Origami programming 3
Jeremy Gibbons

origami (ọrigā·mi)
The Japanese art of making
elegant designs using folds
in all kinds of paper.
(From *ori* fold + *kami* paper.)

3.1 Introduction

One style of functional programming is based purely on recursive equations. Such equations are easy to explain, and adequate for any computational purpose, but hard to use well as programs get bigger and more complicated. In a sense, recursive equations are the 'assembly language' of functional programming, and direct recursion the `goto`. As computer scientists discovered in the 1960s with structured programming, it is better to identify common patterns of use of such too-powerful tools, and capture these patterns as new constructions and abstractions. In functional programming, in contrast to imperative programming, we can often express the new constructions as higher-order operations within the language, whereas the move from unstructured to structured programming entailed the development of new languages.

There are advantages in expressing programs as instances of common patterns, rather than from first principles — the same advantages as for any kind of abstraction. Essentially, one can discover general properties of the abstraction once and for all, and infer those properties of the specific instances for free. These properties may be theorems, design idioms, implementations, optimisations, and so on.

In this chapter we will look at folds and unfolds as abstractions. In a precise technical sense, folds and unfolds are the natural patterns of computation over recursive datatypes; unfolds generate data structures and folds consume them. Functional programmers are very familiar with the *foldr* function on lists, and its directional dual *foldl*; they are gradually coming to terms with the generalisation to folds on other datatypes (IFPH §3.3, §6.1.3, §6.4). The

From "The Fun of Programming" (Jeremy Gibbons and Oege de Moor, editors), Palgrave, 2003

computational duals, unfolds, are still rather unfamiliar [45]; we hope to show here that they are no more complicated than, and just as useful as, folds, and to promote a style of programming based on these and similar recursion patterns.

We explore folds and unfolds on lists, numbers and trees. In fact, a single *generic* definition of fold can be given once and for all such datatypes, and similarly for unfold. This fact has been promoted in the world of functional programming by Meijer and others [93]; for a tutorial introduction, see [44], or in a different style, [12]. However, such 'generic origami' is beyond the scope of this chapter.

3.2 Origami with lists: sorting

The most familiar datatype in functional programming is that of lists. Indeed, from one perspective, functional programming is synonymous with *lis*t *p*rocessing, as reflected in the name of the first functional programming language, LISP. Haskell has a built-in datatype of lists, with a special syntax to aid clarity and brevity. In this chapter, we will not make use of the special privileges enjoyed by lists; we will treat them instead on the same footing as every other datatype, in order to emphasise the commonalities between the different datatypes.

We will start our exploration of origami programming with some algorithms for sorting lists. As pointed out by Augusteijn [2], a number of sorting algorithms are determined purely by their recursion pattern; in some fortunate cases, once the recursion pattern has been fixed, there are essentially no further design decisions to be made. Evidently, therefore, it is important to be familiar with recursion patterns and their properties: this familiarity may lead a programmer straight to an algorithm.

Recall (IFPH §4.1.1) that lists may be defined explicitly via the following datatype declaration:

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

As a concession to Haskell's special syntax, we will define a function *wrap* for constructing singleton lists:

```
wrap   :: α → List α
wrap x = Cons x Nil
```

We also define a function *nil* for detecting empty lists:

```
nil             :: List α → Bool
nil Nil         = True
nil (Cons x xs) = False
```

Folds for lists

The natural fold for lists may be defined as follows:

$$\begin{array}{ll} \mathit{foldL} & :: (\alpha \to \beta \to \beta) \to \beta \to \mathit{List}\ \alpha \to \beta \\ \mathit{foldL}\ f\ e\ \mathit{Nil} & = e \\ \mathit{foldL}\ f\ e\ (\mathit{Cons}\ x\ \mathit{xs}) & = f\ x\ (\mathit{foldL}\ f\ e\ \mathit{xs}) \end{array}$$

This is equivalent to Haskell's *foldr* function; the '*L*' here is for 'list', not for 'left'.

The crucial fact about *foldL* is the following *universal property*:

$$\begin{array}{l} \quad h = \mathit{foldL}\ f\ e \\ \Leftrightarrow \\ \quad h\ \mathit{xs} = \mathbf{case}\ \mathit{xs}\ \mathbf{of} \\ \qquad\qquad \mathit{Nil} \to e \\ \qquad\qquad \mathit{Cons}\ y\ \mathit{ys} \to f\ y\ (h\ \mathit{ys}) \end{array}$$

(Recall that Haskell's **case** expression matches a value against each of a sequence of patterns in turn, yielding the right-hand side corresponding to the first successful match.)

Exercise 3.1 Using the universal property, prove the *fusion law*: for strict *h*,

$$\begin{array}{l} \quad h \cdot \mathit{foldL}\ f\ e = \mathit{foldL}\ f'\ e' \\ \Leftarrow \\ \quad (h\ (f\ a\ b) = f'\ a\ (h\ b)) \wedge (h\ e = e') \end{array}$$

Why does the law not hold for non-strict *h*? Where does the proof break down?
□

Exercise 3.2 Define as instances of *foldL* equivalents of the standard prelude functions *map*, ++ and *concat*:

$$\begin{array}{ll} \mathit{mapL} & :: (\alpha \to \beta) \to \mathit{List}\ \alpha \to \mathit{List}\ \beta \\ \mathit{appendL} & :: \mathit{List}\ \alpha \to \mathit{List}\ \alpha \to \mathit{List}\ \alpha \\ \mathit{concatL} & :: \mathit{List}\ (\mathit{List}\ \alpha) \to \mathit{List}\ \alpha \end{array}$$

□

Exercise 3.3 As a corollary of the general fusion law, and using your answer to Exercise 3.2, prove the *map fusion* law

$$\mathit{foldL}\ f\ e \cdot \mathit{map}\ g = \mathit{foldL}\ (f \cdot g)\ e$$

□

One classic application of *foldL* is the insertion sort algorithm [80, §5.2.1], as discussed in IFPH Exercise 4.5.4 and §5.2.4. This may be defined as follows:

```
isort :: Ord α ⇒ List α → List α
isort = foldL insert Nil
  where
    insert              :: Ord α ⇒ α → List α → List α
    insert y Nil        = wrap y
    insert y (Cons x xs)
      | y < x           = Cons y (Cons x xs)
      | otherwise       = Cons x (insert y xs)
```

IFPH defines *insert* using *takeWhile* and *dropWhile*, but we make the recursion pattern explicit so that we can study it.

