Chapter 5
Calculating Functional Programs

Jeremy Gibbons

Abstract. Functional programs are merely equations; they may be manipulated by straightforward equational reasoning. In particular, one can use this style of reasoning to *calculate* programs, in the same way that one calculates numeric values in arithmetic. Many useful theorems for such reasoning derive from an *algebraic* view of programs, built around datatypes and their operations. Traditional algebraic methods concentrate on initial algebras, constructors, and values; dual *co-algebraic* methods concentrate on final co-algebras, destructors, and processes. Both methods are elegant and powerful; they deserve to be combined.

1 Introduction

These lecture notes on algebraic and coalgebraic methods for calculating functional programs derive from a series of lectures given at the *Summer School on Algebraic and Coalgebraic Methods in the Mathematics of Program Construction* in Oxford in April 2000. They are based on an earlier series of lectures given at the Estonian Winter School on Computer Science in Palmse, Estonia, in 1999.

1.1 Why calculate programs?

Over the past few decades there has been a phenomenal growth in the use of computers. Alongside this growth, concern has naturally grown over the correctness of computer systems, for example as regards human safety, financial security, and system development budgets. Problems in developing software and errors in the final product have serious consequences; such problems are the norm rather than the exception. There is clearly a need for more reliable methods of program construction than the traditional ad hoc methods in use today. What is needed is a *science* of programming, instead of today's *craft* (or perhaps black art). As Jeremy Gunawardena points out [15], computation is inherently more mathematical than most engineering artifacts; hence, practising software engineers should be at least as familiar with the mathematical foundations of software engineering as other engineers are with the foundations of their own branches of engineering.

By 'mathematical foundations', we do not necessarily mean obscure aspects of theoretical computer science. Rather, we are referring to simple properties and laws of computer programs: equivalences between programming constructs, relationships between well-known algorithms, and so on. In particular, we are interested in *calculating* with programs, in the same way that we calculate with numeric quantities in algebra at school.

1.2 Functional programming

One particularly appropriate framework for program calculation is functional programming, simply because the absence of side-effects ensures *referential transparency* — all that matters of any expression is the value it denotes, not any other characteristic such as the method by which it computed, the time taken to evaluate it, the number of characters used to express it, and so on. Expressions in a functional programming language behave as they do in ordinary mathematics, in the sense that an expression in a given context may be replaced with a different expression yielding the same value, without changing its meaning in the surrounding context. This makes calculations much more straightforward.

Functional programming is programming with *expressions*, which denote values, as opposed to *statements*, which denote actions. A program consists of a collection of *equations* defining new functions. For example, here is a simple functional program:

```
square x = x * x
```

This program defines the function `square`. The fact that it is written as an equation implies that any occurrence of an expression `square x` is equivalent to the expression `x * x`, whatever the expression `x`.

1.3 Universal properties

Suppose one has to define a function satisfying a given specification. Two approaches to solving this problem spring to mind. One, the *explicit approach*, is to provide an implementation of the function. The other, the *implicit approach*, is to provide a property that completely characterizes the function. Such a property is known as a *universal property*. The implicit approach is less direct, and requires more machinery, but turns out to be more convenient for calculating with. Universal properties are a central theme of these lectures.

1.3.1 Example: fork

Given two functions $f :: \mathsf{A} \to \mathsf{B}$ (which from an A computes a B) and $g :: \mathsf{A} \to \mathsf{C}$ (which from an A computes a C), consider the problem of constructing a function of type $\mathsf{A} \to \mathsf{B} \times \mathsf{C}$ (which from an A computes both a B and a C). We will write this induced function '$\mathsf{fork}\ (f, g)$'. We will think of fork itself as a *higher-order* operator, taking functions to functions.

1.3.2 Solution using explicit approach

The explicit approach to constructing this function fork consists of providing an implementation

$$\mathsf{fork}\ (f, g)\ a = (f\ a, g\ a)$$

That is, applying the function $\mathsf{fork}\,(f, g)$ to the argument a yields the pair whose left component is $f\ a$ and whose right component is $g\ a$. Now the existence of a solution to the problem is 'obvious'. (Actually, the existence of solutions to equations like this is a central theme in semantics of functional programming, but that is beyond the scope of these lectures.) However, proofs of properties of the function can be rather laborious, as we show below.

1.3.3 Projections eliminate fork

We claim that

$$\begin{aligned} &\mathsf{exl} \circ \mathsf{fork}\ (f, g) = f \\ &\mathsf{exr} \circ \mathsf{fork}\ (f, g) = g \end{aligned}$$

where exl and exr are the pair *projections* or *destructors*, yielding the left and right components of a pair respectively. (Here, $\circ$ is function composition; $\mathsf{exl} \circ \mathsf{fork}\ (f, g)$ is the composition of the two functions exl and $\mathsf{fork}\ (f, g)$, so that

$$(\mathsf{exl} \circ \mathsf{fork}\ (f, g))\ a = \mathsf{exl}\ (\mathsf{fork}\ (f, g)\ a)$$

for any a.) The proof of the first property is as follows:

$$\begin{array}{ll} & (\mathsf{exl} \circ \mathsf{fork}\ (f, g))\ a \\ = & \quad \{\ \text{composition}\ \} \\ & \mathsf{exl}\ (\mathsf{fork}\ (f, g)\ a) \\ = & \quad \{\ \mathsf{fork}\ \} \\ & \mathsf{exl}\ (f\ a, g\ a) \\ = & \quad \{\ \mathsf{exl}\ \} \\ & f\ a \end{array}$$

and so $\mathsf{exl} \circ \mathsf{fork}\,(f, g) = f$ as required. The proof of the second property is similar.

1.3.4 Any pair-forming function is a fork

We claim that, for pair-forming h (that is, $h :: \mathsf{A} \to \mathsf{B} \times \mathsf{C}$),

$$\mathsf{fork}\ (\mathsf{exl} \circ h, \mathsf{exr} \circ h) = h$$

To prove this, assume an arbitrary a, and suppose that $h\ a = (b, c)$ for some particular b and c; then

$$\begin{array}{ll} & \mathsf{fork}\ (\mathsf{exl} \circ h, \mathsf{exr} \circ h)\ a \\ = & \quad \{\ \mathsf{fork},\ \text{composition}\ \} \\ & (\mathsf{exl}\ (h\ a), \mathsf{exr}\ (h\ a)) \\ = & \quad \{\ h\ \} \\ & (\mathsf{exl}\ (b, c), \mathsf{exr}\ (b, c)) \\ = & \quad \{\ \mathsf{exl},\ \mathsf{exr}\ \} \\ & (b, c) \\ = & \quad \{\ h\ \} \\ & h\ a \end{array}$$

as required.

1.3.5 Identity function is a fork

We claim that

$$\mathsf{fork}\,(\mathsf{exl},\mathsf{exr}) = \mathsf{id}$$

The proof:

$$\begin{array}{ll}
 & \mathsf{fork}\,(\mathsf{exl},\mathsf{exr})\,(a,b) \\
= & \quad \{\, \mathsf{fork} \,\} \\
 & (\mathsf{exl}\,(a,b),\mathsf{exr}\,(a,b)) \\
= & \quad \{\, \mathsf{exl},\ \mathsf{exr} \,\} \\
 & (a,b) \\
= & \quad \{\, \text{identity} \,\} \\
 & \mathsf{id}\,(a,b)
\end{array}$$

1.3.6 Solution using implicit approach

The implicit approach to constructing the function fork consists of observing that $\mathsf{fork}\,(f,g)$ is *uniquely determined* by the fact that it returns the pair with components given by f and g. That is, $\mathsf{fork}\,(f,g)$ is the unique solution of the equations

$$\begin{array}{l}
\mathsf{exl} \circ h = f \\
\mathsf{exr} \circ h = g
\end{array}$$

in the unknown h. Equivalently, we have the *universal property of fork*

$$h = \mathsf{fork}\,(f,g) \Leftrightarrow \mathsf{exl} \circ h = f \ \wedge\ \mathsf{exr} \circ h = g$$

It is perhaps not immediately obvious that the system of two equations above has a unique solution (we address this problem later). But, once we can justify the universal property, calculations with forks become much more straightforward, as we illustrate below.

1.3.7 Projections eliminate fork

For the claim

$$\begin{array}{l}
\mathsf{exl} \circ \mathsf{fork}\,(f,g) = f \\
\mathsf{exr} \circ \mathsf{fork}\,(f,g) = g
\end{array}$$

we have the proof

$$\begin{array}{ll}
 & \mathsf{exl} \circ \mathsf{fork}\,(f,g) = f \ \wedge\ \mathsf{exr} \circ \mathsf{fork}\,(f,g) = g \\
\Leftrightarrow & \quad \{\, \text{universal property, letting } h = \mathsf{fork}\,(f,g) \,\} \\
 & \mathsf{fork}\,(f,g) = \mathsf{fork}\,(f,g)
\end{array}$$

1.3.8 Any pair-forming function is a fork

For the claim that, for pair-forming h,

$$\mathsf{fork}\,(\mathsf{exl}\circ h, \mathsf{exr}\circ h) = h$$

we have the proof

$$\begin{array}{ll} & h = \mathsf{fork}\,(\mathsf{exl}\circ h, \mathsf{exr}\circ h) \\ \Leftrightarrow & \quad \{\text{universal property, letting } f = \mathsf{exl}\circ h \text{ and } g = \mathsf{exr}\circ h\} \\ & \mathsf{exl}\circ h = \mathsf{exl}\circ h \ \wedge\ \mathsf{exr}\circ h = \mathsf{exr}\circ h \end{array}$$

1.3.9 Identity function is a fork

For the claim that

$$\mathsf{fork}\,(\mathsf{exl}, \mathsf{exr}) = \mathsf{id}$$

we have the proof

$$\begin{array}{ll} & \mathsf{id} = \mathsf{fork}\,(\mathsf{exl}, \mathsf{exr}) \\ \Leftrightarrow & \quad \{\text{universal property, letting } f = \mathsf{exl} \text{ and } g = \mathsf{exr}\} \\ & \mathsf{exl}\circ \mathsf{id} = \mathsf{exl} \ \wedge\ \mathsf{exr}\circ \mathsf{id} = \mathsf{exr} \end{array}$$

The gain is even more impressive for recursive functions, where the explicit approach requires inductive proofs that the implicit approach avoids. We will see many examples of such gains throughout these lectures.

1.4 The categorical approach to datatypes

In these lectures we will be using category theory as an organizing principle. For our purposes, the use of category theory can be summarized in three slogans:

- *A model of computation is represented by a category.*
- *Types and programs in the model are represented by the objects and arrows of that category.*
- *A type constructor in the model is represented by a functor on that category.*

We will not rely on any deep results of category theory; we will only be using the theory to obtain a streamlined notation.

1.4.1 Definition of a category

A *category* $\mathcal{C}$ consists of a collection $\mathsf{Obj}(\mathcal{C})$ of *objects* and a collection $\mathsf{Arr}(\mathcal{C})$ of *arrows*, such that

- each arrow f in $\mathsf{Arr}(\mathcal{C})$ has a *source* $\mathsf{src}(f)$ and a *target* $\mathsf{tgt}(f)$, both objects in $\mathsf{Obj}(\mathcal{C})$ (we write '$f : \mathsf{src}(f) \to \mathsf{tgt}(f)$');
- for every object A in $\mathsf{Obj}(\mathcal{C})$ there is an *identity arrow* $\mathsf{id}_{\mathsf{A}} : \mathsf{A} \to \mathsf{A}$;
- arrows $g : \mathsf{A} \to \mathsf{B}$ and $f : \mathsf{B} \to \mathsf{C}$ compose to form an arrow $f \circ g : \mathsf{A} \to \mathsf{C}$;
- composition is associative: $f \circ (g \circ h) = (f \circ g) \circ h$;
- the appropriate identity arrows are units: for arrow $f : \mathsf{A} \to \mathsf{B}$, we have $f \circ \mathsf{id}_{\mathsf{A}} = f = \mathsf{id}_{\mathsf{B}} \circ f$.

1.4.2 An example category: $\mathcal{S}et$

The category $\mathcal{S}et$ of sets and total functions is defined as follows.

- The objects $\mathsf{Obj}(\mathcal{S}et)$ are sets of values, or *types*.
- The arrows $f : \mathsf{A} \to \mathsf{B}$ in $\mathsf{Arr}(\mathcal{S}et)$ are total functions equipped with domain A and range B.
- The identity arrows are the identity functions $\mathsf{id_A}\ a = a$.
- Composition of arrows is functional composition: $(f \circ g)\ a = f\ (g\ a)$.

For example, addition is an arrow from the object $\mathsf{Int} \times \mathsf{Int}$ (the set of pairs of integers) to the object Int (the set of integers).

1.4.3 Definition of a functor

An *(endo)-functor* F is an operation on the objects and arrows of a category:

- $\mathsf{F\ A}$ is an object of $\mathcal{C}$ when A is an object of $\mathcal{C}$;
- $\mathsf{F}\,f$ is an arrow of $\mathcal{C}$ when f is an arrow of $\mathcal{C}$.

which respects source and target:

$$\mathsf{F}\,f : \mathsf{F}\ (\mathsf{src}(f)) \to \mathsf{F}(\mathsf{tgt}(f))$$

respects composition:

$$\mathsf{F}\ (f \circ g) = \mathsf{F}\,f \circ \mathsf{F}\,g$$

and respects identities:

$$\mathsf{F}\ \mathsf{id_A} = \mathsf{id_{F\,A}}$$

1.4.4 An example functor in $\mathcal{S}et$: **Pair**

The $\mathcal{S}et$ functor Pair is defined as follows.

- On objects, $\mathsf{Pair\ A} = \{(a_1, a_2) \mid a_1 \in \mathsf{A}, a_2 \in \mathsf{A}\}$.
- On arrows, $(\mathsf{Pair}\,f)\ (a_1, a_2) = (f\ a_1, f\ a_2)$.

We should check that the properties are satisfied (Exercise 1.7.1):

- source and target: $\mathsf{Pair}\,f : \mathsf{Pair\ A} \to \mathsf{Pair\ B}$ when $f : \mathsf{A} \to \mathsf{B}$;
- composition: $\mathsf{Pair}\ (f \circ g) = \mathsf{Pair}\,f \circ \mathsf{Pair}\,g$;
- identities: $\mathsf{Pair}\ \mathsf{id_A} = \mathsf{id_{Pair\,A}}$.

1.4.5 More functors

See Exercise 1.7.2 for the proofs that the following are functors.

Identity functor: The simplest functor Id is defined by

$$\begin{aligned}\mathsf{Id\ A} &= \mathsf{A}\\ \mathsf{Id}\,f &= f\end{aligned}$$

Constant functor: The next most simple is the constant functor $\underline{\mathsf{B}}$ for object B, defined by

$$\begin{aligned} \underline{\mathsf{B}}\,\mathsf{A} &= \mathsf{B} \\ \underline{\mathsf{B}}\,f &= \mathsf{id}_{\mathsf{B}} \end{aligned}$$

List functor: On an object A, this functor yields $\mathsf{List}\;\mathsf{A}$, the type of finite sequences of values all of type A; on arrows, $\mathsf{List} f : \mathsf{List} \mathsf{A} \rightarrow \mathsf{List} \mathsf{B}$ when $f : \mathsf{A} \rightarrow \mathsf{B}$ 'maps' f over a sequence.

Composition of functors: For functors F and G, functor $\mathsf{F} \circ \mathsf{G}$ is defined by

$$\begin{aligned} (\mathsf{F} \circ \mathsf{G})\;\mathsf{A} &= \mathsf{F}\;(\mathsf{G}\;\mathsf{A}) \\ (\mathsf{F} \circ \mathsf{G})\;f &= \mathsf{F}\;(\mathsf{G}\,f) \end{aligned}$$

1.4.6 Binary functors

The notion of a functor may be generalized to functors of more than one argument. A *bifunctor* F is a binary operation on the objects and arrows of a category which respects source and target:

$$\mathsf{F}\;(f, g) : \mathsf{F}(\mathsf{src}(f), \mathsf{src}(g)) \rightarrow \mathsf{F}(\mathsf{tgt}(f), \mathsf{tgt}(g))$$

respects composition:

$$\mathsf{F}\;(f \circ g, h \circ k) = \mathsf{F}\;(f, h) \circ \mathsf{F}\;(g, k)$$

and respects identities:

$$\mathsf{F}\;(\mathsf{id}_{\mathsf{A}}, \mathsf{id}_{\mathsf{B}}) = \mathsf{id}_{\mathsf{F}(\mathsf{A},\mathsf{B})}$$

1.4.7 Examples of bifunctors

See Exercise 1.7.3 for the proofs that the following are bifunctors.

Product: (a generalization of Pair)

$$\begin{aligned} \mathsf{A} \times \mathsf{B} &= \{(a, b) \mid a \in \mathsf{A}, b \in \mathsf{B}\} \\ (f \times g)\;(a, b) &= (f\;a, g\;b) \end{aligned}$$

Projection functors:

$$\begin{aligned} \mathsf{A} \ll \mathsf{B} &= \mathsf{A} \\ f \ll g &= f \end{aligned}$$

1.4.8 Making monofunctors out of bifunctors

Here are two ways of constructing a monofunctor (that is, a functor of a single argument) from a bifunctor.

Sectioning: for bifunctor $\oplus$ and object A, functor $(\mathsf{A}\oplus)$ is defined by

$$\begin{aligned} (\mathsf{A}\oplus)\;\mathsf{B} &= \mathsf{A} \oplus \mathsf{B} \\ (\mathsf{A}\oplus)\;f &= \mathsf{id}_{\mathsf{A}} \oplus f \end{aligned}$$

(so $(\mathsf{A}\ll) = \underline{\mathsf{A}}$, for example), and similarly in the other argument.

Lifting: for bifunctor $\oplus$ and monofunctors F and G, functor $\mathsf{F} \mathbin{\hat{\oplus}} \mathsf{G}$ is defined by

$$\begin{aligned} (\mathsf{F} \mathbin{\hat{\oplus}} \mathsf{G})\, \mathsf{A} &= \mathsf{F}\, \mathsf{A} \oplus \mathsf{G}\, \mathsf{A} \\ (\mathsf{F} \mathbin{\hat{\oplus}} \mathsf{G})\, f &= \mathsf{F}\, f \oplus \mathsf{G}\, f \end{aligned}$$

See Exercise 1.7.4 for the proofs that these do indeed define functors.

1.5 The pair calculus

The pair calculus is an elegant theory of operators on pairs. We have already seen the product bifunctor, one of the two main ingredients of the calculus. The other main ingredient is the *coproduct* bifunctor, the dual of the product, obtained by 'turning all the arrows around' in the definition of product. Along with universal properties, duality is another central theme of these lectures.

