JFP **19** (3 & 4): 469–487, 2009.
doi:10.1017/S0956796809007321 Printed in the United Kingdom

Algebras for combinatorial search

J. MICHAEL SPIVEY

Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, UK
(*e-mail:* mike@comlab.ox.ac.uk)

Abstract

Combinatorial search strategies including depth-first, breadth-first and depth-bounded search are shown to be different implementations of a common algebraic specification that emphasizes the compositionality of the strategies. This specification is placed in a categorical setting that combines algebraic specifications and monads.

1 Introduction

Many combinatorial problems can be expressed in terms of a sequence of constrained choices, with successive choices forming a path within a tree of possibilities. We can model such trees with the type

$$\textbf{data}\ \mathit{Tree}\ \alpha = \mathit{Leaf}\ \alpha \mid \mathit{Fork}\ [\mathit{Tree}\ \alpha].$$

For example, the following function *choose*, given an integer n, describes a tree that contains all integers $\geqslant n$ (see Figure 1).

$$\begin{aligned}&\mathit{choose} :: \mathit{Integer} \to \mathit{Tree}\ \mathit{Integer}\\ &\mathit{choose}\ n = \mathit{Fork}\ [\mathit{Leaf}\ n, \mathit{choose}\ (n+1)].\end{aligned}$$

Each fork in the resulting tree corresponds to a decision whether to stop at n or to choose a number that is $n+1$ or greater. Trees can be made into a monad in a way that makes it natural to describe sequencing of choices. This monad has operations

$$\begin{aligned}&(\triangleright) :: \mathit{Tree}\ \alpha \to (\alpha \to \mathit{Tree}\ \beta) \to \mathit{Tree}\ \beta\\ &\mathit{return} :: \alpha \to \mathit{Tree}\ \alpha.\end{aligned}$$

These can express a program such as

$$\mathit{choose}\ 1 \triangleright (\lambda x \to \mathit{choose}\ 10 \triangleright (\lambda y \to \mathit{return}\ (x, y))),$$

which chooses pairs of integers (x, y) such that $x \geqslant 1$ and $y \geqslant 10$. Figure 2 shows the tree that results. The overall structure near the root of this tree is a copy of the tree shown in Figure 1, but the leaves 1, 2, 3, 4, ... of that tree have each been replaced by another tree, again with a structure that is determined by the function *choose*. The leaves of the tree are labelled with pairs (x, y).

There are some advantages in working with forests – lists of trees – rather than single trees. The operations $\triangleright$ and *return* can be redefined so as to make forests into a monad in a

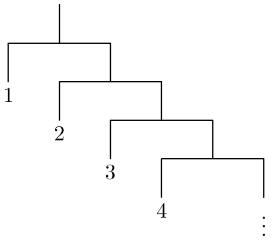


Fig. 1. The tree *choose* 1.

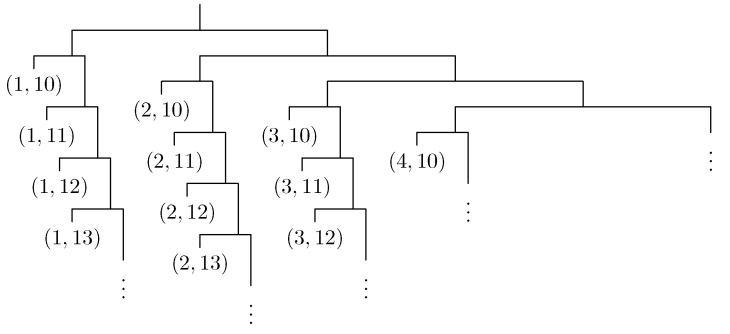


Fig. 2. A composite tree.

similar way to trees, but they also support an associative concatenation operator, with the empty forest as an identity element.

The existence of an empty forest allows us to define an operation *test* that yields a forest over the unit type ( ): either the singleton forest *return* ( ) = [*Leaf* ( )] or the empty forest *zero* = [ ] according to a Boolean condition.

$$\begin{aligned} &\mathit{test} :: \mathit{Bool} \to \mathit{Forest}\ (\,)\\ &\mathit{test}\ b = \textbf{if}\ b\ \textbf{then}\ \mathit{return}\ (\,)\ \textbf{else}\ \mathit{zero}\end{aligned}$$

Although apparently of little use, this operation lets us filter out the results that satisfy a constraint. Thus the program

$$\begin{aligned} p = {} & \mathit{choose}\ 1 \triangleright (\lambda x \to \mathit{choose}\ 1 \triangleright (\lambda y \to\\ & \quad \mathit{test}\ (x * y \equiv 24) \triangleright (\lambda(\,) \to \mathit{return}\ (x, y))))\end{aligned}$$

produces a forest that contains just those pairs (x, y) of integers $\geqslant 1$ that multiply to give 24. The expression can be written more succinctly in Haskell's **do** notation as

$$p = \textbf{do}\ x \leftarrow \mathit{choose}\ 1;\ y \leftarrow \mathit{choose}\ 1;\ \mathit{test}\ (x * y \equiv 24);\ \mathit{return}\ (x, y).$$

Although trees and forests make it easy to compose a sequence of choices, we might prefer to have the answers as a list. There are two ways of achieving this: one is to build the forest first, and then apply a function,

$$\mathit{search} :: \mathit{Forest}\ \alpha \to [\alpha],$$

that traverses the forest in (say) a depth-first order and yields the list of leaves that it finds. The other way is to work in the monad of lists instead of the monad of trees: we can define

https://doi.org/10.1017/S0956796809007321 Published online by Cambridge University Press

operations

$$(\triangleright') :: [\alpha] \to (\alpha \to [\beta]) \to [\beta]$$
$$return' :: \alpha \to [\alpha],$$

as well as others like $test' :: Bool \to [()]$, and write the entire program in terms of lists instead of forests. Using the monad of lists gives the same results as building the tree and traversing it in depth-first order, and this is a consequence of the fact that *search* is a *morphism of monads*, so that (among other things) the following algebraic law holds:

$$search\ (xt \triangleright (\lambda x \to f\ x)) = search\ xt \triangleright' (\lambda x \to search\ (f\ x)).$$

This law lets us take a program that is written using the operator $\triangleright$ that is connected with trees or forests and move one step towards rewriting it to use lists instead. Applying the law three times to the expression *search p*, where *p* is the program above, and simplifying a little using other laws, we can obtain the program

$$search\ (choose\ 1) \triangleright' (\lambda x \to search\ (choose\ 1) \triangleright' (\lambda y \to test'(x * y \equiv 24) \triangleright' (\lambda() \to return'(x, y)))).$$

We can read the law above, and other laws that relate *test* and *return* to $test'$ and $return'$, as saying that *depth-first search is compositional*, in the sense that the result of applying *search* to a compound program built up with $\triangleright$ can also be obtained by applying *search* to the parts of the program, and then combining the results with $\triangleright'$.

If we additionally use the fact that *search* (*choose n*) yields the list $[n..]$, then the whole program can be rewritten as the list comprehension

$$[(x, y) \mid x \leftarrow [1..], y \leftarrow [1..], x * y \equiv 24].$$

This list-based program for the factors of 24 is not very effective, because it finds the pair $(1, 24)$, and then enters an unending search for other numbers y such that $1 * y \equiv 24$, never reaching the solution $(2, 12)$. For this problem and other similar ones, it would be better to use another search strategy such as breadth-first search.

