Chapter 15

Functional Programming

Topics

- Numbers
  - Natural numbers
  - Haskell numbers
- Lists
  - List notation
  - Lists as a data type
  - List operations

Numbers

- Haskell provides a sophisticated hierarchy of type classes for describing various kinds of numbers.
- Although (some) numbers are provided as primitives data types, it is theoretically possible to introduce them through suitable data type declarations.

Natural Numbers

- The natural numbers are the numbers 0, 1, 2, and so on, used for counting.
- Introduced by the declaration

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

  - The constructor `Succ` (short for 'successor') has type `Nat → Nat`.
  - Example: as an element of `Nat` the number `7` would be represented by

```
Succ(Succ(Succ(Succ(Succ(Succ Zero)))))
```


Natural Numbers

- Every natural number is represented by a unique value of `Nat`.
- On the other hand, not every value of `Nat` represents a well-defined natural number.
  - Example: `⊥, Succ ⊥, Succ(Succ ⊥)`
- Addition ca be defined by

```
(+)        :: Nat → Nat → Nat
m + Zero   = m
m + Succ n = Succ(m + n)
```


Natural Numbers

- Multiplication ca be defined by

```
(x)        :: Nat → Nat → Nat
m x Zero   = Zero
m x Succ n = (m x n) + m
```

- `Nat` can be a member of the type class `Eq`

```
instance Eq Nat where
  Zero == Zero      = True
  Zero == Succ n    = False
  Succ m == Zero    = False
  Succ m == Succ n  = (m == n)
```

Natural Numbers

- `Nat` can be a member of the type class `Ord`

```
instance Ord Nat where
  Zero < Zero       = False
  Zero < Succ n     = True
  Succ m < Zero     = False
  Succ m < Succ n   = (m < n)
```

- Elements of `Nat` can be printed by

```
showNat                :: Nat → String
showNatZero            = "Zero"
showNat (Succ Zero)    = "Succ Zero"
showNat (Succ(Succ n)) = "Succ (" ++
                  showNat (Succ n) ++ ")"
```

Haskell Numbers

- Haskell provide, as primitives, the following types:
  - `Int` single-precision integers
  - `Integer` arbitrary-precision integers
  - `Float` singe-precision floating-point numbers
  - `Double` double-precision gloating-point numbers
  - `Rational` rational number

The Numeric Type Classes

- The same symbols, +, x, and so on, are used for arithmetic on each numeric type.
  - Overloaded functions.
- All Haskell number types are instances of the type class `Num` defined by

```
class (Eq α, Show α) ⇒ Num α where
  (+), (-), (x)     :: α → α → α
  negate            :: α → α
  fromInteger       :: Integer → α
  …
  x – y             = x + negate y
```


Integral Types

- The members of the *Integral* type are two primitive types `Int` and `Integer`.
- The operators `div` and `mod` are provided as primitive.
  - If `x` and `y` are integers, and `y` is not zero, then `x div y` = ⌊`x / y`⌋.
    - ⌊13.8⌋ = 13, ⌊-13.8⌋ = -14
  - The value `x mod y` is defined by the equation `x = (x div y) * y + (x mod y)`

Lists

- Lists can be used to fetch and carry data from one function to another.
- Lists can be taken apart, rearranged, and combined with other lists.
- Lists can be summed and multiplied.
- Lists of characters can be read and printed.
- …

List Notation

- A finite list is denoted using square brackets and commas.
  - `[1,2,3]`
  - `["hello","goodbye"]`
- All the elements of a list must have the same type.
- The empty list is written as `[]`.
- A singleton list contains only one element
  - `[x]`
  - `[[]]` the empty list is its only member

List Notation

- If the elements of a list all have type α, then the list itself will be assigned the type `[α]`.
  - `[1,2,3]  :: [Int]`
  - `['h','e','l','l','o'] :: [Char]`
  - `[[1,2],[3]]  :: [[Int]]`
  - `[(+),(x)]  :: [Int → Int → Int]`
- A list may contain the same value more than once.
- Two lists are equal if and only if they contain the same value in the same order.

Lists as a data type

- A list can be constructed fro scratch by starting with an empty list and successively adding elements one by one.
  - Elements can be added to the front of the list, or the rear, or to somewhere in the middle.
- Data type declaration (list):

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

  - The constructor Cons (short for 'construct') add an element to the front of the list.
    - `[1,2,3]  Cons 1 (Cons 2 (Cons 3 Nil))`

Lists as a data type

- In functional programming, lists are defined as elements of `List α`.
  - The syntax `[α]` is used instead of `List α`.
  - The constructor `Nil` is written as `[]`
  - The constructor `Cons` is written as an infix operator `(:)`
    - `(:)` associates to the right
    - `[1,2,3] = 1:(2:(3:[])) = 1:2:3:[]`

Lists as a data type

- Like functions over data types, functions over lists can be defined by pattern matching.

```
instance (Eq α) ⇒ Eq [α] where
  [] == []        = True
  [] == (y:ys)    = False
  (x:xs) == []    = False
  (x:xs) == (y:ys)= (x == y) ∧ (xs == ys)
```


List Operations

- Some of the most commonly used functions and operations on lists.
- For each function: give the definition, illustrate its use, and state some of its properties.

