FUNCTIONAL PROGRAMMING

Geraint Jones

MT2023

1 Function and Lists

1.1 Programs as functions

At one level a program is a mapping from its input to its output. Almost everything we know from mathematics about functions is almost what we need to know about programs. Some functions, for essential logical reasons, cannot be programs; and in any sufficiently interesting programming language some programs cannot be total functions. (A total function is one which has a defined result for all arguments.)

However, apart from efficiency, two programs are equivalent exactly if they implement the same function. Functions are equal exactly when they give the same results when applied to the same arguments.

1.2 Types

Every Haskell function has a type: the notation $f :: X \to Y$ asserts that applying f to anything of type X gives you a thing of type Y. For example

```
sin :: Float -> Float
home :: Person -> Address
add :: (Int, Int) -> Int
logBase :: Float -> (Float -> Float)
```

$Float$ is the type of (single precision) floating point numbers, like 3.1415927 or $2.9979245e8$; Int is the type of (limited precision) integers; (X, Y) is the type of ordered pairs of an element of X and an element of Y.

Notice that Haskell type names start with upper case letters; the names of values, by and large, start with lower case letters.

Mathematicians write $f(x)$ and $\sin(\theta)$ for applications. Sometimes one writes $\sin\ \theta$ with no parentheses; in Haskell it is usual not to write the parentheses. But you often need a space: $sintheta$ is one name, $sin\ theta$ is the application of sin to $theta$. In Haskell `sin pi`, `sin (pi)`, and `sin(pi)` are all ways of writing the application of sin to pi (and they all evaluate to something that is not quite zero), and `logBase 10 2`, `(logBase 10) 2`, and `logBase (10) (2)` are all ways of writing the same expression whose value is almost exactly $\log_{10} 2$. On the other hand, you do need parentheses in `add(2,3)`, `add (2, 3)` and `sin(pi/2)`

1.3 Sequences (lists)

Just as a function is a piece of data that captures the idea of computation, one of the ways that the idea of repetition is captured is by sequences, in Haskell pricipally lists. A list of type $[T]$ is a sequence of things each of which is of type T, for example $[3, 1, 4, 1, 5, 9, 2, 7]$ is a $[Int]$, consisting of eight Int values. (There is a pun here: the same brackets [and] are used in type expressions, and in value expressions.)

Where there is an idea of a *next* element of type T, the expression $[a..b]$ evaluates to $[a, a+1, a+2, \ldots b]$. There are lists of every type, including list types, so $[[1],[2,3],[4,5,6]] :: [[Int]]$.

One particularly handy idea is the list comprehension, meant to be reminiscent of set comprehension notation in mathematics. The value of a list comprehension is a list of values of the expression to the left of the bar.

The parts after the bar are taken left-to-right: generators introduce new variables which successively take values from lists, and Boolean valued expressions are guards.

```
> map :: (a -> b) -> [a] -> [b]
> map f xs = [ f x | x <- xs ]

> filter :: (a -> Bool) -> [a] -> [a]
> filter p xs = [ x | x <- xs, p x ]

> concat :: [[a]] -> [a]
> concat xss  = [ x | xs <- xss, x <- xs ]
```

Notice that these apply (uniformly) to lists of all types a. (map, $filter$ and $concat$ are standard functions, and although equal these definitions are not the standard ones.)

So for example

```
map abs [-5..5] = [5,4,3,2,1,0,1,2,3,4,5]
filter even [1..10] = [2,4,6,8,10]
concat [[1],[2,3],[4,5,6]] = [1,2,3,4,5,6]
```

There is a standard definition of the type $String = [Char]$ and lists of this type have a handy compact representation:

```
['a'..'z'] = "abcdefghijklmnopqrstuvwxyz"
concat ["al","pha","bet","ic","al","ly"] = "alphabetically"
```

1.4 Script files

Definitions are written in *script files*. In our case, script files have names ending in `.lhs` (for *literate Haskell script*). Lines starting with > (marks known as *Bird tracks*) are read as definitions, and the rest are comments. To prevent accidents caused by typos, there has to be a blank line between program lines and comment lines. (Literate Haskell scripts encourage explanatory commentary, and are considered to be a Good Thing by all Right Thinking People.)

If you really prefer (and you may find your tutor prefers), you can have `.hs` files in which all lines are definition lines, unless explicitly marked as comments by either `{-` and `-}` brackets, or `--` which marks a comment extending to the end of the line.

1.5 Composition of functions

You will almost certainly be familiar with composition of functions: the composition $f \circ g$ of f and g is $(f \circ g)(x) = f(g(x))$ or in Haskell terms

```
> (f . g) x = f (g x)
```

Operators whose names are made of symbols, rather than letters, can appear infixed; and the definition means the same as

```
> (.) f g x = f (g x)
```

where the parentheses make `(.)` behave like an ordinary name. The composition operator is a perfectly good function

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

The peculiar order of the types is down to the odd convention that we write arguments to the right of functions, but function types have their argument type on the left!

Compostion is associative, $f \cdot (g \cdot h) = (f \cdot g) \cdot h$, so just as one writes sums like $1+2+3+\cdots+n$ without parentheses, it is perfectly OK to write compositions $f \cdot g \cdot h \cdot i \cdot j \cdot k$ without parentheses. This is quite a common program structure.

1.6 Example: most common words in a text

What are the n most common words in a given text?

The input is a text which is a list of characters, containing visible characters like `'B'` and `','`, and blank characters like spaces and newlines (`' '` and `'\n'`). The Haskell type *Char* is the type of characters, and the type of lists whose elements are of type *Char* is $[Char]$.

The output should be something like:

```
hill: 10
a: 4
names: 3
which: 2
```

This output is also a list of characters, in fact it is the string

```
"  hill: 10\n  a: 4\n  names: 3\n  which: 2\n"
```

Applying *putStr* to a string has the effect of outputting its characters in turn.

So we need to design a function

```
mostCommon :: Int -> [Char] -> [Char]
```

The expression $mostCommon\ n\ xs$ evaluates to a string of this form, for printing. One scheme would be to:

1. convert all the letters in the text to lower case, so as not to distinguish between words such as “The” and “the”;
2. break the text up into words, where by definition a word is a maximal length sequence of visible characters;
3. bring all the duplicated words together, by sorting the list of words into alphabetical order;
4. divide the list of words into runs of the same word;
5. replace each run by a single copy with a count of the number of times it occurs;
6. sort these runs into descending order of count;
7. take the first n elements of the sorted list of runs;
8. convert each run into the characters which will represent it, and concatenate these strings into a single string.

Many of these steps are off-the-shelf library functions in Haskell. Their combination is a composition of functions, right to left.

First of all, for clarity, some types:

```
> type Text = [Char]
> type Word = [Char]
> type Run  = (Int, Word)
```

1. We need a function $canonical :: Char \to Char$ that converts upper case letters to lower case and (design decision) turns everything else into a blank. This function then needs to be applied by

   ```
   map canonical :: Text -> Text
   ```

   character by character to each character in the text.

2. The standard function

   ```
   words :: Text -> [Word]
   ```

   breaks a list of characters into a list of words, where a word is itself a list of characters. Words are non-blanks separated by blanks.

3. To sort a list of words, we need a function $sort :: [Word] \to [Word]$. There is a library function

   ```
   sort :: Ord a => [a] -> [a]
   ```

   which will sort lists of any type, provided there is an ordering on that type. Fortunately, the ordering on lists of characters is exactly what we need.

4. To identify the runs we need a function of type $[Word] \to [Run]$ that replaces runs of identical words with a pair: the length of the run, and the common word. There is a library function

```
group :: Eq a => [a] -> [[a]]
```

which divides a list of any type, provided that there is an equality test on that type, into maximal lists of adjacent equal elements.

5. We need a function $codeRun :: [Word] \to Run$ that will take one of these runs to a length/representative pair.

6. The $sort$ function will order these runs, but into descending order of count. One thing we could do is to use the standard function $reverse$ to reverse the sorted list $reverse \cdot sort$.

7. There is a standard function $take$ which returns an initial segment of the list no longer than a chosen length.

8. We need a function $showRun :: Run \to String$ to turn each run into a corresponding list of characters, then

   ```
   concat . map showRun
   ```

   will produce the required string.

So the whole program is

```
> mostCommon :: Int -> Text -> String
> mostCommon n = concat .               -- :: [String] -> String
>                map showRun .          -- :: [Run] -> [String]
>                take n .               -- :: [Run] -> [Run]
>                reverse .              -- :: [Run] -> [Run]
>                sort .                 -- :: [Run] -> [Run]
>                map codeRun .          -- :: [[Word]] -> [Run]
>                group .                -- :: [Word] -> [[Word]]
>                sort .                 -- :: [Word] -> [Word]
>                words .                -- :: Text -> [Word]
>                map canonical          -- :: Text -> Text
```

and apart from the library functions, all of which are things we *could* write but need not because someone already has, we need only write

```
> canonical :: Char -> Char
> canonical c | isLower c = c
>             | isUpper c = toLower c
>             | otherwise = ' '
```

which uses Haskell conditional equations and library functions to manage characters.

The function

```
> codeRun :: [Word] -> (Int, Word)
> codeRun xs = (length xs, head xs)
```

is only ever applied to a non-empty list of identical elements and returns the length of that list (*length* is a standard function) and a representative element. Here *head* is a standard function that returns the first element of a list.

```
> showRun :: (Int, Word) -> String
> showRun (n,w) =  concat ["  ", w, ": ", show n, "\n"]
```

showRun constructs a *String* containing the word, some punctuation, a numeral representing its frequency, and a newline character.

```
*Main> putStr hill
When the Anglo-saxons invaded Britain it is clear that they
took over many place names as names, without understanding their
meaning.  The evidence is to be found in names like Penhill,
where Old English hyll was added unnecessarily to a word which
represented Old Welsh pann, hill.  A Penhill in Lancashire
developed into Pendle Hill, a name which means hill-hill-hill.
England also has a Torpenhow Hill, or hill-hill-hill-hill.
*Main> putStr (mostCommon 5 hill)
  hill: 10
  a: 4
  names: 3
  which: 2
  to: 2
```

(Don't believe everything in that 'hill' text; it's just a convenient string.)

1.7 How to read Haskell expressions

One of the stumbling blocks in reading Haskell when you first meet it is the idiosyncratic syntax for function application. It soon becomes natural!

Expressions like $f\ x$ represent an application of a function called f to an argument x. No paretheses are necessary for the simple reason that x and (x) are the same: why would they not be? So if you really must write $f(x)$ you can, but it will eventually annoy you as much as it annoys anyone else.

The value of f in $f\ x$ had better be a function; but the same is true about expressions like $x\ f$ where x had better be a function, and $phu\ bar$ where phu has to be a function which is being applied to the argument bar. The spelling of names does not matter.

Function application binds to the left, so if you write $f\ x\ y$ you get the same expression as if it had been $(f\ x)\ y$. In this case, f will be a function which when applied to an argument x returns a value $f\ x$ which is itself a function that is then applied to y.

On the other hand, operators such as $+$ and $+\!\!+$ made out of symbols are treated differently. Expressions like $x + y$ are treated as the application of a function, which can be referred to as $(+)$, to x and y.

Operators bind less tightly than function application, so $p\ q + r\ s$ is the same as $(+)\ (p\ q)\ (r\ s)$.

So, for example $f \cdot g$ is the same as $(\cdot)\ f\ g$, and $(f \cdot g)\ x$ is the same as $(\cdot)\ f\ g\ x$. But neither of these is the same as $f \cdot g\ x$ which means $f \cdot (g\ x)$.

Exercises

1.1 Recall that function application binds more tightly than any other operators. Put in all the parentheses implicit in the expressions

1. `a plus f x + x times y * z`
2. `3 4 + 5 + 6`
3. `2^2^2^2`

("What does it say?", not "What do you think it should say?") It may help to know that `2^2^2^2` evaluates to 65536, as you can check with GHCi.

1.2 Prove that function composition is associative. (Remember that functions are equal precisely when they return the same result whenever applied to the same argument.)

1.3 Suppose that the $+\!\!+$ operator is defined by

```
as ++ bs = concat [as, bs]
```

(This is not the standard definition, but it defines the same function.)

Is this operator associative? Is it commutative? Does it have a unit (identity element)? Does it have a zero?

(e is a unit of $\oplus$ if $e \oplus x = x = x \oplus e$. z is a zero of $\oplus$ if $z \oplus x = z = x \oplus z$.)

You might be able to tell these things without yet being able to prove that you are right.

1.4 Suppose that

```
double :: Integer -> Integer
double x = 2 * x
```

is the function that doubles an integer. What are the values of

```
map double [3,7,4,2]
map (double.double) [3,7,4,2]
map double []
```

You might check your answers on an interpreter.

Suppose that

```
sum :: [ Integer ] -> Integer
```

is a function that adds up all of the elements of its argument. (There is such a standard function.) Which of the following are true, and why?

$$\begin{aligned} sum \cdot map\,double &= double \cdot sum \\ sum \cdot map\,sum &= sum \cdot concat \\ sum \cdot sort &= sum \end{aligned}$$

2 Definitions

2.1 A problem to be solved

The example in this lecture is to design a function

```
convert :: Int -> String
```

that translates a number $0 \leqslant n < 1\,000\,000$ into a string which names that number in English. The type Int is the type of limited precision integers, and $String$ is a synonym for $[Char]$.

2.2 Numbers less than 10

A good approach to a daunting problem is to solve smaller problems first. The most straightforward way to convert a small number, $0 \leqslant n < 10$ is to deal with each value as a special case, giving an equation for each value:

```
> name :: Int -> String
> name 0 = "zero"
> name 1 = "one"
> name 2 = "two"
> name 3 = "three"
> name 4 = "four"
> name 5 = "five"
> name 6 = "six"
> name 7 = "seven"
> name 8 = "eight"
> name 9 = "nine"
```

but perhaps it would be tidier to look up the answer in a list:

```
> units :: Int -> String
> units u = unitStrings!!u

> unitStrings :: [String]
> unitStrings = [ "zero", "one", "two",   "three", "four",
>                 "five", "six", "seven", "eight", "nine" ]
```

The operator

```
(!!) :: [a] -> Int -> a
```

selects an item from a list according to its position, starting to count from position zero.

2.3 Numbers less than 100

There is a pattern (in English, a slightly complicated one) to the names of two digit numbers which it is worth exploiting.

If $0 \leqslant n < 100$, the tens digit is $div\ n\ 10$ and the units digit is $mod\ n\ 10$.

```
> digits2 :: Int -> (Int,Int)
> digits2 n = (div n 10, mod n 10)
```

Perhaps more usually one would write

```
digits2 n = (n `div` 10, n `mod` 10)
```

The ` marks (called backquotes) convert a name like `div` into an infix operator `` `div` ``. (This is a different character from the single quote ' which marks a character constant like `'x'`.)

There is a standard function *divMod* which satisfies

```
p `divMod` q = (p `div` q, p `mod` q)
```

but which only performs one division operation to produce both results, and so is more efficient when as here both results are needed, so we could write

```
> digits2 n = n `divMod` 10
```

but we choose not to here.

Now we can define

```
> convert2 :: Int -> String
> convert2 = combine2 . digits2

> combine2 :: (Int,Int) -> String
> combine2 (t,u)
>   | t==0         = units u
>   | t==1         = teens u
>   | 2<=t && u==0 = tens t
>   | 2<=t && u/=0 = tens t ++ "-" ++ units u
```

where $(+\!\!+)$, pronounced 'cat', concatenates two lists: in this case two lists of characters. The guarded equations in *combine2* are all part of one equation defining the function according to which of the conditions are true.

The guards are written using tests: `(==)` for equality, `(/=)` for inequality, and `(<=)` for no-more-than, all of which return a value of type *Bool*. This type has two values *True* and *False*, and the && operator (read *and*) returns *True* exactly when both arguments are *True*.

As written only one of the guards is ever true for a given (t, u), but we could have written

```
combine2 (t,u)
  | t==0      = units u
  | t==1      = teens u
  | u==0      = tens t
  | otherwise = tens t ++ "-" ++ units u
```

where *otherwise* is a predefined constant equal to *True*. This is because the guards are tested in order from the top to the bottom, and the first *True* guard wins. In this case, the order of the guarded equations matters.

The words for tens and units come from

```
> teens, tens:: Int -> String
> teens u = teenStrings!!u
> tens t  = tenStrings!!(t-2)

> teenStrings :: [String]
> teenStrings = [ "ten",      "eleven",  "twelve", "thirteen",
>                 "fourteen", "fifteen", "sixteen", "seventeen",
>                 "eighteen", "nineteen" ]

> tenStrings :: [String]
> tenStrings = [ "twenty", "thirty",  "forty",  "fifty",
>                "sixty",  "seventy", "eighty", "ninety" ]
```

2.4 Numbers less than 1 000 and less than 1 000 000

This scheme extends to numbers $0 \leqslant n < 1\,000$, with up to three digits, which can be treated as a number of hundreds followed by a two digit number.

```
> digits3 :: Int -> (Int,Int)
> digits3 n = (n `div` 100, n `mod` 100)

> convert3 :: Int -> String
> convert3 = combine3 . digits3

> combine3 :: (Int,Int) -> String
> combine3 (h,n)
>   | h==0      = convert2 n
>   | n==0      = units h ++ " hundred"
>   | otherwise = units h ++ " hundred and " ++ convert2 n
```

And finally for $0 \leqslant n < 1\,000\,000$

```
> digits6 :: Int -> (Int,Int)
> digits6 n = (n `div` 1000, n `mod` 1000)
```

```
> convert6 :: Int -> String
> convert6 = combine6 . digits6

> combine6 :: (Int,Int) -> String
> combine6 (m,n)
>   | m==0      = convert3 n
>   | n==0      = convert3 m ++ " thousand"
>   | otherwise = convert3 m ++ " thousand" ++ link n ++ convert3 n
```

The function *link* inserts an *and* into the answer exactly when there are thousands, but there are no hundreds:

```
> link :: Int -> String
> link n = if n < 100 then " and " else " "
```

(This is the sort of thing that makes natural language so messy.)

The definition might have been given with guarded equations, but here we illustrate the Haskell conditional expression

if *test* **then** *truecase* **else** *falsecase*

Finally

```
> convert :: Int -> String
> convert = convert6
```

The crucial step in this design is to build each of the numbered *convert* functions from a smaller one, in particular the decision to code *convert3* by using *convert2*. An alternative design might separate a three digit number into three digits and deal with more different cases.

2.5 Local definitions: **let** expressions and **where** clauses

Sometimes it is useful to be able to give a name to something without that name being available all over the program. Haskell provides a form of expression for this.

```
> f1 n = let phi    = (1 + root5)/2
>            phibar = -1/phi
>            root5  = sqrt 5
>        in (phi^n-phibar^n)/root5
```

These **let** expressions can be used anywhere where an expression is needed, and the list of local definitions can include any legal defining equation that could have appeared at the top level. (So in particular you can define local functions.)

However **let** expressions are relatively rarely used. Much more common (because it is in general much more natural to read) is the **where** clause

```
> f2 n = (phi^n-phibar^n)/root5
>    where phi    = (1 + root5)/2
>          phibar = -1/phi
>          root5  = sqrt 5
```

which qualifies a defining equation. (The idea of a **where** clause in a program appears to have started with Christopher Strachey.)

For clarity: a **where** clause does not qualify the right-hand side of a definition, it qualifies the equation. A **where** clause cannot be added to anything other than an equation, though of course **where** clauses can be nested by adding them to equations in an enclosing **where** clause.

```
> f3 n = (phi^n-phibar^n)/(phi-phibar)
>        where phi = (1 + root5)/2
>                  where root5 = sqrt 5
>              phibar = -1/phi
```

2.6 Offside rule

This last definition illustrates the usefulness of the Haskell offside rule. Equations start in a particular column, and the whole of the rest of the equation (including, if it has any, its where clauses) have to appear to the right of that. The next line which starts in the same column, or to the left, does not continue this equation.

It is much easier in practice than that explanation makes it seem. The layout of Haskell programs almost always naturally follows the structure, so having to obey the offside rule naturally encourages good layout.

Haskell notionally uses curly braces `{` and `}` and semicolons to outline the structure of code, so

```
let { two = 1+1; three = 1+two } in two + three
```

although the braces and semicolons are almost never used by human programmers. Similarly

```
> five = two + three where { two = 1+1 ; three = 1+two }
```

This punctuation is almost always left out, using layout and the offside rule to make the structure obvious.

The offside rule also applies to other constructs such as **case** expressions, which allow pattern matching like that in function definitions to appear anywhere an expression is allowed. So for example

case *test* **of** $\{$*True* $\rightarrow$ *truecase*; *False* $\rightarrow$ *falsecase*$\}$

is equivalent to

if *test* **then** *truecase* **else** *falsecase*

but might also be written without the extra punctuation using the offside rule

case *test* **of**
    $True \to truecase$
    $False \to falsecase$

2.7 A note about whole-number arithmetic

Although whole-number division is familiar, its mathematical properties might not seem as familiar as those of real-number division. However just as (exact) division behaves as a partial inverse to multiplication, in the sense that $(a/b) \times b = a$ for for all real numbers a and $b \neq 0$, there is a perfectly sound mathematical characterisation of the whole-number operation:

$$a = (a \,\mathbf{div}\, b) \times b + a \,\mathbf{mod}\, b \qquad \text{and} \qquad 0 \leqslant a \,\mathbf{mod}\, b < b$$

at least for $b > 0$. Equivalently, $a \,\mathbf{div}\, b = \lfloor a/b \rfloor$, where $\lfloor x \rfloor$ is the *floor* of x, the largest whole number not exceeding it, or $n \leqslant \lfloor x \rfloor$ if and only if $n \leqslant x$ for whole numbers n.

Haskell has another similar pair of functions **quot** and **rem** for which the absolute value of a **rem** b is less than the absolute value of b, and so **quot** rounds towards zero.

Exercises

2.1 Use the standard function $product :: [Integer] \to Integer$ to define a function $factorial$ that calculates the factorial of its argument.

Hence define a function $choose$ that makes `n `choose` r` be the number of ways of choosing r things from n.

Write a function $check$ that when applied to n returns a $Bool$ indicating whether $\sum_{r=0}^{n} \binom{n}{r} = 2^n$.

2.2 The standard function $not :: Bool \to Bool$ maps each Boolean value to the other one. Without using the not function itself, write four definitions of functions equal to not using essentially different syntactic forms.

2.3 There are two things (the values $True$ and $False$) of type $Bool$. How many different things are there with type

1. $Bool \to Bool$
2. $Bool \to Bool \to Bool$
3. $Bool \to Bool \to Bool \to Bool$
4. $(Bool, Bool)$
5. $(Bool, Bool) \to Bool$
6. $(Bool, Bool, Bool)$
7. $(Bool, Bool, Bool) \to Bool$
8. $(Bool \to Bool) \to Bool$
9. $(Bool \to Bool \to Bool) \to Bool$
10. $((Bool \to Bool) \to Bool) \to Bool$

2.4 Define in Haskell each distinct function of type $(Bool \to Bool) \to Bool$.

You might either explain how to write all of them out without actually doing so, or find a systematic way of writing a function that takes a number and produces one of them.

2.5 Show that $\lfloor x \rfloor < n$ if and only if $x < n$ for integers n.

Conclude that $\lfloor x \rfloor \leqslant x < \lfloor x \rfloor + 1$.

2.6 The *ceiling* function is defined by $\lceil x \rceil = -\lfloor -x \rfloor$. Derive properties of ceiling analogous to those of floor.

2.7 Let l and r be whole numbers with $l+1 < r$. Show that $l < (l+r)\,\mathbf{div}\,2 < r$.

3 Expressions, Types and Parametric Polymorphism

Haskell programs are statically typeable, meaning that the type of an expression can be established from the components without evaluating the expression. The language is strongly typed: an expression has no value unless its type can be inferred, or checked against a claimed type. Its value has to be one of the values of its type.

Given an expression, the interpreter checks that the expression is well-formed, then that it is consistently typed, and only then does it evaluate the expression and print it.

3.1 Well-formed, well-typed expressions

Well-formedness is purely syntactic: parentheses match, operators have arguments, constructs (like guarded equations) are complete. This can be checked in a time bounded by the size of the expression. Purely syntactic errors in programs are caught here.

Type safety involves some information about semantics, but crucially not the values of expressions. Types can be checked or inferred in a bounded amount of time, which depends on the size of a representation of the types involved. The Haskell type system is so constructed that this check is also finite (though by careful construction it is possible to make the types exponentially big in the size of the expression so it can be quite slow; it usually is not.) Many semantic errors can be caught as type errors.

Calculating the value of an expression, though, can take for ever: for example the value of an expression can be unboundedly large. Some errors will necessarily only turn up during evaluation.

3.2 Names and operators

Names consist of a letter, followed perhaps by some alphanumeric characters, followed perhaps by some primes (single quote characters).

If the initial letter is upper case, the name may be that of a type (like Int) or type constructor (like $Either$) or type class (like Eq), or it might be the name of constructor (like $Left$ or $True$). If the initial letter is lower case the name is either that of a variable or of a type variable.

Symbols made of a sequence of one or more non-alphabetical characters (excluding a few things like brackets of various sorts) behave (mostly) like infix binary operators. Some of these are predefined in the prelude, others can be defined just like any other function

```
> x +++ y = if even x then y else -y
> x // y = 2 / (1/x + 1/y)
```

They are left-associative by default, but the language allows for declaring them otherwise (like `^` which is right-associative). Parentheses convert operators into

names: $x + y = (+)\ x\ y = (x+)\ y = (+y)\ x$. Conversely, backquotes convert a name into an operator: $f\ x\ y = x\ `f`\ y$.

3.3 Types

There are some built-in types, such as Int, $Float$, $Char$. There are also some built-in type constructors, in particular the function type $a \to b$ for any types a and b. The types of lists, such as $[Int]$, will turn out to be built-in only in as much as there is a special syntax. Similarly, tuples such as pairs $(Int, Char)$, triples $(Int, Char, Bool)$, and so on. There are also empty tuples, $()$ whose sole value is $()$.

Declarations like

```
type String = [Char]
```

introduce *type synonyms*, new *names* for existing types.

You might think that $Bool$ has to be built in, but it could be (and is) defined by

```
data Bool = False | True
```

This declaration introduces a new type, $Bool$ and new constants $False$ and $True$. Similarly there are predefined types

```
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
```

As well as introducing a range of types, $Maybe\ a$ for each type a and $Either\ a\ b$ for each type a and type b, these declarations introduce

```
Nothing :: Maybe a
Just :: a -> Maybe a
Left :: a -> Either a b
Right :: b -> Either a b
```

Values of $Either\ Int\ Char$ include $Left\ 42$ and $Right$ 'x'.

A **data** declaration something akin to a record in other programming languages

```
> data PairType a b = Pair a b
```

introduces a family of types $PairType\ a\ b$ for each pair of types a and b, together with a function

```
Pair :: a -> b -> PairType a b
```

This new type is equivalent to (a, b).

The function $Pair$, exceptionally for the names of values in Haskell, has a name spelled with an upper case initial, which marks it out (like $True$ and $False$, $Left$ and $Right$ and so on) as a *constructor*: a function which by the form of its definition must be invertible. Constructors can appear in patterns on the left hand side of a function definition, such as

```
> sumPair :: PairType Int Int -> Int
> sumPair (Pair x y) = x + y
```

I might usually have chosen the same name for both $Pair$ and $PairType$. Value constructors and ~~function~~ type constructors come from two different name-spaces, so you can use the same name for both things.

