Embedding PROLOG in HASKELL

Michael Spivey   Silvija Seres

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

Abstract

We propose an embedding of logic programming into lazy functional programming in which each predicate in a Prolog program becomes a Haskell function, in such a way that both the declarative and the procedural reading of the Prolog predicate are preserved.

The embedding computes by means of operations on lazy lists. The state of each step in computation is passed on as a stream of answer substitutions, and all the logic operators of Prolog are implemented by explicit Haskell operators on these streams. The search strategy can be changed by altering the basic types of the embedding and the implementation of these operators. This model results in a perspicuous semantics for logic programs, and serves as a good example of modularisation in functional programming.

1 Introduction

Many researchers have sought a way of combining the virtues of functional and logic programming within a single programming system or language. In this paper, we follow a suggestion of Tony Hoare's [7] and consider a simple and direct embedding of the constructs of Prolog in a lazy functional language. In some ways, this embedding gives a programming language similar to LOGLISP [18, 17], since both approaches attempt to give an embedding *within* the host language. The base language is not extended; the run-time system for both embeddings consists of a set of functions designed to support unification, resolution and search. The difference in our approach is that the power of lazy functional programming offers a simpler and a more natural way to implement the primitive functions of the embedding and to handle infinite sets of answers. Indeed, the higher-order operators on streams and the use of types result in a set of embedding functions that describe the operational semantics of pure logic programs in a strikingly concise way.

There has been a considerable amount of research on combining features of functional and logic programming in a single language. The integrated languages that result embody both rewriting and resolution and thereby result in a functional language with the capability to solve arbitrary constraints for the values of variables. The mainstream approaches use *narrowing* as the operational semantics for the amalgamated language [15]. The list of such *functional logic languages* that have been proposed in an attempt to incorporate the expressive power of both paradigms is long and impressive [5], but we do not aim to compete with them. These projects aspire to build an efficient language that can offer programmers the most useful features of both worlds; to achieve this they have to adopt somewhat complicated semantics. We wish to show how that the pure part of logic languages can be embedded into lazy functional ones with a strikingly simple implementation, and use this as an example of the expressive power and elegance of the lazy functional paradigm.

The primitive functions of our embedding use only stream operators like *map* and *concat* linked by functional composition; this gives us a whole suite of algebraic properties for the primitive functions. A combination of these can serve as a partial algebraic specification of the embedded logic program. The algebraic laws of the primitives can also be used to transform the embedded logic programs by equational reasoning.

In this paper we use Prolog and Haskell as our languages of choice, but the principles presented are general. Prolog is chosen because it is the dominant logic language, although we only implement the pure declarative features of it, i.e., we ignore the impure but practically much used features like cut, `assert` and `retract`. Haskell is chosen because it is a lazy functional language with types and lambda-abstractions, but any other language with these properties could be used.

In the remainder of the paper we proceed to describe the syntax of the embedding and the implementation of the primitives in sections 2 and 3. In section 4 we list some of the algebraic properties of the operators and in section 5 we study the necessary changes to the system to accommodate different search strategies. We conclude the paper with section 6 where we discuss related work and propose some further work in this setting.

2 Syntax

Prolog offers the facility of defining a predicate in many clauses and it allows the applicability of each clause to be tested by pattern matching on the formal parameter list. In our implementation of Prolog, we have to withdraw these notational licences, and require the full logical meaning of the predicate to be defined in a single equation, with the unifications made explicit on the right hand side, together with the implicit existential quantification over the fresh variables.

