Part 0

—

A (Not So) Brief Introduction to Haskell

CSCI 3110 Code

Fall 2015

1 Introduction

This introduction discusses the most important aspects of programming in Haskell hopefully sufficiently well for you to understand the implementations of the different algorithms from class. I would recommend you read the first few sections and then come back to read the remaining sections when you need to, in order to understand the algorithm implementations.

2 Defining Values and Functions

In Haskell, we define values using equations:

$$x = 1$$

defines a variable x with value 1. It cannot be stressed enough that x is a variable in the mathematical sense, not in the sense of other programming languages: we cannot assign a new value to x later; it's not a reference to a memory location that can be updated.

For top-level definitions, it is customary to explicitly specify the types of values. We could specify that x is an integer:

$$x :: Int$$
$$x = 1$$

The effect of not specifying a type is the same as if we had written

$$x :: Num\ a \Rightarrow a$$
$$x = 1$$

which means that x's type cannot be identified precisely here apart from requiring that its type should support basic arithmetic operations. a is called a *type variable* and *Num* is a type class. Any type a that

belongs to the *Num* class needs to support all the basic arithmetic operations. Other type classes you will see often throughout these notes are *Eq*, *Ord*, and *Ix*. A type a belongs to class *Eq* if its values can be tested for equality; it belongs to class *Ord* if there exists an ordering of its values; and it belongs to class *Ix* if the notion of a range is defined for this type. For example, the integer type *Int* belongs to all three classes because we know how to test for equality of integers and how to compare integers, and the range $[a, b]$ refers to the set $\{a, a+1, \ldots, b\}$. You can think of type classes as similar to interfaces in Java, that is, as contracts about the type of operations a given type can be expected to support. Type variables are the equivalent of generics, only they are so pervasive in Haskell that it is more common to program with type variables than with concrete types whenever it makes sense.

Functions are values no different from integers or floats. It just happens that they can be applied to other values to produce new ones. An anonymous function is created using λ notation. For example, here is a function that adds one to its argument:

$$addOne :: Num\ a \Rightarrow a \to a$$
$$addOne = \lambda x \to x + 1$$

The type says that *addOne* is a function that can be applied to any type a in the *Num* class and returns a new value of the same type. Functions on two or more values are defined by adding more arguments:

$$add :: Num\ a \Rightarrow a \to a \to a$$
$$add = \lambda x\, y \to x + y$$

As we will see in Section **??**, *add*'s type is really a single-argument function whose result is a single-argument function:

$$add :: Num\ a \Rightarrow a \to (a \to a)$$
$$add = \lambda x \to \lambda y \to x + y$$

These two definitions are 100% equivalent, as is the following one, which is much more convenient to write:

$$add\ \ :: Num\ a \Rightarrow a \to a \to a$$
$$add\ x\, y = x + y$$

3 Local Variables and Execution Order

In most languages, computations are easier to specify if we can assign intermediate results to variables and then use these variables in subsequent computation steps. Haskell is no exception. We define local variables that are available within a function using a **let** or **where** block.

$$
\begin{aligned}
&\mathit{length} && :: \mathit{Num}\ a \Rightarrow (a,a) \to a \\
&\mathit{length}\ (x,y) &&= \mathit{sqrt}\ (xx+yy) \\
&\quad \textbf{where}\ xx &&= x \cdot x \\
&\qquad\quad\ \ yy &&= y \cdot y
\end{aligned}
$$

or

$$
\begin{aligned}
&\mathit{length} && :: \mathit{Num}\ a \Rightarrow (a,a) \to a \\
&\mathit{length}\ (x,y) &&= \textbf{let}\ xx = x \cdot x \\
&&&\qquad\ \ yy = y \cdot y \\
&&&\quad \textbf{in}\ \ \mathit{sqrt}\ (xx+yy)
\end{aligned}
$$

The former is more common in Haskell code. Just as all top-level expressions can refer to each other, so do all the expressions within the same **let** or **where** block. The expressions within such a block can also refer to all arguments of the function (x and y in this case) and to all top-level expressions.

An important point to be made is that the order in which we list the definitions of xx and yy is irrelevant. It has no impact on their evaluation order. Expressions are evaluated only if their values are needed, when they are needed. Once evaluated, the values are remembered so they don't need to be recomputed if they are used a second time. This is called "lazy evaluation", a key feature of Haskell whose benefits I will discuss more in Section **??**. A consequence is that, if we had written

$$
\begin{aligned}
&\mathit{length} && :: \mathit{Num}\ a \Rightarrow (a,a) \to a \\
&\mathit{length}\ (x,y) &&= \mathit{sqrt}\ (xx+yy) \\
&\quad \textbf{where}\ xx &&= x \cdot x \\
&\qquad\quad\ \ yy &&= y \cdot y \\
&\qquad\quad\ \ xxx &&= x \cdot x \cdot x
\end{aligned}
$$

the expression $x \cdot x \cdot x$ would never be evaluated because we never use the value xxx anywhere in our computation.

Function application is expressed without parentheses in Haskell. To apply a function f to a value x, we simply write $f\ x$, somewhat like in any standard UNIX shell. If f expects more than one argument, we pass them in order, separated by spaces: $f\ x\ y\ z$.

4 Function Composition, Currying, and Partial Application

Since functions are ordinary values in Haskell, it should not come as a surprise that there exist functions to manipulate functions. The most elementary and important one is function composition:

$$
\begin{aligned}
&(\circ) && :: (b \to a) \to (c \to b) \to (c \to a) \\
&(f \circ g)\ x &&= f\ (g\ x)
\end{aligned}
$$

So, just as in mathematics, $f \circ g$ is a function that computes its value by first applying g to its argument and then applying f to the result. The type signature also makes sense: If g is a function that computes

a value of type b from a value of type c and f computes a value of type a from a value of type b, then applying g to some value of type c and then applying f to the result produces a value of type a from a value of type c, so the type of $f \circ g$ is $c \to a$.

Function composition lets us build composite functions rather succinctly. Given a function

$$\begin{aligned} &double \quad :: Num\ a \Rightarrow a \to a \\ &double\ x = 2 \cdot x \end{aligned}$$

we could do things the pedestrian way, as in most languages:

$$\begin{aligned} &f \quad :: Num\ a \Rightarrow a \to a \\ &f\ x = double\ (addOne\ x) \end{aligned}$$

or we could just write the equivalent definition

$$\begin{aligned} &f :: Num\ a \Rightarrow a \to a \\ &f = double \circ addOne \end{aligned}$$

Combined with the initially somewhat odd seeming function application operator

$$\begin{aligned} &(\$) \quad :: (a \to b) \to a \to b \\ &f\ \$\ x = f\ x \end{aligned}$$

it also allows us to eliminate a lot of hard to read parentheses from our code. The following statements are equivalent:

$$\begin{aligned} &f\ (g\ (h\ x)) \\ &f \circ g \circ h\ \$\ x \\ &f \circ g\ \$\ h\ x \\ &f\ \$\ g\ \$\ h\ x \end{aligned}$$

Which one to choose is a bit of a matter of personal preference. I often choose whichever notation expresses the intent best.

Now back to the earlier remark about multi-argument functions. As we already said, $add :: Num\ a \Rightarrow a \to a \to a$ is really a single-argument function whose return value is another single-argument function. This is called a *curried* function. Properly parenthesized, the way we pass multiple arguments to multi-argument functions now also makes sense: $add\ 1\ 2$ is the same as $(add\ 1)\ 2$, so we apply add to 1 and then apply the resulting function to 2. What do we get if we pass only one argument to add? Well, we said that already: we get another function. So, we could have defined our function $addOne$ as

$$addOne = add\ 1$$

This type of partial function application is very useful in combination with higher-order functions, as discussed further in Section **??**.

5 Lists, Tuples, and Arrays

In most imperative languages, arrays are the default data structure for storing collections of many values. Haskell does have support for arrays, and we discuss it shortly. However, arrays, just as any other value in Haskell, are immutable. So, updating a single array cell requires us to create a whole new array with the updated cell changed. This is not efficient unless we update many values simultaneously. Thus, arrays are usually not the data structure chosen to represent sequences in Haskell. As in many functional languages, the standard data structure for representing sequences is the singly linked list because it is easy to create a new list in constant time by prepending an element to an existing list or by removing the first element from the list.

A list storing elements of type a has type $[a]$. The empty list is denoted by $[\,]$. The list obtained by prepending an element x to an existing list xs is constructed as $x : xs$, where $(:)$ is the prepend operation of type $(:) :: a \to [a] \to [a]$. Thus. we can construct the three-element list storing numbers 1, 2, and 3 as $1 : 2 : 3 : [\,]$. This is more conveniently written as a list comprehension $[1,2,3]$. For many elements, say elements 1 to 100, we can write $[1..100]$ to denote the list containing all these elements. The head of a list, that is, the first element of the list can be accessed using *head*. The tail of the list, that is, the list with the first element removed is accessed using *tail*. Using pattern matching (see Section **??**), we can specify these two functions as

```
head       :: [a] -> a
head []    = error "head is undefined for empty list"
head (x : _) = x
```

and

```
tail       :: [a] -> a
tail []    = error "tail is undefined for empty list"
tail (_ : xs) = xs
```

Thus, if ys is the empty list, then it has no head nor tail, so *head ys* and *tail ys* both raise errors. If $ys = x : xs$ for some element x and some list xs, then, as expected, *head ys* $= x$ and *tail ys* $= xs$.

Now, does it take longer to create the list $[1,2,3]$ or the list $[1..1000000]$? The answer is, it takes exactly the same amount of time, namely constant time, courtesy of Haskell's lazy evaluation. The elements in the list aren't needed yet, so why evaluate them? The cost for creating the larger list is incurred once we try to traverse it. If we define $xs = [1..1000000]$, then initially all we have is the expression $[1..1000000]$, which is not a list but rather an expression that produces a list if evaluated. If we evaluate the expression *head xs* or *tail xs* (really the only ways to look at what elements the list contains), then the expression gets replaced with the list $1 : [2..1000000]$. If knowing *head xs* or *tail xs* is all we care about, then that's it—again a constant-time operation. Only if we go on to explore the head or tail of *tail xs* does this get expanded to $1 : 2 : [3..1000000]$, and so on. As we discuss in Section **??**, this allows us to define (but never fully explore) infinite data structures. You may ask what

good does it do to be able to specify infinite data structures if we can only ever use finite portions of them. The answer is: certain things become much cleaner to express than if we didn't have this ability; see Section **??**.