Exercise 3.4 The fact that *insert y* (*Cons x xs*) sometimes requires *xs* as well as *insert y xs* means that *insert y* is difficult to write as an instance of *foldL*. However, the tupled function $insert_1$ satisfying

$$insert_1\ y\ xs = (xs,\ insert\ y\ xs)$$

can be written straightforwardly as a fold; show how. □

Exercise 3.5 As we have just seen, the value of *insert y* (*Cons x xs*) depends not only on the result *insert y xs* of a recursive call on a substructure, but also on the substructure *xs* itself. Our solution above is to define a new function that returns both of these; afterwards we can discard the one we do not want. An alternative solution is to capture this modified recursion pattern explicitly as a higher-order operator; in this case, the operator is known as a *paramorphism* [92]. In the case of lists, we could define

```
paraL                 :: (α → (List α, β) → β) → β → List α → β
paraL f e Nil         = e
paraL f e (Cons x xs) = f x (xs, paraL f e xs)
```

Here, the argument *f* takes a copy of the tail *xs* along with the result *paraL f e xs* of the recursive call on that tail. Define *insert* as an instance of *paraL*. □

Unfolds for lists

The dual of folding is unfolding. The Haskell standard *List* library defines the function *unfoldr* for generating lists:

```
unfoldr :: (β → Maybe (α, β)) → β → [α]
```

Here, an instance of the type *Maybe α* may or may not have an instance of the type *α*:

```
data Maybe α = Just α | Nothing
```

We define an equivalent of *unfoldr* for our list datatype:

$$
\begin{array}{ll}
\mathit{unfoldL'} & :: (\beta \to \mathit{Maybe}\,(\alpha,\beta)) \to \beta \to \mathit{List}\,\alpha \\
\mathit{unfoldL'}\;f\;u = & \mathbf{case}\;f\;u\;\mathbf{of} \\
 & \quad \mathit{Nothing} \to \mathit{Nil} \\
 & \quad \mathit{Just}\,(x,v) \to \mathit{Cons}\;x\;(\mathit{unfoldL'}\;f\;v)
\end{array}
$$

Sometimes it is convenient to provide the single argument of *unfoldL′* as three components: a predicate indicating when that argument should return *Nothing*, and two functions yielding the two components of the pair when it does not. The resulting function *unfoldL* takes a predicate p indicating when the seed should unfold to the empty list, and for when this fails to hold, functions f giving the head of the list and g giving the seed from which to unfold the tail:

$$
\begin{array}{ll}
\mathit{unfoldL} & :: (\beta \to \mathit{Bool}) \to (\beta \to \alpha) \to (\beta \to \beta) \to \beta \to \mathit{List}\,\alpha \\
\mathit{unfoldL}\;p\;f\;g\;b = & \mathbf{if}\;p\;b\;\mathbf{then}\;\mathit{Nil}\;\mathbf{else}\;\mathit{Cons}\;(f\;b)\;(\mathit{unfoldL}\;p\;f\;g\;(g\;b))
\end{array}
$$

Exercise 3.6 Express *unfoldL* in terms of *unfoldL′*, and vice versa. □

The crucial fact about *unfoldL* is the universal property

$$
\begin{array}{l}
\quad h = \mathit{unfoldL}\;p\;f\;g \\
\Leftrightarrow \\
\quad h\;b = \mathbf{if}\;p\;b\;\mathbf{then}\;\mathit{Nil}\;\mathbf{else}\;\mathit{Cons}\;(f\;b)\;(h\;(g\;b))
\end{array}
$$

Exercise 3.7 Using the universal property, prove the *fusion law*:

$$
\begin{array}{l}
\quad \mathit{unfoldL}\;p\;f\;g \cdot h = \mathit{unfoldL}\;p'\;f'\;g' \\
\Leftarrow \\
\quad (p \cdot h = p') \wedge (f \cdot h = f') \wedge (g \cdot h = h \cdot g')
\end{array}
$$

□

Conversely, one could define a function *foldL′* taking a single argument of type $\mathit{Maybe}\,(\alpha,\beta) \to \beta$ in place of *foldL*'s two arguments:

$$
\begin{array}{ll}
\mathit{foldL'} & :: (\mathit{Maybe}\,(\alpha,\ \beta) \to \beta) \to \mathit{List}\,\alpha \to \beta \\
\mathit{foldL'}\;f\;\mathit{Nil} & = f\;\mathit{Nothing} \\
\mathit{foldL'}\;f\;(\mathit{Cons}\;x\;xs) & = f\;(\mathit{Just}\;(x,\ \mathit{foldL'}\;f\;xs))
\end{array}
$$

These primed versions make the duality between the fold and the unfold very clear, although they may sometimes be less convenient for programming with.

