Imports

import *Data.List* (*sort*)
import *Data.Char* (*toLower*)

Introduction to Functional Programming

Carlos Encarnación
encarnacion.carlos@gmail.com

November 7, 2014

Lecture 2: Numbers and lists

Built-in number types

Haskell has the following number types

Int	limited-precision integers
Integer	arbitrary-precision integers
Float	single-precision floating-point numbers
Double	double-precision floating-point numbers
Complex	complex numbers

Declaration

The prelude define the type class *Num* as follows

$$
\begin{array}{l}
\textbf{class } (Eq\ a, Show\ a) \Rightarrow Num\ a \textbf{ where} \\
\quad \begin{array}{ll}
(+), (-), (*) & :: a \rightarrow a \rightarrow a \\
negate & :: a \rightarrow a \\
abs, signum & :: a \rightarrow a \\
fromInteger & :: Integer \rightarrow a
\end{array}
\end{array}
$$

Notice that according to the declaration of *Num* we have that

1. Any two numbers can be compared for equality.
2. Any number can be printed.
3. Any number can be added to, substracted from or multiplied by another number.
4. Any number can be negated.

The type class Real

$$\textbf{class}\ (Num\ a, Ord\ a) \Rightarrow Real\ a\ \textbf{where}$$
$$\quad toRational :: a \to Rational$$

The type class Fractional

$$\begin{array}{l} \textbf{class}\ (\mathit{Num}\ a) \Rightarrow \mathit{Fractional}\ a\ \textbf{where} \\ \quad (/) :: a \to a \to a \\ \quad \mathit{fromRational} :: \mathit{Rational} \to a \end{array}$$

The type class integral

$$
\begin{array}{l}
\textbf{class}\ (\mathit{Real}\ a, \mathit{Enum}\ a) \Rightarrow \mathit{Integral}\ a\ \textbf{where} \\
\quad \mathit{divMod} :: a \to a \to (a, a) \\
\quad \mathit{toInteger} :: a \to \mathit{Integer}
\end{array}
$$

Problem

Write a function *floor* that given a *Float* number x compute the largest integer n not exceding x, or

$$n \leqslant x < n + 1$$

Hence, the type of *floor* is as follows

$$\mathit{floor} :: \mathit{Float} \rightarrow \mathit{Integer}$$

For example

$$\mathit{floor}\ 3.14 = 3$$

and

$$\mathit{floor}\ (-3.14) = -4$$

Loop

Consider the following function

$$
\begin{aligned}
&\mathit{until} && :: (a \rightarrow \mathit{Bool}) \rightarrow (a \rightarrow a) \rightarrow a \rightarrow a\\
&\mathit{until}\ p\ f\ x \mid p\ x && = x\\
&\qquad\qquad\ \ \mid \mathit{otherwise} && = \mathit{until}\ p\ f\ (f\ x)
\end{aligned}
$$

The function *until* returns the first element of the list

$$[x, f\ x, f\ (f\ x), \ldots]$$

such that $p\ y = \mathit{True}$

Solution 1

$$
\begin{aligned}
&\mathit{floor}' &&:: \mathit{Float} \to \mathit{Integer}\\
&\mathit{floor}'\ x \mid x \geqslant 0 &&= \mathit{until}\ (x\text{`}\mathit{lt}\text{`})\ (+1)\ 0 - 1\\
&\qquad\quad \mid \mathit{otherwise} &&= \mathit{until}\ (\text{`}\mathit{leq}\text{`}x)\ (\text{`}\mathit{subtract}\text{`}1)\ 0\\
&\quad \textbf{where}\ \mathit{subtract}\ x\ y = x - y
\end{aligned}
$$

$$
\begin{aligned}
\mathit{lt}\ x\ y &= x < \mathit{fromInteger}\ y\\
\mathit{leq}\ x\ y &= \mathit{fromInteger}\ x \leqslant y
\end{aligned}
$$