While lists can be of arbitrary length but must store elements that are all of the same type, similar to integer or character arrays in C, tuples resemble C structs in that they can store a fixed number of elements, but each can be of a different type. For example, the tuple $(5, \texttt{'c'}, \mathit{False})$ can be used whenever a typle of type $(\mathit{Int}, \mathit{Char}, \mathit{Bool})$ is expected. Pairs are the most commonly used tuples in Haskell, and two useful operations for manipulating them are *fst* and *snd*, defined as:

$$\begin{aligned} &\mathit{fst} && :: (a,b) \to a \\ &\mathit{fst}\ (x,\_) && = x \end{aligned}$$

$$\begin{aligned} &\mathit{snd} && :: (a,b) \to b \\ &\mathit{snd}\ (\_,y) && = y \end{aligned}$$

Now, sometimes, we actually do want to work with arrays. As many other types, Haskell arrays are polymorphic but all the elements stored in the array must be of the same type. An array of type *Array a b* uses elements of type a as array indices and stores elements of type b. For this to work, a must belong to the type class *Ix* because the array indices must form a contiguous range such as $[1..10]$ or $[\texttt{'a'}..\texttt{'z'}]$. To create an array that stores Boolean values associated with the integers between 23 and 44 and to initialize every array entry to *False*, we'd use $\mathit{listArray}\ (23,44)\ (\mathit{repeat}\ \mathit{False})$. *repeat False* produces the infinite list $[\mathit{False}, \mathit{False}..]$:

$$\begin{aligned} &\mathit{repeat} && :: a \to [a] \\ &\mathit{repeat}\ x && = x : \mathit{repeat}\ x \end{aligned}$$

$\mathit{listArray}\ (a,b)\ xs$ expects xs to be a list of at least $b-a+1$ elements and fills in the array using the first $b-a+1$ elements from this list.

If we want to create an array by specifying each array element explicitly, we provide an association list mapping each array index to its corresponding value. For example, the following creates a three-element array mapping 1 to `'a'`, 2 to `'b'`, and 3 to `'c'`:

$$\mathit{array}\ (1,3)\ [(1,\texttt{'a'}),(3,\texttt{'c'}),(2,\texttt{'b'})]$$

If multiple entries in the association list share the same key, then the last entry with this key defines the array element. So, the third entry in the following array is `'d'`:

$$\mathit{array}\ (1,3)\ [(1,\texttt{'a'}),(3,\texttt{'c'}),(2,\texttt{'b'}),(3,\texttt{'d'})]$$

There's also an efficient way to build an array whose elements are accumulations of values in a given list. As an example, consider the problem of computing an array mapping each character between `'a'` and `'z'` to the number of its occurrences in a given text over the alphabet $[\texttt{'a'}..\texttt{'z'}]$. We can do this using the following expression:

frequencies txt = *accumArray* (+) 0 (`'a'`, `'z'`) $ *zip txt* (*repeat* 1)

In the array produced by *frequencies* `"mississippi"`, the entry for `'s'` would be 4, the entry for `'p'` would be 2, and the entry for `'z'` would be 0. So what are we doing here. First we exploit the fact that a Haskell string is just a list of characters: **type** *String* = [*Char*]. We create an infinite list of 1s using *repeat* 1 and *zip* the two lists together producing a list of type [(*Char*, *Int*)] storing a pair $(x, 1)$ for every character x in the given text. Next we create an array with index range (`'a'`, `'z'`) all of whose entries are initially 0. Then, for each entry $(x, 1)$ in *zip txt* (*repeat* 1), we lookup the array entry at position x and combine it with 1 using the function passed as the first argument to *accumArray*. Since this function is (+), we thus simply add 1 to the array entry at position x.

Now, arrays would be pretty useless if we couldn't look up things stored in the array. The element at position i in an array a is accessed using $a\,!\,i$. This is a constant-time operation. There exists an analogous lookup operation for accessing the ith element in a list l: $l\,!!\,i$. This, however, takes $\Theta(i)$ time because we have to traverse the list from the beginning to reach the ith element. We can also update an array similar to the construction of an array. For example,

listArray $(1,3)$ $[1,2,3]$ // $[(2,5),(3,4)]$

produces the same result as

listArray $(1,3)$ $[1,5,4]$

That is, $a//xs$ is the same as array a, except that every entry (i,x) in xs has the effect that the ith entry in a is changed to x. Similar to the *array* function, if multiple entries in xs share the same index i, the last entry determines the value stored in $a\,!\,i$. An important point to be made is that $a//xs$ does not *update* a; it constructs a *new array* whose entries are defined as above. If we perform n updates on an array of size n using a single application of (//), then this update cost is $\Theta(n)$, a constant cost per element. If we perform only one update on an arry of size n, the cost is still $\Theta(n)$ because an entire new array needs to be allocated and populated with the entries of the original array. This clearly is a problem, given that we use arrays for efficient lookup of dynamically changing values in many algorithms in this class. We will see in Section **??** how to use *mutable* arrays in Haskell whenever we cannot do without them.

6 Goodbye Iteration, Hello Tail Recursion

Every non-trivial program require us to repeat certain steps over and over again. In imperative languages, we do this using iteration constructs such as for- or while-loops. However, if we cannot update loop variables because variables are immutable, then we have no way to implement such loop constructs. Thus, Haskell lacks loop constructs and expects us to implement repetitive computations using recursion. Here's the usual awful example of computing the factorial of a number n:

factorial :: *Num a* ⇒ $a \to a$
factorial n = **if** $n \equiv 0$ **then** 1 **else** $n \cdot$ *factorial* $(n-1)$

The problem with this implementation is that we make lots of recursive calls if n is big. This can become a problem because each recursive call requires some stack space. However, if the result of the function is the same as the result of its final recursive call, then there is no need to allocate a stack frame. Instead, we can simply treat this final recursive call as a `goto` and effectively unroll the list of recursive calls into a loop. This is called tail call optimization, something no decent functional language can do without because of the lack of support for native iteration. So, how do we make *factorial* tail-recursive? Like this:

$$\begin{aligned}&\mathit{factorial}\ n = f\ 1\ n\\&\quad \textbf{where}\ f\ p\ x = \textbf{if}\ x \equiv 0\ \textbf{then}\ p\ \textbf{else}\ f\ (p \cdot x)\ (x-1)\end{aligned}$$

factorial calls f once, and f either returns its first argument directly or determines its result by calling itself recursively with arguments $p \cdot x$ and $x-1$. No extra work is to be done after the call $f\ (p \cdot x)\ (x-1)$, so this can be transformed into a loop by the compiler.

7 Pattern Matching

Often, code becomes much more readable by analyzing function arguments using patterns rather than if-statements. For example, our *factorial* function would be clearer if we could simply write $\mathit{factorial}\ 0 = 1$; for any other value x, $\mathit{factorial}\ x = x \cdot \mathit{factorial}\ (x-1)$. Well, we can do exactly this in Haskell:

$$\begin{aligned}&\mathit{factorial}\ 0 = 1\\&\mathit{factorial}\ x = x \cdot \mathit{factorial}\ (x-1)\end{aligned}$$

Whenever multiple equations defining a function are found, the first one matching the function arguments is applied. Thus, if we call *factorial* 0, the first equation matches and the result is 1. For *factorial* 2, we would use the second equation and determine that $\mathit{factorial}\ 2 = 2 \cdot \mathit{factorial}\ 1$. If we had written

$$\begin{aligned}&\mathit{factorial}\ x = x \cdot \mathit{factorial}\ (x-1)\\&\mathit{factorial}\ 0 = 1\end{aligned}$$

the first pattern would always match and applying the function would send us into an infinite loop; the order of the equations matters. Of course, we can also clean up our tail-recursive implementation of *factorial* using pattern matching:

$$\begin{aligned}&\mathit{factorial}\ n = f\ 1\ n\\&\quad \textbf{where}\ f\ p\ 0 = p\\&\qquad\quad f\ p\ x = f\ (p \cdot x)\ (x-1)\end{aligned}$$

Patterns are even more useful for decomposing lists. Consider the definition of *head* we gave earlier:

$$\begin{aligned}&\mathit{head} \quad\quad :: [a] \to a\\&\mathit{head}\ [\,] \quad = \mathit{error}\ \texttt{"head is undefined for empty list"}\\&\mathit{head}\ (x : \_) = x\end{aligned}$$

Here the first equation applies only if the argument is the empty list. The second equation applies if the list is composed of a head and a tail. Since we do not care what that tail looks like, we used the wildcard _ to avoid explicitly naming that tail. If we want to specify different equations for different scenarios that cannot be distinguished by pattern matching alone, pattern guards often help. Let's assume we want to compute the product of the elements in a list but every odd element should be rounded up to the next even element before multiplying with it. Here's how we could do this

$$\begin{array}{lll} \mathit{evenProduct} & :: \mathit{Num}\ a \Rightarrow a \to a & \\ \mathit{evenProduct}\ [\,] & = 1 & \\ \mathit{evenProduct}\ (x : \mathit{xs}) & \mid \mathit{even}\ x = x & \cdot\ \mathit{evenProduct}\ \mathit{xs} \\ & \mid \mathit{odd}\ \ x = (x+1) & \cdot\ \mathit{evenProduct}\ \mathit{xs} \end{array}$$

or using the pattern guards' version of an else-clause:

$$\begin{array}{lll} \mathit{evenProduct} & :: \mathit{Num}\ a \Rightarrow a \to a & \\ \mathit{evenProduct}\ [\,] & = 1 & \\ \mathit{evenProduct}\ (x : \mathit{xs}) & \mid \mathit{even}\ x \quad = x & \cdot\ \mathit{evenProduct}\ \mathit{xs} \\ & \mid \mathit{otherwise} = (x+1) & \cdot\ \mathit{evenProduct}\ \mathit{xs} \end{array}$$

So, using the standard pattern matching rules, we decide that the first equation should apply if we're given the empty list: $\mathit{evenProduct}\ [\,] = 1$. Otherwise, we split the list into its head x and tail xs. The pattern guards associate two possible equations with this case. If x is even, then $\mathit{evenProduct}\ (x : \mathit{xs}) = x \cdot \mathit{evenProduct}\ \mathit{xs}$. Otherwise, $\mathit{evenProduct}\ (x : \mathit{xs}) = (x+1) \cdot \mathit{evenProduct}\ \mathit{xs}$.