3.4 Polymorphic types

Many standard functions have types that mark them out as being applicable to arguments of many types, for example

```
map :: (a -> b) -> [a] -> [b]
reverse :: [a] -> [a]
```

Types with lower case names are *type variables*, and these types should be read as being "for all types a and b, map can have type $(a \to b) \to [a] \to [b]$", and "for all types a, $reverse$ can have type $[a] \to [a]$".

When a polymorphic function is applied to an argument of a more constrained type (or vice versa) the polymorphic type is constrained to match.

If you want an instance of $reverse$ that applies only to lists of $Char$, and cannot accidentally be applied to a $[Int]$, the expression $reverse :: [Char] \to [Char]$ will do that. (This is deliberately similar to the type signature declaration.)

This is *parametric polymorphism* and the value of a parametrically polymorphic expression is very constrained by its type. For example, there is a standard function

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
```

Such a function has to satisfy

```
either f g (Left x)  = ...
either f g (Right y) = ...
```

where the right-hand sides are both of type c, whatever that type happens to be. There is in each case essentially only one possible expression known to be of that type: $f\ x$ and $g\ y$ respectively.

A parametrically polymorphic function has to operate uniformly on all possible instances of its argument type. The type of $reverse$ requires that it treats all lists the same: it cannot inspect the elements of the list because it cannot know what they are. It turns out that a consequence of this is that for any $fun :: [a] \to [a]$

$$map\ f \cdot fun = fun \cdot map\ f$$

You can easily see that this is true for $reverse$; it is also true for any other function (and there are many) of the same type; but the force of the result is that a function that does not satisfy this equation cannot have that type.

Such results, theorems that follow from the types, were called *theorems for free* in a paper by Phil Wadler; clearly a mathematical salesman.

3.5 Selectors, Discriminators, Deconstructors

A constructor like *PairType* makes a value (the jargon is a *product*) which behaves like a record with two fields. These can be recovered by *selector* functions

```
> first :: PairType a b -> a
> first (Pair x y) = x
> second :: PairType a b -> b
> second (Pair x y) = y
```

The constructors of a type like an *Either* make a value (a *sum*) which behaves like a union of two possible fields. The selectors for this type are partial functions

```
> left :: Either a b -> a
> left (Left x) = x
> right :: Either a b -> b
> right (Right y) = y
```

but they are little use without a *discriminator* that tells you which kind of value you have, for example

```
> isLeft :: Either a b -> Bool
> isLeft (Left x) = True
> isLeft (Right x) = False
```

I will try to use the term *deconstructors* for the discriminator and selectors of a type. Informally, the community often refers to the discriminator and selectors for a type as the *destructors*, although this is not a good usage: it confuses them with the operations in languages that requires the programmer to manage the memory occupied by data structures in a program.

Deconstructors for a type are enough to let you write functions on that type without pattern matching, for example

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f g e = if isLeft e then f (left e) else g (right e)
```

Unlike *the* constructors, deconstructors are not unique: the function *either* on its own is itself a perfectly good deconstructor for the *Either* type, and would more naturally have been defined by pattern matching

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f g (Left x)  = f x
either f g (Right y) = g y
```

3.6 Type inference

How does Haskell check the types of expressions? The essential rule is that if f is applied to x, then f needs to have a function type $a \to b$, and x needs to have type a, and the resulting application $f\ x$ has type b.

A definition such as

```
flip f x y = f y x
```

need not have an explicit type signature because Haskell can deduce a type for *flip* from this rule. The algorithm allocates a type variable to each name

$$\mathit{flip} :: \alpha \qquad f :: \beta \qquad x :: \gamma \qquad y :: \delta$$

and to the result of each application, then assembles the constraints:

$$\begin{array}{lcl} \mathit{flip}\ f & \Rightarrow & \alpha = \beta \to \epsilon \\ (\mathit{flip}\ f)\ x & \Rightarrow & \epsilon = \gamma \to \zeta \\ ((\mathit{flip}\ f)\ x)\ y & \Rightarrow & \zeta = \delta \to \eta \\ f\ y & \Rightarrow & \beta = \delta \to \theta \\ (f\ y)\ x & \Rightarrow & \theta = \gamma \to \iota \end{array}$$

and then finally because both sides are equal, $\iota = \eta$. These constraints can be met by substitution:

$$\begin{array}{rcl} \alpha & = & \beta \to \epsilon \\ & = & (\delta \to \theta) \to \gamma \to \zeta \\ & = & (\delta \to \gamma \to \iota) \to \gamma \to \delta \to \eta \\ & = & (\delta \to \gamma \to \iota) \to \gamma \to \delta \to \iota \end{array}$$

which (up to changing the names of the variables) is the type with which *flip* is declared.

When you are doing type inference by hand, you need not be quite so systematic. You can anticipate some of the later substitutions because, for example, you can see that *flip* is applied to three arguments in turn, so its type must have the form $\alpha \to \beta \to \gamma \to \delta$ and you can start from there.

If you give a type declaration, it will be checked against the inferred type. Type declarations, as well as being good documentation, have the advantage of improving the explanation of any type errors found.

Exercises

3.1 Generalising an earlier exercise: for finite types a, b and c there are as many functions of type $a \to b \to c$ as there are are of type $(a, b) \to c$ (because as numbers $(c^b)^a = c^{b \times a}$).

This correspondence, and the similar one for infinte types, is demonstrated by the (predefined) functions

$$\begin{array}{lcll} \mathit{curry} & :: & ((a, b) \to c) & \to \quad (a \to b \to c) \\ \mathit{uncurry} & :: & (a \to b \to c) & \to \quad ((a, b) \to c) \end{array}$$

for which both $\mathit{curry} \cdot \mathit{uncurry}$ and $\mathit{uncurry} \cdot \mathit{curry}$ are identity functions (of the appropriate type).

There is a unique fully defined function of each of these types. Write out what must be the definitions of these two functions, and prove that they are mutually inverse. (If you type these definitions at an interpreter, remember to change their names to avoid clashing with the Prelude functions.)

3.2 Suppose $h\ x\ y = f\ (g\ x\ y)$ for some f and g. Which of the following are true, which are false, and in each case why?

1. $h = f \cdot g$
2. $h\ x = f \cdot g\ x$
3. $h\ x\ y = (f \cdot g)\ x\ y$

3.3 Which of these equations are badly typed? For the others, what can you say about the type of xs, and whether and when the equation holds?

$a)\ [\,] : xs = xs$	$e)\ xs : [\,] = [xs]$	$i)\ [[\,]] +\!\!+ xs = xs$
$b)\ [[\,]] +\!\!+ [xs] = [[\,], xs]$	$f)\ [\,] : xs = [[\,], xs]$	$j)\ xs : xs = [xs, xs]$
$c)\ [[\,]] +\!\!+ xs = [xs]$	$g)\ [xs] +\!\!+ [\,] = [xs]$	$k)\ xs : [\,] = xs$
$d)\ xs : [xs] = [xs, xs]$	$h)\ [[\,]] +\!\!+ xs = [[\,], xs]$	$l)\ [xs] +\!\!+ [xs] = [xs, xs]$

3.4 Give most general types for the following, where possible

```
> subst f g x = (f x) (g x)
> fix f = f (fix f)
> twice f = f . f
> selfie f = f f
```

You should try to work out what the most general type is by hand, but you can check that you are right by using an interpreter; and if you are wrong, check that you understand why.

3.5 Give most general types for the following, where possible

```
> const x y = x
> comp x y z = x (y z)
> flip x y z = x z y
> dup x y = x y y
```

Deduce the types of each of

$dup\ const$
$comp\ (comp\ (comp\ dup)\ flip)\ (comp\ comp)$
$comp\ (comp\ dup)\ (comp\ comp\ flip)$

You should try to work out what the most general type is by hand, but you can check that you are right by using an interpreter; and if you are wrong, check that you understand why.

3.6 Time (as Richard Bird might say) for a song:

```
One man went to mow
Went to mow a meadow
One man and his dog
Went to mow a meadow

Two men went to mow
Went to mow a meadow
Two men, one man and his dog
Went to mow a meadow

Three men went to mow
Went to mow a meadow
Three men, two men, one man and his dog
Went to mow a meadow
```

Write a Haskell function $song :: Int \to String$ so that $song\ n$ is the song when there are n men (and a dog). Assume $n \leqslant 10$. To print the song, type for example: $putStr\ (song\ 5)$. You may want to start from

```
> song 1  = verse 1
> song n  = song (n-1) ++ "\n" ++ verse n
> verse n = line1 n ++ line ++ line3 n ++ line
```

3.7 Suppose that $f\ x = (x, x)$. What is the most general type of f, and what is the most general type of $f \cdot f$? Use f to explain how to construct a family of expressions, E_n, where the number of symbols in E_n is no more than n but there are at least 2^n symbols in the type expression for the type of E_n.

3.8 Let $fix\ f = f(fix\ f)$. What are the types of

fix
$fix \cdot fix$
$fix \cdot fix \cdot fix$

Show how to construct a family of expressions E_n whose size is $\mathrm{O}(n)$, but whose types have expressions whose size is $\Omega(2^n)$.

let $f = fix$ **in** $f \cdot f$
let $\{\ f = fix;\ g = f \cdot f\ \}$ **in** $g \cdot g$

Show how to construct a family of expressions E'_n whose size is $\mathrm{O}(n)$, but whose types have expressions whose size is $\Omega(2^{2^n})$.

4 Overloading and Evaluation

4.1 Evaluation

Evaluation reduces *expressions* to *values*: what is a value? One way of thinking of this is that it is (a representation of) an expression that cannot be evaluated further, often called a *normal form*. Haskell evaluates expressions by (a process equivalent to) rewriting expressions, replacing left-hand sides of equations by right-hand sides, until it reaches a normal form. There are several strategies for reducing an expression to normal form: suppose that $sq\ x = x{*}x$ then:

	Eager		**Normal order**		**Lazy**
	$sq(3+4)$		$sq(3+4)$		$sq(3+4)$
$=$	{addition}	$=$	{def of sq}	$=$	{def of sq}
	$sq\ 7$		$(3+4)*(3+4)$		**let** $x = 3+4$ **in** $x*x$
$=$	{def of sq}	$=$	{addition}	$=$	{addition}
	$7*7$		$7*(3+4)$		$7*7$
$=$	{multiplication}	$=$	{addition}	$=$	{multiplication}
	49		$7*7$		49
		$=$	{multiplication}		
			49		

Here the costs are similar, but the order is different.

```
> inf :: Integer
> inf = 1 + inf

> const :: a -> b -> a
> const x y = x
```

where *const* is standard: $const\ x\ y = x$.

	$const\ 3\ inf$		$const\ 3\ inf$
$=$	{definition of inf}	$=$	{definition of $const$}
	$const\ 3\ (1 + inf)$		**let** $\{\ x = 3; y = inf\ \}$ **in** x
$=$	{definition of inf}	$=$	3
	$const\ 3\ (1 + (1 + inf))$		
$=$	$\ldots$		

Here eager evaluation never terminates!

Eager evaluation may fail to find a normal form. If there is a normal form, normal order reduction will find it, but might be expensive. Lazy evaluation is as safe as normal order, but is usually less expensive because it tries to avoid duplicating work.

4.2 Recursive types

If the type being defined appears on the right hand side of a data definition, such as

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

it allows for recursive values: Nil, $Cons\ 1\ Nil$, $Cons\ 1\ (Cons\ 2\ Nil)$, ... are all of type $List\ Int$. In fact the built-in list type $[a]$ is like this, but with constructors $[]$ and $(:)$. The notation $[1,2,3]$ is shorthand for $1 : (2 : (3 : []))$ (and the parentheses can be left out since $(:)$ is right-associative).

Functions over recursive types are also naturally defined by recursion. Functions over lists will often be naturally expressed by two equations: one for the empty case and one for the non-empty case, and the non-empty case will often include a recursive call of the same function.

Take for example

```
map f xs = [ f x | x <- xs ]
```

then you can calculate that

```
map f    []  = []
map f (x:xs) = f x : map f xs
```

and these two equations will do perfectly well as a definition.

4.3 Recursion

This function calculates a factorial:

```
> fact :: Integer -> Integer
> fact 0 = 1
> fact n = n * fact (n-1)
```

That is, $fact\ n = n!$, at least for $n \geqslant 0$. Why is this?

Provided $n > 0$, the RHS is $n \times fact\ (n-1) = n \times (n-1)! = n!$, and if $n = 0$ the RHS is $1 = 0!$. This is effectively an induction proof in which the inductive hypothesis is that the recursive calls on the RHS satisfy the *invariant* that $fact\ n = n!$.

Why is this recursion safe, whereas that for inf is not? Because the recursive calls are all *smaller* in the sense that we can find a *variant* which is smaller (by a fixed amount) for all recursive calls, and bounded below: in this case the argument n will do.

Of course it is only safe for (whole numbers) $n \geqslant 0$.

4.4 Type classes

Some functions (such as equality) are polymorphic, but not in the same regular way as parametric polymorphism. For example

```
> data UPair a = UPair a a
```

might represent unordered pairs, for which you would want $UPair\ x\ y$ to equal $UPair\ y\ x$. The mechanism for this *ad-hoc polymorphism* in Haskell is the type class. The type of equality is

```
(==) :: Eq a => a -> a -> Bool
```

which you can read as "provided a is an Eq-type, $a \to a \to Bool$". What does it take to be an Eq-type? A definition of `(==)`!

The concept of an Eq-type is introduced by a class declaration:

```
class Eq a where
  (==) :: a -> a -> Bool
```

and a type such as $UPair\ a$ is put in that class by a matching instance declaration:

```
> instance Eq a => Eq (UPair a) where
>    UPair p q == UPair r s | p==r      = q==s
>                           | p==s      = q==r
>                           | otherwise = False
```

or more succinctly

```
> instance Eq a => Eq (UPair a) where
>    UPair p q == UPair r s = (p==r && q==s)||(p==s && q==r)
```

The instance requires $Eq\ a$ in order to implement the equality tests on the components.

It would be usual to make sure that any definition of $(==)$ behaved like equality, or at least an equivalence: that it was reflexive, and transitive. Anything else would be confusing, but nothing enforces this.

In fact the class declaration for Eq is more like

```
class Eq a where
  (==) :: a -> a -> Bool
  x == y = not (x /= y)
  (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

so that it also provides an inequality test. However an *instance* need only provide either an implementation of `(==)` or one of `(/=)` and the equation in the *class* declaration will define the other.

Similarly the instance of the equality class for lists

```
instance Eq a => Eq [a] where
  []   ==   [] = True
  x:xs == y:ys = x == y && xs == ys
  _    ==    _ = False
```

includes equality on values of a (justified by the $Eq\ a$ constraint in the header) and a recursive call of the equality test on lists.

The underlines are dummy arguments, which need not be named because their values are not used. Names could be used, but the convention of the un-named underline is good documentation because the reader does not have to check that a named argument is in fact not used.

4.5 Non-termination

What is value of an expression like $1\ `div`\ 0$? What is the value of an expression like $length\ [0\ ..\ ]$? The interpreter gives it no value, but when we are doing mathematics it is useful to have a value (so that all functions are total). By definition this is a special value $\perp$ (pronounced "*bottom*"), or rather there are values $\perp$ of every type. We identify all kinds of error with this value.

So every constructed type has an additional $\perp$ value distinct from the defined values. There are three Boolean values, and ten pairs of Booleans: the nine actual pairs in which the components take each of three values, plus an entirely undefined value.

There are many partially defined lists such as $\perp : \perp : [\,]$ which is a list of exactly two unknown things, that is $[\perp, \perp]$, and $1 : 2 : \perp$ which is a list of at least two things, the first two of which are known.

Haskell can not be expected to 'produce' a bottom when given an expression whose value is $\perp$. Sometimes it will produce an error message (for example the predefined value $undefined$ does just that), sometimes it will remain silent for a very long time, sometimes the Haskell interpreter will crash... in principle, anything can happen.

Notice that you cannot expect to test for $\perp$ in a program. It is provably impossible to decide mechanically whether an abitrary computation will terminate. The equation $f\ x = (x \neq \perp)$ makes sense as a mathematical definition, but

```
> f x = x /= bottom where bottom = bottom
```

does not compute this function!

4.6 Strict functions

Some functions always demand the value of their arguments, even with normal order reduction. For example arithmetic functions like $(+)$ need both their arguments to be evaluated: $0 + undefined$ and $undefined + 1$ are both undefined; however $const\ undefined\ 1$ is undefined, but $const\ 0\ undefined$ is zero.

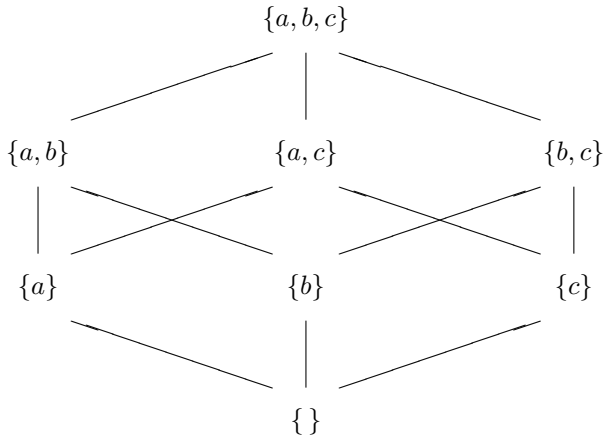
A function f is *strict* if $f\ \perp = \perp$, so $(+)$ is strict in both arguments, and $const$ is strict in its first argument but not in its second. Eager evaluation would make all functions strict, so normal order (and lazy) evaluation is more faithful to the mathematical intention. Crucially, constructors are not strict, and in particular $(:)$ is neither strict in the first element of the list nor in the rest of the list. This allows lazy computations to produce infinite lists a little bit at a time.

Strictness is a way of checking in the mathematics whether a value is needed in the computation. A function which uses its argument is necessarily strict in that argument (although a function can be needlessly strict in an argument which it otherwise ignores). A good compiler might analyse the strictness of functions to decide whether the value of an argument is going to be needed, and use the more efficient eager strategy on that argument.

4.7 Information ordering and computable functions

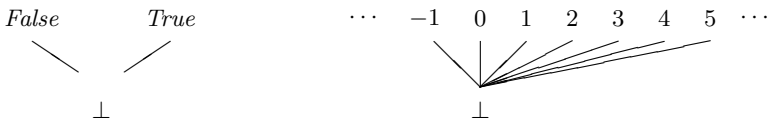
This is definitely not on the undergradate syllabus, but may be enlightening.

Recall that a *partial order* is a relation which, like the subset ordering on the subsets of a set, satisfies three conditions: it is relexive: for every x, $x \subseteq x$; it is antisymmetric: for every x and y if $x \subseteq y$ and $y \subseteq x$ then $x = y$; and it is transitive: for every x and y and z if $x \subseteq y$ and $y \subseteq z$ then $x \subseteq z$.

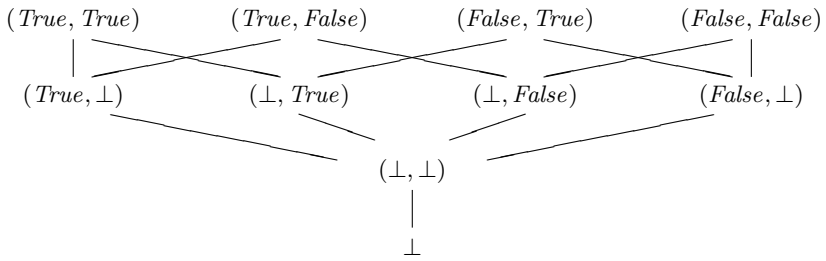


Every finite partial order can be drawn as a *Hasse diagram* in which the explicit upward edges represent just those pairs that are directly related with no value in between, and other related pairs are shown by paths up the diagram.

Informally, we can order partially evaluated expressions by how much information they yield, $x \sqsubseteq y$ if x is less useful than y but might turn into y if we did a bit more evaluation. In this ordering, $\bot \sqsubseteq y$ for all y, but for example if x and y are distinct defined integers $x \not\sqsubseteq y$ and $x \not\sqsupseteq y$.



The ten pairs of Booleans are ordered like this:



This Hasse diagram has a line going upwards between two pairs if the lower one is less well-defined than the upper one, so for example

$$\bot \sqsubseteq (\bot, \bot) \sqsubseteq (\bot, \mathit{False}) \sqsubseteq (\mathit{True}, \mathit{False})$$

and because $\sqsubseteq$ is transitive, $(\bot, \bot) \sqsubseteq (\mathit{True}, \mathit{False})$ and so on.

All computable functions are necessarily monotonic with respect to this ordering: if $x \sqsubseteq y$ then $fx \sqsubseteq fy$. It is the nature of computation that you cannot learn less about the result by supplying more information about an argument.

The monotonic functions in $\mathit{Bool} \to \mathit{Bool}$ are all strict, except for two constant functions $tt\ b = \mathit{True}$ and $ff\ b = \mathit{False}$. A (mathematical) test for equality or inequality with bottom is not monotonic, so cannot be computable. The computable function defined by

```
> f :: Bool -> Bool
> f x = x /= bottom where bottom = bottom
```

is in fact the constant function $f\ x = \bot$.

Exercises

4.1 Show that if f and g are strict, so is the composition $f \cdot g$.
Is the converse true: that if $f \cdot g$ is strict, so must f and g be?

4.2 Suppose that the class of ordered types is declared by something like

```
class Eq a => Ord a where
  (<), (<=), (>), (>=) :: a -> a -> Bool
  x <  y = not (x >= y)
  x >  y = not (x <= y)
  x >= y = x == y || x > y
```

(It includes a couple of other things, and these are not quite the default definitions.) Lists are lexicographically ordered, like the words of a dictionary. Write an instance declaration for $\mathit{Ord}\ [a]$.

4.3 If we count $\bot :: \mathit{Bool}$ as well as the proper values, there are three values of type Bool. So how many functions are there of type $\mathit{Bool} \to \mathit{Bool}$? How many of these are computable? Are all the computable ones definable in Haskell?

4.4 By evaluating expressions like `False && undefined` and others involving *True*, *False*, and *undefined* find exactly which function `(&&)` is implemented in the standard prelude. Give a definition which would produce that behaviour.

4.5 There is exactly one computable function, $(\wedge)$ say, which simultaneously satisfies all three equations

$$\begin{aligned} \mathit{False} \wedge y &= \mathit{False} \\ x \wedge \mathit{False} &= \mathit{False} \\ \mathit{True} \wedge \mathit{True} &= \mathit{True} \end{aligned}$$

for all x and y, including $\bot$. Given that it is computable, explain what this function does for each possible pair of arguments, defined or undefined, and so why there is only one such function. Is it definable in Haskell?

4.6 Explain why $\mathit{const}\ \bot$ must be equal to $\bot$ (of the appropriate function type).

4.7 Which function f is defined by the Haskell definition

```
> f x = x /= bottom where bottom = bottom
```

and why is it not possible to define in Haskell the function described by the mathematical equation $f\ x = (x \neq \bot)$?

4.8 Prove that if a computable function ever returns $\bot$, it must be strict.

4.9 A function f is *strict in its second argument* if *flip* f is strict. Show that this is equivalent to saying that $f\ x$ is strict for all x.

5 Defining functions on lists

When a new data type is introduced by a **data** declaration, such as

```
data Bool = False | True
```

functions from that type are naturally defined by pattern matching using the constructors.

```
not :: Bool -> Bool
not False = True
not True  = False
```

Notice that the constructors are *constants*, and you cannot pattern match with any other expressions that happen to be equal to them.

More generally, pattern matching can cover a range of values and bind local variables to the values of components

```
data Either a b = Left a | Right b

either :: (a -> c) -> (b -> c) -> Either a b -> c
either left right (Left x)  = left x
either left right (Right y) = right y
```

The names of *left* and *right* are chosen because $either\ Left\ Right$ is the identity on $Either\ a\ b$.

Similarly, functions from a recursive data type like

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

will be definable by pattern matching

```
f :: List a -> ...
f  Nil        = ...
f (Cons x xs) = ... x ... xs ...
```

though it would not be surprising were there a recursive call of f on xs.

The test for emptiness of a sequence,

```
null :: [a] -> Bool
null    []  = True
null (x:xs) = False
```

is defined by pattern matching. Since x and xs are not used, indeed the type tells you that they are not used, the pattern matching non-null lists could have been `null (_:_)`. Note that *null* has to be strict, because the pattern matching has to determine which equation applies.

A more interesting example is *map*, which was introduced earlier as

```
map :: (a -> b) -> ([a] -> [b])
map f xs = [ f x | x <- xs ]
```

which you can show satisfies

$$\begin{aligned} map\ f\ [\,] &= [f\ y \mid y \leftarrow [\,]] \\ &= [\,] \\ map\ f\ (x : xs) &= [f\ y \mid y \leftarrow (x : xs)] \\ &= [f\ x] +\!\!+ [f\ y \mid y \leftarrow xs] \\ &= f\ x : map\ f\ xs \end{aligned}$$

These two equations serve as (and are the standard) definition of map. They identify the value of $map\ f$ on any finite list, and on infinite lists.

5.1 Partial functions

Some functions will be partial:

```
> head :: [a] -> a
> head (x:_) = x

> tail :: [a] -> [a]
> tail (_:xs) = xs
```

and can be defined without giving a second equation. Applying such a function to values which do not match is an error.

The other end of a non-empty list can be accessed by

```
> last :: [a] -> a
> last [x]    = x
> last (_:xs) = last xs
```

The $[x]$ notation is just an abbreviation for the pattern $(x : [\,])$ made only of constructors ($[\,]$ and $(:)$) and the variable x which matches the (last) element of a singleton. The second equation overlaps with the first, so the order of these equations matters. You might prefer

```
last (_:y:ys) = last (y:ys)
```

in which the pattern matches only lists of at least two elements, and so is disjoint from $[x]$. The disadvantage of this equation is that (when read as a rewriting rule) it takes $y : ys$ apart into its components, and then puts them together with a new $(:)$.

Catenation, $xs +\!\!+ ys = concat[xs, ys]$, also follows a similar scheme.

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

The resulting function is strict in left argument, $\bot +\!\!+ [3,4] = \bot$ because of the pattern matching; but not strict in right $[1,2] +\!\!+ \bot = 1 : 2 : \bot \neq \bot$. (You can tell that $1 : 2 : \bot \neq \bot$, because $head\ (1 : 2 : \bot) = 1 \neq \bot = head\ \bot$.)

Notice that the cost of (the $(+\!\!+)$ in) $xs +\!\!+ ys$ is proportional to $length\ xs$.

```
> length :: [a] -> Int
> length    []  = 0
> length (_:xs) = 1 + length xs
```

The same pattern of recursion happens in map, $(+\!\!+bs)$ and $length$.

