Programming Languages: Functional Programming
2. Introduction to Haskell: Simple Datatypes & Functions on Lists

Shin-Cheng Mu

Autumn 2023

1 Simple Datatypes

1.1 Booleans

Booleans

The datatype $Bool$ can be introduced with a *datatype declaration*:

$$\mathbf{data}\ Bool\ =\ False \mid True$$

(But you need not do so. The type $Bool$ is already defined in the Haskell Prelude.)

Datatype Declaration

- In Haskell, a **data** declaration defines a new type.

$$\begin{array}{lcl}\mathbf{data}\ Type & = & Con_1\ Type_{11}\ Type_{12} \ldots \\ & \mid & Con_2\ Type_{21}\ Type_{22} \ldots \\ & \mid & \vdots \end{array}$$

- The declaration above introduces a new type, $Type$, with several cases.
- Each case starts with a constructor, and several (zero or more) arguments (also types).
- Informally it means "a value of type $Type$ is either a Con_1 with arguments $Type_{11}$, $Type_{12}$..., or a Con_2 with arguments $Type_{21}$, $Type_{22}$..."
- Types and constructors begin in capital letters.

Functions on Booleans

Negation:

$$\begin{array}{ll} not & :: Bool \to Bool \\ not\ False & = True \\ not\ True & = False \end{array}$$

- Notice the definition by *pattern matching*. The definition has two cases, because $Bool$ is defined by two cases. The shape of the function follows the shape of its argument.

Functions on Booleans

Conjunction and disjunction:

$$\begin{array}{ll} (\wedge), (\vee) & :: Bool \to Bool \to Bool \\ False \wedge x & = False \\ True \wedge x & = x \\ False \vee x & = x \\ True \vee x & = True \end{array}$$

I use the symbols $\wedge$ and $\vee$ due to mathematical convension. In your Haskell code, $\wedge$ should be written &&, and $\vee$ should be ||.

Functions on Booleans

Equality check:

$$\begin{array}{ll} (==), (\neq) & :: Bool \to Bool \to Bool \\ x == y & = (x \wedge y) \vee (not\ x \wedge not\ y) \\ x \neq y & = not\ (x == y) \end{array}$$

- $=$ is a definition, while $==$ is a function.
- $==$ and $\neq$ are written respectively written == and /= in ASCII.

Example

$$\begin{array}{ll} leapyear & :: Int \to Bool \\ leapyear\ y & = (y\ `mod`\ 4 == 0) \wedge \\ & \quad\ (y\ `mod`\ 100 \neq 0 \vee y\ `mod`\ 400 == 0) \end{array}$$

- Note: y ‘mod‘ 100 could be written $mod\ y\ 100$. The backquotes turns an ordinary function to an infix operator.
- It's just personal preference whether to do so.

1.2 Characters

Characters

- You can think of Char as a big **data** definition:

$$\mathbf{data}\ \mathit{Char}\ =\ \text{'}a\text{'}\ |\ \text{'}b\text{'}\ |\ \ldots$$

  with functions:

$$\begin{aligned} &\mathit{ord} :: \mathit{Char} \to \mathit{Int} \\ &\mathit{chr} :: \mathit{Int} \to \mathit{Char} \end{aligned}$$

- Characters are compared by their order:

$$\begin{aligned} &\mathit{isDigit} &&:: \mathit{Char} \to \mathit{Bool} \\ &\mathit{isDigit}\ x &&= \text{'}0\text{'} \leq x \wedge x \leq \text{'}9\text{'} \end{aligned}$$

Equality Check

- Of course, you can test equality of characters too:

$$(==) :: \mathit{Char} \to \mathit{Char} \to \mathit{Bool}$$

- $(==)$ is an *overloaded* name — one name shared by many different definitions of equalities, for different types:
  - $(==) :: \mathit{Int} \to \mathit{Int} \to \mathit{Bool}$
  - $(==) :: (\mathit{Int}, \mathit{Char}) \to (\mathit{Int}, \mathit{Char}) \to \mathit{Bool}$
  - $(==) :: [\mathit{Int}] \to [\mathit{Int}] \to \mathit{Bool}$ ...
- Haskell deals with overloading by a general mechanism called *type classes*. It is considered a major feature of Haskell.
- While the type class is an interesting topic, we might not cover much of it since it is orthogonal to the central message of this course.