8 Higher-Order Functions and Basic List Operations

Since functions are ordinary values in Haskell, they can be passed as function arguments and returned as function results (the latter is exactly what we do anyway with curried functions). Functions that take functions as arguments and/or return functions as results are called *higher-order functions*. They are useful because they can be used to abstract common behaviours that differ only in some small details, which are then captured by providing appropriate functions as arguments to the higher-order function. To illustrate this using a number of common examples, I discuss a number of standard functions from Haskell's extensive list processing library.

Iteration in imperative languages is usually used to implement a number of common program patterns that are not all that apparent from the imperative code but can be encapsulated quite nicely in higher-order functions. The first pattern is mapping a function over a list:

$$\mathit{map} :: (a \to b) \to [\,a\,] \to [\,b\,]$$

As the type signature suggests, $\mathit{map}\ f\ \mathit{xs}$ produces a new list by applying the function f to each element of xs. For example,

$$\mathit{map}\ (\cdot 2)\ [\,1,2,3,4,5\,] \equiv [\,2,4,6,8,10\,]$$

There's nothing magical about *map*. It can be defined as

$$\begin{aligned} &map\ \_\ [\,] &&= [\,] \\ &map\ f\ \ (x : xs) &&= f\ x : map\ f\ xs \end{aligned}$$

Note that *map* is not tail-recursive. Nevertheless, it won't build up a huge stack because $map\ f\ (x : xs)$ doesn't do anything unless we actually want to inspect the elements of the resulting list—remember, Haskell is lazy. To inspect the first element of the list, we only need to evaluate $f\ x$, to get the tail, we don't really need to do anything other than remembering that the elements of the tail can be generated using $map\ f\ xs$ if we need them, and so on. So, $map\ xs$ doesn't do anything immediately, and subsequently, we do constant work to produce each element of the list when we need it, if we need it.

The next pattern is combining the elements of two lists into a list of pairs:

$$\begin{aligned} &zip &&:: [a] \to [b] \to [(a,b)] \\ &zip\ [\,] \quad\ \_ &&= [\,] \\ &zip\ \_ \quad\ [\,] &&= [\,] \\ &zip\ (x : xs)\ (y : xs) &&= (x,y) : zip\ xs\ ys \end{aligned}$$

Note that this definition implies that the list produced by $zip\ xs\ ys$ has length $min\ |xs|\ |ys|$, so the remaining elements of the longer list are simply discarded. This can be very useful, as we will see later.

Often, we don't just want to group the elements of two lists into pairs. Rather, we want to combine the elements of the two lists using an arbitrary function. This is similar to *map*, but *map* applies a single-argument function to a single list. To combine two lists using a two-argument function, we need

$$zipWith :: (a \to b \to c) \to [a] \to [b] \to [c]$$

If we didn't use *zip* to define *zipWith* below, we could have defined *zip* using *zipWith* as

$$zip = zipWith\ (,)$$

where $(,) :: a \to b \to (a,b)$ is the two-argument function producing a pair of elements consisting of its two arguments.

Instead, we define *zipWith* in terms of *zip* and *map*:

$$zipWith\ f = map\ (uncurry\ f) \circ zip$$

where

$$\begin{aligned} &uncurry &&:: (a \to b \to c) \to (a,b) \to c \\ &uncurry\ f\ (x,y) &&= f\ x\ y \end{aligned}$$

and its inverse is

$$\begin{aligned} &curry &&:: ((a,b) \to c) \to a \to b \to c \\ &curry\ f\ x\ y &&= f\ (x,y) \end{aligned}$$

We can use *zipWith* to sum the elements in two lists together, for example:

$$zipWith\ (+)\ [1,3,5]\ [2,4,6] \equiv [3,7,11]$$

Another common pattern is folding: reducing the elements in a list to a single value using an appropriate accumulation function. This comes in two variants, depending on whether we want to be able to deal with empty lists and whether the final value we want to produce is of the same type as the list elements. The most general version, allowing for different element and result types and for empty lists is implemented using *foldl* or *foldr*:

$$\begin{aligned} &foldl :: (b \to a \to b) \to b \to [a] \to b \\ &foldr :: (a \to b \to b) \to b \to [a] \to b \end{aligned}$$

The difference is the direction in which the list is traversed to accumulate the elements: *foldl* goes forward, left to right:

$$\begin{aligned} &foldl\ \_\ a\ [\,] \quad\quad = a \\ &foldl\ f\ \ a\ (x : xs) = foldl\ f\ (f\ a\ x)\ xs \end{aligned}$$

foldr traverses the list in reverse:

$$\begin{aligned} &foldr\ \_\ a\ [\,] \quad\quad = a \\ &foldr\ f\ \ a\ (x : xs) = f\ x\ (foldr\ f\ a\ xs) \end{aligned}$$

The results may be different if f is not associative. If f is associative, then a strict version of *foldl*, named $foldl'$, is more space efficient because it does not build up a linear-size stack as *foldr* does. *foldl* also does not build up such a stack, as it is obviously tail-recursive, but it builds up a linear-size expression instead that is only collapsed into a single value once this value is used. $foldl'$ does this collapsing along the way and thus never uses more than constant space assuming the evaluation of f itself or the result type do not use more than constant space.

We could, for example, define two standard functions Σ and Π for computing the sum and the product of a list of numbers using $foldl'$ (*foldl* or *foldr* would also work but would be less efficient):

$$\begin{aligned} &\Sigma :: Num\ a \Rightarrow [a] \to a \\ &\Sigma = foldl'\ (+)\ 0 \end{aligned}$$

$$\begin{aligned} &\Pi :: Num\ a \Rightarrow [a] \to a \\ &\Pi = foldl'\ (\cdot)\ 1 \end{aligned}$$

Sometimes, there is no meaningful value that can be associated with the empty list when folding. For example, let us say we want to compute the minimum of a list of elements:

$$minimum :: Ord\ a \Rightarrow [a] \to a$$

This function has two useful properties that help us out to generalize the underlying computational pattern: First, its result type is the same as the type of the elements in the list. Second, if the argument

list has only one element, the result is simply this element. This computational pattern is captured by the *foldl1* and *foldr1* functions:

$$\begin{aligned}&\mathit{foldl1} :: (a \to a \to a) \to [a] \to a\\&\mathit{foldr1} :: (a \to a \to a) \to [a] \to a\end{aligned}$$

foldl1 can in fact be defined in terms of its more general counterpart *foldl*:

$$\mathit{foldl1}\ f\ (x : xs) = \mathit{foldl}\ f\ x\ xs$$

foldr1 is best implemented from scratch:

$$\begin{aligned}&\mathit{foldr1}\ \_\ [x] \quad = x\\&\mathit{foldr1}\ f\ \ (x : xs) = f\ x\ (\mathit{foldr1}\ f\ xs)\end{aligned}$$

minimum can now be defined as

$$\mathit{minimum} = \mathit{foldl1}\ \mathit{min}$$

The final common pattern to discuss here is the opposite of folding: unfolding. The idea is to produce a list from a single seed value. Since this seed value can itself be a list or some other complicated structure, unfolding can be a powerful tool. Here's a contrived simple example: generating the list $[n, n-1\,..\,1]$ from the input parameter n without using a list comprehension:

$$\begin{aligned}&\mathit{makeList} :: \mathit{Int} \to [\mathit{Int}]\\&\mathit{makeList} = \mathit{unfoldr}\ (\lambda x \to \textbf{if}\ x \equiv 0\ \textbf{then}\ \mathit{Nothing}\ \textbf{else}\ \mathit{Just}\ (x, x-1))\end{aligned}$$

unfoldr is defined as follows:

$$\begin{aligned}&\mathit{unfoldr} \quad :: (b \to \mathit{Maybe}\ (a, b)) \to b \to [a]\\&\mathit{unfoldr}\ f\ x = \textbf{case}\ f\ x\ \textbf{of}\\&\qquad\qquad\quad \mathit{Nothing} \quad \to [\,]\\&\qquad\qquad\quad \mathit{Just}\ (y, x') \to y : \mathit{unfoldr}\ f\ x'\end{aligned}$$

Thus, the given function f takes the current seed value and returns *Nothing* if the list should end. Otherwise, it returns *Just* (y, x'), where y is the next element in the list and x' is the seed value to be used to generate the tail of the list.

Here's a more interesting example: splitting a list into sublists of length k, for some parameter k.

$$\begin{aligned}&\mathit{kPartition} \quad :: \mathit{Int} \to [a] \to [[a]]\\&\mathit{kPartition}\ k = \mathit{unfoldr}\ (\lambda xs \to \textbf{if}\ \mathit{null}\ xs\ \textbf{then}\ \mathit{Nothing}\ \textbf{else}\ \mathit{Just}\ (\mathit{splitAt}\ k\ xs))\end{aligned}$$

where $\mathit{splitAt}\ k\ xs = (\mathit{take}\ k\ xs, \mathit{drop}\ k\ xs)$. $\mathit{take}\ k\ xs$ returns the list containing the first k elements of xs. $\mathit{drop}\ k\ xs$ returns the list obtained by removing the first k elements from xs.

