

Algebras for Combinatorial Search

Mike Spivey

Oxford University Computing Laboratory

Visiting Professor, CSE@UNSW

Many answers – or none

- Programs may not return just a single answer (databases, AI).
- Traditional technique: replace a function $f : X \rightarrow Y$ with one that returns a ‘list of successes,’ $f : X \rightarrow \text{Stream } Y$.
- Rely on laziness to avoid large transient memory demand.

Example: factorization

This function splits its argument into factors in all possible ways:

$$\mathit{factor}_1 :: \mathit{Int} \rightarrow \mathit{Stream}(\mathit{Int}, \mathit{Int})$$
$$\mathit{factor}_1 n =$$
$$[(a, b) \mid a \leftarrow [1 \dots], b \leftarrow [1 \dots], a * b \equiv n]$$

Let's try it!

Fixing the problem

We could explicitly visit the pairs (a, b) in a fair order:

$$\begin{aligned} \text{factor}_2 n = \\ [(a, b) \mid (a, b) \leftarrow \text{diagprod} [1 \dots] [1 \dots], a * b \equiv n] \end{aligned}$$

But it would be nice if Haskell had a fair comprehension.¹ Can we add one?

¹ After all Miranda² had one years ago.

² Another footnote needed here; I forget why.

Using *do* notation

The original:

```
factor1 n =  
  do a ← [1 ..]; b ← [1 ..]; test (a * b ≡ n); return (a, b)
```

Improved version:

```
factor2 n =  
  do a ← Diag [1 ..]; b ← Diag [1 ..];  
      test (a * b ≡ n); return (a, b)
```

For this we need to define a new monad *Diag*.

The *Diag* monad

newtype *Diag* α = *Diag* (*Stream* α)

unDiag (*Diag* xs) = xs

instance *Monad* *Diag* **where**

return x = *Diag* [x]

(*Diag* xs) \triangleright f =

Diag (*concat* (*diag* (*map* (*unDiag* \cdot f) xs)))

for a suitable function

diag :: *Stream* (*Stream* α) \rightarrow *Stream* [α]

Let's try it!

What went wrong?

That **do** expression is (according to Haskell) equivalent to

```
do  $a \leftarrow \text{Diag } [1 \dots];$   
    (do  $b \leftarrow \text{Diag } [1 \dots];$   
       $\text{test } (a * b \equiv n); \text{return } (a, b)$ )
```

And the inner **do** expression searches for a b that goes with a fixed a , chosen by the outer **do** expression.

We can do better by forcibly associating the expression the other way, but that's just what we wanted to avoid.

So we end up with ...

diag [(1, 24) : \perp , (2, 12) : \perp , (3, 8) : \perp , (4, 6) : \perp , \perp , ...]

- In a sequential language (lazy or not), once we have decided to ask for the head of a list, we must wait for it – perhaps forever.
- Diagonalization doesn't help with programs that can compute forever without returning any answers.

Algebra matters

This bug points to the fact that the *Diag* ‘monad’ doesn’t obey the *associative law* that all monads should:

$$(xs \triangleright f) \triangleright g = xs \triangleright (\lambda x \rightarrow f x \triangleright g)$$

If it did, then both ways of writing the expression would be equivalent.

[A paper in this year’s ICFP commits this error.]

Breadth-first search

We need a way for a computation to signal that it is working hard, but not finding any answers.

newtype *Breadth* $\alpha = \text{Breadth} (\text{Stream} [\alpha])$

The idea is that in *Breadth* [$\chi s_0, \chi s_1, \chi s_2, \dots$], each (finite) list χs_i contains the answers that can be computed with cost i , for some suitable notion of cost.

The binding operator \triangleright is defined (using *diag*) so that in $\chi m \triangleright f$ it collects together results with the same total cost – computing result x from χm plus computing result y from $f x$.

Using the *Breadth* monad

$factor_3 :: Int \rightarrow Breadth (Int, Int)$

$factor_3 n =$

do $a \leftarrow choose [1 ..]; b \leftarrow choose [1 ..];$
 $test (a * b \equiv n); return (a, b)$

Let's try it!

Building collections

Each computation returns a collection of results.
The collection type is a monad:

```
class Monad m where  
  return ::  $\alpha \rightarrow m \alpha$   
  ( $\triangleright$ ) ::  $m \alpha \rightarrow (\alpha \rightarrow m \beta) \rightarrow m \beta$ 
```

But it also supports three more operations:

```
class Monad m => Bunch m where  
  zero ::  $m \alpha$   
  ( $\oplus$ ) ::  $m \alpha \rightarrow m \alpha \rightarrow m \alpha$   
  wrap ::  $m \alpha \rightarrow m \alpha$ 
```

(cf. the *MonadPlus* class of Haskell.)