1.3 Products

Tuples

- The polymorphic type (a, b) is essentially the same as the following declaration:

$$\mathbf{data}\ \mathit{Pair}\ a\ b = \mathit{MkPair}\ a\ b$$

- Or, had Haskell allow us to use symbols:

$$\mathbf{data}\ (a, b) = (a, b)$$

- Two projections:

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

2 Functions on Lists

Lists in Haskell

- Traditionally an important datatype in functional languages.
- In Haskell, all elements in a list must be of the same type.
  - $[1, 2, 3, 4] :: \mathit{List}\ \mathit{Int}$
  - $[\mathit{True}, \mathit{False}, \mathit{True}] :: \mathit{List}\ \mathit{Bool}$
  - $[[1, 2], [\,], [6, 7]] :: \mathit{List}\ (\mathit{List}\ \mathit{Int})$
  - $[\,] :: \mathit{List}\ a$, the empty list (whose element type is not determined).
- String is an abbreviation for $\mathit{List}\ \mathit{Char}$; "abcd" is an abbreviation of $[\text{'}a\text{'}, \text{'}b\text{'}, \text{'}c\text{'}, \text{'}d\text{'}]$.

List as a Datatype

- $[\,] :: \mathit{List}\ a$ is the empty list whose element type is not determined.
- If a list is non-empty, the leftmost element is called its *head* and the rest its *tail*.
- The constructor $(:) :: a \to \mathit{List}\ a \to \mathit{List}\ a$ builds a list. E.g. in $x : xs$, x is the head and xs the tail of the new list.
- You can think of a list as being defined by

$$\mathbf{data}\ \mathit{List}\ a\ =\ [\,]\ |\ a : \mathit{List}\ a$$

- $[1, 2, 3]$ is an abbreviation of $1 : (2 : (3 : [\,]))$.

Head and Tail

- $\mathit{head} :: \mathit{List}\ a \to a$. e.g. $\mathit{head}\ [1, 2, 3] = 1$.
- $\mathit{tail} :: \mathit{List}\ a \to \mathit{List}\ a$. e.g. $\mathit{tail}\ [1, 2, 3] = [2, 3]$.
- $\mathit{init} :: \mathit{List}\ a \to \mathit{List}\ a$. e.g. $\mathit{init}\ [1, 2, 3] = [1, 2]$.
- $\mathit{last} :: \mathit{List}\ a \to a$. e.g. $\mathit{last}\ [1, 2, 3] = 3$.
- They are all partial functions on non-empty lists. e.g. $\mathit{head}\ [\,] = \bot$.
- $\mathit{null} :: \mathit{List}\ a \to \mathit{Bool}$ checks whether a list is empty.

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

2.1 List Generation

List Generation

- $[0..25]$ generates the list $[0, 1, 2..25]$.
- $[0, 2..25]$ yields $[0, 2, 4..24]$.
- $[2..0]$ yields $[\,]$.
- The same works for all *ordered* types. For example $Char$:
  - $[\texttt{'}a\texttt{'}..\texttt{'}z\texttt{'}]$ yields $[\texttt{'}a\texttt{'}, \texttt{'}b\texttt{'}, \texttt{'}c\texttt{'}..\texttt{'}z\texttt{'}]$.
- $[1..]$ yields the *infinite* list $[1, 2, 3..]$.

List Comprehension

- Some functional languages provide a convenient notation for list generation. It can be defined in terms of simpler functions.
- e.g. $[x \times x \mid x \leftarrow [1..5], odd\ x] = [1, 9, 25]$.
- Syntax: $[e \mid Q_1, Q_2..]$. Each Q_i is either
  - a generator $x \leftarrow xs$, where x is a (local) variable or pattern of type a while xs is an expression yielding a list of type $List\ a$, or
  - a guard, a boolean valued expression (e.g. $odd\ x$).
  - e is an expression that can involve new local variables introduced by the generators.

