Abstract

In programming, sets are represented as lists. This lecture explains how to work with them. We also comment on the differences between sets and lists, and on how to represent sets as lists.
module WWL

where
import List
import Char
The truth values true and false are rendered in Haskell as True and False, respectively. The type of a truth value is called Bool.

All function definitions are typed: in a type declaration we indicate the type of the argument or arguments and the type of the value. A function \texttt{foo} that takes an integer as its first argument, and an integer as its second argument and yields a truth value has type \texttt{Integer -> Integer -> Bool}.

Here is a type declaration for such a function, together with the actual definition:

\begin{verbatim}
divides :: Integer -> Integer -> Bool
divides m n = rem n m == 0
\end{verbatim}
The type Integer -> Integer -> Bool should be read as Integer -> (Integer -> Bool).

A type of the form \( a \rightarrow b \) classifies a procedure that takes an argument of type \( a \) to produce a result of type \( b \).

Thus, divides takes an argument of type Integer and produces a result of type Integer -> Bool.

Note that the result of applying divides to 5 is a function.

The function divides 5 yields True for arguments of type Integer that are divisible by 5, and False for all other Integer arguments.
The result of applying `divides` to an integer is a function that takes an argument of type `Integer`, and produces a result of type `Bool`.

WWL> :t divides 5
`divides 5 :: Integer -> Bool`

WWL> :t divides 5 7
`divides 5 7 :: Bool`

WWL> divides 5 7
`False`

WWL> divides 5
`ERROR - Cannot find "show" function for:
*** Expression : divides 5
*** Of type : Integer -> Bool`
Lambda Abstraction

Take the statement Diana loves Charles. By means of abstraction, we can get all kinds of properties and relations from this statement:

- ‘loving Charles’
- ‘being loved by Diana’
- ‘loving’
- ‘being loved by’

This works as follows. We replace the element that we abstract over by a variable, and we bind that variable by means of a lambda operator.
Lambda Abstraction – 2

Like this:

• $\lambda x. x$ loves Charles’ expresses ‘loving Charles’.
• $\lambda x. \text{Diana loves } x$ expresses ‘being loved by Diana’.
• $\lambda y. x$ loves $y$’ expresses ‘being loved by $x$’.
• $\lambda x\lambda y. x$ loves $y$’ expresses ‘loving’.
• $\lambda y\lambda x. x$ loves $y$’ expresses ‘being loved by’.
Lambda Abstraction – 3

In Haskell, \ x expresses lambda abstraction over variable x.

```
sqr :: Int -> Int
sqr = \ x -> x * x
```

The intention is that variabele x stands proxy for a number of type Int. The result, the squared number, also has type Int. The function sqr is a function that, when combined with an argument of type Int, yields a value of type Int. This is precisely what the type-indication Int -> Int expresses.
String Functions in Haskell

Hugs.Base> (\ x -> x ++ " emeritus") "professor"
"professor emeritus"

This combines lambda abstraction and concatenation. More on this below.

The types:

Hugs.Base> :t (\ x -> x ++ " emeritus")
\x -> x ++ " emeritus" :: [Char] -> [Char]
Hugs.Base> :t "professor"
"professor" :: String
Hugs.Base> :t (\ x -> x ++ " emeritus") "professor"
(\x -> x ++ " emeritus") "professor" :: [Char]
Concatenation

The type of the concatenation function:

```
WWL> :t (++)
(++) :: [a] -> [a] -> [a]
```

The type indicates that (++) not only concatenates strings. It works for lists in general.
More String Functions in Haskell

Hugs.Base> (\ x -> "nice " ++ x) "guy"
"nice guy"
Hugs.Base> (\ f -> \ x -> "very " ++ (f x))
          (\ x -> "nice " ++ x) "guy"
"very nice guy"

The types:

Hugs.Base> :t "guy"
"guy" :: String
Hugs.Base> :t (\ x -> "nice " ++ x)
\x -> "nice " ++ x :: [Char] -> [Char]
Hugs.Base> :t (\ f -> \ x -> "very " ++ (f x))
\f -> \ x -> "very " ++ f x :: (a -> [Char]) -> a -> [Char]
Properties of Things, Characteristic Functions

The property ‘being divisible by three’ can be represented as a function from numbers to truth values. The numbers

\[ \ldots, -9, -6, -3, 0, 3, 6, 9, \ldots \]

get mapped to True by that function, all other numbers get mapped to False.