In the proposed embedding of Prolog into a functional language, we aim to give rules that allow any pure Prolog predicate to be translated into a Haskell function 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:

$$(\&), (\|) : \mathit{Predicate} \longrightarrow \mathit{Predicate} \longrightarrow \mathit{Predicate},$$
$$(\doteq) : \mathit{Term} \longrightarrow \mathit{Term} \longrightarrow \mathit{Predicate},$$
$$\mathit{exists} : (\mathit{Term} \longrightarrow \mathit{Predicate}) \longrightarrow \mathit{Predicate}.$$

The intention is that the operators & and $\|$ denote conjunction and disjunction of predicates, $\doteq$ forms a predicate expressing the equality of two terms, and the operation *exists* expresses existential quantification. We shall abbreviate the expression $\mathit{exists}\ (\lambda x \rightarrow p\ x)$ by the form $\exists x \rightarrow p\ x$ in this paper, although the longer form shows how the expression can be written in any lazy functional language that has λ-expressions. We shall also write $\exists x, y \rightarrow p(x, y)$ for $\exists x \rightarrow (\exists y \rightarrow p(x, y))$.

These four operations suffice to translate any pure Prolog program, provided we are prepared to exchange pattern matching for explicit equations, to bind local variables with explicit quantifiers, and to gather all the clauses defining a predicate into a single equation. These steps can be carried out systematically, and could easily be automated. 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. These equations are computed by unification in Prolog.

```
append(Ps, Qs, Rs) :-
    Ps = [], Qs = Rs.
append(Ps, Qs, Rs) :-
    Ps = [X|Xs], Rs = [X|Ys], append(Xs, Qs, Ys).
```

The head of each clause now contains only a list of distinct variables, and by renaming if necessary we can ensure that the list of variables is the same in each clause. We complete the translation by joining the clause bodies with the $\|$ operation, the literals in a clause with the & operation, and existentially quantifying any variables that appear in the body but not in the head of a clause:

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

Here *nil* is used for the value of type *Term* that represents the empty list, and *cons* is written for the function on terms that corresponds to the Prolog list constructor `[|]`. We assume the following order of precedence on the operators, from highest to lowest: $\doteq, \&, \|, \exists$.

The function *append* defined by this recursive equation has the following type:

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

The Haskell function *append* is constructed by making the *declarative* reading of the Prolog predicate explicit. However, the relationship between the Haskell function and the Prolog predicate extends beyond their declarative semantics. The next section shows that the *procedural* reading of the Prolog predicate is also preserved through the implementation of the functions & and $\|$. The embedding essentially allows the *mapping of the computation of the Prolog program into lazy lists* by embedding the structure of a SLD-tree of a Prolog program into a Haskell stream.

3 Implementation

The translation described above depends on the four operations &, $\|$, $\doteq$ and *exists*. We now give definitions to the type of predicates and to these four operations that correspond to the depth-first search of Prolog. Later, we shall be able to give alternative definitions that correspond to breadth-first search, or other search strategies based on the search tree of the program.

The key idea is that each predicate is a function that takes an 'answer', representing the state of knowledge about the values of variables at the time the predicate is solved, and produces a lazy stream of answers, each corresponding to a solution of the predicate that is consistent with the input. This approach is similar to that taken by Wadler [22]. An unsatisfiable query results in an empty stream, and a query with infinitely many answers results in an infinite stream.[1]

$$\textbf{type } Predicate = Answer \longrightarrow Stream\ Answer.$$

An answer is (in principle) just a substitution, but we augment the substitution with a counter that tracks the number of variables that have been used so far, so that a fresh variable can be generated at any stage by incrementing the counter:

$$\textbf{type } Answer = (Subst, Int).$$

We can now give definitions for the four operators. The operators & and $\|$ act as predicate combinators; they slightly resemble the notion of *tacticals* [14], but

[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.

in our case they combine the computed streams of answers, rather that partially proved statements.

The $\|$ operator simply concatenates the streams of answers returned by its two operands:

$$(\|) :: \mathit{Predicate} \longrightarrow \mathit{Predicate} \longrightarrow \mathit{Predicate}$$
$$(p \,\|\, q)\ x = p\ x \mathbin{+\!\!+} q\ x.$$

This definition implies that the answers are returned in a left-to-right order as in Prolog. If the left-hand argument of $\|$ is unsuccessful and returns an empty answer stream, it corresponds to an unsuccessful branch of the search tree in Prolog and backtracking is simulated by evaluating the right-hand argument.

