J. Functional Programming **1** (1): 1–000, January 1998. Printed in the United Kingdom

FUNCTIONAL PEARLS

Combinators for Breadth-First Search

MICHAEL SPIVEY

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

1 Introduction

Every functional programmer knows the technique of "replacing failure by a list of successes" (Wadler, 1985), but wise programmers are aware also of the possibility that the list will be empty or (worse) divergent. In fact, the "lists of successes" technique is equivalent to the incomplete depth-first search strategy used in Prolog.

At heart, the idea is quite simple: whenever we might want to use a 'multi-function' such as 'f' $:: \alpha \twoheadrightarrow \beta$ that can return many results or none, we replace it by a genuine function $f :: \alpha \to \beta\ stream$ that returns a lazy stream of results,and rely on lazy evaluation to compute the answers one at a time, and only as they are needed. For the sake of clarity, I will distinguish between the types of finite lists ($\alpha\ list$) and of potentially infinite, lazy streams ($\alpha\ stream$), though both may be implemented in the same way. Following the conventions used in ML, type constructors follow their argument types.

Give two such functions, $f :: \alpha \to \beta\ stream$ and $g :: \beta \to \gamma\ stream$, we can define their composition $g \wedge f :: \alpha \to \gamma\ stream$ by

$$g \wedge f = concat \cdot g* \cdot f,$$

where (using notation introduced by Bird (1987)) $g* = map\ g$ denotes the function of type

$$\beta\ stream \to (\beta\ stream)\ stream$$

that applies g to each element of its argument stream, and collects the results as a new stream – a stream of streams, because each result returned by g is itself a stream. Equivalently, we can define the composition operator by

$$(g \wedge f)\ x = [\, z \mid y \leftarrow f\ x; z \leftarrow g\ y \,],$$

using list comprehension in place of explicit functions on lists. We use the symbol $\wedge$ for composition, because as we shall see, composition is closely related to conjunction in Prolog.

This composition operator is associative, and has as a unit element the function $unit :: \alpha \to \alpha\ stream$ defined by $unit\ x = [x]$. Because we shall want to prove associativity later for other versions of the composition operator, we give the simple proof here. The composition $(h \wedge g) \wedge f$ simplifies as follows, using the functor law,

that $(q \cdot p)* = q* \cdot p*$:

$$\begin{aligned}&(h \wedge g) \wedge f\\ &\quad = \quad concat \cdot (concat \cdot h* \cdot g)* \cdot f\\ &\quad = \quad concat \cdot concat* \cdot h** \cdot g* \cdot f.\end{aligned}$$

The composition $h \wedge (g \wedge f)$ simplifies as follows:

$$\begin{aligned}&h \wedge (g \wedge f)\\ &\quad = \quad concat \cdot h* \cdot concat \cdot g* \cdot f\\ &\quad = \quad concat \cdot concat \cdot h** \cdot g* \cdot f.\end{aligned}$$

Here we use the fact that $h* \cdot concat = concat \cdot h**$, which holds because *concat* is a natural transformation. The two expressions we have obtained are equal because of the law,

$$concat \cdot concat* = concat \cdot concat.$$

This law is intimately connected with the associativity of $\wedge$, so we pronounce it "*concat* is associative".

For an example of the use of the $\wedge$ combinator, let us consider a simple program for finding the factors of numbers. Define a function $choose :: int\ list \to (int\ list)\ stream$ by

$$choose\ xs = [\, xs +\!\!+ [x] \mid x \leftarrow [1\,..\,]\,].$$

This function takes a list of potential factors already chosen, and extends it in every possible way with an additional choice of factor, returning the results as a lazy stream. Now define a function $test :: int \to int\ list \to (int\ list)\ stream$ by

$$test\ n\ [x, y] = \mathbf{if}\ x \times y = n\ \mathbf{then}\ [[x, y]]\ \mathbf{else}\ [\,].$$

The function $test\ n$ takes a pair of potential factors $[x, y]$ and returns just that pair if they are factors of n, and nothing otherwise. With these definitions, we can define a function *factorize* as follows:

$$factorize\ n = (test\ n \wedge choose \wedge choose)\ [\,].$$

This function expresses the idea of choosing a pair of factors in every possible way, then filtering out the pairs that actually multiply to give the desired product n.