More useful list functions, given without their definition here are

$$
\begin{array}{ll}
\mathit{elem} & :: \mathit{Eq}\ a \Rightarrow a \to [a] \to \mathit{Bool} \\
\mathit{find} & :: (a \to \mathit{Bool}) \to [a] \to \mathit{Maybe}\ a \\
\mathit{break} & :: (a \to \mathit{Bool}) \to [a] \to ([a],[a]) \\
\mathit{span} & :: (a \to \mathit{Bool}) \to [a] \to ([a],[a]) \\
\mathit{partition} & :: (a \to \mathit{Bool}) \to [a] \to ([a],[a]) \\
\mathit{inits} & :: [a] \to [[a]] \\
\mathit{tails} & :: [a] \to [[a]]
\end{array}
$$

$\mathit{elem}\ x\ xs = \mathit{True}$ if and only if $x \in xs$. $\mathit{find}\ p\ xs$ returns Just the first element x in xs such that $p\ x \equiv \mathit{True}$ if such an element exists; otherwise it returns $\mathit{Nothing}$. $\mathit{break}\ p\ xs$ returns the pair (ys, zs) such that $xs = ys +\!\!+ zs$, $p\ y \equiv \mathit{False}$ for all $y \in ys$ and either $zs \equiv [\,]$ or $p\ (\mathit{head}\ zs) \equiv \mathit{True}$. In other words, break scans the list to find the first element that satisfies the predicate p and splits the list into the prefix preceding this element and the remainder of the list. span does the opposite, that is, it can be defined as $\mathit{span}\ p = \mathit{break}\ (\neg \circ p)$. $\mathit{partition}\ p\ xs = (ys, zs)$ where ys is the list of all elements $x \in xs$ such that $p\ x \equiv \mathit{True}$ and zs is the list of all elements $x \in xs$ such that $p\ x \equiv \mathit{False}$. Finally, $\mathit{inits}\ xs$ and $\mathit{tails}\ xs$ return the list of all prefixes and suffixes of xs. Examples:

$$
\begin{array}{lll}
\mathit{elem}\ 5\ [1,7,5,8] & \equiv \mathit{True} \\
\mathit{elem}\ 5\ [1,7,8] & \equiv \mathit{False} \\
\mathit{find}\ \mathit{even}\ [1,2,3,4] & \equiv \mathit{Just}\ 2 \\
\mathit{find}\ \mathit{even}\ [1,3,5] & \equiv \mathit{Nothing} \\
\mathit{break}\ \mathit{even}\ [1,2,3,4] & \equiv ([1],[2,3,4]) \\
\mathit{break}\ \mathit{even}\ [1,3,5] & \equiv ([\,],[1,3,5]) \\
\mathit{span}\ \mathit{even}\ [1,2,3,4] & \equiv ([\,],[1,2,3,4]) \\
\mathit{span}\ \mathit{odd}\ [1,2,3,4] & \equiv ([1],[2,3,4]) \\
\mathit{partition}\ \mathit{even}\ [1,2,3,4] & \equiv ([2,4],[1,3]) \\
\mathit{inits}\ [1,2,3] & \equiv [[\,],[1],[1,2],[1,2,3]] \\
\mathit{tails}\ [1,2,3] & \equiv [[1,2,3],[2,3],[3],[\,]]
\end{array}
$$

9 Lazy Evaluation and Infinite Data Structures

As already pointed out before, Haskell evaluates expressions lazily: they are evaluated only if and when their values are needed. Thus, defining a really long list takes constant time. The cost of determining the contents of the list is incurred only once we inspect those contents. Laziness has a number of benefits, three of which I discuss here. Remember, however, that it also comes at a price: instead of storing a plain value, the program needs to store a pointer to a *thunk*, which either simply returns this value (if it has been evaluated before) or computes the value and then stores it in case it is requested again. This incurs a constant overhead.[1]

[1] If the compiler can figure out that some value will be used pretty much right away, then it doesn't generate a thunk but rather a plain old value, so there is no cost to laziness because, well, there is none in this case. The compiler is surprisingly

First take the notion of "really long" to the limit and you get "infinite". $[1..]$ denotes the list of all positive integers. If we tried to inspect all the elements in this list using *putStrLn* (*show* $[1..]$), this would take forever, literally. On the other hand, taking any finite prefix of this list takes time linear in the length of the prefix. For example, *take* $5\,[1..]$ would expand the expression $[1..]$ to $1:2:3:4:5:[6..]$ and collect the expanded elements in the finite list $[1,2,3,4,5]$, leaving the tail $[6..]$ unevaluated. What is this good for?, you may ask. Consider the problem of numbering the elements of a finite list *xs* in their order of appearance in *xs*:

$$number :: [a] \to [(a, Int)]$$

The pedestrian implementation close to the solution we would have to choose in most imperative languages would look like this:

$$\begin{aligned} &number && = nGo\ 0 \\ &\quad \textbf{where}\ nGo\ \_\ [\,] && = [\,] \\ &\qquad\quad nGo\ i\ \ (x:xs) && = (x,i) : nGo\ (i+1)\ xs \end{aligned}$$

Since we already know about higher-order functions and in particular about Haskell's *zip* function, we could get a bit more functional and obtain the solution we would have to choose in a functional language with strict evaluation:

$$\begin{aligned} &number\ xs = zip\ xs\ [1..n] \\ &\quad \textbf{where}\ n = |xs| \end{aligned}$$

However, this is both inelegant and inefficient because the list xs is traversed twice, once to compute its length and again to zip it with $[1..n]$. What's worse, between the computation of $|xs|$ and zipping *xs* with $[1..n]$, it would have to be stored explicitly, which could cause real memory problems if the list is long. The standard Haskell solution is to write

$$number\ xs = zip\ xs\ [1..]$$

This works just fine because the length of the list produced by *zip* is the minimum of the lengths of the input lists; if *xs* is finite, only a finite prefix of $[1..]$ is ever evaluated. Moreover, the elements of *zip xs* $[1..]$ are produced one by one, on demand, from the elements of *xs* and $[1..]$, which in turn can be produced one by one, on demand. Thus, assuming all we do is scan through these lists, without caching list elements for later reuse, it allows us to process even very long lists using little memory.

The second benefit of lazy evaluation is that we can define recursive data structures that rely on their own contents, as long as there is no element in the data structure whose value depends on its own value. To make this less abstract, here's a definition of the—yes, infinite—list of all Fibonacci numbers:

---

good at applying this optimization. In situations where it fails to apply this optimization but we want strict evaluation, we can use *seq* and $!. *a* `seq` *b* first forces the evaluation of *a* before returning *b* (possibly as a thunk). So *a* `seq` *a* ensures that *a* is immediately generated as a value, not a thunk. *f* $! *x* forces the full evaluation of *x* before applying *f* to it. How do we figure out whether the compiler needs this type of help? We need to look at the intermediate code generated by the compiler to see where it generates plain values and where it generates thunks. This is not for the faint of heart but is a common technique for the Haskell wizards (which I am not (yet)).

$$fib = 1 : 1 : zipWith\ (+)\ fib\ (tail\ fib)$$

This works just fine because, to compute the third element of the list (the first element produced by the *zipWith* expression), we only need to know the first element of the list and the second element (the first element of *tail fib*). To compute the fourth element, we only need to know the second and the third element, and so on.

The third benefit is that we can easily define expressions whose evaluation would raise errors under some circumstances, as long as we do not evaluate them under these circumstances. A trivial example is the use of $\bot$, written *undefined* in the actual code. Trying to access its value raises an error. So, *putStrLn* (*show* $[\bot, 1]$) raises an error but *putStrLn* (*show* \$ *tail* $[\bot, 1]$) works just fine because we never asked for the value of the first list element. A more useful example is given by the following function, which returns *Nothing* if the list is empty and *Just* the minimum of the list otherwise:[2]

$$\begin{aligned}
&safeMax\ \ \ :: Ord\ a \Rightarrow [a] \rightarrow Maybe\ a\\
&safeMax\ xs \mid null\ xs\ \ \ \ = Nothing\\
&\qquad\qquad \mid otherwise = Just\ m\\
&\quad \textbf{where}\ m = maximum\ xs
\end{aligned}$$

So we define $m = maximum\ xs$. Asking for m's value raises an error if $xs \equiv [\,]$. However, if $xs \equiv [\,]$, then $null\ xs \equiv True$, so we return *Nothing* without ever asking for m and thus without evaluating $maximum\ xs$. If $xs \not\equiv [\,]$, $null\ xs \equiv False$ and we return *Just* m, but that's safe now because the maximum of a non-empty list is well defined.

10 Functors

I've already mentioned type classes *Eq*, *Ord*, *Num*, and *Ix* before. Another important type class in Haskell is a *Functor*:

$$\begin{aligned}
&\textbf{class}\ Functor\ f\ \textbf{where}\\
&\quad fmap :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b
\end{aligned}$$

So, formally, a *Functor* is a container f that can hold values of some arbitrary type a, augmented with a function *fmap* that can turn any function of type $a \rightarrow b$ into a function of type $f\ a \rightarrow f\ b$. More concretely, a *Functor* is usually some type of data structure. It could for example be a binary tree. Given a binary tree $t :: BinaryTree\ a$ and a function $f :: a \rightarrow b$, then we could (and should) define $fmap\ f\ t$ to be the binary tree of the same shape as t and each of whose nodes has its value determined by applying f to the corresponding node value in t. For a type to be an instance of *Functor*, its definition of *fmap* has to obey certain laws, but let's ignore these here.

To make this less dry, two examples. The list type is a functor, as can be seen by the following definition of *fmap*:

[2] This isn't very idiomatic Haskell, but it illustrates the point I'm trying to make.

```
instance Functor [ ] where
  fmap _ [ ]       = [ ]
  fmap f  (x : xs) = f x : fmap f xs
```

So, the *fmap* instance for lists turns a function $f :: a \to b$ into a function $fmap\ f :: [a] \to [b]$ that applies f to each element in the input list. If you paid attention, there is a simpler way to define *fmap* for lists, with the same result:

```
instance Functor [ ] where
  fmap = map
```

If we define a binary tree type as

```
data BinaryTree a = BinaryTree {val                  :: a
                               , leftChild, rightChild :: Maybe (BinaryTree a)
                               }
```

then this can also be made an instance of *Functor* using the following definition:

```
instance Functor BinaryTree where
  fmap f (BinaryTree v l r) = BinaryTree (f l) (fmap (fmap f) l) (fmap (fmap f) r)
```

Holy cow, why are we using *fmap* twice in this definition? Well, the *Maybe* type

```
data Maybe a = Nothing
             | Just a
```

is itself a functor with the quite obvious definition

```
instance Functor Maybe where
  fmap _ Nothing = Nothing
  fmap f  (Just x) = Just (f x)
```

Applying f to the element in the empty container *Nothing* must produce *Nothing*. Applying f to the element in the non-empty container *Just* x must produce another non-empty container storing $f\ x$, that is, *Just* $(f\ x)$. So, since the instance of *fmap* f defined for binary trees is a function of type *BinaryTree* $a \to$ *BinaryTree* b if f is a function of type $a \to b$, then the instance of *fmap* (*fmap* f) defined for *Maybe* (*BinaryTree* a) is of type *Maybe* (*BinaryTree* a) $\to$ *Maybe* (*BinaryTree* b), exactly what we need to transform l and r in the above definition.

