FMFP - Lecture Notes Week 3

Ruben Schenk, ruben.schenk@inf.ethz.ch

March 16, 2022

0.0.1 Patterns

Pattern matching has two main purposes:

- checks if an argument has the proper form
- binds values to variables

Example: `(x : xs)` matches with `[2, 3, 4]` and binds:

```
x  = 2
xs = [3, 4]
```

Patterns are *inductively* defined:

- Constants: `-2, '1', True, []`
- Variables: `x, foo`
- Wild card: `_`
- Tuples: `(p1, p2,..., pk)`, where `p_i` are patterns
- Non-empty list: `(p1 : p2)`, where `p_i` are patterns

Moreover, patterns require to be **linear**, this means that each variable can occur at most once.

0.0.2 Advice on Recursion

Defining a recursion is best done by obeying the following simple steps:

- Step 1: Define the type of the function
- Step 2: Enumerate all different cases
- Step 3: Define the most simple cases
- Step 4: Define the remaining cases
- Step 5: Generalize and simplify

Example: The following code snippet shows an example of how we implement *insertion sort* recursively in Haskell:

```
isort :: [Int] -> Int
isort []       = []
isort (x : xs) = ins x (isort xs)

ins :: Int -> [Int] -> [Int]
ins a [] = [a]
ins a (x : xs)
  | a >= x    = a : (x : xs)
  | otherwise = x : ins a xs
```

Example: The following code snippet shows how we can implement *quicksort* recursively in Haskell:

```
qsort [] = []
qsort (x : xs) =
  qsort (lesseq x xs) ++ [x] ++ qsort (greater x xs)
  where
    lesseq _ [] = []
    lesseq x (y : ys)
      | (y <= x)  = y : lesseq x ys
      | otheriwse =     lesseq x ys
    greater _ [] = []
    greater x (y : ys)
      | (y > x)   = y : greater x ys
      | otherwise =     greater x ys
```

0.0.3 List Comprehensions

List comprehension is a notation for sequential processing of list elements. It is analogous to set comprehension in set theory, i.e. $\{2 \cdot x \,|\, x \in X\}$. In Haskell, this is equivalent to $[2 * x \,|\, x \leftarrow xs]$.

List comprehensions are very powerful! The following code snippet, again, implements *quicksort* as shown previously:

```
q []       = []
q (p : xs) = q [x | x <- xs, x <= p] ++ [p] ++ q [x | x <- xs, x > p]
```

0.0.4 Induction over Lists

How are elements in `[T]` constructed? `[] :: [T]` and `(y : ys) :: [T]` if `y :: T` and `ys :: [T]`. This corresponds to the following rule:

- Proof by induction: to prove P for all `xs` in `[T]`
- Base case: prove $P[xs \to []]$
- Step case: prove $\forall y :: T,\, ys :: [T].P[xs \to ys] \to P[xs \to y : ys]$, i.e.
  - Fix arbitrary: $y :: T$ and $ys :: [T]$ (both not free in P)
  - Induction hypothesis: $P[xs \to ys]$
  - To prove: $P[xs \to y : ys]$

0.1 Abstractions

0.1.1 Polymorhpic Types

If we consider the `length` function, it should output the length of a list of *any* type. We say that the type of the function is **polymorphic,** i.e. `[t] -> Int` for all types t.

This is often called **parametric polymorphism,** which is different from *subtyping polymorphism,* where methods can be applied to objects of sub-classes only.

Definition: A type w for f is a **most general** (also called **principal**) type iff. for all types s for f, s is an instance of w.

It is important to note that type variables in Haskell start with a *lower-case letter!*

Example: Consider the following polymorphic types:

```
:type (++)
(++) :: [a] -> [a] -> [a]

:type zip
zip :: [a] -> [b] -> [(a, b)]

:type []
[] :: [a]
```

0.1.2 Higher-order Functions

We can distinguish the order of functions in the following way:

- First order: Arguments are base types or constructor types

```
Int -> [Int]
```

- Second order: Arguments are themselves functions

```
(Int -> Int) -> [Int]
```

- Third order: Arguments are functions, whose arguments are functions

```
((Int -> Int) -> Int) -> [Int]
```

- Higher-order functions: Functions of arbitrary order

Example: Consider the `map` function:

```
map :: (a -> b) -> [a] -> [b]
map f []       = []
map f (x : xs) = f x : map f xs

times2 x = 2 * x

double xs = map times2 xs
```

Example: Consider the `foldr` function:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []       = z
foldr f z (x : xs) = f x (foldr f z xs)

sumList xs = foldr (+) 0 xs
```

0.1.3 λ-Expressions

Consider the following two functions:

```
times2 x = 2 * x
double xs = map times2 xs

atEnd x xs = xs ++ [x]
rev xs = foldr atEnd [] xs
```

Haskell provides a notation to write functions like `times2` and `atEnd` in-line via so-called **λ-expressions:**

```
? map (\x -> 2 * x) [2, 3, 4]
[4, 6, 8]

? foldr (\x xs -> xs ++ [x]) [] [1, 2, 3, 4]
[4, 3, 2, 1]
```

This is also called *Church's λ-notation,* i.e. replacing λ by the character `'\'`.

0.1.4 Functions as Values

In Haskell, functions can be returned as values! Consider the following simple example where we return the two-times-application of some function f:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)

twice :: (t -> t) -> (t -> t)
twice f = f . f

? twice times2 3
12 :: Int
```

0.1.5 Differece Lists

Difference lists are functions `[a] -> [a]` that prepend a list to its argument.

```
type DList a = [a] -> [a]

empty :: DList a
empty = \xs -> xs                       -- empty list

sngl :: a -> DList a
sngl x = \xs -> x : xs                  -- singleton list

app :: DList a -> DList a -> DList a
ys 'app' zs = \xs -> ys (zs xs)         -- concatenation

fromList :: [a] -> DList a
fromList ys = \xs -> ys ++ xs           -- conversion from lists

toList :: DList a -> [a]
toList ys = ys []                       -- conversion to lists
```