Design and Analysis of Algorithms

Contents

1 Functional Notation

1.1 Reduction Orders

Haskell uses lazy evaluation when processing an instruction: it works from the outermost function inwards, only evaluating expressions as they are needed.

$$\begin{aligned}
&\texttt{square}(3+4) && \\
&= (3+4)\times(3+4) && \text{definition of } \texttt{square}\\
&= 7\times 7 && \text{definition of } +\\
&= 49 && \text{definition of } \times
\end{aligned}$$

when there are no more steps to evaluate, an expression is said to be in *normal form*.

This method has the advantage that if there is *any* order to reduce a function to normal form, then the function will terminate.

If a function does not terminate, it returns $\perp$, any function that is defined as `f` $\perp = \perp$ is said to be strict. Nonstrict functions can only be defined with lazy evaluation.

1.2 Precedence

Functional application takes precedence over all other operators and associates to the left, hence `f x + 3 = (f x) + 3` and `f x y = (f x) y`

1.3 Functions as objects

It is possible for functions to take other functions as parameters, such functions are known as "first-class functions".

1.4 Functions as operators

If $\otimes$ is any infix function, then $(\otimes)$ is the corresponding prefix function, that is $x \otimes y = (\otimes)\ x\ y$.

Additionally, the following are all equivalent $(\otimes\ m)\ n = n \otimes m = (n\ \otimes)\ m = (\otimes)\ m\ n$ – this is called sectioning.

1.5 Guards

It is possible to define a function with specific cases, or base cases in mind. For example:

```
fact :: Integer → Integer
fact n
  | n == 0 = 1
  | n == 1 = 1
  | otherwise = n * fact (n-1)
```

1.6 Pattern Matching

It is possible to define a function with specific cases, or base cases in mind. For example:

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

data Day = Saturday | Sunday
    | Monday | Tuesday | Wednesday | Thursday | Friday
weekend :: Day → Bool
weekend Saturday = True
weekend Sunday = True
weekend d = False
```

It is important to define base cases first, as the compiler works from top-to-bottom, if `fact:3` and `fact:1` were swapped, the compiler would simply continue without ever checking for the base case.

This method also has the advantage that equality tests are not required in order for this to work, as opposed to using guards, and is considered to be closer to the mathematical definition.

1.7 Functional Composition

To compose two functions together, the following typing rules must be obeyed.

```
( ∘ ) :: (b → c) → (a → b) → a → c
(f ∘ g) x = f (g x)
```

1.8 Lambda Notation

This is a useful shorthand to express functions, and $(\lambda x \mapsto e) \equiv fx = e$. For example,

```
filter p ∘ filter q = filter (λx → p x && q x)
```

shows how lambda expressions can be used to produce a function inline instead of specifying it as a separate function.

1.9 Currying and Uncurrying

Instead of passing parameters to a function as `f (x,y)`, it is posible to pass them as two separate objects: `f x y`. In this case, it is possible to think of `f` as a function that takes an object `x` and returns a function that takes an object `y` and returns the result.

```
curry :: ((a,b) → c) → a → b → c
curry f x y = f (x,y)

uncurry :: (a → b → c) → (a,b) → c
uncurry f (x,y) = f x y
```

2 Types and Strong-Typing

Haskell is a strongly typed language, that is, every well-formed expression has an accompanying well-formed type. All functions should be accompanied by a type definition, which can also help in developing the function, and proving its correctness. For example:

```
square :: Integer → Integer
square (square 3) :: Integer
(1, 2/3) :: (Integer, Float)
sqrt :: Float → Float
```

When defining types, and creating new datatypes, a capital letter must be used to start the type.

2.1 Polymorphism

When an individual type is too specific, it is possible to specify a more general type.

```
id :: a → a
id x = x
```

`a` is used to denote any type, and is written in lower case, as opposed to starting with a capital.

When a function takes inputs of multiple types, each type is given its own specific letter.

```
flip :: (a → b → c) → b → a → c
flip f x y = f y x
```

When a function is passed as an argument, its types are contained within brackets. List operations, when polymorphic are denoted using square brackets around the type.