A particular type of functor is an *applicative* functor:

```
class Functor f ⇒ Applicative f where
  pure :: a → f a
  (⊛)  :: f (a → b) → f a → f b
```

Again, these operations have to satisfy certain laws, which we ignore here. *pure* allows us to "wrap" an arbitrary value in an applicative functor, while $(\circledast)$ allows us to combine a "functor of functions" with a "functor of values" to produce a new functor of values. Lists and *Maybe* are applicative functors:

```
instance Applicative [ ] where
  pure x = [x]
  fs ⊛ xs = concatMap (flip fmap xs) fs
```

The *concatMap* function is defined as

```
concatMap :: (a → [b]) → [a] → [b]
concatMap = concat ∘ map
```

It takes a function f that produces a list of type $[b]$ from a value of type a and applies it to every element in a list of type $[a]$. The resulting list, produced using *map*, has type $[[b]]$; it's a list of lists. *concat* takes a list of lists of type $[[b]]$ and concatenates them into a single list of type $[b]$:

```
concat :: [[b]] → [b]
concat = foldr (++) [ ]
```

$(+\!\!+)$ is the concatenation operator defined as:

```
(++)          :: [a] → [a] → [a]
[ ]      ++ ys = ys
xs       ++ [ ] = xs
(x : xs) ++ ys = x : (xs ++ ys)
```

Back to [ ] as an applicative functor, $fs \circledast xs$ is the list of values produced by applying every function in fs to every value in xs.

To define the *Applicative* instance of *Maybe*, let's observe that applying a function to no value at all (*Nothing*) cannot produce anything but no value at all: *Nothing*. Applying no function at all to a value must also produce *Nothing*.[3] Finally, applying a function to a value should produce *Just* the result:

```
instance Applicative Maybe where
  pure x              = Just x
  Nothing ⊛ _         = Nothing
  _       ⊛ Nothing   = Nothing
  (Just f) ⊛ (Just x) = Just (f x)
```

A more idiomatic definition that reuses the *Functor* instance of *Maybe* is

[3] Producing the original value unaltered would be the same as applying the identity function, not the result of applying no function at all.

instance *Applicative Maybe* **where**
  $pure\ x \quad = Just\ x$
  $Nothing \circledast \_ = Nothing$
  $(Just\ f) \circledast x = Just\ (fmap\ f\ x)$

On the other hand, it seems hard to define an *Applicative* instance of binary trees. The main reason I care about applicative functors in these notes is a pretty shallow one. They support an operation ($\circledS$) defined as

$(\circledS) :: Functor\ f \Rightarrow (a \to b) \to f\ a \to f\ b$
$f \circledS x = pure\ f \circledast x$

What this boils down to is that $f \circledS x$ is essentially syntactic sugar for writing $fmap\ f\ x$ or f `` `fmap` `` x. It looks more like applying a mathematical operation, so I prefer to write this.

11 Imperative Programming Using Monads

Now, a purely functional language would be pretty useless if it didn't also have some means to break out of the functional paradigm. After all, interacting with the real world involves side effects. We modify the world by writing files to disk, calling a read function on a file may produce different results depending on the contents of the file, etc. Haskell's approach to this was messy in early versions until the designers discovered monads. I won't bore you with what a monad is mathematically, mainly because it doesn't seem to help with understanding what monads do in Haskell. I should also point out that monads are used for many more things than for capturing things that are more efficiently done imperatively, and many Haskell experts would boo me for presenting monads this way. A more appropriate definition would be that monads capture stateful computations: what a computation does depends on the current state (of the world, of global variables, etc.)

Intuitively, a Haskell monad is a structure that allows us to specify the order in which certain things have to happen. If some computation f depends on some state and g modifies this state, then clearly the outcome can be dramatically different if we execute f before g or the other way around. We obviously wouldn't want the compiler to turn a sequence of writing an array cell and then reading it later into a sequence where we read the cell first and then write it; the results could be dramatically different.

Formally, a monad is a type m in the monad type class defined as follows:

class *Monad m* **where**
  $return :: a \to m\ a$
  $fail \quad :: String \to m\ a$
  $(\gg\!=) \quad :: m\ a \to (a \to m\ b) \to m\ b$

So we have to be able to do three things in a monad. First, we have to be able to augment a pure value with some state in the monad using *return*. This is similar to wrapping a value in a functor and

in fact every monad is an applicative functor. Second, we have to be able to fail a computation using an error message of type *String*. Third, the $(\gg\!=)$ operation, called "bind", executes its first argument, which manipulates the state of the monad and produces a value of type a. Then it executes the second argument, which takes the result of the first computation as an argument and whose behaviour can further depend on the current state of the monad. The final result is the result returned by this second computation.

The most fundamental monad in Haskell is the *IO* monad. It provides us with the means to interact with the real world. For example, to read a line of text from stdin and write it back to stdout, we could write

$$getLine \gg\!= putStrLn$$

The types of these two functions are

$$\begin{aligned} &getLine &&:: IO\ String \\ &putStrLn &&:: String \to IO\ () \end{aligned}$$

Now, let us say we want to instead read the line, drop multiple consecutive occurrences of the same character and write the resulting string back to stdout. First we define a function *dropDuplicates*:

$$\begin{aligned} &dropDuplicates && :: Eq\ a \Rightarrow [a] \to [a] \\ &dropDuplicates\ [\,] && = [\,] \\ &dropDuplicates\ l@[\,\_\,] && = l \\ &dropDuplicates\ (x : xs@(y : \_)) && \mid x \equiv y \quad = dropDuplicates\ xs \\ &&& \mid otherwise = x : dropDuplicates\ xs \end{aligned}$$

In this definition, I used a notation we haven't used before. Using @, we can name parts of a pattern to refer to it on the right-hand side of the equation. Now we have a gazillion different ways to express the computation we want to implement. Using the constructs we know already, we could define

$$printWithoutDuplicates = getLine \gg\!= putStrLn \circ dropDuplicates$$

or with parenthesization

$$printWithoutDuplicates = getLine \gg\!= (putStrLn \circ dropDuplicates)$$

This is perfectly correct because $putStrLn \circ dropDuplicates$ is a function that takes a string, applies *dropDuplicates* to it and passes the result to *putStrLn* to print it to stdout. $getLine \gg\!= putStrLn \circ dropDuplicates$ thus reads a string from stdin, passes it to *dropDuplicates* and prints the result to stdout. However, if we build more complicated computations this way, this can get confusing because the functions aren't even listed in the order they are applied.

A much improved version in this instance can be obtained by using the right-to-left version of $(\gg\!=)$

$$(=\!\ll) :: Monad\ m \Rightarrow (a \to m\ b) \to m\ a \to m\ b$$

So $x =\!\!\langle\!\langle\, y$ executes y, passes the result to x, and then returns the result returned by x. Using this, we can define

$$printWithoutDuplicates = putStrLn \circ dropDuplicates =\!\!\langle\!\langle\, getLine$$

which clearly shows the flow of the computation if read from right to left. A similar effect can be obtained by exploiting that every monad is also an applicative functor. So, we can write

$$printWithoutDuplicates = putStrLn =\!\!\langle\!\langle\, (dropDuplicates \mathbin{\$} getLine)$$

So, if $f :: a \to b$ is a pure function and $x :: m\ a$ is a computation, then $f \mathbin{\$} x$ is a computation that executes x and then produces its result by applying the function f to the value produced by x. Now, in the case of *printWithoutDuplicates*, this or the previous version is probably the clearest possible way of writing this. For more complicated computations involving the mixing of monadic computations and pure functions and the creation of named intermediate results, this gets rather tedious very quickly. To clean things up, Haskell offers us do-notation, a way of writing such computations in a fairly familiar imperative style as a sequence of steps:

$$\begin{aligned} printWithoutDuplicates = \mathbf{do}\ & line \leftarrow getLine \\ & \mathbf{let}\ line' = dropDuplicates\ line \\ & putStrLn\ line' \end{aligned}$$

An intuitive way of thinking about what this computation does is: Execute *getLine* and assign the result to *line*. Then apply *dropDuplicates* to *line* to produce a new string $line'$. Finally, output $line'$ by passing it to *putStrLn*. The execution order may not be exactly as we have just discussed, due to laziness, but semantically the result is the same.

It is interesting to observe that some standard Haskell types are monads. *Maybe* is a monad and can be used to build computations that may fail. Consider the following example: We have a function $f :: a \to Maybe\ b$ and a function $g :: b \to Maybe\ c$. So, the function application $f\ x$ may "succeed" by producing a value $Just\ y$ or it may "fail" by producing *Nothing*. If f fails, the whole computation fails. If f succeeds, then g should by applied to y and may in turn succeed or fail. This pattern is all too common in computations consisting of multiple parts that can each in turn fail. Using the standard combinator

$$\begin{aligned} &maybe && :: b \to (a \to b) \to Maybe\ a \to b \\ &maybe\ def\ \_\ Nothing &&= def \\ &maybe\ \_\ \ f\ (Just\ x) &&= f\ x \end{aligned}$$

we can write the above computation as

$$maybe\ Nothing\ g\ (f\ x)$$

If $f\ x = Nothing$, then return *Nothing*. Otherwise, return $g\ y$, where $f\ x = Just\ y$. That's clean enough, but now let's string five steps together like that. If none of the five steps can fail, we can use function composition:

$$fiveSteps = f_5 \circ f_4 \circ f_3 \circ f_2 \circ f_1$$

If each of these functions can fail and the whole computation should fail if one of the functions fails, we get

$$fiveSteps = maybe\ Nothing\ (maybe\ Nothing\ (maybe\ Nothing\ (maybe\ Nothing\ f_5 \circ f_4) \circ f_3) \circ f_2) \circ f_1$$