1.5.1 Product bifunctor

As we saw above, product $\times$ forms a bifunctor; in $\mathcal{S}et$, for types A and B, the type $\mathsf{A} \times \mathsf{B}$ consists of pairs (a, b) where $a :: \mathsf{A}$ and $b :: \mathsf{B}$. We saw earlier the product destructors $\mathsf{exl} :: \mathsf{A} \times \mathsf{B} \to \mathsf{A}$ and $\mathsf{exr} :: \mathsf{A} \times \mathsf{B} \to \mathsf{B}$. We also saw the product *morphisms* ('forks') $f \vartriangle g :: \mathsf{A} \to \mathsf{B} \times \mathsf{C}$ when $f :: \mathsf{A} \to \mathsf{B}$ and $g :: \mathsf{A} \to \mathsf{C}$, defined by the *universal property*

$$h = f \vartriangle g \Leftrightarrow \mathsf{exl} \circ h = f \ \wedge\ \mathsf{exr} \circ h = g$$

(Some would write '$\langle f, g \rangle$' where we now write '$f \vartriangle g$'.) Now we can define product *map* (that is, the action of the product bifunctor on arrows) using fork:

$$f \times g = (f \circ \mathsf{exl}) \vartriangle (g \circ \mathsf{exr})$$

Here are some properties of fork and product:

$$\begin{aligned} \mathsf{exl} \circ (f \vartriangle g) &= f \\ \mathsf{exr} \circ (f \vartriangle g) &= g \\ (\mathsf{exl} \circ h) \vartriangle (\mathsf{exr} \circ h) &= h \\ \mathsf{exl} \vartriangle \mathsf{exr} &= \mathsf{id} \\ (f \times g) \circ (h \vartriangle k) &= (f \circ h) \vartriangle (g \circ k) \\ \mathsf{id} \times \mathsf{id} &= \mathsf{id} \\ (f \times g) \circ (h \times k) &= (f \circ h) \times (g \circ k) \\ (f \vartriangle g) \circ h &= (f \circ h) \vartriangle (g \circ h) \end{aligned}$$

The proofs are simple consequences of the universal property. We have seen some proofs already; see also Exercise 1.7.5.

1.5.2 Coproduct bifunctor

We define the $\mathcal{S}et$ bifunctor $+$ on objects by

$$\mathsf{A} + \mathsf{B} = \{\mathsf{inl}\ a \mid a \in \mathsf{A}\} \cup \{\mathsf{inr}\ b \mid b \in \mathsf{B}\}$$

The intention here is that inl and inr are injections such that $\mathsf{inl}\ a$ and $\mathsf{inr}\ b$ are distinct, even when $a = b$; thus, coproduct gives a disjoint union. (For example, one might define inl and inr by

$$\begin{aligned}&\mathsf{inl}\ a = (0, a)\\&\mathsf{inr}\ b = (1, b)\end{aligned}$$

but we will not assume any particular definition.) The coproduct *constructors* are the functions $\mathsf{inl} :: \mathsf{A} \to \mathsf{A} + \mathsf{B}$ and $\mathsf{inr} :: \mathsf{B} \to \mathsf{A} + \mathsf{B}$. We define the coproduct *morphisms* ('joins') $f \triangledown g :: \mathsf{A} + \mathsf{B} \to \mathsf{C}$ when $f :: \mathsf{A} \to \mathsf{C}$ and $g :: \mathsf{B} \to \mathsf{C}$, by the *universal property*

$$h = f \triangledown g \Leftrightarrow h \circ \mathsf{inl} = f \ \wedge\ h \circ \mathsf{inr} = g$$

(Some would write '$[f, g]$' where we write '$f \triangledown g$'.) We can now define coproduct map using a join:

$$f + g = (\mathsf{inl} \circ f) \triangledown (\mathsf{inr} \circ g)$$

Here are some properties of join and coproduct:

$$\begin{aligned}
(f \triangledown g) \circ \mathsf{inl} &= f\\
(f \triangledown g) \circ \mathsf{inr} &= g\\
(h \circ \mathsf{inl}) \triangledown (h \circ \mathsf{inr}) &= h\\
\mathsf{inl} \triangledown \mathsf{inr} &= \mathsf{id}\\
(f \triangledown g) \circ (h + k) &= (f \circ h) \triangledown (g \circ k)\\
\mathsf{id} + \mathsf{id} &= \mathsf{id}\\
(f + g) \circ (h + k) &= (f \circ h) + (g \circ k)\\
h \circ (f \triangledown g) &= (h \circ f) \triangledown (h \circ g)
\end{aligned}$$

See Exercise 1.7.5 for the proofs.

1.5.3 Duality

Notice that each of the above properties of join and coproduct is the dual of a property of fork and product, obtained by reversing the order of composition and by exchanging products, forks, and destructors for coproducts, joins and constructors. Duality gives a 'looking-glass world', in which everything is the mirror image of something in the 'everyday' world.

1.5.4 Exchange law

Here is a law relating products and coproducts, a bridge between the everyday world and the looking-glass world:

$$\begin{aligned}
& (f \vartriangle g) \triangledown (h \vartriangle j) = (f \triangledown h) \vartriangle (g \triangledown j)\\
\Leftrightarrow \quad & \qquad \left\{ \text{universal property of } \vartriangle \right\}\\
& \mathsf{exl} \circ ((f \vartriangle g) \triangledown (h \vartriangle j)) = f \triangledown h \ \wedge\\
& \mathsf{exr} \circ ((f \vartriangle g) \triangledown (h \vartriangle j)) = g \triangledown j
\end{aligned}$$

$$
\begin{array}{ll}
\Leftrightarrow & \{\text{composition distributes over join}\} \\
& (\mathsf{exl} \circ (f \vartriangle g)) \triangledown (\mathsf{exl} \circ (h \vartriangle j)) = f \triangledown h \;\wedge \\
& (\mathsf{exr} \circ (f \vartriangle g)) \triangledown (\mathsf{exr} \circ (h \vartriangle j)) = g \triangledown j \\
\Leftrightarrow & \{\text{projections eliminate forks}\} \\
& \mathsf{true}
\end{array}
$$

In fact, there is also a dual proof, using the universal property of joins (Exercise 1.7.6); one might think of it as a proof from the other side of the looking-glass.

1.5.5 Distributivity

In *Set*, the objects $\mathsf{A} \times (\mathsf{B} + \mathsf{C})$ and $(\mathsf{A} \times \mathsf{B}) + (\mathsf{A} \times \mathsf{C})$ are isomorphic. We say that *Set* is a *distributive category*. The isomorphism in one direction,

$$\mathsf{undistl} :: (\mathsf{A} \times \mathsf{B}) + (\mathsf{A} \times \mathsf{C}) \to \mathsf{A} \times (\mathsf{B} + \mathsf{C})$$

is easy to write, in two different ways (Exercise 1.7.7):

$$
\begin{aligned}
\mathsf{undistl} &= (\mathsf{exl} \triangledown \mathsf{exl}) \vartriangle (\mathsf{exr} + \mathsf{exr}) \\
&= (\mathsf{id} \times \mathsf{inl}) \triangledown (\mathsf{id} \times \mathsf{inr})
\end{aligned}
$$

We could also have defined it in a pointwise style:

$$
\begin{aligned}
\mathsf{undistl}\ (\mathsf{inl}\ (a, b)) &= (a, \mathsf{inl}\ b) \\
\mathsf{undistl}\ (\mathsf{inr}\ (a, c)) &= (a, \mathsf{inr}\ c)
\end{aligned}
$$

The inverse operation

$$\mathsf{distl} :: \mathsf{A} \times (\mathsf{B} + \mathsf{C}) \to (\mathsf{A} \times \mathsf{B}) + (\mathsf{A} \times \mathsf{C})$$

is straightforward to define in a pointwise style:

$$
\begin{aligned}
\mathsf{distl}\ (a, \mathsf{inl}\ b) &= \mathsf{inl}\ (a, b) \\
\mathsf{distl}\ (a, \mathsf{inr}\ c) &= \mathsf{inr}\ (a, c)
\end{aligned}
$$

Moreover, these two functions are indeed inverses, as is easy to verify.

However, this inverse cannot be defined in a pointfree style in terms of the product and coproduct operations alone. (Indeed, some categories have products and coproducts, and hence a function undistl as defined above, but no inverse function distl, and so are not distributive categories. Typically, such categories do not support definitions in a pointwise style. The category *Rel* of sets and binary relations is an example.)

1.5.6 Booleans and conditionals

In a distributive category, we can model the datatype of booleans by

$$
\begin{aligned}
\mathsf{Bool} &= 1 + 1 \\
\mathsf{true} &= \mathsf{inl}\ () \\
\mathsf{false} &= \mathsf{inr}\ ()
\end{aligned}
$$

where $()$ is the unique element of the unit type $\mathsf{1}$. For predicate $p :: \mathsf{A} \to \mathsf{Bool}$, we define the *guard*

$$\begin{aligned} &p? :: \mathsf{A} \to (\mathsf{A} + \mathsf{A}) \\ &p? = (\mathsf{exl} + \mathsf{exl}) \circ \mathsf{distl} \circ (\mathsf{id} \vartriangle p) \end{aligned}$$

or, in an equivalent pointwise form,

$$\begin{aligned} p?\,x &= \mathsf{inl}\ x, \text{ if } p\ x \\ &= \mathsf{inr}\ x, \text{ otherwise} \end{aligned}$$

We can then define the conditional

$$\textsc{if}\ p\ \textsc{then}\ f\ \textsc{else}\ g = (f \triangledown g) \circ p?$$

1.6 Bibliographic notes

The *program calculation* field is a flourishing branch of programming methodology. One recent textbook (based on a theory of relations rather than functions, but similar in spirit to the material presented in these lectures) is [4]. Also relevant are the proceedings of the *Mathematics of Program Construction* conferences [39, 2, 30, 21]. There are many good books on *functional programming*; we recommend [5] in particular. The classic reference for *category theory* is [23], but this is rather heavy going for non-mathematicians; for a computing perspective, we recommend [8, 9, 31, 45].

The observation that *universal properties* are very convenient for calculating programs was made originally by Backhouse [1]. The *categorical approach to datatypes* dates back to the ADJ group [13, 14] in the 1970's, but was brought back into fashion by Hagino [16, 17] and Malcolm [24, 25]. The *pair calculus* is probably folklore, but our presentation of it was inspired by Malcolm's thesis. The claim that *distributive categories* are the appropriate venue for discussing datatypes is championed mainly by Walters [44–46].

1.7 Exercises

1. Check that Pair (as defined in §1.4.4) does indeed satisfy the properties required of a functor.
2. Check that operations claimed in §1.4.5 to be functors (identity, constant, list, composition) satisfy the necessary properties.
3. Check that operations claimed in §1.4.7 to be bifunctors ($\times$, $\ll$) satisfy the necessary properties.
4. Check that sectioning and lifting operations claimed in §1.4.8 to construct monofunctors from bifunctors satisfy the necessary properties.
5. Prove the properties of product (from §1.5.1) and of coproduct (from §1.5.2) using the corresponding universal properties.
6. Prove the exchange law from §1.5.4

   $$(f \vartriangle g) \triangledown (h \vartriangle j) = (f \triangledown h) \vartriangle (g \triangledown j)$$

   using the universal property of joins (instead of the universal property of forks).

7. Prove the equivalence of the two characterizations of $\mathsf{undistl}$ from §1.5.5:

   $$(\mathsf{exl} \triangledown \mathsf{exl}) \vartriangle (\mathsf{exr} + \mathsf{exr}) = (\mathsf{id} \times \mathsf{inl}) \triangledown (\mathsf{id} \times \mathsf{inr})$$

   In fact, there are two different proofs, one for each universal property.
8. Prove the following properties of conditionals:

   $$\begin{array}{ll}
h \circ \textsc{if}\ p\ \textsc{then}\ f\ \textsc{else}\ g & = \textsc{if}\ p\ \textsc{then}\ h \circ f\ \textsc{else}\ h \circ g \\
(\textsc{if}\ p\ \textsc{then}\ f\ \textsc{else}\ g) \circ h & = \textsc{if}\ p \circ h\ \textsc{then}\ f \circ h\ \textsc{else}\ g \circ h \\
\textsc{if}\ p\ \textsc{then}\ f\ \textsc{else}\ f & = f \\
\textsc{if}\ \mathsf{not} \circ p\ \textsc{then}\ f\ \textsc{else}\ g & = \textsc{if}\ p\ \textsc{then}\ g\ \textsc{else}\ f \\
\textsc{if}\ \mathsf{const\ true}\ \textsc{then}\ f\ \textsc{else}\ g & = f \\
\textsc{if}\ p\ \textsc{then}\ (\textsc{if}\ q\ \textsc{then}\ f\ \textsc{else}\ g) & = \textsc{if}\ q\ \textsc{then}\ (\textsc{if}\ p\ \textsc{then}\ f\ \textsc{else}\ h) \\
\qquad \textsc{else}\ \ (\textsc{if}\ q\ \textsc{then}\ h\ \textsc{else}\ j) & \qquad \textsc{else}\ \ (\textsc{if}\ p\ \textsc{then}\ g\ \textsc{else}\ j)
\end{array}$$

   (Here, not is negation of booleans, and const is the function such that $\mathsf{const}\ a\ b = a$ for any b.)

2 Recursive datatypes in the category *Set*

The pair calculus is elegant, but not very powerful; descriptive power comes with recursive datatypes. In this section we will discuss a simple first approximation to what we really want, namely recursive datatypes in the category *Set*. We will construct monomorphic and polymorphic datatypes, and their duals. However, there are inherent limitations in working within the category *Set*, which we will remedy in Section 3.

2.1 Overview

The Haskell-style recursive datatype definitions

```
data IntList = Nil | Cons Int IntList
data List a  = Nil | Cons a (List a)
```

(one monomorphic, one polymorphic) give for free:

- a 'map' operator;
- a 'fold' (like join for coproducts), to consume a data structure;
- an 'unfold' (like fork for products), to generate a data structure;
- universal properties for fold and unfold;
- a number of theorems about fold and unfold.

Actually, we will discover that we cannot simultaneously achieve all of these goals in *Set*, which will motivate the move to another category, *Cpo*, in Section 3.

2.2 Monomorphic datatypes

We consider first the case of monomorphic datatypes. The first problem is to identify a common form, encompassing all the datatype declarations in which we are interested. Consider the Haskell-style datatype definition

```
data IntList = Nil | Cons Int IntList
```

This defines two constructors

$$
\begin{aligned}
&\mathit{Nil} \;::\; \mathsf{IntList} \\
&\mathit{Cons} \;::\; \mathsf{Int} \rightarrow (\mathsf{IntList} \rightarrow \mathsf{IntList})
\end{aligned}
$$

Different datatype definitions, of course, will introduce different constructors. This raises some problems for a general theory:

- there may be arbitrarily many constructors;
- the constructors may be constants or functions;
- the constructor functions may be of arbitrary arities.

How can we circumvent these problems, and unify all datatype definitions into a common form?

2.2.1 Unifying constructors

The third problem identified above, constructors of arbitrary arities, can be resolved by 'uncurrying' the constructor functions; that is, by tupling the arguments together using products. For example, the binary *Cons* constructor for lists introduced above is equivalent to the unary constructor

$$\mathit{Cons} :: \mathsf{Int} \times \mathsf{IntList} \rightarrow \mathsf{IntList}$$

The second problem, that some constructors may be constants rather than functions, can be resolved by treating a constant constructor such as *Nil* as a function from the unit type $\mathsf{1}$:

$$\mathit{Nil} :: \mathsf{1} \rightarrow \mathsf{IntList}$$

Now the first problem, of an arbitrary number of constructors, may be resolved by taking the join of the existing collection of unary constructor functions (because they all share a common target, the defined type):

$$\mathit{Nil} \triangledown \mathit{Cons} :: \mathsf{1} + (\mathsf{Int} \times \mathsf{IntList}) \rightarrow \mathsf{IntList}$$

This yields a single constructor $\mathit{Nil} \triangledown \mathit{Cons}$. Being a constructor for the defined type $\mathsf{IntList}$, its target type is that type. Its source type $\mathsf{1} + (\mathsf{Int} \times \mathsf{IntList})$ is some type expression involving the target type $\mathsf{IntList}$ — in fact, some functor applied to $\mathsf{IntList}$.

2.2.2 Datatype definitions

Therefore, it suffices to consider datatypes T with a single unified constructor $\mathsf{in}_{\mathsf{T}} :: \mathsf{F}\ \mathsf{T} \to \mathsf{T}$ for some functor F. We write

$$\mathsf{T} = \textsc{data}\ \mathsf{F}$$

For example, for $\mathsf{IntList}$, the functor is $\mathsf{F}_{\mathsf{IntList}}$, where

$$\mathsf{F}_{\mathsf{IntList}}\ \mathsf{X} = 1 + (\mathsf{Int} \times \mathsf{X})$$

That is,

$$\mathsf{F}_{\mathsf{IntList}} = \underline{1}\ \hat{+}\ (\underline{\mathsf{Int}}\ \hat{\times}\ \mathsf{Id})$$

so we could define

$$\mathsf{IntList} = \textsc{data}\ (\underline{1}\ \hat{+}\ (\underline{\mathsf{Int}}\ \hat{\times}\ \mathsf{Id}))$$

2.3 Folds

We have identified a common form for all monomorphic datatype definitions. However, datatypes are not much use without functions over them. It is now widely accepted that program structure should, where possible, reflect data structure [18]. Accordingly, we should identify a program structure that reflects the data structure of monomorphic datatypes. It turns out that the right kind of structure is one of *homomorphisms* between *algebras*, which we explore in this section.

2.3.1 Fixpoints

The definition '$\mathsf{T} = \textsc{data}\ \mathsf{F}$' defines T to be a *fixpoint* of the functor F; that is, T is isomorphic to $\mathsf{F}\,\mathsf{T}$. In one direction, the isomorphism is given by $\mathsf{in}_{\mathsf{T}} :: \mathsf{F}\,\mathsf{T} \to \mathsf{T}$. In the other direction, we suppose an inverse $out_{\mathsf{T}} :: \mathsf{T} \to \mathsf{F}\ \mathsf{T}$. (In fact, we see how to define out_{T} shortly.)

However, to say that the datatype definition '$\mathsf{T} = \textsc{data}\ \mathsf{F}$' defines T to be a fixpoint of the functor F does not completely determine T, as a functor may have more than one fixpoint. For example, the types 'finite sequences of integers' and 'finite and infinite sequences of integers' are both fixpoints of the functor $\mathsf{F}_{\mathsf{IntList}}$ (Exercise 2.9.3). Informally, what we want is the 'least fixpoint', that is, the 'smallest such type' — finite rather than finite-and-infinite sequences of integers. How can we formalize this idea?

2.3.2 Algebras

We define an F-*algebra* to be a pair (A, f) such that $f :: \mathsf{F}\ \mathsf{A} \to \mathsf{A}$. Thus, the datatype definition $\mathsf{T} = \textsc{data}\ \mathsf{F}$ defines $(\mathsf{T}, \mathsf{in}_{\mathsf{T}})$ to be an F-algebra. For example, $(\mathsf{IntList}, Nil \triangledown Cons)$ is an $\mathsf{F}_{\mathsf{IntList}}$-algebra. However, F-algebras are not unique either. For example, $(\mathsf{Int}, zero \triangledown plus)$ is another $\mathsf{F}_{\mathsf{IntList}}$-algebra (Exercise 2.9.4), where $zero :: 1 \to \mathsf{Int}$ and $plus :: \mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$; that is, $zero \triangledown plus :: 1 + (\mathsf{Int} \times \mathsf{Int}) \to \mathsf{Int}$.