5.2 A natural pattern

This same pattern also occurs in functions such as

```
> sum :: Num a => [a] -> a
> sum    []  = 0
> sum (x:xs) = x + sum xs
```

and $product$, in $filter\ p\ xs = [x \mid x \leftarrow xs, p\ x]$

```
> filter :: (a -> Bool) -> [a] -> [a]
> filter p    []  = []
> filter p (x:xs) | p x       = x:rest
>                 | otherwise = rest
>       where rest = filter p xs
```

and in $takeWhile\ p\ xs$ which returns a maximal initial segment of xs all of which satisfies p

```
> takeWhile :: (a -> Bool) -> [a] -> [a]
> takeWhile p [] = []
> takeWhile p (x:xs) | p x       = x:rest
>                    | otherwise = []
>       where rest = takeWhile p xs
```

and many others, including $concat\ xss = [x +\!\!+ xs \leftarrow xss, x \leftarrow xs]$

```
> concat :: [[a]] -> [a]
> concat      []  = []
> concat (xs:xss) = xs ++ concat xss
```

Abstracting this pattern leads to

```
> fold :: (a -> b -> b) -> b -> [a] -> b
> fold cons nil    []  = nil
> fold cons nil (x:xs) = cons x (fold cons nil xs)
```

This is (nearly) a standard function: it is essentially *foldr* and *foldr* is defined this way in most books. In more recent implementations of Haskell, *foldr* has a slightly more abstract type

```
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

The list type constructor is *Foldable*, so

```
fold = foldr :: (a -> b -> b) -> b -> [a] -> b
```

but you can (almost always) use *foldr* for *fold* and rely on the context to identify the type.

It is often possible to spot the right $cons$ and nil values to implement a given function. However if a function is a fold, it is always possible to compute them.

Suppose that $map\ f = fold\ cons\ nil$, then solve for

$$
\begin{array}{ll}
 & nil \\
= & \{\text{definition of } fold\} \\
 & fold\ cons\ nil\ [\,] \\
= & \{\text{assumption}\} \\
 & map\ f\ [\,] \\
= & \{\text{definition of } map\} \\
 & [\,]
\end{array}
$$

Solving for $cons$ is slightly harder:

$$
\begin{array}{ll}
 & cons\ x\ (fold\ cons\ nil\ xs) \\
= & \{\text{definition of } fold\} \\
 & fold\ cons\ nil\ (x : xs) \\
= & \{\text{assumption}\} \\
 & map\ f\ (x : xs) \\
= & \{\text{definition of } map\} \\
 & f\ x : map\ f\ xs \\
= & \{\text{assumption, for a smaller argument}\} \\
 & f\ x : fold\ cons\ nil\ xs
\end{array}
$$

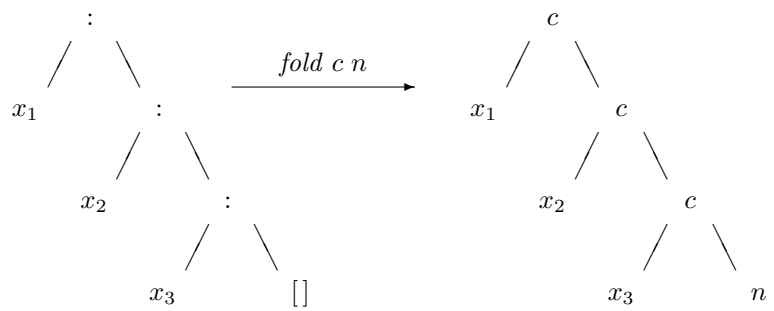
and whilst this equation is not itself a definition of $cons$, it can be generalised into one: it is certainly satisfied if $cons\ x\ ys = f\ x : ys$ for all ys. Similarly you can calculate that

$$
\begin{array}{rcl}
sum & = & fold\ (+)\ 0 \\
product & = & fold\ (\times)\ 1 \\
(+\!\!+ bs) & = & fold\ (:)\ bs \\
concat & = & fold\ (+\!\!+)\ [\,]
\end{array}
$$

and so on.

Of course if f is not a fold, you can try to go through the same calculation, but you will not get a solution to $f = fold\ cons\ nil$. For example, if f is not strict, it cannot equal the right-hand side (exercise 5.5).

The effect of *fold* is to substitute its arguments for the constructors, $(:)$ and $[\,]$, of a list; for example $fold\ c\ n$ applied to $[x_1, x_2, x_3]$



Notice that $fold\ (:)\ [\,] = id$.

In the same way $either\ left\ right$ substitutes *left* and *right* for the constructors *Left* and *Right* of $Either\ a\ b$ and you can think of it as the fold for *Either* types, and $either\ Left\ Right = id$. Of course, *either* is not a recursive function, but then neither is $Either\ a\ b$ a recursive type.

Given a data type definition, there is a natural fold function which substitutes its arguments for the constructors of the type, and which when applied to the constructors yields the identity function. The *fold* function is recursive where the type is recursive.

As a footnote: the fold is unique up to the order in which the arguments appear. The order of the alternatives in a **data** definition is almost immaterial, and it might have seemed more natural to have the arguments to *fold* in the same order as the constructors of the list type. However that proves confusing because of the history of *foldr*.

Exercises

5.1 The predefined functions

```
take :: Int -> [a] -> [a]
drop :: Int -> [a] -> [a]
```

divide a list into an initial segment and the rest, so that $take\ n\ xs +\!\!+ drop\ n\ xs = xs$ and $take\ n\ xs$ is of length n or $length\ xs$, whichever is less.

Write your own definitions for these functions and check that they give the same answer as the predefined functions for some representative arguments. Is $take\ n\ xs$ strict in n? Is it strict in xs? Can it be strict in neither?

5.2 Is *map* strict? Is $map\ f$ strict?

5.3 Define a function $evens :: [a] \to [a]$ which returns a list of the elements of its input that are in even numbered locations:

```
*Main> evens ['a'..'z']
"acegikmoqsuwy"
```

and a function $odds$ of the same type which returns the remaining elements. (Hint: you might use the one function in defining the other...)

Suppose you need both $evens\ xs$ and $odds\ xs$ for the same xs. Find an alternative definition for

```
> alts :: [a] -> ([a],[a])
> alts xs = (evens xs, odds xs)
```

which calculates the result in a single pass along the list.

Ideally, you should derive the definition showing that it is right.

5.4 Suppose $f\ xs = (fold\ c\ n\ xs, fold\ d\ m\ xs)$. Can f itself be expressed as a fold?

5.5 Suppose we solve $fold\ cons\ nil = const\ x$ for $cons$ and nil, ignoring the non-strictness of $const\ x$, what is the resulting function $fold\ cons\ nil$.

6 Sorting

To sort a list, the elements of the list must be orderable, and sorting amounts to finding a permutation of the list that is sorted.

```
> sort :: Ord a => [a] -> [a]
> sort = head . filter sorted . permutations
```

A list is *sorted* if every adjacent pair of elements of the list is in the right order.

```
> sorted :: Ord a => [a] -> Bool
> sorted = all ordered . pairs where ordered = uncurry (<=)
```

Here $all\ p = and \cdot map\ p$ is predefined, as is and which is the fold of $(\&\&)$, and $uncurry$ turns a function of two arguments into one that takes a pair.

There is a handy idiom for the adjacent pairs of a list

```
> pairs :: [a] -> [(a, a)]
> pairs xs = xs `zip` tail xs
```

which takes two copies of the list, shifts one along by a place, and makes pairs of corresponding elements with the predefined function zip which might have been (but is not in fact) defined by

```
> zip :: [a] -> [b] -> [(a, b)]
> zip (x:xs) (y:ys) = (x,y) : zip xs ys
> zip    _      _   = []
```

Notice that it discards any of either list that is left over after reaching the end of the other.

What about the $permutations$ of a list? One natural definition considers all the permutations that begin with each element of the list:

```
> permutations [] = [[]]
> permutations xs =
>         [ x:ys | x <- xs, ys <- permutations (delete x xs) ]
```

which produces the list of permutations in lexicographical order, in the sense that $permutations\ [1\,..\,n]$ is in sorted order. Unfortunately excluding an element requires a test for equality

```
> delete x xs =
>           takeWhile (/= x) xs ++ tail (dropWhile (/= x) xs)
```

so this definition gives a function whose type needs Eq on the type of elements. (The need for Eq can be avoided by generating x and $delete\ x\ xs$ at the same time, but doing that is a little involved and left as exercise 6.4.)

An alternative definition considers all permutations of the tail of the list, and inserts the head into each possible position in the list.

```
> permutations' :: [a] -> [[a]]
> permutations' [] = [[]]
> permutations' (x:xs) =
>          [ zs | ys <- permutations' xs, zs <- include x ys ]

> include :: a -> [a] -> [[a]]
> include x    []  = [[x]]
> include x (y:ys) = (x:y:ys) : map (y:) (include x ys)
```

Of course the result is that the permutations are in a different order, but the order does not matter here.

The only problem with this definition of $sort$ is that it takes infeasibly long for quite small inputs because of the huge number of permutations. There are $n!$ permuations of n things, and in the worst case $sort$ must check them all, so takes at least a time proportional to $n!$. Sorting a list of twenty things takes $20!/10! \approx 6 \times 10^{11}$ times as long as sorting ten things.

6.1 Insertion sort

One way to think of a better strategy is to imagine picking up and sorting a hand of cards. Many players will keep a partial hand in sorted order, inserting each new card into its proper place.

```
> isort :: Ord a => [a] -> [a]
> isort    []  = []
> isort (x:xs) = insert x (isort xs)
```

This is of course another instance of $fold$:

```
> isort' :: Ord a => [a] -> [a]
> isort' = fold insert []
```

Inserting an element into a sorted list is a matter of finding the right place:

```
> insert :: Ord a => a -> [a] -> [a]
> insert x xs = takeWhile (<x) xs ++ [x] ++ dropWhile (<x) xs
```

Notice that on every call of $insert$ the xs will be sorted, so in consequence the result will also be sorted (and so the result of $isort$ will be sorted).

Since $insert$ is in the worst case linear in the length of the list into which it inserts, and is called a number of times linear in the length of the input, insertion sort takes an effort quadratic in the length of the input.

6.2 Selection sort

Selection sort works in the opposite way: the sorted hand is built up in order by extracting the smallest element (which is the head of the answer) and then proceeding to sort the remainder.

```
> ssort :: Ord a => [a] -> [a]
> ssort [] = []
> ssort xs = y : ssort ys
>            where y = minimum xs
>                  ys = delete y xs
```

The predefined function *minimum* is a fold (on non-empty lists), and we met *delete* earlier. However *ssort* is itself not obviously a fold; it is however an instance of an *unfold*.

The (non-standard) function *unfold* is in a sense dual to *fold*. Lists are constructed from the constructors $(:)$ and $[\,]$ and it is a property of folds that $fold\ (:)\ [\,] = id$. Dually, lists can be deconstructed by *head* and *tail* having first checked whether they are *null*. The dual of $fold\ (:)\ [\,] = id$ is that $unfold\ null\ head\ tail = id$. Provided that

$$\begin{aligned} p\ n &= True \\ p\ (c\ x\ y) &= False \\ h\ (c\ x\ y) &= x \\ t\ (c\ x\ y) &= y \end{aligned}$$

it can be proved that $unfold\ p\ h\ t \cdot fold\ c\ n = id$.

One way of defining *unfold* would be

```
> unfold :: (b->Bool) -> (b->a) -> (b->b) -> b -> [a]
> unfold null head tail = u
>     where u x = if null x then [] else head x : u (tail x)
```

So for example

```
*Main> unfold (==0) (`mod`10) (`div`10) 123456
[6,5,4,3,2,1]
```

Selection sort can be written as an unfold

```
> ssort' = unfold null minimum deleteMin
>          where deleteMin xs = delete (minimum xs) xs
```

showing that it is a mechanism in this sense dual to insertion sort.

This definition is less efficient than the direct recursion, since it has to calculate the *minimum* twice for each tail of the input. (This is an artefact of our definition of *unfold* and can be eliminated.)

However we do it, selection sort is similarly quadratic in the length of the input. You might know, or might perhaps learn next term that a sorting function with the type of *sort* has in the worst case to take at least $\mathrm{O}(n \log n)$ work on a list of length n.

6.3 Quicksort

Quicksort was invented in about 1960 by Tony Hoare. He had been a computer salesman, and perhaps the very best thing about Quicksort is the compelling (and not entirely inaccurate) name.

This is a particular case of the *divide and conquer strategy* for breaking down big problems into smaller ones, until there are only easy problems left. The idea is to pick a value from the list, and to divide the rest of the list into smaller and larger elements. These can then be sorted independently and the sorted answer is just the concatenation of the sorted sublists.

```
> qsort :: Ord a => [a] -> [a]
> qsort    []  = []
> qsort (x:xs) = qsort [ y | y <- xs, y < x  ] ++
>                  x : [ y | y <- xs, y == x ] ++
>                qsort [ y | y <- xs, y > x  ]
```

This recursion works because each of the calls on the right-hand side has an argument shorter than the one on the left-hand side.

The idea is that if you are lucky, the so called pivot x will divide the list in half and the recursion never becomes more than $\log n$ deep, so the total amount of work never exceeds $\mathrm{O}(n \log n)$. Unfortunately if you happen to have a pivot which is an extreme value (because the input was already sorted, or reverse-sorted) this algorithm is also quadratic.

You will find that presentations of Quicksort normally involve elements of arrays, and are full of array indexes. (In fairness, they are achieving more than our version because they sort the array *in place* and often access the array in fairly predictable ways, both of which matter if you have a huge collection to sort and a cached memory system.) One of the beauties of functional programming is the ability to present an algorithm like this in a more abstract way that clearly shows its structure.

6.4 Merge sort

In order to guarantee worst-case performance as good as $\mathrm{O}(n \log n)$ a divide-and-conquer algorithm has to guarantee a fairly even split into subtasks. One such algorithm is *merge sort* probably invented about 1945 by John von Neumann. Provided there are at least two elements in the list, it can be divided into two smaller lists which can be sorted recursively, reducing the original problem into one of merging two sorted runs into one.

```
> msort :: Ord a => [a] -> [a]
> msort []  = []
> msort [x] = [x]
> msort xs  = merge (msort ls) (msort rs)
>             where (ls,rs) = halve xs
```

Merging two runs can be done by a recursion not entirely unlike that for *zip*

```
> merge :: Ord a => [a] -> [a] -> [a]
> merge (x:xs) (y:ys) | x <= y    = x : merge xs (y:ys)
>                     | otherwise = y : merge (x:xs) ys
> merge    []     ys  = ys
> merge    xs     []  = xs
```

This *merge* is linear in the size of the result, so if *halve* guarantees two half-sized sub-problems, *msort* is guaranteed asymptotically optimally even in the worst case.

As written, *merge* breaks ties between equal values by putting the value from the left run first, so if *halve* divides a list into an initial segment and a final segment the resulting *msort* will be *stable*: it does not permute those values which compare as equal to each other.

Finally

```
> halve xs = (take n xs, drop n xs) where n = length xs `div` 2
```

although this version unnecessarily makes three passes over much of the list.

6.5 Some optimisations

As happened in this lecture, often when one wants $take\ n\ xs$ one also needs $drop\ n\ xs$ for the same n and xs. It seems wasteful to compute both, since the computation of one necessarily discovers the other. There is a predefined function

```
> splitAt :: Int -> [a] -> ([a], [a])
```

which satisfies

```
splitAt n xs = (take n xs, drop n xs)
```

although that is of course not the defining equation. So

```
> halve xs = splitAt (length xs `div` 2) xs
```

makes only two passes through the list. I chose to use the separate functions for clarity.

Similarly there is a function

```
span :: (a -> Bool) -> [a] -> ([a], [a])
```

which satisfies

```
span p xs = (takeWhile p xs, dropWhile p xs)
```

but calculates the result in one pass over the list. So for example

```
> insert x xs = before ++ [x] ++ after
>               where (before, after) = span (<x) xs
```

You might expect that the point of these paired functions was to save time, and if applying p is expensive that might be the case. If p is cheap however, the extra book-keeping means that not much time is saved.

The real point of *span* is that it saves space.

Separate computations of $takeWhile\ p\ xs$ and $dropWhile\ p\ xs$ have to happen in some order or another. While one is happening, the whole of xs has to be kept in existence for the as yet unevaluated computation of the other. However a call of *span* can discard parts of xs as they are used up in the $takeWhile$ result, since they are known not to be needed in the $dropWhile$.

In practice, it is much more likely that one would eliminate the $takeWhile$ and $dropWhile$ by writing functions like *insert* directly

```
> insert x    []              = [x]
> insert x (y:ys) | y >= x    = x : y : ys
>                 | otherwise = y : insert x ys
```

Exercises

6.1 In the lecture, *zip* was defined by two equations whose left-hand side patterns overlapped. The order of these two equations matters: what happens if they are switched? Find a set of equations defining *zip* where the order of the equations does not matter.

6.2 The predefined function

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

clearly, from its type, is related to *zip*. Give a definition of $zipWith$ in terms of *zip* and other standard functions.

In practice, $zipWith$ is defined directly and *zip* is then defined in terms of $zipWith$. Write a recursive definition of $zipWith$ and use it to define *zip*.

6.3 Show that $zipWith\ f$ can be written as a fold.

6.4 Write (perhaps using *unfold* to do so) a function

```
> splits :: [a] -> [(a,[a])]
```

for which *splits* xs is a list of all the $(x, as +\!\!+ bs)$ that satisfy $as +\!\!+ [x] +\!\!+ bs = xs$, so that you can get permutations in lexicographical order by

```
> permutations [] = [[]]
> permutations xs =
>     [ x:zs | (x,ys) <- splits xs, zs <- permutations ys ]
```

(Hint: start by trying to write an *unfold* which returns a list of all the (x, bs) that satisfy $as +\!\!+ [x] +\!\!+ bs = xs$, and see if you can modify that.)

6.5 The function *permutations′*

```
> permutations' :: [a] -> [[a]]
> permutations' [] = [[]]
> permutations' (x:xs) =
>       [ zs | ys <- permutations' xs, zs <- include x ys ]
```

has the form of a fold, as (almost) does *include x*

```
> include :: a -> [a] -> [[a]]
> include x    []  = [[x]]
> include x (y:ys) = (x:y:ys) : map (y:) (include x ys)
```

Rewrite them to use *fold* (or *foldr*) and no explicit recursion.

(Hint: in rewriting *include x* you need to be able to recover *ys* from the value of *include x ys*.)

6.6 The standard function *iterate* is equal to *unfold* (*const False*) *undefined*. Find a recursive definition of *iterate*.

Use this to show that *unfold null head tail* can be written as a composition of calls of *map*, *takeWhile*, and *iterate*.

6.7 Calls of *unfold null head tail* are often inefficient because at least two of the parameter functions have to do the same work and each is obliged to do it for itself. One possible solution to this problem is to have a parameter modelled on the library function

```
> uncons :: [a] -> Maybe (a, [a])
> uncons    []  = Nothing
> uncons (x:xs) = Just (x,xs)
```

from *Data.List*, which is an alternative deconstructor for $[a]$.

Write out a definition for the function

```
> unfoldr :: (b -> Maybe (a,b)) -> b -> [a]
```

(which is in the same library) and use it to implement selection sort more efficiently.

6.8 It might be argued that the deconstructor *uncons* in exercise 6.7 merely translates $[a]$ lists into *Maybe a* values which then have themselves to be deconstructed. An alternative would be to use as a deconstructor

```
> uncons' :: b -> (a -> [a] -> b) -> [a] -> b
> uncons' nil cons    []  = nil
> uncons' nil cons (x:xs) = x `cons` xs
```

What is the unfold function *unfoldr′* corresponding to this deconstructor, for which *unfoldr′ uncons′* is the identity on lists?

Show that *null*, *head*, and *tail* can all be implemented by calls of *uncons′*.

Show that *fold* can be defined without using pattern-matching or any other form of deconstruction except a call of *uncons′*.

Use *unfoldr′* to reimplement selection sort more efficiently.

6.9 A variation of insertion sort uses a carefully ordered tree to keep the partially sorted values. In this question a *binary search tree* is an element of

```
> data Tree a = Fork (Tree a) a (Tree a) | Empty
```

which is ordered so that all values that appear in the left tree of a fork are smaller than the value at the fork, and all values that appear in forks in the right subtree are bigger.

In order to deal with repetitions in this question use a $Tree\ [a]$ to contain the elements of a list of type $[a]$.

Write a function $insert :: Ord\ a \Rightarrow a \to Tree\ [a] \to Tree\ [a]$ which inserts a value into a tree, keeping its ordering property.

Write a function $flatten :: Tree\ [a] \to [a]$ which takes a binary search tree and produces a list of its elements in order.

Use these to write

```
> bsort :: Ord a => [a] -> [a]
> bsort = flatten . foldr insert Empty
```

6.10 Given the specification

```
> insert x xs = takeWhile (<x) xs ++ [x] ++ dropWhile (<x) xs
```

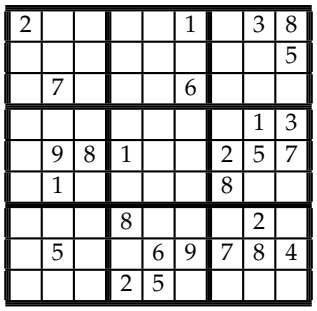
show by calculation that $insert$ satisfies

```
> insert x    []              = [x]
> insert x (y:ys) | y >= x    = x : y : ys
>                 | otherwise = y : insert x ys
```

6.11 Show that the standard function $take$ can be written as an unfold (from the pair of its arguments).

7 Sudoku (fit the first)

Standard Sudoku puzzles consist of a 9×9 grid of squares, some of which are blank and the others which contain a single digit drawn from 1 to 9.



```
["2....1.38",
 "........5",
 ".7...6...",
 ".......13",
 ".981..257",
 "31....8..",
 "9..8...2.",
 ".5..69784",
 "4..25...."]
```

The solution is the unique square obtained by filling the empty squares with single digits so that each row, each column, and each of the non-overlapping 3×3 blocks contains all digits exactly once.

We don't deign to solve the puzzle, what we do is write a function that computes all possible completions of a puzzle.

```
> type Solver = Grid -> [ Grid ]
```

7.1 Specifying the problem

A grid will be a matrix of digits. The contents of a cell in the grid will be a digit or a blank: we might use a data-type with ten values and have the type checker make sure that we always have valid contents in a cell, but it will be easier not to. We assume the input contains only the ten valid characters. For pragmatic reasons, blanks are represented by a visible character. Similarly we choose to represent a matrix as a list of rows, and a row as a list of cells, so that we can use familiar list operations.

```
> type Grid = Matrix Digit
> type Digit = Char

> type Matrix a = [ Row a ]
> type Row a    = [ Cell a ]
> type Cell a   = a

> digits :: [ Digit ]
> digits = ['1'..'9']

> blank :: Digit -> Bool
> blank = (== '.')
```

Perhaps the most straightforward way of defining a solver

```
> solve :: Solver
> solve = filter legal . expand . freedoms
```

is to list all possible choices available for each cell, consider all possible grids that can be made to match those possibilities, and filter for the one(s) that are legal solutions.

```
freedoms :: Grid -> Matrix Freedoms
expand   :: Matrix Freedoms -> [ Grid ]
legal    :: Grid -> Bool
```

We can represent the freedom for each cell as a list of possible (genuine) digits

```
> type Freedoms = [ Digit ]

> freedoms :: Grid -> Matrix Freedoms
> freedoms = map ( map choice )
>            where choice d | blank d   = digits
>                           | otherwise = [d]
```

Expanding a matrix of such lists is the Cartesian product for matrices.

```
> expand :: Matrix Freedoms -> [ Grid ]
> expand = cp . map cp
```

where the Cartesian product for lists

```
> cp :: [[a]] -> [[a]]
> cp     []   = [[]]
> cp (xs:xss) = [ x:ys | x <- xs, ys <- cp xss ]
```

lists all possible ways of choosing exactly one value from each list in a list of lists. For example, $cp\ [[1,2],[3],[4,5]] = [[1,3,4],[1,3,5],[2,3,4],[2,3,5]]$.

(Oh, look, cp is a fold.)

Finally, an acceptable grid is one in which no row, column or box contains duplicates:

```
> legal :: Grid -> Bool
> legal g = all nodups (rows g) &&
>           all nodups (cols g) &&
>           all nodups (boxs g)
```

The most direct implementation of *nodups* is

```
> nodups :: Eq a => [a] -> Bool
> nodups    []  = True
> nodups (x:xs) = x `notElem` xs && nodups xs
```

where the function

```
notElem x xs = all (/= x) xs
```

is predefined.

This test takes $O(n^2)$ steps on a list of length n. We might choose to sort the list and check that it is strictly increasing, and sorting can be done in $O(n \log n)$ steps so this is $O(n \log n)$. However for small problems, here $n = 9$, it is not clear that using an efficient sorting algorithm is worthwhile because of the constant of proportionality. (Would you prefer $9n^2$ steps or $30n \log_2 n$ steps?)

7.2 Rows, columns and box(e)s

Since a matrix is given by a list of its rows, the function *rows* is just the identity function on matrices:

```
> rows :: Matrix a -> [ Row a ]
> rows = id
```

and the function *cols* computes the transpose of a matrix:

```
> cols :: Matrix a -> [ Row a ]
> cols    [xs]  = [ [x] | x <- xs ]
> cols (xs:xss) = zipWith (:) xs (cols xss)
```

The predefined function

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

zips together a pair of lists, like *zip*, but by applying its function argument to each pair of corresponding elements to produce the corresponding result.

(Oh, look, *cols* is almost a fold... in fact it is a fold on non-empty lists, but by extending it to empty lists it can be made to be an instance of *fold*.)

Dividing a list into groups of n can be done by

```
> by :: Int -> [a] -> [[a]]
> by n [] = []
> by n xs = take n xs : by n (drop n xs)
```

(which is an unfold, known as *chunk* in some libraries).

The function that identifies the squares that are ninths of the grid

```
> boxs :: Matrix a -> [ Row a ]
> boxs = map concat . concat . map cols . by 3 . map (by 3)
```

divides each row into threes, parts the rows into threes, transposes the small matrices in each part, and then joins the whole back into a single list of the boxes.

This might be better illustrated by a similar transformation of a smaller 4×4 matrix into quarters:

```
[[a,b,c,d],       [[[a,b],[c,d]],      [[[[a,b],[c,d]],
 [e,f,g,h],   ⟶    [[e,f],[g,h]],   ⟶    [[e,f],[g,h]]]   ⟶
 [i,j,k,l],        [[i,j],[k,l]],       [[[i,j],[k,l]],
 [m,n,o,p]]        [[m,n],[o,p]]]        [[m,n],[o,p]]]]

  [[[[a,b],[e,f]],        [[[a,b],[e,f]],       [[a,b,e,f],
    [[c,d],[g,h]]]   ⟶     [[c,d],[g,h]],   ⟶    [c,d,g,h],
   [[[i,j],[m,n]],         [[i,j],[m,n]],        [i,j,m,n],
    [[k,l],[o,p]]]]        [[k,l],[o,p]]]        [k,l,o,p]]
```

One of the many virtues of functional programming is that it encourages this holistic style of programming, which manages whole objects rather than fiddling with many subscripts.

7.3 Infeasibility

This completes the implementation of

```
> solve :: Solver
> solve = filter legal . expand . freedoms
```

This implements exactly the function that maps a Sudoku problem to its solution. It is however entirely useless for solving Sudoku problems.

The uniqueness of the solution can be enforced by as few as 17 non-blank squares in the puzzle, so there can be $9^{9\times9-17} \approx 10^{61}$ potential solutions to be checked. There are about 3×10^7 seconds in a year, so checking 10^{61} grids at the rate of a million a second would take of the order of 3×10^{47} years, which compares unfavourably with estimates of the age of the universe of the order of 10^{10} years.

Even a relatively easy problem such that given earlier in the notes will have many fewer than half of the squares filled in. No technological improvement is going to help here, nor is any realistic amount of concurrent computation.

This algorithm is not *slow*, it is *infeasible*.

7.4 Holistic reasoning

Thinking about whole data structures makes it easier to reason about programs, helped by the *point-free* (or *tacit*) style which encourages the omission of unnecessary arguments, so