For the & operator, we start with applying the first argument to the incoming answer; this produces a stream of answers, to each of which we apply the second argument of &. Finally, we concatenate the resulting stream of streams into a single stream:

$$(\&) :: \mathit{Predicate} \longrightarrow \mathit{Predicate} \longrightarrow \mathit{Predicate}$$
$$p \mathbin{\&} q = \mathit{concat} \cdot \mathit{map}\ q \cdot p.$$

Because of Haskell's lazy evaluation, the function p returns answers only when they are needed by the function q. This corresponds nicely with the backtracking behaviour of Prolog, where the predicate $p \mathbin{\&} q$ is implemented by enumerating the answers of p one at a time and filtering them with the predicate q. Infinite list of answers in Prolog are again modelled gracefully with infinite streams.

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

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

The pattern matching of Prolog is implemented by the operator $\doteq$. It is defined in terms of a function *unify* which performs unification of two terms relative to a given input substitution. The type of *unify* is thus:

$$\mathit{unify} :: \mathit{Subst} \longrightarrow (\mathit{Term}, \mathit{Term}) \longrightarrow \mathit{List}\ \mathit{Subst}.$$

More precisely, the result of $\mathit{unify}\ s\ (t, u)$ is either $[s \triangleright r]$, where r is a most general unifier of $t[s]$ and $u[s]$, or $[\,]$ if these two terms have no unifier.[2]Thus if $\mathit{unify}\ s\ (t, u) = [s']$, then s' is the most general substitution such that $s \sqsubseteq s'$ and $t[s'] = u[s']$.

[2]We use $s \triangleright r$ to denote composition of substitutions s and r, and $t[s]$ to denote the instance of term t under substitution s. We use $s \sqsubseteq s'$ to denote the preorder on substitutions that holds iff $s' = s \triangleright r$ for some substitution r.

The $\doteq$ operator is just a wrapper around *unify* that passes on the counter for fresh variables:

$$(\doteq) :: (\mathit{Term}, \mathit{Term}) \longrightarrow \mathit{Predicate}$$
$$(t \doteq u)\ (s, n) = [(s', n) \mid s' \leftarrow \mathit{unify}\ s\ (t, u)]$$

Finally, the operator *exists* is responsible for allocating fresh names for all the local (or existentially quantified) variables in the predicates. This is necessary in order to guarantee that the computed answer is the most general result. The function *exists* takes a λ-expression as its first argument and the usual input as its second argument. The bound variable in the λ-expression becomes one of the quantified variables in the predicate. So we have:

$$\mathit{exists} :: (\mathit{Term} \longrightarrow \mathit{Predicate}) \longrightarrow \mathit{Predicate}$$
$$\mathit{exists}\ p\ (s, n) = p\ (\mathit{makevar}\ n)\ (s, n + \mathit{1}),$$

where *makevar* n is a term representing the n'th generated variable. The slightly convoluted flow of information here may be clarified by a small example. The argument p of *exists* will be a function that expects a variable, such as $(\lambda X \rightarrow \mathit{append}(t, X, u))$. We apply this function to a newly-invented variable $v = \mathit{makevar}\ n$ to obtain the predicate $\mathit{append}(t, v, u)$, and finally apply this predicate to the answer $(s, n + \mathit{1})$, in which all variables up to the n'th are marked as having been used.

The function *solve* evaluates the main query. It simply applies its argument, the predicate of the query, to an answer with an empty substitution and a zero variable counter, and converts the resulting stream of answers to a stream of strings.

$$\mathit{solve} :: \mathit{Predicate} \longrightarrow \mathit{Stream\ String}$$
$$\mathit{solve}\ p = \mathit{map}\ \mathit{print}\ (p\ ([\,], \mathit{0})).$$

We do not provide proofs of the soundness and completeness (relative to Prolog) of the embedding, since they follow directly from the way the embedding is constructed. The encoding we have described is about the simplest possible mechanised formal definition of Prolog.