The ideas outlined here raise a number of questions, which the present paper aims to answer. First, what is an appropriate set of composition operators (in addition to $\triangleright$) for programs that produce search trees or forests? How can the same operators be defined on lists so that the function *search* respects each operator as it does $\triangleright$? We will see the answer to these questions in Section 2, where we introduce a set of operators similar to the *MonadPlus* class of Haskell, but with an additional operation *wrap* that permits the representation of tree-like structures.

Second, other search algorithms like breadth-first search and depth-bounded search are more effective than depth-first search, particularly on infinite trees. Can these search algorithms also be made compositional in the sense alluded to here? This question is answered in Section 3, where these algorithms – the canonical examples of 'oblivious' search algorithms in artificial intelligence – are shown to be expressible as other implementations of the same set of operators.

Third, what is an appropriate algebraic setting in which to place these results? For standard collection types like sets, bags and lists, the Boom hierarchy (Meertens 1986;

https://doi.org/10.1017/S0956796809007321 Published online by Cambridge University Press

Bird 1987) provides a uniform setting within which similar programs can be developed in a similar way. Is there an analogous setting for the monads that are associated with combinatorial search? This question is answered in Section 4, which proposes a slight generalization of the category-theoretic concept of a monad, and Section 5, where this extended concept is applied to search strategies. Section 6 contains an outline of a way in which these results can be extended from finite to infinite trees.

2 Combinators for search

The first task is to find a set of combinators that allow us to build up search trees. We will define five operations that work with trees: *zero*, *return*, $\triangleright$, $\oplus$ and *wrap*, and then look at the algebraic laws that they satisfy; in effect, this will give us an algebraic specification of search as an abstract data type. Next, we will look for other implementations that satisfy the same specification, corresponding to different search algorithms. We will show that the implementation based on trees is an initial object in a certain category, and this will imply that any other implementation can be viewed as compositional in the sense outlined in Section 1.

Rather than operating directly on trees, our initial set of combinators will work on forests – that is, finite lists of trees – because this will allow us to define the empty collection of results, and an associative *or* operator that combines two collections of results into one. Each node in a tree is either a labelled leaf or an unlabelled internal node with a finite list – a forest – of children.

$$\textbf{type}\ \mathit{Forest}\ \alpha = [\mathit{Tree}\ \alpha]$$
$$\textbf{data}\ \mathit{Tree}\ \alpha = \mathit{Leaf}\ \alpha \mid \mathit{Fork}\ (\mathit{Forest}\ \alpha)$$

There is an empty forest, which we will call *zero*, and given any value $x :: \alpha$, we can form the tiny forest *return* x that contains just the leaf x:

$$\mathit{zero} :: \mathit{Forest}\ \alpha$$
$$\mathit{zero} = [\,]$$

$$\mathit{return} :: \alpha \to \mathit{Forest}\ \alpha$$
$$\mathit{return}\ x = [\mathit{Leaf}\ x].$$

The function *return* will become the unit of our monad of forests.

Given two forests xm and ym, we can concatenate them to form a wider forest $xm \oplus ym$. Also, we can wrap up any forest xm as a new forest that contains a single, slightly taller, tree; in this way we can obtain forests that are not just lists of leaves.

$$(\oplus) :: \mathit{Forest}\ \alpha \to \mathit{Forest}\ \alpha \to \mathit{Forest}\ \alpha$$
$$xm \oplus ym = xm +\!\!+ ym$$

$$\mathit{wrap} :: \mathit{Forest}\ \alpha \to \mathit{Forest}\ \alpha$$
$$\mathit{wrap}\ xm = [\mathit{Fork}\ xm].$$

It is easy to see by an inductive argument that any forest can be constructed using the operations we have so far introduced.

https://doi.org/10.1017/S0956796809007321 Published online by Cambridge University Press

Using $\oplus$, *wrap* and *return*, we can define a function *choose* that produces the single tree shown in Figure 1:

$$\begin{aligned}&\mathit{choose} :: \mathit{Integer} \to \mathit{Forest}\ \mathit{Integer}\\&\mathit{choose}\ n = \mathit{wrap}\ (\mathit{return}\ n \oplus \mathit{choose}\ (n+1)).\end{aligned}$$

The final operation in our suite of combinators is the binding operator $\triangleright$ that makes forests into a monad. As we shall see later, the definition of this operator is more or less forced on us by algebraic considerations, but for now we can give a recursive definition as follows:

$$\begin{aligned}&(\triangleright) :: \mathit{Forest}\ \alpha \to (\alpha \to \mathit{Forest}\ \beta) \to \mathit{Forest}\ \beta\\&[\,] \triangleright f = [\,]\\&(\mathit{Leaf}\ x : xm) \triangleright f = f\ x +\!\!+ (xm \triangleright f)\\&(\mathit{Fork}\ ym : xm) \triangleright f = \mathit{Fork}\ (ym \triangleright f) : (xm \triangleright f).\end{aligned}$$

Now we turn to the algebraic laws that are satisfied by the operations we have introduced; these are important because we will require that they should also hold for the other search strategies we wish to consider. First of all, we find that *return* and $\triangleright$ form a monad so that the following laws apply:

$$(xm \triangleright f) \triangleright g = xm \triangleright (\lambda x \to f\ x \triangleright g), \tag{1}$$

$$(\mathit{return}\ x) \triangleright f = f\ x, \tag{2}$$

$$xm \triangleright \mathit{return} = xm. \tag{3}$$

We also find that *zero* and $\oplus$ form a monoid:

$$(xm \oplus ym) \oplus zm = xm \oplus (ym \oplus zm), \tag{4}$$

$$\mathit{zero} \oplus xm = xm, \tag{5}$$

$$xm \oplus \mathit{zero} = xm. \tag{6}$$

The monad and the monoid interact according to the following laws, which state that for any function $f :: \alpha \to \mathit{Forest}\ \beta$, the function $(\triangleright f) :: \mathit{Forest}\ \alpha \to \mathit{Forest}\ \beta$ is a homomorphism of monoids:

$$\mathit{zero} \triangleright f = \mathit{zero}, \tag{7}$$

$$(xm \oplus ym) \triangleright f = (xm \triangleright f) \oplus (ym \triangleright f). \tag{8}$$

Finally, *wrap* and $\triangleright$ are related by the law

$$(\mathit{wrap}\ xm) \triangleright f = \mathit{wrap}\ (xm \triangleright f). \tag{9}$$

Now we define a *bunch* to be a type constructor M, together with five polymorphic operations,

$$\begin{aligned}&\mathit{return} :: \alpha \to M\alpha,\\&(\triangleright) \quad :: M\alpha \to (\alpha \to M\beta) \to M\beta,\\&\mathit{zero} \quad :: M\alpha,\\&(\oplus) \quad :: M\alpha \to M\alpha \to M\alpha,\\&\mathit{wrap} \quad :: M\alpha \to M\alpha,\end{aligned}$$

https://doi.org/10.1017/S0956796809007321 Published online by Cambridge University Press

such that the laws (2) to (9) are satisfied. As we shall see in the next section, several well-known search methods can be expressed as bunches.

We have observed in earlier work (Spivey & Seres 2003) that Haskell's system of *type classes* provides a good framework for programming different bunches so that an entire program can be written in a way that is indifferent to the bunch that is to be used for its execution. We forbear to do this in the present paper, as it is not needed for the exposition, and we can therefore also suppress the **newtype** construction that is needed in Haskell to allow instances of type classes to be associated with compound type constructors.