Oh dear, that's not very clear at all. Wouldn't it be nice if we were able to say something like, "Try to apply $f_1, \ldots, f_5$" one after another and give up as soon as one of them fails"? Sure we can:

$$fiveSteps\ x = f_1\ x \gg\!= f_2 \gg\!= f_3 \gg\!= f_4 \gg\!= f_5$$

or even more closely to the above pure version:

$$fiveSteps = f_1 \gg\!\!= f_2 \gg\!\!= f_3 \gg\!\!= f_4 \gg\!\!= f_5$$

So, how does this work? First we define the $(\gg\!\!=)$ operator (which is in fact defined already):

$$\begin{aligned}
&(\gg\!\!=) \quad :: Monad\ m \Rightarrow (a \to m\ b) \to (b \to m\ c) \to (a \to m\ c)\\
&(a \gg\!\!= b)\ x = a\ x \gg\!= b
\end{aligned}$$

With this definition, it should be clear that the above two versions of fiveSteps are equivalent. Essentially, $(\gg\!\!=)$ is to stateful computations what $(\circ)$ is to pure functions. Now, how does the above encode our idea of "keep going until we finish or we encounter the first failure"? This is encoded in the definition of *Maybe* as a *Monad* instance:

```
instance Monad Maybe where
  return      = Just
  fail        = const Nothing
  (Just x) >>= f = f x
  Nothing  >>= _ = Nothing
```

Lists are also monads, except that the following isn't exactly valid Haskell code due to the special nature of list syntax:

```
instance Monad [ ] where
  return x = [x]
  fail     = const [ ]
  xs >>= f = concatMap f xs
```

This can be used quite nicely to simulate non-deterministic computations: A non-deterministic function can return a number of different results for the same argument. We can simulate this by returning the list of all of these values. Stringing together two non-deterministic functions can produce any value that can be obtained by applying the second function to any value that could be produced by the first function. That's exactly what the definition of $(\gg\!=)$ for lists captures.

12 The Strict State Monad

The monad instances we have seen so far fail to deliver what I promised: imperative behaviour inside pure functions whenever we need it to make the algorithm fast. The *IO* monad allows us to implement imperative behaviour and we can certainly lift pure values into the *IO* monad, but we can't go the other way: once a value is of type *IO a*, there is no way to make it pure again—we're stuck inside the *IO* monad from here on. This makes sense. Since the *IO* monad can interact with the outside world, we have no guarantee that the value returned by a computation of type *IO a* doesn't depend on the state of the outside world, so this value can never be pure. The *Monad* instances of *Maybe* and lists are really just vehicles to abstract certain common behaviours—failure and non-determinism—in otherwise pure computations. So it looks like we need yet another monad: the pure state monad *ST s*. A computation of type *ST s a* returns a value of type *a*, and we can turn this into a pure function using *runST*:

$$runST :: (\forall\ s \circ ST\ s\ a) \to a$$

Alright, something weird is going on here. We have a type parameter *s* that is never used in the computation and has nothing to do with the return type of the computation. I won't get into the details, but the gist is that this phantom type parameter ensures that we cannot pass any state information from one computation in the *ST* monad to another computation. This is the key to containing the non-pure behaviour of the computation to the argument of *runST*, so we can guarantee that from the outside this computation is pure.

Now, the strict state monad gives us two key features needed for imperative programming. The first one is memory locations whose values we can update. Such a memory location is created by calling

$$newSTRef :: a \to ST\ s\ (STRef\ s\ a)$$

As the type signature states, this creates a new mutable memory location storing a value of type *a*. This type is the same type as the argument we pass to *newSTRef*, which is also the value initially stored in this memory location. For example, *newSTRef* 1 creates an *STRef s Int*. Note that this is a computation in the *ST s* monad, as it should be because it manipulates the state of the monad by creating a new reference cell. Given an *STRef*, its contents can be read using *readSTRef*:

$$readSTRef :: STRef\ s\ a \to ST\ s\ a$$

and written using *writeSTRef*:

$$writeSTRef :: STRef\ s\ a \to a \to ST\ s\ ()$$

The type signature here gives us a clue that we'd use *writeSTRef* purely for its side effect of storing the given value of type *a* in the *STRef*. The return value is of type (), that is, essentially `void`.

As an illustration. here is how we could implement the *sum* function we've seen earlier in an imperative style using the strict state monad. Note that, from the outside, the function is pure, that is, side-effect-free:

```
sum    :: Num a ⇒ [a] → a
sum xs = runST (sumM xs)

sumM   :: Num a ⇒ [a] → ST s a
sumM xs = do sum ← newSTRef 0
             mapM_ (addVal sum) xs
             readSTRef sum

addVal         :: Num a ⇒ STRef s a → a → ST s ()
addVal sum x = do s ← readRef sum
                  writeRef sum (s + x)
```

*mapM_* and *mapM* are similar to *map* but for stateful computations:

```
mapM  :: (a → m b) → [a] → m [b]
mapM_ :: (a → m b) → [a] → m ()
```

mapM_ f xs and *mapM f xs* both apply *f* to every element in *xs* but *mapM_* discards the results, that is, is used whenever we care only about the side effects of *f*, as we do here.

There is another useful function that allows us to modify the contents of an *STRef* by applying a pure function to it:

```
modifySTRef    :: STRef s a → (a → a) → ST s ()
modifySTRef r f = writeSTRef r ∘ f =≪ readSTRef r
```

Using this function, we could have implemented *addVal* more easily:

```
addVal sum x = modifySTRef sum (+x)
```

Now a mutable array is essentially a whole number of consecutive mutable memory locations. This is made available to us in the form of an *STArray*. There are lots of functions to manipulate these. We'll discuss only the four most important ones here.

First, given an *Array a b*, we can turn it into an *STArray s a b* using *thaw*:

```
thaw :: Array a b → ST s (STArray s a b)
```

This means we make an immutable array mutable. Well, not quite; we make a mutable *copy* of the given immutable array. To get an immutable copy of a mutable array back (after we are done messing with it), we *freeze* it:

```
freeze :: STArray s a b → ST s (Array a b)
```

Finally, to read and write from and to an *STArray*, we use *readArray* and *writeArray*:

```
readArray  :: Ix a ⇒ STArray a b → a → ST s b
writeArray :: Ix a ⇒ STArray a b → a → b → ST s ()
```

What they do should be fairly obvious from their names and type signatures. Their types are in fact a bit more daunting:

$$readArray :: (MArray\ a\ e\ m, Ix\ i) \Rightarrow a\ i\ e \to i \to m\ e$$
$$writeArray :: (MArray\ a\ e\ m, Ix\ i) \Rightarrow a\ i\ e \to i \to e \to m\ ()$$

The reason is that the *IO* monad and possibly other monads also offer mutable arrays. The *IO* monad has *IOArray*s. *readArray* and *writeArray* are overloaded functions that can be used for any monad *m* that offers mutable arrays *a* capable of storing elements of type *e*. This is what the *MArray* type class captures.

13 Some Useful Monads and Monad Transformers

Let me close this introduction with some real Haskell craziness, which I will use very sparingly, if at all, throughout the remainder of these notes. The concept seems rather daunting, but once you wrap your head around it, it can lead to really clean and elegant code. The topic I'm talking about is monad transformers. We'll first look at four common basic monads reflecting four common types of stateful computations and then move on to the question how to combine these types of computations.

13.1 Maintaining State

There are numerous situations where different parts of the computation share some state their behaviour may depend on and which they may update. There are many ways to make this state available to these computations:

- **Global variables:** That's the way it's commonly done in C, and it has come into disrepute because it is hard to keep track of the way information flows through the program because any function can modify any global variable at any time.
- **State objects:** In an object-oriented language, we can package the state and the functions that can modify it in an object. This enhances modularity but is not available to us in Haskell.
- **Explicit state passing:** We can of course write functions that take the current state as an argument and return the new state explicitly as part of their results. This makes the flow of information explicit but can become tedious if the state is shared by many functions.

Explicit state passing is a perfectly acceptable solution if the state is localized to a small number of functions because the inconvenience of explicit state passing is fairly low and it provides the benefit of making the flow of information explicit. If the state needs to be shared by many functions, then we can use the *State* monad, whose bind operator (⟩⟩=) takes care of this state passing for us. Any simple example cannot really demonstrate the benefits of the *State* monad because, as just said, when the state passing is limited to a small set of functions, explicit state passing is often the cleanest solution. With this in mind, let's still look at a simple example that demonstrates how the *State* monad works.

Every rooted tree can be encoded using a sequence of balanced parentheses: A single-node tree is encoded as `"()"`. A node v with children $w_1, w_2, \ldots, w_k$ is encoded as `"("`, followed by the encodings of the subtrees of $w_1, w_2, \ldots, w_k$, followed by `")"`. Now let's assume we're given the parenthesis sequence encoding some tree T and we want to output a sequence of pairs (i, c_i) such that c_i is the number of children of the ith node; nodes are numbered in preorder. For example, for the input `"((()(()))(()(()())()))"`, the output would be $[(1,2),(2,0),(3,2),(4,0),(5,0),(6,3),(7,0),(8,2),(9,0),(10,0),(11,0)]$ or some permutation thereof; we do not require the pairs in the output to be in any particular order. An algorithm that carries out this computation would scan the input string and thereby effectively perform an in-order traversal of the tree. As it does this, it maintains a stack storing the preorder numbers and numbers of children of all ancestors of the current node, as well as a counter that gets incremented every time we encounter a new node. Here's how we would implement this algorithm:

$$
\begin{aligned}
&\mathit{numChildren} :: \mathit{String} \to [(\mathit{Int}, \mathit{Int})] \\
&\mathit{numChildren} = \mathit{scan}\ 1\ [(0,0)] \\
&\quad \textbf{where}\ \mathit{scan}\ \_\ \_\ \quad [\,] \quad\quad = [\,] \\
&\qquad\quad \mathit{scan}\ i\ ((j,c) : st)\ (\text{'('} : xs) = \quad \mathit{scan}\ (i+1)\ ((i,0) : (j, c+1) : st)\ xs \\
&\qquad\quad \mathit{scan}\ i\ (jc \quad : st)\ (\text{')'} : xs) = jc : \mathit{scan}\ i \qquad\qquad\qquad\qquad st\ \ xs
\end{aligned}
$$