Programmers call a truth value a Boolean, in honour of the British logician George Boole (1815–1864).

As the type of threefold we can therefore take \texttt{Int $\rightarrow$ Bool}. 
Here is a definition of the property of being a threefold with lambda abstraction. This uses the predefined function \( \text{rem} \). \( \text{rem} \ x \ y \) gives the remainder when \( x \) gets divided by \( y \).

\[
\text{threefold} :: \text{Int} \rightarrow \text{Bool} \\
\text{threefold} = \lambda x \rightarrow \text{rem} \ x \ 3 \ == \ 0
\]

WWL> threefold 5
False
WWL> threefold 12
True
Less than and less than or equal are examples of relations on the integers, and on various other number domains, in fact.

Less than is predefined in Haskell as (<), less than or equal as (<=). Other examples of relations are equality and inequality, predefined as (==) and (/=).

These characteristic functions have the following types (Ord a and Eq a are constraints on the type of a):

WWL> :t (<)
(<) :: Ord a => a -> a -> Bool
WWL> :t (<=)
(<=) :: Ord a => a -> a -> Bool
WWL> :t (==)
(==) :: Eq a => a -> a -> Bool
WWL> :t (/=)
(/=) :: Eq a => a -> a -> Bool
Characters and Strings

The Haskell type of characters is Char. Strings of characters have type [Char]. Similarly, lists of integers have type [Int]. The empty string (or the empty list) is []. The type [Char] is abbreviated as String. Examples of characters are 'a', 'b' (note the single quotes) examples of strings are "Montague" and "Chomsky" (note the double quotes). In fact, "Chomsky" can be seen as an abbreviation of the following character list:

[ 'C' , 'h' , 'o' , 'm' , 's' , 'k' , 'y' ].
Properties of Strings

If strings have type [Char] (or String), properties of strings have type [Char] -> Bool. Here is a simple property:

```haskell
aword :: [Char] -> Bool
aword [] = False
aword (x:xs) = (x == 'a') || (aword xs)
```

This definition uses pattern matching: (x:xs) is the prototypical non-empty list. The head of (x:xs) is x, the tail is xs. The head and tail are glued together by means of the operation :, of type a -> [a] -> [a]. The operation combines an object of type a with a list of objects of the same type to a new list of objects, again of the same type.
List Patterns

It is common Haskell practice to refer to non-empty lists as $x:xs$, $y:ys$, and so on, as a useful reminder of the facts that $x$ is an element of a list of $x$'s and that $xs$ is a list.

Note that the function `aword` is called again from the body of its own definition. We will encounter such recursive function definitions again and again in the course of this book.

What the definition of `aword` says is that the empty string is not an `aword`, and a non-empty string is an `aword` if either the head of the string is the character `a`, or the tail of the string is an `aword`. As you can see, characters are indicated in Haskell with single quotes.
The following calls to the definition show that strings are indicated with double quotes:

WWL> aword "Diana"
True
WWL> aword "loves"
False
More on List processing in Haskell

Integer is the type of arbitrary precision integers, Int the type of fixed precision integers.

[Integer] is the type of lists of Integers, [Int] the type of lists of Ints.

Here is a function that gives the minimum of a list of integers:

\[
\begin{align*}
mnmInt :: [Int] \to Int \\
mnmInt [] &= error "empty list" \\
mnmInt [x] &= x \\
mnmInt (x:xs) &= \text{min } x \ (\text{nmnInt } xs)
\end{align*}
\]

This uses a predefined function \text{min} for the minimum of two integers.
Pattern matching for lists

- The list pattern `[]` matches only the empty list,
- the list pattern `[x]` matches any singleton list,
- the list pattern `(x:xs)` matches any non-empty list.
Haskell Types

The basic Haskell types are:

- **Int** and **Integer**, to represent integers. Elements of **Integer** are unbounded.