2.3.3 Homomorphisms

A *homomorphism* between F-algebras (A, f) and (B, g) is a function $h :: \mathsf{A} \to \mathsf{B}$ such that

$$h \circ f = g \circ \mathsf{F}\, h$$

Pictorially,

$$\begin{tikzcd}
\mathsf{F\,A} \arrow[r, "f"] \arrow[d, "\mathsf{F}\,h"'] & \mathsf{A} \arrow[d, "h"] \\
\mathsf{F\,B} \arrow[r, "g"'] & \mathsf{B}
\end{tikzcd}$$

For example, the function $sum :: \mathsf{IntList} \to \mathsf{Int}$, which sums an $\mathsf{IntList}$,

$$\begin{aligned}
&sum\ (Nil\ ()) &&= 0 \\
&sum\ (Cons\ (a, x)) &&= a + sum\ x
\end{aligned}$$

is a homomorphism from $(\mathsf{IntList}, Nil \triangledown Cons)$ to $(\mathsf{Int}, zero \triangledown plus)$, because

$$sum \circ (Nil \triangledown Cons) = (zero \triangledown plus) \circ \mathsf{F}_{\mathsf{IntList}}\ sum$$

(see Exercise 2.9.5).

2.3.4 Initial algebras

We say that an F-algebra (A, f) is *initial* if, for any F-algebra (B, g), there is a unique homomorphism from (A, f) to (B, g). Then the datatype definition '$\mathsf{T} = \textsc{data}\ \mathsf{F}$' defines $(\mathsf{T}, \mathsf{in}_{\mathsf{T}})$ to be 'the' initial F-algebra. There may be more than one initial algebra, but all initial algebras are equivalent (Exercise 2.9.6); thus, it does not really matter which one we pick.

2.3.5 Existence of initial algebras

It is well-known that for *polynomial* F (built out of identity and constant functors using product and coproduct) on many categories including $\mathcal{S}et$ and $\mathcal{R}el$, initial algebras always exist. Malcolm [24] shows existence also for *regular* F (adding fixpoints), allowing us to define *mutually recursive* datatypes such as

```
data IntTree   = Node Int IntForest
data IntForest = Empty | ConsF IntTree IntForest
```

2.3.6 Definition of folds

Suppose that $(\mathsf{T}, \mathsf{in}_{\mathsf{T}})$ is the initial F-algebra. Then there is a unique homomorphism to any F-algebra (B, f) — that is, for any such f, there exists a unique h such that $h \circ \mathsf{in}_{\mathsf{T}} = f \circ \mathsf{F}\, h$. We would like a notation for 'the unique solution

h of this equation involving f'; we write '$\mathsf{fold}_\mathsf{T}\,f$' for this unique solution. Thus, $\mathsf{fold}_\mathsf{T}\,f$ has type $\mathsf{T} \to \mathsf{B}$ when $f :: \mathsf{F\,B} \to \mathsf{B}$. Pictorially,

$$\begin{tikzcd}
\mathsf{F\,T} \arrow[r, "\mathsf{in}"] \arrow[d, "\mathsf{F}\,(\mathsf{fold}_\mathsf{T}\,f)"'] & \mathsf{T} \arrow[d, "\mathsf{fold}_\mathsf{T}\,f"] \\
\mathsf{F\,B} \arrow[r, "f"'] & \mathsf{B}
\end{tikzcd}$$

Uniqueness provides the universal property

$$h = \mathsf{fold}_\mathsf{T}\,f \Leftrightarrow h \circ \mathsf{in}_\mathsf{T} = f \circ \mathsf{F}\,h$$

2.4 Polymorphic datatypes

The type $\mathsf{IntList}$ has the 'base type' Int built in: it cannot be used for lists of booleans, lists of strings, and so on. We would like *polymorphic* datatypes, parameterized by an arbitrary base type A: lists of As, trees of As, and so on. For example, the Haskell-style type definition

```
data List a  = Nil | Cons a (List a)
```

defines a type $\mathsf{List\,A}$ for each type A; now List is a type *constructor*, whereas $\mathsf{IntList}$ is just a type.

2.4.1 Using bifunctors

The essential idea in constructing polymorphic datatypes is to use a bifunctor $\oplus$. A polymorphic type T is then defined by sectioning $\oplus$ with the type parameter as one argument, and then taking the fixpoint:

$$\mathsf{T\,A} = \textsc{data}\ (\mathsf{A}\oplus)$$

Now the constructor has type

$$\mathsf{in}_{\mathsf{T\,A}} :: \mathsf{A} \oplus \mathsf{T\,A} \to \mathsf{T\,A}$$

though usually we will write just 'in_T' as a polymorphic function, omitting the A. For example, we can define a polymorphic list type by

$$\mathsf{List\,A} = \textsc{data}\ (\mathsf{A}\oplus)$$

where

$$\mathsf{A} \oplus \mathsf{B} = 1 + (\mathsf{A} \times \mathsf{B})$$

Equivalently, we could write

$$\mathsf{List\,A} = \textsc{data}\ (\underline{1} \mathbin{\hat{+}} (\underline{\mathsf{A}} \mathbin{\hat{\times}} \mathsf{Id}))$$

without naming the bifunctor.

2.4.2 Polymorphic folds

Folds over monomorphic datatypes generalize in a straightforward fashion to polymorphic datatypes. The datatype definition

$$\mathsf{T\,A} = \textsc{data}\ (\mathsf{A}\oplus)$$

defines $(\mathsf{T\,A}, \mathsf{in_T})$ to be the initial $(\mathsf{A}\oplus)$-algebra; therefore there exists a unique homomorphism $\mathsf{fold}_{\mathsf{T\,A}} f$ to any other $(\mathsf{A}\oplus)$-algebra (B, f). (Again, we will usually write just '$\mathsf{fold_T}\,f$', leaving the fold operator polymorphic in A.) The fold $\mathsf{fold_T}\,f$ has type $\mathsf{T\,A} \to \mathsf{B}$ when $f :: \mathsf{A} \oplus \mathsf{B} \to \mathsf{B}$; pictorially,

$$\begin{tikzcd}
\mathsf{A} \oplus \mathsf{T\,A} \arrow[r, "\mathsf{in_T}"] \arrow[d, "\mathsf{id} \oplus \mathsf{fold_T}\,f"'] & \mathsf{T\,A} \arrow[d, "\mathsf{fold_T}\,f"] \\
\mathsf{A} \oplus \mathsf{B} \arrow[r, "f"'] & \mathsf{B}
\end{tikzcd}$$

Uniqueness gives the universal property

$$h = \mathsf{fold_T}\,f \Leftrightarrow h \circ \mathsf{in_T} = f \circ (\mathsf{id} \oplus h)$$

2.4.3 Making it a functor: map

The datatype definition $\mathsf{T\,A} = \textsc{data}\ (\mathsf{A}\oplus)$ makes T a type constructor, an operation on types. This suggests that perhaps we can make T a functor: all we need is a corresponding operation on functions $\mathsf{T}\,f$ with type $\mathsf{T\,A} \to \mathsf{T\,B}$ when $f :: \mathsf{A} \to \mathsf{B}$ (satisfying the functor laws). We define $\mathsf{T}\,f = \mathsf{fold}_{\mathsf{T\,A}}\,(\mathsf{in}_{\mathsf{T\,B}} \circ (f \oplus \mathsf{id}))$. Pictorially,

$$\begin{tikzcd}
\mathsf{A} \oplus \mathsf{T\,A} \arrow[rr, "\mathsf{in}_{\mathsf{T\,A}}"] \arrow[d, "\mathsf{id} \oplus \mathsf{T}\,f"'] & & \mathsf{T\,A} \arrow[d, "\mathsf{T}\,f"] \\
\mathsf{A} \oplus \mathsf{T\,B} \arrow[r, "f \oplus \mathsf{id}"'] & \mathsf{B} \oplus \mathsf{T\,B} \arrow[r, "\mathsf{in}_{\mathsf{T\,B}}"'] & \mathsf{T\,B}
\end{tikzcd}$$

(We should check that this does indeed satisfy the requirements for being a functor; see Exercise 2.9.7.) For historical reasons, we will write '$\mathsf{map_T}\,f$' rather than '$\mathsf{T}\,f$'.

2.5 Properties of folds

There are a number of general theorems about folds that arise as simple consequences of the universal property. These include: an *evaluation rule*, which shows 'one step of evaluation' of a fold; an exact *fusion law*, which states when a function can be fused with a fold; a *weak fusion law*, a simpler but weaker corollary of the exact fusion law; the *identity law*, which states that the identity function is a fold; and a definition of the *destructor* of a datatype as a fold.

2.5.1 Evaluation rule

The evaluation rule describes the composition of a fold and the constructors of its type; informally, it gives 'one step of evaluation' of the fold.

$$\begin{array}{ll} & \mathsf{fold}_{\mathsf{T}}\, f \circ \mathsf{in}_{\mathsf{T}} \\ = & \qquad \{\text{universal property, letting } h = \mathsf{fold}\, f\,\} \\ & f \circ \mathsf{F}\, (\mathsf{fold}_{\mathsf{T}}\, f) \end{array}$$

2.5.2 Fusion (exact version)

Fusion laws for folds are of the form

$$h \circ \mathsf{fold}_{\mathsf{T}}\, f = \mathsf{fold}_{\mathsf{T}}\, g \Leftrightarrow \ldots$$

(or sometimes with the composition the other way around). They give conditions under which one can *fuse* two computations, one a fold, to yield a single monolithic computation. In this case, we have

$$\begin{array}{ll} & h \circ \mathsf{fold}_{\mathsf{T}}\, f = \mathsf{fold}_{\mathsf{T}}\, g \\ \Leftrightarrow & \qquad \{\text{universal property}\} \\ & h \circ \mathsf{fold}_{\mathsf{T}}\, f \circ \mathsf{in}_{\mathsf{T}} = g \circ \mathsf{F}\, (h \circ \mathsf{fold}_{\mathsf{T}}\, f) \\ \Leftrightarrow & \qquad \{\text{functors}\} \\ & h \circ \mathsf{fold}_{\mathsf{T}}\, f \circ \mathsf{in}_{\mathsf{T}} = g \circ \mathsf{F}\, h \circ \mathsf{F}\, (\mathsf{fold}_{\mathsf{T}}\, f) \\ \Leftrightarrow & \qquad \{\text{evaluation rule}\} \\ & h \circ f \circ \mathsf{F}\, (\mathsf{fold}_{\mathsf{T}}\, f) = g \circ \mathsf{F}\, h \circ \mathsf{F}\, (\mathsf{fold}_{\mathsf{T}}\, f) \end{array}$$

2.5.3 Fusion (weaker version)

The above fusion law is an equivalence, so it is as strong as possible. However, it is a little unwieldy, because the premise (the last line in the calculation above) is rather long. Here is a fusion law with a simpler but stronger premise (which therefore is a weaker law).

$$\begin{array}{ll} & h \circ \mathsf{fold}_{\mathsf{T}}\, f = \mathsf{fold}_{\mathsf{T}}\, g \\ \Leftrightarrow & \qquad \{\text{exact fusion}\} \\ & h \circ f \circ \mathsf{F}\, (\mathsf{fold}_{\mathsf{T}}\, f) = g \circ \mathsf{F}\, h \circ \mathsf{F}\, (\mathsf{fold}_{\mathsf{T}}\, f) \\ \Leftarrow & \qquad \{\text{Leibniz}\} \\ & h \circ f = g \circ \mathsf{F}\, h \end{array}$$

2.5.4 Identity

The identity function id is a fold:

$$\begin{array}{ll}
 & \mathsf{id} = \mathsf{fold}_{\mathsf{T}}\, f \\
\Leftrightarrow & \qquad \{\, \text{universal property} \,\} \\
 & \mathsf{id} \circ \mathsf{in}_{\mathsf{T}} = f \circ \mathsf{F}\, \mathsf{id} \\
\Leftrightarrow & \qquad \{\, \text{identity} \,\} \\
 & f = \mathsf{in}_{\mathsf{T}}
\end{array}$$

That is, $\mathsf{fold}_{\mathsf{T}}\, \mathsf{in}_{\mathsf{T}} = \mathsf{id}$.

2.5.5 Destructors

Also, the destructor $\mathit{out}_{\mathsf{T}}$ of a datatype, the inverse of the constructor in_{T}, can be written as a fold; this is known as *Lambek's Lemma*.

$$\begin{array}{ll}
 & \mathsf{in}_{\mathsf{T}} \circ \mathsf{fold}_{\mathsf{T}}\, f = \mathsf{id} \\
\Leftrightarrow & \qquad \{\, \text{identity as a fold} \,\} \\
 & \mathsf{in}_{\mathsf{T}} \circ \mathsf{fold}_{\mathsf{T}}\, f = \mathsf{fold}_{\mathsf{T}}\, \mathsf{in}_{\mathsf{T}} \\
\Leftarrow & \qquad \{\, \text{weak fusion} \,\} \\
 & \mathsf{in}_{\mathsf{T}} \circ f = \mathsf{in}_{\mathsf{T}} \circ \mathsf{F}\, \mathsf{in}_{\mathsf{T}} \\
\Leftarrow & \qquad \{\, \text{Leibniz} \,\} \\
 & f = \mathsf{F}\, \mathsf{in}_{\mathsf{T}}
\end{array}$$

Therefore we can define

$$\mathit{out}_{\mathsf{T}} = \mathsf{fold}_{\mathsf{T}}\, (\mathsf{F}\, \mathsf{in}_{\mathsf{T}})$$

We should check that this also makes out the inverse of in when the composition is reversed:

$$\begin{array}{ll}
 & \mathit{out}_{\mathsf{T}} \circ \mathsf{in}_{\mathsf{T}} \\
= & \qquad \{\, \text{above} \,\} \\
 & \mathsf{fold}_{\mathsf{T}}\, (\mathsf{F}\, \mathsf{in}_{\mathsf{T}}) \circ \mathsf{in}_{\mathsf{T}} \\
= & \qquad \{\, \text{evaluation rule} \,\} \\
 & \mathsf{F}\, \mathsf{in}_{\mathsf{T}} \circ \mathsf{F}\, \mathit{out}_{\mathsf{T}} \\
= & \qquad \{\, \text{functors} \,\} \\
 & \mathsf{F}\, (\mathsf{in}_{\mathsf{T}} \circ \mathit{out}_{\mathsf{T}}) \\
= & \qquad \{\, \mathsf{in} \circ \mathit{out} = \mathsf{id} \,\} \\
 & \mathsf{id}
\end{array}$$

Lambek's Lemma is a corollary of the more general theorem that an injective function (that is, a function with a post-inverse) on a recursive datatype is a fold (Exercise 2.9.8). Since the destructor is by assumption the inverse of the constructors, it is injective.

2.6 Co-datatypes and unfolds

All of this theory of datatypes dualizes, to give a theory of *co-datatypes* and *unfolds*. The dualization is quite straightforward; nevertheless, we present the facts here for completeness.

2.6.1 Co-algebras and homomorphisms

An F-*co-algebra* is a pair (A, f) such that $f :: \mathsf{A} \to \mathsf{F\,A}$. A *homomorphism* between F-co-algebras (A, f) and (B, g) is a function $h :: \mathsf{A} \to \mathsf{B}$ such that

$$\mathsf{F}\,h \circ f = g \circ h$$

Pictorially,

$$\begin{tikzcd}
\mathsf{A} \arrow[r, "f"] \arrow[d, "h"'] & \mathsf{F\,A} \arrow[d, "\mathsf{F}\,h"] \\
\mathsf{B} \arrow[r, "g"'] & \mathsf{F\,B}
\end{tikzcd}$$

An F-co-algebra (A, f) is *final* if, for any F-co-algebra (B, g), there is a unique homomorphism from (B, g) to (A, f). The datatype definition $\mathsf{T} = \textsc{codata}\ \mathsf{F}$ defines $(\mathsf{T}, \mathsf{out_T})$ to be 'the' final F-co-algebra.

2.6.2 Unfolds

Suppose that $(\mathsf{T}, \mathsf{out_T})$ is the final F-co-algebra. Then there is a unique homomorphism to $(\mathsf{T}, \mathsf{out_T})$ from any F-co-algebra (B, f) — that is, there exists a unique h such that $\mathsf{out_T} \circ h = \mathsf{F}\,h \circ f$. We write '$\mathsf{unfold_T}\,f$' for this homomorphism. The unfold $\mathsf{unfold_T}\,f$ has type $\mathsf{B} \to \mathsf{T}$ when $f :: \mathsf{B} \to \mathsf{F\,B}$:

$$\begin{tikzcd}
\mathsf{B} \arrow[r, "f"] \arrow[d, "\mathsf{unfold_T}\,f"'] & \mathsf{F\,B} \arrow[d, "\mathsf{F}\,(\mathsf{unfold_T}\,f)"] \\
\mathsf{T} \arrow[r, "\mathsf{out_T}"'] & \mathsf{F\,T}
\end{tikzcd}$$

Uniqueness provides the universal property

$$h = \mathsf{unfold_T}\,f \Leftrightarrow \mathsf{out_T} \circ h = \mathsf{F}\,h \circ f$$

2.6.3 Polymorphic co-datatypes

In the same way,

$$\mathsf{T\,A} = \textsc{codata}\ (\mathsf{A}\oplus)$$

defines a polymorphic co-datatype, with destructor

$$\mathsf{out}_{\mathsf{T\,A}} :: \mathsf{T\,A} \to \mathsf{A} \oplus \mathsf{T\,A}$$

This induces a polymorphic unfold with universal property

$$h = \mathsf{unfold}_{\mathsf{T\,A}}\,f \Leftrightarrow \mathsf{out}_{\mathsf{T}} \circ h = (\mathsf{id}{\oplus}h) \circ f$$

The typing is $\mathsf{unfold}_{\mathsf{T}}\,f :: \mathsf{B} \to \mathsf{T\,A}$ when $f :: \mathsf{B} \to \mathsf{A} \oplus \mathsf{B}$; pictorially,

$$\begin{tikzcd}
\mathsf{B} \arrow[r, "f"] \arrow[d, "\mathsf{unfold}_{\mathsf{T}}\,f"'] & \mathsf{A} \oplus \mathsf{B} \arrow[d, "\mathsf{id} \oplus \mathsf{unfold}_{\mathsf{T}}\,f"] \\
\mathsf{T\,A} \arrow[r, "\mathsf{out}_{\mathsf{T}}"'] & \mathsf{A} \oplus \mathsf{T\,A}
\end{tikzcd}$$

Co-datatypes too form functors; the map for $f :: \mathsf{A} \to \mathsf{B}$ is given by

$$\mathsf{map}_{\mathsf{T\,A}}\,f = \mathsf{unfold}_{\mathsf{T\,B}}\,((f{\oplus}\mathsf{id}) \circ \mathsf{out}_{\mathsf{T\,A}})$$

2.6.4 An example: streams

The polymorphic datatype of streams (infinite lists) is defined

$$\mathsf{Stream\,A} = \textsc{codata}\ (\mathsf{A}\times)$$

Thus, the destructor for this type is $\mathsf{out}_{\mathsf{Stream}} :: \mathsf{Stream\,A} \to \mathsf{A} \times \mathsf{Stream\,A}$. The unfold $\mathsf{unfold}_{\mathsf{Stream}}\,f$ has type $\mathsf{A} \to \mathsf{Stream\,B}$ for $f :: \mathsf{A} \to \mathsf{B} \times \mathsf{A}$. For example,

$$\mathit{from} = \mathsf{unfold}_{\mathsf{Stream\,Int}}\,(\mathsf{id} \vartriangle (1+))$$

yields increasing streams of naturals: $\mathit{from}\ n = n, n+1, n+2, \ldots$. For another example,

$$\mathit{fibs} = (\mathsf{unfold}_{\mathsf{Stream\,Int}}\,(\mathsf{exl} \vartriangle (\mathsf{exr} \vartriangle \mathit{plus})))\ (0,1)$$

defines the Fibonacci sequence $0, 1, 1, 2, 3, 5, 8, \ldots$.