```
> solve = filter legal . expand . freedoms
```

rather than

```
> solve xss = filter legal (expand (freedoms xss))
```

Not only are there no subscripts, nor individual elements of the grids, mentioned in the program. Point-free style here omits all mention of the grid itself, except perhaps in the type.

For example, the grid arranging functions are all their own inverses

$$rows \cdot rows = cols \cdot cols = boxs \cdot boxs = id$$

at least on 9×9 matrices. Moreover,

$$\begin{aligned} map\ rows \cdot expand &= expand \cdot rows \\ map\ cols \cdot expand &= expand \cdot cols \\ map\ boxs \cdot expand &= expand \cdot boxs \end{aligned}$$

on all 9×9 matrices of choices. There are also a wealth of laws about standard functions like

$$\begin{aligned} map\ f \cdot concat &= concat \cdot map\ (map\ f) \\ filter\ p \cdot concat &= concat \cdot map\ (filter\ p) \\ map\ f \cdot filter\ (p \cdot f) &= filter\ p \cdot map\ f \\ filter\ (all\ p) \cdot cp &= cp \cdot map\ (filter\ p) \end{aligned}$$

Exercises

7.1 Express the Cartesian product function

```
> cp :: [[a]] -> [[a]]
> cp []       = [[]]
> cp (xs:xss) = [ x:ys | x <- xs, ys <- cp xss ]
```

from the lectures as an instance of *fold* (or the standard function *foldr*).

7.2 The function

```
> cols :: [[a]] -> [[a]]
> cols    [xs]  = [ [x] | x <- xs ]
> cols (xs:xss) = zipWith (:) xs (cols xss)
```

is not quite in the form of a fold (on lists) because there is a special case for singelton lists. Without changing the third equation, define a function $cols'$ which agrees with $cols$ wherever that function is defined, and for which $cols'\ [\,]$ has a value that makes the equation for $cols\ [xs]$ redundant. This should give the definition of $cols'$ the form of a fold. Finally, write $cols'$ as an instance of *fold*.

7.3 A type of non-empty lists with elements of type a might have been defined by

```
> data Pist a = Wrap a | Pons a (Pist a)
```

Recall that the fold function for lists

```
> fold :: (a -> b -> b) -> b -> [a] -> b
```

can be characterised by $fold\ (:)\ [\,]$ being the identity function on lists. The corresponding property for the fold on non-empty lists

```
>  foldp :: ... -> ... -> Pist a -> b
```

is that $foldp\ Pons\ Wrap$ is the identity on $Pist\ a$.

Complete the definition of $foldp$, and write functions

```
pist :: [a] -> Pist a
list :: Pist a -> [a]
```

which embed $Pist\ a$ in $[a]$, so that for example

$$pist\ [1,2,3] = Pons\ 1\ (Pons\ 2\ (Wrap\ 3))$$

and

$$list\ (Pons\ 1\ (Pons\ 2\ (Wrap\ 3))) = [1,2,3]$$

What can you say about $list \cdot pist$ and $pist \cdot list$?

Let $fold1\ cons\ wrap = foldp\ cons\ wrap \cdot pist$. Calculate, or write out, a direct recursive definition of $fold1$.

Show that $cols$ can be expressed as an instance of $fold1$.

There is a predefined function

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f   [x]  = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

Can you write $foldr1\ f$ as an instance of $fold1$?

Can you write $fold1\ cons\ wrap$ as an instance of $foldr1$?

8 Sudoku (fit the second)

The naïve solver from the previous lecture

```
> solve = filter legal . expand . freedoms
```

was too inefficient because it might have to consider as many as $9^{9\times 9-17}$ possible matrices of choices.

8.1 Pruning

One way of cutting down on the scale of the search is to remove from the freedoms in each cell those elements that are already committed to another cell in the same row, column or box. We need a function $prune$ which satisfies

$$filter\ legal \cdot expand \quad = \quad filter\ legal \cdot expand \cdot prune$$

But how?

A single row can be pruned by

```
> pruneRow :: Row Freedoms -> Row Freedoms
> pruneRow row = map (remove (ones row)) row
```

removing from each cell of the row any element that is already fixed somewhere in that row. The committed values are the ones that appear as a singleton

```
> ones :: [[a]] -> [a]
> ones row = [ d | [d] <- row ]
```

or, if you prefer (and I might do so)

```
> ones = map head . filter singleton
```

Removing elements from a cell is almost a filter,

```
> remove :: Freedoms -> Freedoms -> Freedoms
> remove xs [d] = [d]
> remove xs ds  = filter (`notElem` xs) ds
```

but had better leave the fixed choices themselves in place.

This function satisfies

$$filter\ nodups \cdot cp = filter\ nodups \cdot cp \cdot pruneRow$$

because any element of the Cartesian product of a row of choices contains every one of the singletons, and so can only be free of duplicates if it does not choose one of these from any other cell.

8.2 Promoting to rows, columns and boxed

Writing n for $nodups$ and r, c and b for $rows$, $cols$ and $boxs$, recall that

$$legal\ g \quad = \quad all\ n\ (r\ g)\ \&\&\ all\ n\ (c\ g)\ \&\&\ all\ n\ (b\ g)$$

So

$$filter\ legal \quad = \quad filter\ (all\ n \cdot b) \cdot filter\ (all\ n \cdot c) \cdot filter\ (all\ n \cdot r)$$

and if d is one of r, c or b,

$$\begin{array}{ll}
 & filter\ (all\ n \cdot d) \cdot expand \\
= & \{\, map\ id = id \text{ and } d \cdot d = id \,\} \\
 & \underline{map\ (d \cdot d)} \cdot filter\ (all\ n \cdot d) \cdot expand \\
= & \{\, map\ f \cdot map\ g = map\ (f \cdot g) \,\} \\
 & map\ d \cdot \underline{map\ d \cdot filter\ (all\ n \cdot d)} \cdot expand \\
= & \{\, map\ f \cdot filter\ (p \cdot f) = filter\ p \cdot map\ f \,\} \\
 & map\ d \cdot filter\ (all\ n) \cdot \underline{map\ d \cdot expand} \\
= & \{\, map\ d \cdot expand = expand \cdot d \,\} \\
 & map\ d \cdot filter\ (all\ n) \cdot \underline{expand} \cdot d \\
= & \{\text{definition of } expand = cp \cdot map\ cp \,\} \\
 & map\ d \cdot \underline{filter\ (all\ n) \cdot cp} \cdot map\ cp \cdot d \\
= & \{\, filter\ (all\ p) \cdot cp = cp \cdot map\ (filter\ p) \,\} \\
 & map\ d \cdot cp \cdot \underline{map\ (filter\ n) \cdot map\ cp} \cdot d \\
= & \{\, map\ f \cdot map\ g = map\ (f \cdot g) \,\} \\
 & map\ d \cdot cp \cdot map\ (\underline{filter\ n \cdot cp}) \cdot d \\
= & \{\, filter\ n \cdot cp = filter\ n \cdot cp \cdot pruneRow \,\} \\
 & map\ d \cdot cp \cdot \underline{map\ (filter\ n \cdot cp \cdot pruneRow)} \cdot d \\
= & \{\, map\ f \cdot map\ g = map\ (f \cdot g) \,\} \\
 & map\ d \cdot cp \cdot map\ (filter\ n \cdot cp) \cdot map\ pruneRow \cdot d \\
= & \{\, d \cdot d = id \,\} \\
 & \underline{map\ d \cdot cp \cdot map\ (filter\ n \cdot cp) \cdot d} \cdot d \cdot map\ pruneRow \cdot d \\
= & \{\text{derivation above, reversed}\,\} \\
 & filter\ (all\ n \cdot d) \cdot expand \cdot d \cdot map\ pruneRow \cdot d
\end{array}$$

Because the three filters commute with each other we can use this calculation three times to deduce that

```
> prune :: Matrix Freedoms -> Matrix Freedoms
> prune = pruneBy boxs . pruneBy cols . pruneBy rows
>         where pruneBy f = f . map pruneRow . f
```

will do.

This suggests that we would be better to try

```
> prunesolve :: Solver
> prunesolve = filter legal . expand . prune . freedoms
```

Sadly this doesn't help much.

However, as in other walks of life, one *prune* has only a small effect. The same calculation shows that we can have two *prune*s, three *prune*s, ... as many *prune*s as we need. What if we have as many as are needed for one more to make no differnce?

```
> bowlsolve :: Solver
> bowlsolve =
>     filter legal . expand . repeatedly prune . freedoms
```

This turns out to be enough to solve easy problems.

The function *repeatedly* is not standard: it might have been defined by

```
> repeatedly :: Eq a => (a -> a) -> a -> a
> repeatedly f x | fx == x   = x
>                | otherwise = repeatedly f fx
>                  where fx = f x
```

but it could equally have been defined by

```
repeatedly f =
    fst . head . dropWhile (uncurry (/=)) . pairs . iterate f
  where pairs xs = xs `zip` tail xs
```

We can be sure that *repeatedly prune* cannot go on for ever looking for a fixed point. (Why?)

Unfortunately, however many times we prune, the more devious Sudoku problems will still have so much freedom left that this solver takes too long.

8.3 Parsimonious expansion

Another strategy that human solvers apply is to look for cells that are highly constrained, and consider the consequences of making each possible choice there.

Suppose we can define

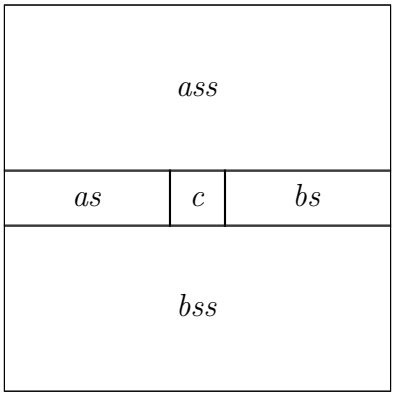
```
expand1 :: Matrix Freedoms -> [ Matrix Freedoms ]
```

to expand only one cell. This ought to satisfy

$$expand \quad \approx \quad concat \cdot map\ expand \cdot expand1$$

meaning that they are equal up permutation.

A good choice of cell on which to perform this expansion is any with a minimal number of choices, but not yet fixed. There might be no choices, in which case there is no expansion; or there might be at least two.




```
> expand1 :: Matrix Freedoms -> [ Matrix Freedoms ]
> expand1 fss = [ ass++[as++[[c]]++bs]++bss | c <- cell ]
>    where
>      (ass,row:bss) = break (any optimal) fss
>      (as, cell:bs) = break optimal row
>      optimal cell  = length cell == n
>      n             = minimum (lengths fss)
> lengths = filter (/=1) . map length . concat
```

where the predefined function $break$ satisfies

$$break\ p\ xs \quad = \quad (takeWhile\ (not \cdot p)\ xs, dropWhile\ (not \cdot p)\ xs)$$

and divides a list into two just before the first p-satisfying element of the list.

Some care is needed: if there are singletons in all of the cells of the matrix, this function will be undefined. So

$$expand \quad \approx \quad concat \cdot map\ expand \cdot expand1$$

can only hold if there is at least one non-singleton cell (possibly an empty one).

Say that a matrix of freedoms is *complete* if all cells are singletons, and *safe* if the singletons in the matrix are not contradictory. Unsafe matrices cannot lead to a solution. A complete and safe matrix corresponds to a unique legal grid.

```
> complete :: Matrix Freedoms -> Bool
> complete = all (all singleton)

> safe :: Matrix Freedoms -> Bool
> safe fss = all (nodups.ones) (rows fss) &&
>            all (nodups.ones) (cols fss) &&
>            all (nodups.ones) (boxs fss)
```

Assuming that it is applied to a matrix that is safe but incomplete,

$$
\begin{aligned}
& \mathit{filter}\ \mathit{legal} \cdot \mathit{expand} \\
\approx\ & \{\text{construction of } \mathit{expand1}\} \\
& \mathit{filter}\ \mathit{legal} \cdot \mathit{concat} \cdot \mathit{map}\ \mathit{expand} \cdot \mathit{expand1} \\
=\ & \{\mathit{filter}\ p \cdot \mathit{concat} = \mathit{concat} \cdot \mathit{map}\ (\mathit{filter}\ p)\} \\
& \mathit{concat} \cdot \mathit{map}\ (\mathit{filter}\ \mathit{legal}) \cdot \mathit{map}\ \mathit{expand} \cdot \mathit{expand1} \\
=\ & \{\mathit{map}\ f \cdot \mathit{map}\ g = \mathit{map}\ (f \cdot g)\} \\
& \mathit{concat} \cdot \mathit{map}\ (\mathit{filter}\ \mathit{legal} \cdot \mathit{expand}) \cdot \mathit{expand1} \\
=\ & \{\text{construction of } \mathit{prune}\} \\
& \mathit{concat} \cdot \mathit{map}\ (\mathit{filter}\ \mathit{legal} \cdot \mathit{expand} \cdot \mathit{prune}) \cdot \mathit{expand1} \\
=\ & \{\text{let } \mathit{search} = \mathit{filter}\ \mathit{legal} \cdot \mathit{expand} \cdot \mathit{prune}\} \\
& \mathit{concat} \cdot \mathit{map}\ \mathit{search} \cdot \mathit{expand1}
\end{aligned}
$$

This suggests we can write a solver

```
> solve :: Solver
> solve = search . freedoms
```

where *search* uses this derivation to solve an incomplete problem and in the case of a completed puzzle, either to return it or not according to whether or not it is safe (and so legal).

```
search fss | complete pss       = [ map (map head) pss | safe pss ]
           | not (complete pss) = (concat . map search . expand1) pss
             where pss = prune fss
```

Since, once the committed squares become unsafe, there is no way of filling in the other squares in a safe way, it would be more efficient to abandon the search as soon as the problem becomes unsafe

```
> search :: Matrix Freedoms -> [ Grid ]
> search fss | not (safe pss) = []
>            | complete pss   = [ map (map head) pss ]
>            | otherwise      = (concat . map search . expand1) pss
>              where pss = prune fss
```

and this solves problems much faster than I can.

Exercises

8.1 Aligning text in columns involves *justification*: perhaps to the right or left. One way of doing involves padding strings to a given length:

```
*Main> rjustify 10 "word"
"      word"
*Main> ljustify 10 "word"
"word      "
```

Define functions

```
> rjustify :: Int -> String -> String
> ljustify :: Int -> String -> String
```

to do this. What do your functions do if the string is wider than the target length? Is that what you would want, and if not how would you do it differently?

8.2 Suppose we represent an $n \times m$ matrix by a list of n rows, each of which is a list of m elements. These matrices will be elements of

```
> type Matrix a = [[a]]
```

that are, additionally to what the type says, rectangular and non-empty. Without writing any new recursive definitions:

1. define $scale :: Num\ a \Rightarrow a \rightarrow Matrix\ a \rightarrow Matrix\ a$ which multiplies each element of a matrix by a scalar (the qualification $Num\ a$ in the type means that $scale$ can use arithmetic on values of type a);
2. define a function $dot :: Num\ a \Rightarrow [a] \rightarrow [a] \rightarrow a$ which calculates the dot-product of two vectors of the same length;
3. define $add :: Num\ a \Rightarrow Matrix\ a \rightarrow Matrix\ a \rightarrow Matrix\ a$ which adds to matrices (of the same size as each other);
4. define $mul :: Num\ a \Rightarrow Matrix\ a \rightarrow Matrix\ a \rightarrow Matrix\ a$ which multiplies matrices of appropriately matching sizes;
5. define $table :: Show\ a \Rightarrow Matrix\ a \rightarrow String$ that translates a matrix (of printable elements) into a string that can be printed to show the matrix with each element right-justified in a column just wide enough to contain each of its elements. You may want to use the predefined $unwords$ and $unlines$.

```
*Main> putStr (table [[1,-500,-4], [100,15043,6], [5,3,10]])
  1  -500 -4
100 15043  6
  5     3 10
```

9 Proof by induction

The laws we use in equational reasoning about programs have to be justified. Because we are dealing with lists, which are a recursive data type, functions on lists are often defined by recursion, so proofs about them they will usually be justified by induction.

Think about

```
> pow :: Num a => a -> Int -> a
> pow x 0 = 1
> pow x n = pow x (n-1) * x
```

How would we prove

$$pow\ x\ (m+n) \quad = \quad pow\ x\ m \times pow\ x\ n$$

Being honest we would really want the n to be a natural number, and

```
> pow x 0         = 1
> pow x n | n > 0 = pow x (n-1) * x
```

is closer to it. In writing mathematics, however, one would be more likely to say

```
pow x  0    = 1
pow x (n+1) = pow x n * x
```

with the observation that since $n \in \mathbb{N}$, the $n+1$ cannot be zero. Older textbooks may have Haskell definitions that look like this, but the syntactic form was misleading and has been removed from the language.

9.1 Induction over natural numbers

A proof of $P(n)$ for every natural number n is an infinite number of proofs, one for $P(0)$, one for $P(1)$, one for $P(2)$, ... Clearly providing all of those will take a long time. A better alternative would be to provide a systematic way of generating any one of those proofs.

Every natural number is either 0, or is $n+1$ for some natural number n. To prove $P(n)$ for every natural number n it is enough to prove

1. $P(0)$
2. for every natural number n, if $P(n)$ then $P(n+1)$.

Since any natural number n can be made of a 0 and some number of applications of $(+1)$, the proof of $P(n)$ can be built out of a proof of $P(0)$ and some

(the same) number of proofs of $P(0) \Rightarrow P(1)$, $P(1) \Rightarrow P(2)$, $\ldots P(n-1) \Rightarrow P(n)$. So a proof of

$$pow\ x\ (m+n) \quad = \quad pow\ x\ m \times pow\ x\ n$$

by induction on n would consist to two parts: one for $n = 0$,

$$\begin{array}{ll} & pow\ x\ (m+0) \\ = & \{\text{unit of addition}\} \\ & pow\ x\ m \\ = & \{\text{unit of multiplication}\} \\ & pow\ x\ m \times 1 \\ = & \{\text{definition of } pow\} \\ & pow\ x\ m \times pow\ x\ 0 \end{array}$$

and assuming, temporarily, $pow\ x\ (m+n) = pow\ x\ m \times pow\ x\ n$ for some n,

$$\begin{array}{ll} & pow\ x\ (m+(n+1)) \\ = & \{\text{definition of } (+) \text{ (perhaps associativity of } (+))\} \\ & pow\ x\ ((m+n)+1) \\ = & \{\text{definition of } pow\} \\ & pow\ x\ (m+n) \times x \\ = & \{\text{induction hypothesis}\} \\ & (pow\ x\ m \times pow\ x\ n) \times x \\ = & \{\text{associativity of multiplication}\} \\ & pow\ x\ m \times (pow\ x\ n \times x) \\ = & \{\text{definition of } pow\} \\ & pow\ x\ m \times pow\ x\ (n+1) \end{array}$$

Then we can discharge the assumption and read this as a proof of

$$\begin{array}{l} \text{if } pow\ x\ (m+n) = pow\ x\ m \times pow\ x\ n \text{ then} \\ \qquad pow\ x\ (m+(n+1)) = pow\ x\ m \times pow\ x\ (n+1) \end{array}$$

for every natural number n.

9.2 The take lemma

Sometimes proofs about lists can be reduced to proofs about natural numbers. The *take lemma* is the observation that two lists xs and ys are equal provided that $take\ n\ xs = take\ n\ ys$ for all natural numbers n. This is a trivial observation for finite xs and ys, and a subtle one (to which we will return later) for infinite lists, which says that there is nothing in the infinite lists which does not appear in some finite prefix of the list.

That $take\ n\ xs = take\ n\ ys$ can often be proved by induction on n, noting that the case $n = 0$ always holds because both sides are null. For example, $map\ f\ (iterate\ f\ x) = iterate\ f\ (f\ x)$ because

$$
\begin{aligned}
& take\ (n+1)\ (map\ f\ (iterate\ f\ x)) \\
= \quad & \{\text{definition of } iterate\} \\
& take\ (n+1)\ (map\ f\ (x : iterate\ f\ (f\ x))) \\
= \quad & \{\text{definition of } map\} \\
& take\ (n+1)\ (f\ x : map\ f\ (iterate\ f\ (f\ x))) \\
= \quad & \{\text{definition of } take\} \\
& f\ x : take\ n\ (map\ f\ (iterate\ f\ (f\ x))) \\
= \quad & \{\text{induction hypothesis, with } f\ x \text{ substituted for } x\} \\
& f\ x : take\ n\ (iterate\ f\ (f\ (f\ x))) \\
= \quad & \{\text{definition of } take\} \\
& take\ (n+1)\ (f\ x : iterate\ f\ (f\ (f\ x))) \\
= \quad & \{\text{definition of } iterate\} \\
& take\ (n+1)\ (iterate\ f\ (f\ x)))
\end{aligned}
$$

Notice that a proof about $n + 1$ and x requires a hypothesis about n and $f\ x$. Formally, the induction hypothesis is that *for all* x the result holds for n.

It should be clear that a proof by induction on n, like the one above, shows only that corresponding finite prefixes of xs and ys are equal. The force of the take lemma is that this means that xs and ys must be equal even if infinite.

An equivalent presentation of the take lemma uses the function

```
> take' n    []  | n > 0 = []
> take' n (x:xs) | n > 0 = x : take' (n-1) xs
```

which prunes its list argument to no more than n constructors, but leaves $\perp$ after that. It can be generalised to similar proof schemes for other data types, for example

```
> data Stream a = Cons a (Stream a)
```

the type of infinite streams which have no end, or types where there are a several possible last constructors.

9.3 Induction over finite lists

Every finite list is either $[\,]$, or is $x : xs$ for some finite list xs. To prove $P(xs)$ for every finite list xs it is enough to prove

1. $P([\,])$
2. for every finite list xs, if $P(xs)$ then $P(x : xs)$.

Recall that

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

We prove that $(xs +\!\!+ ys) +\!\!+ zs = xs +\!\!+ (ys +\!\!+ zs)$ for all finite lists xs by induction on xs. Firstly for empty lists

$$
\begin{aligned}
& ([\,] +\!\!+ ys) +\!\!+ zs \\
= \quad & \{\text{definition of } (+\!\!+)\} \\
& ys +\!\!+ zs \\
= \quad & \{\text{definition of } (+\!\!+)\} \\
& [\,] +\!\!+ (ys +\!\!+ zs)
\end{aligned}
$$

then assuming $(xs +\!\!+ ys) +\!\!+ zs = xs +\!\!+ (ys +\!\!+ zs)$

$$
\begin{aligned}
& ((x : xs) +\!\!+ ys) +\!\!+ zs \\
= \quad & \{\text{definition of } (+\!\!+)\} \\
& (x : (xs +\!\!+ ys)) +\!\!+ zs \\
= \quad & \{\text{definition of } (+\!\!+)\} \\
& x : ((xs +\!\!+ ys) +\!\!+ zs) \\
= \quad & \{\text{induction hypothesis}\} \\
& x : (xs +\!\!+ (ys +\!\!+ zs)) \\
= \quad & \{\text{definition of } (+\!\!+)\} \\
& (x : xs) +\!\!+ (ys +\!\!+ zs)
\end{aligned}
$$

so the result is true for all finite lists xs (and all lists ys and zs).

9.4 A second example

Given the definition