Unfortunately, the behaviour of this function is far from ideal. For example, the result of *factorize* 60 is not a list of different factorizations, but just $[1, 60]{:}\bot$. That is, the program returns one answer, then it diverges, computing forever without either returning another answer or yielding the empty list. The reason for this is easy enough to see: the right-hand *choose* produces the stream

$$[[1], [2], [3], \ldots],$$

the left-hand *choose* adds a second choice, producing

$$[1, 1], [1, 2], [1, 3], \ldots].$$

Because this invocation of *choose* produces an infinite stream of results from the

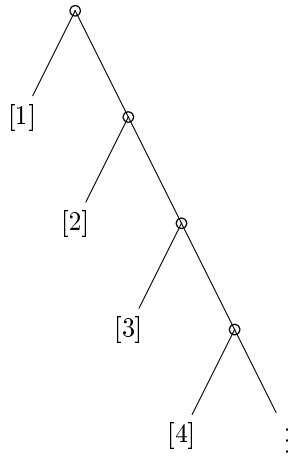


Fig. 1. A tree of choices

first result returned by the right-hand *choose*, we never reach any choice that has 2 as its first element. In consequence, the *test* function examines an infinite stream of pairs $[[1, 61], [1, 62], \ldots]$, without ever finding a second solution.

This behaviour is exactly the one produced by a Prolog program that is written in the same way. The Prolog program,

```
factorize(N, X, Y) :- choose(X), choose(Y), X * Y =:= N.
```

will solve a goal such as `?- factorize(60, X, Y)` by first fixing on a choice of X, then exploring every choice for Y; if there are infinitely many choices of Y, then it will never revise its initial choice of X, and so never reach $X = 2$. In short, both programs embody depth-first search: the Prolog program does so implicitly, because depth-first search is part of the meaning of Prolog, but the functional program does so explicitly, because depth-first search emerges as a consequence of the way we have implemented lists of successes.

2 Breadth-first search

A better search strategy for this program relies on viewing the choice of X not as an atomic action, but as a sequence of finite choices arranged as a tree. It does not greatly matter what shape the tree has, so long as each node has only finitely many children; for simplicity, we can choose the tree structure shown in Figure 1. To each leaf of the tree, we assign a *cost* that is related to the number of choices made in reaching the leaf. Let us say that the cost of choosing $[n]$ is $n - 1$.

In a program like *choose* $\wedge$ *choose*, two choices are made, and we can visualize these choices as a composite tree (Figure 2). We begin with the tree structure generated by the first choice, and at each leaf, we graft in a copy of the tree generated by the second choice. Each level in this tree corresponds to a different total cost of making the two choices. A fair search strategy visits each leaf of the composite tree in finite time, unlike depth-first search, which can become stuck in one infinite branch of the tree while leaves remain unvisited in other branches.

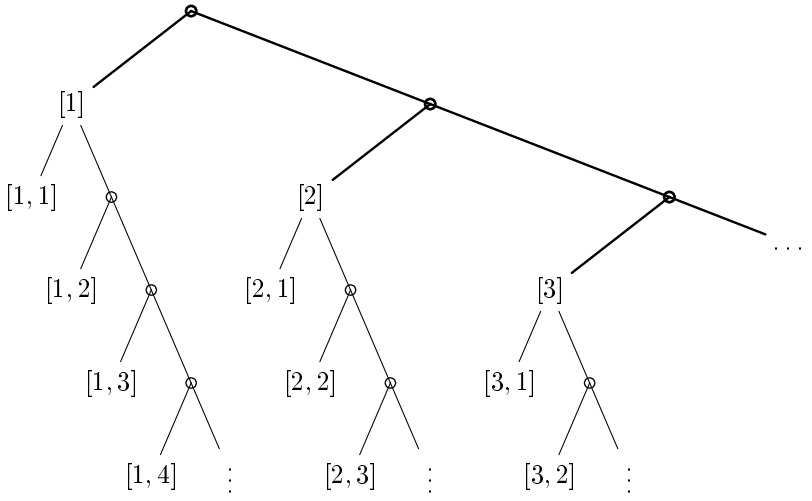


Fig. 2. Nested choices

In breadth-first search, we visit the pairs $[x, y]$ in order of increasing cost; this guarantees that we will reach each pair eventually. This means searching the tree level-by-level. In order to implement this idea, we must replace the type α *stream* that we have used to represent the results of a computation by a type in which the results can be presented in order of increasing cost. For this, we use the type α *matrix* $=$ (α *list*) *stream*. A value of this type consists of an infinite stream of finite lists, each corresponding to successive costs, counting from zero. Each finite list contains all the solutions of a particular cost. The name 'matrix' was suggested by Silvija Seres, and refers to the idea that we present the results row-by-row. Unlike conventional matrices, ours have infinitely many rows, and the rows, though finite, differ in length.

