FMFP - Lecture Notes Week 3

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

March 16, 2022

ruben.schenk@inf.ethz.ch

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. $\{2 \cdot x \mid x \in X\}$. In Haskell, this is equivalent to $[2 * x \mid x \leftarrow xs]$.

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: $P[xs \to y : ys]$