```
> reverse    []  = []
> reverse (x:xs) = reverse xs ++ [x]
```

a proof of $reverse\ (reverse\ xs) = xs$ for all finite lists xs proceeds by induction on xs.

$$
\begin{aligned}
& reverse\ (reverse\ [\,]) \\
= \quad & \{\text{definition of } reverse\} \\
& reverse\ [\,] \\
= \quad & \{\text{definition of } reverse\} \\
& [\,]
\end{aligned}
$$

and then assuming $reverse\ (reverse\ xs) = xs$ for some xs

$$
\begin{array}{ll}
 & reverse\ (reverse\ (x : xs)) \\
= & \{\text{definition of } reverse\} \\
 & reverse\ (reverse\ xs +\!\!+ [x]) \\
= & \{\text{unjustified step}\} \\
 & x : reverse\ (reverse\ xs) \\
= & \{\text{inductive hypothesis}\} \\
 & x : xs
\end{array}
$$

The (as yet) unjustified step requires a lemma:

$$reverse\ (ys +\!\!+ [x]) = x : reverse\ ys$$

which can be proved by induction on ys.

$$
\begin{array}{ll}
 & reverse\ ([\,] +\!\!+ [x]) \\
= & \{\text{definition of } (+\!\!+)\} \\
 & reverse\ [x] \\
= & \{\text{definition of } reverse\} \\
 & reverse\ [\,] +\!\!+ [x] \\
= & \{\text{definition of } reverse\} \\
 & [\,] +\!\!+ [x] \\
= & \{\text{definition of } (+\!\!+)\} \\
 & [x] \\
= & \{\text{definition of } reverse\} \\
 & x : reverse\ [\,]
\end{array}
$$

and assuming it to be true for ys

$$
\begin{array}{ll}
 & reverse\ ((y : ys) +\!\!+ [x]) \\
= & \{\text{definition of } (+\!\!+)\} \\
 & reverse\ (y : (ys +\!\!+ [x])) \\
= & \{\text{definition of } reverse\} \\
 & reverse\ (ys +\!\!+ [x]) +\!\!+ [y] \\
= & \{\text{inductive hypothesis}\} \\
 & (x : reverse\ ys) +\!\!+ [y] \\
= & \{\text{definition of } (+\!\!+)\} \\
 & x : (reverse\ ys +\!\!+ [y]) \\
= & \{\text{definition of } reverse\} \\
 & x : reverse\ (y : ys)
\end{array}
$$

9.5 Induction over partial lists

A partial list is one with a tail that is either $\bot$ or a smaller partial list. There is a similar principle of induction over partial lists; to prove $P(xs)$ for every partial list xs it is enough to prove

1. $P(\bot)$
2. for every partial list xs, if $P(xs)$ then $P(x : xs)$.

For example $xs +\!\!+ ys = xs$ for all partial lists xs.

The case $\bot +\!\!+ ys = \bot$ is immediate from the strictness of $(+\!\!+)$, since it is defined by pattern matching on its left argument. Then assuming the result for some partial xs

$$
\begin{aligned}
& (x : xs) +\!\!+ ys \\
= \quad & \{\text{definition of } (+\!\!+)\} \\
& x : (xs +\!\!+ ys) \\
= \quad & \{\text{inductive hypothesis}\} \\
& x : xs
\end{aligned}
$$

9.6 Infinite lists

An infinite list can be thought of as a limit of a sequence of partial lists, for example $[0..]$ is the limit of the chain $\bot$, $0 : \bot$, $0 : 1 : \bot$, $0 : 1 : 2 : \bot$, ... and so on. Successive elements of this chain are related by the information ordering, $(\sqsubseteq)$. This is the partial order that has $\bot \sqsubseteq x$ for all x, and $(x : xs) \sqsubseteq (y : ys)$ whenever $x \sqsubseteq y$ and $xs \sqsubseteq ys$.

A sequence of finite proper lists, such as $[\,]$, $[0]$, $[0, 1]$, $[0, 1, 2]$, ... and so on, will not do to approximate $[0..]$ because each of them contains some wrong information: that there is a $[\,]$ at some point in the list. Not only are these finite lists not related to each other by $\sqsubseteq$, they are not even related to the infinite list.

A property P is called chain complete if whenever $P(xs_i)$ holds for every element xs_i of a chain it also holds for the limit of that chain. So, chain complete properties that also hold for all partial lists will be ones that also hold for infinite lists.

It is enough to know that positive properties are chain complete. These include universally quantified (mathematical) equations between Haskell-definable expressions, and conjunctions ('and's) of positive properties. Inequalities need not be chain complete, nor need properties involving existential quantification be. For example "$drop\ n\ xs = \bot$ for some n" is true for all partial lists, but is not true for infinite lists. It is not one equation, but the disjunction ('or') of an infinite number of equations.

The earlier proof of $(xs +\!\!+ ys) +\!\!+ zs = xs +\!\!+ (ys +\!\!+ zs)$ for finite lists can be extended to partial lists xs, because $(\bot +\!\!+ ys) +\!\!+ zs = \bot +\!\!+ zs = \bot$ (by strictness of $(+\!\!+)$) so the result holds for all finite and partial lists, and since it is an

equation between Haskell expressions it is chain complete and also holds for infinite lists. (Informally, both sides are equal to xs if xs is an infinite list.)

Can we extend the proof of $reverse\ (reverse\ xs) = xs$ to infinite lists? Certainly it holds for bottom: $reverse\ (reverse\ \bot) = reverse\ \bot = \bot$ (again, by strictness of $reverse$). This looks promising; but running `reverse (reverse [0..])` will produce no output: $reverse\ (reverse\ xs) = \bot$ for all partial xs.

What went wrong? The proof that

$$reverse\ (reverse\ xs) = xs \Rightarrow reverse\ (reverse\ (x : xs)) = x : xs$$

required a lemma $reverse\ (ys +\!\!+ [x]) = x : reverse\ ys$ which we only proved for finite lists ys. It is not true for infinite lists, nor indeed for partial lists, because $reverse\ (\bot +\!\!+ [x]) = \bot$ and so $reverse\ (ys +\!\!+ [x]) = \bot$ for all partial lists ys.

9.7 Summary: induction schemes

A proof by (structural) induction on xs that a property $P(xs)$ holds for all of some class of lists xs involves assembling proof components for all of the ways of constructing list in that class.

To prove that $P(xs)$ for all finite xs, it is enough to prove (i) $P([\,])$ and (ii) if $P(xs)$ for some finite xs then $P(x : xs)$.

To prove that $P(xs)$ for all partial xs, it is enough to prove (i) $P(\bot)$ and (ii) if $P(xs)$ for some partial xs then $P(x : xs)$.

To prove that $P(xs)$ for all infinite xs it is enough to prove (i) $P(\bot)$ and (ii) if $P(xs)$ for some partial xs then $P(x : xs)$, and (iii) P is chain complete (for example, because it is an equation between Haskell-definable expressions).

In the special case of an equation $E = F$ it is also possible to use the take lemma which requires that you can prove only that if $take\ n\ E = take\ n\ F$ then $take\ (n+1)\ E = take\ (n+1)\ F$.

9.8 Aside: Continuity (beyond the scope of this course)

This use of the words *limit* and *continuity* turn out to be exactly the same as you might meet in analysis.

You can define a distance function $d :: ([a], [a]) \to \mathbb{R}_{\geqslant 0}$, a *metric*, which behaves like the distance between points in space. In particular, the triangle inequality $d(xs, ys) + d(ys, zs) \geqslant d(xs, zs)$ holds.

A suitable distance between distinct lists xs and ys might be $\frac{1}{n}$ for the biggest n for which the $take'\ n\ xs = take'\ n\ ys$ (or the equivalent equation for the standard $take$ function). The successive elements of the sequence of partial approximations to an infinite list are arbitrarily close together, just like Cauchy sequences in $\mathbb{R}$. The infinite list is the limit of that Cauchy sequence.

Continuity for real-valued functions turns out to be equivalent to preserving limits. A function has an abrupt discontinuity exactly when you can find a convergent sequence leading up to a point of discontinuity, so that the image

of the limit under the function is not the limit of the images of the elements of the sequence.

Computable functions have to be not only monotonic (put in more information into an argument, they cannot retract any information that has already come out in the result). More than that they need to be continuous (apply a continuous function to a limit of a chain, and you get back the limit of the images of the sequence).

Monotonicity tells you that none of the information in the image of of the limit of a sequence can contradict anything you might have learned from the image of an element of that sequence. Continuity tells you that all of the information in the image of the limit of the sequence will already have come out in the image of one of the elements of the sequence.

That is, if f is a computable function, and xs is an infinte list, then it is OK to compute $f\ xs$ by successively working out what you can know about $f\ xs$ by calculating $f\ (take'\ n\ xs)$ for successively bigger n.

Chain completeness also turns out to be continuity, and continuity guarantees soundness. An assertion can be thought of as a function from its free variables to the truth of the assertion. A proof of a comntinuous assertion which is valid for all (except perhaps a finite number) of the elements of a chain cannot then make an abrupt change from valid to invalid at the infinite limit of the chain. Mathematical equality is continuous, so a mathematical equality of two Haskell expressions is the composition of continuous functions.

Exercises

9.1 Suppose a type of natural numbers is defined by

```
> data Nat = Zero | Succ Nat
```

Use recursion to define functions $int :: Nat \to Int$ and $nat :: Int \to Nat$ which embed the natural numbers in Int in the obvious way.

Use recursion (on the second argument) to define functions

$$add, mul, pow, tet :: Nat \to Nat \to Nat$$

which implement addition, multiplication, exponentiation, and what Goodstein calls tetration.

($x\ `tet`\ n = x \uparrow x \uparrow \cdots \uparrow x$ where there are n copies of x.)

9.2 What property characterises $foldNat$, the fold for Nat? Define $foldNat$.

What are the deconstructors for Nat, and what characterises the unfold $unfoldNat$? Define $unfoldNat$.

Express each of int and nat as either $foldNat$ or $unfoldNat$.

Finally express add, mul, pow and tet as folds.

9.3 By definition, a *metric* on a set S is a function $d : S \times S \to \mathbb{R}$ which for all $x, y, z \in S$ satisfies

1. $d(x, y) \geqslant 0$
2. $d(x, y) = 0 \Leftrightarrow x = y$
3. $d(x, y) = d(y, x)$
4. $d(x, z) \leqslant d(x, y) + d(y, z)$

Show that d is a metric on lists (all of the same type $[\alpha]$ for some α) if $d(xs, ys) = 1/(k + 1)$ where k is the smallest index for which $take\ k\ xs \neq take\ k\ ys$, if there is one; and $d(xs, ys) = 0$ if there is no such k.

9.4 A *Cauchy sequence* is, by definition, an infinite sequence whose elements get arbitrarily close together, and is expected therefore to have a limit. Specifically a sequence x is Cauchy if

$$\forall \varepsilon > 0.\ \exists n.\ \forall m \geqslant n.\ d(x_n, x_m) < \varepsilon.$$

Informally, which sequences of lists are Cauchy sequences with respect to the list metric in exercise 9.3? Describe the limit of such a sequence.

9.5 Prove the claim on page 41 that

$$unfold\ p\ h\ t \cdot fold\ c\ n \quad = \quad id$$

provided that

$$\begin{aligned} p\ n &= True \\ p\ (c\ x\ y) &= False \\ h\ (c\ x\ y) &= x \\ t\ (c\ x\ y) &= y \end{aligned}$$

9.6 The standard function

```
(\\) :: Eq a => [a] -> [a] -> [a]
```

(in the library $Data.List$) calculates the list difference of its arguments:

```
> xs \\    []  = xs
> xs \\ (y:ys) = delete y xs \\ ys
> delete y []                 = []
> delete y (x:xs) | x == y    = xs
>                 | otherwise = x : delete y xs
```

Prove that $(xs +\!\!+ ys) \setminus\!\setminus xs = ys$ for all finite xs.

Is it true for infinite xs? Why, or why not?

9.7 A complete partial order is one in which every ascending chain has a limit. Given a complete partial order $\sqsubseteq$ and an ascending chain x_n, that is a sequence for which $x_n \sqsubseteq x_{n+1}$ for all n, the limit of the sequence is characterised by the two properties: $x_n \sqsubseteq limx_n$, and if $x_n \sqsubseteq x$ for all n then $limx_n \sqsubseteq x$.

Given that the information order $\sqsubseteq$ on Haskell values is complete, show that $xs = lim\ (take'\ n\ xs)$ for all lists xs.

9.8 Prove the $take'$ form of the take lemma for lists, that if $take'\ n\ xs = take'\ n\ ys$ for all n, then $xs = ys$.

9.9 Use the result in exercise 9.8 to prove the take lemma as presented in section 9.2, that if $take\ n\ xs = take\ n\ ys$ for all n, then $xs = ys$.

10 Right and left folds

The fold on lists, an instance of $foldr$, is

```
> fold :: (a -> b -> b) -> b -> [a] -> b
> fold cons nil    []  = nil
> fold cons nil (x:xs) = cons x (fold cons nil xs)
```

and $fold\ (:)\ [\,] = id$. It captures a possible pattern of computation for many functions on lists

$$\begin{aligned} sum &= fold\ (+)\ 0 \\ product &= fold\ (\times)\ 1 \\ concat &= fold\ (+\!\!+)\ [\,] \\ map\ f &= fold\ ((:) \cdot f)\ [\,] \end{aligned}$$

notice that none of these equations is recursive: only equations defining $fold$ are recursive. We might hope to be able to prove things about the others, such as

$$\begin{aligned} sum\ (xs +\!\!+ ys) &= sum\ xs + sum\ ys \\ product\ (xs +\!\!+ ys) &= product\ xs \times product\ ys \\ concat\ (xs +\!\!+ ys) &= concat\ xs +\!\!+ concat\ ys \\ map\ f\ (xs +\!\!+ ys) &= map\ f\ xs +\!\!+ map\ f\ ys \end{aligned}$$

without resorting to induction for every one of them.

What is needed is a proof that

$$fold\ c\ n\ (xs +\!\!+ ys) = fold\ c\ n\ xs \oplus fold\ c\ n\ ys$$

Setting out to prove this, just once, will reveal what relationship has to exist between c, n and $(\oplus)$.

$$\begin{aligned} & fold\ c\ n\ ([\,] +\!\!+ ys) \\ = & \{\text{definition of } (+\!\!+)\} \\ & fold\ c\ n\ ys \end{aligned} \qquad \begin{aligned} & fold\ c\ n\ [\,] \oplus fold\ c\ n\ ys \\ = & \{\text{definition of } fold\} \\ & n \oplus fold\ c\ n\ ys \end{aligned}$$

so these will be equal if $x = n \oplus x$ for all x.

$$\begin{aligned} & fold\ c\ n\ ((x : xs) +\!\!+ ys) \\ = & \{\text{definition of } (+\!\!+)\} \\ & fold\ c\ n\ (x : (xs +\!\!+ ys)) \\ = & \{\text{definition of } fold\} \\ & c\ x\ (fold\ c\ n\ (xs +\!\!+ ys)) \\ = & \{\text{inductive hypothesis}\} \\ & c\ x\ (fold\ c\ n\ xs \oplus fold\ c\ n\ ys) \end{aligned} \qquad \begin{aligned} & fold\ c\ n\ (x : xs) \oplus fold\ c\ n\ ys \\ = & \{\text{definition of } fold\} \\ & c\ x\ (fold\ c\ n\ xs) \oplus fold\ c\ n\ ys \end{aligned}$$

and these will be equal if $x \text{ `}c\text{` } (y \oplus z) = (x \text{ `}c\text{` } y) \oplus z$. Furthermore,

$$\begin{array}{ll} & \mathit{fold}\ c\ n\ (\bot +\!\!+\ ys) \\ = & \{\text{definition of } (+\!\!+)\} \\ & \mathit{fold}\ c\ n\ \bot \\ = & \{\mathit{fold} \text{ is strict in the list}\} \\ & \bot \end{array} \qquad \begin{array}{ll} & \mathit{fold}\ c\ n\ \bot \oplus \mathit{fold}\ c\ n\ ys \\ = & \{\mathit{fold} \text{ is strict in the list}\} \\ & \bot \oplus \mathit{fold}\ c\ n\ ys \end{array}$$

and these will be equal if $(\oplus)$ is strict, that is if the operator is strict in its left argument.

None of these three properties involves any recursion so they can be checked by induction-free proofs.

10.1 Fusion

The most generally useful property of folds is that, given the right properties of f, g, h, a, and b,

$$f \cdot \mathit{fold}\ g\ a \quad = \quad \mathit{fold}\ h\ b$$

These are functions of a list so the proof of equality is by induction on an argument list

$$\begin{array}{ll} & (f \cdot \mathit{fold}\ g\ a)\ \bot \\ = & \{\text{definition of } (\cdot)\} \\ & f\ (\mathit{fold}\ g\ a\ \bot) \\ = & \{\mathit{fold} \text{ is strict in the list}\} \\ & f\ \bot \end{array} \qquad \begin{array}{ll} & \mathit{fold}\ h\ b\ \bot \\ = & \{\mathit{fold} \text{ is strict in the list}\} \\ & \bot \end{array}$$

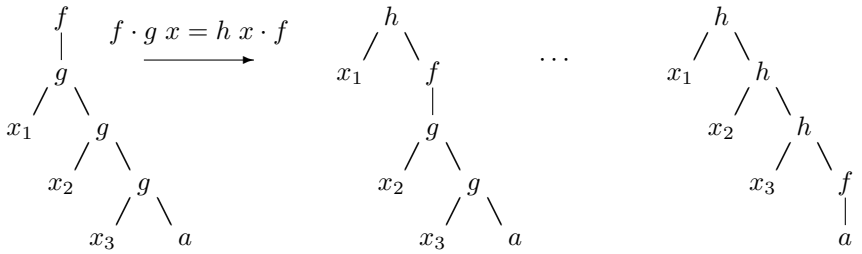
so f must be strict.

$$\begin{array}{ll} & (f \cdot \mathit{fold}\ g\ a)\ [\,] \\ = & \{\text{definition of } (\cdot)\} \\ & f\ (\mathit{fold}\ g\ a\ [\,]) \\ = & \{\text{definition of } \mathit{fold}\} \\ & f\ a \end{array} \qquad \begin{array}{ll} & \mathit{fold}\ h\ b\ [\,] \\ = & \{\text{definition of } \mathit{fold}\} \\ & b \end{array}$$

so $b = f\ a$.

$$\begin{array}{ll} & (f \cdot \mathit{fold}\ g\ a)\ (x : xs) \\ = & \{\text{definition of } (\cdot)\} \\ & f\ (\mathit{fold}\ g\ a\ (x : xs) \\ = & \{\text{definition of } \mathit{fold}\} \\ & f\ (g\ x\ (\mathit{fold}\ g\ a\ xs)) \\ = & \{\text{definition of } (\cdot)\} \\ & (f \cdot g\ x)(\mathit{fold}\ g\ a\ xs) \end{array} \qquad \begin{array}{ll} & \mathit{fold}\ h\ b\ (x : xs) \\ = & \{\text{definition of } \mathit{fold}\} \\ & h\ x\ (\mathit{fold}\ h\ b\ xs) \\ = & \{\text{inductive hypothesis}\} \\ & h\ x\ ((f \cdot \mathit{fold}\ g\ a)\ xs) \\ = & \{\text{definition of } (\cdot), \text{ twice}\} \\ & (h\ x \cdot f)\ (\mathit{fold}\ g\ a\ xs) \end{array}$$

and these will be equal at least if $h\ x \cdot f = f \cdot g\ x$ or equivalently if $h\ x\ (f\ y) = f\ (g\ x\ y)$.



Most of the laws that we have used that are about functions that are folds have been instances of fusion. We have also been relying on a special case of fusion to show that some function f on lists is a fold, because

$$
\begin{array}{ll}
 & f \\
= & \{\,\text{unit of composition}\,\} \\
 & f \cdot id \\
= & \{\,\text{fold of constructors}\,\} \\
 & f \cdot fold\ (:)\ [\,] \\
= & \{\,\text{fusion}\,\} \\
 & fold\ h\ (f\ [\,])
\end{array}
$$

provided f is strict, and $f\ (x : xs) = h\ x\ (f\ xs)$.

10.2 Left and right folds

One intuition about *fold* is that it produces a right-heavy expression where the arguments replace the constructors of a list:

$$fold\ (\oplus)\ e\ [x_0, x_1, x_2, \ldots x_n] \quad = \quad (x_0 \oplus (x_1 \oplus (x_2 \oplus \cdots (x_n \oplus e) \cdots)))$$

There is a predefined function *foldr* which when restricted to lists agrees with *fold*. We might compute a similar left-heavy expression

$$loop\ (\oplus)\ e\ [x_0, x_1, x_2, \ldots x_n] \quad = \quad (\cdots (((e \oplus x_0) \oplus x_1) \oplus x_2) \oplus \cdots x_n)$$

and might specify this by

$$loop\ s\ n \quad = \quad fold\ (flip\ s)\ n \cdot reverse$$

and calculate from this that it is strict; that

$$
\begin{aligned}
& loop\ s\ n\ [\,] \\
= \quad & \{\text{specification}\} \\
& (fold\ (flip\ s)\ n \cdot reverse)\ [\,] \\
= \quad & \{\text{composition}\} \\
& fold\ (flip\ s)\ n\ (reverse\ [\,]) \\
= \quad & \{\text{definition of } reverse\} \\
& fold\ (flip\ s)\ n\ [\,] \\
= \quad & \{\text{definition of } fold\} \\
& n
\end{aligned}
$$

and that

$$
\begin{aligned}
& loop\ s\ n\ (x : xs) \\
= \quad & \{\text{specification}\} \\
& (fold\ (flip\ s)\ n \cdot reverse)\ (x : xs) \\
= \quad & \{\text{composition}\} \\
& fold\ (flip\ s)\ n\ (reverse\ (x : xs)) \\
= \quad & \{\text{definition of } reverse\} \\
& fold\ (flip\ s)\ n\ (reverse\ xs +\!\!+ [x]) \\
= \quad & \{\text{lemma (exercise 10.1 or 10.2)}\} \\
& fold\ (flip\ s)\ (fold\ (flip\ s)\ n\ [x])\ (reverse\ xs) \\
= \quad & \{\text{definition of } fold\text{, twice}\} \\
& fold\ (flip\ s)\ (flip\ s\ x\ n)\ (reverse\ xs) \\
= \quad & \{\text{definition of } flip\} \\
& fold\ (flip\ s)\ (s\ n\ x)\ (reverse\ xs) \\
= \quad & \{\text{composition}\} \\
& (fold\ (flip\ s)\ (s\ n\ x) \cdot reverse)\ xs \\
= \quad & \{\text{specification}\} \\
& loop\ s\ (s\ n\ x)\ xs
\end{aligned}
$$

This justifies defining

```
> loop s n    []  = n
> loop s n (x:xs) = loop s (s n x) xs
```

and this is essentially the same as the predefined *foldl* (restricted to lists).

10.3 Scans

One commonly needs to think about 'partial sums'. The natural thing to think of first for lists is

$$scan\ c\ n \quad = \quad map\ (fold\ c\ n) \cdot tails$$

where the tails of a list are all the suffix segments, in decreasing order of length.

```
> tails :: [a] -> [[a]]
> tails    []  = [[]]
> tails (x:xs) = (x:xs): tails xs
```

There is a very similar standard function $tails$ in `Data.List`.

In the way you probably expect, $tails$ can be cast as a $fold$ because

$$
\begin{array}{ll}
 & tails\ (x : xs) \\
= & \{\,\text{definition of } tails\,\} \\
 & (x : xs) : tails\ xs \\
= & \{\,head\ (tails\ xs) = xs\,\} \\
 & (x : head\ ys) : ys\ \mathbf{where}\ ys = tails\ xs
\end{array}
$$

so $tails = fold\ g\ [[\,]]\ \mathbf{where}\ g\ x\ ys = (x : head\ ys) : ys$. This means that if the conditions of fusion are satisfied, $scan$ can be expressed as a $fold$.

Firstly, $map\ (fold\ c\ n)$ is strict; then

$$
\begin{array}{ll}
 & map\ (fold\ c\ n)\ [[\,]] \\
= & \{\,\text{definition of } map\,\} \\
 & [fold\ c\ n\ [\,]] \\
= & \{\,\text{definition of } fold\,\} \\
 & [n]
\end{array}
$$

and then

$$
\begin{array}{ll}
 & map\ (fold\ c\ n)\ (g\ x\ ys) \\
= & \{\,\text{definition of } g\,\} \\
 & map\ (fold\ c\ n)\ ((x : head\ ys) : ys) \\
= & \{\,\text{definition of } map\,\} \\
 & (fold\ c\ n\ (x : head\ ys) : map\ (fold\ c\ n)\ ys \\
= & \{\,\text{definition of } fold\,\} \\
 & c\ x\ (fold\ c\ n\ (head\ ys)) : map\ (fold\ c\ n)\ ys \\
= & \{\,f \cdot head = head \cdot map\ f\,\} \\
 & c\ x\ (head\ zs) : zs\ \mathbf{where}\ zs = map\ (fold\ c\ n)\ ys
\end{array}
$$

from which conclude that

$$
\begin{array}{ll}
 & scan\ c\ n \\
= & \{\,\text{specification}\,\} \\
 & map\ (fold\ c\ n) \cdot tails \\
= & \{\,\text{fusion}\,\} \\
 & fold\ h\ [n]\ \mathbf{where}\ h\ x\ zs = c\ x\ (head\ zs) : zs
\end{array}
$$

Notice that executing the specification directly gives a quadratic algorithm: for a list xs of length n there are about $\frac{1}{2}n^2$ applications of c. However there are only n applications of h, each of which calls c exactly once (and does a constant amount of consing and unconsing). The result is a linear algorithm for

```
> scan c n = fold h [n] where h x zs = c x (head zs):zs
```

The predefined function $scanr$ is equal to $scan$, and even has the same strictness.

10.4 Aside: the names of *fold* and *loop*

In text books you will find $fold$ being called $foldr$. It would have been natural for the arguments to be in the same order as the constructors appear in the type of lists, but the name and the argument order of $foldr$ have been the same at least since David Turner's *SASL* in the mid 1970s. The first argument has also been called f and the second either r or e, and you will find $foldr\ f\ e$ in textboooks. My use of c and n is for menmonic reasons: c replaces the conses in a list, and n replaces the nil.

Similarly, my $loop\ s\ n$ appears as $foldl\ f\ e$ in texts. I used to call it $tailfold$ because it is a tail call, but justifying calling it a fold is initially harder. It is (exercise 10.4) the fold on lists built with $snoc$ constructors that add a last element to a list, hence my use of s for its argument.

10.5 Aside: strictness

The function $tails$ defined above is strict, but $Data.List.tails$ is not. However the implementation of $scan$ as a fold is strict (as is the predefined $scanr$ which is equal to $scan$), because folds are strict.

Had we defined $tails$ to be non-strict,

```
> tails xs = xs : if null xs then [] else tails (tail xs)
```

it would not have been possible to implement it by a fold. The rest of the derivation of the implementation of $scan$ as a fold is sound.

You might argue that the efficient implementation of $scan$ is not a faithful implementation of $map\ (fold\ c\ n) \cdot tails$ if $tails$ is not strict.

10.6 Aside: fusion and computability

The statement of the fold fusion law $f \cdot fold\ g\ a = fold\ h\ b$ was proved by showing that $f\ (fold\ g\ a\ xs) = fold\ h\ b\ xs$ by induction on xs, but no mention was made in section 10.1 of chain completeness. In fact the three conditions for fold fusion are only enough to prove this equality for finite or partial xs.

To complete the proof by induction for infinite xs it is necessary to show that the proposition $f\ (fold\ g\ a\ xs) = fold\ h\ b\ xs$ is chain complete with respect to xs.

Fortunately if both sides are computable functions of xs, as they would be if they were Haskell-definable functions, an equation is chain complete.

However an equation proved to be true on all partial lists does not necessarily hold at infinite lists if the two sides are non-computable functions of the list, and fold fusion cannot necessarily be applied to such functions. (See exercise 10.6.)

Exercises

10.1 Prove directly by induction that

$$fold\ c\ n\ (xs +\!\!+ ys) \quad = \quad fold\ c\ (fold\ c\ n\ ys)\ xs$$

for all lists xs and ys (whether partial, finite or infinite).

10.2 Use fold fusion to show that the section $(+\!\!+ bs)$ is a fold.

Deduce without resort to induction that

$$fold\ c\ n\ (xs +\!\!+ ys) \quad = \quad fold\ c\ (fold\ c\ n\ ys)\ xs$$

10.3 Use fold fusion to show that $filter\ p$ is a fold.

Deduce that

$$filter\ p\ (xs +\!\!+ ys) \quad = \quad filter\ p\ xs +\!\!+ filter\ p\ ys$$

10.4 A data type very like that of lists might be defined by

```
> data Liste a = Snoc (Liste a) a | Lin
```

There will be elements of $Liste\ \alpha$ and of $[\alpha]$ corresponding to finite lists, for example $((Lin\ `Snoc`\ 1)\ `Snoc`\ 2)\ `Snoc`\ 3$ corresponds to $1 : (2 : (3 : [\,]))$, that is $[1, 2, 3]$.

Write a recursive definition of a function $cat :: Liste\ \alpha \to Liste\ \alpha \to Liste\ \alpha$ which concatenates two elements of $Liste$.

Define a function $folde$ which is the natural fold for $Liste\ \alpha$.

Express cat in terms of $folde$.

Define (as folds) functions $list :: Liste\ \alpha \to [\alpha]$ and $liste :: [\alpha] \to Liste\ \alpha$ which express the identification of finite lists represented as elements of $Liste\ \alpha$ and of $[\alpha]$. (That is, they should be mutually inverse on finite lists.)

What does $liste$ return when applied to an infinite list? What are the infinite objects of type $Liste\ \alpha$?

Find equivalent definitions of $list$ and $liste$ as instances of $loop$ and the corresponding function for $Liste\ \alpha$.

10.5 Recall that the unfold function for $[\alpha]$

```
> unfold n h t x | n x       = []
>                | otherwise = h x : unfold n h t (t x)
```

yields the identity function $unfold\ null\ head\ tail$ when applied to the deconstructors for $[\alpha]$.

Using the corresponding property for the deconstructors and the identity of $Liste\ \alpha$ define the unfold function $unfolde$ for $Liste\ \alpha$.

Write $list$ and $liste$ as the appropriate unfolds.

You may want to know that there are predefined functions $init :: [\alpha] \to [\alpha]$ and $last :: [\alpha] \to \alpha$ for which $xs = init\ xs +\!\!+ [last\ xs]$ for all non-null xs. It might help to work out first how these might be defined by recursion.

10.6 A function $f :: [a] \to [a]$ satisfies $f\ xs = xs$ for all infinite lists xs, and $f\ xs = \bot$ otherwise. Show that f is not computable.

The function $(!:)$, read as *tail-strict cons*, is defined so that $x\,!:\,\bot = \bot$, and $x\,!:\,xs = x : xs$ otherwise.

Show that whilst f satisfies the three conditions for the corresponding fold fusion,

$$f \cdot fold\ (:)\ [\,] \quad \neq \quad fold\ (!:)\ \bot$$

11 Types and trees

Recall (from lecture 3) that a data type declaration like

```
> data Either a b = Left a | Right b
```

introduces three named things:

- a type-value function of types, $Either$ that constructs a new type for any two types a and b;
- an injection $Left :: a \to Either\ a\ b$, and
- an injection $Right :: b \to Either\ a\ b$, in such a way that every proper value of $Either\ a\ b$ is either in the image of $Left$ or the image of $Right$, but never both.

The functions $Left$ and $Right$ are *constructors* which are necessarily invertible. The data declaration also allows constructors to be used in pattern matching, so we could define functions

```
> left  (Left x)  = x
> right (Right y) = y
```

as the left inverses of the constructors, known as *selectors*. Haskell also provides a convenient abbreviation for the selector declarations:

```
> data Either a b = Left {left :: a} | Right {right :: b}
```

Care: the inverses are partial functions, and the claim is that the constructors have inverses on the left.

If pattern matching were not allowed we would still need in addition a *discriminator* such as

```
> isLeft (Left  _) = True
> isLeft (Right _) = False
```

to make a complete set of deconstructors. Then anything that can be written in Haskell using pattern matching can be written in terms of $isLeft$, $left$ and $right$, without resort to pattern matching.

These are not the only set of functions that would do as deconstructors for the $Either$ type. The single general function

```
> either :: (a -> c) -> (b -> c) -> Either a b -> c
> either left right (Left x)  = left x
> either left right (Right y) = right y
```

will do, because

$$\begin{aligned} left &= either\ id\ undefined \\ right &= either\ undefined\ id \\ isLeft &= either\ (const\ True)\ (const\ False) \end{aligned}$$

More generally any contructor can have any number of arguments, for example

```
> data Pair a = Pair { first :: a,  second :: a }
```

The type $Pair\ a$ is not quite the same as the type (α, β) of pairs (x, y) of elements $x :: \alpha$ and $y :: \beta$ because a $Pair\ \alpha$ has elements both of which are α. The pun between the name of the type function $Pair$ and its unique constructor $Pair :: \alpha \to \alpha \to Pair\ \alpha$ is common, but some people prefer a different name like $mkPair$ for the constructor. The obvious single deconstructor for this is

```
> pair :: (a -> a -> b) -> Pair a -> b
> pair p (Pair x y) = p x y
```

from which you can recover $first = pair\ const$ and $second = pair\ (flip\ const)$.

In the same way the natural single deconstructor for (α, β) is $uncurry$, and $fst = uncurry\ const$ and $snd = uncurry\ (flip\ const)$.

It is worth mentioning the predefined type

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

where different constructors have different numbers of arguments. I am reasonably confident $Maybe$ was named by Mike Spivey in a paper published in 1990. He has been known talk about *adverb-oriented programming*. There is a corresponding predefined deconstructor function

```
> maybe :: b -> (a -> b) -> Maybe a -> b
> maybe nothing just  Nothing = nothing
> maybe nothing just (Just x) = just x
```

In the special case where constructors have no arguments, they are distinct constants, for example

```
> data Bool = False | True
> data Day = Sunday | Monday   | Tuesday | Wednesday |
>                   Thursday | Friday  | Saturday
```

These are not strings: you have to explain how to print them, for example by

```
> instance Show Day where
>     show Sunday = "Sunday"
>     show Monday = "Monday"
>     ...
```

(Of course, you might want `show Sunday = "Sul"`, etc.)

Even though they are distinct there is no equality test unless you instantiate

```
> instance Eq Day where
>     Sunday == Sunday = True
>     Monday == Monday = True
>                    ...
>     _      == _      = False
```

This requires 49 equations if written out in full, but eight will do: seven for the *True* cases and one catch-all for all the others. Although it might perhaps be better to define

```
> daynum :: Day -> Int
> daynum Sunday = 0
> daynum Monday = 1
>              ...