Those three operations in detail

- *zero* – the empty collection

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

- \oplus – union of collections

$$\begin{aligned} (\mathit{Breadth} \ xm) \oplus (\mathit{Breadth} \ ym) = \\ \mathit{Breadth} (\mathit{longZipWith} \ (++) \ xm \ ym) \end{aligned}$$

- *wrap* – counting the cost

$$\mathit{wrap} (\mathit{Breadth} \ xm) = \mathit{Breadth} ([] : xm)$$

Using *wrap* for guarded recursion

The idea is that *wrap xs* contains the same elements as *xs*, but each with a cost that's \$1 greater than in *xs*.

$choose :: Bunch\ m \Rightarrow [\alpha] \rightarrow m\ \alpha$

$choose\ [] = zero$

$choose\ (x : xs) = return\ x \oplus wrap\ (choose\ xs)$

The aim is to approach infinite searches by making $m\ \alpha$ into a complete metric space with *wrap* a contraction.

What about the laws?

We expect that for any bunch type-constructor m :

- $\langle m, \text{return}, (\triangleright) \rangle$ form a monad.
- $\langle m \ \alpha, (\oplus), \text{zero} \rangle$ form a monoid for each α .
- Distributive laws:

$$\text{zero} \triangleright f = \text{zero}$$

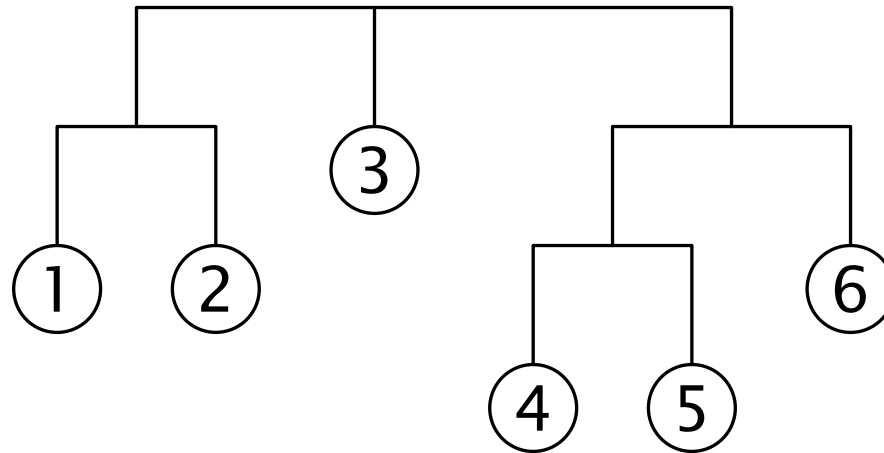
$$(xm \oplus ym) \triangleright f = (xm \triangleright f) \oplus (ym \triangleright f)$$

$$(\text{wrap } xm) \triangleright f = \text{wrap } (xm \triangleright f)$$

This is the logic of Prolog programming.

An algebraic specification

Like all good algebraic specifications, it has an initial model. What is it? Trees! (Or rather, forests).



wrap is the operation that takes a forest and makes it into the branches of a single tree, one level higher.

Adding more equations

What happens if we add more equations? (cf. the “Böhm hierarchy” trees–lists–bags–sets.)

Adding $wrap\ xs = xs$ turns

$$w ([1] \oplus [2]) \oplus [3] \oplus w (w ([4] \oplus [5]) \oplus [6])$$

into

$$[1] \oplus [2] \oplus [3] \oplus [4] \oplus [5] \oplus [6]$$

We get back simple lists and depth-first search.

Recovering breadth-first search

Alternatively, specify that *wrap* distributes over \oplus , and \oplus is commutative; then

$$w([1] \oplus [2]) \oplus [3] \oplus w(w([4] \oplus [5]) \oplus [6])$$

turns into

$$[3] \oplus w([1] \oplus [2] \oplus [6]) \oplus w^2([4] \oplus [5]).$$

The normal form is ‘polynomials in *wrap*’, and we get an ADT that is implemented as *Breadth*.