4 Algebraic Laws

The operators & and $\|$ enjoy many algebraic properties as a consequence of their simple definitions in terms of streams.

The & operator is associative with unit element *true*. This is a consequence of the fact that *map*, *concat* and *true* form a structure that Category Theory calls a *monad*, and the composition operator & is obtained from this by a standard construction called *Kleisli composition*.

All the algebraic properties we quote can be proved with simple equational reasoning, using only the standard laws (see [2]) for *concat*, *map* and functional composition. We omit most of the elementary proofs here, but will revisit these properties later when we examine other implementations of our fundamental operations. As an example, given:

$$map\ f \cdot concat = concat \cdot map\ (map\ f), \tag{1}$$

$$concat \cdot concat = concat \cdot map\ concat, \tag{2}$$

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

we can prove the associativity of & by the following rewriting:

$$\begin{aligned}
&(p \mathbin{\&} q) \mathbin{\&} r \\
&= concat \cdot map\ r \cdot concat \cdot map\ q \cdot p && \text{by defn. of \&} \\
&= concat \cdot concat \cdot map\ (map\ r) \cdot map\ q \cdot p && \text{by (1)} \\
&= concat \cdot map\ concat \cdot map\ (map\ r) \cdot map\ q \cdot p && \text{by (2)} \\
&= concat \cdot map\ (concat \cdot map\ r \cdot q) \cdot p && \text{by (3)} \\
&= p \mathbin{\&} (q \mathbin{\&} r). && \text{by defn. of \&}
\end{aligned}$$

The predicate *false* is a left zero for &, but this operator is strict in its left argument, so *false* is not a right zero. This corresponds to the feature of Prolog that *false* & q has that same behaviour as *false*, but p & *false* may fail infinitely if p does. Owing to the properties of *concat* and [ ], the $\|$ operator is associative and has *false* as a left and right identity.

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, the $\|$ operator is not idempotent, because *true* $\|$ *true* produces its input answer twice as an output, but *true* itself produces only one answer. The & operator also fails to be idempotent, because the predicate

$$(true \parallel true) \mathbin{\&} (true \parallel true)$$

produces the same answer four times rather than just twice.

We might also expect

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

that is, for & to distribute over $\|$, but this is not the case. For a counterexample, take for p the predicate $X \doteq a \parallel X \doteq b$, for q the predicate $Y \doteq c$, and for r the predicate $Y \doteq d$. Then the left-hand side of the above equation produces the four answers $[X{=}a,\ Y{=}c]$; $[X{=}a,\ Y{=}d]$; $[X{=}b,\ Y{=}c]$; $[X{=}b,\ Y{=}d]$ in that order, but the right-hand side produces the same answers in the order $[X{=}a,\ Y{=}c]$; $[X{=}b,\ Y{=}c]$; $[X{=}a,\ Y{=}d]$; $[X{=}b,\ Y{=}d]$.

However, the other distributive law,

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

does hold, and it is vitally important to the unfolding steps of program transformation. The simple proof depends on the fact that both *map r* and *concat* are homomorphisms with respect to $+\!\!+$:

$$\begin{aligned}
&((p \parallel q) \mathbin{\&} r)\ x \\
&= \mathit{concat}\ (\mathit{map}\ r\ (p\ x +\!\!+ q\ x)) && \text{by defn. of } \parallel, \& \\
&= \mathit{concat}\ (\mathit{map}\ r\ (p\ x) +\!\!+ \mathit{map}\ r\ (q\ x)) && \mathit{map} \\
&= \mathit{concat}\ (\mathit{map}\ r\ (p\ x)) +\!\!+ \mathit{concat}\ (\mathit{map}\ r\ (q\ x)) && \mathit{concat} \\
&= ((p \mathbin{\&} r) \parallel (q \mathbin{\&} r))\ x. && \text{by defn. of } \&
\end{aligned}$$