Concatenation

- Two lists, both of the same type, can be concatenated to form one longer list.
- This function is denoted by the binary operator `++`.

```
? [1,2,3] ++ [4,5]
[1,2,3,4,5]
? [1,2] ++ [] ++ [1]
[1,2,1]
```

Concatenation

- The formal definition of ++ is

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

  - The definition of ++ is by pattern matching on the left-hand argument.
  - The two patterns are disjoint and cover all cases, apart from the undefined list ⊥.
  - It follows by case exhaustion that

```
⊥ ++ ys = ⊥
```


Concatenation

  - It is not the case that `xs ++ ⊥ = ⊥`

```
? [1,2,3] ++ undefined
[1,2,3{Interrupted!}
```

  - The list `[1,2,3] ++ ⊥` is a *partial list*; in full form it is the list `1:2:3:⊥`.
    - The evaluator can compute the first three elements, but thereafter it goes into a nonterminating computation, so we interrupt it.
- Some properties:

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
xs ++ [] = [] ++ xs = xs
```


Reverse

- This function reverses the order of elements in a finite list.

```
? reverse [1,2,3,4,5]
[5,4,3,2,1]
```

- The definition is

```
reverse          :: [α] → [α]
reverse []       =[]
reverse (x:xs)   = reverse xs ++ [x]
```

- In words, to reverse a list (x:xs) one reverses xs and then adds x to the end.

Length

- The length of a list is the number of elements it contains.
- The definition is

```
length         :: [α] → Int
length []      = 0
length (x:xs)  = 1 + length(xs)
```

- The nature of the list elements is irrelevant when computing the length:

```
? length [undefined,undefined]
2
```


Length

- Not every list has a well-defined length.
  - The partial lists have an undefined length
    - `⊥, x:⊥, x:y:⊥`
  - Only finite lists have well-defined lengths.
    - The list `[⊥,⊥]` is a finite list, not a partial list because it is the list `⊥:⊥:[]`, which ends in `[]` not ⊥. The computer cannot produce the elements, but it can produce the length of the list.
- The function length satisfies a distribution property:

```
length(xs ++ ys) = length xs + length ys
```


Head and Tail

- The function *head* selects the first element of a nonempty list, and *tail* selects the rest:

```
head           :: [α] → α
head []        = error "empty list"
head (x:xs)    = x
tail           :: [α] → [α]
tail []        = error "empty list"
tail (x:xs)    = xs
```

  - These are constant-time operations, since they deliver their result in one reduction step.

Init and last

- The function *last* and *init* select the last element of a nonempty list and what remains after the last element has been removed.

```
? last [1,2,3,4,5]
5
? init [1,2,3,4,5]
[1,2,3,4]
```


Init and last

- First attempt (definition):

```
last      :: [α] → α
last      = head · reverse

init      :: [α] → α
init      = reverse · tail · reverse
```

- Problem?
  - `init xs` = ⊥ for all partial and infinite lists `xs`

Init and last

- Second attempt (definition):

```
last (x:xs) = if null xs then x else last xs

init (x:xs) = if null xs then [] else x:init xs
```

- With this definition
  - `init xs = xs` for all partial and infinite lists `xs`

Init and last

- Third attempt (definition):
  - Since `[x]` is an abbreviation for `x:[]`

```
last [x]        = x
last (x:xs)     = last xs
init [x]        = []
init (x:xs)     = x:init xs
```

- Problem?
  - There is a serious danger of confusion because the patterns `[x]` and `(x:xs)` are not *disjoint*.
    - The second includes the first as a special case.

Init and last

- If the order of the equations are reversed:

```
last' (x:xs)   = last' xs
last' [x]      = x
```

- The definition of `last'` would simply be incorrect.
  - `last' xs` = ⊥
- It is not a good practice to write definition that depend critically on the order of the equations.

Init and last

- Definition

```
last           :: [α] → α
last []        = error "empty list"
last [x]       = x
last (x:y:ys)  = last(y:ys)

init           :: [α] → [α]
init []        = error "empty list"
init [x]       = []
init [x:y:xs)  = x:init(y:xs)
```