unit*. As in the definition of *graft*, the right-hand sides describe a monoid, since T is an SDM, and this makes it immaterial what decomposition we take of the argument of h. For reasons of symmetry, we prefer this style of definition to the usual style in terms of *nil* and *cons*.

Second, the four equations given are just a translation into concrete terms of four of the five conditions for h to be a morphism of SDM's; for example, the condition $h \cdot \mathit{wrap} = \mathit{wrap}' \cdot h$ becomes $h\ [\mathit{Fork}\ xf] = \mathit{wrap}\ xf$ when we make the appropriate substitutions $\mathit{wrap}\ xf = [\mathit{Fork}\ xf]$ and $\mathit{wrap}' = \mathit{wrap}$. This means that if there is any morphism between these SDM's, then it must be equal to h, the function uniquely defined by these equations.

Third, h is in fact a morphism, because it satisfies the fifth condition,

$$h \cdot \mathit{graft} = \mathit{join} \cdot (h \star h).$$

This may be proved by induction.

5 Breadth-first search

If depth-first search and search based on trees form two objects in the category of semi-distributive monads, can breadth-first search be put in the same picture, so as to get a compositional formulation of breadth-first search also? The answer is yes, but there are some difficulties to be overcome.

The basic idea is to enumerate the solutions in a search tree level-by-level, so as to get a potentially infinite stream, each element of which is a finite collection of solutions. It's tempting to say that each level is a finite *list*, but as we shall see, it is impossible to model breadth-first search in a compositional way if we insist on knowing the order of the solutions in each level; so instead, we shall say that each level will be a finite bag of solutions. We'll use the term *matrix* for a lazy stream of bags:

$$\textbf{type}\ \ \mathit{Matrix}\ \alpha = \mathit{Stream}(\mathit{Bag}\ \alpha).$$

We'll use $\diamond$ for the mapping operator on bags, $\cup$ for the binary union operator on bags, and $\mathit{union} :: \mathit{Bag}(\mathit{Bag}\ \alpha) \to \mathit{Bag}\ \alpha$ for the function that gathers together all elements of a bag of bags.

We will now model relations as functions in $\alpha \to \mathit{Matrix}\ \beta$. It's easy to see that total failure corresponds to the matrix that has the empty bag $[\![\,]\!]$ in every level, and a computation that immediately returns the single answer x should have $[\![x]\!]$ as its first level and $[\![\,]\!]$ ever after:

$$\begin{aligned} \mathit{fail} &= \mathit{repeat}\ [\![\,]\!] \\ \mathit{succeed}\ x &= [\![x]\!] : \mathit{repeat}\ [\![\,]\!] \end{aligned}$$

These will become the *zero* and *unit* elements of our SDM.[2]

Given two matrices of the same type, we can combine them level-by-level with the binary union operator:

$$xm \uplus ym = \mathit{zipWith}\ (\cup)\ xm\ ym.$$

This operation is associative, has *fail* as its unit, and provides an appropriate disjunction operator for our SDM. Again, we can take a matrix and move each level down by one, adding an empty level at the start:

$$\mathit{delay}\ xm = [\,] : xm.$$

This gives an appropriate *wrap* operation, delaying all answers by one unit of time.

The operations we have defined so far are all that we need to define a function *bfs* from forests to matrices that shows how to search a forest breadth-first:

$$\begin{aligned} \mathit{bfs}\ [\,] &= \mathit{repeat}\ [\![\,]\!] \\ \mathit{bfs}\ [\mathit{Tip}\ x] &= [\![x]\!] : \mathit{repeat}\ [\![\,]\!] \\ \mathit{bfs}\ [\mathit{Fork}\ xf] &= [\![\,]\!] : \mathit{bfs}\ xf \\ \mathit{bfs}\ (xf +\!\!+ yf) &= \mathit{zipWith}\ (\cup)\ (\mathit{bfs}\ xf)\ (\mathit{bfs}\ yf) \end{aligned}$$

[2] By small changes to these definitions and those that follow, we could make a version of our model in which searches terminate if the underlying search tree is finite. For simplicity, we refrain from doing this.

The final ingredient is a vital one in a compositional treatment of breadth-first search; so far, we can explain breadth-first search only by saying "first form search tree of the program, then search it using the function *bfs*." We want instead to define a way of composing relations directly when they are modelled by functions that return a matrix. It is here that the decision to use a bag for each level of the matrix becomes important.

The problem is as follows: we have two relations $p :: \alpha \to \mathit{Matrix}\ \beta$ and $q :: \beta \to \mathit{Matrix}\ \gamma$, and we wish to form their conjunction, a relation of type $\alpha \to \mathit{Matrix}\ \gamma$. Following the pattern established earlier, we wish to define

$$p \wedge q = \mathit{join} \cdot q\diamond* \cdot p,$$

for some suitable function $\mathit{join} : \mathit{Matrix}(\mathit{Matrix}\ \alpha) \to \mathit{Matrix}\ \alpha$.