Solution 2

$$
\begin{aligned}
&\mathit{floor}'' \quad :: \mathit{Float} \to \mathit{Integer} \\
&\mathit{floor}''\ x = \mathit{fst}\ (\mathit{until}\ \mathit{unit}\ (\mathit{shrink}\ x)\ (\mathit{bound}\ x)) \\
&\quad \textbf{where}\ \mathit{unit}\ (m, n) = (m + 1) \equiv n
\end{aligned}
$$

Solution 2: defining *shrink*

$$\textbf{type}\ \mathit{Interval} = (\mathit{Integer}, \mathit{Integer})$$

$$\begin{aligned}
&\mathit{shrink} && :: \mathit{Float} \rightarrow \mathit{Interval} \rightarrow \mathit{Interval}\\
&\mathit{shrink}\ x\ (m, n) && = \textbf{if}\ p\ \text{`}\mathit{leq}\text{`}\ x\ \textbf{then}\ (p, n)\ \textbf{else}\ (m, p)\\
&\quad \textbf{where}\ p = \mathit{choose}\ (m, n)
\end{aligned}$$

Solution 2: defining *choose*

$$\begin{aligned}
&\mathit{choose} && :: \mathit{Interval} \rightarrow \mathit{Integer}\\
&\mathit{choose}\ (m, n) && = (m + n)\ \text{`}\mathit{div}\text{`}\ 2
\end{aligned}$$

Solution 2: Proof

Proof that $(m + n)$ ‘div‘ 2 implies $m + 1 < n$

$$
\begin{array}{ll}
& m < (m + n) \text{ `} div \text{` } 2 < n \\
= & \quad \{ \text{ ordering on integers } \} \\
& m + 1 \leqslant (m + n) \text{ `} div \text{` } 2 < n \\
= & \quad \{ \ (m + n) \text{ `} div \text{` } 2 = \lfloor (m + n) \, / \, 2 \rfloor \ \} \\
& m + 1 \leqslant (m + n) \, / \, 2 < n \\
= & \quad \{ \text{ algebra } \} \\
& m + 2 < n \wedge m < n \\
= & \quad \{ \text{ algebra } \} \\
& m + 1 < n
\end{array}
$$

Solution 2: defining *bound*

$$\begin{aligned} &\mathit{bound} \quad :: \mathit{Float} \rightarrow \mathit{Interval} \\ &\mathit{bound}\ x = (\mathit{lower}\ x, \mathit{upper}\ x) \end{aligned}$$

$$\begin{aligned} &\mathit{lower} \quad :: \mathit{Float} \rightarrow \mathit{Integer} \\ &\mathit{lower}\ x = \mathit{until}\ (\text{`}\mathit{leq}\text{`}x)\ (2*)\ (-1) \end{aligned}$$

$$\begin{aligned} &\mathit{upper} \quad :: \mathit{Float} \rightarrow \mathit{Integer} \\ &\mathit{upper}\ x = \mathit{until}\ (x\text{`}\mathit{lt}\text{`})\ (2*)\ 1 \end{aligned}$$

Declaration

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

Nat could be an instance of *Eq*

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Eq}\ \mathit{Nat}\ \textbf{where} \\
\quad \mathit{Zero} \equiv \mathit{Zero} = \mathit{True} \\
\quad \mathit{Zero} \equiv \mathit{Succ}\ \_ = \mathit{False} \\
\quad \mathit{Succ}\ \_ \equiv \mathit{Zero} = \mathit{False} \\
\quad \mathit{Succ}\ m \equiv \mathit{Succ}\ n = m \equiv n
\end{array}
$$

Nat could be an instance of *Show*

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Show}\ \mathit{Nat}\ \textbf{where}\\
&\quad \mathit{show}\ \mathit{Zero} = \texttt{"Zero"}\\
&\quad \mathit{show}\ (\mathit{Succ}\ \mathit{Zero}) = \texttt{"Succ Zero"}\\
&\quad \mathit{show}\ (\mathit{Succ}\ (\mathit{Succ}\ n)) = \texttt{"Succ ("} +\!\!+\ \mathit{show}\ (\mathit{Succ}\ n) +\!\!+ \texttt{")"}
\end{aligned}
$$