The declarative reading of logic programs suggests that also the following properties of $\doteq$ and $\exists$ ought to hold, where $p\ x$ and $q\ x$ are predicates and u is a term not containing x:

$$\begin{aligned}
&(\exists x \to p\ x \parallel q\ x) = (\exists x \to p\ x) \parallel (\exists x \to q\ x), \\
&(\exists x \to x \doteq u \mathbin{\&} p\ x) = p\ u, \\
&(\exists x \to (\exists y \to p\ (x, y))) = (\exists y \to (\exists x \to p(x, y))).
\end{aligned}$$

These properties are important in program transformations that manipulate quantifiers and equations, since they allow local variables to be introduced and eliminated, and allow equals to be substituted for equals in arbitrary formulas.

However, these properties of $\doteq$ and $\exists$ depend on properties of predicates p and q that are not shared by all functions of this type, but are shared by all predicates that are defined purely in terms of our operators. In future work, we plan to formulate precisely the 'healthiness' properties of definable predicates on which these transformation laws depend, such as monotonicity and substitutivity.

It might be seen as a weakness of our approach based on a 'shallow' embedding of Prolog in Haskell that these properties must be expressed in terms of the weak notion of a predicate definable in terms of our operators, when a 'deep' embedding (i.e., an interpreter for Prolog written in Haskell) would allow us to formulate and prove them as an inductive property of program texts. We believe that this is a price well worth paying for the simplicity and directness of our marriage between functional and logic programming.

5 Different Search Strategies

Our implementation of $\parallel$, together with the laziness of Haskell, causes the search for answers to behave like depth-first search in Prolog: when computing $p\ x +\!\!+ q\ x$ all the answers corresponding to the $p\ x$ part of the search tree are returned before the other part is explored. A fair *search* strategy would share the computation effort more evenly between the two parts. Similarly, our implementation of & results in a left-to-right selection of the literals of a clause. A fair *selection* rule would allow one to chose the literals in a different order.

One possible solution (inspired by [11]) is to *interleave* the streams of answers, taking one answer from each stream in turn. A function *twiddle* that interleaves two lists can be defined as:

$$
\begin{aligned}
&\mathit{twiddle} \;::\; [a] \longrightarrow [a] \longrightarrow [a]\\
&\mathit{twiddle}\ [\,]\ \mathit{ys} = \mathit{ys}\\
&\mathit{twiddle}\ (x : \mathit{xs})\ \mathit{ys} = x : (\mathit{twiddle}\ \ \mathit{ys}\ \ \mathit{xs}).
\end{aligned}
$$

The operators $\|$ and & can be redefined by replacing $+\!\!+$ with *twiddle* and recalling that $\mathit{concat} = \mathit{foldr}\ (+\!\!+)\ [\,]$:

$$
\begin{aligned}
&(p \parallel q)\ x = \mathit{twiddle}\ (p\ \ x)\ (q\ \ x)\\
&(p\ \&\ q)\ x = \mathit{foldr}\ (\mathit{twiddle})\ [\,]\ (\mathit{map}\ \ q\ \ (p\ \ x)).
\end{aligned}
$$

This implementation of & is fairer, producing in a finite time solutions of q that are based on later solutions returned by p, even if the first such solution produces an infinite stream of answers from q. The original implementation of & produces all solutions of q that are based on the first solution produced by p before producing any that are based on the second solution from p.

Note that this implementation of operators does *not* give breadth-first search of the search tree; it deals with infinite success but not with infinite failure. Even in the interleaved implementation, the first element of the answer list is produced before all the others; if this takes an infinite number of steps the other branch or literals will not be reached.