To use these types in our factorization example, we should redefine *choose* so that it returns a stream of finite lists, in which each list contains one choice:

$$\mathit{choose}\ \mathit{xs} = [\,[\mathit{xs} +\!\!+ [x]] \mid x \leftarrow [1\,..\,]\,].$$

We also need to redefine *test* so that it returns its answers as a stream of lists:

$$\mathit{test}\ n\ [x, y] = \mathbf{if}\ x \times y = n\ \mathbf{then}\ [[[x, y]]]\ \mathbf{else}\ [\,].$$

Here, the result $[[[x, y]]]$ denotes the pair $[x, y]$ as a single answer with cost zero. With these auxiliary definitions, we ought to be able to define

$$\mathit{factorize}\ n = (\mathit{test}\ n \wedge' \mathit{choose} \wedge' \mathit{choose})\ [\,],$$

and compute all factorizations of n as answers, not just the single answer $[1, n]$. The program will still diverge, in the sense that after all factorizations have been found, the stream of lists will continue, showing the empty list of answers for each subsequent level; since the search space is infinite, this is unavoidable, except by changing the algorithm and thus making the search space finite.

3 Composition

The piece of the puzzle still missing is a definition of the new composition operator $\wedge'$ that works with our new result type. The rest of this article is devoted to finding a suitable definition for this operator, and verifying that it is associative, as it must be if we are to write expressions like $test\ n \wedge' choose \wedge' choose$ without fear of ambiguity.

If $f :: \alpha \to \beta\ matrix$ and $g :: \beta \to \gamma\ matrix$, an analogy with depth-first search suggests defining $g \wedge' f$ as

$$g \wedge' f = join \cdot g{**} \cdot f,$$

where *join* is an (as yet unknown) function of type $(\gamma\ matrix)\ matrix \to \gamma\ matrix$. This definition raises the hope that we will be able to prove that $\wedge'$ is associative by repeating the argument we used earlier, and relying on the associativity of our function *join*.

We will now define some auxiliary functions that can be assembled to produce the desired function *join*. For simplicity, let us assume from this point on that all potentially infinite streams are actually infinite, so that we do not need to deal with the possibility that they terminate. The definitions we give could be augmented with extra clauses that deal with this case, but it would complicate our presentation to do so, without adding much of substance to the discussion.

A useful function $trans :: (\alpha\ stream)\ list \to (\alpha\ list)\ stream$ may be defined as a fold on lists:

$$\begin{aligned} trans\ [\,] &= repeat\ [\,] \\ trans\ (xs{:}xss) &= zipWith\ (:)\ xs\ (trans\ xss), \end{aligned}$$

in other words, $trans = foldr\ (zipWith\ (:))\ (repeat\ [])$. An alternative definition, better for our purposes, can formulated as an unfold on streams:

$$trans\ xss = (head * xss){:}(trans\ (tail * xss)).$$

For example,

$$\begin{aligned} &trans\ [[x_{00}, x_{01}, x_{02}, \ldots], [x_{10}, x_{11}, x_{12}, \ldots], \ldots, \\ &\qquad [x_{(n-1)\,0}, x_{(n-1)\,1}, x_{(n-1)\,2}, \ldots]] \\ = \quad &[[x_{00}, x_{10}, \ldots, x_{(n-1)\,0}], [x_{01}, x_{11}, \ldots, x_{(n-1)\,1}], \\ &\qquad [x_{02}, x_{12}, \ldots, x_{(n-1)\,2}], \ldots]. \end{aligned}$$

Since *join* takes arguments of type

$$(\alpha\ matrix)\ matrix = (((\alpha\ list)\ \underline{stream})\ \underline{list})\ stream,$$

it is tempting to use *trans* to switch the occurrences of *stream* and *list* that are underlined in this type expression, and define *join* as the following composition, where for compactness, I have written $\alpha LSLS$, etc., in place of types such as $(((\alpha\ list)\ stream)\ list)\ stream$:

$$\alpha LSLS \xrightarrow{\quad trans{*}\quad} \alpha LLSS \xrightarrow{\quad concat_L \star concat_S\quad} \alpha LS$$

where $concat_L \star concat_S = concat_L{*} \cdot concat_S = concat_S \cdot concat_L{**}$ is a combination of the concatenation functions $concat_L$ on finite lists and $concat_S$ in streams. Unfortunately, this definition is wrong, because the use of $concat_S$ fails to take into account the requirement that the result should be arranged in order of increasing total cost.