2.6.5 Properties of unfolds

The theorems dualize too, of course. See Exercise 2.9.10 for the proofs.

Evaluation rule:

$$\mathsf{out}_{\mathsf{T}} \circ \mathsf{unfold}_{\mathsf{T}}\,f = \mathsf{F}\,(\mathsf{unfold}_{\mathsf{T}}\,f) \circ f$$

Exact and weak fusion:

$$\begin{aligned}&\mathsf{unfold}_{\mathsf{T}}\, f \circ h = \mathsf{unfold}_{\mathsf{T}}\, g\\ &\qquad\Leftrightarrow \mathsf{F}\,(\mathsf{unfold}_{\mathsf{T}}\, f) \circ f \circ h = \mathsf{F}\,(\mathsf{unfold}_{\mathsf{T}}\, f) \circ \mathsf{F}\, h \circ g\\ &\qquad\Leftarrow f \circ h = \mathsf{F}\, h \circ g\end{aligned}$$

Identity:

$$\mathsf{unfold}_{\mathsf{T}}\ \mathsf{out}_{\mathsf{T}} = \mathsf{id}$$

Constructors: (the dual of the 'destructor' law for folds)

$$in_{\mathsf{T}} = \mathsf{unfold}_{\mathsf{T}}\,(\mathsf{F}\,\mathsf{out}_{\mathsf{T}})$$

Again, this dual is a corollary of a more general law (Exercise 2.9.11), that any surjective function (one with a pre-inverse) to a recursive datatype is an unfold.

2.6.6 Example: insertion sort

Given the datatype $\mathsf{List\ A} = \textsc{data}\ (\underline{\mathsf{1}} \mathbin{\hat{+}} (\underline{\mathsf{A}} \mathbin{\hat{\times}} \mathsf{Id}))$, suppose we have an insertion operation

$$ins :: \mathsf{1} + (\mathsf{A} \times \mathsf{List\ A}) \to \mathsf{List\ A}$$

that gives an empty list, or inserts an element into a sorted list. Then insertion sort is defined by

$$insertsort = \mathsf{fold}_{\mathsf{List}}\ ins$$

2.6.7 Example: selection sort

Given the codatatype $\mathsf{CList\ A} = \textsc{codata}\ (\underline{\mathsf{1}} \mathbin{\hat{+}} (\underline{\mathsf{A}} \mathbin{\hat{\times}} \mathsf{Id}))$, suppose we have an operation

$$del :: \mathsf{CList\ A} \to \mathsf{1} + (\mathsf{A} \times \mathsf{CList\ A})$$

that finds and removes the minimum element of a non-empty list. Then selection sort is defined by

$$selectsort = \mathsf{unfold}_{\mathsf{CList}}\ del$$

2.7 . . . and never the twain shall meet

Unfortunately, this elegant theory is severely limited when it comes to actual programming. Datatypes and co-datatypes are different things, so one cannot combine them. For example, one cannot write programs of the form 'unfold then fold'; one instance of this scheme is quicksort, which builds a binary search tree (an unfold) then flattens it to a list (a fold), and another is mergesort, which repeatedly halves a list (unfolding to a tree) then repeatedly merges the fragments (folding the tree). This pattern of computation is known as a *hylomorphism*, and is very common in programming.

Moreover, $\mathcal{S}et$ is not a good model of programs. As it contains only total functions, it necessarily suffers from some lack of power, and corresponds only vaguely to most programming languages. (Indeed, the selection sort given in §2.6.7 does not really work: the function *del* is necessarily partial, as it makes no sense on an infinite list, and so neither *del* nor *selectsort* are arrows in $\mathcal{S}et$.)

The solution to both problems is to move to the category $\mathcal{C}po$, imposing more structure on the objects and arrows of the category than there is in $\mathcal{S}et$.

2.8 Bibliographic notes

As mentioned in the bibliographic notes for the previous section, the categorical approach to datatypes is due originally to the ADJ group [13, 14] and later to Hagino [16, 17]. However, the presentation in these notes owes more to Malcolm [24, 25]. The proof that, for the kinds of functor that interest us, initial algebras and final coalgebras always exist, is (a corollary of a more general theorem) due to Smyth and Plotkin [34]. The term 'hylomorphism' is due to Meijer [27].

2.9 Exercises

1. Translate the following Haskell-style definition of binary trees with boolean external labels into the categorical style:

   ```
   data BoolTree = Tip Bool | Bin BoolTree BoolTree
   ```

2. Translate the following categorical-style datatype definition

   $$\mathsf{StringTree} = \textsc{data}\ (\underline{\mathsf{1}} \mathbin{\hat{+}} (\mathsf{Id} \mathbin{\hat{\times}} \underline{\mathsf{String}} \mathbin{\hat{\times}} \mathsf{Id}))$$

   into your favourite programming languages (for example, Haskell, Modula 2, Java).
3. Show that the types 'finite sequences of integers' and 'finite and infinite sequences of integers' are both fixpoints of the functor $\underline{\mathsf{1}} \mathbin{\hat{+}} (\underline{\mathsf{Int}} \mathbin{\hat{\times}} \mathsf{Id})$.
4. Check that $(\mathsf{IntList}, \mathit{Nil} \triangledown \mathit{Cons})$ and $(\mathsf{Int}, \mathit{zero} \triangledown \mathit{plus})$ are $\mathsf{F}_{\mathsf{IntList}}$-algebras, where

   $$\begin{aligned} \mathit{zero}\ () &= 0 \\ \mathit{plus}\ (m, n) &= m + n \end{aligned}$$

5. Check that *sum*, the function which sums an $\mathsf{IntList}$,

   $$\begin{aligned} \mathit{sum}\ (\mathit{Nil}\ ()) &= 0 \\ \mathit{sum}\ (\mathit{Cons}\ (a, x)) &= a + \mathit{sum}\ x \end{aligned}$$

   is an $\mathsf{F}_{\mathsf{IntList}}$-homomorphism from $(\mathsf{IntList}, \mathit{Nil} \triangledown \mathit{Cons})$ to $(\mathsf{Int}, \mathit{zero} \triangledown \mathit{plus})$.
6. Show that any two initial F-algebras are isomorphic. (Hint: the identity function is a homomorphism from an F-algebra to itself; use uniqueness.) So, given the existence of an initial algebra, we are justified in talking about 'the' initial algebra.

7. Check that defining

   $$\mathsf{T}\,f = \mathsf{fold}_{\mathsf{T\,A}}\,(\mathsf{in}_{\mathsf{T\,B}} \circ (f \oplus \mathsf{id}))$$

   does indeed make T a functor.
8. Show that if $g \circ h = \mathsf{id}_{\mathsf{T}}$ for recursive datatype T, then h is a fold. Thus, any injective function on a recursive datatype is a fold.
9. In fact, one can say something stronger. Show that h is a fold for recursive datatype $\textsc{data}\ \mathsf{F}$ if and only if $\textsc{ker}\,(\mathsf{F}\,h) \subseteq \textsc{ker}\,(h \circ \mathsf{in})$, where the kernel $\textsc{ker} f$ of a function $f :: \mathsf{A} \to \mathsf{B}$ is the set of pairs $\{\ (a, a') \in \mathsf{A} \times \mathsf{A} \mid f\ a = f\ a'\ \}$. Use this to solve Exercise 2.9.8.
10. Prove the properties of unfolds from §2.6.5, using the universal property.
11. Dually to Exercise 2.9.8, show that any surjective function to a recursive datatype is an unfold.
12. Dually to Exercise 2.9.9, show that h is a unfold for recursive codatatype $\textsc{codata}\ \mathsf{F}$ if and only if $\textsc{img}\,(\mathsf{F}\,h) \supseteq \textsc{img}\,(\mathsf{out} \circ h)$, where the image $\textsc{img}\,f$ of a function $f :: \mathsf{A} \to \mathsf{B}$ is the set $\{\ b \in \mathsf{B} \mid \exists a \in \mathsf{A}.\ f\ a = b\ \}$. Use this to solve Exercise 2.9.11.
13. Prove that the fork of two folds is a fold:

   $$\mathsf{fold}_{\mathsf{T}}\,f \vartriangle \mathsf{fold}_{\mathsf{T}}\,g = \mathsf{fold}_{\mathsf{T}}\,((f \circ \mathsf{F\,exl}) \vartriangle (g \circ \mathsf{F\,exr}))$$

   (This is known fondly as the 'banana split theorem', by those who know the fork operation as 'split' and write folds using 'banana brackets'.)
14. Prove the special cases *fold-map fusion*

   $$\mathsf{fold}_{\mathsf{T}}\,f \circ \mathsf{map}_{\mathsf{T}}\,g = \mathsf{fold}_{\mathsf{T}}\,(f \circ (g \oplus \mathsf{id}))$$

   of the fusion law for folds, and *map-unfold fusion*

   $$\mathsf{map}_{\mathsf{T}}\,g \circ \mathsf{unfold}_{\mathsf{T}}\,f = \mathsf{unfold}_{\mathsf{T}}\,((g \oplus \mathsf{id}) \circ f)$$

   of the fusion law for unfolds.
15. For datatype $\mathsf{T} = \textsc{data}\ \mathsf{F}$, Meertens [26] defines the notion of a *paramorphism* $\mathsf{para}_{\mathsf{T}}\,f :: \mathsf{T} \to \mathsf{C}$ when $f :: \mathsf{F}\,(\mathsf{C} \times \mathsf{T}) \to \mathsf{C}$ as follows:

   $$\mathsf{para}_{\mathsf{T}}\,f = \mathsf{exl} \circ \mathsf{fold}_{\mathsf{T}}\,(f \vartriangle (\mathsf{in}_{\mathsf{T}} \circ \mathsf{F\,exr}))$$

   It enjoys the universal property

   $$h = \mathsf{para}_{\mathsf{T}}\,f \Leftrightarrow h \circ \mathsf{in}_{\mathsf{T}} = f \circ \mathsf{F}\,(h \vartriangle \mathsf{id})$$

   Informally, a paramorphism is a generalization of a fold: the result on a larger structure may depend on results on substructures, but also on the substructures themselves. For example, the factorial function is a paramorphism over the naturals:

   $$\mathit{fact} = \mathsf{para}_{\mathsf{Nat}}\,(\mathsf{const}\,1 \triangledown (\mathit{mult} \circ (\mathsf{id} \times \mathit{succ})))$$

   where $\mathsf{const}\,a\,b = a$ and mult multiplies a pair of numbers. That is, $\mathit{fact}\,0 = 1$, and $\mathit{fact}\,(\mathit{succ}\ n) = \mathit{mult}\,(\mathit{fact}\ n, \mathit{succ}\ n)$.
   (a) Show that the second component of the above fold is merely the identity function:

   $$\mathsf{exr} \circ \mathsf{fold}_{\mathsf{T}}\,(f \vartriangle (\mathsf{in}_{\mathsf{T}} \circ \mathsf{F\,exr})) = \mathsf{id}$$

   Hence $\mathsf{fold}_{\mathsf{T}}\,(f \vartriangle (\mathsf{in}_{\mathsf{T}} \circ \mathsf{F\,exr})) = \mathsf{para}_{\mathsf{T}}\,f \vartriangle \mathsf{id}$.

(b) Show that the identity function is a paramorphism:

$$\mathsf{id} = \mathsf{para}\ (\mathsf{in} \circ \mathsf{F}\ \mathsf{exl})$$

(c) Prove the (weak) fusion law for paramorphisms:

$$h \circ \mathsf{para}\ f = \mathsf{para}\ g \Leftarrow h \circ f = g \circ \mathsf{F}\ (h \times \mathsf{id})$$

(d) Show that any fold is a paramorphism:

$$\mathsf{fold}\ f = \mathsf{para}\ (f \circ \mathsf{F}\ \mathsf{exl})$$

(This is a generalization of Exercise 2.9.15b.)

(e) Show that *any* function on a recursive datatype can be written as a paramorphism:

$$h = \mathsf{para}\ (h \circ \mathsf{in} \circ \mathsf{F}\ \mathsf{exr})$$

Thus, paramorphisms are extremely general.

16. On the codatatype of lists from §2.6.7, define as an unfold the function *interval*, such that

$$\begin{aligned} &interval\ (1,5) = [1,2,3,4,5] \\ &interval\ (5,5) = [5] \\ &interval\ (6,5) = [\,] \end{aligned}$$

17. On the codatatype $\mathsf{Stream\ A} = \textsc{codata}\ (\mathsf{A}\times)$, the function *iterate* is defined by

$$iterate\ f = \mathsf{unfold}_{\mathsf{Stream}}\ (\mathsf{id} \vartriangle f)$$

Using unfold fusion, prove that

$$\mathsf{map}\ f \circ iterate\ f = iterate\ f \circ f$$

18. For codatatype $\mathsf{T} = \textsc{codata}\ \mathsf{F}$, Uustalu and Vene [40, 38] dualize paramorphisms to get *apomorphisms* $\mathsf{apo_T}\ f :: \mathsf{C} \to \mathsf{T}$ when $f :: \mathsf{C} \to \mathsf{F}\ (\mathsf{C} + \mathsf{T})$ as follows:

$$\mathsf{apo_T}\ f = \mathsf{unfold_T}\ (f \triangledown (\mathsf{F}\ \mathsf{inr} \circ \mathsf{out_T})) \circ \mathsf{inl}$$

They enjoy the universal property

$$h = \mathsf{apo_T}\ f \Leftrightarrow \mathsf{out_T} \circ h = \mathsf{F}\ (h \triangledown \mathsf{id}) \circ f$$

Informally, an apomorphism is a generalization of an unfold: a larger structure may be generated recursively from new seeds, but may also be generated 'all at once' without recursion. For example, on the codatatype $\mathsf{CList\ A} = \textsc{codata}\ (\underline{1}\ \hat{+}\ (\underline{\mathsf{A}}\ \hat{\times}\ \mathsf{Id}))$ of lists, the append function is an apomorphism:

$$append = \mathsf{apo_{Clist}}\ f$$

where

$$\begin{aligned} f\ (x,y) &= \mathsf{inl}\ (), &&\text{if } null\ x \wedge null\ y \\ &= \mathsf{inr}\ (head\ y, \mathsf{inr}\ (tail\ y)), &&\text{if } null\ x \wedge \mathsf{not}\ (null\ y) \\ &= \mathsf{inr}\ (head\ x, \mathsf{inl}\ (tail\ x, y)), &&\text{if } \mathsf{not}\ (null\ x) \end{aligned}$$

That is, $append(x,y)$ is the empty list if both are empty, $cons\ (head\ y, tail\ y)$ (which is just y) if only x is empty, and $cons\ (head\ x, append\ (tail\ x, y))$ if neither x nor y is empty. This definition copies just the first list; in contrast, the simple unfold characterization of append

$$append = \mathsf{unfold}_{\mathsf{CList}}\ g$$

where

$$\begin{aligned} g\ (x, y) &= \mathsf{inl}\ (), && \text{if } null\ x \wedge null\ y \\ &= \mathsf{inr}\ (head\ y, (x, tail\ y)), && \text{if } null\ x \wedge \mathsf{not}\ (null\ y), \\ &= \mathsf{inr}\ (head\ x, (tail\ x, y)), && \text{if } \mathsf{not}\ (null\ x) \end{aligned}$$

copies both lists.

(a) Show that on the second summand the above unfold acts merely as the identity function:

$$\mathsf{unfold}_{\mathsf{T}}\ (f \triangledown (\mathsf{F}\ \mathsf{inr} \circ \mathsf{out}_{\mathsf{T}})) \circ \mathsf{inr} = \mathsf{id}$$

Hence $\mathsf{unfold}_{\mathsf{T}}\ (f \triangledown (\mathsf{F}\ \mathsf{inr} \circ \mathsf{out}_{\mathsf{T}})) = \mathsf{apo}_{\mathsf{T}}\ f \triangledown \mathsf{id}$.

(b) Show that the identity function is an apomorphism:

$$\mathsf{id} = \mathsf{apo}\ (\mathsf{F}\ \mathsf{inl} \circ \mathsf{out})$$

(c) Prove the (weak) fusion law for apomorphisms:

$$\mathsf{apo}\ f \circ h = \mathsf{apo}\ g \Leftarrow f \circ h = \mathsf{F}\ (h + \mathsf{id}) \circ g$$

(d) Show that any unfold is an apomorphism:

$$\mathsf{unfold}\ f = \mathsf{apo}\ (\mathsf{F}\ \mathsf{inl} \circ f)$$

(This is a generalization of Exercise 2.9.18b.)

(e) Show that any function yielding a recursive datatype can be written as an apomorphism:

$$h = \mathsf{apo}\ (\mathsf{F}\ \mathsf{inr} \circ \mathsf{out} \circ h)$$

(f) Write $ins :: \mathsf{A} \times \mathsf{CList}\ \mathsf{A} \to \mathsf{CList}\ \mathsf{A}$, which inserts a value into a sorted list, as an apomorphism.

19. Datatypes and codatatypes for the same functor are different structures, but they are not unrelated. Suppose we have the datatype definitions

$$\begin{aligned} &\mathsf{T} = \textsc{data}\ \mathsf{F} \\ &\mathsf{U} = \textsc{codata}\ \mathsf{F} \end{aligned}$$

Lambek's Lemma shows how to write $out_{\mathsf{T}} :: \mathsf{T} \to \mathsf{F}\ \mathsf{T}$, giving an F-coalgebra $(\mathsf{T}, out_{\mathsf{T}})$ and hence a function $\mathsf{unfold}_{\mathsf{U}}\ out_{\mathsf{T}} :: \mathsf{T} \to \mathsf{U}$. This function 'coerces' an element of T to the type U. Give the dual construction, expressing this coercion as a fold. Prove (in two different ways) that these two coercions are equal. Thus, we have two ways of writing the coercion from the datatype T to the codatatype U, and no way of going back again. This is what one might expect: embedding finite lists into finite-or-infinite lists is easy, but the opposite embedding is more difficult. In the following section we move to a setting in which the two types coincide, and so the coercions become the identity function.