3 Examples of bunches

3.1 Depth-first search

To obtain a bunch that gives the effect of depth-first search, we can take $M\alpha$ to be the type of (lazy) streams over α, and define the five operations as follows:

$$\textbf{type}\ M\alpha = \mathit{Stream}\ \alpha$$

$$\mathit{return}\ x = [x]$$

$$xm \triangleright f = \mathit{concat}\ (\mathit{map}\ f\ xm)$$

$$\mathit{zero} = [\,]$$

$$xm \oplus ym = xm +\!\!+ ym$$

$$\mathit{wrap}\ xm = xm$$

(We use $\mathit{Stream}\ \alpha$ as a synonym for $[\alpha]$ in order to emphasize that the streams are potentially infinite.) These definitions do in fact satisfy the nine laws needed for a bunch.

3.2 Breadth-first search

A bunch that gives the effect of breadth-first search can be based on the type of *streams of bags* over α, with the idea that successive (finite) bags in the potentially infinite stream give the solutions from successive levels of the search tree. The stream is finite or infinite according to whether the search tree for the same problem is finite or infinite.

Let us write $\mathit{Bag}\ \alpha$ for the type of bags over α, and $\mathit{bag}\ xs$ for the bag whose members are listed as xs. We write $\uplus$ for bag union, and *bagmap* and *bagfold* for the obvious functions with types

$$\mathit{bagmap} :: (\alpha \to \beta) \to \mathit{Bag}\ \alpha \to \mathit{Bag}\ \beta$$
$$\mathit{bagfold} :: (\alpha \to \alpha \to \alpha) \to \alpha \to \mathit{Bag}\ \alpha \to \alpha,$$

with the proviso that $\mathit{bagfold}\ f\ e$ is well defined only if f is associative and commutative with unit element e.

With these conventions, we can define the basic bunch operators as follows:

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

$$\mathit{return}\ x = [\mathit{bag}\ [x]]$$

$$\mathit{zero} = [\,].$$

https://doi.org/10.1017/S0956796809007321 Published online by Cambridge University Press

If we think of the bags in xm and ym as successive levels in two forests, it becomes clear that joining the two forests together corresponds to taking the union of each level, so we define,

$$xm \oplus ym = longZipWith\ (\uplus)\ xm\ ym,$$

where *longZipWith* is a version of *zipWith* that works even when its arguments are lists of different lengths. When one list is exhausted, the remainder of the other list is copied over to the result:

$$\begin{aligned}
&longZipWith :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha] \to [\alpha]\\
&longZipWith\ f\ (x : xs)\ (y : ys) = (f\ x\ y) : longZipWith\ f\ xs\ ys\\
&longZipWith\ f\ [\,]\ ys = ys\\
&longZipWith\ f\ xs\ [\,] = xs.
\end{aligned}$$

Obviously, since $\uplus$ is associative and commutative, so is $longZipWith\ (\uplus)$, and the empty list *zero* is a unit element.

To define *wrap*, we need to observe that when a forest becomes wrapped up as a single tree, the first level of that tree contains no leaves, and each subsequent level contains the same leaves as the *preceding* level of the original forest. This motivates the definition

$$wrap\ xm = [\,] : xm.$$

To complete the bunch, we need to find an appropriate definition of $\triangleright$. We can use Equations (2), (7), (8) and (9) to work out what this definition must be, though it will still remain to show that the definition we derive has the appropriate properties.

Equation (7) gives us immediately that

$$[\,] \triangleright f \quad = \quad [\,].$$

Also, if $xb = bag\ [x_1, \ldots, x_n]$ is any finite bag, then we can write $[xb] = return\ x_1 \oplus \cdots \oplus return\ x_n$, and so a stream $xb : xbs$ can be written

$$xb : xbs = [xb] \oplus ([\,] : xbs) = (return\ x_1 \oplus \cdots \oplus return\ x_n) \oplus wrap\ xbs.$$

Applying Equations (8), (2) and (9) to this, we obtain

$$\begin{aligned}
(xb : xbs) \triangleright f \quad &= \quad (f\ x_1 \oplus \cdots \oplus f\ x_n) \oplus wrap\ (xbs \triangleright f)\\
&= \quad join\ (bagmap\ f\ xb) \oplus wrap\ (xbs \triangleright f),
\end{aligned}$$

where $join = bagfold\ (\oplus)\ zero$. This yields the following definition of $\triangleright$:

$$\begin{aligned}
&[\,] \triangleright f = [\,]\\
&(xb : xbs) \triangleright f = join\ (bagmap\ f\ xb) \oplus wrap\ (xbs \triangleright f).
\end{aligned}$$

We will discuss later in this paper why we can be sure that the operation defined by these equations satisfies all the laws needed to be a bunch.

As we explained in an earlier paper (Spivey & Seres 2003), and will later explore from a different point of view, it is necessary to use streams of *bags*, ignoring the order of elements in each layer, if the composition operator $\triangleright$ is to obey the associative law (1). If we try to use streams of lists instead, then in Equation (1), the same results are obtained on both sides, but they may appear in a different order. Our use of bags really amounts to a convention that

https://doi.org/10.1017/S0956796809007321 Published online by Cambridge University Press

the order of results within each level is not significant; with this convention, it is perfectly acceptable to implement bags and bag union as lists with concatenation.

3.3 Depth-bounded search

Standard texts on artificial intelligence – for example, Russell and Norvig (2003) – observe that breadth-first search typically has large memory requirements, and recommend *depth-bounded search* as a more efficient alternative. This is similar to depth-first search (and so can be implemented efficiently with a stack), but the search is made finite by cutting off all branches of the search tree at a given depth. In order to make the search complete, the technique of *iterative deepening* is used: this amounts to performing repeated searches with a depth bound that slowly increases. Naturally, this entails recomputing the shallow parts of the tree repeatedly, but in practice this is often faster than maintaining the data structure that would be needed (as in breadth-first search) to avoid the recomputation.

In order to make depth-bounded search compositional, we need to know the depth at which each answer is found, and so we perform the search with a function that takes a depth bound and returns a list of answers, each paired with the unused portion of the bound:

$$\textbf{type } \mathit{DBound}\ \alpha = \mathit{Int} \to [(\alpha, \mathit{Int})].$$

Thus if the supplied bound is n, and the answer x occurs at depth $d \leqslant n$, then the pair $(x, n-d)$ will appear among the results.

For consistency, we require that each answer that appears for a depth bound n should continue to appear for each greater bound $n+k$. Thus if $p :: \mathit{DBound}\ \alpha$ and $k \geqslant 0$, then $p\,n$ contains (x, r) if and only if $p\,(n+k)$ contains $(x, r+k)$. To implement iterative deepening, we simply need to form the list,

$$\mathit{deepen}\ p = [\,x \mid n \leftarrow [0\,..], (x, r) \leftarrow p\,n, r \equiv 0\,].$$

The basic operations *return* and $\triangleright$ are defined as follows:

$$\mathit{return}\ x = (\lambda n \to [(x, n)])$$

$$p \triangleright f = (\lambda n \to [\,(y, s) \mid (x, r) \leftarrow p\,n, (y, s) \leftarrow f\ x\ r\,]).$$

These satisfy the equations needed for a monad; for example, the associative law (1) is satisfied because both $(p \triangleright f) \triangleright g$ and $p \triangleright (\lambda x \to f\ x \triangleright g)$, when applied to a bound n, yield the list,

$$[\,(z, t) \mid (x, r) \leftarrow p\,n, (y, s) \leftarrow f\ x\ r, (z, t) \leftarrow g\ y\ s\,].$$

The convention of pairing each solution with the remaining depth budget simplifies the details of both the definition of $\triangleright$ and the argument that is needed to prove associativity.