To reflect this requirement faithfully, we need an additional component: the function $diag :: (\alpha\ stream)\ stream \to (\alpha\ list)\ stream$, defined by

$$diag\ ((x{:}xs){:}xss) = [x]{:}(zipWith\ (:)\ xs\ (diag\ xs)).$$

For example,

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

This function takes a stream of streams, which we can think of as a two-dimensional infinite array, and arranges the elements into a stream of finite lists, each list corresponding to a diagonal of the two-dimensional array.

It is worth asking why this idea of diagonalization is not sufficient on its own: why do we not retain the idea that a multi-function returns a simple lazy stream of results, but define the composition operator so that it uses diagonalization? We might perhaps put

$$g \wedge f = concat \cdot diag \cdot g{*} \cdot f.$$

The answer becomes clear if we consider the effect of taking $f = choose$ and $g = test\ n \wedge choose$, so that f chooses one number, and g chooses a second number in all ways that complete the factorization of n. If f chooses a number that is not a factor of n, then g will vainly search forever for a way of completing the factorization, and the result will be divergence; if f chooses a factor of n, then g will find the complementary factor before diverging. Thus $(g{*} \cdot f)\ [\,]$ is the following stream of streams:

$$[[1, 6]{:}\bot, [2, 3]{:}\bot, [3, 2]{:}\bot, \bot, \bot, [6, 1]{:}\bot, \bot, \bot, \ldots]$$

If we now apply *diag* to this, the result is a stream of lists that diverges after the first element: $[[1, 6]]{:}\bot$. This happens because the model based on streams cannot deal with infinite failure except by diverging: there is no way to represent the cost of a computation in the answers it returns.

Returning to the model based on matrices, we can combine *trans* and *diag* to obtain a definition of the function *join* that does suit our purposes: we define it as the composition

$$\alpha LSLS \xrightarrow{trans{*}} \alpha LLSS \xrightarrow{diag} \alpha LLLS \xrightarrow{(concat \cdot concat){*}} \alpha LS$$

We hoped to find that the composition operator $\wedge'$ defined in terms of this *join* is associative, but unfortunately, this turns out not to be true. Each finite list in the result returned by $h \wedge g \wedge f$ contains all solutions with a given total cost $x + y + z$, where x, y and z are the costs devoted to computing f, g and h respectively. We can picture this set of solutions as a triangle $\{\, (x, y, z) \mid x + y + z = \text{const} \,\}$ in

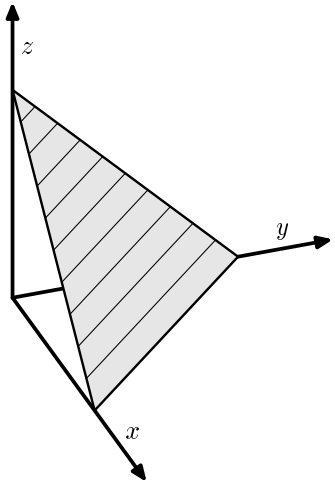


Fig. 3. The region $x + y + z = \text{const}$

positive 3-space (see Figure 3). Both $h \wedge (g \wedge f)$ and $(h \wedge g) \wedge f$ contain an element that lists all the solutions in this triangular region, but they present these solutions in different orders, corresponding to two different ways of cutting the region into strips; the figure shows the strips used in $h \wedge (g \wedge f)$.