3 Recursive datatypes in the category *Cpo*

As we observed above, the simple and elegant model of datatypes and the corresponding characterization of the 'natural patterns' of recursion over them in the category *Set* has a number of problems. We solve these problems by moving to the category *Cpo*. This category is a refinement of the category *Set*. Some structure is imposed on the objects of the category, so that they are no longer merely sets of unrelated elements, and correspondingly some structure is induced on the arrows. Some things become neater (for example, we will be able to compose unfolds and folds) but some things become messier (specifically, strictness conditions have to be attached to some of the laws).

3.1 The category *Cpo*

The category *Cpo* has as objects *pointed complete partial orders*: sets equipped with a partial order on the elements, with a least element and closed under limits of ascending chains. The arrows are *continuous* functions on these structured sets: functions which distribute over limits of ascending chains. (We will explain these notions below.)

Intuitively, we will use the partial order to represent 'approximations' in a 'definedness' or 'information' ordering: $x \sqsubseteq y$ will mean that element x is an approximation to (or less well defined than, or provides less information than) element y. Closure under limits means that we can consider complex, perhaps infinite, structures as the limit of their finite approximations, and be assured that such limits always exist. Continuity means that computations (that is, arrows) respect these limits: the behaviour of a computation on the limit of a chain of approximations can be understood purely in terms of its behaviour on each of the approximations.

3.1.1 Posets

A poset is a pair $(\mathsf{A}, \sqsubseteq)$, where A is a set and $\sqsubseteq$ is a *partial order* on A. That is, the following properties hold of $\sqsubseteq$:

reflexivity: $a \sqsubseteq a$
transitivity: $a \sqsubseteq b$ and $b \sqsubseteq c$ imply $a \sqsubseteq c$
antisymmetry: $a \sqsubseteq b$ and $b \sqsubseteq a$ imply $a = b$

The *least element* of a poset $(\mathsf{A}, \sqsubseteq)$ is the $a \in \mathsf{A}$ such that $a \sqsubseteq a'$ for all $a' \in \mathsf{A}$, if this element exists. By antisymmetry, a poset has at most one least element. The *upper bounds* in A of the poset $(\mathsf{B}, \sqsubseteq)$ where $\mathsf{B} \subseteq \mathsf{A}$ are the elements $\{a \in \mathsf{A} \mid b \sqsubseteq a \quad \text{for all } b \in \mathsf{B}\}$; note that they are elements of A, and not necessarily of B. The *least upper bound* (lub) $\bigsqcup \mathsf{B}$ in A of the poset $(\mathsf{B}, \sqsubseteq)$ where $\mathsf{B} \subseteq \mathsf{A}$ is the least element of the upper bounds in A of $(\mathsf{B}, \sqsubseteq)$, if this least element exists.

3.1.2 Cpos and pcpos

A *chain* $\langle a_i \rangle$ in a poset $(\mathsf{A}, \sqsubseteq)$ is a sequence $a_0, a_1, a_2 \ldots$ of elements in A such that $a_0 \sqsubseteq a_1 \sqsubseteq a_2 \sqsubseteq \cdots$. The lub of the chain $\langle a_i \rangle$, if it exists, is denoted $\bigsqcup_i \langle a_i \rangle$. A poset $(\mathsf{A}, \sqsubseteq)$ is a *complete partial order* (cpo) if every chain of elements in A has a lub in A. A cpo is a *pointed cpo* (pcpo) if it has a least element (which is denoted $\bot_{\mathsf{A}}$). From now on, we will often write just 'A' instead of '$(\mathsf{A}, \sqsubseteq)$' for a pcpo.

3.1.3 Strictness, monotonicity and continuity

A function $f :: \mathsf{A} \to \mathsf{B}$ between pcpos A and B is *strict* if

$$f\ \bot_{\mathsf{A}} = \bot_{\mathsf{B}}$$

A function $f :: \mathsf{A} \to \mathsf{B}$ between pcpos $(\mathsf{A}, \sqsubseteq_{\mathsf{A}})$ and $(\mathsf{B}, \sqsubseteq_{\mathsf{B}})$ is *monotonic* if

$$a \sqsubseteq_{\mathsf{A}} a' \Rightarrow f\ a \sqsubseteq_{\mathsf{B}} f\ a'$$

A monotonic function between pcpos A and B is *continuous* if

$$f\ (\textstyle\bigsqcup_i \langle a_i \rangle) = \bigsqcup_i (\langle f\ a_i \rangle)$$

3.1.4 Examples of pcpos

The following are all pcpos:

- for set A such that $\bot \notin \mathsf{A}$, the *lifted discrete* set $\{\bot\} \cup \mathsf{A}$ with ordering

  $$a \sqsubseteq b \Leftrightarrow a = \bot \ \vee \ a = b$$

- for pcpos A and B, the *cartesian product* $\{(a, b) \mid a \in \mathsf{A} \wedge b \in \mathsf{B}\}$ with ordering

  $$(a, b) \sqsubseteq (a', b') \Leftrightarrow a \sqsubseteq_{\mathsf{A}} a' \ \wedge \ b \sqsubseteq_{\mathsf{B}} b'$$

  (so the least element is $(\bot_{\mathsf{A}}, \bot_{\mathsf{B}})$);
- for pcpos A and B, the *separated sum* $\{\bot\} \cup \{(0, a) \mid a \in \mathsf{A}\} \cup \{(1, b) \mid b \in \mathsf{B}\}$ with ordering

  $$\begin{aligned} x \sqsubseteq y \Leftrightarrow\ & (x = \bot) && \vee \\ & (x = (0, a) \ \wedge \ y = (0, a') \ \wedge \ a \sqsubseteq_{\mathsf{A}} a') && \vee \\ & (x = (1, b) \ \wedge \ y = (1, b') \ \wedge \ b \sqsubseteq_{\mathsf{B}} b') \end{aligned}$$

- for pcpos A and B, the set of continuous functions from A to B, with ordering

  $$f \sqsubseteq g \Leftrightarrow (f\ a \sqsubseteq_{\mathsf{B}} g\ a \text{ for all } a \in \mathsf{A})$$

  (so the least element is the function f such that $f\ a = \bot_{\mathsf{B}}$ for any a).

3.1.5 Modelling datatypes in *Cpo*

As suggested above, the idea is that we will use pcpos to model datatypes. The elements of a pcpo model (possibly partially defined) values of that type. The ordering $\sqsubseteq$ models 'is no more defined than' or 'approximates'. For example, $(\bot, \bot) \sqsubseteq (1, \bot) \sqsubseteq (1, 2)$ and $(\bot, \bot) \sqsubseteq (\bot, 2) \sqsubseteq (1, 2)$, but $(1, \bot)$ and $(\bot, 2)$ are unrelated. 'Completely defined' values are the lubs of chains of approximations. All 'reasonable' functions are continuous, so we are justified in restricting attention just to continuous functions.

3.1.6 The category

We move from the category *Set* to the category *Cpo*. The objects $\mathsf{Obj}(\mathcal{C}po)$ are pcpos; the arrows $\mathsf{Arr}(\mathcal{C}po)$ are continuous functions between pcpos. Later, we will also use the category $\mathcal{C}po_\bot$, which has the same objects, but only the *strict* continuous functions as arrows.

3.2 Continuous algebras

Fokkinga and Meijer [11] have generalized the *Set*-based definitions of datatypes and their morphisms to *Cpo*. This provides a number of advantages over *Set*:

- we can now model *partial* functions, because all types have a least-defined element that can be used as the 'meaning' of an undefined computation;
- unfolds generate and folds consume the same kind of entity, so they can be composed to form hylomorphisms;
- we can give a meaning to arbitrary recursive definitions, not just to folds and unfolds.

(However, these advantages come at the cost of a more complex theory.) In these lectures we will only use the middle benefit of the three.

3.2.1 The main theorem

A functor F is *locally continuous* if, for all objects A and B, the action of F on functions of type $\mathsf{A} \to \mathsf{B}$ is continuous. All functors that we will be using are locally continuous.

Suppose F is a locally continuous functor on *Cpo*. Suppose also that F preserves strictness, that is, $\mathsf{F}\,f$ is strict when f is strict; so F is also a functor on $\mathcal{C}po_\bot$. Then there exists an object T, and strict functions $\mathsf{in}_\mathsf{T} :: \mathsf{F\,T} \to \mathsf{T}$ and $\mathsf{out}_\mathsf{T} :: \mathsf{T} \to \mathsf{F\,T}$, each the inverse of the other; hence T is isomorphic to $\mathsf{F\,T}$. The functor F determines T up to isomorphism, and T uniquely determines in_T and out_T. We write

$$\mathsf{T} = \textsc{fix}\ \mathsf{F}$$

The pair $(\mathsf{T}, \mathsf{in}_\mathsf{T})$ is the initial F-algebra in $\mathcal{C}po_\bot$; that is, for any type A and strict function $f :: \mathsf{F\,A} \to \mathsf{A}$, there is a unique strict h satisfying the equation

$$h \circ \mathsf{in}_\mathsf{T} = f \circ \mathsf{F}\ h$$

We write $\mathsf{fold}_\mathsf{T}\,f$ for this unique solution. It has the universal property that

$$h = \mathsf{fold}_\mathsf{T}\,f \Leftrightarrow h \circ \mathsf{in}_\mathsf{T} = f \circ \mathsf{F}\ h \qquad \text{for strict } f \text{ and } h$$

(The strictness condition on f is necessary; see Exercise 3.6.1.)

Also, the pair $(\mathsf{T}, \mathsf{out}_\mathsf{T})$ is the final F-co-algebra in $\mathcal{C}po$; that is, for any type A and (not necessarily strict) function $f :: \mathsf{A} \rightarrow \mathsf{F\,A}$, there is a unique h satisfying

$$\mathsf{out}_\mathsf{T} \circ h = \mathsf{F}\ h \circ f$$

We write $\mathsf{unfold}_\mathsf{T} f$ for this unique solution. It has the universal property (without any strictness conditions)

$$h = \mathsf{unfold}_\mathsf{T}\,f \Leftrightarrow \mathsf{out}_\mathsf{T} \circ h = \mathsf{F}\ h \circ f$$

(Apparently the strictness requirements of folds and unfolds are asymmetric. Exercise 3.6.2 shows that this apparent asymmetry is illusory.)

3.3 The pair calculus again

The cool, clear waters of the pair calculus are muddied slightly by the presence of $\perp$ and the possibility of non-strict functions. The cartesian product works fine, as before; all the same properties hold. Unfortunately, the separated sum is no longer a proper coproduct, because the injections inl and inr are non-strict, and so the equations

$$h \circ \mathsf{inl} = f \wedge h \circ \mathsf{inr} = g$$

no longer have a unique solution (because they do not specify $h \perp$). However, there is a unique *strict* solution, which is the one we take for join:

$$h = f \triangledown g \Leftrightarrow h \circ \mathsf{inl} = f \ \wedge\ h \circ \mathsf{inr} = g \ \wedge\ h \text{ strict}$$

Such strictness conditions are the price we pay for the extra power and flexibility of $\mathcal{C}po$. In view of this, we use the term ‘sum’ instead of ‘coproduct’ from now on.

3.3.1 Distributivity

Even worse than the extra strictness conditions, we no longer have a distributive category: product no longer distributes over sum. Because the function distl takes $(a, \perp)$ to $\perp$, there is no way of inverting it to retrieve the a. There is more information in $\mathsf{A} \times (\mathsf{B} + \mathsf{C})$ than in $(\mathsf{A} \times \mathsf{B}) + (\mathsf{A} \times \mathsf{C})$; now $\mathsf{distl} \circ \mathsf{undistl} = \mathsf{id}$ but $\mathsf{undistl} \circ \mathsf{distl} \sqsubseteq \mathsf{id}$. Nevertheless, we continue to use the guard $p?$, but with care: for example, the equation

$$\textsc{if}\ p\ \textsc{then}\ f\ \textsc{else}\ f = f$$

now holds only for total p (more precisely, when $p\ x = \perp$ implies $f\ x = \perp$).

3.4 Hylomorphisms

So much for the disadvantages. To compensate, we can now express the common pattern of computation of an unfold followed by a fold, because now unfolds produce and folds consume the same kind of datatype. We present two examples here: quicksort and mergesort.

3.4.1 Lists

We use the datatype

$$\mathsf{List\ A} = \textsc{fix}\ (\underline{\mathsf{1}} \mathbin{\hat{+}} (\underline{\mathsf{A}} \mathbin{\hat{\times}} \mathsf{Id}))$$

of possibly-empty lists. For brevity, we define separate constructors

$$\begin{aligned} &\mathit{nil} &&= \mathsf{in}\ (\mathsf{inl}\ ()) \\ &\mathit{cons}\ (a, x) &&= \mathsf{in}\ (\mathsf{inr}\ (a, x)) \end{aligned}$$

and destructors

$$\begin{aligned} &\mathit{isNil} &&= (\mathsf{const\ true} \triangledown \mathsf{const\ false}) \circ \mathsf{out} \\ &\mathit{head} &&= (\bot \triangledown \mathsf{exl}) \circ \mathsf{out} \\ &\mathit{tail} &&= (\bot \triangledown \mathsf{exr}) \circ \mathsf{out} \end{aligned}$$

We introduce the following syntactic sugar for folds on this type:

$$\begin{aligned} &\mathit{foldL} :: (\mathsf{B} \times (\mathsf{A} \times \mathsf{B} \to \mathsf{B})) \to \mathsf{List\ A} \to \mathsf{B} \\ &\mathit{foldL}\ (e, f) \quad\ = \mathsf{fold}_{\mathsf{List}}\ (\mathsf{const}\ e \triangledown f) \\ &\mathit{unfoldL} :: ((\mathsf{B} \to \mathsf{Bool}) \times (\mathsf{B} \to \mathsf{A} \times \mathsf{B})) \to \mathsf{B} \to \mathsf{List\ A} \\ &\mathit{unfoldL}\ (p, f) = \mathsf{unfold}_{\mathsf{List}}\ ((\mathsf{const}\ () + f) \circ p?) \end{aligned}$$

For example, concatenation on these lists is given by

$$\mathit{cat}\ (x, y) = \mathit{foldL}\ (y, \mathit{cons})\ x$$

3.4.2 Flatten

We also use the datatype

$$\mathsf{Tree\ A} = \textsc{fix}\ (\underline{\mathsf{1}} \mathbin{\hat{+}} (\underline{\mathsf{A}} \mathbin{\hat{\times}} (\mathsf{Id} \mathbin{\hat{\times}} \mathsf{Id})))$$

of internally-labelled binary trees, for which the fold may be sweetened to

$$\begin{aligned} &\mathit{foldT} :: (\mathsf{B} \times (\mathsf{A} \times (\mathsf{B} \times \mathsf{B}) \to \mathsf{B})) \to \mathsf{Tree\ A} \to \mathsf{B} \\ &\mathit{foldT}\ (e, f) = \mathsf{fold}_{\mathsf{Tree}}\ (\mathsf{const}\ e \triangledown f) \end{aligned}$$

The function *flatten* turns one of these trees into a possibly-empty list:

$$\begin{aligned} &\mathit{flatten} :: \mathsf{Tree\ A} \to \mathsf{List\ A} \\ &\mathit{flatten} &&= \mathit{foldT}\ (\mathit{nil}, \mathit{glue}) \\ &\mathit{glue}\ (a, (x, y)) &&= \mathit{cat}\ (x, \mathit{cons}\ (a, y)) \end{aligned}$$

3.4.3 Partition

The function *filter* takes a predicate p and a list x, and returns a pair of lists: those elements of x that satisfy p, and those elements of x that do not.

$$
\begin{aligned}
&\mathit{filter} :: (\mathsf{A} \to \mathsf{Bool}) \to \mathsf{List\ A} \to \mathsf{List\ A} \times \mathsf{List\ A} \\
&\mathit{filter}\ p \qquad\quad = \mathit{foldL}\ ((\mathit{nil}, \mathit{nil}), \mathit{step}) \\
&\mathit{step}\ (a, (x, y)) = (\mathit{cons}\ (a, x), y), \text{ if } p\ a \\
&\qquad\qquad\qquad\ = (x, \mathit{cons}\ (a, y)), \text{ otherwise}
\end{aligned}
$$

An alternative, point-free but perhaps less clear, definition of *step* is

$$
\begin{aligned}
\mathit{step} = \textsc{if}\ p\ \textsc{then}\ &(\mathit{cons} \circ (\mathsf{id} \times \mathsf{exl})) \vartriangle (\mathsf{exr} \circ \mathsf{exr}) \\
\textsc{else}\ \ &(\mathsf{exl} \circ \mathsf{exr}) \vartriangle (\mathit{cons} \circ (\mathsf{id} \times \mathsf{exr}))
\end{aligned}
$$

For example, we can partition a non-empty list into those elements of the tail that are less than the head, and those elements of the tail that are not:

$$
\begin{aligned}
&\mathit{partition} :: \mathsf{List\ A} \to \mathsf{List\ A} \times \mathsf{List\ A} \\
&\mathit{partition}\ x = \mathit{filter}\ (< \mathit{head}\ x)\ (\mathit{tail}\ x)
\end{aligned}
$$

3.4.4 Quicksort

The unfold on our type of trees is equivalent to

$$
\begin{aligned}
&\mathit{unfoldT} :: ((\mathsf{B} \to \mathsf{Bool}) \times (\mathsf{B} \to \mathsf{A}) \times (\mathsf{B} \to \mathsf{B} \times \mathsf{B})) \to \mathsf{B} \to \mathsf{Tree\ A} \\
&\mathit{unfoldT}\ (p, f, g) = \mathsf{unfold}_{\mathsf{Tree}}\ ((\mathsf{const}\ () + (f \vartriangle g)) \circ p?)
\end{aligned}
$$

Now we can build a binary search tree from a list:

$$
\begin{aligned}
&\mathit{buildBST} :: \mathsf{List\ A} \to \mathsf{Tree\ A} \\
&\mathit{buildBST} = \mathit{unfoldT}\ (\mathit{isNil}, \mathit{head}, \mathit{partition})
\end{aligned}
$$

(Note that *partition* is applied only to non-empty lists.) Then we can sort by building then flattening a tree:

$$
\begin{aligned}
&\mathit{quicksort} :: \mathsf{List\ A} \to \mathsf{List\ A} \\
&\mathit{quicksort} = \mathit{flatten} \circ \mathit{buildBST}
\end{aligned}
$$

This is a fold after an unfold.