```
head :: [a] → a
head (x:xs) = x
```

2.2 Type Inference Rules

- `x :: a` and `y :: b` iff `(x,y) :: (a,b)`
- If `x :: a` and `f :: a -> b` then `f x :: b`
- If `x :: a` and `f x :: b` then `f :: a -> b`
- If `f x :: b` then `x :: a` and `f :: a -> b` for some type `a`.

2.3 Type Classes

When defining the type definition for multiplication, a type of `a → a` is too general, but specifying the type for each numeric datatype is too labourious.

There exist a number of predefined type classes in Haskell:

`Eq` Types with equality tests. Functions and IO are not part of this class.

`Ord` Types both with equality tests and order relations. Again, functions and IO are not part of this class.

`Show` Types that can be converted in to strings

`Num` Types with numerical representations

In the case when there is an obvious way to define operations to make a datatype a member of a given type class, it is possible to use the `deriving` keyword.

```
data Day
     = Monday | Tuesday |... deriving (Eq, Ord, Show)
```

To use a type class in a function definition, the following syntax is used:

```
sort :: Ord a ⇒ [a] → [a]
```

3 Lists and their operations

Lists are ordered collections of elements, all of the same type, and are represented by enclosing the list of data with square brackets, and separating each item with commas. For the type `a`, a list of `a` is given the type `[a]`.

The empty list is represented as `[]` and can be considered as the initial building block for lists. `(:)` appends a single elements to the front of a list and associates to the right (hence no need for brackets). The following definitions for lists are all equivalent: `[1,2,3,4]=1:[2,3,4]=1:2:3:4:[]`. The string representation in Haskell is actually just a list of characters, and `type String = [Char]`.

It is possible to pattern match over lists, by using the fact that lists are either empty, or a single element followed by a list (including the empty list).

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

Additionally, such pattern matching can be used when recursively defining functions.

```
length :: [a] → Integer
length [] = 0
length (x:xs) = 1 + length xs
```

3.1 List Comprehensions

List comprehensions provide a way to define a list, similar to set comprehensions for produce sets, with the following rules:

```
[ e | x ← xs ] = map (λx → e) xs
[ e | p ] = if p then [e] else []
[ e | x ←
     xs, Q ] = concat (map f xs) where f x = [ e | Q ]
[ e | p, Q] = if p then [ e | Q ] else []
[ f x | x ← xs ] = map f xs
[ x | x ← xs, p x ] = filter p xs
[ e | Q, P ] =  concat [ [ e | P ] | Q ]
```

where `P` and `Q` are qualifiers made up of a possibly empty sequence of generators and guards.

A generator is an expression similar to `x <- xs` and a guard is a predicate.

3.2 Maps over Lists

To apply a function to every element of a list, the standard higher-order function `map` exists.

```
map :: (a → b) → [a] → [b]
map _ [] = []
map f (x:xs) = f x : map f xs

map square [1..4] = [1,4,9,16]
```

3.3 Laws of `map`

```
(map f ∘ map g) = map (f ∘ g)
map id = id
head ∘ map f = f ∘ head
tail ∘ map f = map f ∘ tail
length ∘ map f = length
```

3.4 Filtering Lists

The function `filter` selects all elements in a list satisfying some property.

```
filter :: (a → Bool) → [a] → [a]
filter p [] = []
filter p (x:xs)
  | p x = x : filter p xs
  | otherwise = filter p xs
```

3.5 Laws of `filter`

```
filter p ∘ filter q = filter r
                        where r x = p x && q x
filter p ∘ map f = map f ∘ filter (p ∘ f)
```

3.6 Concatenating Lists

The function `concat` takes a list of lists and concatenates them all into a single list.

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

3.7 Laws of `concat`

```
concat (xss ++ yss) = concat xss ++ concat yss
map f ∘ concat = concat ∘ map (map f)
filter p ∘ concat = concat ∘ map (filter p)
```

3.8 Zipping and Unzipping Lists

The function `zip` takes a pair of lists and returns a list of pairs, the function `zipWith` takes a function, a pair of lists and returns a list with the function applied to each pair of elements. Similarly, `unzip` takes a list of pairs and returns a pair of lists.