Exercise 3.8 Define *foldL′* in terms of *foldL*, and vice versa. □

Exercise 3.9 The adaptation of the single-argument fold and unfold to the multi-argument interface is simplified by functions of the following types:

```
foldLargs   :: (α → β → β) → β → (Maybe (α, β) → β)
unfoldLargs :: (β → Bool) → (β → α) → (β → β) → (β → Maybe (α, β))
```

Implement these two functions. □

One sorting algorithm expressible as a list unfold is *selection sort* [80, §5.2.3], which operates by at each step removing the minimum element of the list to be sorted, but leaving the other elements in the same order. We first define the function *delmin* to do this removal:

```
delmin     :: Ord α ⇒ List α → Maybe (α, List α)
delmin Nil = Nothing
delmin xs  = Just (y, deleteL y xs)
  where
    y = minimumL xs
```

Here, *minimumL* and *deleteL* are *List* equivalents of the standard library functions *minimum* and *delete*:

```
minimumL               :: Ord α ⇒ List α → α
minimumL (Cons x xs)   = foldL min x xs

deleteL                :: Eq α ⇒ α → List α → List α
deleteL y Nil          = Nil
deleteL y (Cons x xs)
   | y == x            = xs
   | otherwise         = Cons x (deleteL y xs)
```

Then selection sort is straightforward to define:

```
ssort :: Ord α ⇒ List α → List α
ssort = unfoldL′ delmin
```

Exercise 3.10 The case *deleteL y* (*Cons x xs*) requires both the tail *xs* and the result *deleteL y xs* on that tail, so this function is another paramorphism. Redefine *deleteL* using *paraL*. □

Exercise 3.11 In fact, *delmin* itself is a paramorphism. Redefine *delmin* using *paraL* as the only form of recursion, taking care to retain the order of the remainder of the list. □

There is another sorting algorithm with a very similar form, known as *bubble sort* [80, §5.2.2]. The overall structure is the same — an unfold to lists — but the body is slightly different. The function *bubble* has (of course) the same type as *delmin*, but it does not preserve the relative order of remaining list elements. This relaxation means that it is possible to define *bubble* as a fold:

```
bubble :: Ord α ⇒ List α → Maybe (α, List α)
bubble = foldL step Nothing
  where
    step x Nothing       = Just (x, Nil)
    step x (Just (y, ys))
       | x < y           = Just (x, Cons y ys)
       | otherwise       = Just (y, Cons x ys)
```

Exercise 3.12 Of course, the type *Maybe* (α, *List* α) is equivalent to simply *List* α. Use this fact to define *bubble* to return a list instead, with the minimum element bubbled to the top; one would deconstruct this list as a final step. This definition might seem more natural to the imperative programmer handicapped by an impoverished type system. □

Given *bubble*, bubble sort is very simple to define:

```
bsort :: Ord α ⇒ List α → List α
bsort = unfoldL′ bubble
```

Exercise 3.13 In fact, the function *insert* above can be defined as an unfold. The idea is that the 'state' of the unfold consists of a pair: the list into which to insert, and *Maybe* an element to be inserted. Initially there is an element to insert; once it has been inserted, the remainder of the list is merely copied. Complete the definition, using *unfoldL′*. □

Exercise 3.14 The characterisation of *insert* as an unfold is a bit unsatisfactory, because once the correct position is found at which to insert the element, the remainder of the list must still be copied item by item. The directly recursive definition did not have this problem: one branch shares the remainder of the original list without making a recursive call. This general pattern can be captured as another recursion operator, known as an *apomorphism* [133, 132]:

```
apoL′     :: (β → Maybe (α, Either β (List α))) → β → List α
apoL′ f u = case f u of
              Nothing            → Nil
              Just (x, Left v)   → Cons x (apoL′ f v)
              Just (x, Right xs) → Cons x xs
```

For non-empty lists, the generating function f yields *Either* a new seed, on which a recursive call is made, or a complete list, which is used directly:

```
data Either α β = Left α | Right β
```

Define *insert* as an instance of *apoL′*. □

Hylomorphisms

Unfolds generate data structures, and folds consume them; it is natural to compose these two operations. The pattern of computation consisting of an unfold followed by a fold is a fairly common one; for somewhat obscure reasons, such compositions have been called *hylomorphisms* [93]. A simple example of a hylomorphism is given by the factorial function: the factorial of n is the product of the predecessors $[n, \; n-1, \ldots, 1]$ of n, and so

$$fact = foldL\ (\times)\ 1\ \cdot\ unfoldL\ (==\ 0)\ id\ pred$$

More elaborate examples of hylomorphisms (on trees) are provided by traditional compilers, which may be thought of as constructing an abstract syntax tree (unfolding to the tree type) from which to generate code (folding the abstract syntax tree).

We define

$$hyloL\ f\ e\ p\ g\ h = foldL\ f\ e\ \cdot\ unfoldL\ p\ g\ h$$

and so we have

$$fact = hyloL\ (\times)\ 1\ (==\ 0)\ id\ pred$$

Now, hylomorphisms may be fused — the intermediate list argument need never actually be constructed. This technique has been called *deforestation* [137], and may be performed automatically [46, 103] by a compiler. In general, this gives

$$hyloL\ f\ e\ p\ g\ h\ b = \mathbf{if}\ p\ b\ \mathbf{then}\ e\ \mathbf{else}\ f\ (g\ b)\ (hyloL\ f\ e\ p\ g\ h\ (h\ b))$$

and for factorial

$$fact\ n = \mathbf{if}\ n\ ==\ 0\ \mathbf{then}\ 1\ \mathbf{else}\ n \times fact\ (pred\ n)$$

as expected.

The converse pattern, of an unfold after a fold, does not seem to have received much attention. One possible application is for dealing with *structure clashes* [69], or more precisely for translation between data formats modelled as recursive datatypes. The input data structure is folded into some encapsulated form, which is then unfolded out to the output data structure. Some data compression algorithms can be expressed in terms of a fold, encoding a structured message as unstructured data, followed by an unfold, decoding the unstructured data to retrieve the message [16].