> instance Eq Day where
>     d1 == d2 = daynum d1 == daynum d2
```

11.1 Strictness and polynomial types

These data types are made of sums (coproducts or alternatives) of products (or tuples). Constructors are never strict: the construction of a tuple necessaily distinguishes (say) $\bot :: Pair\ \alpha$ and $Pair\ \bot\ \bot$ and so on. Since the predefined pairs (and other tuples) are also products, the same is true for them.

However, pattern matching on constuctors is strict, even when there is only one constructor:

```
> zero :: (a,b) -> Int
> zero (x,y) = 0
```

is strict in its argument: $zero\ \bot = \bot$. A less strict function is defined by

```
> zero' :: (a,b) -> Int
> zero' xy = 0
```

which really is the constant zero function, because $zero'\ \bot = 0 \neq \bot$. These types are called *lifted* because the values you want have all been 'lifted' up the ($\sqsubseteq$) ordering by putting a $\bot$ underneath them.

Sometimes you really do not need the lifted type. A type like

```
> data Value a = Value a
> data Count a = Count a
```

might be used to distinguish a $Count\ Int$ used for one purpose from a $Value\ Int$ used for another. The typechecker would prevent one from being confused with the other.

However you would like the type of $Value\ Int$ to be the same as Int: you do not want to distinguish $Value\ \bot$ from $\bot$. Just for this there is a declaration

```
> newtype Value a = Value a
```

which makes the constructor $Value$ be strict. (The practical effect if this is that a $Value\ Int$ can be represented at run time by an Int: all the type checking happens at compile time, and there is no run time cost either in space or time.)

11.2 Recursive types

We have seen types like

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

which is isomorphic to $[\alpha]$, in which the actual contructors are $[\,]$ and $(:)$). Types like this include values with arbitrary numbers of constructors, like infinite lists.

The obvious deconstructors for $[a]$ are $null$, $head$, and $tail$, but the natural single function is

```
> list:: (a -> [a] -> b) -> b -> [a] -> b
> list cons nil    []  = nil
> list cons nil (x:xs) = cons x xs
```

This looks superficially like $fold$, but notice that it is not recursive. The arguments to $list$ would naturally be in the other order, and there are instances of this function in some libraries with the arguments the other way around. The function with arguments in the other order also appears as $uncons'$ in exercise 6.8. I have chosen to put the arguments here in the same order as those of $fold$, which in turn has them in the order of the arguments of $foldr$, for which there is a long historical precedent.

We might represent sets of things by

```
> data Set a = Empty | Singleton a | Union (Set a) (Set a)
```

but the values of these are trees, so equality is not *structural equality*. If you really wanted them to behave like sets, you would want something like

```
> instance Eq a => Eq (Set a) where
>   xs == ys = (xs `subset` ys) && (ys `subset` xs)
```

and then you would need to define (exercise 11.3) a recursive function $subset$.

11.3 Deriving standard instances

It can be tedious to define obvious instances of classes for types, so Haskell can provide these automatically. For example

```
> data Either a b = Left a | Right b
>                                deriving (Eq, Ord, Read, Show)
```

makes default definitions for $Eq\ (Either\ \alpha\ \beta)$ and so on. The instances of $Read$ makes $read :: String \to Either\ \alpha\ \beta$ be the function that parses your input to the interpreter to turn it unto an $Either\ a\ b$, and the $Show$ makes $show :: Either\ \alpha\ \beta \to String$ produce the representation that the interpreter would use to print an answer.

The equality defined by deriving Eq is structural equality, which is what you want for $Either\ \alpha\ \beta$, and will itself require $Eq\ \alpha$ and $Eq\ \beta$; but structural equality is not what you want for $Set\ \alpha$.

The ordering derived for $Either\ \alpha\ \beta$ makes all $Left\ x$ be less than all $Right\ y$ and otherwise compares $Left\ x < Left\ y$ iff $x < y$ and so on.

The class $Enum\ \alpha$ includes

```
> class Enum a where
>     succ     :: a -> a
>     pred     :: a -> a
>     toEnum   :: Int -> a
>     fromEnum :: a -> Int
```

used in translating list enumerations like $[x\,..\,y]$. The derived instance for Day would make $fromEnum\ Tuesday = 2$ and $succ\ Friday = Saturday$, and so on.

The class $Bounded\ \alpha$ includes

```
> class Bounded a where
>    minBound, maxBound :: a
```

The derived instance of $Bounded\ Day$ would make $minBound = Sunday$ and $maxBound = Saturday$.

11.4 Maps

The function $map :: (\alpha \to \beta) \to ([\alpha] \to [\beta])$ can be generalised from lists (of α) to other datatypes with one argument. One of the properties is that $map\ id$ is the identity on $[\alpha]$, another that $map\ f \cdot map\ g = map\ (f \cdot g)$.

The map for $Pair$ is

```
> mapPair :: (a -> b) -> (Pair a -> Pair b)
> mapPair f (Pair x y) = Pair (f x) (f y)
```

that for $Maybe$ is

```
> mapMaybe :: (a -> b) -> (Maybe a -> Maybe b)
> mapMaybe f Nothing  = Nothing
> mapMaybe f (Just x) = Just (f x)
```

that for Set is

```
> mapSet :: (a -> b) -> (Set a -> Set b)
> mapSet f  Empty        = Empty
> mapSet f (Singleton x) = Singleton (f x)
> mapSet f (Union xs ys) = Union (mapSet f xs) (mapSet f ys)
```

It would be good to be able to capture this commonality. The type class *Functor* does roughly this:

```
> class Functor f where
>   fmap :: (a -> b) -> f a -> f b
```

It does not capture the invariants that *fmap* preserves identity and composition: these are expected properties of any function that you define as an instance of *fmap*. (The predefined class also contains $(<\$) :: b \ \rightarrow f\ a \rightarrow f\ b$.)

Notice that f is not a type: it is a $type \rightarrow type$ function, so the instances are for example

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

> instance Functor Pair where
>    fmap f (Pair x y) = Pair (f x) (f y)

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

The effect of these definitions is that if $f :: \alpha \rightarrow \beta$ you can use $fmap\ f$ as a $[\alpha] \rightarrow [\beta]$ or as a $Maybe\ \alpha \rightarrow Maybe\ \beta$ and so on.

There is no instance of $Functor\ Either$, because the type constructor expects too many arguments, but there is an instance of $Functor\ (Either\ \alpha)$, as a functor on the second type. I am pretty sure I think there should not be, because this emphasises an accidental asymmetry between the two part of $Either$.

11.5 Folds

Recall that the *fold* for a data type T acts to replace each of the constructors of T with the arguments of the fold function, and when applied to the constructors returns the identity $T \rightarrow T$.

Where the type is not recursive neither is the fold function:

```
> foldPair :: (a -> a -> t) -> Pair a -> t
> foldPair pair (Pair x y) = pair x y

> foldMaybe :: b -> (a -> b) -> Maybe a -> b
> foldMaybe nothing just Nothing  = nothing
> foldMaybe nothing just (Just x) = just x
```

```
> foldEither :: (a -> c) -> (b -> c) -> Either a b -> c
> foldEither left right (Left x)  = left x
> foldEither left right (Right y) = right y
```

These non-recursive functions are all the same as the corresponding deconstructors.

The fold function for a recursive type of leaf-labelled binary trees like

```
> data BTree a = Leaf a | Fork (BTree a) (BTree a)
```

would itself be a recursive function

```
> foldBTree :: (a -> b) -> (b -> b -> b) -> BTree a -> b
> foldBTree leaf fork (Leaf x)   = leaf x
> foldBTree leaf fork (Fork l r) = fork (foldBTree leaf fork l)
>                                       (foldBTree leaf fork r)
```

or perhaps more sensibly

```
> foldBTree leaf fork = rec
>   where rec (Leaf x)   = leaf x
>         rec (Fork l r) = fork (rec l) (rec r)
```

Here is another kind of tree, a *rose tree*:

```
> data RTree a = RTree a [RTree a]
```

which is never empty, and where each node can have any number of children. There is a natural *map* on rose trees

```
> instance Functor RTree where
>       fmap f (RTree a ts) = RTree (f a) (map (fmap f) ts)
```

The corresponding fold function is

```
> foldRTree :: (a -> [b] -> b) -> RTree a -> b
> foldRTree node (RTree x ts) = node x (map (foldRTree node) ts)
```

These ideas generalise to a bushy tree with other arrangments of children, provided the map function on the list of children can be replaced by the appropriate *fmap*

```
> data Bush t a = Bush a (t (Bush t a))
```

so that $Bush\ []\ a$ is essentially the same as $RTree\ a$, and $Bush\ Maybe\ a$ is a type of non-empty lists, and so on.

```
> instance Functor t => Functor (Bush t) where
>       fmap f (Bush x ts) = Bush (f x) (fmap (fmap f) ts)
```

where on the right hand side the first, outer, call of *fmap* is that for the functor t, and the second, inner one is a recursive call of *fmap* for $Bush\ t$, and

```
> foldBush :: Functor t => (a -> t b -> b) -> Bush t a -> b
> foldBush bush (Bush x ts) = bush x (fmap (foldBush bush) ts)
```

Just as with lists, once the *foldBush* function is defined, *fmap* could be defined by

```
> instance Functor f => Functor (Bush f) where
>       fmap f = foldBush (Bush . f)
```

There is no type class with a general folding function in it, because folds have very different types from each other, depending on the numbers and types of constructors. (This is *not* what *Foldable* is! That is sadly something much more confusing.)

However, in general it is possible to write a map function as an instance of the corresponding fold.

11.6 Unfolds

There is an *unfold* to lists which is dual to *fold* from lists,

```
> unfold null head tail = rec
>          where rec x | null x    = []
>                      | otherwise = head x : rec (tail x)
```

The *fold* has parameters that replace the constructors of its list argument, and this *unfold* has parameters that echo the list deconstructors making decisions about placing constructors in the result.

The equivalent for the binary trees *BTree* might be

```
> unfoldBTree :: (b->Bool) -> (b->a) -> (b->b) -> (b->b) ->
>                                                 b -> BTree a
> unfoldBTree single value left right = rec
>       where rec x | single x  = Leaf (value x)
>                   | otherwise = Branch (rec (left x))
>                                        (rec right x))
```

Then, for example, the function that unfolds a non-empty list into a balanced tree would be

```
> build = unfoldBTree (null.tail) head left right
>           where left  xs = take (length xs `div` 2) xs
>                 right xs = drop (length xs `div` 2) xs
```

11.7 Aside: Generalising folds and unfolds

The usual presentation of *fold* for lists

```
> fold :: (a -> b -> b) -> b -> [a] -> b
> fold cons nil    []  = nil
> fold cons nil (x:xs) = cons x (fold cons nil xs)
```

uses pattern matching, but it could be written in terms deconstructors

```
> fold cons nil = rec
>      where rec xs | null xs   = nil
>                   | otherwise = cons (head xs) (rec (tail xs))
```

but it can also be written in terms of the single deconstructor

```
> list:: (a -> [a] -> b) -> b -> [a] -> b
> list cons nil    []  = nil
> list cons nil (x:xs) = cons x xs
```

In terms of *list* you can see that

```
> fold cons nil = rec where rec = list (shape rec cons) nil
```

Here the *shape* function

```
> shape f c x xs = c x (f xs)
```

captures that the $(:)$ constructor has a recursive instance of the list type as its second argument, and applied f to that sub-list.

If I were being honest, all the folds and deconstructors would take a tuple of arguments, and the functions corresponding to *shape* would act on the whole tuple. It happens that there are no recursive instances of list as components of $[\,]$ so *shape* does not have to affect *nil*.

An example of the use of this *fold* function, with

```
> shift = shape (*10) (+)
```

so that $shift\ d\ n = d + 10 \times n$ in place of $(:)$ in a list

```
fold shift 0 [1,2,3,4,5] = 54321
```

converts a list of decimal digit values (least significant digit first) into the integer which they represent.

The corresponding *unfold* function would take an argument which had a type reminiscent of the *list* deconstructor

```
> unfold :: ((a -> b -> [a]) -> [a] -> b -> [a]) -> b -> [a]
> unfold list = rec where rec = list (shape rec (:)) []
```

The declaration of *rec* is similar to that for the *fold*, supporting the claim that this is a recursion with the same structure as the recursive type (of lists). However, in the case of the recursion for *fold* the *list* was the real deconstructor and *cons* and *nil* were arguments to *fold*. In this definition the *cons* and *nil* are the real constructors, and the *list* is the argument to unfold.

For example,

```
> unshift cons nil 0 = nil
> unshift cons nil x = cons (x`mod`10) (x`div`10)
```

is the deconstructor-shaped function that when used with *unfold* undoes the mapping from a list of digits to an integer:

```
unfold unshift 54321 = [1,2,3,4,5]
```

You can show that $fold\ shift\ 0 \cdot unfold\ unshift$ is the identity on natural numbers. The composition in the other order is the identity on lists of digits that have no leading zeroes.

Similar constructions work for other data types, for example for

```
> data BTree a = Leaf a | Fork (BTree a) (BTree a)
```

the natural single deconstructor is

```
> btree leaf fork (Leaf x)   = leaf x
> btree leaf fork (Fork l r) = fork l r
```

then the fold can be written in this style

```
> foldBTree :: (a -> b) -> (b -> b -> b) -> BTree a -> b
> foldBTree btree = rec
>                   where rec = btree Leaf (bshape rec Fork)
```

where this time both components of a *Fork* are trees, so both arguments to *Fork* need recursive calls of *foldBTree*

```
> bshape f fork l r = fork (f l) (f r)
```

The corresponding unfold is then

```
> unfoldBTree :: ((a->BTree a)->(b->b->BTree a)->b->BTree a)
>                                                 ->b->BTree a
> unfoldBTree leaf fork = rec
>                   where rec = btree leaf (bshape rec fork)
```

and in this form the unfold to a *BTree* that divides a non-empty list into a balanced binary tree is

```
> build = unfoldBTree split
>         where split leaf fork [x] = leaf x
>               split leaf fork xs  = uncurry fork (halve xs)
>               halve xs = splitAt (length xs `div` 2) xs
```

or slightly more efficiently, unfolding from a pair of a list and its length

```
> build = unfoldBTree split
>     where split leaf fork (1,[x]) = leaf x
>           split leaf fork (n, xs) = fork (n', ls) (n-n', rs)
>                                where n' = n `div` 2
>                                      (ls,rs) = splitAt n' xs
```

Exercises

11.1 What are the natural folds on *Bool* and

```
> data Day = Sunday | Monday   | Tuesday | Wednesday |
>                   Thursday | Friday  | Saturday
```

11.2 Given that the ordering on *Bool* is the one that would be obtained by **deriving**(*Ord*), to what logical function of two variables does (<=) correspond?

11.3 Write out the fold function for the data type

```
> data Set a = Empty | Singleton a | Union (Set a) (Set a)
```

and use it to define a function

```
> isIn :: Eq a => a -> Set a -> Bool
```

which tests whether an element appears as a value in the tree. Hence define a function

```
> subset :: Eq a => Set a -> Set a -> Bool
```

which tests whether all the elements of the first set are elements of the second. Use this to implement

```
> (==) :: Eq a => Set a -> Set a -> Bool
```

for equality of the sets represented by two trees from *Set*.

11.4 Define a function

```
> find :: Eq a => a -> BTree a -> Maybe Path
```

which searches for a value in the leaves of a *BTree*,

```
> data BTree a = Leaf a | Fork (BTree a) (BTree a)
```

returning a path, a sequence of *go left* and *go right* instructions, from the root to the leftmost occurrence of the value, if there is one, where

```
> data Direction = L | R
> type Path = [ Direction ]
```

You should aim to make use of folds and maps where possible.

11.5 A total version $head' :: [a] \to Maybe\ a$ of the standard function $head$ might be defined by

```
> head'   []  = Nothing
> head'(x:xs) = Just x
```

Show, by deriving a definition, that there is a unique function $(\bigoplus)$ satisfying

$$head'(xs +\!\!+ ys) \quad = \quad head'\ xs \bigoplus head'\ ys$$

11.6 The Quicksort algorithm (from section 6.3) can be expressed as a composition,

$$qsort = \mathit{flatten} \cdot build$$

where

$$\begin{array}{ll} \mathit{flatten} :: & QTree\ a \to [a] \\ build\ \ :: Ord\ a \Rightarrow & [a] \to QTree\ a \end{array}$$

for a type $QTree\ a$ which represents the recursive structure of $qsort$, and where $\mathit{flatten}$ is a fold from $QTree\ a$, and $build$ is an unfold to $QTree\ a$.

Define a suitable type, its fold and unfold functions, and use these to define $build$ and $\mathit{flatten}$.

11.7 The merge sort algorithm (from section 6.4) can be expressed as a composition,

$$msort = \mathit{flatten} \cdot build$$

where

$$\begin{array}{ll} \mathit{flatten} :: Ord\ a \Rightarrow & MTree\ a \to [a] \\ build\ \ :: & [a] \to MTree\ a \end{array}$$

for a type $MTree\ a$ which represents the recursive structure of $msort$, and where $\mathit{flatten}$ is a fold from $MTree\ a$, and $build$ is an unfold to $MTree\ a$.

Define a suitable type, its fold and unfold functions, and use these to define $build$ and $\mathit{flatten}$.

12 Some efficiency concerns

Recall that we first defined the $reverse$ function by recursion:

$$\begin{array}{rcl} reverse\ [\,] & = & [\,] \\ reverse\ (x : xs) & = & reverse\ xs +\!\!+ [x] \\ & = & snoc\ (reverse\ xs)\ x \\ & = & \mathit{flip}\ snoc\ x\ (reverse\ xs) \end{array}$$

Deduce from this that

$$reverse \quad = \quad fold\ (\mathit{flip}\ snoc)\ [\,]\ \textbf{where}\ snoc\ xs\ x = xs +\!\!+ [x]$$

However, this algorithm is quadratic: it takes about $\frac{1}{2}n^2$ steps to reverse a list of length n. Why is this? Each catenation

$$\begin{array}{rcl} [\,] +\!\!+ ys & = & ys \\ (x : xs) +\!\!+ ys & = & x : (xs +\!\!+ ys) \end{array}$$

(or $(+\!\!+ys) = fold\ (:)\ ys$) takes a number of steps linear in the length of its left argument. It follows that $snoc$ takes a number of steps linear in its list argument, and $reverse$ applies $snoc$ to (the reverse of) each tail of its argument.

The insight is that we could accumulate the answer: invent

$$revcat\ ys\ xs \quad = \quad reverse\ xs +\!\!+ ys$$

Notice that this is intended as a specfication, not the definition for execution: evaluating this would be at least as bad as the existing $reverse$. We could however use $revcat$ to calculate

$$\begin{array}{rl} & reverse\ xs \\ = & \{\,\text{unit of } (+\!\!+)\ \text{(proof?)}\,\} \\ & reverse\ xs +\!\!+ [\,] \\ = & \{\,\text{specification of } revcat\,\} \\ & revcat\ [\,]\ xs \end{array}$$

and then make the $(+\!\!+)$ vanish, by synthesizing a $(+\!\!+)$-less recursive definition of $revcat$

$$\begin{array}{rl} & revcat\ ys\ [\,] \\ = & \{\,\text{specification of } revcat\,\} \\ & reverse\ [\,] +\!\!+ ys \\ = & \{\,\text{definition of } reverse\,\} \\ & [\,] +\!\!+ ys \\ = & \{\,\text{definition of } (+\!\!+)\,\} \\ & ys \end{array}$$

and for non-empty lists

$$
\begin{array}{ll}
 & \mathit{revcat}\ \mathit{ys}\ (x : \mathit{xs}) \\
= & \{\text{specification of } \mathit{revcat}\} \\
 & \mathit{reverse}\ (x : \mathit{xs}) +\!\!+ \mathit{ys} \\
= & \{\text{definition of } \mathit{reverse}\} \\
 & (\mathit{reverse}\ \mathit{xs} +\!\!+ [x]) +\!\!+ \mathit{ys} \\
= & \{\text{associativity of } (+\!\!+)\} \\
 & \mathit{reverse}\ \mathit{xs} +\!\!+ ([x] +\!\!+ \mathit{ys}) \\
= & \{\text{definition of } (+\!\!+)\} \\
 & \mathit{reverse}\ \mathit{xs} +\!\!+ (x : \mathit{ys}) \\
= & \{\text{specification of } \mathit{revcat}\} \\
 & \mathit{revcat}\ (x : \mathit{ys})\ \mathit{xs}
\end{array}
$$

This gives us a definition

```
> reverse = revcat []
>           where revcat ys    []  = ys
>                 revcat ys (x:xs) = revcat (x:ys) xs
```

This one is linear in the length of the list being reversed: each call of *revcat* corresponds to one of the conses in the list, and each call does a constant amount of work before the recursive call.

The correspondence between conses and calls of *revcat* suggests that we think of a fold, but it is not a *fold* on cons-lists. Compare it with

$$
\begin{array}{rcl}
\mathit{loop}\ s\ n\ [\,] & = & n \\
\mathit{loop}\ s\ n\ (x : \mathit{xs}) & = & \mathit{loop}\ s\ (s\ n\ x)\ \mathit{xs}
\end{array}
$$

(which is *foldl*, the fold on snoc-lists) and by inspection $\mathit{revcat} = \mathit{loop}\ (\mathit{flip}\ (:))$ so

$$\mathit{reverse} \quad = \quad \mathit{loop}\ (\mathit{flip}\ (:))\ [\,]$$

12.1 Flattening trees

The flatten function for