- **Float** and **Double** represent floating point numbers. The elements of **Double** have higher precision.

- **Bool** is the type of Booleans.

- **Char** is the type of characters.

Note that the name of a type always starts with a capital letter.

To denote arbitrary types, Haskell allows the use of **type variables**. For these, a, b, ..., are used.
New types can be formed in several ways:

- By list-formation: if \( a \) is a type, \([a]\) is the type of lists over \( a \). Examples: \([\text{Int}]\) is the type of lists of integers; \([\text{Char}]\) is the type of lists of characters, or strings.

- By pair- or tuple-formation: if \( a \) and \( b \) are types, then \((a,b)\) is the type of pairs with an object of type \( a \) as their first component, and an object of type \( b \) as their second component. If \( a \), \( b \) and \( c \) are types, then \((a,b,c)\) is the type of triples with an object of type \( a \) as their first component, an object of type \( b \) as their second component, and an object of type \( c \) as their third component . . .

- By function definition: \( a \rightarrow b \) is the type of a function that takes arguments of type \( a \) and returns values of type \( b \).

- By defining your own datatype from scratch, with a data type declaration. More about this in due course.
Working with Lists: The map and filter Functions

If you use the Hugs command :t to find the types of the function map, you get the following:

Prelude> :t map
map :: (a -> b) -> [a] -> [b]

The function map takes a function and a list and returns a list containing the results of applying the function to the individual list members.

If \( f \) is a function of type \( a \rightarrow b \) and \( xs \) is a list of type \([a]\), then \( \text{map} \ f \ \text{xs} \) will return a list of type \([b]\). E.g., \( \text{map} \ (\cdot^2) \ [1..9] \) will produce the list of squares

\[[1, 4, 9, 16, 25, 36, 49, 64, 81]\]
Sections

In general, if op is an infix operator, (op x) is the operation resulting from applying op to its righthand side argument, (x op) is the operation resulting from applying op to its lefthand side argument, and (op) is the prefix version of the operator. Thus (2^) is the operation that computes powers of 2, and map (2^) [1..10] will yield

[2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]

Similarly, (>3) denotes the property of being greater than 3, and (3>) the property of being smaller than 3.
map

If $p$ is a property (an operation of type $a \rightarrow \text{Bool}$) and $l$ is a list of type $[a]$, then $\text{map } p \ l$ will produce a list of type $\text{Bool}$ (a list of truth values), like this:

Prelude> map (>3) [1..6]
[False, False, False, True, True, True]
Prelude>

map is predefined in Haskell.

Home-made definition:

\[
\begin{align*}
\text{map} &: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
\text{map } f \; [] &= [] \\
\text{map } f \; (x:xs) &= (f \; x) : (\text{map } f \; xs)
\end{align*}
\]
Another useful function is `filter`, for filtering out the elements from a list that satisfy a given property. This is predefined, but here is a home-made version:

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

Here is an example of its use:

```
WWL> filter (>3) [1..10]
[4,5,6,7,8,9,10]
```
List comprehension

List comprehension is defining lists by the following method:

\[ [ x \mid x \leftarrow xs, \text{property} \ x ] \]

This defines the sublist of \( xs \) of all items satisfying \( \text{property} \). It is equivalent to:

\[
\text{filter property xs}
\]
someEvens    = [ x | x <- [1..1000], even x ]
evensUntil n = [ x | x <- [1..n], even x ]
allEvens     = [ x | x <- [1..], even x ]

Equivalently:

someEvens    = filter even [1..1000]
evensUntil n = filter even [1..n]
allEvens     = filter even [1..]
sort

sort :: Ord a => [a] -> [a]
sort [] = []
sort (x:xs) = insert x (sort xs)

insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) | x <= y = x:y:ys
| otherwise = y: insert x ys
nub

nub removes duplicates, as follows:

\[
\begin{align*}
nub &: \text{Eq } a \Rightarrow [a] \rightarrow [a] \\
nub [ ] &= [ ] \\
nub (x:xs) &= x : nub (\text{filter } (/= x) \, xs)
\end{align*}
\]
Contained in

$$A \subseteq B \equiv \forall x \in A : x \in B.$$
**elem, all, and**

elem and all and and and and are predefined.

```
elem :: Eq a => a -> [a] -> Bool
elem x []     = False
elem x (y:ys) = x == y || elem x ys

all :: Eq a => (a -> Bool) -> [a] -> Bool
all p = and . map p
```

Definition of and: do it yourself.