Exercise 3.15 Show how to convert from decimal (as a *String*) to binary (as a *List Bool*) with a program in the form of an unfold after a fold. □

3.3 Origami by numbers: loops

If lists are the most familiar of recursive datatypes, then natural numbers are the simplest:

$$\textbf{data}\ \mathit{Nat} = \mathit{Zero}\ \mid\ \mathit{Succ}\ \mathit{Nat}$$

Folds for naturals

Naturals, of course, have a corresponding fold:

$$\begin{aligned}
&\mathit{foldN} && :: \alpha \to (\alpha \to \alpha) \to \mathit{Nat} \to \alpha \\
&\mathit{foldN}\ z\ s\ \mathit{Zero} && = z \\
&\mathit{foldN}\ z\ s\ (\mathit{Succ}\ n) && = s\ (\mathit{foldN}\ z\ s\ n)
\end{aligned}$$

This may not look very familiar to the working functional programmer, but if we reverse the order of the three arguments we see that *foldN* is in fact an old friend, the higher-order function that applies a given function of type $\alpha \to \alpha$ a given number of times:

$$\begin{aligned}
&\mathit{iter} && :: \mathit{Nat} \to (\alpha \to \alpha) \to (\alpha \to \alpha) \\
&\mathit{iter}\ n\ f\ x && = \mathit{foldN}\ x\ f\ n
\end{aligned}$$

The function *iter n* is sometimes (IFPH §3.7) called the *Church numeral* of *n*.

Exercise 3.16 What is the single-argument version *foldN′* of *foldN*? Express each in terms of the other. □

Exercise 3.17 What is the universal property of *foldN*, and what is the fusion law? □

Exercise 3.18 Express *addN*, *mulN* and *powN* — binary operators for addition, multiplication and exponentiation on *Nat* — using *foldN*. □

Exercise 3.19 The function $\mathit{predN} :: \mathit{Nat} \to \mathit{Maybe}\ \mathit{Nat}$, which returns the predecessor of a number (or *Nothing* as the predecessor of *Zero*) is easily expressed by pattern matching:

$$\begin{aligned}
&\mathit{predN} && :: \mathit{Nat} \to \mathit{Maybe}\ \mathit{Nat} \\
&\mathit{predN}\ \mathit{Zero} && = \mathit{Nothing} \\
&\mathit{predN}\ (\mathit{Succ}\ n) && = \mathit{Just}\ n
\end{aligned}$$

Express it instead as an instance of *foldN*. In fact, this result can be generalised considerably; how? □

Exercise 3.20 Using the previous exercise, define

$$subN :: Nat \to Nat \to Maybe\ Nat$$

as an instance of *foldN*, and hence define

$$eqN, lessN :: Nat \to Nat \to Bool$$

□

Unfolds for naturals

The dual function again may look unfamiliar:

```
unfoldN'       :: (α → Maybe α) → α → Nat
unfoldN' f x =  case f x of
                  Nothing → Zero
                  Just y   → Succ (unfoldN' f y)
```

However, we can do the same here as we did for list unfolds, splitting the single argument into simpler components:

```
unfoldN        :: (α → Bool) → (α → α) → α → Nat
unfoldN p f x =  if p x then Zero else Succ (unfoldN p f (f x))
```

Then we find another old friend: this is the minimisation function from recursive function theory, which takes a predicate p, a function f, and a value x, and computes the least number n such that $p\ (iter\ n\ f\ x)$ holds.

Exercise 3.21 Express *unfoldN* in terms of *unfoldN′*, and vice versa. □

Exercise 3.22 What is the universal property of *unfoldN*, and what is the fusion law? □

Exercise 3.23 Define $divN :: Nat \to Nat \to Nat$ as an instance of *unfoldN*, or if you prefer, of *unfoldN′*. □

Exercise 3.24 Hence define $logN :: Nat \to Nat$ as an instance of *unfoldN*. (Hint: define it to round down). □

Beyond primitive recursion

Apparently, therefore, in the presence of higher-order functions, simple folds and unfolds on natural numbers provide a power beyond primitive recursion (which cannot express minimisation). Indeed, using only folds and unfolds on naturals, we can capture the full power of iterative imperative programs. The standard prelude function *until* repeatedly applies a given function of type $\alpha \to \alpha$ until the result satisfies a given predicate, and so is the functional equivalent of the `while` loop of imperative programming; it is well-known that repetition, alternation and composition of suitable primitive statements makes a complete programming language. We might define an equivalent as follows:

$$\begin{aligned}&\mathit{untilN} && :: (\alpha \to \mathit{Bool}) \to (\alpha \to \alpha) \to \alpha \to \alpha\\ &\mathit{untilN}\ p\ f\ x && = \mathit{foldN}\ x\ f\ (\mathit{unfoldN}\ p\ f\ x)\end{aligned}$$

At first sight, this appears somewhat different from the prelude's definition:

$$\begin{aligned}&\mathit{until} && :: (\alpha \to \mathit{Bool}) \to (\alpha \to \alpha) \to \alpha \to \alpha\\ &\mathit{until}\ p\ f\ x && = \mathbf{if}\ p\ x\ \mathbf{then}\ x\ \mathbf{else}\ \mathit{until}\ p\ f\ (f\ x)\end{aligned}$$

Our definition first computes the number of iterations that will be required, and then iterates the loop body that many times; the prelude's definition uses but a single loop. Nevertheless, as the following series of exercises shows, the prelude's definition arises by deforesting the number of iterations — at least, for strict f.

