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

Shin-Cheng Mu

Autumn 2023

National Taiwan University and Academia Sinica

Simple Datatypes

Booleans

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

$$\textbf{data}\ \mathit{Bool} \;=\; \mathit{False} \mid \mathit{True}$$

(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.

$$\begin{aligned} \textbf{data}\ \mathit{Type} \;=\; & \mathit{Con}_1\ \mathit{Type}_{11}\ \mathit{Type}_{12}\ldots \\ \mid\; & \mathit{Con}_2\ \mathit{Type}_{21}\ \mathit{Type}_{22}\ldots \\ \mid\; & \vdots \end{aligned}$$

- 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:

$$\begin{aligned} &\mathit{not} && :: \mathit{Bool} \to \mathit{Bool} \\ &\mathit{not}\ \mathit{False} && = \mathit{True} \\ &\mathit{not}\ \mathit{True} && = \mathit{False} \end{aligned}$$

- 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:

$$
\begin{aligned}
(\wedge), (\vee) \quad &:: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool} \\
\mathit{False} \wedge x &= \mathit{False} \\
\mathit{True} \wedge x &= x \\
\mathit{False} \vee x &= x \\
\mathit{True} \vee x &= \mathit{True}
\end{aligned}
$$

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:

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

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

Example

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

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

Characters

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

  $$\textbf{data}\ \mathit{Char} = \text{'}a\text{'} \mid \text{'}b\text{'} \mid \ldots$$

  with functions:

  $$\mathit{ord} :: \mathit{Char} \to \mathit{Int}$$
  $$\mathit{chr} :: \mathit{Int} \to \mathit{Char}$$

- Characters are compared by their order:

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

Equality Check

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

$$(==) :: Char \to Char \to Bool$$

- $(==)$ is an *overloaded* name — one name shared by many different definitions of equalities, for different types:
  - $(==) :: Int \to Int \to Bool$
  - $(==) :: (Int, Char) \to (Int, Char) \to Bool$
  - $(==) :: [Int] \to [Int] \to Bool$ ...
- 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.

Tuples

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

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

- Or, had Haskell allow us to use symbols:

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

- Two projections:

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

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.
  - $[1, 2, 3, 4] :: \mathit{List\ Int}$
  - $[\mathit{True}, \mathit{False}, \mathit{True}] :: \mathit{List\ Bool}$
  - $[[1, 2], [\,], [6, 7]] :: \mathit{List}\ (\mathit{List\ Int})$
  - $[\,] :: \mathit{List}\ a$, the empty list (whose element type is not determined).
- *String* is an abbreviation for *List Char*; "abcd" is an abbreviation of ['a', 'b', 'c', 'd'].

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

$$\textbf{data}\ \mathit{List}\ a\ =\ [\,] \mid a : \mathit{List}\ a$$

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

Head and Tail

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

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*:
  - $['a'..'z']$ yields $['a', 'b', 'c'..'z']$.
- $[1..]$ 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:

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

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

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

Append and Concatenation

- Append: $(+\!\!+) :: \mathit{List}\ a \to \mathit{List}\ a \to \mathit{List}\ a$. 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 $(:) :: a \to \mathit{List}\ a \to \mathit{List}\ a$. It is a type error to write $[\,] : [3,4,5]$. $(+\!\!+)$ is defined in terms of $(:)$.
- $\mathit{concat} :: \mathit{List}\ (\mathit{List}\ a) \to \mathit{List}\ a$.
  - e.g. $\mathit{concat}\ [[1,2],[\,],[3,4],[5]] = [1,2,3,4,5]$.
  - concat is defined in terms of $(+\!\!+)$.

Take and Drop

- *take n* takes the first *n* elements of the list.
  - For 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 example, *drop* 0 *xs* $=$ *xs*
  - *drop* 3 "abcde" $=$ "cd"
  - *drop* 3 "ab" $=$ ""
- *take n xs* $+\!\!+$ *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 $\backslash$.
- 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. $zip\ \text{"abcde"}\ [1, 2, 3] = [('a', 1), ('b', 2), ('c', 3)]$
- 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?

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

  - Due to lazy evaluation (normal order evaluation), positions of the other occurrences are *not* evaluated!

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!

λ EXPRESSIONS

λ EXPRESSIONS

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

$$\begin{aligned} & (\lambda x \to x \times x)\ (3+4) \\ = \quad & (3+4) \times (3+4) \\ = \quad & 49 \quad . \end{aligned}$$

- 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

$$\begin{aligned} &\mathit{square} :: \mathit{Int} \to \mathit{Int} \\ &\mathit{square} = \lambda x \to x \times x \ . \end{aligned}$$

  - 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".

- $\lambda x \to \lambda y \to e$ is abbreviated to $\lambda x\, y \to e$.
- The following definitions are all equivalent:

$$
\begin{aligned}
\mathit{smaller}\ x\ y &= \textbf{if}\ x \leq y\ \textbf{then}\ x\ \textbf{else}\ y \\
\mathit{smaller}\ x &= \lambda y \to \textbf{if}\ x \leq y\ \textbf{then}\ x\ \textbf{else}\ y \\
\mathit{smaller} &= \lambda x \to \lambda y \to \textbf{if}\ x \leq y\ \textbf{then}\ x\ \textbf{else}\ y \\
\mathit{smaller} &= \lambda x\, y \to \textbf{if}\ x \leq y\ \textbf{then}\ x\ \textbf{else}\ y\ .
\end{aligned}
$$