The extra operations *zero*, $\oplus$ and *wrap* are equally straightforward to define:

$$\mathit{zero} = (\lambda n \to [\,])$$

$$p_1 \oplus p_2 = (\lambda n \to p_1\,n +\!\!+ p_2\,n)$$

$$\mathit{wrap}\ p = q$$

https://doi.org/10.1017/S0956796809007321 Published online by Cambridge University Press

where
$q\ 0 = [\,]$
$q\ (n+1) = p\ n.$

Again, the idea behind *wrap* can be understood by thinking about forests. When a forest is wrapped up as a single tree, that tree contains no solutions at level 0, and the solutions with depth $\leqslant n+1$ in the tree are the same as those at depth $\leqslant n$ in the original forest. The remaining depth budget that is paired with each solution is unchanged.

3.4 Non-examples

Equation (8), the distributive law $(xm \oplus ym) \triangleright f = (xm \triangleright f) \oplus (ym \triangleright f)$, is interesting because it is not satisfied by a putative bunch based on the type constructor *Maybe*. It is straightforward to make this type constructor into a monad, and we can define an associative operation $\oplus$ by

$(\oplus) :: Maybe\ \alpha \to Maybe\ \alpha \to Maybe\ \alpha$
$Nothing \oplus ym = ym$
$Just\ x \oplus ym = Just\ x.$

However, these do not satisfy the equation. If *even* is a function such that $even\ 2 = Just\ 2$ and $even\ 3 = Nothing$, then

$$(Just\ 3 \oplus Just\ 2) \triangleright even = Just\ 3 \triangleright even = Nothing,$$

but

$$(Just\ 3 \triangleright even) \oplus (Just\ 2 \triangleright even) = Nothing \oplus Just\ 2 = Just\ 2,$$

so the law does not hold. This reflects the fact that once a computation of type $Maybe\ \alpha$ has delivered a result, it is not possible to backtrack and obtain a different result if the first one causes the rest of the computation to fail. This means that the sort of non-backtracking failure represented by the *Maybe* type is not part of the family of search strategies being considered here, even though *Maybe* is made a member of the *MonadPlus* type class in Haskell's standard library.

Another non-example of our laws can be found in a recent paper by Kiselyov *et al.* (2005). They enhance a stream-based model of backtracking with alternative combinators that implement fair interleaving (including *bindi*, a fair version of $\triangleright$), and they show how this interleaving can be realized in a continuation-passing style. As we have noted earlier (Spivey & Seres 2003), such an attempt cannot yield a monad, because there is no way for a computation that returns a stream to signal to its environment that it has done some work but found no answers. Briefly, in the computation

do $a \leftarrow [2\,..]$; (**do** $b \leftarrow [2\,..]$; $test\ (a * b \equiv 9)$),

the inner computation (**do** $b \leftarrow \cdots$) necessarily diverges when applied to the value $a = 2$. However, the computation

do $(a, b) \leftarrow interleave\ [2\,..]\ [2\,..]$; $test\ (a * b \equiv 9)$
where $interleave\ xm\ ym =$ **do** $x \leftarrow xm$; $y \leftarrow ym$; $return\ (x, y)$,

https://doi.org/10.1017/S0956796809007321 Published online by Cambridge University Press

which ought to be equivalent according to Equation (1), does have a chance to interleave the two streams of integers fairly, and so can produce the answer (3, 3) – before also diverging in the hunt for other pairs that multiply to give 9. The failure of these two expressions to be equivalent boils down to a failure of the *bindi* combinator to satisfy the associative law (1). This failure of associativity seems unimportant in a small example such as this, because it is no trouble to write the second of the two expressions in place of the first; but in a larger example, the two non-deterministic choices may be far apart in the text of a complex program, and it may be impossible to bring them together in order to make the interleaving fair without damaging the structure of the program.

4 A categorical view

A more abstract view of the relationships between different search strategies can be obtained using concepts from category theory. We know that each strategy involves a monad, but a strategy is more than that, because it also contains the combining operators $\oplus$, *wrap* and *zero*. This makes it natural to consider a situation where there are two related categories: the category $\mathscr{X}$ of ordinary types and functions, and a category $\mathscr{A}$, in which the objects are 'algebras' consisting of a type that is equipped with these operations, and the arrows are homomorphisms between these algebras. The monads we are interested in somehow create algebras rather than just types, and we seek an appropriate setting in which to study such monads. In the next section, we shall focus on the situation where the category $\mathscr{A}$ is a category of homogeneous algebras with specified operations that satisfy certain equations. First, however, it is fruitful to study briefly a more abstract setting in which we assume no more than a pair of categories $\mathscr{X}$ and $\mathscr{A}$, with a 'forgetful' functor $U : \mathscr{A} \to \mathscr{X}$.

It may be helpful to give a reminder of how the notation for monads that is used in category theory is related to that used in functional programming. In programming, a monad consists of a type constructor M, together with polymorphic functions $return :: \alpha \to M\alpha$ and $(\triangleright) :: M\alpha \to (\alpha \to M\beta) \to M\beta$. In the notation of category theory, each instance of *return* becomes an arrow $\eta_x : x \to Mx$. For each arrow $f : x \to My$, we can form an arrow $f^* : Mx \to My$. This corresponds to the function that maps $xm :: M\alpha$ to $xm \triangleright f :: M\,\beta$, in other words, to the operator section $(\triangleright f)$. In this notation, the 'associative law' (1) becomes the equation $g^* \cdot f^* = (g^* \cdot f)^*$, and the unit laws (2) and (3) become $f^* \cdot \eta_x = f$ and $\eta_x^* = \mathrm{id}_{Mx}$. (A summary of the notation for category theory that is used in this paper appears in an appendix.)

The following definition generalizes this notion of monad by identifying for each object $x \in \mathscr{X}$ not just an object $Mx \in \mathscr{X}$ in the original category, but also an object $Fx \in \mathscr{A}$ in the richer category.

Definition 1
If $\mathscr{A}$ and $\mathscr{X}$ are categories with a functor $U : \mathscr{A} \to \mathscr{X}$, then we define an *$\mathscr{A}$-monad on $\mathscr{X}$* to be a triple $\langle F, \_^\sharp, \eta \rangle$ that assigns

- an object $Fx \in \mathscr{A}$ to each $x \in \mathscr{X}$,
- an arrow $f^\sharp : Fx \to Fy$ in $\mathscr{A}$ to each arrow $f : x \to UFy$ in $\mathscr{X}$ and
- an arrow $\eta_x : x \to UFx$ to each object $x \in \mathscr{X}$

https://doi.org/10.1017/S0956796809007321 Published online by Cambridge University Press

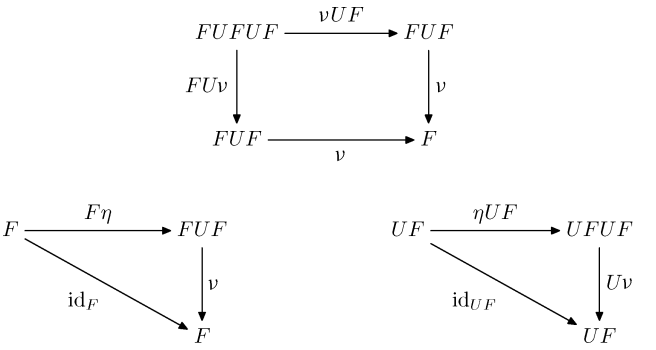


Fig. 3. Laws for an $\mathscr{A}$-monad.

in such a way that if $f : x \to UFy$ and $g : y \to UFz$, then

$$g^\sharp \cdot f^\sharp = (Ug^\sharp \cdot f)^\sharp, \tag{10}$$

$$\eta_x^\sharp = \mathrm{id}_{Fx}, \tag{11}$$

$$Uf^\sharp \cdot \eta_x = f. \tag{12}$$

This completes the definition.