Exercise 3.25 Give the deforested version of the number hylomorphism

$$\begin{aligned}&\mathit{hyloN'} && :: (\mathit{Maybe}\ \alpha \to \alpha) \to (\alpha \to \mathit{Maybe}\ \alpha) \to \alpha \to \alpha\\ &\mathit{hyloN'}\ f\ g && = \mathit{foldN'}\ f \cdot \mathit{unfoldN'}\ g\end{aligned}$$

□

Exercise 3.26 The function *untilN* is not in quite the right form for deforestation, because the argument x is duplicated. We define a version that separates the two copies of the argument:

$$\begin{aligned}&\mathit{untilN}_2 && :: (\alpha \to \mathit{Bool}) \to (\alpha \to \alpha) \to \alpha \to \alpha \to \alpha\\ &\mathit{untilN}_2\ p\ f\ x\ y && = \mathit{foldN}\ x\ f\ (\mathit{unfoldN}\ p\ f\ y)\end{aligned}$$

so that

$$\mathit{untilN}\ p\ f\ x = \mathit{untilN}_2\ p\ f\ x\ x$$

Use fusion to prove that

$$\mathit{untilN}_2\ p\ f\ x\ y = \mathbf{if}\ p\ y\ \mathbf{then}\ x\ \mathbf{else}\ \mathit{untilN}_2\ p\ f\ (f\ x)\ (f\ y)$$

and hence deduce that

$$\mathit{untilN}\ p\ f\ x = \mathbf{if}\ p\ x\ \mathbf{then}\ x\ \mathbf{else}\ \mathit{untilN}\ p\ f\ (f\ x)$$

□

Exercise 3.27 How do *until* $p\ f$ and *untilN* $p\ f$ differ for non-strict f? □

Exercise 3.28 Meijer and Hutton [94] provide another explanation of computational adequacy. The fixpoint $\mathit{fix}\ f$ of a recursive function is the infinite application $f\ (f\ (\ldots))$. Show how to compute this as a list hylomorphism. □

3.4 Origami with trees: traversals

Let us now turn our attention to trees, and in particular, *rose* trees, in which every node has a list of children. We call a list of rose trees a *forest*, so the types *Rose* and *Forest* are mutually recursive:

$$\begin{aligned}&\textbf{data}\ \mathit{Rose}\ \alpha &&= \mathit{Node}\ \alpha\ (\mathit{Forest}\ \alpha)\\ &\textbf{type}\ \mathit{Forest}\ \alpha &&= \mathit{List}\ (\mathit{Rose}\ \alpha)\end{aligned}$$

(This is essentially identical to the type *Rose* in IFPH §6.4, except that we have also provided a name for forests.)

Folds for trees and forests

Since the types of trees and forests are mutually recursive, it seems 'sweetly reasonable' that the folds too should be mutually recursive:

$$\begin{aligned}&\mathit{foldR} &&:: (\alpha \to \gamma \to \beta) \to (\mathit{List}\ \beta \to \gamma) \to \mathit{Rose}\ \alpha \to \beta\\ &\mathit{foldR}\ f\ g\ (\mathit{Node}\ a\ ts) &&= f\ a\ (\mathit{foldF}\ f\ g\ ts)\end{aligned}$$

$$\begin{aligned}&\mathit{foldF} &&:: (\alpha \to \gamma \to \beta) \to (\mathit{List}\ \beta \to \gamma) \to \mathit{Forest}\ \alpha \to \gamma\\ &\mathit{foldF}\ f\ g\ ts &&= g\ (\mathit{mapL}\ (\mathit{foldR}\ f\ g)\ ts)\end{aligned}$$

Because of the mutual recursion, as well as the function each uses to make progress, it must also be given the function it must pass to its partner; therefore, both take the same pair of functions.

Exercise 3.29 IFPH §6.4 defines instead a function

$$\begin{aligned}&\mathit{foldRose} &&:: (\alpha \to \mathit{List}\ \beta \to \beta) \to \mathit{Rose}\ \alpha \to \beta\\ &\mathit{foldRose}\ f\ (\mathit{Node}\ a\ ts) &&= f\ a\ (\mathit{mapL}\ (\mathit{foldRose}\ f)\ ts)\end{aligned}$$

Show that this is equivalent to what we have defined: that is, define *foldRose* in terms of *foldR* and *foldF*, and vice versa. □

Exercise 3.30 Give the universal properties for *foldR* and *foldF*, and state and prove the corresponding fusion laws. □

Unfolds for trees and forests

Similarly, there is a mutually recursive pair of unfold functions, both taking the same functional arguments. In this case, the arguments generate from a seed a root label and a list of new seeds; the two unfolds grow from a seed a tree and a forest respectively.

$$\begin{aligned}&\mathit{unfoldR} &&:: (\beta \to \alpha) \to (\beta \to \mathit{List}\ \beta) \to \beta \to \mathit{Rose}\ \alpha\\ &\mathit{unfoldR}\ f\ g\ x &&= \mathit{Node}\ (f\ x)\ (\mathit{unfoldF}\ f\ g\ x)\end{aligned}$$

$$\begin{aligned}&\mathit{unfoldF} && :: (\beta \to \alpha) \to (\beta \to \mathit{List}\ \beta) \to \beta \to \mathit{Forest}\ \alpha\\ &\mathit{unfoldF}\ f\ g\ x && = \mathit{mapL}\ (\mathit{unfoldR}\ f\ g)\ (g\ x)\end{aligned}$$

For convenience in what follows, we define separate destructors for the root and the list of children of a tree.

$$\begin{aligned}&\mathit{root} && :: \mathit{Rose}\ \alpha \to \alpha\\ &\mathit{root}\ (\mathit{Node}\ a\ ts) && = a\end{aligned}$$

$$\begin{aligned}&\mathit{kids} && :: \mathit{Rose}\ \alpha \to \mathit{Forest}\ \alpha\\ &\mathit{kids}\ (\mathit{Node}\ a\ ts) && = ts\end{aligned}$$