```
zip :: [a] → [b] → [(a,b)]
zip [] ys = []
zip xs [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys

zipWith :: (a → b → c) → [a] → [b] → [c]
zipWith f [] ys = []
zipWith f xs [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys

unzip : : [(a,b)] → ([a], [b])
unzip [] = ([], [])
unzip ((x,y):ps) = (x:l1,y:l2)
  where (l1,l2) = unzip ps
```

3.9 Laws of `zip` and `zipWith`

```
zip = zipWith (,)
zipWith f = map (uncurry f) ∘ zip
              where uncurry f (x,y) = f x y
```

3.10 `take` and `drop`

The functions `take` and `drop` return the first and all but the first n elements of a list respectively.

```
take :: Int → [a] → [a]
take 0 xs = []
take n [] = []
take n (x:xs) = x : take (n-1) xs

drop :: Int → [a] → [a]
drop 0 xs = xs
drop n [] = []
drop n (x:xs) = drop (n-1) xs

take n xs ++ drop n xs = xs
```

3.11 `takeWhile` and `dropWhile`

Similar to `take` and `drop`, these functions continue taking or dropping elements until a predicate does not hold.

```
takeWhile :: (a → Bool) → [a] → [a]
takeWhile p [] = []
takeWhile p (x:xs)
  | p x = x : takeWhile p xs
  | otherwise = []

dropWhile :: (a → Bool) → [a] → [a]
dropWhile p [] = []
dropWhile p (x:xs)
  | p x = dropWhile p xs
  | otherwise = (x:xs)
```

3.12 Folds over Lists

Many functions can be defined as

```
f [] = e
f (x:xs) = x ⊗ f xs
```

which, when expanded, gives $f x = x_0 \otimes (x_1 \otimes (\dots (x_n \otimes e)))$. The functions `foldr` and `foldr1` use this pattern, where `foldr1` is undefined for empty lists.

```
foldr :: (a → b → b) → b → [a] → b
foldr f e [] = e
foldr f e (x:xs) = f x (foldr f e xs)

foldr1 :: (a → a → a) → [a] → a
foldr1 f [x] = x
foldr1 f (x:xs) = f x (foldr1 f xs)

fact :: Integer → Integer
fact n = foldr (∗) 1 [1..n]
```

The conversion of a list of digits to decimal does not fit the above pattern

```
decimal [x0,x1,x2] = x0 ∗ 10^2 + x1 ∗ 10^1 + x2 ∗ 10^0
decimal [x0,x1,x2] = ((0 ⊗ x0) ⊗ x ) $λotimes$ x2where
    n $λotimes$ x = 10*n + x
```

The functions `foldl` and `foldl1` capture this pattern.

```
foldl :: (b → a → b) → b → [a] → b
foldl f e [] = e
foldl f e (x:xs) = foldl f (f e x) xs

foldl1 :: (a → a → a) → [a] → a
foldl1 f (x:xs) = foldl f x xs
```

3.13 Laws for Folds

```
map f = foldr g [] where g x xs = f x : xs
filter p = foldr g []
             where g x xs = if p then x :xs else xs
concat = foldr (++) []
reverse = foldr snoc [] where snoc x xs = xs ++ [x]
```

```
sum = foldr (+) 0 = foldl (+) 0
product = foldr (*) 1 = foldl (*) 1
and = foldr (&&) True
or = foldr (||) False
maximum = foldr1 max
minimum = foldr1 mi
```

3.14 Scanning over Lists

The function `scanl` applies `foldl` to each of the initial segments of a list.

```
inits : : [a] → [[a]]
inits [ ] = [ [ ]]
inits (x:xs) = [] : map (x:) (inits xs)

scanl : : (b → a → b) → b → [a] → [b]
scanl f e [ ] = [e]
scanl f e (x:xs) = e : scanl f (f e x) xs
```

and similarly for `scanr`