3.4.5 Merge

For this example, we define the datatype

$$\mathsf{PList\ A} = \textsc{fix}\ (\underline{\mathsf{A}}\ \hat{+}\ (\underline{\mathsf{A}}\ \hat{\times}\ \mathsf{Id}))$$

of non-empty lists. Again, for brevity, we define separate destructors

$$
\begin{aligned}
&\mathit{isSing} = (\mathsf{const\ true} \triangledown \mathsf{const\ false}) \circ \mathsf{out} \\
&\mathit{hd} \qquad = (\mathsf{id} \triangledown \mathsf{exl}) \circ \mathsf{out} \\
&\mathit{tl} \qquad\ = (\bot \triangledown \mathsf{exr}) \circ \mathsf{out}
\end{aligned}
$$

We also specialize the unfold to

$$\mathit{unfoldPL} :: ((\mathsf{B} \to \mathsf{Bool}) \times (\mathsf{B} \to \mathsf{A}) \times (\mathsf{B} \to \mathsf{B})) \to \mathsf{B} \to \mathsf{PList\ A}$$

$$unfoldPL\ (p, f, g) = \mathsf{unfold}_{\mathsf{PList}}\ ((f + (f \vartriangle g)) \circ p?)$$

Then the function *merge*, which merges a pair of sorted lists into a single sorted list, is

$$\begin{aligned}
&merge :: \mathsf{PList\ A} \times \mathsf{PList\ A} \to \mathsf{PList\ A}\\
&merge = unfoldPL\ (p, f, g) \circ \mathsf{inl}
\end{aligned}$$

where

$$\begin{aligned}
&p && = \mathsf{const\ false} \triangledown isSing\\
&f && = (min \circ (hd \times hd)) \triangledown hd\\
&g\ (\mathsf{inl}\ (x, y)) && = \mathsf{inr}\ y, && \text{if } hd\ x \le hd\ y \wedge isSing\ x\\
& && = \mathsf{inl}\ (tl\ x, y), && \text{if } hd\ x \le hd\ y \wedge \mathsf{not}\ (isSing\ x)\\
& && = \mathsf{inr}\ x, && \text{if } hd\ x > hd\ y \wedge isSing\ y\\
& && = \mathsf{inl}\ (x, tl\ y), && \text{if } hd\ x > hd\ y \wedge \mathsf{not}\ (isSing\ y)\\
&g\ (\mathsf{inr}\ x) && = \mathsf{inr}\ (tl\ x)
\end{aligned}$$

and *min* is the binary minimum operator. Note that the 'state' for the unfold is either a pair of lists (which are to be merged) or a single list (which is simply to be copied). Exercise 3.6.9 concerns the characterization of *merge* as an apomorphism, whereby the single list is copied to the result 'all in one go' rather than element by element.

3.4.6 Split

Similarly, we define separate constructors

$$\begin{aligned}
&wrap\ a && = \mathsf{in}\ (\mathsf{inl}\ a)\\
&cons\ (a, x) && = \mathsf{in}\ (\mathsf{inr}\ (a, x))
\end{aligned}$$

and specialize the fold to

$$\begin{aligned}
&foldPL :: ((\mathsf{A} \to \mathsf{B}) \times (\mathsf{A} \times \mathsf{B} \to \mathsf{B})) \to \mathsf{PList\ A} \to \mathsf{B}\\
&foldPL\ (f, g) = \mathsf{fold}_{\mathsf{PList}}\ (f \triangledown g)
\end{aligned}$$

Then non-singleton lists can be split into two roughly equal halves:

$$\begin{aligned}
&split :: \mathsf{PList\ A} \to \mathsf{PList\ A} \times \mathsf{PList\ A}\\
&split\ x = foldPL\ (step, start\ (hd\ x))\ (tl\ x) \qquad \text{where}\\
&\qquad\qquad start\ a\ b \qquad\quad = (wrap\ a, wrap\ b)\\
&\qquad\qquad step\ (a, (y, z)) = (cons\ (a, z), y)
\end{aligned}$$

3.4.7 Mergesort

We also define the datatype

$$\mathsf{PTree\ A} = \textsc{fix}\ (\underline{\mathsf{A}} \mathbin{\hat{+}} (\mathsf{Id} \mathbin{\hat{\times}} \mathsf{Id}))$$

of non-empty externally-labelled binary trees. We use the specializations

$$\begin{aligned}
&foldPT :: ((\mathsf{A} \to \mathsf{B}) \times (\mathsf{B} \times \mathsf{B} \to \mathsf{B})) \to \mathsf{PTree\ A} \to \mathsf{B}\\
&foldPT\ (f, g) = \mathsf{fold}_{\mathsf{PTree}}\ (f \triangledown g)
\end{aligned}$$

of fold, and

$$\begin{array}{l} \mathit{unfoldPT} :: ((\mathsf{B} \to \mathsf{Bool}) \times (\mathsf{B} \to \mathsf{A}) \times (\mathsf{B} \to \mathsf{B} \times \mathsf{B})) \to \mathsf{B} \to \mathsf{PList\ A} \\ \mathit{unfoldPT}\ (p, f, g) = \mathsf{unfold}_{\mathsf{PTree}}\ ((f + g) \circ p?) \end{array}$$

of unfold. Then mergesort is

$$\mathit{foldPT}\ (\mathit{wrap}, \mathit{merge}) \circ \mathit{unfoldPT}\ (\mathit{isSing}, \mathit{hd}, \mathit{split})$$

(Note that *split* is applied only to non-singleton lists.)

3.5 Bibliographic notes

Complete partial orders are standard material from denotational semantics; see for example [10] for a straightforward algebraic point of view, and [33, 35] for the specifics of the applications to denotational semantics. Meijer, Fokkinga and Paterson [27] argue for the move from $\mathcal{S}et$ to $\mathcal{C}po$. The *Main Theorem* above is from [11], where it is in turn acknowledged to be another corollary of the results of Smyth and Plotkin [34] and Reynolds [32] mentioned earlier.

3.6 Exercises

1. Show that, even for strict f, the equation

   $$h \circ \mathsf{in}_\mathsf{T} = f \circ \mathsf{F}\ h$$

   may have non-strict solutions for h as well as the unique strict solution. Thus, the strictness condition on the universal property of fold in §3.2.1 is necessary.
2. Show that the categorical dual of the notion of 'strictness' vacuously holds of any function. Therefore there really is no asymmetry between the universal properties of fold and unfold in §3.2.1.
3. Show that the definitions of map as a fold (§2.4.3) and as an unfold (§2.6.3) are equal in $\mathcal{C}po$.
4. Suppose $\mathsf{T} = \textsc{fix}\ \mathsf{F}$. Let functor G be defined by $\mathsf{G\,X} = \mathsf{F}\ (\mathsf{X} \times \mathsf{T})$, and let $\mathsf{U} = \textsc{fix}\ \mathsf{G}$. Show that any paramorphism (Exercise 2.9.15) on T can be written as a hylomorphism, in the form of a fold (on U) after $\mathsf{preds}_\mathsf{T}$, where

   $$\mathsf{preds}_\mathsf{T} = \mathsf{unfold}_\mathsf{U}\ (\mathsf{F}\ (\mathsf{id} \vartriangle \mathsf{id}) \circ \mathsf{out}_\mathsf{T})$$

5. The datatype of natural numbers is $\mathsf{Nat} = \textsc{fix}\ (\underline{1}+)$. (Actually, this type necessarily includes also 'partial numbers' and one 'infinite number' as well as all the finite ones.) We can define the following syntactic sugar for the folds and unfolds:

   $$\begin{array}{ll} \mathit{foldN} & :: (\mathsf{A} \times (\mathsf{A} \to \mathsf{A})) \to \mathsf{Nat} \to \mathsf{A} \\ \mathit{foldN}\ (e, f) & = \mathsf{fold}_\mathsf{Nat}\ (\mathsf{const}\ e \triangledown f) \\ \mathit{unfoldN} & :: ((\mathsf{A} \to \mathsf{Bool}) \times (\mathsf{A} \to \mathsf{A})) \to \mathsf{A} \to \mathsf{Nat} \\ \mathit{unfoldN}\ (p, f) & = \mathsf{unfold}_\mathsf{Nat}\ ((\mathsf{const}\ () + f) \circ p?) \end{array}$$

   Informally, $\mathit{foldN}\ (e, f)\ n$ computes $f^n\ e$, by n-fold application of f to e, and $\mathit{unfoldN}\ (p, f)\ x$ returns the least n such that $f^n\ x$ satisfies p. Write addition, subtraction, multiplication, division, exponentiation and logarithms on naturals, using folds and unfolds as the only form of recursion. (Hint: define a 'predecessor' function using the destructor $\mathsf{out}_{\mathsf{Nat}}$, but make it total, taking zero to zero. You may find it easier to make division and logarithms round up rather than down.)

6. Using the datatype of lists from §3.4.1, write the insertion function

   $$ins :: 1 + (\mathsf{A} \times \mathsf{List\ A}) \to \mathsf{List\ A}$$

   as an unfold. Hence write *insertsort* using folds and unfolds as the only form of recursion.
7. Using the same datatype as in Exercise 3.6.6, write the deletion function

   $$del :: \mathsf{List\ A} \to 1 + (\mathsf{A} \times \mathsf{List\ A})$$

   as a fold. Hence write *selectsort* using folds and unfolds as the only form of recursion.
8. *Eratosthenes' Sieve* is a method for generating primes. It maintains a collection of 'candidates' as a stream, initially containing $[2, 3, \ldots]$. The first element of the collection is a prime; a new collection is obtained by deleting all multiples of that prime. Write this program using folds and unfolds on streams as the only form of recursion. (You can use *mod* on natural numbers.)
9. Write *merge* from §3.4.5 as an apomorphism rather than an unfold.
10. Show that if

    $$h = \mathsf{fold}_{\mathsf{T}}\ g \circ \mathsf{unfold}_{\mathsf{T}}\ f$$

    then

    $$h = g \circ \mathsf{F}\ h \circ f$$

    (Indeed, this is an equivalence, not just an implication; but the proof in the opposite direction requires some machinery that we have not covered.) This is a fusion law for hylomorphisms, sometimes known as *deforestation*: instead of separate unfold and fold phases, the two can be combined into a single monolithic recursion, which does not explicitly construct the intermediate data structure. The now absent datatype T is sometimes known as a *virtual data structure* [36].
11. On $\mathsf{Stream\ A} = \textsc{fix}\ (\mathsf{A}\times)$, define as an unfold a function

    $$do :: (\mathsf{A} \to \mathsf{A}) \to \mathsf{A} \to \mathsf{Stream\ A}$$

    such that $do\, s\, a$ returns the infinite stream a, $s\, a$, $s\, (s\, a)$ and so on. Also define as a fold a function $while :: (\mathsf{A} \to \mathsf{Bool}) \to \mathsf{Stream\ A} \to \mathsf{A}$ such that $while\ p\ x$ yields the first element of stream x that satisfies p. Now $while\, p \circ do\, s$ models a while loop in an imperative language. Use deforestation (Exercise 3.6.10) to calculate a function *whiledo* such that $whiledo\ (p, s) = while\ p \circ do\ s$, but which does not generate the intermediate stream.
12. Write the function *whiledo* from Exercise 3.6.11 using the folds and unfolds on naturals (Exercise 3.6.5) instead of on streams. (Hint: $whiledo\ (p, s)\ x$ applies s a certain number n of times; the number n is the least such that $s^n\ x$ fails to satisfy p.)

13. Folds and unfolds on the datatype of streams are sufficient to compute arbitrary fixpoints, so give the complete power of recursive programming. The fixpoint-finding function *fix* is defined using explicit recursion by

    $$\begin{aligned} &\mathit{fix} &&:: (\mathsf{A} \rightarrow \mathsf{A}) \rightarrow \mathsf{A} \\ &\mathit{fix}\ f &&= f\ (\mathit{fix}\ f) \end{aligned}$$

    Equivalently, given the function $\mathit{apply} :: (\mathsf{A} \rightarrow \mathsf{B}) \times \mathsf{A} \rightarrow \mathsf{B}$, it may be defined

    $$\mathit{fix}\ f = \mathit{apply}\ (f, \mathit{fix}\ f)$$

    Show that *fix* may also be defined as the composition of a stream fold (using *apply*) and a stream unfold (generating infinitely many copies of a value). Use deforestation (Exercise 3.6.10) to remove the intermediate stream, and show that this yields the explicitly recursive characterization of *fix*. (This exercise is due to Graham Hutton [20].)
14. Under certain circumstances, the post-inverse of a fold is an unfold, and the pre-inverse of an unfold is a fold:

    $$\mathsf{unfold}_\mathsf{T}\ f \circ \mathsf{fold}_\mathsf{T}\ g = \mathsf{id} \Leftarrow f \circ g = \mathsf{id}$$

    Prove this law.
15. The function *cross* takes two infinite streams of values, and returns an infinite stream containing every possible pair of values, the first component drawn from the first list and the second component drawn from the second list. The difficulty is in enumerating this two-dimensional collection in a suitable order; the standard approach is diagonalization. Define

    $$\mathit{cross} = \mathit{concat} \circ \mathit{diagonals}$$

    where

    $$\begin{aligned} &\mathit{diagonals} &&:: \mathsf{Stream\ A} \times \mathsf{Stream\ B} \rightarrow \mathsf{Stream\ (List\ (A \times B))} \\ &\mathit{concat} &&:: \mathsf{Stream\ (List\ (A \times B))} \rightarrow \mathsf{Stream\ (A \times B)} \end{aligned}$$

    Express *cross* as a hylomorphism (that is, express *diagonals* as an unfold, and *concat* as a fold). (Hint: first construct the obvious stream of streams incorporating all possible pairs. Then the 'state' of the iteration for *diagonals* consists of a pair, a finite list of those streams seen so far and a stream of streams not yet seen. At each step, strip another diagonal off from the streams seen so far, and include another stream from those not yet seen.) This example is due to Richard Bird [3].

4 Applications

We conclude these lecture notes with three more substantial examples of the concepts we have described: a simple compiler for arithmetic expressions; laws for monads and comonads; and efficient programs for breadth-first traversal of trees.

4.1 A simple compiler

In this example, we define a datatype of simple (arithmetic) expressions. We present the obvious recursive algorithm for evaluating such expressions; it turns out to be a fold. We also develop a compiler to translate such expressions into code for a stack machine; this too turns out to be a fold. Clearly, running the compiled code should be equivalent to evaluating the original expression. The proof of this fact turns out to be a straightforward application of the universal properties concerned.

4.1.1 Expressions and evaluation

We assume a datatype Op of operators. The arities of the operators are given by a function $\mathit{arity} :: \mathsf{Op} \to \mathsf{Nat}$. We also assume a datatype Val of values, and a function $\mathit{apply} :: \mathsf{Op} \times \mathsf{List}\,\mathsf{Val} \to \mathsf{Val}$ (where List is as in §3.4.1) to characterize the operators. Operator application is partial: $\mathit{apply}\ (\mathit{op}, \mathit{args})$ is defined only when $\mathit{arity}\ \mathit{op} = \mathsf{length}\ \mathit{args}$, where length computes the length of a list. Now we can define a datatype of expressions

$$\mathsf{Expr} = \textsc{fix}\ (\underline{\mathsf{Op}}\ \hat{\times}\ \mathsf{List})$$

on which evaluation, which provides the 'denotational semantics' of an expression, is simply a fold:

$$\mathit{eval} = \mathsf{fold}_{\mathsf{Expr}}\ \mathit{apply} :: \mathsf{Expr} \to \mathsf{Val}$$

4.1.2 Compilation

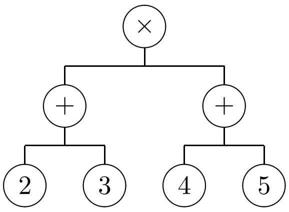
For the 'operational semantics', we assume a datatype Instr of instructions, and an encoding $\mathit{code} :: \mathsf{Op} \to \mathsf{Instr}$ of operators as instructions. Then compilation is also a fold:

$$\begin{aligned} &\mathit{compile} :: \mathsf{Expr} \to \mathsf{List}\ \mathsf{Instr} \\ &\mathit{compile} = \mathsf{fold}\ (\mathit{cons} \circ (\mathit{code} \times \mathit{concat})) \end{aligned}$$

Here, $\mathit{concat} :: \mathsf{List}\ (\mathsf{List}\ \mathsf{A}) \to \mathsf{List}\ \mathsf{A}$, and $\mathit{cons} :: \mathsf{A} \times \mathsf{List}\ \mathsf{A} \to \mathsf{List}\ \mathsf{A}$.

4.1.3 An example

For example, we might want to manipulate expressions like



We could define in Haskell

```
> data Op = Sum | Product | Num Int
> type Val = Int

> arity Sum     = 2
> arity Product = 2
> arity (Num x) = 0

> apply (Sum, [x,y])  = x+y
> apply (Product, [x,y]) = x*y
> apply (Num x, [])   = x

> data Instr = Bop ((Val,Val)->Val) | Push Val

> code Sum     = Bop (uncurry (+))
> code Product = Bop (uncurry (*))
> code (Num x) = Push x
```

and so the compiled code of the example expression will be

```
  [Bop mul, Bop add, Push 2, Push 3, Bop add, Push 4, Push 5]
    where add = uncurry (+)
          mul = uncurry (*)
```

4.1.4 Execution steps

We assume also a single-step execution function

$$exec :: \mathsf{Instr} \rightarrow \mathsf{List\ Val} \rightarrow \mathsf{List\ Val}$$

such that

$$exec\ (code\ op)\ (cat\ args\ vals) = cons\ (apply\ (op, args), vals)$$

when $arity\ op = \mathsf{length}\ args$. Continuing the example, we might have

```
> exec (Bop f) (x:y:xs) = f (x,y) : xs
> exec (Push x) xs  = x : xs
```

4.1.5 Complete execution

Now, running the program may be defined as follows:

$$\begin{aligned} &run :: \mathsf{List\ Instr} \rightarrow \mathsf{List\ Val} \rightarrow \mathsf{List\ Val} \\ &run\ nil\ vals &&= vals \\ &run\ (cons\ (instr, prog))\ vals &&= exec\ instr\ (run\ prog\ vals) \end{aligned}$$

Equivalently, discarding the last variable:

$$\begin{aligned} &\mathit{run}\ \mathit{nil} &&= \mathsf{id} \\ &\mathit{run}\ (\mathit{cons}\ (\mathit{instr}, \mathit{prog})) &&= \mathit{exec}\ \mathit{instr} \circ \mathit{run}\ \mathit{prog} \end{aligned}$$

Define $\mathit{comp}\ (f, g) = f \circ g$, and its curried version $\mathit{comp}'\ f\ g = \mathit{comp}\ (f, g)$; then

$$\mathit{run} = \mathit{foldL}'\ (\mathsf{id}, \mathit{comp}' \circ \mathit{exec})$$

(where $\mathit{foldL}' :: (\mathsf{B} \times (\mathsf{A} \to \mathsf{B} \to \mathsf{B})) \to \mathsf{List}\ \mathsf{A} \to \mathsf{B}$, using a curried function as one of its arguments). Equivalently again

$$\mathit{run} = \mathit{compose} \circ \mathsf{map}\ \mathit{exec}$$

where

$$\begin{aligned} &\mathit{compose} :: \mathsf{List}\ (\mathsf{A} \to \mathsf{A}) \to (\mathsf{A} \to \mathsf{A}) \\ &\mathit{compose} = \mathit{foldL}'\ (\mathsf{id}, \mathit{comp}') \end{aligned}$$

4.1.6 The correctness criterion

We assume that expressions are well-formed, each operator having exactly the right number of arguments. Then compiling an expression and running the resulting code on a given starting stack should have the effect of prefixing the value of that expression onto the stack:

$$\mathit{run}\ (\mathit{compile}\ \mathit{expr})\ \mathit{vals} = \mathit{cons}\ (\mathit{eval}\ \mathit{expr}, \mathit{vals})$$

Equivalently, discarding the last two variables,

$$\mathit{run} \circ \mathit{compile} = \mathit{cons}' \circ \mathit{eval}$$

where cons' is the curried version of cons.