To implement breadth-first search in the embedding, the *Predicate* data-type needs to be changed. It is no longer adequate to return a single, flat stream of answers; this model is not refined enough to take into account the number of *computation steps* needed to produce a single answer. The key idea is to let *Predicate* return a stream of lists of answers, where each list represents the answers reached at the same depth, or level, of the search tree. These lists of answers with the same cost are always finite since there is only a finite number of nodes at each level of the search tree. The new type of *Predicate* is thus:

$$\mathit{Predicate} :: \mathit{Answer} \longrightarrow \mathit{Stream}\ (\mathit{List}\ \ \mathit{Answer}).$$

Intuitively, each successive list of answers in the stream contains the answers with the same computational "cost". The cost of an answer increases with every resolution step in its computation. This can be captured by adding a new function *step* in the definition of predicates. For example, *append* should be coded as:

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

In the depth-first model, *step* is the identity function on predicates, but in the breadth-first model it is defined as follows:

$$
\begin{aligned}
&step :: Predicate \longrightarrow Predicate \\
&step\ p\ x = [\,] : (p\ x).
\end{aligned}
$$

Thus, in the stream returned by *step* p, there are no answers of cost 0, and for each n, the answers of *step* p with cost $n + 1$ are the same as the answers of p that have cost n.

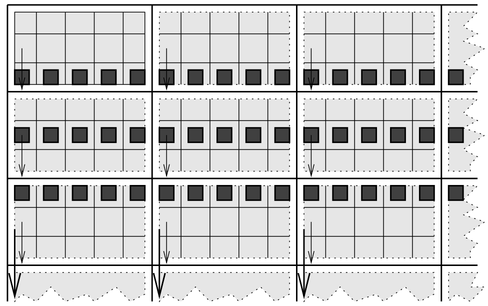
The implementations of the *Predicate* combinators $\parallel$ and & need to be changed so that they no longer operate on lists but on streams of lists. They must preserve the cost information that is embedded in the input lists. Since the cost corresponds to the level of the answer in the search tree, only resolution steps are charged for, while the applications of $\parallel$, & and *equals* are cost-free. The $\parallel$ operator simply zips the two streams into a single one, by concatenating all the sublists of answers with the same cost. If the two streams are of different lengths, the zipping must not stop when it reaches the end of the shorter stream. We give the name *mergewith* to a specialized version of *zipwith* that has this property:

$$(p \parallel q)\ x = mergewith\ (+\!\!+)\ (p\ x)\ (q\ x)$$

The implementation of & is harder. The cost of each of the answers to $(p \mathbin{\&} q)$ is a sum of the costs of the computation of p and the computation of q. The idea is first to compute all the answers, and then to flatten the resulting stream of lists of streams of lists of answers to a stream of lists of answers according to the cost. This flattening is done by the *shuffle* function which is explained below. The &-operator is thus:

$$p \mathbin{\&} q = shuffle \cdot map\ (map\ q) \cdot p$$

We write S for streams and L for finite lists for sake of brevity. The result of $map\ (map\ q) \cdot p$ is of type *SLSL*. It can be visualised as a matrix of matrices, where each element of the outer matrix corresponds to a single answer of p. Each such answer is used as an input to q and consequently gives rise to a new stream of lists of answers, which are represented by the elements of the inner matrices. The rows of both the main matrix and the sub-matrices are finite, while the columns of both can be infinite. For example, the answers of $map\ (map\ q) \cdot p$ with cost 2 are marked in the drawing below:



The function *shuffle* collects all the answers marked in the drawing into a single list, the third in the resulting stream of lists of answers. It is given an *SLSL* of answers, and it returns an *SL*. Two auxiliary functions are required to do this: *diag* and *transpose*. A stream of streams is converted to a stream of lists by *diag*, and a list of streams can be converted to a stream of lists by *transpose*.