The concept of an $\mathscr{A}$-monad generalizes that of a monad on $\mathscr{X}$ in two ways. Firstly, each $\mathscr{A}$-monad induces an ordinary monad $\langle M, \_^*, \eta\rangle$ on $\mathscr{X}$, where $Mx = UFx$, and if $f : x \to My$ then $f^* = Uf^\sharp$. It is a matter of routine to verify that the requisite equations hold in this monad. Secondly, if we set $\mathscr{A} = \mathscr{X}$ and set U to be the identity functor $\mathrm{Id}_{\mathscr{X}} : \mathscr{X} \to \mathscr{X}$, then the concept of an $\mathscr{A}$-monad reduces to that of an ordinary monad on $\mathscr{X}$.

Just as ordinary monads can be defined equationally in terms of functors and natural transformations, so can the concept of an $\mathscr{A}$-monad be presented equationally. Given the data in the definition above, we can extend F to a functor by defining its action on an arrow $f : x \to y$ to be

$$Ff = (\eta_y \cdot f)^\sharp : Fx \to Fy.$$

The family of arrows η_x then becomes a natural transformation $\eta : \mathrm{Id}_{\mathscr{X}} \dot{\to} UF$. Moreover, we can define a natural transformation $\nu : FUF \dot{\to} F$ by $\nu_x = \mathrm{id}^\sharp_{UFx}$, and the three equations above then give the following three equations relating η and ν (see Figure 3):

$$\nu \cdot FU\nu = \nu \cdot \nu UF, \tag{13}$$

$$\nu \cdot F\eta = \mathrm{id}_F, \tag{14}$$

$$U\nu \cdot \eta UF = \mathrm{id}_{UF}. \tag{15}$$

These are slightly adjusted copies of the equations in the equational definition of an ordinary monad.

Conversely, a translation the other way is obtained by setting $f^\sharp = \nu_y \cdot Ff$ for each $f : x \to UFy$ in $\mathscr{X}$. The laws defining an $\mathscr{A}$-monad then follow from the three equations just stated; in particular, the associative law (10) follows because expanding the definition of $\_^\sharp$ and using the naturality of ν gives

$$g^\sharp \cdot f^\sharp = \nu_z \cdot Fg \cdot \nu_y \cdot Ff = \nu_z \cdot \nu_{UFz} \cdot FUFg \cdot Ff,$$

https://doi.org/10.1017/S0956796809007321 Published online by Cambridge University Press

while expanding the definition and exploiting the fact that F and U are functors gives

$$(Ug^\sharp \cdot f)^\sharp = v_z \cdot F(U(v_z \cdot Fg) \cdot f) = v_z \cdot FUv_z \cdot FUFg \cdot Ff.$$

The two expressions on the right are equal according to Equation (13).

An $\mathscr{A}$-monad is more than a monad on $\mathscr{X}$, but it is less than an adjunction between $\mathscr{X}$ and $\mathscr{A}$. In particular, a forgetful functor $U : \mathscr{A} \to \mathscr{X}$ can have (up to isomorphism) at most one left adjoint F, but as we shall see, there may be multiple non-isomorphic $\mathscr{A}$-monads on $\mathscr{X}$. The key difference between the data given above and those that determine an adjunction is that the $\_^\sharp$ operation is restricted to arrows $f : x \to UFy$, and does not apply to all arrows $f : x \to Ua$ for any $a \in \mathscr{A}$.

We define morphisms of $\mathscr{A}$-monads as follows:

Definition 2
If $\langle F, \_^\sharp, \eta\rangle$ and $\langle F', \_^\flat, \eta'\rangle$ are $\mathscr{A}$-monads on $\mathscr{X}$, then a morphism θ between them is a family of arrows $\theta_x : Fx \to F'x$ in $\mathscr{A}$ for each $x \in \mathscr{X}$, such that whenever $f : x \to UFy$ in $\mathscr{X}$,

$$U\theta_x \cdot \eta_x = \eta'_x, \tag{16}$$
$$\theta_y \cdot f^\sharp = (U\theta_y \cdot f)^\flat \cdot \theta_x. \tag{17}$$

This completes the definition.

Equivalently, in terms of functors and natural transformations, we may define a morphism of $\mathscr{A}$-monads to be a natural transformation $\theta : F \dot{\to} F'$ such that

$$U\theta \cdot \eta = \eta', \tag{18}$$
$$\theta \cdot v = v' \cdot (\theta \star U \star \theta), \tag{19}$$

where $v' : F'UF' \dot{\to} F'$ and $\star$ denotes horizontal composition of natural transformations, so that $\theta \star U \star \theta = \theta UF' \cdot FU\theta = F'U\theta \cdot \theta UF$. In either style, the obvious definitions of identity morphisms and composition of morphisms make the class of $\mathscr{A}$-monads on $\mathscr{X}$ into a category. If $\theta : F \dot{\to} F'$ is any morphism of $\mathscr{A}$-monads, then $U\theta : UF \dot{\to} UF'$ is a morphism of the underlying monads on $\mathscr{X}$.

For ordinary monads, it is well known that an adjunction between $\mathscr{X}$ and $\mathscr{A}$ creates a monad on $\mathscr{X}$, and it seems reasonable to expect that it should create an $\mathscr{A}$-monad also. In fact, this $\mathscr{A}$-monad is an initial object in the category.

Lemma 1
Let $\langle F, U, \eta, \epsilon\rangle : \mathscr{X} \rightharpoonup \mathscr{A}$ be an adjunction with unit $\eta : \mathrm{Id}_{\mathscr{X}} \dot{\to} UF$ and counit $\epsilon : FU \dot{\to} \mathrm{Id}_{\mathscr{A}}$, and let $v = \epsilon F$. Then $\langle F, \eta, v\rangle$ is an initial object in the category of $\mathscr{A}$-monads.

Proof
We first show that $\langle F, \eta, v\rangle$ is an $\mathscr{A}$-monad. For Equation (13), we have $\epsilon \cdot FU\epsilon = \epsilon \cdot \epsilon FU$ by naturality of ϵ. Post-multiplying by F gives

$$v \cdot FUv = \epsilon F \cdot FU\epsilon F = \epsilon F \cdot \epsilon FUF = v \cdot vUF.$$

Equations (14) and (15) are instances of the two triangle laws of the adjunction:

$$v \cdot F\eta = \epsilon F \cdot F\eta = \mathrm{id}_F$$

https://doi.org/10.1017/S0956796809007321 Published online by Cambridge University Press