4.1.7 Strategy

The universal property of fold on expressions is

$$\begin{aligned} &\quad h = \mathsf{fold}\, f \\ &\Leftrightarrow \\ &\quad h \circ \mathsf{in} = f \circ (\mathsf{id} \times \mathsf{map}\ h) \end{aligned}$$

We will use this universal property to show that both operational semantics $\mathit{run} \circ \mathit{compile}$ and denotational semantics $\mathit{cons}' \circ \mathit{eval}$ above are folds. We want to find an f such that

$$\mathit{run} \circ \mathit{compile} \circ \mathsf{in} = f \circ (\mathsf{id} \times \mathsf{map}\ (\mathit{run} \circ \mathit{compile}))$$

so that $\mathit{run} \circ \mathit{compile} = \mathsf{fold}\, f$. Then to complete the proof, we need only show that, for the same f,

$$\mathit{cons}' \circ \mathit{eval} \circ \mathsf{in} = f \circ (\mathsf{id} \times \mathsf{map}\ (\mathit{cons}' \circ \mathit{eval}))$$

4.1.8 Operational semantics as a fold

Now,

$$\begin{array}{ll}
& run \circ compile \circ \mathsf{in} \\
= & \quad \{\, compile = \mathsf{fold}\,(cons \circ (code \times concat)) \,\} \\
& run \circ cons \circ (code \times concat) \circ (\mathsf{id} \times \mathsf{map}\; compile) \\
= & \quad \{\, run \circ cons = comp \circ (exec \times run) \,\} \\
& comp \circ (exec \times run) \circ (code \times concat) \circ (\mathsf{id} \times \mathsf{map}\; compile) \\
= & \quad \{\, \text{pairs} \,\} \\
& comp \circ ((exec \circ code) \times (run \circ concat \circ \mathsf{map}\; compile)) \\
= & \quad \{\, run \circ concat = compose \circ \mathsf{map}\; run \,\} \\
& comp \circ ((exec \circ code) \times (compose \circ \mathsf{map}\; run \circ \mathsf{map}\; compile))
\end{array}$$

and so

$$run \circ compile = \mathsf{fold}\,(comp \circ ((exec \circ code) \times compose))$$

4.1.9 Denotational semantics as a fold

We have

$$\begin{array}{ll}
& (cons' \circ eval \circ \mathsf{in})\,(op, exprs) \\
= & \quad \{\, eval = \mathsf{fold}\; apply \,\} \\
& cons'\,(apply\,(op, \mathsf{map}\; eval\; exprs)) \\
= & \quad \{\, arity\; op = \mathsf{length}\; exprs;\ \text{requirement of } exec \,\} \\
& exec\,(code\; op) \circ cat\,(\mathsf{map}\; eval\; exprs) \\
= & \quad \{\, cat = compose \circ \mathsf{map}\; cons' \,\} \\
& exec\,(code\; op) \circ compose\,(\mathsf{map}\,(cons' \circ eval)\; exprs) \\
= & \quad \{\, \text{pairs} \,\} \\
& (comp \circ ((exec \circ code) \times (compose \circ \mathsf{map}\,(cons' \circ eval))))\,(op, exprs)
\end{array}$$

and so

$$cons' \circ eval = \mathsf{fold}\,(comp \circ ((exec \circ code) \times compose))$$

too, completing the proof.

4.2 Monads and comonads

Monads and comonads are categorical concepts; each consists of a type functor and a couple of operations that satisfy certain laws. They turn out to have useful applications in the semantics of programming languages. A monad can be used to model a *notion of computation*; in a sense, monads correspond to operational semantics. Dually, comonads correspond to denotational semantics of programming languages. But we will not get into that here. Rather, we simply observe that many constructions in functional programming are either monads or comonads, and that the proofs of the monad and comonad laws are often simple applications of the universal properties of the functors concerned. We present two simple examples, one a monad and the other a comonad.

4.2.1 Monads

A *monad* is a functor M together with two operations

$$\begin{aligned} &\mathsf{unit} :: \mathsf{A} \rightarrow \mathsf{M\ A} \\ &\mathsf{mult} :: \mathsf{M\ (M\ A)} \rightarrow \mathsf{M\ A} \end{aligned}$$

The two operations should be *natural transformations*, which is to say that the laws

$$\begin{aligned} \mathsf{unit} \circ f &= \mathsf{M}\ f \circ \mathsf{unit} \\ \mathsf{mult} \circ \mathsf{M\ (M}\ f) &= \mathsf{M}\ f \circ \mathsf{mult} \end{aligned}$$

should be satisfied. Moreover, the following 'coherence laws' relating the two operations should hold:

$$\begin{aligned} \mathsf{mult} \circ \mathsf{unit} &= \mathsf{id} \\ \mathsf{mult} \circ \mathsf{M\ unit} &= \mathsf{id} \\ \mathsf{mult} \circ \mathsf{mult} &= \mathsf{mult} \circ \mathsf{M\ mult} \end{aligned}$$

4.2.2 The list monad

Ordinary lists are one example of a monad. The datatype is defined in §3.4.1. We define the two functions

$$\begin{aligned} &wrap\ a = cons\ (a, nil) \\ &concat\ = foldL\ (nil, cat) \end{aligned}$$

We claim that List is a monad, with unit *wrap* and multiplication *concat*.

4.2.3 Laws

We must verify the following five laws:

$$\begin{aligned} wrap \circ f &= \mathsf{map}\ f \circ wrap \\ concat \circ \mathsf{map\ (map}\ f) &= \mathsf{map}\ f \circ concat \\ concat \circ wrap &= \mathsf{id} \\ concat \circ \mathsf{map}\ wrap &= \mathsf{id} \\ concat \circ concat &= concat \circ \mathsf{map}\ concat \end{aligned}$$

We address them one by one.

4.2.4 Naturality of unit

$$\begin{aligned} & (\mathsf{map}\ f \circ wrap)\ a \\ = & \qquad \{\ wrap\ \} \\ & \mathsf{map}\ f\ (cons\ (a, nil)) \\ = & \qquad \{\ \mathsf{map}\ \} \\ & cons\ (f\ a, nil) \\ = & \qquad \{\ wrap\ \} \\ & (wrap \circ f)\ a \end{aligned}$$

4.2.5 Naturality of mult

$$\begin{array}{ll}
 & \mathsf{map}\,f \circ \mathit{concat} \\
= & \quad \{\, \mathit{concat} \,\} \\
 & \mathsf{map}\,f \circ \mathit{foldL}\,(\mathit{nil}, \mathit{cat}) \\
= & \quad \{\, \text{fusion: } \mathsf{map}\,f \circ \mathit{cat} = \mathit{cat} \circ (\mathsf{map}\,f \times \mathsf{map}\,f) \,\} \\
 & \mathit{foldL}\,(\mathit{nil}, \mathit{cat} \circ (\mathsf{map}\,f \times \mathsf{id})) \\
= & \quad \{\, \text{fold-map fusion (Exercise 2.9.14)} \,\} \\
 & \mathit{foldL}\,(\mathit{nil}, \mathit{cat}) \circ \mathsf{map}\,(\mathsf{map}\,f) \\
= & \quad \{\, \mathit{concat} \,\} \\
 & \mathit{concat} \circ \mathsf{map}\,(\mathsf{map}\,f)
\end{array}$$

4.2.6 Mult-unit

$$\begin{array}{ll}
 & (\mathit{concat} \circ \mathit{wrap})\,x \\
= & \quad \{\, \mathit{wrap} \,\} \\
 & \mathit{concat}\,(\mathit{cons}\,(x, \mathit{nil})) \\
= & \quad \{\, \mathit{concat} \,\} \\
 & x \\
= & \quad \{\, \text{identity} \,\} \\
 & \mathsf{id}\,x
\end{array}$$

4.2.7 Mult-map-unit

$$\begin{array}{ll}
 & \mathit{concat} \circ \mathsf{map}\,\mathit{wrap} \\
= & \quad \{\, \text{fold-map fusion} \,\} \\
 & \mathit{foldL}\,(\mathit{nil}, \mathit{cat} \circ (\mathit{wrap} \times \mathsf{id})) \\
= & \quad \{\, \mathit{cat} \circ (\mathit{wrap} \times \mathsf{id}) = \mathit{cons} \,\} \\
 & \mathit{foldL}\,(\mathit{nil}, \mathit{cons}) \\
= & \quad \{\, \text{identity as a fold} \,\} \\
 & \mathsf{id}
\end{array}$$

4.2.8 Mult-mult

$$
\begin{aligned}
& \mathit{concat} \circ \mathit{concat} \\
= \quad & \{ \mathit{concat} \} \\
& \mathit{concat} \circ \mathit{foldL}\,(\mathit{nil}, \mathit{cat}) \\
= \quad & \{ \text{fold fusion: } \mathit{concat} \circ \mathit{cat} = \mathit{cat} \circ (\mathit{concat} \times \mathit{concat}) \} \\
& \mathit{foldL}\,(\mathit{nil}, \mathit{cat} \circ (\mathit{concat} \times \mathsf{id})) \\
= \quad & \{ \text{fold-map fusion} \} \\
& \mathit{concat} \circ \mathsf{map}\ \mathit{concat}
\end{aligned}
$$

4.2.9 Comonads

Dually, a *comonad* is a functor M together with two operations

$$
\begin{aligned}
&\mathsf{extr} :: \mathsf{M\,A} \to \mathsf{A} \\
&\mathsf{dupl} :: \mathsf{M\,A} \to \mathsf{M\,(M\,A)}
\end{aligned}
$$

Again, the two operations should be *natural transformations*:

$$
\begin{aligned}
f \circ \mathsf{extr} &= \mathsf{extr} \circ \mathsf{M}\,f \\
\mathsf{M}\,(\mathsf{M}\,f) \circ \mathsf{dupl} &= \mathsf{dupl} \circ \mathsf{M}\,f
\end{aligned}
$$

Moreover, the following coherence laws should hold:

$$
\begin{aligned}
\mathsf{extr} \circ \mathsf{dupl} &= \mathsf{id} \\
\mathsf{M\,extr} \circ \mathsf{dupl} &= \mathsf{id} \\
\mathsf{dupl} \circ \mathsf{dupl} &= \mathsf{M\,dupl} \circ \mathsf{dupl}
\end{aligned}
$$

4.2.10 The stream comonad

One example of a comonad is the datatype of streams:

$$\mathsf{Stream\,A} = \textsc{fix}\,(\underline{\mathsf{A}}\hat{\times})$$

We introduce the separate destructors

$$
\begin{aligned}
\mathit{head} &= \mathsf{exl} \circ \mathsf{out} \\
\mathit{tail} &= \mathsf{exr} \circ \mathsf{out}
\end{aligned}
$$

Thus, the function *tails*, which turns a stream into the stream of streams of all of its infinite suffices, is an unfold:

$$\mathit{tails} = \mathsf{unfold}_{\mathsf{Stream}}\,(\mathsf{id} \vartriangle \mathit{tail})$$

We claim that Stream is a comonad, with extraction *head* and duplication *tails*.

4.2.11 Laws

To say that the datatype of streams is a comonad with the above operations is to claim the following five laws:

$$\begin{array}{ll}
f \circ \mathit{head} & = \mathit{head} \circ \mathsf{map}\, f \\
\mathsf{map}\,(\mathsf{map}\, f) \circ \mathit{tails} & = \mathit{tails} \circ \mathsf{map}\, f \\
\mathit{head} \circ \mathit{tails} & = \mathsf{id} \\
\mathsf{map}\, \mathit{head} \circ \mathit{tails} & = \mathsf{id} \\
\mathit{tails} \circ \mathit{tails} & = \mathsf{map}\, \mathit{tails} \circ \mathit{tails}
\end{array}$$

We verify them one by one, below.

4.2.12 Naturality of extract

$$\begin{array}{ll}
 & \mathit{head} \circ \mathsf{map}\, f \\
= & \qquad \{\, \mathit{head} \,\} \\
 & \mathsf{exl} \circ \mathsf{out} \circ \mathsf{map}\, f \\
= & \qquad \{\, \mathsf{map} \,\} \\
 & \mathsf{exl} \circ (f \times \mathsf{map}\, f) \circ \mathsf{out} \\
= & \qquad \{\, \text{pairs} \,\} \\
 & f \circ \mathsf{exl} \circ \mathsf{out} \\
= & \qquad \{\, \mathit{head} \,\} \\
 & f \circ \mathit{head}
\end{array}$$

4.2.13 Naturality of duplicate

$$\begin{array}{ll}
 & \mathsf{map}\,(\mathsf{map}\, f) \circ \mathit{tails} \\
= & \qquad \{\, \mathit{tails} \,\} \\
 & \mathsf{map}\,(\mathsf{map}\, f) \circ \mathsf{unfold}\,(\mathsf{id} \vartriangle \mathit{tail}) \\
= & \qquad \{\, \text{map-unfold fusion (Exercise 2.9.14)} \,\} \\
 & \mathsf{unfold}\,(\mathsf{map}\, f \vartriangle \mathit{tail}) \\
= & \qquad \{\, \text{unfold fusion: } \mathsf{map}\, f \circ \mathit{tail} = \mathit{tail} \circ \mathsf{map}\, f \,\} \\
 & \mathsf{unfold}\,(\mathsf{id} \vartriangle \mathit{tail}) \circ \mathsf{map}\, f \\
= & \qquad \{\, \mathit{tails} \,\} \\
 & \mathit{tails} \circ \mathsf{map}\, f
\end{array}$$

4.2.14 Extract-duplicate

$$
\begin{array}{ll}
& \mathit{head} \circ \mathit{tails} \\
= & \qquad \{\, \mathit{head},\ \mathit{tails} \,\} \\
& \mathsf{exl} \circ \mathsf{out} \circ \mathsf{unfold}\ (\mathsf{id} \vartriangle \mathit{tail}) \\
= & \qquad \{\, \text{unfolds} \,\} \\
& \mathsf{exl} \circ (\mathsf{id} \times \mathit{tails}) \circ (\mathsf{id} \vartriangle \mathit{tail}) \\
= & \qquad \{\, \text{pairs} \,\} \\
& \mathsf{id}
\end{array}
$$

4.2.15 Map-extract-duplicate

$$
\begin{array}{ll}
& \mathsf{map}\ \mathit{head} \circ \mathit{tails} \\
= & \qquad \{\, \mathit{tails} \,\} \\
& \mathsf{map}\ \mathit{head} \circ \mathsf{unfold}\ (\mathsf{id} \vartriangle \mathit{tail}) \\
= & \qquad \{\, \text{map-unfold fusion} \,\} \\
& \mathsf{unfold}\ (\mathit{head} \vartriangle \mathit{tail}) \\
= & \qquad \{\, \text{identity as unfold} \,\} \\
& \mathsf{id}
\end{array}
$$

4.2.16 Duplicate-duplicate

$$
\begin{array}{ll}
& \mathit{tails} \circ \mathit{tails} \\
= & \qquad \{\, \mathit{tails} \,\} \\
& \mathsf{unfold}\ (\mathsf{id} \vartriangle \mathit{tail}) \circ \mathit{tails} \\
= & \qquad \{\, \text{unfold fusion: } \mathit{tail} \circ \mathit{tails} = \mathit{tails} \circ \mathit{tail} \,\} \\
& \mathsf{unfold}\ (\mathit{tails} \vartriangle \mathit{tail}) \\
= & \qquad \{\, \text{map-unfold fusion} \,\} \\
& \mathsf{map}\ \mathit{tails} \circ \mathsf{unfold}\ (\mathsf{id} \vartriangle \mathit{tail}) \\
= & \qquad \{\, \mathit{tails} \,\} \\
& \mathsf{map}\ \mathit{tails} \circ \mathit{tails}
\end{array}
$$

4.3 Breadth-first traversal

As a final example, we discuss breadth-first traversal of a tree. Depth-first traversal is an obvious program to write recursively, but breadth-first traversal takes a little more thought; one might say that it 'goes against the grain'. We present a number of algorithms, and demonstrate their equivalence.

4.3.1 Lists

Once again, we use the datatype of lists from §3.4.1. We will use the function *concat*, which concatenates a list of lists:

$$concat = foldL\ (nil, cat)$$

4.3.2 Trees

Of course, we will also require a datatype of trees:

$$\mathsf{Tree\ A} = \textsc{fix}\ (\underline{\mathsf{A}} \mathbin{\hat{\times}} \mathsf{List})$$

for which we introduce the separate destructors

$$\begin{aligned} root &= \mathsf{exl} \circ \mathsf{out} \\ kids &= \mathsf{exr} \circ \mathsf{out} \end{aligned}$$

Now, depth-first traversal is easy to write:

$$df = \mathsf{fold}\ (cons \circ (\mathsf{id} \times concat))$$

but breadth-first traversal is a little more difficult.

4.3.3 Levels

The most profitable approach to solving the problem is to split the task into two stages. The first stage computes the *levels* of tree — a list of lists, organized by level:

$$levels :: \mathsf{Tree\ A} \to \mathsf{List\ (List\ A)}$$

The second stage is to concatenate the levels. Thus,

$$bf = concat \circ levels$$

4.3.4 Long zip

The crucial component for constructing the levels of a tree is a function *lzw* (for 'long zip with'), which glues together two lists using a given binary operator:

$$lzw :: (\mathsf{A} \times \mathsf{A} \to \mathsf{A}) \to \mathsf{List\ A} \times \mathsf{List\ A} \to \mathsf{List\ A}$$

Corresponding elements are combined using the binary operator; the remaining elements are merely 'copied' to the result. The length of the result is the greater of the lengths of the arguments. We have

$$lzw\ op = unfoldL\ (p, f)$$

where

$$\begin{aligned} p\ (x, y) &= isNil\ x \wedge isNil\ y \\ f\ (x, y) &= (head\ x, (tail\ x, y)), && \text{if } isNil\ y \\ &= (head\ y, (x, tail\ y)), && \text{if } isNil\ x \\ &= (op\ (head\ x, head\ y), (tail\ x, tail\ y)), && \text{otherwise} \end{aligned}$$

(This definition is rather inefficient, as the 'remaining elements' are copied one by one. It would be better to use an apomorphism, which would allow the remainder to be copied all in one go; see Exercise 4.5.8.)