$$\mathit{diag} :: \mathit{Stream}\ (\mathit{Stream}\ a) \longrightarrow \mathit{Stream}\ (\mathit{List}\ a)$$
$$\mathit{diag}\ \mathit{xss} = [[(\mathit{xss}\ !\ i)\ !\ (n - i) \mid i \leftarrow [0..n]] \mid n \leftarrow [0..]]$$

$$\mathit{transpose} :: \mathit{List}\ (\mathit{Stream}\ a) \longrightarrow \mathit{Stream}\ (\mathit{List}\ a)$$
$$\mathit{transpose}\ \mathit{xss} = \mathit{map}\ \mathit{hd}\ \mathit{xss} : \mathit{transpose}\ (\mathit{map}\ \mathit{tl}\ \mathit{xss})$$

Given *diag* and *transpose*, the function *shuffle* can be implemented as follows. The input to *shuffle* is of type *SLSL*. The application of *map transpose* swaps the middle *SL* to a *LS*, and gives *SSLL*. Then the application of *diag* converts the outermost *SS* to *SL* and returns *SLLL*. This can now be used as input to *map* $(\mathit{concat} \cdot \mathit{concat})$ which flattens the three innermost levels of lists into a single list, and returns *SL*.

$$\mathit{shuffle} = \mathit{map}\ (\mathit{concat} \cdot \mathit{concat}) \cdot \mathit{diag} \cdot \mathit{map}\ \mathit{transpose}$$

In this model the & operator is not quite associative, but it *is* associative modulo the permutation of finite lists of answers. Some other algebraic properties of the operators also need to be re-interpreted as holding modulo permutation.

To implement *both* depth-first search and breadth-first search in the embedding, the model has to be further refined. It is not sufficient to implement predicates as functions returning streams of answer lists; they have to operate on forests of trees. The operators $\|$ and & are redefined to be operations on trees, where the first one connects two subtrees in a single tree and the second "grafts" trees with small subtrees at the leaves to form normal trees. This is described in our next paper.

It is interesting how concise the definitions of $\|$ and & remain in all three models. To recapitulate the three definitions of & in the depth-first model, breadth-first model and the tree model which accommodates both search strategies, respectively:

$$p \mathbin{\&} q = \mathit{concat} \cdot \mathit{map}\ q \cdot p$$
$$p \mathbin{\&} q = \mathit{shuffle} \cdot \mathit{map}\ (\mathit{map}\ q) \cdot p$$
$$p \mathbin{\&} q = \mathit{graft} \cdot \mathit{treemap}\ q \cdot p$$

These closely parallel definitions hint at a deeper algebraic structure, and in fact the definitions are all instances of the so-called Kleisli construction from category theory. This topic has been explored for the special case of exceptions in [21], and we give a more detailed study of it in [19].

6 Related and Further Work

There has been a long effort to combine the two important paradigms of declarative programming, functional and logic programming (some good surveys are [1, 3, 5, 10, 16]) and that effort is still ongoing. The two primary goals of the research in this area of paradigm integration are to make tools that exploit the most powerful concepts from both paradigms and to gain a better understanding of declarative computing. It is a fruitful combination; functional programming contributes higher-order functions and efficient operational behaviour whereas logic programming contributes function inversion, partial data structures and logical variables.

It is hoped that this integration can reduce the duplication of research and the fragmentation in the field of declarative programming. The integration has been approached through the implementation of languages that combine concepts from logic and functional programming. Some recently developed languages that combine the two paradigms are Babel [13], Kernel-LEAF [4], Escher [9] and Curry [6]. This approach provides programmers with efficient hybrid tools for declarative programming, but most of those implementations lack the semantical clarity that our embedding possesses.

LogLisp [18, 17] and a few other languages embed logic programming in Lisp. The embedding is at the same level of abstraction as ours (this is raised as the main point in [8]), but they do not have an equally expressive base language, since Lisp is eager and untyped. By using lazy streams of answers we get a natural model for backtracking and the possibly infinite search space of Prolog. By using types to describe the predicates and their answers, we can easily alter the base answer type and thereby replace the default depth-first search strategy with others.