Deriving instances

$$\textbf{data}\ \mathit{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \mathit{Nat}\ \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord}, \mathit{Show})$$

Nat could be an instance of *Num*

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Num}\ \mathit{Nat}\ \textbf{where} \\
\quad \begin{array}{ll}
m + \mathit{Zero} & = m \\
m + \mathit{Succ}\ n & = \mathit{Succ}\ (m + n) \\
m * \mathit{Zero} & = \mathit{Zero} \\
m * (\mathit{Succ}\ n) & = m * n + m \\
\mathit{abs}\ n & = n \\
\mathit{signum}\ \mathit{Zero} & = \mathit{Zero} \\
\mathit{signum}\ (\mathit{Succ}\ n) & = \mathit{Succ}\ \mathit{Zero} \\
m - \mathit{Zero} & = m \\
\mathit{Zero} - \mathit{Succ}\ n & = \mathit{Zero} \\
\mathit{Succ}\ m - \mathit{Succ}\ n & = m - n
\end{array} \\
\quad \mathit{fromInteger}\ x \begin{array}{|ll} x \leqslant 0 & = \mathit{Zero} \\ \mathit{otherwise} & = \mathit{Succ}\ (\mathit{fromInteger}\ (x - 1)) \end{array}
\end{array}
$$

General remarks

1. The type $[a]$ denote a list of elements of type a.
2. The empty list is denoted by $[\,]$.

As examples

The operator cons

List notation, such as [1,2,3], is in fact an abbreviation for a more basic form

$$1 : 2 : 3 : [\,]$$

The operator $(:) :: a \rightarrow [a] \rightarrow [a]$, pronounced *cons*, is a constructor for lists. Notice that since it associates to the right parentheses are not required.

Lists as a recursive data type

Lists can be introduced as a Haskell recursive data type as follows

$$\textbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$

where we have the following equivalences

$$\begin{aligned} \mathit{Nil} &\equiv [\,] \\ \mathit{List}\ a &\equiv [a] \end{aligned}$$

Kinds of list

Every list of $[a]$ takes one of three forms

1. The undefined list $\perp : [a] :: [a]$.
2. The empty list $[\,] :: [a]$.
3. A list of the form $x : xs$ where $x :: a$ and $xs :: [a]$.

As a result there are three kinds of list

1. A finite list, which is built from (:) and [ ].
2. A partial list, which is built from (:) and undefined.
3. An infinite list, which is built from (:) alone.

Infinite lists

Consider the following function

$$\begin{aligned} &\mathit{iterate} &&:: (a \to a) \to a \to [a] \\ &\mathit{iterate}\ f\ x &&= x : \mathit{iterate}\ f\ (f\ x) \end{aligned}$$

$$\begin{aligned} &\mathit{perfect1} = \mathit{head}\ (\mathit{filter}\ \mathit{perfect}\ [1\mathinner{..}]) \\ &\quad \textbf{where}\ \mathit{perfect} = n \equiv \mathit{sum}\ (\mathit{divisors}\ n) \end{aligned}$$

$$\mathit{until}\ p\ f\ = \mathit{head} \circ \mathit{filter}\ p \circ \mathit{iterate}\ f$$

Notation

When m and n are integers we can write

1. $[m\,..\,n]$ for the list $[m, m+1, \ldots, n]$
2. $[m\,..]$ for the infinite list $[m, m+1, m+2, \ldots]$
3. $[m, n\,..\,p]$ for the list $[m, m+(n-m), m+2\,(n-m), \ldots, p]$
4. $[m, n\,..]$ for the infinite list $[m, m+(n-m), m+2\,(n-m), \ldots]$

Numbers are not special, also we can write

$$[\texttt{'a'}\,..\,\texttt{'z'}] = \texttt{"abcdefghijklmnopqrstuvwxyz"}$$

Indeed, any instance of the type class *Enum* can use the notation.