This isn't particularly messy, but that's because it doesn't actually use state passing at all; it uses plain old tail recursion. So, for the sake of this exercise, let's turn this into an implementation that uses state passing:

$$
\begin{aligned}
&\mathit{numChildren} \quad :: \mathit{String} \to [(\mathit{Int}, \mathit{Int})] \\
&\mathit{numChildren}\ xs = cs \\
&\quad \textbf{where}\ (cs, \_, \_) = \mathit{foldl}\ \mathit{scan}\ ([\,], 1, [(0,0)])\ xs \\
&\\
&\qquad \mathit{scan}\ (cs, i, (j,c) : st)\ \text{'('} = (\quad cs, i+1, (i,0) : (j, c+1) : st) \\
&\qquad \mathit{scan}\ (cs, i, jc \quad : st)\ \text{')'} = (jc : cs, i, \qquad\qquad\qquad st)
\end{aligned}
$$

Still not too messy, but we're starting to construct rather complicated return values here, in order to pass the state of the computation along from one application of *scan* to the next. Now let's turn this into a computation in the *State* monad. A computation of type *State s a* has a state of type s and returns a value of type a. We run such a computation using

$$\mathit{runState} :: \mathit{State}\ s\ a \to s \to (a, s)$$

runState f s runs the computation f with the initial state s and returns a pair consisting of the return value of f and the state at the end of f. If we only care about the return value, we can use

$$
\begin{aligned}
&\mathit{evalState} \quad :: \mathit{State}\ s\ a \to s \to a \\
&\mathit{evalState}\ f\ s = \mathit{fst}\ (\mathit{runState}\ f\ s)
\end{aligned}
$$

If we only care about the final state, we can use

```
execState    :: State s a → s → s
execState f s = snd (runState f s)
```

Inside a computation of type *State s a*, we have three functions at our disposal:

```
get    :: State s s
put    :: s → State s ()
modify :: (s → s) → State s ()
```

get returns the current state. *put* stores the given value as the new state. *modify* modifies the current state using the given function. We'll worry a bit later about how they're implemented; there's nothing magical about them. First, though, let's look at how we can implement *numChildren* using the *State* monad:

```
numChildren xs = fst $ execState (mapM_ numChildrenM xs) ([], (1, [(0, 0)]))

numChildrenM    :: Char → State ([(Int, Int)], (Int, [(Int, Int)])) ()
numChildrenM '(' = modify (λ(cs, (i, (j, c) : st)) → (    cs, (i + 1, (i, 0) : (j, c + 1) : st)))
numChildrenM ')' = modify (λ(cs, (i, jc     : st)) → (jc : cs, (i,                     st)))
```

In this case, there is little benefit to using the *State* monad because the function passed to *modify* isn't any cleaner than the *scan* function above implemented using explicit state passing. However, imagine a sequence of functions only some of which need to manipulate the state. They all need to pass the state along regardless. In the implementation of *numChildrenM*, observe that the state is not part of the argument list, nor is it part of the return value of *numChildren*. So, a function in the *State* monad that doesn't actually need to manipulate the state includes no reference to the state whatsoever in its implementation and still passes it along "behind the scenes".

So how is the *State* monad implemented? You'll laugh: using explicit parameter passing. So it's really nothing but a façade that makes our code cleaner. Here's the *State* type:

```
newtype State s a = State {runState :: s → (a, s)}
```

So *runState* has exactly the right type: *runState* :: *State s a* → *s* → (*a*, *s*). The *Monad* instance of *State* is defined as

```
instance Monad (State s) where
  return x = State $ λs → (x, s)
  f >>= g  = State $ λs → let (y, s') = runState f s in runState (g y) s'
  fail     = error
```

So, *return x* lifts a pure value *x* into the *State* monad by turning it into a function that turns a state *s* into a pair (*x*, *s*), that is, *x* is the return value of the computation and the state is passed through unmodified. *fail* simply raises an exception using *error* because the *State* monad has no other meaningful way to

deal with exceptions. The interesting stuff happens in the implementation of the bind operator: $f \gg\!= g$ is supposed to be a computation that runs f and passes the return value of f as well as the updated state on to g. According to our implementation $f \gg\!= g$ first runs f on the current state to access the pair (y, s'), where y is f's return value and s' is the new state at the end of f. It then passes the return value y to g and runs the resulting computation on the updated state s'. This is exactly what we want.

Now the state access functions available inside the *State* monad can be implemented as follows:

$$
\begin{aligned}
&\mathit{get} &&= \mathit{State}\ \$\ \lambda s \rightarrow (s,\ s)\\
&\mathit{put}\ s &&= \mathit{State}\ \$\ \lambda\_ \rightarrow ((), s)\\
&\mathit{modify}\ f &&= \mathit{put} \circ f =\!\!\ll \mathit{get}
\end{aligned}
$$

So, *get* takes the current state s and returns it as the new state as well as the return value. *put* ignores the current state and makes its argument the new state. *modify f* can of course be implemented by first reading the state using *get* and then writing the updated state obtained by applying f to it using *put*.

13.2 Read-Only Configuration

The *Reader* monad is like half a *State* monad: the state can bet read but cannot be written. So the state is immutable. Why would such a monad be useful? Well, first of all, there are programs that rely on some configuration settings to influence their behaviour. These configuration settings should be made available to all functions in the program that depend on them. Here's a stub of such a function:

```
funWithConfig :: Reader r a
funWithConfig = do    -- Do things
                  cfg ← ask
                    -- Do things that depend on cfg
```

So you may have guessed that we read the configuration using *ask*, which has type

```
ask :: Reader r r
```

To run the above function with a given configuration *cfg*, we'd use

```
runReader funWithConfig cfg
```

The type of *runReader* is

```
runReader :: Reader r a → r → a
```

Now, you may be wondering why we don't use the *State* monad to do all this. Why do we need a separate *Reader* monad. After all, we could just as well implement the above as

```
funWithConfig :: State r a
funWithConfig = do    -- Do things
                  cfg ← get
                    -- Do things that depend on cfg

evalState funWithConfig cfg
```

The result would be exactly the same. The main reason the *Reader* monad is useful is that using it instead of *State* communicates intent—remember, good coding practices are about communicating intent not only to the machine but also to human readers of your code. By using *Reader* instead of *State*, we make it clear that we are dealing not with mutable state that can be manipulated by the computation but with read-only information made available to various functions. A less important reason is that the *Reader* monad is a bit more efficient to implement:

```
newtype Reader r a = Reader {runReader :: r → a}

instance Monad (Reader r) where
  return x = Reader $ λ_ → x
  f >>= g  = Reader $ λr → runReader (g (runReader f r)) r
  fail     = error
```

The "state" is only passed into each function but is not returned as part of the result because, by definition, the function cannot modify the state. *ask* is easily implemented as

```
ask = Reader id
```

where

```
id  :: a → a
id x = x
```

is the identity function.

13.3 Logging Results

Another common program pattern is to produce results in a write-only fashion, to be used as the output of the computation but not to be accessed by the computation itself. So it would be nice to have a construct that allows us to just output parts of the result whenever we produce them. That's what the *Writer* monad is for. We could implement good old *map* using the *Writer* monad (rather awkwardly, but it illustrates the idea):

```
map     :: (a → b) → [a] → [b]
map f xs = execWriter (mapM_ appF xs)
  where appF x = tell [f x]
```

In this instance, the computation *appF x* outputs a singleton list $[f\,x]$ using *tell*. We apply *appF* to every element in *xs* using *mapM_ appF xs*. Each such application produces one singleton list. The final result of this computation is the concatenation of these singleton lists, exactly what we would expect *map* to produce.[4] So how does this all work? In this example, we just produced a list of outputs of the individual steps of the computation. More generally, the writer can use any type in the *Monoid* class as its output "stream". Oh dear, let's take this slowly: In mathematics, a monoid is a semigroup with identity. A semigroup $(S, \circ)$ is a set S equipped with an associative operation $\circ : S \times S \to S$. That is, for two elements $a, b \in S$, $a \circ b$ is also in S, and $(a \circ b) \circ c = a \circ (b \circ c)$. An identity is an element $\epsilon \in S$ such that $a \circ \epsilon = \epsilon \circ a = a$. For example, if $\mathbb{Z}^+$ is the set of non-negative integers, then $(\mathbb{Z}^+, +)$ and $(\mathbb{Z}^+, \cdot)$ are semigroups with identities 0 and 1, respectively. $(\mathbb{Z}^+, \max)$ is also a semigroup with identity 0. So are lists if we define $\circ$ to be list concatenation; the identity is the empty list. Borrowing the naming from lists, the *Monoid* class in Haskell refers to the identity element as *mempty* and to the semigroup operation $\circ$ as *mappend*:

```
class Monoid a where
  mempty  :: a
  mappend :: a → a → a
```

The list instance is now easily defined as

```
instance Monoid [a] where
  mempty  = []
  mappend = (++)
```

Now back to the *Writer* monad. It is defined as

```
newtype Writer w a = Writer {runWriter :: (a, w)}
```

If we aren't interested in the result of the computation but only in its output, we can use

```
execWriter :: Writer w a → w
execWriter = snd ∘ runWriter
```

Here's the *Monad* instance of *Writer*:

```
instance Monoid w ⇒ Monad (Writer w) where
  return x = Writer (x, mempty)
  f >>= g  = let (x, w ) = runWriter f
                 (y, w') = runWriter (g x)
             in Writer (y, mappend w w')
```

And finally *tell*:

```
tell x = Writer ((), x)
```

To see more of the *Writer* monad in action, see Section **??**.

[4] Just not as fast as one would hope. Since this implementation uses left-to-right list concatenation inside the *Writer* monad, this would take quadratic time as opposed to linear.