Note the use of (.) for function composition (predefined).
Function Composition

The composition of two functions \( f \) and \( g \), pronounced ‘\( f \) after \( g \)’ is the function that results from first applying \( g \) and next \( f \).

Here is the Haskell implementation:

\[
(\cdot) :: (a \rightarrow b) \rightarrow (c \rightarrow a) \rightarrow (c \rightarrow b) \\
f \cdot g = \lambda x \rightarrow f (g \ x)
\]
Expressing Set Equality

Two sets $A$ and $B$ are equal if all elements of $A$ are elements of $B$ and vice versa. This allows us to define a set equality predicate for sets represented as lists, as follows:

```haskell
sameSet :: Eq a => [a] -> [a] -> Bool
sameSet xs ys = 
    containedIn xs ys && containedIn ys xs
```

WWL> sameSet [1,2] [2,1,1]
True
WWL> sameSet [1,2] [1,1,1]
False
WWL> sameSet (map (2*) [1..10]) (filter even [1..20])
True

WWL> (==) (map (2*) [1..10]) (filter even [1..20])
True

WWL> sameSet (map (2*) [1..]) (filter even [1..])
{Interrupted!}

WWL> (==) (map (2*) [1..]) (filter even [1..])
{Interrupted!}

WWL> sameSet (map (2*) [1..]) (filter even [0..])
{Interrupted!}

WWL> (==) (map (2*) [1..]) (filter even [0..])
False
Using Lists as Sets

There are several possibilities for this. One useful convention is to use represent sets as **ordered lists without duplicates**.

For a full implementation of sets as lists, one has to implement all the set operations as list operations, in such a way that the fundamental properties of order and non-multiplicity are preserved by each operation. One says: order and non-multiplicity have to be invariants of each operation that involves sets.

How to achieve this? The computer lab exercises of this week will make this clear.
Representing Relations

Various options:

- Lists of pairs, type [(a, a)].
- Characteristic functions, type a -> a -> Bool
- Characteristic functions of pairs, type (a, a) -> Bool
- Range functions, type a -> [a]
- Sets of pairs (using some suitable representation of sets . . . )
Relations as Lists of Pairs

type Rel a = [(a,a)]

Example relations:

r1 = [(1,2),(2,1)]

r2 = [(1,2),(2,1),(2,1)]

These relations have the same pairs, so they are in fact equal.
Test for equality of relations

```haskell
sameR :: Ord a => Rel a -> Rel a -> Bool
sameR r s = sort (nub r) == sort (nub s)
```

But the following is also possible:

```haskell
sameR' :: Eq a => Rel a -> Rel a -> Bool
sameR' r s = sameSet r s
```
Operations on relations: converse

Relational converse $R^\sim$ is given by:

$$R^\sim = \{ (y, x) \mid (x, y) \in R \}$$

Implementation

```haskell
cnv :: Rel a -> Rel a
cnv r = [ (y,x) | (x,y) <- r ]
```
Operations on relations: composition

The relational composition of two relations $R$ and $S$ on a set $A$:

$$R \circ S = \{(x, z) \mid \exists y \in A (xRy \land ySz)\}$$

For the implementation, it is useful to declare a new infix operator for relational composition.

```haskell
infixr 5 @@

(@@) :: Eq a => Rel a -> Rel a -> Rel a
r @@ s =
    nub [ (x,z) | (x,y) <- r, (w,z) <- s, y == w ]
```

Note that (@@) is the prefix version of @@.
Back and Forth Between Various Representations

See computer lab exercises for this week.
Next Time

Languages and Grammars . . .