Notation

A list comprehension has the form

$$[e \mid q_1, ..., q_n]$$

where the q_i qualifiers are either

- *generators* of the form $p \leftarrow e$ where p is a pattern of type t and e is an expression of type $[t]$
- *local bindings* let expressions
- *boolean guards*, which are arbitrary expressions of type *Bool*

Examples

$$\mathit{divisors}\ n = [x \mid x \leftarrow [2 \mathinner{..} n], n\ \text{`}\mathit{mod}\text{`}\ x \equiv 0]$$

map, *filter* and *concat* in terms of list comprehension

$$map\ f\ xs = [f\ x \mid x \leftarrow xs]$$

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

$$concat\ xss = [x \mid xs \leftarrow xss, x \leftarrow xs]$$

Recursive definition of the function *null*

$$
\begin{aligned}
&\mathit{null} && :: [a] \to \mathit{Bool} \\
&\mathit{null}\ [\,] && = \mathit{True} \\
&\mathit{null}\ (x : xs) && = \mathit{False}
\end{aligned}
$$

Recursive definition of the function *head*

$$
\begin{aligned}
&head && :: [a] \to a \\
&head\ (x{:}\ \_) && = x
\end{aligned}
$$

Recursive definition the function *tail*

$$\begin{aligned} &tail && :: [a] \to [a] \\ &tail\ (\_ : xs) && = xs \end{aligned}$$

Recursive definition the function *last*

$$
\begin{aligned}
&last && :: [a] \to a \\
&last\ (x : [\,]) && = x \\
&last\ (\_ : xs) && = last\ xs
\end{aligned}
$$

Definition

$$
\begin{aligned}
(+\!\!+) &:: [a] \to [a] \\
(+\!\!+)\ [\,]\ ys &= ys \\
(+\!\!+)\ (x : xs)\ ys &= x : (xs +\!\!+ ys)
\end{aligned}
$$

Understanding concatenation

Associative

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

concat

$$
\begin{aligned}
&\mathit{concat} &&:: [[a]] \to [a] \\
&\mathit{concat}\ [\,] &&= [\,] \\
&\mathit{concat}\ (\mathit{xs} : \mathit{xss}) &&= \mathit{xs} +\!\!+\ \mathit{concat}\ \mathit{xss}
\end{aligned}
$$

map

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

filter

$$
\begin{aligned}
&\mathit{filter} && :: (a \to \mathit{Bool}) \to [a] \to [a] \\
&\mathit{filter}\ p\ [\,] && = [\,] \\
&\mathit{filter}\ p\ (x : xs) \mid p\ x && = x : \mathit{filter}\ xs \\
&\phantom{\mathit{filter}\ p\ (x : xs)} \mid \mathit{otherwise} && = \mathit{filter}\ xs
\end{aligned}
$$

filter in terms of *concat* and *map*

$$
\begin{aligned}
&\mathit{filter} \quad :: (a \rightarrow \mathit{Bool}) \rightarrow [a] \rightarrow [a] \\
&\mathit{filter}\ p = \mathit{concat} \circ \mathit{map}\ \mathit{test} \\
&\quad \textbf{where}\ \mathit{test}\ x = \textbf{if}\ p\ \textbf{then}\ [x]\ \textbf{else}\ [\,]
\end{aligned}
$$

Functor property of *map*

$$map\ id = id$$
$$map\ (g \circ f) = map\ g \circ map\ f$$

Functor type class

class *Functor f* **where**
  $fmap :: (a \to b) \to f\ a \to f\ b$

Tree is an instance of the class functor

$$
\begin{array}{l}
\textbf{data}\ \mathit{Tree}\ a = \mathit{Tip}\ a \mid \mathit{Fork}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a) \\
\quad \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord}, \mathit{Show})
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Functor}\ \mathit{Tree}\ \textbf{where} \\
\quad \mathit{fmap}\ g\ (\mathit{Tip}\ x) \qquad = \mathit{Tip}\ (g\ x) \\
\quad \mathit{fmap}\ g\ (\mathit{Fork}\ u\ v) = \mathit{Fork}\ (\mathit{fmap}\ g\ u)\ (\mathit{fmap}\ g\ v)
\end{array}
$$