```
> data BTree a = Leaf a | Fork (BTree a) (BTree a)
```

is $\mathit{flatten} :: \mathit{BTree}\ \alpha \to [\alpha]$ for which

$$
\begin{array}{rcl}
\mathit{flatten}\ (\mathit{Leaf}\ x) & = & [x] \\
\mathit{flatten}\ (\mathit{Fork}\ \mathit{ls}\ \mathit{rs}) & = & \mathit{flatten}\ \mathit{ls} +\!\!+ \mathit{flatten}\ \mathit{rs}
\end{array}
$$

The length of the result is the number of leaves in the tree, the size of the tree

$$size \quad = \quad foldBTree\ (const\ 1)\ (+)$$

however in general it takes more steps than that to produce it.

To flatten a balanced tree of size n there will be a $(+\!\!+)$ at the root that takes about $\frac{1}{2}n$ steps, below that two that take $\frac{1}{4}n$ steps each, and so on, which amounts to about $\frac{1}{2}n \log n$ steps. If the tree has a long left spine, the algorithm can be as bad as quadratic.

As before the insight is that to eliminate the $(+\!\!+)$ we should specify

$$flatcat\ t\ ys \quad = \quad flatten\ t +\!\!+ ys$$

and synthesize

$$\begin{array}{ll} & flatcat\ (Leaf\ x)\ ys \\ = & \{\,\text{specification of } flatcat\,\} \\ & flatten\ (Leaf\ x) +\!\!+ ys \\ = & \{\,\text{definition of } flatten\,\} \\ & [x] +\!\!+ ys \\ = & \{\,\text{definition of } (+\!\!+)\,\} \\ & x : ys \end{array}$$

and

$$\begin{array}{ll} & flatcat\ (Fork\ ls\ rs)\ ys \\ = & \{\,\text{specification of } flatcat\,\} \\ & flatten\ (Fork\ ls\ rs) +\!\!+ ys \\ = & \{\,\text{definition of } flatten\,\} \\ & (flatten\ ls +\!\!+ flatten\ rs) +\!\!+ ys \\ = & \{\,\text{associativity of } (+\!\!+)\,\} \\ & flatten\ ls +\!\!+ (flatten\ rs +\!\!+ ys) \\ = & \{\,\text{specification of } flatcat\,\} \\ & flatcat\ ls\ (flatcat\ rs\ ys) \end{array}$$

so $flatcat = foldBTree\ (:)\ (\cdot)$ and $flatten\ t = foldBTree\ (:)\ (\cdot)\ t\ [\,]$. Relying on the associativity of $(+\!\!+)$, synthesis has produced a linear algorithm from a less efficient one.

12.2 Associativity and folds

When is $fold\ (\oplus)\ e = loop\ (\otimes)\ f$?

Suppose we try to prove this by induction. The assertion is an equation so is chain complete (assuming $(\oplus)$ and $(\otimes)$ are computable) and both sides are

strict by virtue of the definitions of $fold$ and $loop$. Applying both sides to $[\,]$ shows that it is necessary that $e = f$. The substantial part of the proof is

$$\begin{array}{ll} & fold\ (\oplus)\ e\ (x : xs) \\ = & \{\text{definition of } fold\} \\ & x \oplus fold\ (\oplus)\ e\ xs \\ = & \{\text{lemma to be proved}\} \\ & loop\ (\otimes)\ (e \otimes x)\ xs \\ = & \{\text{definition of } loop\} \\ & loop\ (\otimes)\ e\ (x : xs) \end{array}$$

The essence of the result is the missing lemma, again to be proved by induction.

The lemma is chain complete (again assuming $(\oplus)$ and $(\otimes)$ are computable). If $xs = \bot$ conclude that $x \oplus \bot = \bot$ for all x, so $(\oplus)$ must be strict in its second argument. If $xs = [\,]$ conclude that $e \otimes x = x \oplus e$. The substantial part of the proof of the lemma is

$$\begin{array}{ll} & loop\ (\otimes)\ (e \otimes x)\ (y : ys) \\ = & \{\text{definition of } loop\} \\ & loop\ (\otimes)\ ((e \otimes x) \otimes y)\ ys \\ = & \{\text{suppose } (a \otimes b) \otimes c = a \otimes (b \odot c)\} \\ & loop\ (\otimes)\ (e \otimes (x \odot y))\ ys \\ = & \{\text{induction hypothesis}\} \\ & (x \odot y) \oplus fold\ (\oplus)\ e\ ys \\ = & \{\text{suppose } (a \odot b) \oplus c = a \oplus (b \oplus c)\} \\ & x \oplus (y \oplus fold\ (\oplus)\ e\ ys) \\ = & \{\text{definition of } fold\} \\ & x \oplus fold\ (\oplus)\ e\ (y : ys) \end{array}$$

Notice that this is a proof for all values of x, and the induction hypothesis is that it holds for a particular ys and all values in the x position, in particular $(x \odot y)$.

Collecting the requirements:

$$fold\ (\oplus)\ e \quad = \quad loop\ (\otimes)\ e$$

is proved for right-strict $(\oplus)$, provided $e \otimes x = x \oplus e$ and provided there is a $(\odot)$ for which $a \otimes (b \odot c) = (a \otimes b) \otimes c$ and $(a \odot b) \oplus c = a \oplus (b \oplus c)$. The obvious case is when all three of $(\oplus)$, $(\otimes)$ and $(\odot)$ are equal, are right-strict, are associative, and have e as a left and right unit.

$$\begin{array}{rcl} sum & = & fold\ (+)\ 0 = loop\ (+)\ 0 \\ product & = & fold\ (\times)\ 1 = loop\ (\times)\ 1 \\ concat & = & fold\ (+\!\!+)\ [\,] \neq loop\ (+\!\!+)\ [\,] \end{array}$$

This last inequality is possible because $xs +\!\!+ \bot \neq \bot$. The *fold* form produces output when applied to an infinite list of lists provided at least one of them is non-empty, but the *loop* form cannot produce any output for an infinite (or partial) input.

12.3 Bounding space

One reason for preferring *loop* $(+)\ 0$ to *fold* $(+)\ 0$ is that the *fold* is generally obliged to build up the whole expression before any evaluation:

$$
\begin{aligned}
& \mathit{fold}\ (+)\ 0\ [1,2,3,4] \\
= \quad & 1 + \mathit{fold}\ (+)\ 0\ [2,3,4] \\
= \quad & 1 + (2 + \mathit{fold}\ (+)\ 0\ [3,4]) \\
= \quad & 1 + (2 + (3 + \mathit{fold}\ (+)\ 0\ [4])) \\
= \quad & 1 + (2 + (3 + (4 + \mathit{fold}\ (+)\ 0\ [\,]))) \\
= \quad & 1 + (2 + (3 + (4 + 0))) \\
= \quad & 1 + (2 + (3 + 4)) \\
= \quad & 1 + (2 + 7) \\
= \quad & 1 + 9 \\
= \quad & 10
\end{aligned}
$$

whereas the *loop* can safely evaluate the expression as it goes. In practice, because of lazy evaluation

$$
\begin{aligned}
& \mathit{loop}\ (+)\ 0\ [1,2,3,4] \\
= \quad & \mathit{loop}\ (+)\ (0+1)\ [2,3,4] \\
= \quad & \mathit{loop}\ (+)\ ((0+1)+2)\ [3,4] \\
= \quad & \mathit{loop}\ (+)\ (((0+1)+2)+3)\ [4] \\
= \quad & \mathit{loop}\ (+)\ ((((0+1)+2)+3)+4)\ [\,] \\
= \quad & (((0+1)+2)+3)+4 \\
= \quad & ((1+2)+3)+4 \\
= \quad & (3+3)+4 \\
= \quad & 6+4 \\
= \quad & 10
\end{aligned}
$$

the same space build-up can happen. To prevent it, *loop* would have to be made strict in this argument.

```
> loop' s   n     []  = n
> loop' s (!n) (x:xs) = loop' s (s n x) xs
```

The `!` decoration ensures that the argument is evaluated before the recursive call. (This decoration is now a language extension in Haskell and requires a flag, or the pragma `{-# LANGUAGE BangPatterns #-}` at the top of a script.)

12.4 Fast exponentiation

On the face of it, calculating x^n appears to require about n multiplications. But multiplication is associative, so $x^{2n} = (x^2)^n$ and x^{2n} can be calculated in only one more multiplication than x^n. So we could specify $pow\ x\ n = x^n$ and synthesize

```
pow x 0 = 1
pow x n | even n = pow (x*x) (n`div`2)
        | odd n  = pow x (n-1) * x
```

This function will be called no more than $2 \log n$ times in x^n.

However, just like the *fold* version of *product*, this function must unnecessarily build up a big expression before any evaluation. Specify $power\ y\ x\ n = pow\ x\ n \times y$ and synthesize

$$\begin{array}{rcl} power\ y\ x\ 0 & = & pow\ x\ 0 \times y \\ & = & 1 \times y \\ & = & y \\ power\ y\ x\ n \mid even\ n & = & pow\ x\ n \times y \\ & = & pow\ (x \times x)\ (n\ \mathbf{div}\ 2) \times y \\ & = & power\ y\ (x \times x)\ (n\ \mathbf{div}\ 2) \\ power\ y\ x\ n \mid odd\ n & = & pow\ x\ n \times y \\ & = & (pow\ x\ (n-1) \times x) \times y \\ & = & pow\ x\ (n-1) \times (x \times y) \\ & = & power\ (x \times y)\ x\ (n-1) \end{array}$$

We could also abstract on the multiplication:

```
> power (*) y x n -- x^n*y
>       | n == 0 = y
>       | even n = power (*) y (x*x) (n`div`2)
>       | odd n  = power (*) (x*y) x (n-1)
```

Notice that the development of this code used only the associativity of $(\times)$, so it will calculate other repeated operations such as repeated matrix multiplication.

Exercises

12.1 A *queue* is a data type with (at least) four operations

```
> empty   :: Queue a
> isEmpty :: Queue a -> Bool
> add     :: a -> Queue a -> Queue a
> get     :: Queue a -> (a, Queue a)
```

The value of $empty$ is a queue with nothing in it; a queue satisfies $isEmpty$ if all of the values that have been added to it have already been removed; add puts a value into a queue; and get returns the oldest value still waiting in the queue, along with a queue from which just that value has been removed.

Implement a queue type using a list of the elements in the queue in the order in which they joined. That is, give a declaration of the $Queue$ type, and implement each of these four functions.

Estimate roughly how expensive your operations are. Would your answer be any different if the queue were represented by a list of its remaining elements in the reverse of the order in which they join the queue?

Reimplement the $Queue$ using two lists of elements, $front$ and $back$ so that the elements in the queue are those in the list $front +\!\!+ reverse\ back$. What effect does this have on the cost of the operations?

12.2 The Fibonacci sequence

```
> fib 0 = 0
> fib 1 = 1
> fib n = fib (n-1) + fib (n-2)
```

grows very quickly (each value is about 1.6 times bigger than its predecessor).

Use this definition in a GHCi script and try evaluating $fib\ 10$, $fib\ 20$ and $fib\ 30$. Give a brief explanation of why the later calls are so slow.

Let $two\ n = (fib\ n,\ fib\ (n+1))$, and synthesize a definition of two by direct recursion. Use this to give a more efficient definition of fib. How does the time it takes to calculate $fib\ n$ in this way depend on n?

Roughly how big is the $10\,000$th Fibonacci number? You might want to use

```
> roughly :: Integer -> String
> roughly n = x : 'e' : show (length xs) where x:xs = show n
```

to produce a readable estimate.

Let F be the matrix $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$, and F^n be its nth power, the product of n copies of it.

Explain why $F^n = \begin{pmatrix} fib\ (n-1) & fib\ n \\ fib\ n & fib\ (n+1) \end{pmatrix}$ for $n \geqslant 1$. Use the function $power$ from the lecture notes to calculate F^n in no more than about $2 \log n$ matrix multiplications, and use this to give another more efficient definition of fib.

Roughly how big is the $1\,000\,000$th Fibonacci number?

12.3 Find a closed form for value of the nth Fibonacci number, $fib\ n$, as defined in exercise 12.2, and for

```
> nfib 0 = 1
> nfib 1 = 1
> nfib n = 1 + nfib (n-1) + nfib (n-2)
```

which counts the number of calls of *fib* which result from a call of *fib* n.

12.4 Recall that the Haskell function

```
error :: String -> a
```

never terminates successfully, but prints out a message including its argument. Using the definitions of *loop* and *loop*′ from the lectures, and a function

```
> test f = f (const error) () ["strict","lazy"]
```

try to predict what happens when you evaluate each of *test loop* and *test loop*′.

Use GHCi to check your prediction, and explain the difference between the two.

What about *test foldl*?

13 Countdown, the setup

Countdown is a daytime television 'quiz' programme, in which in one of rounds contestants are given six source numbers and a target, all positive integers. The aim is to find an arithmetic expression using some or all of the source numbers with a value as close as possible to the target. Each source number can be used at most once, expressions are built using only (+), (-), (*), and (/), and all intermediate expressions have to be positive integers.

13.1 Abstract syntax trees

Expressions will be represented by trees

```
> data Expr = Val Int | App Op Expr Expr
> data Op = Add | Sub | Mul | Div
```

with all compound expressions represented by the same *App* node, labelled with its operator.

We will not use it here, but there would of course be a fold function

```
fold :: (Int -> b) -> (Op -> b -> b -> b) -> Expr -> b
fold val app = f where f (Val n)     = val n
                       f (App o l r) = app o (f l) (f r)
```

In the context of this problem it will make sense to *show* expressions with apparently unnecessary parentheses to reveal the structure.

The value of an expression, if it has one, is an *Int*.

```
> eval :: Expr -> [ Int ]
> eval (Val n)     = [ n | n > 0 ]
> eval (App o l r) =
>       [ apply o x y | x <- eval l, y <- eval r, valid o x y ]
```

The form of *eval* suggests *fold id apply* where

```
> apply :: Op -> Int -> Int -> Int
> apply Add = (+)
> apply Sub = (-)
> apply Mul = (*)
> apply Div = div
```

but the results have to pass a validity test, which is that every node in the evaluation satisfies

```
> valid :: Op -> Int -> Int -> Bool
> valid Add _ _ = True
> valid Sub x y = x > y
> valid Mul _ _ = True
> valid Div x y = x `mod` y == 0
```

ensuring that intermediate results are positive and divisions are exact.

We are using $[Int]$ for the outcome instead of $Maybe\ Int$ purely for the convenience of being able to use catenation later.

13.2 Checking solutions

A *Countdown* problem consists of some numbers ns, and a target number n; an expression e is a solution to that problem if the value of e is n, and the values that appear in e are chosen from ns.

```
> solution :: Expr -> [Int] -> Int -> Bool
> solution e ns n = (values e `elem` choices ns) && eval e == [n]
```

The values that appear in an expression can be found by flattening the expression

```
> values :: Expr -> [Int]
> values (Val n)     = [n]
> values (App _ l r) = values l ++ values r
```

which is again a fold on expressions.

The choices available from some list xs are subsequences of its permutations

```
> choices :: [a] -> [[a]]
> choices xs = [ ps | ys <- subs xs, ps <- permutations ys ]
```

The $permutations$ function we have already seen; and subsequences can be defined by

```
> subs :: [a] -> [[a]]
> subs xs = [ y:zs | y:ys <- tails xs, zs <- [] : subs ys ]
```

in this problem, we only need non-empty subsequences, though including the empty one would not affect the value of $solution$ because every result from $values$ is non-empty.

There are 1956 permutations of non-empty subsequences of six numbers, so given a solution e it is straightforward to check that it is a $solution$ to a given problem.

13.3 Generating solutions

The function $choices$ already generates all permutations of subsequences of the available numbers. All expressions made up from those numbers can be constructed by building all possible trees on each one of these $choices$.

```
> exprs :: [Int] -> [Expr]
> exprs [n] = [Val n]
```

```
> exprs ns = [ App o l r | (ls,rs) <- split ns,
>                          l <- exprs ls,
>                          r <- exprs rs,
>                          o <- [ Add, Sub, Mul, Div ] ]
```

For non-leaves, the (non-empty) sequence of numbers must be split into two non-empty segments

```
> split :: [a] -> [([a],[a])]
> split   [_]  = []
> split (x:xs) = ([x],xs) : [(x:ls,rs) | (ls,rs) <- split xs ]
```

Each of these must be non-empty, so that the other is a strict subsequence of the whole.

13.4 Checking generated solutions

The solutions to a problem (ns, n) are those expressions e that can be generated from choices of ns and which have value n.

$$solutions\ ns\ n \quad = \quad [e \mid cs \leftarrow choices\ ns, e \leftarrow exprs\ xs, v \leftarrow eval\ e, v == n]$$

One way of organising this computation (guided by changes to be made later) is to collect all the expression-value pairs

```
> type Result = (Expr, Int)

> results :: [Int] -> [Result]
> results ns = [ (e, v) | e <- exprs ns, v <- eval e ]
```

and to filter these for whether they are solutions

```
> solutions :: ([Int] -> [Result]) -> [Int] -> Int -> [Expr]
> solutions results ns n
>     = [ e | cs <- choices ns, (e,v) <- results cs, v == n]
```

This completes a solver, which can (often) rapidly find a solution to a solvable problem,

```
*Lecture13> head (solutions results [1,3,7,10,25,50] 832)
7+(3*((1+10)*25))
(0.12 secs, 72,199,336 bytes)
```

but takes a very long time to search for a solution to a problem with no exact solution

```
*Lecture13> solutions results [1,3,7,10,25,50] 831
[]
(140.54 secs, 86,036,396,088 bytes)
```

because it explores more than thirty million possible expressions, more than four and a half million of which are valid.

13.5 Pruning the range of possible expressions

Since validity of an expression requires validity of its subsexpressions, there is no point constructing an expression out of invalid subexpressions and then checking for validity. A fusion of the test for validity with evaluation gives

```
> prunedresults :: [Int] -> [Result]
> prunedresults [n] = [(Val n, n) | n > 0 ]
> prunedresults ns = [ (App o l r, apply o x y)
>                    | (ls, rs) <- split ns,
>                      (l,x) <- prunedresults ls,
>                      (r,y) <- prunedresults rs,
>                      o <- [ Add, Sub, Mul, Div ],
>                      valid o x y ]
```

This program still constructs as many valid expressions, but dismisses the invalid ones more quickly

```
Lecture13> layn (solutions prunedresults numbers 832)
  1) 7+(3*((1+10)*25))
  2) 7+((3*(1+10))*25)
     ...
271) 50+((25*((10*3)+1))+7)
272) (50+(25*((10*3)+1)))+7
(8.17 secs, 4,335,352,752 bytes)
```

Even the pruned results include many expressions that add nothing to the existence of a solution: commutative operators mean that many subexpressions will appear both ways round with the same result. We could eliminate many expressions by requiring the values of the subexpressions in additions and multiplications to be ordered, and by eliminating multipication and division by one.

This could be done by strengthening the *valid* test to

```
> useful :: Op -> Int -> Int -> Bool
> useful Add x y = x <= y
> useful Sub x y = x > y
> useful Mul x y = x /= 1 && y /= 1 && x <= y
> useful Div x y = y /= 1 && x `mod` y == 0
```

Most of the solutions have been eliminated, leaving in this example fewer than a quarter of a million useful valid expressions, however there is guaranteed to be a remaining solution whenever there are any solutions.

```
Lecture13> layn (solutions usefulresults numbers 832)
  1) 7+(3*((1+10)*25))
  2) 7+((1+10)*(3*25))
  3) 7+(25*(3*(1+10)))
     ...
 13) 7+((50-25)*(3*(1+10)))
```

```
 14) 50+(7+(25*(1+(3*10))))
(1.39 secs, 744,579,248 bytes)
```

There remain many expressions that are equivalent up to the associativity of addition and multiplication. Eliminating wasted work here seems best achieved by generating only expressions in which each combination of associative operations appears in a canonical order.

This code

```
> gen :: [Op] -> [Int] -> [Result]
> gen ops [n] = [(Val n, n) | n > 0 ]
> gen ops ns = [ (App o l r, apply o x y)
>              | (ls, rs) <- split ns,
>                o <- ops,
>                (l,x) <- gen (left o) ls,
>                (r,y) <- gen (right o) rs,
>                useful o x y ]
```

restricts the operators that can appear to the left and right of an operation so that any sum is written as a list (represented as a right-spine tree) of factors, and any difference is a difference of sums. Similarly a quotient is a quotient of products each of which is a right-spine list of differences or sums.

```
> left, right :: Op -> [Op]
> left Add = [Mul,Div]
> left Sub = [Add,Mul,Div]
> left Mul = [Add,Sub]
> left Div = [Add,Sub,Mul]

> right Add = [Add,Mul,Div]
> right Sub = [Add,Mul,Div]
> right Mul = [Add,Sub,Mul]
> right Div = [Add,Sub,Mul]
```

Whilst this reduces the number of solutions

```
Lecture13> layn (solutions (gen [Add, Sub, Mul, Div]) numbers 832)
  1) 7+(3*((1+10)*25))
  2) 7+((1+10)*(3*25))
  3) 7+(25*(3*(1+10)))
     ...
 11) 7+((50-25)*(3*(1+10)))
 12) 50+(7+(25*(1+(3*10))))
(14.81 secs, 8,344,349,064 bytes)
```

it slows the process down considerably.

Why is this? Because of the order of the generators, the whole process of generating subtrees is repeated up to four times, for each o drawn from ops. This means that grandchildren are generated up to sixteen times, greatgrandchildren up to sixty-four times and so on. It turns out to be much cheaper to generate the subtrees once (independently of the operator at this node)

```
> gen' ops ns  = [ (App o l r, apply o x y)
>                | (ls, rs) <- split ns,
>                  ((l,x),(r,y)) <- gen' ops ls `cp` gen' ops rs,
>                  o <- ops,
>                  needed o l r,
>                  useful o x y ]
```

and to discard the ones not needed

```
> needed :: Op -> Expr -> Expr -> Bool
> needed Add (App Add _ _) _ = False
> needed Add (App Sub _ _) _ = False
> needed Add _ (App Sub _ _) = False
> needed Sub (App Sub _ _) _ = False
> needed Sub _ (App Sub _ _) = False
> needed Mul (App Mul _ _) _ = False
> needed Mul (App Div _ _) _ = False
> needed Mul _ (App Div _ _) = False
> needed Div (App Div _ _) _ = False
> needed Div _ (App Div _ _) = False
> needed _   _             _  = True
```

Exercise

13.1 What is the fold function for the data type

```
> data Expr  = Num Int | App Op Expr Expr
> data Op    = Add | Sub | Mul | Div
```

of expressions used in the Countdown program?

Express *value* in terms of this fold function.

Explain how the test

```
> valid :: Op -> Value -> Value -> Bool
```

which checks for the legality of a node in an *Expr* can be used in a fold to check that an expression is entirely legal.

14 Countdown, dynamic programming solutions

The previous lecture defined solvers for the Countdown problem which took the form

$$\begin{array}{lcll} \mathit{solutions}\ ns\ n & = & [(e,v) \mid & ys \leftarrow \mathit{subs}\ ns, \\ & & & cs \leftarrow \mathit{permutations}\ ys, \\ & & & (e,v) \leftarrow \mathit{results}\ cs, \\ & & & v == n] \end{array}$$

and where the $\mathit{results}$ function was repeatedly refined. In each case the $\mathit{results}$ function then decomposes cs into subseqences on which it builds expressions. Consequently the same work is being done repeatedly.

14.1 Tabulation for efficiency

The well-known Fibonacci sequence, the image of the natural numbers under

```
> fib :: Int -> Integer
> fib 0 = 0
> fib 1 = 1
> fib n = fib (n-1) + fib (n-2)
```

is most naturally specified by that function definition. However, this is an inefficient way of computing elements of the sequence.

Since $\mathit{fib}\ n \approx \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n$, and since the result is produced by adding 1 to itself (and to some zeroes) this function (read as a program) gives an exponentially slow way of calculating elements of the sequence.

Observe that in calculating $\mathit{fib}\ n$ there are about $2 \times \mathit{fib}\ n$ calls of fib but only n different calls, since each call has an argument less than n. It would be better to calculate each of these once, and then use their values as needed.

One way of understanding tabulation is first to extract the *kernel* of the recursion for fib.

```
> fibK :: (Int -> Integer) -> (Int -> Integer)
> fibK f 0 = 0
> fibK f 1 = 1
> fibK f n = f (n-1) + f (n-2)
```

This function captures the meaning of the 'body' of the original function definition, without the recursion.

This kernel function is not itself recursive: all previously recursive calls are replaced by calls to the argument f, however $\mathit{fib}_K\ \mathit{fib} = \mathit{fib}$. That is, fib is a fixed point of fib_K, in the same way that 1 is a fixed point of $g\ x = (x^2+1)/2$, because $g\ 1 = 1$.

Define $\mathit{fix}\ k = x$ **where** $x = k\ x$, then $\mathit{fix}\ k$ is clearly a fixed point of k, and indeed $\mathit{fix}\ \mathit{fib}_K$ is equal to fib. (However, $\mathit{fix}\ g$ is not 1, because g is strict and so $\perp$ is also a fixed point of g; $\mathit{fix}\ k$ is always the least-defined fixed point of k.)

Of course, $fix\ fib_K$ will not only implement the same function as fib, but also it calls fib_K exactly as many times as the original program called fib, and so has the same exponentially poor performance.

The *dynamic programming* strategy for evaluating a function such as fib is to construct a table of values of the function, and to implement the function by looking up entries in that table. The table itself is constructed by calling the kernel, but with the table-lookup implementation as an argument.

In the case of fib, which is a function from natural numbers, the table, tab, might be a list of values of the function, in order. Then $fib\ n = tab\,!!\,n$. The table can be constructed by mapping an implementation of fib over the domain of the function, $[0..]$. That implementation consists of the kernel of the recursion, but with recursive calls implemented by table lookup.

```
> tabulate :: ((Int -> a) -> (Int -> a)) -> (Int -> a)
> tabulate kernel = fun
>                   where fun = (tab !!)
>                         tab = map (kernel fun) [0..]
```

Think of $tabulate$ as a more efficient implementation of fix, specialised to the type $Int \to a$, with arguments restricted to non-negative values of Int.

$$
\begin{array}{ll}
 & fun\ n \\
= & \{\text{local definition of } fun\} \\
 & tab\ !!\ n \\
= & \{\text{local definition of } tab\} \\
 & (map\ (kernel\ fun)\ [0..])\ !!\ n \\
= & \{(map\ f\ xs)\ !!\ n = f\ (xs\ !!\ n), \text{ for infinite } xs, \text{ by induction on } n\} \\
 & kernel\ fun\ n
\end{array}
$$

so $fun = kernel\ fun$ and (because it is unique) this is $fix\ kernel = fib$.

With this definition, $tabulate\ fib_K = fib$ is an implementation of fib which only ever calculates $fib_K\ (\ldots)\ n$ once for each value of n.

The table tab is infinite, but only a finite amount of it (the first $n + 1$ elements) will be explored and filled in for any call of $tabulate\ fib_K\ n$. However, that piece of the table which is instantiated grows with n so the cost of reducing the time taken by the program is an increase in the space which it must occupy.

(This is not a linear implementation of fib, because it calls fun about twice for each natural number less than n, and fun is linear in its numerical argument, so the resulting implementation of fib is quadratic.)

Strictly speaking, the name *dynamic programming* is usually used to mean tabulation, with predetermined bounds on the table to be used, along with a schedule for evaluating the elements of the table. The schedule guarantees that each entry in the table is filled in before it is looked up. In our program, we rely on lazy evaluation to evaluate the table entries in an order which provides the answers when they are needed. This works provided that there is an ordering on the arguments with respect to which recursive calls are smaller than the

call which causes them. Such an ordering must have existed in order for the original recursive function to terminate. In the case of *fib*, this is just the usual ordering on numbers.

14.2 Tabulating results

The calculation of solutions to a countdown problem can reorganised to use a function which lists the solution expressions that use all of the numbers in some list. (The reasons for wanting to do that will become clear later.)

```
> solutions :: ([Int] -> [Result]) -> [Int] -> Int -> [Expr]
> solutions results ns n = [ e | (e,v) <- results ns, v == n ]

> results :: [Int] -> [Result]
> results ns = [ ev | ys <- subs ns, ev <- fix resultsK ys ]
```

The *results* function lists those expressions that use exactly ys for each subsequence ys of its argument ns, using a recursion described by $results_K$.

```
> resultsK :: ([Int]-> [Result]) -> ([Int] -> [Result])
> resultsK f [n] = [ (Val n, n) | n > 0 ]
> resultsK f ns  = [ (App o l r, apply o x y)
>                  | (ls, rs) <- subsplits ns,
>                    ((l,x),(r,y)) <- f ls `cp` f rs,
>                    o <- ops,
>                    needed o l r,
>                    useful o x y ]
```

This recursion filters for *useful* arrangements of values of subexpressions, but also for the associativity properties eliminated by *gen* from the previous lecture.

Binary expressions are constructed by dividing ns into two subsequences, rather than just an initial segment and a final segment. The *subsplits* function divides a non-empty list into a pair of non-empty subsequences in all possible ways. (They must be non-empty sequences, because the program later relies on *the other* element of the pair being smaller than the list being split.)