Depth-first traversal

Because folds on trees and on forests are mutually recursive with the same functions as arguments, a common idiom when using them is to define the two simultaneously as a pair of functions. For example, consider performing the depth-first traversal of a tree or a forest. The traversal of a tree is one item longer than the traversal of its children; the traversal of a forest is obtained by concatenating the traversals of its trees.

```
dft          :: Rose α → List α
dff          :: Forest α → List α
(dft, dff) = (foldR f g, foldF f g)
  where
    f = Cons
    g = concatL
```

Exercise 3.31 Depth-first traversal expressed in this way is inefficient; why? A more efficient version can be calculated via fusion; how? □

Breadth-first traversal

Depth-first traversal is in a sense the natural traversal on trees; in contrast, breadth-first traversal goes ‘against the grain’. We cannot define breadth-first traversal as a fold in the same way as we did for depth-first traversal, because it is not compositional — it is not possible to construct the traversal of a forest from the traversals of its trees.

The usual implementation of breadth-first traversal in an imperative language involves queues. Queuing does not come naturally to functional programmers, although Okasaki [100] has done a lot towards rectifying that situation. In contrast, depth-first traversal is based on a stack, and stacks come for free with recursive programs.

Level-order traversal

However, one can make some progress: one can compute the *level-order traversal* [41, 42] compositionally. This yields not just a list, but a list of lists of elements, with one list for each level of the tree (see IFPH §6.4.2).

```
levelt             :: Rose α → List (List α)
levelf             :: Forest α → List (List α)
(levelt, levelf) = (foldR f g, foldF f g)
  where
    f x xss = Cons (wrap x) xss
    g       = foldL (lzw appendL) Nil
```

The level-order traversal of a forest is obtained by gluing together the traversals of its trees; two lists of lists may be glued appropriately by concatenating corresponding elements. This gluing is performed above by the function *lzw appendL* (called '*combine*' in IFPH §6.4.2). The identifier *lzw* here stands for 'long zip with'; it is like the *zipWith* function from the standard prelude, but returns a list whose length is the length of the longer argument, as opposed to that of the shorter one.

```
lzw                              :: (α → α → α) → List α → List α → List α
lzw f Nil ys                     = ys
lzw f xs Nil                     = xs
lzw f (Cons x xs) (Cons y ys) = Cons (f x y) (lzw f xs ys)
```

Exercise 3.32 Write *lzw* as an unfold. □

Exercise 3.33 While the characterisation of *lzw* as an unfold is extensionally equivalent to the directly recursive definition, it takes longer to execute — in fact, time proportional to the length of the longer argument, since it must copy the tail of the longer list. Redefine *lzw* in terms of *apoL′*, so that it takes time proportional to the length of the shorter argument instead. □

Of course, having obtained the level-order traversal of a tree or a forest, it is straightforward to obtain the breadth-first traversal: simply concatenate the levels.

```
bft :: Rose α → List α
bft = concatL · levelt

bff :: Forest α → List α
bff = concatL · levelf
```

Accumulating parameters

The naive definitions above of *levelt* and *levelf* are inefficient, because of the repeated list concatenations.

Exercise 3.34 What is the time complexity of these naive definitions of *levelt* and *levelf*? □

The standard *accumulating parameter* technique [14] can be used here. In each case, the accumulating parameter is a list of lists; the specifications of the two new functions are

$$\begin{aligned}
&\mathit{levelt}' && :: \mathit{Rose}\ \alpha \to \mathit{List}\ (\mathit{List}\ \alpha) \to \mathit{List}\ (\mathit{List}\ \alpha)\\
&\mathit{levelt}'\ t && = \mathit{lzw}\ \mathit{appendL}\ (\mathit{levelt}\ t)
\end{aligned}$$

$$\begin{aligned}
&\mathit{levelf}' && :: \mathit{Forest}\ \alpha \to \mathit{List}\ (\mathit{List}\ \alpha) \to \mathit{List}\ (\mathit{List}\ \alpha)\\
&\mathit{levelf}'\ ts && = \mathit{lzw}\ \mathit{appendL}\ (\mathit{levelf}\ ts)
\end{aligned}$$

Exercise 3.35 Derive, using the fusion laws for folds on lists and trees, efficient accumulating-parameter programs for *levelt′* and *levelf′*. □

Exercise 3.36 The programs derived in the previous exercise take linear time, provided that the efficient version of *lzw* is used, rather than the inefficient one defined as an unfold. What is the time complexity of *levelt′* and *levelf′* when the inefficient definition of *lzw* as an unfold is used? □

Exercise 3.37 In effect, the accumulating parameter optimisation above is a data refinement, using $\mathit{lzw}\ \mathit{appendL}\ \mathit{xss} :: \mathit{List}\ (\mathit{List}\ \alpha) \to \mathit{List}\ (\mathit{List}\ \alpha)$ as a 'novel representation' [65] of a lists of lists *xss*. As an alternative, use Hughes' representation of lists as list transformers directly, deriving programs of type

$$\begin{aligned}
&\mathit{levelt}'' :: \mathit{Rose}\ \alpha \to \mathit{List}\ (\mathit{List}\ \alpha \to \mathit{List}\ \alpha)\\
&\mathit{levelf}'' :: \mathit{Forest}\ \alpha \to \mathit{List}\ (\mathit{List}\ \alpha \to \mathit{List}\ \alpha)
\end{aligned}$$

What is the abstraction function for the result type $\mathit{List}\ (\mathit{List}\ \alpha \to \mathit{List}\ \alpha)$? That is, how does one turn something of this type back into a list of lists? □

Level-order traversal as an unfold

We've seen that *levelt* and *levelf* can be expressed as folds over trees and forests in a variety of ways. In fact, they can also be expressed as unfolds to lists.