map

$$
\begin{aligned}
f \circ head &= head \circ map\ f \\
map\ f \circ tail &= tail \circ map\ f \\
map\ f \circ concat &= concat \circ map\ (map\ f)
\end{aligned}
$$

filter

$$\mathit{filter}\ p \circ \mathit{map}\ f = \mathit{map}\ f \circ \mathit{filter}\ (p \circ f)$$

zip

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

zipWith

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

zip in terms of *zipWith*

$$zip = zipWith\ (,)$$

nondec

$$\begin{aligned}
&nondec && :: (Ord\ a) \Rightarrow [a] \rightarrow Bool \\
&nondec\ [\,] && = True \\
&nondec\ [x] && = True \\
&nondec\ (x : y : xs) && = (x \leqslant y) \wedge nondec\ (y : xs)
\end{aligned}$$

$$\begin{aligned}
&nondec' && :: (Ord\ a) \Rightarrow [a] \rightarrow Bool \\
&nondec'\ xs && = and\ (zipWith\ (\leqslant)\ xs\ (tail\ xs))
\end{aligned}$$

$$\begin{aligned}
&and && :: [Bool] \rightarrow Bool \\
&and\ [\,] && = True \\
&and\ (x : xs) && = x \wedge (and\ xs)
\end{aligned}$$

position

$$
\begin{aligned}
&\mathit{position} && :: (\mathit{Eq}\ a) \Rightarrow a \rightarrow [a] \rightarrow \mathit{Int} \\
&\mathit{position}\ x\ \mathit{xs} && = \mathit{head}\ ([j \mid (j, y) \leftarrow \mathit{zip}\ [0\,..]\ \mathit{xs}, x \equiv y] \mathbin{+\!\!+} [-1])
\end{aligned}
$$

Common words

$$\mathbf{type}\ \mathit{Text} = [\mathit{Char}]$$
$$\mathbf{type}\ \mathit{Word} = [\mathit{Char}]$$

$$\begin{aligned}&\mathit{commonWords} :: \mathit{Int} \to \mathit{Text} \to \mathit{Text}\\ &\mathit{commonWords}\ n = \mathit{concat} \circ \mathit{map}\ \mathit{showRun} \circ \mathit{take}\ n \circ\\ &\quad \mathit{sortRuns} \circ \mathit{countRuns} \circ \mathit{sortWords} \circ \mathit{words} \circ \mathit{map}\ \mathit{toLower}\end{aligned}$$

Which functions we have to define?
showRun, *sortRuns*, *countRuns*, *sortWords*

Defining *showRun*

$$
\begin{aligned}
&\mathit{showRun} && :: (\mathit{Int}, \mathit{Word}) \to [\mathit{Char}] \\
&\mathit{showRun}\ (n, w) && = w \mathbin{+\!\!+} \texttt{": "} \mathbin{+\!\!+} \mathit{show}\ n \mathbin{+\!\!+} \texttt{"\textbackslash n"}
\end{aligned}
$$

Defining *countRuns*

The prelude define the function *span* with the following type

$$span :: (a \to Bool) \to [a] \to ([a],[a])$$

hence we can define

$$\begin{aligned}
&countRuns && :: [Word] \to [(Int, Word)] \\
&countRuns\ [\,] && = [\,] \\
&countRuns\ (w : ws) && = (1 + length\ us, w) : countRuns\ vs \\
&\quad \mathbf{where}\ (us, vs) && = span\ (\equiv w)\ ws
\end{aligned}$$

Defining *sortWords*

The module *Data.List* define the function *sort* with the following type

$$sort :: (Ord\ a) \Rightarrow [a] \rightarrow [a]$$

thus we can define

$$sortWords :: [Word] \rightarrow [Word]$$
$$sortWords = sort$$

Defining *sortRuns*

How can we define *sortRuns*?

First, recall the type

$$sortRuns :: [(Int, Word)] \to [(Int, Word)]$$

$$\begin{aligned} sortRuns &:: [(Int, Word)] \to [(Int, Word)] \\ sortRuns &= reverse \circ sort \end{aligned}$$

The end

Yes, we've finished!