4 Replacing lists by bags

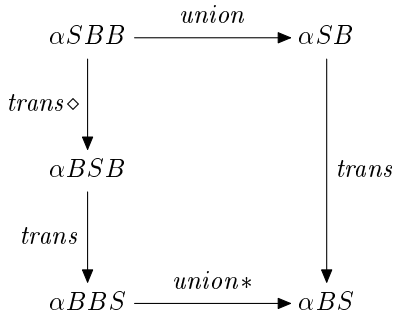
The solution to this small difficulty is simple: we must agree not to care in what order solutions of equal cost are presented, and we can do this by replacing finite lists with finite bags of type $\alpha\ bag$. If $f :: \alpha \to \beta$, we use the notation $f\diamond$ for the corresponding function $\alpha\ bag \to \beta\ bag$, and we write *union* for the function $(\alpha\ bag)\ bag \to \alpha\ bag$ that is analogous to *concat* on lists. One acceptable implementation of bags would be to represent them by finite lists, ignoring the order of elements in the list. In this case, $f\diamond$ would become $f*$ again, and *union* would be implemented by *concat*; two lists that represented bags would, however, be considered equal if one was a permutation of the other.

If we make the change from lists to bags, then *join* does become associative, in the sense that $join \cdot join = join \cdot join\diamond *$. To prove this, we need three lemmas about the interaction between the auxiliary functions *trans* and *diag* that we used to define *join*.

Lemma $\mathcal{A}$ concerns the interaction between *trans* and *union*. If we have a value of type $((\alpha\ stream)\ bag)\ bag$, we can use take the union of the bag of bags to obtain a single bag of streams, then we can use *trans* to turn this into a stream of bags. Alternatively, we can use *trans* twice on the original value, then use *union* on the result; and the final answer is the same either way:

$$trans \cdot union = union* \cdot trans \cdot trans\diamond.$$

This equation can also be represented by the following commutative diagram:

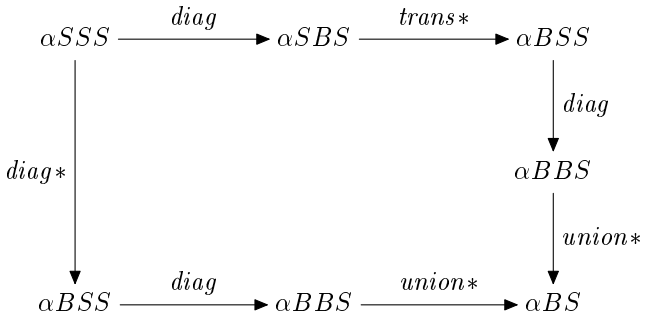


The truth of this lemma can be seen by trying a small example, and it is easily proved by induction.

Lemma $\mathcal{B}$ governs the interaction between two instances of *diag*, and can be seen as a highly modified assertion that *diag* is associative:

$$union* \cdot diag \cdot trans* \cdot diag = union* \cdot diag \cdot diag*.$$

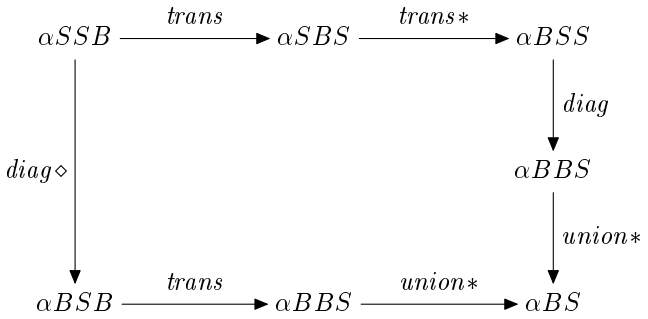
The basic pattern $diag \cdot diag = diag \cdot diag*$ is modified first by the need for an intervening *trans* to make the types come out right, then by the need to compose *union*∗ to both sides; this is needed to avoid the problem illustrated in Figure 3. This lemma is represented by the following commutative diagram:



Lemma $\mathcal{C}$ concerns a different interaction between *trans* and *diag*:

$$union* \cdot diag \cdot trans* \cdot trans = union* \cdot trans \cdot diag\diamond.$$

This is shown by the following commutative diagram:



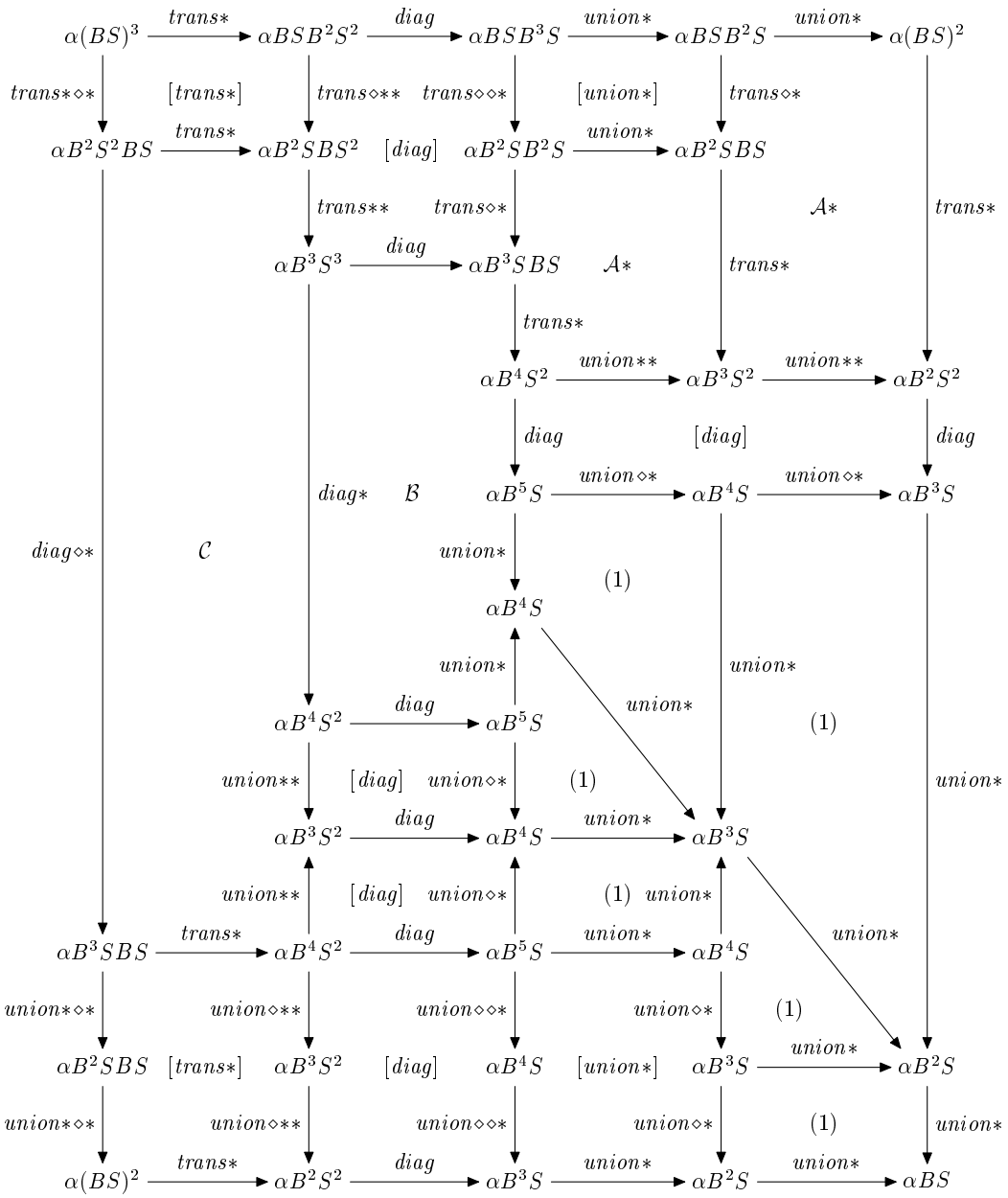


Fig. 4. Associativity of *join*

Like Lemma $\mathcal{A}$, these lemmas can be proved by simple inductions.

The three lemmas, together with some standard laws, suffice to prove the associativity of *join*. The laws in question are that $*$ and $\diamond$ are functors, in the sense that $(g \cdot f)* = g* \cdot f*$ and $(g \cdot f)\diamond = g\diamond \cdot f\diamond$, and that truly polymorphic functions like *diag* are natural transformations, in the sense that for any function f,

$$diag \cdot f** = f\diamond* \cdot diag,$$

as represented in the following commutative diagram:

$$\begin{array}{ccc} (\alpha\ stream)\ stream & \xrightarrow{f**} & (\beta\ stream)\ stream \\ \Big\downarrow{\scriptstyle diag_\alpha} & & \Big\downarrow{\scriptstyle diag_\beta} \\ (\alpha\ bag)\ stream & \xrightarrow{f\diamond*} & (\beta\ bag)\ stream \end{array}$$

The functors $**$ and $\diamond*$ that appear in this equation are determined by the type of *diag*. If a polymorphic function $t : \alpha\ T \to \alpha\ T'$ is a natural transformation, then it is easy to show that $t* : (\alpha\ T)\ list \to (\alpha\ T')\ list$ is also a natural transformation; this fact is used several times in the proof. Finally, the proof requires us to make six applications of the associative law for *union*, that $union \cdot union = union \cdot union\diamond$. By applying the functor $*$ to both sides of this equation, we obtain

$$union* \cdot union* = union* \cdot union**,$$

which we refer to as (1) below.