A bit of abstract nonsense

Definition Given categories \mathcal{X} and \mathcal{A} with $U : \mathcal{A} \rightarrow \mathcal{X}$, an \mathcal{A} -*monad* on \mathcal{X} is a triple $\langle F, -^\#, \eta \rangle$ that assigns:

- an object $Fx \in \mathcal{A}$ to each $x \in \mathcal{X}$,
 - an arrow $f^\# : Fx \rightarrow Fy$ in \mathcal{A} to each arrow $f : x \rightarrow UBy$ in \mathcal{X} ,
 - an arrow $\eta_x : x \rightarrow UFx$ to each object $x \in \mathcal{X}$,
- such that ...

Comparing with monads

For comparison, a monad $\langle T, -^*, \eta \rangle$ assigns

- an object $Tx \in \mathcal{X}$ to each $x \in \mathcal{X}$,
- an arrow $f^* : Tx \rightarrow Ty$ to each arrow $f : x \rightarrow y$,
- an arrow $\eta_x : x \rightarrow Tx$ to each object $x \in \mathcal{X}$.

So an \mathcal{A} -monad is a bit more than a monad on \mathcal{X} , and a bit less than an adjunction between \mathcal{X} and \mathcal{A} .

Equationally,

Natural transformations $\nu : FUF \xrightarrow{\bullet} F$ and $\eta : Id_X \xrightarrow{\bullet} UF$ such that ...

$$\begin{array}{ccc}
 FUFUF & \xrightarrow{\nu UF} & FUF \\
 \downarrow UF\nu & & \downarrow \nu \\
 FUF & \xrightarrow{\nu} & F
 \end{array}$$

$$\begin{array}{ccc}
 F & \xrightarrow{F\eta} & FUF \\
 & \searrow id_F & \downarrow \nu \\
 & & F
 \end{array}$$

$$\begin{array}{ccc}
 UF & \xrightarrow{\eta UF} & UFUF \\
 & \searrow id_{UF} & \downarrow U\nu \\
 & & UF
 \end{array}$$

The category of \mathcal{A} -monads

Fixing \mathcal{X} , \mathcal{A} and U , we can form a category of \mathcal{A} -monads by defining a morphism

$\theta : \langle F, \nu, \eta \rangle \rightarrow \langle F', \nu', \eta' \rangle$ to be a natural transformation $\theta : F \xrightarrow{\bullet} F'$ such that

$$U\theta \cdot \eta = \eta',$$

and

$$\theta \cdot \nu = \nu' \cdot (\theta \star U \star \theta).$$

Two lemmas

1. An adjunction between \mathcal{X} and \mathcal{A} gives rise to an initial object in the category of \mathcal{A} -monads.
2. Every \mathcal{A} -monad arises from an adjunction between \mathcal{X} and some full subcategory of \mathcal{A} .

In our context, each refinement of the equational theory has a search algebra as its initial model.

The unique arrow from the initial object shows how to build a tree and search it.

Depth-bounded search

- Corresponds to making *wrap* distribute over \oplus , but not making \oplus commutative.
- Implemented as functions

$$Int \rightarrow [(\alpha, Int)]$$

that take a cost budget, and return a list of affordable results, each paired with the balance remaining.

Implementing depth-bounded search

newtype *DBS* $\alpha = \text{DBS } (\text{Int} \rightarrow [(\alpha, \text{Int})])$

runDBS (*DBS g*) = *g*

instance *Monad* *DBS* **where**

return $x = \text{DBS } (\lambda n \rightarrow [(x, n)])$

$(\text{DBS } g) \triangleright f =$

$\text{DBS } (\lambda n \rightarrow \text{concat } [\text{runDBS } (f \ x) \ r \mid (x, r) \leftarrow g \ n])$

instance *Bunch* *DBS* **where**

zero = $\text{DBS } (\lambda n \rightarrow [])$

alt ($\text{DBS } g_1$) ($\text{DBS } g_2$) = $\text{DBS } (\lambda n \rightarrow g_1 \ n \ ++ \ g_2 \ n)$

wrap ($\text{DBS } g$) = $\text{DBS } (\lambda n \rightarrow \mathbf{if} \ n \equiv 0 \ \mathbf{then} \ [] \ \mathbf{else} \ g \ (n-1))$

I haven't told you about

- Representing Prolog's substitutions and unification.
- Implementing search with stacks and queues.
- Infinite search spaces.

Conclusions

This approach offers:

- A *compositional* semantics for the procedural reading of logic programs, *leading to ...*
- Transformation rules for logic programs using rewriting and UFP.
- A uniform treatment of unification and constraint programming.