13.4 Computations That May Fail

Before moving on to the really cool stuff one can do with monads, here's one more common pattern in programming: computations may fail (e.g., if we try to take the head of an empty list). Much has been written about how to deal with such failures. We can return a default value, we can return error codes a la C or we can raise exceptions. Haskell's approach to "hard" failures one cannot reasonably recover from, such as division by zero, is to raise an exception. Haskell also has mechanisms to catch these exceptions and deal with them much like in many other languages that support exceptions. The issue with exceptions is that, as the name indicates, they should refer to exceptional situations. If a computation finishes successfully about as often as it raises an exception, then the situation where an exception is raised is no longer exceptional but is rather part of the expected control flow of the program. In this case, exceptions are the wrong tool. So back to our example of taking the head of an empty list. If we believe that whenever we do this the argument list *xs* is non-empty, then we should just use *head xs*, which raises an exception if *xs* happens to be empty. Depending on the type of circumstances that can lead to an empty list here (user input or error in the program logic), we should catch this exception in the appropriate place or go back and examine whether our assumption that *xs* should be non-empty was in fact wrong. On the other hand, if we expect *xs* to be equally likely to be empty as non-empty, what's the right approach. Clearly, we cannot take the head of an empty list. In this case, we can use our good old *Maybe* type, which was mentioned before:

$$\textbf{data}\ \mathit{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a$$

For an empty list, the head should be *Nothing*; for a non-empty list, it should be *Just* the head:

$$\begin{aligned}
&\mathit{safeHead} && :: [a] \to \mathit{Maybe}\ a \\
&\mathit{safeHead}\ [\,] && = \mathit{Nothing} \\
&\mathit{safeHead}\ (x : _) && = \mathit{Just}\ x
\end{aligned}$$

Now what if we have a sequence of steps that can each fail and the whole sequence should fail if any of these steps fails? Then we can use the *Monad* instance of *Maybe* discussed earlier in these notes.

13.5 Stateful Computations That Log Results?

Now let's get back to our problem of counting the number of children of the nodes of a tree encoded as a balanced parenthesis sequence. You'll observe that the implementation using the *State* monad dragged the final output along as part of the state but never did anything with it other than adding to it whenever a new output element gets generated. The issue is that the computation mixes two patterns: maintaining the next preorder number and the path to the current node as state and logging a list of results. The first is a case for the *State* monad, the latter a case for the *Writer* monad. Wouldn't it be nice if we could build a Writer-State or a State-Writer? It turns out we can.

I lied about the types of *ask*, *tell*, *get*, *put*, and *modify*, and about the implementations of the *State*, *Reader*, and *Writer* monads above. The functions are provided as part of type classes. First the state monad:

```
class MonadState s m | m → s where
  get   :: m s
  put   :: s → m ()
  state :: (s → (a, s)) → m a
```

So, a state monad is one that allows us to get the current state, write the state, and turn a pure function that passes explicitly passes the state into a monadic computation. Default implementations of these functions are provided so a new instance needs to define only *get* and *put* or *state*. These default implementations implement *get* and *put* in terms of *state* and vice versa. So, if an instance doesn't override *get*/*put* or *state*, we get an endless loop.

```
get     = state (λs → (s, s))
put s   = state (λ_ → ((), s))
state f = do (x, s) ← f <$> get
             put s
             return x
```

The *State* monad itself is obtained by applying the monad transformer *StateT* to the *Identity* monad:

```
type State s = StateT s Identity
```

So how are these defined? First the *Identity* monad. It doesn't do anything interesting and only serves the purpose of a basis on which to stack monad transformers:

```
newtype Identity a = Identity {runIdentity :: a}

instance Monad Identity where
  return x = Identity x
  f >>= g  = Identity (runIdentity (g $ runIdentity f))
  fail     = error
```

The *StateT* transformer allows us to augment any existing monad *m* with state:

```
newtype StateT s m a = StateT {runStateT :: s → m (a, s)}
```

Its monad instance is defined almost the same as the monad instance for *State* we defined before. We only have to deal with the fact that the result of *runStateT s* is a monadic action in the underlying monad *m*, not a pure value:

```
instance Monad m ⇒ Monad (StateT s m) where
  return x = StateT $ λs → return (x, s)
  f >>= g  = StateT $ λs → runStateT f s >>= λ(x, s') → runStateT (g x) s'
  fail     = error
```

It is important to note that the *return* and $\gg=$ on the left-hand side are the ones defined for the *StateT s m* monad; the ones on the right-hand side are the ones in the underlying monad m. The *MonadState* instance is defined as

$$\begin{aligned}&\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadState}\ s\ (\mathit{StateT}\ s\ m)\ \textbf{where}\\&\quad \mathit{state}\ f = \mathit{StateT}\ (\mathit{return} \circ f)\end{aligned}$$

We also have equivalent functions to *evalState* and *execState*:

$$\begin{aligned}&\mathit{evalStateT} \quad :: \mathit{StateT}\ s\ m\ a \to m\ a\\&\mathit{evalStateT}\ f\ s = \mathit{fst}\ \$\ \mathit{runStateT}\ f\ s\end{aligned}$$

$$\begin{aligned}&\mathit{execStateT} \quad :: \mathit{StateT}\ s\ m\ a \to m\ s\\&\mathit{execStateT}\ f\ s = \mathit{snd}\ \$\ \mathit{runStateT}\ f\ s\end{aligned}$$

Next up, writer monads. Again, we actually have a *MonadWriter* class

$$\begin{aligned}&\textbf{class}\ (\mathit{Monad}\ m, \mathit{Monoid}\ w) \Rightarrow \mathit{MonadWriter}\ w\ m \mid m \to w\ \textbf{where}\\&\quad \mathit{writer} :: (a, w) \to m\ a\\&\quad \mathit{tell} \quad :: w \to m\ ()\\&\quad \mathit{listen}\ :: m\ a \to m\ (a, w)\\&\quad \mathit{pass} \quad :: m\ (a, w \to w) \to m\ a\end{aligned}$$

Any monad m in this class needs to implement at least one of *writer* and *tell*, as well as both *listen* and *pass*. *listen f* runs f and returns a pair consisting of the return value produced by f and the output written by f. In the context of *pass f*, f is a computation that writes some output and returns a pair consisting of a return value and a function to transform the output. *pass f* runs f, returns the return value and defines its output to be whatever is obtained by applying the function returned by f to the output produced by f. Since only one of *writer* and *tell* need to be implemented, there are again default implementations available that depend on each other:

$$\begin{aligned}&\mathit{writer}\ (x, w) = \mathit{tell}\ w \gg \mathit{return}\ x\\&\mathit{tell} \quad w \quad = \mathit{writer}\ ((), w)\end{aligned}$$

Again, the *Writer* monad is defined using a *WriterT* transformer:

$$\textbf{type}\ \mathit{Writer}\ w = \mathit{WriterT}\ w\ \mathit{Identity}$$

$$\textbf{newtype}\ \mathit{WriterT}\ w\ m\ a = \mathit{WriterT}\ \{\mathit{runWriterT} :: m\ (a, w)\}$$

$$\begin{aligned}&\mathit{execWriterT} \quad :: \mathit{Monad}\ m \Rightarrow \mathit{WriterT}\ w\ m\ a \to m\ w\\&\mathit{execWriterT}\ f = \mathit{snd}\ \$\ \mathit{runWriterT}\ f\end{aligned}$$

The *Monad* instance is defined as

```
instance (Monad m, Monoid w) ⇒ Monad (WriterT w m) where
  return x = WriterT $ return (x, mempty)
  f >>= g  = WriterT $ do (x, w ) ← runWriterT f
                          (y, w') ← runWriterT (g x)
                          return (y, mappend w w')
  fail     = error
```

And the *MonadWriter* instance is

```
instance (Monad m, Monoid w) ⇒ MonadWriter w (WriterT w m) where
  writer = WriterT ∘ return
  listen = WriterT ∘ runWriterT
  pass f = WriterT $ do ((x, g), w) ← runWriterT f
                        return (x, g w)
```

There is also a *ReaderT* transformer whose details I will omit here. Using *StateT* and *WriterT*, we can now build State-Writer and Writer-State monads:

```
type WriterState w s = WriterT w (State s)
type StateWriter s w = StateT s (Writer w)
```

Which one is better? Well, let's see after we define functions *runWriterState* and *runStateWriter*:

```
runWriterState :: WriterState w s a → s → ((a, w), s)
runWriterState f s = runState (runWriterT f) s

runStateWriter :: StateWriter s w a → s → ((a, s), w)
runStateWriter f s = runWriter (runState f s)
```

Apart form a different packaging of the different parts of the result, there isn't much difference here. However, there are differences if we had, for example, a *MaybeT* transformer, which is sadly missing form the standard monad transformer library but is provided by a separate library available from Hackage.

```
newtype MaybeT m a = MaybeT {runMaybeT :: m (Maybe a)}
```

Now, should we use a Maybe-Writer or a Writer-Maybe monad if we want a computation that may fail but uses the Writer component to log its progress so far, so if the computation fails, we can look at the log to see what's been going on? Well, let's see:

```
type MaybeWriter w = MaybeT (Writer w)
type WriterMaybe w = WriterT w Maybe
```

Using *MaybeWriter w*, we can define

```
runMaybeWriter :: MaybeWriter w a → (Maybe a, w)
runMaybeWriter f = runWriter (runMaybeT f)
```

That's exactly what we want. Now let's try the same with the *WriterMaybe*. Running *runWriterT f*, where $f :: \mathit{WriterMaybe}\ w\ a$, gives a result of type $\mathit{Maybe}\ (a, w)$. And now we're stuck because we either get *Just* a value together with a log that tells us how it was obtained or we obtain *Nothing*, not even the log. So in some cases the stacking order matters.

So, after all this, back to our tree traversal problem. Since we care only about the output produced by the function, we'll opt for a *StateWriter*. It makes discarding the return value of the function and the state easier:

$$\mathit{numChildren}\ xs = \mathit{snd}\ (\mathit{runStateWriter}\ (\mathit{mapM\_}\ \mathit{numChildrenM}\ xs)\ (1, [\,(0,0)\,]))$$

$$\begin{aligned}
&\mathit{numChildrenM} && :: \mathit{Char} \to \mathit{StateWriter}\ (\mathit{Int}, [\,(\mathit{Int}, \mathit{Int})\,])\ [\,(\mathit{Int}, \mathit{Int})\,]\ () \\
&\mathit{numChildrenM}\ \texttt{'('} && = \mathit{modify}\ (\lambda(i,(j,c):st) \to (i+1, (i,0):(j,c+1):st)) \\
&\mathit{numChildrenM}\ \texttt{')'} && = \mathbf{do}\ (i, jc : st) \leftarrow \mathit{get} \\
&&& \qquad \mathit{put}\ (i, st) \\
&&& \qquad \mathit{tell}\ jc
\end{aligned}$$

There are lots and lots of more fancy things that can be done using monad transformers, but this introduction has got way too long already, so I'll stop here.