Exercise 3.38 Use the universal property of *unfoldL* to derive the characterisation

$$\mathit{levelf} = \mathit{unfoldL}\ \mathit{nil}\ (\mathit{mapL}\ \mathit{root})\ (\mathit{concatL} \cdot \mathit{mapL}\ \mathit{kids})$$

of *levelf* as an unfold. How can *levelt* be expressed using unfolds? □

Exercise 3.39 We now have that *bff* is a fold (*concatL*) after an unfold (*levelf*). Using these facts, define *bff* as an instance of *hyloL*, and deforest the intermediate list of lists. □

Exercise 3.40 IFPH Exercise 6.4.5 sets as a 'difficult exercise in synthesis' the problem of deriving the standard queue-based traversal algorithm:

```
bff = unfoldL nil first rest
  where
    first (Cons t ts) = root t
    rest (Cons t ts)  = appendL ts (kids t)
```

In fact, it is a simple consequence of the universal property of *unfoldL*, given the crucial property of *lzw* that, for associative f,

$$foldL\ f\ e\ (lzw\ f\ (Cons\ x\ xs)\ ys) = f\ x\ (foldL\ f\ e\ (lzw\ f\ ys\ xs))$$

Prove this crucial property (by induction on the combined length of *xs* and *ys*), and complete the derivation. □

3.5 Other sorts of origami

We conclude this chapter with two other sorting algorithms with interesting algorithmic structure, *shell sort* and *radix sort*.

Shell sort

Insertion sort is a very simple and elegant sorting algorithm, but it is rather inefficient. When implemented in terms of arrays, the actual insertion is performed by repeated exchanges between adjacent elements; many exchanges are required to move an element a long distance. *Shell sort* [117] improves on insertion sort by allowing exchanges initially between distant elements.

Shell sort is based on *h-sorting* the list, for various values of *h*. A list is *h*-sorted if, for each value of *i*, the subsequence of elements at positions $i, i + h, i + 2h, \ldots$ is sorted. By *h*-sorting initially for some large values of *h*, it becomes easier to *h*-sort for smaller values of *h*, because out-of-order elements have a shorter distance to move. Shell sort consists of *h*-sorting for a decreasing sequence of increments *h*, ending with 1: a 1-sorted list is sorted. Good increment sequences are known, but the problem of determining an optimal increment sequence is still open [112].

In a functional setting, the simplest way to describe *h*-sorting a list is as *unravelling* the list into *h* sublists, sorting each, then *ravelling* the sublists together again.

Exercise 3.41 Ravelling and unravelling is essentially a matter of *transposition* of a list of lists. Define the function

$$trans :: List\ (List\ \alpha) \to List\ (List\ \alpha)$$

twice, once as an instance of *foldL* and once of *unfoldL*. □

Given *trans*, ravelling is simple:

$$
\begin{aligned}
&\mathit{ravel} :: \mathit{List}\ (\mathit{List}\ \alpha)\ \rightarrow\ \mathit{List}\ \alpha \\
&\mathit{ravel} = \mathit{concatL}\ \cdot\ \mathit{trans}
\end{aligned}
$$

Unravelling is a bit trickier, as it requires a pre-inverse to *concat* to split a list into consecutive sublists of a given length (the last sublist possibly being shorter).

Exercise 3.42 Define functions

$$\mathit{takeL}, \mathit{dropL} :: \mathit{Nat}\ \rightarrow\ \mathit{List}\ \alpha\ \rightarrow\ \mathit{List}\ \alpha$$

as analogues of the Haskell prelude functions *take* and *drop*. (Hint: write *takeL* as a *List* unfold, and *dropL* as a *Nat* fold.) Hence define

$$\mathit{unravel} :: \mathit{Nat}\ \rightarrow\ \mathit{List}\ \alpha\ \rightarrow\ \mathit{List}\ (\mathit{List}\ \alpha)$$

□

Exercise 3.43 Using *unravel* and *ravel*, define a function

$$\mathit{hsort} :: \mathit{Ord}\ \alpha\ \Rightarrow\ \mathit{Nat}\ \rightarrow\ \mathit{List}\ \alpha\ \rightarrow\ \mathit{List}\ \alpha$$

to perform a single *h*-sort, and a function

$$\mathit{shell} :: \mathit{Ord}\ \alpha\ \Rightarrow\ \mathit{List}\ \mathit{Nat}\ \rightarrow\ \mathit{List}\ \alpha\ \rightarrow\ \mathit{List}\ \alpha$$

to compose the *h*-sorts for each of a sequence of increments. □

Exercise 3.44 Can the *ravel* of one *hsort* and the *unravel* of the next be fused as a hylomorphism? □

Exercise 3.45 All that remains is to choose a sequence of increments. The asymptotically best known increment sequence, due to Pratt [108], consists of all increments of the form $2^p 3^q$ less than the list length N; for this sequence, Shell sort takes $\Theta(N(\log N)^2)$ time. Define using *unfoldL* a function

$$\mathit{incs} :: \mathit{Nat}\ \rightarrow\ \mathit{List}\ \mathit{Nat}$$

yielding this sequence of increments, given the length of the list to be sorted. You might want to use *Integer* rather than *Nat* for the arithmetic. (Hint: it is similar to the so-called *Hamming Problem* (IFPH §9.4.3).) □

Exercise 3.46 Although asymptotically optimal, Pratt's sequence is not competitive in practice because of constant factors. For $N < 1000$, Knuth [80, §5.2.1] recommends the sequence $h_0, h_1, \ldots$ with $h_0 = 1$, $h_{i+1} = 3h_i + 1$, stopping with the first h_{t-1} for which $3h_t \geqslant N$. Redefine *incs*, again using *unfoldL*, to yield this sequence instead. □