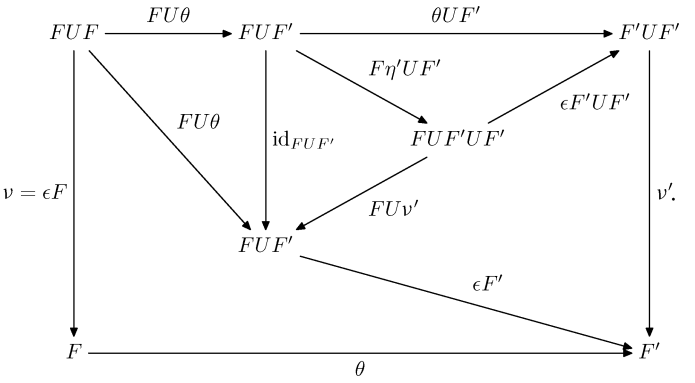
and (again post-multiplying by F)

$$U\nu \cdot \eta UF = U\epsilon F \cdot \eta UF = \mathrm{id}_{UF}.$$

Now suppose $\langle F', \eta', \nu' \rangle$ is any $\mathscr{A}$-monad, and define a morphism $\theta : F \dot{\to} F'$ by $\theta = \epsilon F' \cdot F\eta'$. This does give a morphism: Equation (18) holds because $UF\eta' \cdot \eta = \eta UF \cdot \eta'$ by naturality of η, and we can apply the triangle law $U\epsilon \cdot \eta U = \mathrm{id}_U$, post-multiplying it by F':

$$U\theta \cdot \eta = U\epsilon F' \cdot UF\eta' \cdot \eta = U\epsilon F' \cdot \eta UF' \cdot \eta' = \eta'.$$

For Equation (19), consider the following diagram of functors and natural transformations:



In this diagram, the top and right-hand sides form $\nu' \cdot (\theta \star U \star \theta)$, and the bottom and left-hand sides form $\theta \cdot \nu$. Both the (distorted) squares commute by naturality of ϵ. The top triangle is an instance of the definition of θ, and the central triangle is a version of Equation (15).

Lastly, suppose $\theta_1 : F \dot{\to} F'$ is another morphism. We can calculate as follows, using Equation (18) applied to θ_1, and then the naturality of ϵ and the other triangle law for the adjunction:

$$\theta = \epsilon F' \cdot F\eta' = \epsilon F' \cdot FU\theta_1 \cdot F\eta = \theta_1 \cdot \epsilon F \cdot F\eta = \theta_1.$$

So θ is a unique morphism from F to F'. □

While an adjunction between $\mathscr{X}$ and the whole of the category $\mathscr{A}$ gives an initial object in the category of $\mathscr{A}$-monads, other $\mathscr{A}$-monads can be obtained from adjunctions between $\mathscr{X}$ and subcategories of $\mathscr{A}$. If $\langle F', U', \eta', \epsilon' \rangle : \mathscr{X} \rightharpoonup \mathscr{B}$ for some subcategory $\mathscr{B} \subseteq \mathscr{A}$ (where $U' = U|_{\mathscr{B}}$ is the restriction of U), then we can define $\nu' = \epsilon' F'$ and obtain an $\mathscr{A}$-monad $\langle F', \eta', \nu' \rangle$ as before. This result is interesting for us, because we can obtain subcategories of a category $\mathscr{A}$ of algebras by restricting attention to those algebras that satisfy additional equations.

5 Application to combinatorial search

The general picture of the category of $\mathscr{A}$-monads can now be applied to the particular case of search strategies. For the moment, we will simplify the discussion by restricting attention to finite search spaces and ignore the problem of modelling programs where the search space is infinite. This restriction will be removed in Section 6.

https://doi.org/10.1017/S0956796809007321 Published online by Cambridge University Press

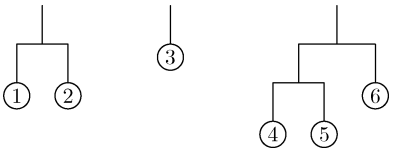


Fig. 4. A typical forest.

A search strategy is a monad equipped with the operations $\oplus$, *zero* and *wrap* and satisfying the equations given in Section 2. This makes it appropriate to take $\mathscr{A}$ to be the category of (Ω, E)-algebras and homomorphisms, where Ω is the set of operators $\{\oplus_2, zero_0, wrap_1\}$ with arities as shown, and E is the set of equations,

$$\{(x \oplus y) \oplus z = x \oplus (y \oplus z), zero \oplus x = x = x \oplus zero\}.$$

These equations correspond to laws (4), (5) and (6) from Section 2. Laws (1), (2) and (3) state that $\triangleright$ and *return* are the operations of a monad. Laws (7), (8) and (9) state that the function $(\triangleright f)$, for any function $f : \alpha \to M\,\beta$, is a homomorphism of algebras. Together, these observations amount to the fact that what we have called a bunch is exactly an $\mathscr{A}$-monad for this particular category $\mathscr{A}$.

A standard result states that there is an adjunction between the category of sets and the category of (Ω, E)-algebras that gives the free algebra generated by any set. See, for example, MacLane (1971, p. 120). Lemma 1 of Section 4 then tells us that the $\mathscr{A}$-monad containing these free algebras is an initial object in its category.

For our choice of Ω and E, this free algebra is the set of forests with leaves from the given set. A typical element where the leaves are integers is the forest

$$[Fork\ [Tip\ 1, Tip\ 2], Tip\ 3, Fork\ [Fork\ [Tip\ 4, Tip\ 5], Tip\ 6]],$$

shown in Figure 4. Writing w for *wrap* and $\{x\}$ for *return* x, this can be expressed in terms of the operations as

$$w(\{1\} \oplus \{2\}) \oplus \{3\} \oplus w(w(\{4\} \oplus \{5\}) \oplus \{6\}).$$

In this case, it should be clear that the type of forests, together with the operations defined in Section 2, represents exactly the set of expressions of this form, and that distinct expressions give distinct forests. The results of Section 4 then tell us that the type of forests can be made into an $\mathscr{A}$-monad, and it remains only to calculate a definition of the binding operator $\triangleright$. Any forest xm can be written in the form

$$xm = [t_1] +\!\!+ [t_2] +\!\!+ \cdots +\!\!+ [t_n],$$

and then we see that

$$xm \triangleright f = ([t_1] \triangleright f) +\!\!+ ([t_2] \triangleright f) +\!\!+ \cdots +\!\!+ ([t_n] \triangleright f).$$

Furthermore, we can calculate that

$$[Leaf\ x] \triangleright f = f\ x,$$

and

$$[Fork\ xm] \triangleright f = [Fork\ (xm \triangleright f)].$$

https://doi.org/10.1017/S0956796809007321 Published online by Cambridge University Press

In this way, we can derive the definition of $\triangleright$ shown in Section 2 from the algebraic laws that characterize a bunch, something we already did in Section 3 for the bunch of breadth-first search.

This calculation could have been carried out without knowing the results of the previous section or their application to bunches. But what these results guarantee is that the adjunction between $\mathscr{X}$ and $\mathscr{A}$ does indeed create an $\mathscr{A}$-monad, so that there does exist an operation $\triangleright$ that satisfies all the required laws. What we have just shown is that any operation that satisfies the laws must also satisfy the equations that make up a recursive definition of $\triangleright$; since this recursive definition describes a unique operation, we may be sure that this operation is the required binding operator for the monad.

We can now play a game like the one that leads to the 'Boom hierarchy' trees–lists–bags–sets of collection types: by adding equations to the algebraic specification, we obtain alternative models that identify previously distinct terms. For each set of equations E' that extends our original set E, we obtain a full subcategory of the category of (Ω, E)-algebras, containing just those algebras that satisfy the equations E', together with an adjunction between sets and the subcategory. This gives an $\mathscr{A}$-monad on sets, and in particular a unique definition of the binding operator $\triangleright$ for the resulting notion of search.

For example, if we put $wrap\ x = x$, this reduces the example term to

$$\{1\} \oplus \{2\} \oplus \{3\} \oplus \{4\} \oplus \{5\} \oplus \{6\},$$

that is, to something that can be represented by the list $[1, 2, 3, 4, 5, 6]$, and we obtain the monad of depth-first search. Again, once we see that the expressions in this normal form are faithfully represented by lists, it follows that there is an $\mathscr{A}$-monad based on lists, and the definition of the binding operator can be calculated.