4.3.5 Levels as a fold

Now we can define level-order traversal by

$$\mathit{levels} = \mathsf{fold}\ (\mathit{cons} \circ (\mathit{wrap} \times \mathit{glue}))$$

where $\mathit{wrap}\, a = \mathit{cons}\, (a, \mathit{nil})$. Here, the function glue glues together the traversals of the children:

$$\mathit{glue} = \mathit{foldL}\ (\mathit{nil}, \mathit{lzw}\ \mathit{cat})$$

4.3.6 Levels as a fold, efficiently

The characterization of *levels* above is inefficient, because the traversals of children are re-traversed in building the traversal of the parent. We can use an *accumulating parameter* to avoid this problem. We define

$$\mathit{levels}'\ (t, \mathit{xss}) = \mathit{lzw}\ \mathit{cat}\ (\mathit{levels}\ t, \mathit{xss})$$

and so $\mathit{levels}\ t = \mathit{levels}'\ (t, \mathit{nil})$. We can now calculate (Exercise 4.5.9) that

$$\begin{aligned} &\mathit{levels}'\ (\mathsf{in}\ (a, \mathit{ts}), \mathit{xss}) = \mathit{cons}\ (\mathit{cons}\ (a, \mathit{ys}), \mathit{foldL}\ (\mathit{yss}, \mathit{levels}')\ \mathit{ts}) \\ &\qquad \text{where } (\mathit{ys}, \mathit{yss}) = \mathit{split}\ \mathit{xss} \end{aligned}$$

where

$$\begin{aligned} \mathit{split}\ \mathit{xss} &= (\mathit{nil}, \mathit{nil}), && \text{if } \mathit{isNil}\ \mathit{xss} \\ &= (\mathit{head}\ \mathit{xss}, \mathit{tail}\ \mathit{xss}), && \text{otherwise} \end{aligned}$$

With the efficient apomorphic definition of *lzw* from Exercise 4.5.8, taking time proportional to the length of the shorter argument, this program for level-order traversal takes linear time. However, it is no longer written as a fold.

4.3.7 Levels as an unfold

A better solution is to use an unfold. We generalize to the level-order traversal of a *forest*:

$$\mathit{levelsf} :: \mathsf{List}\ (\mathsf{Tree}\ \mathsf{A}) \to \mathsf{List}\ (\mathsf{List}\ \mathsf{A})$$

Again, we can calculate (using the universal property) that *levelsf* is an unfold:

$$\mathit{unfoldL}\ (\mathit{isNil}, \mathsf{map}\ \mathit{root} \vartriangle (\mathit{concat} \circ \mathsf{map}\ \mathit{kids}))$$

See Exercise 4.5.11 for the details.

4.4 Bibliographic notes

The compiler example was inspired by Hutton [19]. The application of monads to semantics is due to Moggi [28], and of comonads to Brookes [6, 7] and Turi [37]; Wadler [43, 41, 42] brought monads to the attention of functional programmers. The programs for breadth-first tree traversal are joint work with Geraint Jones [12].

4.5 Exercises

1. An alternative definition of compilation is as an unfold, from a list of expressions to a list of instructions. Define *compile* in this way, and repeat the proof of correctness of the compiler.
2. Use fold-map fusion (Exercise 2.9.14) on lists to show that

   $$\mathit{foldL}'\ (\mathsf{id}, \mathit{comp}' \circ \mathit{exec}) = \mathit{foldL}'\ (\mathsf{id}, \mathit{comp}) \circ \mathsf{map}\ \mathit{exec}$$

   (so the two definitions of *run* in §4.1.5 are indeed equivalent).
3. Show that

   $$\mathit{foldL}'\ (\mathsf{id}, f) \circ \mathit{concat} = \mathit{foldL}'\ (\mathsf{id}, f) \circ \mathsf{map}\ (\mathit{foldL}'\ (\mathsf{id}, f))$$

   when f is associative. (Hence $\mathit{run} \circ \mathit{concat} = \mathit{compose} \circ \mathsf{map}\ \mathit{run}$).
4. The compiler example would be more realistic and more general if the code for each operation were a list of instructions instead of a single instruction. Repeat the proof for this scenario.
5. The datatype of externally-labelled binary trees from §3.4.7 forms a monad, with unit operation

   $$\mathit{leaf} = \mathsf{in} \circ \mathsf{inl}$$

   and multiplication operation

   $$\mathit{collapse} = \mathsf{fold}\ (\mathsf{id} \triangledown (\mathsf{in} \circ \mathsf{inr}))$$

   Prove that the monad laws are satisfied.
6. The datatype $\mathsf{Tree\,A} = \textsc{fix}\ (\underline{\mathsf{A}} \mathbin{\hat{+}} (\underline{\mathsf{A}} \mathbin{\hat{\times}} (\mathsf{Id} \mathbin{\hat{\times}} \mathsf{Id})))$ of homogeneous binary trees forms a comonad, with extraction operation

   $$\mathit{root} = (\mathsf{id} \triangledown \mathsf{exl}) \circ \mathsf{out}$$

   and duplication operation

   $$\mathit{subs} = \mathsf{unfold}\ ((\mathit{leaf} + (\mathit{node} \vartriangle \mathsf{exr})) \circ \mathsf{out})$$

   where *leaf* and *node* are the separate constructors:

   $$\begin{aligned} \mathit{leaf} &= \mathsf{in} \circ \mathsf{inl} \\ \mathit{node} &= \mathsf{in} \circ \mathsf{inr} \end{aligned}$$

   Prove that the comonad laws are satisfied.
7. On the datatype of lists in §3.4.1 we defined concatenation of two lists as a fold

   $$\mathit{cat}\ (x, y) = \mathsf{fold}\ (\mathsf{const}\ y \triangledown (\mathsf{in} \circ \mathsf{inr}))\ x$$

   Calculate a definition as an unfold, using the universal property. Also calculate a definition as an apomorphism.
8. Calculate a definition of $\mathit{lzw}\ f$ (§4.3.4) as an apomorphism.
9. Calculate the accumulating-parameter optimization of level-order traversal, from §4.3.6.
10. The program in Exercise 4.5.9 is not a fold. However, if we define instead the curried version $\mathit{levels}''\ t\ \mathit{xss} = \mathit{levels}'\ (t, \mathit{xss})$, then levels'' is a fold. Use the universal property to calculate the f such that $\mathit{levels}'' = \mathsf{fold}\, f$.
11. Calculate the version of level-order traversal from §4.3.7 as an unfold.

12. The final program for breadth-first traversal of a forest was of the form

    $$bff = concat \circ levelsf$$

    where *concat* is a list fold and *levelsf* a list unfold. Use hylomorphism deforestation (Exercise 3.6.10) to write this as a single recursion, avoiding the intermediate generation of the list of lists. (This program was shown to us by Bernhard Möller [29]; it is interesting that it arises as a 'mere compiler optimization' from the more abstract program developed here.)
13. To most people, breadth-first traversal is related to queues, but there are no queues in the programs presented here. Show that in fact

    $$bff = unfoldL\,(null, step)$$

    where *null* holds precisely of empty forests, and *step* is defined by

    $$step\,(cons\,(t, ts)) = (root\ t, cat\,(ts, kids\ ts))$$

    (Hint: the crucial observation is that, for associative operator $\oplus$ with unit e, the equation

    $$\begin{aligned} &foldL\,(e,\oplus)\,(lzw\,(\oplus)\,(cons\,(x, xs), ys)) \\ &\quad = x \oplus foldL\,(e,\oplus)\,(lzw\,(\oplus)\,(ys, xs)) \end{aligned}$$

    holds.)

5 Bibliography

1. Roland Backhouse. An exploration of the Bird-Meertens formalism. In *International Summer School on Constructive Algorithmics, Hollum, Ameland*. STOP project, 1989. Also available as Technical Report CS 8810, Department of Computer Science, Groningen University, 1988.
2. R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors. *LNCS 669: Mathematics of Program Construction*. Springer-Verlag, 1993.
3. Richard Bird. Personal communication, 1999.
4. Richard Bird and Oege de Moor. *The Algebra of Programming*. Prentice-Hall, 1996.
5. Richard S. Bird. *Introduction to Functional Programming Using Haskell*. Prentice-Hall, 1998.
6. Stephen Brookes and Shai Geva. Computational comonads and intensional semantics. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Categories in Computer Science*, London Mathematical Society Lecture Notes, pages 1–44. Cambridge University Press, 1992. Also Technical Report CMU-CS-91-190, School of Computer Science, Carnegie Mellon University.
7. Stephen Brookes and Kathryn Van Stone. Monads and comonads in intensional semantics. Technical Report CMU-CS-93-140, CMU, 1993.
8. Rod Burstall and David Rydeheard. *Computational Category Theory*. Prentice-Hall, 1988.
9. Roy L. Crole. *Categories for Types*. Cambridge University Press, 1994.
10. B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Mathematical Textbooks Series. Cambridge University Press, 1990.
11. Maarten M. Fokkinga and Erik Meijer. Program calculation properties of continuous algebras. Technical Report CS-R9104, CWI, Amsterdam, January 1991.

12. Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 273–279, Baltimore, Maryland, September 1998.
13. J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. An introduction to categories, algebraic theories and algebras. Technical report, IBM Thomas J. Watson Research Centre, Yorktown Heights, April 1975.
14. J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24(1):68–95, January 1977.
15. Jeremy Gunawardena. Towards an applied mathematics for computer science. In M. S. Alber, B. Hu, and J. J. Rosenthal, editors, *Current and Future Directions in Applied Mathematics*. Birkhäuser, Boston, 1997.
16. Tatsuya Hagino. *A Categorical Programming Language*. PhD thesis, Department of Computer Science, University of Edinburgh, September 1987.
17. Tatsuya Hagino. A typed lambda calculus with categorical type constructors. In D. H. Pitt, A. Poigné, and D. E. Rydeheard, editors, *LNCS 283: Category Theory and Computer Science*, pages 140–157. Springer-Verlag, September 1987.
18. C. A. R. Hoare. Notes on data structuring. In Ole-Johan Dahl, Edsger W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*, APIC studies in data processing, pages 83–174. Academic Press, 1972.
19. Graham Hutton. Fold and unfold for program semantics. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, Baltimore, Maryland, September 1998.
20. Graham Hutton. Personal communication, 1999.
21. Johan Jeuring, editor. *LNCS 1422: Proceedings of Mathematics of Program Construction*, Marstrand, Sweden, June 1998. Springer-Verlag.
22. Johan Jeuring and Erik Meijer, editors. *LNCS 925: Advanced Functional Programming*. Springer-Verlag, 1995. Lecture notes from the First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden.
23. Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.
24. Grant Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, Rijksuniversiteit Groningen, September 1990.
25. Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.
26. Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.
27. Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *LNCS 523: Functional Programming Languages and Computer Architecture*, pages 124–144. Springer-Verlag, 1991.
28. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.
29. Bernhard Möller. Personal communication, 1995.
30. Bernhard Möller, editor. *LNCS 947: Mathematics of Program Construction*. Springer-Verlag, 1995.
31. Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.
32. John C. Reynolds. Semantics of the domain of flow diagrams. *Journal of the ACM*, 24(3):484–503, 1977.

33. David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, 1986.
34. M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing*, 11(4):761–783, November 1982.
35. Joseph Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.
36. Doaitse Swierstra and Oege de Moor. Virtual data structures. In Bernhard Möller, Helmut Partsch, and Steve Schumann, editors, *LNCS 755: IFIP TC2/WG2.1 State-of-the-Art Report on Formal Program Development*, pages 355–371. Springer-Verlag, 1993.
37. Daniele Turi. *Functorial Operational Semantics and its Denotational Dual.* PhD thesis, Vrije Universiteit Amsterdam, June 1996.
38. Tarmo Uustalu and Varmo Vene. Primitive (co)recursion and course-of-value (co)iteration. Research Report TRITA-IT R 98:02, Dept of Teleinformatics, Royal Institute of Technology, Stockholm, January 1998.
39. J. L. A. van de Snepscheut, editor. *LNCS 375: Mathematics of Program Construction.* Springer-Verlag, 1989.
40. Varmo Vene and Tarmo Uustalu. Functional programming with apomorphisms (corecursion). *Proceedings of the Estonian Academy of Sciences: Physics, Mathematics*, 47(3):147–161, 1998. 9th Nordic Workshop on Programming Theory.
41. Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992. Earlier version appeared in ACM Conference on Lisp and Functional Programming, 1990.
42. Philip Wadler. The essence of functional programming. In *19th Annual Symposium on Principles of Programming Languages*, 1992.
43. Philip Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the Marktoberdorf Summer School*, 1992. Also in [22].
44. R. F. C. Walters. Datatypes in distributive categories. *Bulletin of the Australian Mathematical Society*, 40:79–82, 1989.
45. R. F. C. Walters. *Categories and Computer Science.* Computer Science Texts Series. Cambridge University Press, 1991.
46. R. F. C. Walters. An imperative language based on distributive categories. *Mathematical Structures in Computer Science*, 2:249–256, 1992.

6 Appendix: Implementation in Haskell

The programs we derive in these lectures are easily translated into a lazy functional programming language such as Haskell. We present one example, the *quicksort* program from §3.4, to illustrate.

6.1 Products

We start by encoding the pair calculus. Here are products:

```
> data Prod a b = Prod a b
> exl (Prod a b) = a
```

```
> exr (Prod a b) = b

> fork :: (a -> b) -> (a -> c) -> a -> Prod b c
> fork f g a = Prod (f a) (g a)

> prod :: (a->c) -> (b->d) -> (Prod a b) -> (Prod c d)
> prod f g = fork (f . exl) (g . exr)
```

6.2 Sums

Here are the definitions for sums:

```
> data Sum a b = Inl a | Inr b

> join :: (a -> c) -> (b -> c) -> Sum a b -> c
> join l r (Inl x) = l x
> join l r (Inr y) = r y

> dsum :: (a->c) -> (b->d) -> Sum a b -> Sum c d
> dsum f g = join (Inl . f) (Inr . g)

> query :: (a -> Bool) -> a -> Sum a a
> query p a | p a = Inl a
>           | otherwise = Inr a
```

We include a function `query p` to model p? (we cannot use the names `sum` or `guard`, because they are already used in the standard Haskell prelude).

6.3 Functors

Haskell's *type classes* allow us to encode the property of being a functor. This allows us to use the same name `mapf` for the 'map' operation of any type functor. (The standard prelude defines the class `Functor` and the function `fmap` for this purpose; we simply repeat the definition with different names here.)

```
> class TypeFunctor f where
>   mapf :: (a -> b) -> (f a -> f b)
```

Actually, all we can encode is the type of the corresponding map operations; we cannot express the laws that should hold.

6.4 Datatypes

A type functor `f` induces a datatype `Fix f`; the constructor is `In` and the destructor `out`. (The difference in capitalization is an artifact of Haskell's rules for identifiers.)

```
> data TypeFunctor f => Fix f = In (f (Fix f))

> out :: TypeFunctor f => Fix f -> f (Fix f)
> out (In x) = x
```

6.5 Folds and unfolds

These are now straightforward translations:

```
> fold :: TypeFunctor f => (f a -> a) -> (Fix f -> a)
> fold f = f . mapf (fold f) . out

> unfold :: TypeFunctor f => (a -> f a) -> (a -> Fix f)
> unfold f = In . mapf (unfold f) . f
```

6.6 Lists

The encoding of a datatype is almost straightforward. The only wrinkle is that Haskell requires a type constructor identifier (`ListF` below) in order to make something an instance of a type class, so we need to introduce a function to remove this constructor too:

```
> data ListF a b = ListF (Sum () (Prod a b))
> unListF (ListF x) = x

> instance TypeFunctor (ListF a)
>   where
>     mapf f (ListF x) = ListF (dsum id (prod id f) x)
```

Now the datatype itself can be given as a mere synonym:

```
> type List a = Fix (ListF a)
```

We introduce some syntactic sugar for functions on lists:

```
> nil :: List a
> nil = In (ListF (Inl ()))
> cons :: Prod a (List a) -> List a
> cons (Prod a x) = In (ListF (Inr (Prod a x)))

> isNil :: List a -> Bool
> isNil = join (const True) (const False) . unListF . out

> hd :: List a -> a
> hd = join (error "Head of empty list") exl . unListF . out

> tl :: List a -> List a
> tl = join (error "Tail of empty list") exr . unListF . out

> foldL :: Prod b (Prod a b -> b) -> List a -> b
> foldL (Prod e f) = fold (join (const e) f . unListF)

> cat :: Prod (List a) (List a) -> List a
> cat (Prod x y) = foldL (Prod y cons) x
```

(The names `head` and `tail` are already taken in the Haskell standard prelude, for the corresponding operations on the built-in lists.)

6.7 Trees

Trees can be defined in the same way as lists:

```
> data TreeF a b = TreeF (Sum () (Prod a (Prod b b)))
> unTreeF (TreeF x) = x

> instance TypeFunctor (TreeF a)
>   where
>     mapf f (TreeF x) = TreeF (dsum id (prod id (prod f f)) x)
> type Tree a = Fix (TreeF a)

> empty :: Tree a
> empty = In (TreeF (Inl ()))
> bin :: Tree a -> a -> Tree a -> Tree a
> bin t a u = In (TreeF (Inr (Prod a (Prod t u))))

> foldT :: Prod b (Prod a (Prod b b) -> b) -> Tree a -> b
> foldT (Prod e f) = fold (join (const e) f . unTreeF)
```

```
> unfoldT :: Prod (b -> Bool) (Prod (b -> a) (b -> Prod b b))
>              -> b -> Tree a
> unfoldT (Prod p (Prod f g))
>   = unfold (TreeF . dsum (const ()) (fork f g) . query p)
```

6.8 Quicksort

The flattening stage of Quicksort encodes simply:

```
> flatten :: Tree a -> List a
> flatten = foldT (Prod nil glue)
>   where glue (Prod a (Prod x y)) = cat (Prod x (cons (Prod a y)))
```

We define filtering as follows (the name `filter` is already taken):

```
> filt :: (a -> Bool) -> List a -> Prod (List a) (List a)
> filt p = foldL (Prod (Prod nil nil) step)
>   where step = join (fork (cons . prod id exl) (exr . exr))
>                     (fork (exl . exr) (cons . prod id exr))
>                . query (p . exl)
```

The definition of `step` here is a point-free presentation of the more perspicuous definition using variable names and pattern guards:

```
    where step (Prod a (Prod x y))
            | p a       = Prod (cons (Prod a x)) y
            | otherwise = Prod x (cons (Prod a y))
```

Now partitioning a non-empty list is an application of filter:

```
> partition :: Ord a => List a -> Prod (List a) (List a)
> partition x = filt (< hd x) (tl x)
```

(The *context* '`Ord a =>`' states that this definition is only applicable to ordered types, namely those supporting the operation `<`.)

Then the remainder of the Quicksort algorithm translates naturally:

```
> build :: Ord a => List a -> Tree a
> build = unfoldT (Prod isNil (Prod hd partition))
```

```
> quicksort :: Ord a => List a -> List a
> quicksort = flatten . build
```