```
> subsplits :: [a] -> [([a], [a])]
> subsplits (x:[]) = []
> subsplits (x:xs) = [ ([x],xs), (xs,[x]) ] ++
>                    concat [ [ (x:ls,rs), (ls,x:rs) ]
>                           | (ls,rs) <- subsplits xs ]
```

This leads to fewer arguments to *results* than the strategy in the previous lecture, because the numbers in the argument of $results_K$ are always in the order in which they appeared in the problem.

There is as yet no tabulation in this new implementation, but it is slightly faster than the fastest solution from the previous lecture.

```
Lecture14> head (Lecture13.approx Lecture13.usefulresults numbers 831)
(7+(3*((1+10)*25)),832)
(1.89 secs, 849,541,120 bytes)

Lecture14> head (approx results numbers 831)
(7+(3*((1+10)*25)),832)
(0.94 secs, 416,264,712 bytes)
```

14.3 Naïve tabulation

A table for the $results$ function has to represent a function $[Int] \to Int$, with domain $subs\ ns$ for some six numbers ns.

One perfectly general solution represents an $Eq\ a \Rightarrow a \to b$ by an *association list* of pairs $[(a, b)]$, which for clarity later, we tag with the constructor $Mapping$

```
> data Mapping a b = Mapping [(a,b)]
```

then the association list with domain xs is just a list of pairs of elements of the domain together with their image under the function which the list represents.

```
> toMapping :: [a] -> (a -> b) -> Mapping a b
> toMapping xs f = Mapping [ (x, f x) | x <- xs ]
```

Values of $f\ x$ can be recovered from the map by finding the (or the first) value paired with x.

```
> getMapping :: Eq a => Mapping a b -> a -> b
> Mapping m `getMapping` x = head [ b | (a,b) <- m, a == x ]
```

With this implementation of a mapping the fixed point of $results_K$ can be tabulated with an association list indexed by the subsequences of ns.

```
> tabresults :: [Int] -> [Result]
> tabresults ns = [ ev | xs <- dom, ev <- fun xs ]
>                 where fun = (tab `getMapping`)
>                       tab = dom `toMapping` (resultsK fun)
>                       dom = subs ns
```

This function constructs the table once only for each problem (one with numbers ns) so calls $results_K$ only $2^6 = 64$ times.

```
Lecture14> head (approx tabresults numbers 831)
(7+(3*((1+10)*25)),832)
(0.33 secs, 143,709,704 bytes)
```

However it makes many calls of $getMapping$, each of which involves exploring the table, which is itself 64 long.

14.4 Using a Trie for tabulation

The domain of the mapping is a set of lists, and a possibly more efficient strategy for tabulating a prefix-closed set of lists is to use a *Trie* (possibly meant to be pronounced *tree*) which is a particular form of *rose tree*.

```
> data Trie a b = Trie b (Mapping a (Trie a b))
```

just as every rose tree contains a forest of subtrees, you can think of this trie as containing a (small) forest of tries

```
> data Trie a b = Trie b (Copse a b)
> type Copse a b = Mapping a (Trie a b))
```

The nodes of a $Trie\ a\ b$ for some function $f :: [a] \to b$ correspond to a set of prefixes of some lists from $[a]$ together with the values of the function at those prefixes. The root contains $f\ [\,]$; the copse contains the values of f at non-empty lists with $f(x : xs)$ in the trie found by looking for x in the mapping in the root:

```
> getTrie :: Eq a => Trie a b -> [a] -> b
> (Trie y _) `getTrie` [] = y
> (Trie _ m) `getTrie` xs = m `getCopse` xs

> getCopse :: Eq a => Copse a b -> [a] -> b
> m `getCopse` (x:xs) = (m `getMapping` x) `getTrie` xs
```

How to construct a trie for a function on the subsequences of a list? One way is to consider the subsequences generated in lexicographical order of the positions in the original list.

```
> subs :: [a] -> [[a]]
> subs xs = [ y:zs | (y:ys) <- tails xs, zs <- [] : subs ys ]
```

Here the *tails* are the non-empty tails of a list.

```
> tails :: [a] -> [[a]]
> tails [] = []
> tails xs = xs : tails (tail xs)
```

An analogous recursion produces a trie for the subsequences of xs.

```
> toCopse :: [a] -> ([a] -> b) -> Copse a b
> xs `toCopse` f = Mapping [ (y, ys `toTrie` (f.(y:)))
>                          |  y:ys <- tails xs ]

> toTrie :: [a] -> ([a] -> b) -> Trie a b
> xs `toTrie` f = Trie (f []) (xs `toCopse` f)
```

Here the function $f \cdot (y :)$ is tabulated by the trie that is reached from the root by looking up y, so a value found by looking up a path ys in that subtrie can be found by looking up $y : ys$ from the root.

The tabulation of the results looks very similar to the earlier one, except that $toTrie$ produces a trie with a domain that is not its list argument, but the subsequences of that argument

```
> trieresults :: [Int] -> [Result]
> trieresults ns = [ ev | xs <- subs ns, ev <- fun xs ]
>                  where fun = (tab `getTrie`)
>                        tab = ns `toTrie` (resultsK fun)
```

and since we only want to tabulate values for non-empty sequences

```
> copseresults :: [Int] -> [Result]
> copseresults ns = [ ev | xs <- subs ns, ev <- fun xs ]
>                   where fun = (tab `getCopse`)
>                         tab = ns `toCopse` (resultsK fun)
```

and the resulting program is significantly faster. The effect is more noticeable with bigger problems, for example with seven numbers a trie is a clear factor of two better than an association list.

14.5 Approximate solutions

Countdown rewards the nearest solution to a problem, at least provided that it is within ten of the target. The nearest solution might be calculated by

```
> approx :: ([Int] -> [Result]) -> [Int] -> Int -> [Result]
> approx results ns n
>  = sortOn (near n) [ r | cs <- choices ns, r <- results cs]
>    where near n (_,v) = abs (n-v)
```

The library function

$$Data.List.sortOn :: Ord\ b \Rightarrow (a \to b) \to [a] \to [a]$$

sorts a list so that $map\ f\ (sortOn\ f\ xs)$ is in ascending order.

15 Building a parser

A *parser* takes a concrete representation of some abstract data and identifies the abstract object. You can think of it as an inverse to $show$, and indeed there is a predefined polymorphic $read :: Read\ a \Rightarrow String \to a$. How might you write instances of $Read$ for complex objects?

The sort of thing we want to be able to do is

```
*Main> eval `fmap` (expr `from` "12*(3+4)")
Just 84
```

where the language of the string is described by a little grammar

```
> expr, term, factor :: Parser Expr
> expr   = addop `chain` term
> term   = mulop `chain` factor
> factor = number <|> parens "(" expr ")"
```

and the type $Expr$ contains abstract syntax trees like those from the previous lectures.

A parser for things of type a must take a string and consume enough of the string to find a representation of an a. It should also hand back the rest of the string in case the context is looking for an a followed by something else. So perhaps a parser for a maps a $String$ to a pair $(a, String)$.

However there might be some ambiguity: "$1 + 2 + 3$" might be parsed into 1 and "$+ 2 + 3$", or 3 and "$+ 3$", or 6 and "" (the empty string), so our parsers will return a list of possible parses:

```
> type Parser a = String -> [(a, String)]
```

15.1 Primitive parsers

The simplest parser fails to parse anything

```
> fail :: Parser a
> fail xs = []
```

and the simplest successful one consumes none of its input, returning a value:

```
> return :: a -> Parser a
> return x xs = [(x,xs)]
```

A $char$ is the next single character, if there is one:

```
> char :: Parser Char
> char    []  = []
> char (x:xs) = return x xs
```

More usefully we might want to accept in a parse only a value that satisfies some predicate:

```
> (<?>) :: (a -> Bool) -> Parser a -> Parser a
> (c <?> p) xs = [ (a,ys) | (a,ys) <- p xs, c a ]
```

The $(\texttt{<?>})$ operator (read *satisfying*) is an analogue of $filter$ on parsers. For example

```
> character :: Char -> Parser Char
> character c = (c==) <?> char
```

matches exactly the given character from the input string.

15.2 Sequencing parsers

More generally

```
> string :: String -> Parser String
> string    ""  = return ""
> string (c:cs) = character c >> string cs >> return (c:cs)
```

matches exactly the sequence of characters in a string by matching first the head, and then (by recursion) the tail.

The sequencing operator $(\gg)$ runs two parsers in sequence

```
> (>>)  :: Parser a -> Parser b -> Parser b
> (p >> q) xs = [ (b,zs) | (a,ys) <- p xs, (b,zs) <- q ys ]
```

discarding the result of the first one, and returning the result of running the second on that part of the string not already consumed by the first parser.

More generally we might want not to discard the parsed value from the first parser. This operator

```
> (>>=) :: Parser a -> (a -> Parser b) -> Parser b
> (p >>= f) xs = [ (b,zs) | (a,ys) <- p xs, (b,zs) <- f a ys ]
```

(pronounced *bind* for reasons that might become more obvious in the next lecture) parses an a using the first parser p and then applies f to a to find a second parser to produce the final result. Typically, the $f\ a$ runs a parser essentially independent of the value a, and then combines a with the result of that parse. Of course, $p \gg q = p \gg\!= const\ q$.

Notice that the $string$ parser returns the string that was being matched, not something composed of the parsed values. An alternative formulation

```
> string :: String -> Parser String
> string    ""  = return ""
> string (c:cs) = character c >>= tail
>                 where tail y    = string cs >>= done y
>                       done y ys = return (y:ys)
```

would construct the result from the parsed characters.

15.3 Alternative parsers

A choice operator has type $Parser\ a \to Parser\ a \to Parser\ a$ and produces a parse from one or the other of its arguments. For example

```
> (<++>) :: Parser a -> Parser a -> Parser a
> (p <++> q) xs = p xs ++ q xs
```

is a sort of non-deterministic choice: $p <+\!\!+> q$ will parse and return anything that can be parsed by either p or by q. Usually we will be interested only in the best parse, and if we make sure that the left argument parser is more specific than the right argument, we will prefer a deterministic choice:

```
> (<|>) :: Parser a -> Parser a -> Parser a
> p <|> q = take 1 . (p <++> q)
```

which parses the best match that comes from the first parser, and only resorts to the second if the first fails entirely.

For example, if $some\ p$ parses a seqence of one or more of the things parsed by p, then

```
> many :: Parser a -> Parser [a]
> many p = some p <|> return []
```

will parse none or more, that is a potentially empty sequence of p. Of course, $some$ is defined in terms of $many$.

```
> some :: Parser a -> Parser [a]
> some p = p >>= ps
>          where ps a      = many p >>= done a
>                done a as = return (a:as)
```

A non-empty sequence consists of a head parsed by p and a tail parsed by ps. Equivalently

```
> some p = p >>= ps
>          where ps a  = many p >>= done
>                        where done as = return (a:as)
```

where the definitions of the local functions are nested. If you are familiar with lambda expressions, you might prefer

```
> some p = p >>= (\a -> many p >>= (\as -> return (a:as)))
```

however I think this is just a sign that we need an even better notation. Again, I will return to this in the next lecture.

A concrete example would be a parser that matches whitespace

```
> space :: Parser String
> space = many (isSpace <?> char)
```

This might be part of a parser that matches some token, followed by white space which is ignored

```
> token :: Parser a -> Parser a
> token p = p >>= done
>           where done a = space >> return a
```

for example a symbol matching a given string

```
> symbol :: String -> Parser String
> symbol xs = token (string xs)
```

15.4 Expression parsers

For a concrete example, *symbol* "+" would match an addition operator. However this would return a '+' character as the corresponding value, and we would want to have the addition function $(+)$.

```
> addop = ((+) <$ symbol "+") <|> ((-) <$ symbol "-")
> mulop = ((*) <$ symbol "*") <|> (div <$ symbol "/")

> (<$>) :: (a -> b) -> (Parser a -> Parser b)
> (f <$> p) xs = [ (f x, ys) | (x,ys) <- p xs ]

> (<$) :: b -> Parser a -> Parser b
> x <$ p = const x <$> p
```

The type of $(\texttt{<\$>})$ is deliberately reminiscent of the type of *map* and *fmap*, and indeed if we had not carefully hidden it, $(\texttt{<\$>})$ is a predefined synomym for *fmap*.

The same mechanism can translate strings of digits

```
> digits :: Parser Int
> digits = foldl shift 0 <$> token (some digit)
>          where shift n d = 10*n+d
```

where *foldl* is the *loop* function for lists, and

```
> digit :: Parser Int
> digit = digitToInt <$> (isDigit <?> char)
```

The functions *isDigit* and *digitToInt* are defined in *Data.Char* and protect the program against having to know details of the character set.

In practice, *isDigit* c is likely to be `'0'` $\leqslant c \,\&\&\, c \leqslant$ `'9'`, and *digitToInt* $c =$ *fromEnum* $c -$ *fromEnum* `'0'`.

```
> number :: Parser Expr
> number = Val <$> digits
```

The *chain* function represents a number of p, each pair separated by an op, and associating to the left.

```
> chain :: Parser (a->a->a) -> Parser a -> Parser a
> chain op p = p >>= tail
>              where tail a     = (op >>= more a) <|> return a
>                    more a f   = p >>= done a f
>                    done a f b = tail (f a b)
```

The left association comes from the way that *tail* recurses, reminiscent of *foldl*. As before the mechanism required to carry around the results makes this difficult to read, and a better notation would be helpful.

To complete the functions used in the example grammar

```
> parens :: String -> Parser a -> String -> Parser a
> parens open p close = symbol open >> p >>= rest
>                       where rest n = symbol close >> return n
```

parses a p between two symbols which act as parethneses. This produces the same value as p but the concrete syntax is different.

15.5 Running the parser

A parser is run by applying it to the string that is to be parsed; here with *leading* space ignored.

```
> from :: Parser a -> String -> Maybe a
> from p xs =
>     case [ v | (v,ys) <- (space >> p) xs, null ys ] of
>       [v] -> Just v
>       _   -> Nothing
```

This returns a *Maybe* in case the parse is unsuccessful.

The parse is successful if it consumes all of the string, and is unambiguous. It is unsuccessful if either there is something left over, or if there are two possible parses.

16 Sequencing and Monads

In Haskell, the notion of a *Monad* abstracts from a common program structure like that of

```
> return :: a -> Parser a
> (>>=) :: Parser a -> (a -> Parser b) -> Parser b
```

There is a predefined type class (roughly)

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  j >> k = j >>= const k
```

where a value of $m\ a$ can be thought of as a computation returning a value or values of type a. It makes sense to define instances of $Monad$ only if the components satisfy the laws

$$\begin{aligned} return\ x \gg\!= k &= k\ x \\ j \gg\!= return &= j \\ j \gg\!= (\lambda x \to (k\ x \gg\!= h)) &= (j \gg\!= k) \gg\!= h \end{aligned}$$

These laws express (roughly) that $return$ is the left and right unit of bind, and the third law is an associative law.

The type of $Parser$ can be made an instance, although we need a name for the type function, and a type constructor to identify that type:

```
> newtype Parser a = Parser { parse :: String -> [(a, String)] }

> instance Monad Parser where
>   return x = Parser (\xs -> [(x,xs)])
>   p >>= f  = Parser (\xs -> [(v,zs) | (a,ys) <- p `parse` xs,
>                                       (v,zs) <- f a `parse` ys])
```

It might not be immediately obvious, but this satisfies the monad laws.

There are many other instances of the $Monad$ class which may be illuminating, for example

```
instance Monad []  where
  return x = [ x ]
  xs >>= f = [ y | x <- xs, y <- f x ]
```

In this case $xs \gg\!= f = concatMap\ f\ xs = concat\ (map\ f\ xs)$. The 'computation' in a list can perhaps be thought of as a list of possible results from some sort of non-deterministic calculation. It is perhaps easier to check that the monad laws hold here.

Similarly with

```
instance Monad Maybe where
  return = Just
  Nothing >>= f = Nothing
  Just x  >>= f = f x
```

where the 'computation' is one that might or might not succeed, and *Nothing* is a sort of exception representing failure.

16.1 do notation

Haskell provides a special notation for Monad-valued expressions.

$$\begin{aligned} \mathbf{do}\ \{x \leftarrow m; \mathit{stuff}\} &= m \gg\!= (\lambda x \rightarrow \mathbf{do}\ \{\mathit{stuff}\}) \\ \mathbf{do}\ \{m; \mathit{stuff}\} &= m \gg \mathbf{do}\ \{\mathit{stuff}\} \\ \mathbf{do}\ \{m\} &= m \end{aligned}$$

As with other constructs, the braces and semicolons are usually omitted when the **do** expressions are laid out on several lines, using the offside rule.

In the case of the Monad of lists,

$$\begin{aligned} & \mathbf{do}\ \{x \leftarrow xs; y \leftarrow f\ x; \mathit{return}\ (g\ x\ y)\} \\ = \quad & xs \gg\!= (\lambda x \rightarrow f\ x \gg\!= (\lambda y \rightarrow \mathit{return}\ (g\ x\ y))) \\ = \quad & \mathit{concat}\ (\mathit{map}\ (\lambda x \rightarrow f\ x \gg\!= (\lambda y \rightarrow \mathit{return}\ (g\ x\ y)))\ xs) \\ = \quad & \mathit{concat}\ (\mathit{map}\ (\lambda x \rightarrow \mathit{concat}\ (\mathit{map}\ (\lambda y \rightarrow \mathit{return}\ (g\ x\ y))\ (f\ x)))\ \ xs) \\ = \quad & \mathit{concat}\ (\mathit{map}\ (\lambda x \rightarrow \mathit{concat}\ (\mathit{map}\ (\lambda y \rightarrow [g\ x\ y])\ (f\ x)))\ \ xs) \\ = \quad & \mathit{concat}\ (\mathit{map}\ (\lambda x \rightarrow \mathit{concat}\ [[g\ x\ y] \mid y \leftarrow f\ x])\ \ xs) \\ = \quad & \mathit{concat}\ (\mathit{map}\ (\lambda x \rightarrow [g\ x\ y \mid y \leftarrow f\ x])\ \ xs) \\ = \quad & \mathit{concat}\ [[g\ x\ y \mid y \leftarrow f\ x] \mid x \leftarrow xs] \\ = \quad & [g\ x\ y \mid x \leftarrow xs, y \leftarrow f\ x] \end{aligned}$$

so the similarity between **do**-notation and list comprehension is deliberate. (You might wonder why list comprehension notation is not used for comprehensions in other monads, but that way madness lies.)

In terms of **do** notation, the Monad laws are

$$\left.\begin{array}{l} \mathbf{do} \\ \quad y \leftarrow \mathit{return}\ x \\ \quad k\ y \end{array}\right\} = k\ x$$

$$\left.\begin{array}{l} \quad\ \mathbf{do} \\ \quad\quad x \leftarrow j \\ \quad\quad \mathit{return}\ x \end{array}\right\} = j$$

$$\left.\begin{array}{l} \mathbf{do} \\ \quad x \leftarrow j \\ \quad \mathbf{do} \\ \quad\quad y \leftarrow k\ x \\ \quad\quad h\ y \end{array}\right\} = \left\{\begin{array}{l} \mathbf{do} \\ \quad y \leftarrow\ \mathbf{do} \\ \qquad\qquad x \leftarrow j \\ \qquad\qquad k\ x \\ \quad h\ y \end{array}\right.$$

The associative law justifies writing both sides of this last equation as

do
  $x \leftarrow j$
  $y \leftarrow k\ x$
  $h\ y$

and so on, and the unit laws allow for unnecessary *return* calls to be removed.
The beauty of *do* notation is that the plumbing in

```
> some :: Parser a -> Parser [a]
> some p = p >>= ps
>          where ps a      = many p >>= done a
>                done a as = return (a:as)
```

or

```
> some p = p >>= (\a -> many p >>= (\as -> return (a:as)))
```

is much easier to express as

```
> some p = do { a <- p; as <- many p; return (a:as) }
```

where of course the **do** expression in this example is a value in the *Parser* monad. The point of the *Monad* abstraction is that there will be a *some* function with similar properties in other monads.

16.2 Kleisli composition

It might not be obvious that

$$j \gg\!= (\lambda x \to (k\ x \gg\!= h)) \quad = \quad (j \gg\!= k) \gg\!= h$$

is an associative law: what exactly is associative?
Define the *Kleisli composition* by

```
> (>=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
> (f >=> g) x = f x >>= g
```

then the monad laws can be expressed as properties of $(\gg\!\Rightarrow)$

$$\begin{aligned} \mathit{return} \gg\!\Rightarrow k &= k \\ j \gg\!\Rightarrow \mathit{return} &= j \\ j \gg\!\Rightarrow (k \gg\!\Rightarrow h) &= (j \gg\!\Rightarrow k) \gg\!\Rightarrow h \end{aligned}$$

For example the Kleisli composition in the list monad is

$$\begin{aligned} & (f \gg\!\Rightarrow g)\ x \\ = \quad & f\ x \gg\!= g \\ = \quad & [z \mid y \leftarrow f\ x,\ z \leftarrow g\ y] \\ = \quad & (\mathit{concat} \cdot \mathit{map}\ g \cdot f)\ x \end{aligned}$$

16.3 Monadic join

In the list monad, $xs \gg\!= f = concat\ (map\ f\ xs)$. In every monad, it turns out, this same factorisation is possible. The equivalent of map is of course $(<\$>)$, and the equivalent of $concat$ is

```
> join :: Monad m => m (m a) -> m a
> join m = m >>= id
```

so in particular in the list monad

$$\begin{aligned} & join\ m \\ = \quad & m \gg\!= id \\ = \quad & concat\ (map\ id\ m) \\ = \quad & concat\ m \end{aligned}$$

and in general

$$m \gg\!= f \quad = \quad join\ (f <\$> m)$$

This means that it is possible to define the operations on a monad by giving not $return$ and $(\gg\!=)$ but $return$, $(<\$>)$ and $join$. Sometimes that is a more intuitive presentation.

In any case the Kleisli composition is then $f \gg\!\!> g = join \cdot (g <\$>) \cdot f$.

16.4 Applicatives and Functors

In any monad, you can define an operation (called *ap* or *apply*) that looks like application of a monadic function to a monadic argument:

```
(<*>) :: Monad m => m (a -> b) -> m a -> m b
fs <*> xs = do { f <- fs; x <- xs ; return (f x) }
```

(Notice the parentheses in $return\ (f\ x)$, because $return$ here is a function, not a syntactic component of the do construct.)

It is convenient for now also to have a different name for

```
pure :: Monad m => a -> m a
pure = return
```

In the case of the list monad, *pure* makes a singleton list and *apply* would do all of the applications of a function to an argument that you would find in the Cartesian product of a list of functions and a list of arguments. In the case of the Parser monad *pure* makes a parser that successfully returns a given value without consuming anything from the string, and *apply* would parse a function followed by an argument, and the result is a parse in which the two bits of input are both consumed and the result is the result of applying the function to the argument.

This *apply* operation is well behaved in many ways:

$$\begin{aligned} \mathit{pure}\ \mathit{id} <\!\!*\!\!> v &= v \\ \mathit{pure}\ (\cdot) <\!\!*\!\!> u <\!\!*\!\!> v <\!\!*\!\!> w &= u <\!\!*\!\!> (v <\!\!*\!\!> w) \\ \mathit{pure}\ f <\!\!*\!\!> \mathit{pure}\ x &= \mathit{pure}\ (f\ x) \\ u <\!\!*\!\!> \mathit{pure}\ y &= \mathit{pure}\ (\$\ y) <\!\!*\!\!> u \end{aligned}$$

Here $(\$\, y)$ is a right-section of the application operator, $f\ \$\ x = f\ x$.

These are the qualifications for being an instance of the $\mathit{Applicative}$ type class:

```
class Applicative m where
  pure :: a -> ma
  (<*>) :: m (a -> b) -> m a -> m b
```

so every monad is necessarily an *applicative*.

If you are bothered that *applicative* is not a noun, you are right: it turns out to be an *applicative functor*. Why is that? Given an applicative m (in particular, given any monad m) it is possible to define

```
(<$>) :: Applicative m => (a -> b) -> (m a -> m b)
f <$> xs = pure f <*> xs
```

which necessarily obeys the laws for *map*

$$\begin{aligned} \mathit{id} <\!\!\$\!\!> xs &= xs \\ (f \cdot g) <\!\!\$\!\!> xs &= f <\!\!\$\!\!> (g <\!\!\$\!\!> xs) \end{aligned}$$

or equivalently

$$\begin{aligned} (\mathit{id} <\!\!\$\!\!>) &= \mathit{id} \\ ((f \cdot g) <\!\!\$\!\!>) &= (f <\!\!\$\!\!>) \cdot (g <\!\!\$\!\!>) \end{aligned}$$

so every monad is a legitimate instance of the class

```
class Functor m where
  fmap :: (a -> b) -> m a -> m b
(<$>) = fmap
```

(A little Haskell oddness here is that $(<\!\!\$\!\!>)$ is not a component of $\mathit{Functor}$, so there is no way you can accidentally define it to be different from fmap.)

In fact, on the dubious grounds that "because you can, you should" (exercises 16.5 and 16.6), the three classes are defined in the standard libraries by headings that say

```
class Functor m where ...
class Functor m => Applicative m where ...
class Applicative m => Monad m where ...
```

which obliges you, when defining an instance of $Monad$ also to define the instance of $Applicative$ and when defining that also the instance of $Functor$. The intention is that the explicit definitions in $Functor$ will be more efficient than the general one that can be derived from $Applicative$, and those in $Applicative$ than those that can be derived from $Monad$.

16.5 Monadic Input and Output

The reason that monads first became such an important part of Haskell is that they capture the idea of sequencing effects, and this gives a way of sequencing the effects of input and output without leaving the functional programming language.

In the $Parser$ monad the effects being sequenced are the extent to which each parser consumes the input string. Parsers which appear in sequence in a do expression consume (if any) parts of the input string which appear in the same sequence in the string.

In the same way the effects in the IO monad are interactions with the real world, which happen in the order described by their sequence in a do. A thing of type $IO\ a$ is an interaction with the real world which yields a value of type a, so for example

```
readFile :: FilePath -> IO String
```

so when applied to the name of a file $readFile$ produces an IO value from which you can get a $String$ containing the sequence of characters in the file.

Simple output operations like

```
putStr :: String -> IO ()
```

have nothing significant to return, so produce an IO value from which you can get $()$ which is the only value of the type $()$ of null-tuples.

Exercises

16.1 Show that the monad laws hold in the list monad.

16.2 Show that the monad laws hold in the $Maybe$ monad.

16.3 Find the values of $join$ and $(\gg\!\!>)$ in the $Maybe$ monad.

16.4 If the Kleisli composition in some monad is defined by $(f \gg\!\!> g)\ x = f\ x \gg\!= g$ show that it satisfies each of

$$\begin{aligned} return \gg\!\!> k &= k \\ j \gg\!\!> return &= j \\ j \gg\!\!> (k \gg\!\!> h) &= (j \gg\!\!> k) \gg\!\!> h \end{aligned}$$

16.5 Show that in any *Applicative* the definition $f \mathbin{<\$>} xs = pure\ f \mathbin{<*>} xs$ satisfies the functor laws.

16.6 Show that in any *Monad* the definitions $pure = return$ and

$$fs \mathbin{<*>} xs = \mathbf{do}\,\{f \leftarrow fs;\ x \leftarrow xs;\ return\ (f\ x)\}$$

satisfy the *Applicative* laws.

16.7 Show that *Parser* satisfies the *Monad* laws.

16.8 Identify the *join*, $(<*>)$, $(>\!\!>\!\!=)$ of the *Parser* monad.