```
tails : : [a] → [[a]]
tails [ ] = [[]]
tails (x:xs) = (x:xs) : tails xs

scanr : : (a → b → b) → b → [a] → [b]
scanr f e [ ] = [e]
scanr f e (x:xs)
     = f x (head y) : y where y = scanr f e xs
```

4 Induction

4.1 Induction over the Natural Numbers

As all natural numbers are either 0 or of the form $n + 1$, to show that $\forall n \in \mathbb{N} \cdot P(n)$, $P(0)$ must be shown to hold, as well as $P(n + 1)$ assuming that $P(n)$ holds.

4.2 Induction over Lists

Similarly to the natural numbers, all finite lists are either empty, or of the form `(x:xs)` for some `x` and some finite list `xs`. To show that some $P(\texttt{xs})$ holds for all lists, it must be shown that $P(\texttt{[]})$ holds, as well as $P(\texttt{(x:xs)})$ assuming $P(\texttt{xs})$.

4.3 Induction over Data Structures

To prove that a property P is satisfied for all finite elements of a datatype, it must be shown to hold for all atomic values, and all recursive values, assuming that P holds for the immediate predecessors.

5 Infinite Lists and their applications

Infinite lists are defined as either `[x(,y)..]`, where x is the first element and y is optionally given as the second element to indicate the interval between each element (defaults to 1 when y is not specified). All the usual functions can be applied to infinite lists, and will often result in infinite lists as outputs. Some functions, such as folds will not terminate. Application of functions such as `take` will result in finite lists, making lazy evaluation particularly important.

An infinite list can be thought of as the limit of a sequence of partial lists. For example, the list `[0..]` is the limit of the sequence $\bot, 0 :\bot, 0 : 1 :\bot, \ldots$. This property allows for induction over infinite lists. To show that P holds for an infinite list, P must be shown to be chain complete: if P holds for all partial lists $\texttt{xs}_i$ tending towards `xs`, then $P(\texttt{xs})$ holds.

6 Partial Lists

By definition, a finite list has a specified number of elements, whereas an infinite list will have infinitely many elements. Partial lists are those that are not ended by the empty list, but by $\bot$. For example $0 : 1 : 2 :\bot \neq [0, 1, 2, \bot]$. The latter is a finite list containing $\bot$. In order to extend induction over lists to include all partial lists, for some given P, it must also be shown that $P(\bot)$ holds.

7 Non-linear Data structures

7.1 Tip-Labelled Binary Trees

These are binary trees, only containing data at the leaf nodes, and are represented in Haskell as

```
data Tree a = Leaf a | Fork (Tree a) (Tree a)

foldTree : : (b → b → b) → (a → b) → Tree a → b
foldTree f g (Leaf x) = g x
foldTree f g (Fork
     t1 t2) = f (foldTree f g t1) (foldTree f g t2)

flatten : : Tree a → [a]
flatten (Leaf x) = [x]
flatten (Fork t1 t2) = flatten t1 ++ flatten t2
flatten = foldTree (++) (λx → [x])

size : : Tree a → Int
size (Leaf x) = 1
size (Fork t1 t2) = size t1 + size t2
size = foldTree (+) (const 1)

const x y = x

depth : : Tree a → Int
depth (Leaf x) = 0
depth (Fork t1 t2) = 1 + (depth t1 max depth t2)
depth = foldTree (λx y → 1 + (x max y)) (const 0)
```

Note that `f` is applied to the sub-trees and `g` is applied to the leaves.

7.2 Node-Labelled Binary Trees

These are binary trees, containing data both at the leaf nodes, and at the root nodes, and are represented in Haskell as

```
data NTree a = Empty | Node (NTree a) a (NTree a)

foldNTree :: (b → a → b → b) → b → NTree a → b
foldNTree f e Empty = e
foldNTree f e (Node t1
     x t2) = f (foldNTree f e t1) x (foldBTree f e t2)
```

7.3 Binary Search Trees

These are node-labelled binary trees, also satisfying the property that the value on each node is greater than all the values in its left subtree and less than all the values in its right subtree, and are represented in Haskell by:

```
data Ord
     a ⇒ STree a = Empty | Node (NTree a) a (NTree a)

isIn : : Ord a ⇒ a → STree a → Bool
isIn x Empty = False
isIn x (Node t1 n t2)
  | x == n = True
  | x < n = isIn x t1
  | x > n = isIn x t2

insert : : Ord a ⇒ a → STree a → STree a
insert x Empty = Node Empty x Empty
insert x (Node t1 n t2)
  | x == n = Node t1 n t2
  | x < n  = Node (insert x t1) n t2
  | x > n  = Node t1 n (insert x t2)
```