We now turn to the associativity of *join*, i.e., to the equation

$$join \cdot join = join \cdot join\diamond*,$$

or equivalently,

$$\begin{aligned} & union* \cdot union* \cdot diag \cdot trans* \cdot union* \cdot union* \cdot diag \cdot trans* \\ = \quad & union* \cdot union* \cdot diag \cdot trans* \\ & \quad \cdot union*\diamond* \cdot union*\diamond* \cdot diag\diamond* \cdot trans*\diamond*. \end{aligned}$$

The complete proof is shown as a diagram in Figure 4. Each cell of the diagram is labelled with the reason why it commutes: the notation $\mathcal{A}*$ denotes a copy of lemma $\mathcal{A}$ in which $*$ has been applied to both sides, and a notation like $[union*]$ refers to the fact that, e.g., $union*$ is a natural transformation. It is instructive to try this proof as a benchmark for automatic or interactive theorem proving software. The frequent "changes of direction" in the argument pose a problem for programs based on algebraic simplification, whilst the author's implementation of the Knuth-Bendix completion procedure enters an infinite computation, but with the right guidance quickly produces a set of rewrite rules sufficient to prove the desired result.

5 In conclusion

This definition of an associative composition operator for breadth-first search fits into a broader algebraic theory of search strategies for logic programming, which the author and Silvija Seres have begun to investigate in (Spivey and Seres, 2000). In addition to the operator $\wedge'$, which behaves like the 'and' of logic programming, there is an 'or' operator defined by

$$(f \vee' g)\ x = zipWith\ (+\!\!+)\ (f\ x)\ (g\ x),$$

and together they enjoy a number of algebraic properties, including a distributive law. There is a function $true :: \alpha \to \alpha\ matrix$ and a function $false :: \alpha \to \beta\ matrix$,

defined by

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

where $[\![\,]\!]$ denotes the empty bag, and $[\![x]\!]$ denotes a bag containing just x. The function *true* is a unit element for $\wedge'$, and *false* is a unit element for $\vee'$ and a zero for $\wedge'$.

The families of operators that satisfy these properties form a category, in which the model of search that produces the search tree for a logic program is an initial object. Morphisms in this category give various searching functions on these trees, showing how the same results can be obtained in a compositional fashion without forming the search tree explicitly.

The benefits of this theory include a compositional semantics for logic programming, in which the meaning of a predicate $p \wedge q$ is defined in terms of the meanings of p and q, rather than in terms of executions in which the effects of p and q are mingled (Seres *et al.*, 1999; Spivey and Seres, 1999). This semantics is able to support transformation of logic programs by algebraic rewriting. Its chief attraction is that it provides a uniform framework within which ordinary unification and constraint-based programming can be treated alike.

Acknowledgements

The author wishes to thank Silvija Seres for her close collaboration in the investigations reported here. Richard Bird and Quentin Miller made many comments that have led to substantial improvements in presentation.

References

Bird, R. S. 1987. Introduction to the theory of lists. In M. Broy (editor), *Logics of Programming and Calculi of Discrete Design*, pp. 5–42. Springer-Verlag.

Seres, S., Spivey, J. M. and Hoare, C. A. R. 1999. Algebra of logic programming. In D. De Schreye (editor), *Proceedings of the 1999 International Conference on Logic Programming*, pp. 184–199. MIT Press.

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

Spivey, J. M. and Seres, S. 2000. The algebra of searching. In J. Davies and J. C. P. Woodcock (editors), *Proceedings of a symposium in celebration of the work of C. A. R. Hoare*. MacMillan (to appear).

Wadler, P. L. 1985. How to replace failure by a list of successes. In J.-P. Jouannaud (editor), *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science, 201*, pp. 113–128. Springer-Verlag.