List Comprehension

Examples:

- $[(a, b) \mid a \leftarrow [1..3], b \leftarrow [1..2]] = [(1, 1), (1, 2), (2, 1), (2, 2), (3, 1), (3, 2)]$
- $[(a, b) \mid b \leftarrow [1..2], a \leftarrow [1..3]] = [(1, 1), (2, 1), (3, 1), (1, 2), (2, 2), (3, 2)]$
- $[(i, j) \mid i \leftarrow [1..4], j \leftarrow [i + 1..4]] = [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]$
- $[(i, j) \mid\leftarrow [1..4], even\ i, j \leftarrow [i + 1..4], odd\ j] = [(2, 3)]$

2.2 Combinators on Lists

Two Modes of Programming

- Functional programmers switch between two modes of programming.
  - Inductive/recursive mode: go into the structure of the input data and recursively process it.
  - Combinatorial mode: compose programs using existing functions (combinators), process the input in stages.
- We will try the latter style today. However, that means we have to familiarise ourselves to a large collection of library functions.
- In the next lecture we will talk about how these library functions can be defined, in the former style.

Length and Indexing

- $(!!) :: List\ a \to Int \to a$. List indexing starts from zero. e.g. $[1, 2, 3]!!0 = 1$.
- $length :: List\ a \to Int$. e.g. $length\ [0..9] = 10$.

Append and Concatenation

- Append: $(+\!\!+) :: List\ a \to List\ a \to List\ a$. In ASCII one types $(++)$.
  - $[1, 2] +\!\!+ [3, 4, 5] = [1, 2, 3, 4, 5]$
  - $[\,] +\!\!+ [3, 4, 5] = [3, 4, 5] = [3, 4, 5] +\!\!+ [\,]$
- Compare with $(:) :: a \to List\ a \to List\ a$. It is a type error to write $[\,] : [3, 4, 5]$. $(+\!\!+)$ is defined in terms of $(:)$.
- $concat :: List\ (List\ a) \to List\ a$.
  - e.g. $concat\ [[1, 2], [\,], [3, 4], [5]] = [1, 2, 3, 4, 5]$.
  - $concat$ is defined in terms of $(+\!\!+)$.

Take and Drop

- $take\ n$ takes the first n elements of the list. For a definition:

$$
\begin{array}{ll}
take & :: Int \to List\ a \to List\ a \\
take\ 0\ xs & = [\,] \\
take\ (n+1)\ [\,] & = [\,] \\
take\ (n+1)\ (x : xs) & = x : take\ n\ xs
\end{array}
$$

- For example, $take\ 0\ xs = []$
- $take\ 3$ "abcde" = "abc"
- $take\ 3$ "ab" = "ab"

- Working with infinite list: $take\ 5\ [1..] = [1,2,3,4,5]$. Thanks to normal order (lazy) evaluation.
- Dually, $drop\ n$ drops the first n elements of the list. For a definition:

$$\begin{array}{ll} drop & :: Int \to List\ a \to List\ a \\ drop\ 0\ xs & = xs \\ drop\ (n+1)\ [] & = [] \\ drop\ (n+1)\ (x:xs) & = drop\ n\ xs \end{array}$$

  - For example, $drop\ 0\ xs = xs$
  - $drop\ 3$ "abcde" = "cd"
  - $drop\ 3$ "ab" = ""

- $take\ n\ xs \mathbin{+\!\!+} drop\ n\ xs = xs$, as long as $n \neq \bot$.

Map and λ

- $map :: (a \to b) \to List\ a \to List\ b$. e.g. $map\ (1+)\ [1,2,3,4,5] = [2,3,4,5,6]$.
- $map\ square\ [1,2,3,4] = [1,4,9,16]$.
- Every once in a while you may need a small function which you do not want to give a name to. At such moments you can use the λ notation:
  - $map\ (\lambda x \to x \times x)\ [1,2,3,4] = [1,4,9,16]$
  - In ASCII λ is written \.
- λ is an important primitive notion. We will talk more about it later.