Alternatively, we may add the equation that *wrap* distributes over $\oplus$, reducing the example term to

$$w(\{1\}) \oplus w(\{2\}) \oplus \{3\} \oplus w^2(\{4\}) \oplus w^2(\{5\}) \oplus w(\{6\}),$$

where we write, e.g., $w^2(x)$ for $w(w(x))$. With these equations, any finite term can be reduced to a normal form that is a sum of terms, each of the form $w^n(\{x\})$; such a normal form reveals the left-to-right order of the solutions and the depth in the tree at which each one appears, without preserving the branching structure that links the solutions to roots in the forest. Thus it corresponds to the bunch of depth-bounded search.

Adding the further equation that *wrap* is commutative lets us group together the terms $w^n(\{x\})$ that have the same depth n, and reduces our example term to

$$\{3\} \oplus w(\{1\} \oplus \{2\} \oplus \{6\}) \oplus w^2(\{4\} \oplus \{5\}).$$

In general, the normal form is a sum of terms $w^n(s)$ for distinct n, where s is itself a sum of singletons in an arbitrary order. This corresponds closely to breadth-first search, where we may discover the bag of solutions that occur at each depth in the tree.

We remarked earlier that it was impossible to get an associative binding operator for breadth-first search without making $\oplus$ commutative, and so treating each level of the result as a bag rather than a list. Here we see that imposing commutativity has the positive effect of simplifying the normal forms and leading us directly to the stream-of-bags representation.

https://doi.org/10.1017/S0956796809007321 Published online by Cambridge University Press

If we accept that the type containing lists of bags is a proper representation of the normal forms just described, then the calculation in Section 2 that led to a definition of $\triangleright$ is correct, and the operator it defines satisfies all the required laws, including the associative law (1). This is reassuring, since proving the associative law directly is rather difficult (see Spivey 2000).

In addition to the notion of an $\mathscr{A}$-monad, our categorical approach points us to the notion of a morphism of $\mathscr{A}$-monads, a parametric system of homomorphisms of algebras that is also a morphism of monads. For our initial object, there is such a unique morphism to any other bunch M, and we can easily derive a definition of this morphism as a functional program. It gives the value in any bunch that results from searching a forest:

$$\begin{aligned}
&\mathit{search} :: \mathit{Forest}\ \alpha \to M\alpha \\
&\mathit{search}\ [\,] = \mathit{zero} \\
&\mathit{search}\ (\mathit{Leaf}\ x : xm) = \mathit{return}\ x \oplus \mathit{search}\ xm \\
&\mathit{search}\ (\mathit{Fork}\ ym : xm) = \mathit{wrap}\ (\mathit{search}\ ym) \oplus \mathit{search}\ xm.
\end{aligned}$$

The fact that *search* is a bunch morphism reassures us that for any bunch M, the same results are obtained by running a program that builds a forest and then searching it as are obtained by running the same program directly in M.

Gibbons and Jones (1998) have shown that the familiar implementations of depth-first and breadth-first search that use a stack and a queue respectively can be derived from this definition of *search* by recasting it as an *unfold*.

6 Infinite search spaces

The discussion of the previous section indicates that, at least for finite searches, all compositional search strategies can be made part of a single algebraic framework. It would be nice to do the same for infinite searches, and preferable to do it in a way that reflects the fairness of strategies like breadth-first search. One possible approach is based on the theory of metric spaces.

Let us define a *fair* bunch to be a bunch M such that type $M\alpha$ is also a complete bounded metric space under some metric d, subject to certain restrictions. For the theory of metric spaces see, for example, Sutherland (1975). Our first example will be the bunch of forests, where we now allow infinite trees, provided that they remain finitely branching. If f_1 and f_2 are forests, we can define $d(f_1, f_2) = 2^{-n}$, where n is the first level at which f_1 and f_2 differ, or $d(f_1, f_2) = 0$ if $f_1 = f_2$. More precisely, we can define a function

$$\mathit{prune} :: \mathit{Integer} \to \mathit{Forest}\ \alpha \to \mathit{Forest}\ \alpha$$

that cuts off all branches in a forest below a specified depth; we then look for the smallest n such that $\mathit{prune}\ n\ f_1 \neq \mathit{prune}\ n\ f_2$.

The operations of the bunch are related to this metric in a number of ways:

1. The function *wrap* is a contraction, in the sense that

$$d(\mathit{wrap}\ f_1, \mathit{wrap}\ f_2) \leqslant \rho d(f_1, f_2)$$

for some $\rho < 1$; actually, we can take $\rho = 1/2$.

https://doi.org/10.1017/S0956796809007321 Published online by Cambridge University Press

2. The operation $\oplus$ satisfies $d(f_1 \oplus f_2, f_1' \oplus f_2') \leqslant \max(d(f_1, f_1'), d(f_2, f_2'))$.
3. The finite elements of $M\alpha$ – those that are the value of some finite term in the operations – are dense in $M\alpha$, so that every element of $M\alpha$ is the limit of some sequence of finite elements.

The first two of these properties are sufficient to guarantee that any *guarded* recursive definition has a unique solution. By a guarded recursion, I mean an equation of the form $t = F(t)$, where $F(x)$ is defined by a term where each occurrence of x is 'guarded' by at least one application of *wrap*. Standard techniques allow this result to be extended to recursive definitions of functions of the form $f(x) = F(f(g(x)))$, where F is guarded. The third property ensures that any continuous function from $M\alpha$ to some other complete metric space is uniquely determined by its behaviour on the finite elements.

Of the other bunches mentioned earlier, we cannot hope to make a fair bunch from the monad of depth-first search, because there the *wrap* operation is the identity function, and that cannot be a contraction. The lack of a general result about the existence of free continuous algebras of this kind means that we must also treat the other bunches one at a time.

For breadth-first search, we can define the distance between two streams of bags m_1 and m_2 to be $d(m_1, m_2) = 2^{-n}$, where n is smallest such that $take\, n\, m_1 \neq take\, n\, m_2$. Similarly, for depth-bounded search we may instead define $d(p_1, p_2) = 2^{-n}$, where n is smallest such that $p_1\, n \neq p_2\, n$.

It is not appropriate to develop the theory here at length, since it reveals no more structure than we found in the finite case, so I will just give a very brief outline of it. The appropriate notion of morphism for fair bunches is a family of uniformly continuous functions that is also a morphism of the underlying bunches. In fact, all the functions we need to consider in the existing examples are actually Lipschitz-continuous.

Using uniform continuity ensures that the image of a Cauchy sequence is again a Cauchy sequence, and allows us to extend functions defined on finite elements to cover infinite elements too, in a way that is unique because we assume that the finite elements are dense. This allows us to extend the definitions of the operations from finite to infinite elements in a way that preserves the algebraic laws, and gives us a unique morphism from the fair bunch of forests to any other fair bunch.

7 Conclusion

We have seen that in functional programming, several different strategies for combinatorial search can be made part of the same algebraic framework, one that emphasizes the compositional nature of the strategies. This algebraic framework adds one operation, *wrap*, to the standard *MonadPlus* type-class of Haskell, in effect making it possible for a computation to signal that it has produced all the results with a specified cost. We have shown, at least for finite search spaces, that these different search strategies all arise by adding various equations to an algebraic theory that describes the set of outcomes of the search, and we have suggested a way in which the theory could be extended to cover infinite search spaces also.