The work presented in this paper has not addressed the question of an efficient implementation of these ideas, although a language implementation based on our embedding is conceivable. Rather, this work is directed towards producing and using a theoretical tool (with a simple implementation) for the analysis of different aspects of logic programs. The simplicity is the key idea and the main strength of our embedding, and it has served well in opening several directions for further research.

We are presently working on two applications of the embedding. One is a study of program transformation by equational reasoning, using the algebraic laws of the embedding. The other is a categorical study of a model in which trees are used as the data-structure for the answers, and we show that there exists a *morphism of monads* between the new tree model and the stream model that is presented in this paper. This line of research is inspired by [23, 12, 20].

Among other questions that we plan to address soon are also the implementation of higher-order functions and the implementation of nested functions in the embedded predicates. Constraint logic programming also has a simple model in our embedding: one only has to pass equations (instead of substitutions) as parts of answers. These equations are evaluated when they become sufficiently instantiated. An efficient language implementation is also a challenging goal in this setting.

References

[1] M. Bellia and G. Levi. The relation between logic and functional languages: a survey. *Journal of Logic Programming*, 3(3):317–236, 1986.

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

[3] J. Darlington, Y. Guo, and H. Pull. A new perspective on integrating functional and logic languages. In *Proceedings of FGCS*, pages 682–693, 1992.

[4] E. Giovanetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel-LEAF: A logic plus functional language. *Journal of Computer and System Sciences*, 42(2):139–185, April 1991.

[5] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19(20):583–628, 1994.

[6] M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A truly functional logic language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.

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

[8] H.J. Komorowski. Qlog – the programming environment for Prolog in Lisp. In K. Clark and Tarnlund S.A., editors, *Logic Programming.* Academic, New York, 1982.

[9] J.W. Lloyd. Declarative programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, University of Bristol, June 1995.

[10] J.W. Lloyd. Programming in an integrated functional and logic language. *The Journal of Functional and Logic Programming*, 1998. to appear.

[11] R. McPhee and O. de Moor. Compositional logic programming. In *Proceedings of the JICSLP'96 post-conference workshop: Multi-paradigm logic programming*, Report 96-28. Technische Universität Berlin, 1996.

[12] E. Moggi. Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science*. IEEE, June 1989.

[13] J. Moreno-Navarro and M. Roderiguez-Artalejo. Logic programming with functions and predicates: The language Babel. *Journal of Logic Programming*, 12(3):191–223, 1992.

[14] L.C. Paulson. Lessons learned from LCF: a survey of natural deduction proofs. *Computer Journal*, pages 474–479, 1985.

[15] U.S. Reddy. Narrowing as the operational semantics of functional languages. In *Symposium on Logic Programming*, Boston, 1985. IEEE.

[16] U.S. Reddy. Functional logic languages part I. In *Proceedings of a Workshop on Graph Reduction*, volume 279 of *Lecture Notes in Computer Science*, pages 401–425. Springer-Verlag, 1987.

[17] J.A. Robinson. Beyond LogLisp: combining functional and relational programming in a reduction setting. *Machine intelligence*, 11:57–68, 1988.

[18] J.A. Robinson and E.E. Sibert. LogLisp: An alternative to Prolog. *Machine Intelligence*, 10:399–419, 1982.

[19] S. Seres, J.M. Spivey, and Hoare C.A.R. Algrebra of logic programming. to be submitted to International Conference on Logic Programming, 1999.

[20] J.M. Spivey. A categorial approach to the theory of lists. In *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 399–408, 1989.

[21] J.M. Spivey. A functional theory of exceptions. *Science of Computer Programming*, 14(1):25–42, 1990.

[22] P. Wadler. How to replace failure by a list of successes. In *2'nd International Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985. Springer-Verlag.

[23] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.