Filter

- $filter :: (a \to Bool) \to List\ a \to List\ a$.
  - e.g. $filter\ even\ [2,7,4,3] = [2,4]$
  - $filter\ (\lambda n \to n\ `mod`\ 3 == 0)\ [3,2,6,7] = [3,6]$
- Application: count the number of occurrences of $'a'$ in a list:
  - $length \cdot filter\ ('a' ==)$
  - Or $length \cdot filter\ (\lambda x \to 'a' == x)$
- **Note** a list comprehension can always be translated into a combination of primitive list generators and map, $filter$, and $concat$.

Zip

- $zip :: List\ a \to List\ b \to List\ (a,b)$
- e.g. zip "abcde" $[1,2,3] = [('a',1), ('b',2), ('c',3)]$
- The length of the resulting list is the length of the shorter input list.

Positions

- Exercise: define $positions :: Char \to String \to List\ Int$, such that $positions\ x\ xs$ returns the positions of occurrences of x in xs. E.g. $positions\ 'o'$ "roodo" $= [1,2,4]$.
- $positions\ x\ xs = map\ snd\ (filter\ ((x\ ==) \cdot fst)\ (zip\ xs\ [0..]))$
- Or, $positions\ x\ xs = map\ snd\ (filter\ (\lambda(y,i) \to x == y)\ (zip\ xs\ [0..]))$
- What if you want only the position of the *first* occurrence of x?

$$\begin{array}{ll} pos & :: Char \to String \to Int \\ pos\ x\ xs & = head\ (positions\ x\ xs) \end{array}$$

  - Due to lazy evaluation (normal order evaluation), positions of the other occurrences are *not* evaluated!
  - **Note** For now, think of "lazy evaluation" as another (more popular) name for normal order evaluation. Some people distinguish them by saying that normal order evaluation is a mathematical idea while lazy evaluation is a way to implement normal order evaluation.

Morals of the Story

- Lazy evaluation helps to improve modularity.
  - List combinators can be conveniently re-used. Only the relevant parts are computed.
- The combinator style encourages "wholemeal programming".
  - Think of aggregate data as a whole, and process them as a whole!

3 λ expressions

- $\lambda x \rightarrow e$ denotes a function whose argument is x and whose body is e.
- $(\lambda x \rightarrow e_1)\ e_2$ denotes the function $(\lambda x \rightarrow e_1)$ applied to e_2. It can be reduced to e_1 with its *free* occurrences of x replaced by e_2.
- E.g.

$$\begin{array}{ll} & (\lambda x \rightarrow x \times x)\ (3+4) \\ = & (3+4) \times (3+4) \\ = & 49 \quad . \end{array}$$

- λ expression is a primitive and essential notion. Many other constructs can be seen as syntax sugar of λ's.
- For example, our previous definition of $square$ can be seen as an abbreviation of

$$\begin{array}{l} square :: Int \rightarrow Int \\ square = \lambda x \rightarrow x \times x \quad . \end{array}$$

  - Indeed, $square$ is merely a value that happens to be a function, which is in turn given by a λ expression.
- λ's are like all values — they can appear inside an expression, be passed as parameters, returned as results, etc.
- In fact, it is possible to build a complete programming language consisting of only λ expressions and applications. Look up "λ calculus".
- $\lambda x \rightarrow \lambda y \rightarrow e$ is abbreviated to $\lambda x\, y \rightarrow e$.
- The following definitions are all equivalent:

$$\begin{array}{l} smaller\ x\ y = \mathbf{if}\ x \leq y\ \mathbf{then}\ x\ \mathbf{else}\ y \\ smaller\ x = \lambda y \rightarrow \mathbf{if}\ x \leq y\ \mathbf{then}\ x\ \mathbf{else}\ y \\ smaller = \lambda x \rightarrow \lambda y \rightarrow \mathbf{if}\ x \leq y\ \mathbf{then}\ x\ \mathbf{else}\ y \\ smaller = \lambda x\, y \rightarrow \mathbf{if}\ x \leq y\ \mathbf{then}\ x\ \mathbf{else}\ y \quad . \end{array}$$