Since $\mathit{Matrix}\ \alpha = \mathit{Stream}(\mathit{Bag}\ \alpha)$, and *Stream* and *Bag* are both monads, it is very tempting to think that we ought to use a distributive law of one monad over the other, like this:

$$SBSB\alpha \xrightarrow{trans*} SSBB\alpha \xrightarrow{union**} SSB\alpha \xrightarrow{concat} SB\alpha,$$

where $\mathit{trans} :: \mathit{Bag}(\mathit{Stream}\ \alpha) \to \mathit{Stream}(\mathit{Bag}\ \alpha)$ is the obvious transposition function. But that would not be the right thing at all! Once we have rearranged things with *trans*, we have a stream of streams; successive elements of the outer stream correspond to increasing computation cost devoted to solving p, and successive elements in each inner stream correspond to increasing cost of solving q. We do not want to arrange these in lexicographic order of increasing cost for p and within that, increasing cost for q. Instead, we would like to arrange the solutions in order of increasing total cost. We can do this using a function

$$\mathit{diag} :: \mathit{Stream}(\mathit{Stream}\ \alpha) \to \mathit{Stream}(\mathit{Bag}\ \alpha)$$

that arises in Cantor's famous proof that the set of pairs of natural numbers $\mathcal{N} \times \mathcal{N}$ (under the guise of the rationals $\mathcal{Q}$) is countable. This function takes a two-dimensional infinite array and slices it into diagonals, making each diagonal into a bag, thus:

$$\begin{aligned}&\mathit{diag}\ [[x_{00}, x_{01}, x_{02}, \ldots], [x_{10}, x_{11}, x_{12}, \ldots], [x_{20}, \ldots], \ldots] \\ &\quad = [[\![x_{00}]\!], [\![x_{01}, x_{10}]\!], [\![x_{02}, x_{11}, x_{20}]\!], \ldots]\end{aligned}$$

Using this function, we define *join* as follows:

$$\mathit{join} = (\mathit{union} \cdot \mathit{union})* \cdot \mathit{diag} \cdot \mathit{trans}*.$$

In outline:

$$SBSB\alpha \xrightarrow{trans*} SSBB\alpha \xrightarrow{diag} SBBB\alpha \xrightarrow{(union\cdot union)*} SB\alpha.$$

The proof that, together with the other functions we have defined, *join* satisfies the laws of an SDM is mostly straightforward, apart from the proof that *join* is associative, that is,

$$\mathit{join} \cdot \mathit{join} = \mathit{join} \cdot \mathit{join}\diamond*.$$

This proof relies on a small theory of *trans*, *diag* and their interaction that we have no space to present here, but have written about elsewhere [3].

This question of associativity bears also on the question why we insist on using bags instead of lists. It is easiest to explain this in terms of the matrix returned by a conjunction of three predicates $p \wedge q \wedge r$. The n'th level in this matrix enumerates all the solutions with total cost n, and can be imagined as the equilateral triangular region $\{ (x, y, z) \mid x + y + z = n \}$ in positive 3-space, where the three coordinate axes correspond to the costs of solving p, q and r. In a model based on lists, if the conjunction were bracketed as $p \wedge (q \wedge r)$, the solutions would be enumerated in order of decreasing x, and within that, decreasing y. Bracketed the other way, the formula would call for the answers in order of increasing z, and within that, increasing y – and these two orders are not the same. By moving from lists to bags, we avoid this problem.

Returning to the link with Cantor's proof, we are asking the question, "How many essentially different proofs are there that $\mathcal{N} \times \mathcal{N} \times \mathcal{N}$ is countable?" Our answer is that there is just one, once we have removed the parts of Cantor's construction that are purely accidental.

6 Summary and conclusions

By identifying the SDM of search trees as an initial object in this category, we have shown that all interpretations of Prolog programs that are (in our sense) compositional consist of traversing the search tree of the program in one way or another. To look at it another way, we have established a set of algebraic laws that are common to all compositional interpretations; these are exactly the laws that are satisfied by the SDM of search trees. Thus we have identified exactly the laws that are valid regardless of the search strategy, and can be used for transforming logic programs without commitment to a particular execution mechanism.

This is (we hope) one small step towards a treatment of logic programming that reconciles the procedural and declarative readings of programs by providing a single algebraic framework that subsumes both of them.

References

[1] R. S. Bird, *Introduction to the theory of lists*, in *Logics of Programming and Calculi of Discrete Design* (M. Broy, ed.), Springer-Verlag, 1987.

[2] S. Seres, J. M. Spivey and C. A. R. Hoare, 'Algebra of logic programming', in *Proceedings of the 1999 International Conference on Logic Programming* (D. De Schreye, ed.), MIT Press, 1999.

[3] J. M. Spivey, 'The monad of breadth-first search', submitted to *Journal of Functional Programming*.

[4] J. M. Spivey and S. Seres, 'Embedding Prolog in Haskell', in *Proceedings of Haskell'99* (E. Meier, ed.), Technical Report UU–CS–1999–28, Department of Computer Science, University of Utrecht.

[5] P. L. Wadler, 'How to replace failure by a list of successes', in *Functional Programming Languages and Computer Architecture*, (J.-P. Jouannaud, ed.), LNCS 201, Springer-Verlag, 1985.