The search strategies we have covered are all oblivious, in the sense that they depend only on the depth of solutions in the tree of choices, and are not guided by any heuristic

https://doi.org/10.1017/S0956796809007321 Published online by Cambridge University Press

estimate of distance from a solution. It would be interesting to see whether the present framework could be extended to include a (real-valued) notion of distance, so that strategies such as best-first and A^* search could also be covered.

Many others have written about implementations of combinatorial search in higher-order functional programming, and it is appropriate to mention some recent related work here. Much of this work focusses on implementations of depth-first search. Hinze (2001) has shown that an implementation of depth-first search using success and failure continuations can be more efficient in Haskell than one based on lazy streams. He also considers the algebraic laws that are satisfied by our combinators and by others that correspond to Prolog's control structures, including cut. Kiselyov *et al.* (2005) extend this work to show how fair interleaving can also be implemented, though with the failure of associativity that we noted earlier. In a strikingly elegant paper, Wand and Vaillancourt (2001) use operational semantics to show the equivalence of the continuation-based and stream-based models of depth-first search.

Acknowledgments

The author is happy to acknowledge that part of this work was carried out while he held a visiting chair at the University of New South Wales, and thanks to Carroll Morgan for his generous hospitality and helpful suggestions. Thanks are also due to Paul Blain Levy for pointing out an error in an earlier version of this paper.

References

Bird, R. S. (1987) An introduction to the theory of lists. In *Calculi of Discrete Design*, Broy, M. (ed.), NATO ASI Series F, vol. 36. Springer, pp. 5–42.

Gibbons, J. & Jones, G. (1998) The under-appreciated unfold. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming, ICFP '98 (Baltimore, MD, Sept. 1998)*. ACM Press, pp. 273–279.

Hinze, R. (2001) Prolog's control constructs in a functional setting – axioms and implementation, *Int. J. Found. Comput. Sci.*, **12** (2): 125–170.

Kiselyov, O., Shan, C.-C., Friedman, D. P. & Sabry, A. (2005) Backtracking, interleaving and terminating monad transformers, In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP '05 (Tallinn, Sept. 2005)*. ACM Press, pp. 192–203.

MacLane, S. (1971) *Categories for the Working Mathematician*, Graduate Texts in Mathematics, vol. 5. Springer.

Meertens, L. G. L. T. (1986) Algorithmics – towards programming as a mathematical activity. In *Proceedings of the CWI Symposium on Mathematics and Computer Science*, de Bakker, J., Hazewinkel, M. & Lenstra, J. K. (eds), CWI Monographs, vol. 1. CWI, pp. 289–334.

Russell, S. J. & Norvig, P. (2003) *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice Hall.

Spivey, J. M. (2000) Combinators for breadth-first search, *J. Funct. Program.*, **10** (4): 397–408.

Spivey, J. M. & Seres, S. (2003) Combinators for logic programming. In *The Fun of Programming*, Gibbons, J. & de Moor, O. (eds), Cornerstones of Computing. Palgrave MacMillan, pp. 177–200.

Sutherland, W. A. (1975) *Introduction to Metric and Topological Spaces*. Oxford University Press.

Wand, M. & Vaillancourt, D. (2001) Relating models of backtracking. In *Proceedings of the 9th International Conference on Functional Programming, ICFP '04 (Snowbird, UT, Sept. 2004)*. ACM Press, pp. 54–65.

Notation

We use curly letters $\mathscr{X}$, $\mathscr{Y}$, $\mathscr{Z}$, $\mathscr{A}$ for categories, and uppercase Roman letters for functors $F : \mathscr{X} \to \mathscr{Y}$ between them. The identity functor on $\mathscr{X}$ is written as $\mathrm{Id}_{\mathscr{X}} : \mathscr{X} \to \mathscr{X}$.

A natural transformation $\theta : F \dot{\to} F'$ from a functor F to a functor F' (with $F, F' : \mathscr{X} \to \mathscr{Y}$) is a family of arrows $\theta_x : Fx \to F'x$ such that for each arrow $f : x \to x'$ in $\mathscr{X}$, the following diagram commutes:

$$\begin{array}{ccc} Fx & \xrightarrow{Ff} & Fx' \\ {\scriptstyle\theta_x}\downarrow & & \downarrow{\scriptstyle\theta_{x'}}. \\ F'x & \xrightarrow[F'f]{} & F'x' \end{array}$$

In other words, the equation $Gf \cdot \theta_x = \theta'_x \cdot Ff$ holds.

We use an infix dot for composition of arrows $g \cdot f$, and also for 'vertical' composition of natural transformations. If $\theta : F \dot{\to} F'$ and $\phi : F' \dot{\to} F''$, then $\phi \cdot \theta : F \dot{\to} F''$ is defined by $(\phi \cdot \theta)_x = \phi_x \cdot \theta_x$. This composition, together with the obvious definition of the identity transformation id_F from a functor F to itself, makes the collection of functors $\mathscr{X} \to \mathscr{Y}$ and the natural transformations between them into a category.

Composition of functors with functors and with natural transformations is denoted by juxtaposition. As examples of the latter, if $\theta : F \dot{\to} F'$ and $\phi : G \dot{\to} G'$ for functors $F, F' : \mathscr{X} \to \mathscr{Y}$ and $G, G' : \mathscr{Z} \to \mathscr{X}$, then $\theta G : FG \dot{\to} FG'$ and $F\phi : FG \dot{\to} FG'$ are defined by $(\theta G)_z = \theta_{Gx}$ and $(F\phi)_z = F(\phi_z)$.

With this set-up, if the arrow $f : x \to x'$ in the diagram above is in fact a component $\phi_z : Gz \to G'z$ of ϕ, then the diagram can be redrawn as a diagram of functors and natural transformations:

$$\begin{array}{ccc} FG & \xrightarrow{F\phi} & FG' \\ {\scriptstyle\theta G}\downarrow & & \downarrow{\scriptstyle\theta G'}. \\ F'G & \xrightarrow[F'\phi]{} & F'G' \end{array}$$

We say that this diagram, like the one above, commutes 'by naturality of θ', and we use the notation $\theta \star \phi = \theta G' \cdot F\phi = F'\phi \cdot \theta G$ for the diagonal of the square, the *horizontal composition* of θ and ϕ. This composition is associative, and as a special case, if $\theta : F \dot{\to} F'$ for $F, F' : \mathscr{X} \to \mathscr{A}$ and $U : \mathscr{A} \to \mathscr{X}$, we write $\theta \star U \star \theta$ for the natural transformation $\phi = \theta \star \mathrm{id}_U \star \theta : FUF \to F'UF'$ with $\phi_x = \theta_{UF'x} \cdot FU\theta_x = F'U\theta_x \cdot \theta_{UFx}$.

Following MacLane (1971), we use the notation $\langle F, U, \eta, \epsilon \rangle : \mathscr{X} \rightharpoonup \mathscr{A}$ for an adjunction. This consists of a pair of functors $F : \mathscr{X} \to \mathscr{A}$ and $U : \mathscr{A} \to \mathscr{X}$ with natural transformations $\eta : \mathrm{Id}_{\mathscr{X}} \dot{\to} UF$ and $\epsilon : FU \dot{\to} \mathrm{Id}_{\mathscr{A}}$, such that the triangle laws $\epsilon F \cdot F\eta = \mathrm{id}_F$ and $U\epsilon \cdot \eta U = \mathrm{id}_U$ hold. We call η the *unit* and ϵ the *counit* of the adjunction, and say that F is a *left adjoint* of U, and U is a *right adjoint* of F.

https://doi.org/10.1017/S0956796809007321 Published online by Cambridge University Press