Now we can complete the definition of Shell sort:

$$\begin{array}{l} \mathit{shellsort} \;\; :: \mathit{Ord}\ \alpha \Rightarrow \mathit{List}\ \alpha \to \mathit{List}\ \alpha \\ \mathit{shellsort}\ \mathit{xs} = \mathit{shell}\ (\mathit{incs}\ (\mathit{lengthL}\ \mathit{xs}))\ \mathit{xs} \\ \quad \textbf{where} \\ \qquad \mathit{lengthL} = \mathit{foldL}\ (\mathit{const}\ \mathit{Succ})\ \mathit{Zero} \end{array}$$

Radix sort

In this section, we consider a different kind of sorting algorithm: one that is not based on comparisons at all. Rather, the algorithm is based on the assumption that the elements to be sorted are *records*, distinguished on the basis of a collection of *fields*, and each field is of a type that is ordered, enumerable and bounded. For simplicity, we stick to the special case that all the fields have the same type, but that is not strictly necessary. This section is based on [43].

To be more precise, we will look at two algorithms for permuting a list so that it is *lexically ordered* with respect to a list of field-extracting functions. The function *lexle* performs the comparison, given such a list and two records:

$$\begin{array}{l} \mathit{lexle} :: \mathit{Ord}\ \beta \Rightarrow \mathit{List}\ (\alpha \to \beta) \to \alpha \to \alpha \to \mathit{Bool} \\ \mathit{lexle} = \mathit{foldL}\ \mathit{step}\ (\mathit{const}\ (\mathit{const}\ \mathit{True})) \\ \quad \textbf{where} \\ \qquad \mathit{step}\ d\ f\ x\ y = (d\ x < d\ y) \vee ((d\ x == d\ y) \wedge f\ x\ y) \end{array}$$

A canonical example is where, say, the elements to be sorted are three-digit numbers, and the three fields are the three digits, most significant first.

The obvious algorithm for sorting a list of records in this way might be called *bucket sort*. It proceeds by partitioning the given list into buckets according to the first field, with one bucket for each possible value of the field. Each bucket is recursively sorted on the remaining fields, and the resulting lists are concatenated. Thus, we are effectively growing a tree of more and more refined lists, and then flattening the tree.

We will use the following type of tree:

$$\textbf{data}\ \mathit{Tree}\ \alpha = \mathit{Leaf}\ \alpha \mid \mathit{Branch}\ (\mathit{List}\ (\mathit{Tree}\ \alpha))$$

Here, the internal nodes are unlabelled, and all the elements are at the leaves. The intention is that each list of children should be non-empty, but we haven't gone to the bother here of defining a separate type of non-empty lists.

Exercise 3.47 Give definitions of the fold *foldT* and unfold *unfoldT* for this type of tree. What are the fusion laws and the hylomorphism? □

The crucial ingredient in bucket sort is the function *ptn*, which takes a *discriminator* of type $\alpha \to \beta$ (for some bounded, enumerable type β) and a list of αs, and partitions the list into a list of lists, one for each value in β:

```
ptn        :: (Bounded β, Enum β) ⇒ (α → β) → List α → List (List α)
ptn d xs = mapL (pick d xs) rng
  where
    pick          :: Enum β ⇒ (α → β) → List α → β → List α
    pick d xs m = filterL ((fromEnum m ==) · fromEnum · d) xs
```

where *rng* and *filterL* are defined below.

Exercise 3.48 Define as an instance of *unfoldL* the polymorphic constant

```
rng :: (Bounded α, Enum α) ⇒ List α
```

consisting of all the values between the lower and upper bound, inclusive. (Be careful: *maxBound* will typically not have a successor!) □

Exercise 3.49 Define as an instance of *foldL* a *List* analogue *filterL* of the standard prelude function *filter*. □

Exercise 3.50 Using *ptn*, define as an instance of *unfoldT* the function that builds a tree by repeatedly partitioning a list of records, as described above.

```
mktree :: (Bounded β, Enum β) ⇒ List (α → β) → List α → Tree (List α)
```

□

We can now complete the definition of *bucketsort*:

```
bucketsort         :: (Bounded β, Enum β) ⇒ List (α → β) → List α → List α
bucketsort ds xs = flatten (mktree ds xs)
  where
    flatten = foldT id concatL
```

Exercise 3.51 Since *flatten* is a tree fold and *mktree* a tree unfold, *bucketsort* is a hylomorphism. Use this observation to deduce that

```
bucketsort = foldL step id
  where
    step d f = concatL · mapL f · ptn d
```

□

The crucial step that gets us from here to radix sort is that *ptn* is *stable*, preserving the relative order of elements in each bucket:

Exercise 3.52 Prove that

```
mapL (bucketsort ds) · ptn d = ptn d · bucketsort ds
```

(This is not easy, but if you get stuck you can refer to [43].) Hence deduce that

```
  bucketsort = foldL step id
    where
      step d f = concatL · ptn d · f
```

□

This is the classic radix sort algorithm: only a single list of values need be maintained. In contrast, the original characterisation of *bucketsort* requires a tree (or equivalently, a stack) of lists.

Exercise 3.53 An alternative, more complicated, route to the same destination is taken in [43]. First, give an alternative definition of *mktree* as an instance of *foldL* over the list of field-extracting functions. Now use *foldL* fusion to promote the *flatten* into the fold. □

3.6 Chapter notes

Credit is owed to various colleagues for many of the examples used in this chapter: Section 3.3 arose from conversations with Graham Hutton, Section 3.4 is joint work with Geraint Jones, and Section 3.5 was inspired by a program written by Jeremy Jacob.