1.1.3 Patterns

Dienstag, 19. April 2016 15:30

Patterns are special expressions that are used to describe the forms of expected arguments.

```
len :: [Int] -> Int
len [] = 0
len (x:xs) = 1 + len xs
```

```
append :: [Int] -> [Int] -> [Int]
append [] ys = ys
append (x:xs) ys = x : append xs ys
```

append is pre-defined as an infix-function called ++:

[1,2] ++ [3,4,5] = [1,2,3,4,5]

---

len (append $\underbrace{[1]}_{1:[]}$ [2]) = ← to evaluate len, we need to evaluate append a few steps until we know whether the first defining equation of len matches

len (1 : append [] [2]) =

1 + len (append [] [2]) = ... = 2

```
zeros :: [Int]
zeros = 0 : zeros
```

```
f :: [Int] -> [Int] -> [Int]
f [] ys = []
f xs [] = []
```

f [] zeros = [] terminates

f zeros [] = f (0 : zeros) [] = [] terminates

↑ at this point, one notices that the pattern [] in the first f-equation does not match.

Patterns have to be <u>linear</u>, i.e., no variable may occur twice on the lhs of a defining equation.

Reason:

```
equal :: [Int] -> [Int] -> Bool
equal xs xs     = True
equal xs (x:xs) = False
```

} not allowed in Haskell

Up to now: evaluation strategy influences termination and efficiency, but not the result.

Here: evaluation strategy does influence result:

equal zeros zeros

↙ True

↘ equal zeros (0:zeros) ↓ False

For every form of pattern (Slide 12), we describe which arguments are matched by this pattern and how the variables of the pattern are instantiated.

- <u>Var</u> : a variable matches any expression, matching instantiates the variable by the expression

  ```
  square x = x * x
  ```

  x: pattern

  square 11 = 11 * 11 = 121

- _ : joker pattern, matches any expression, but no variable is instantiated. "_" may occur multiple times, stands for possibly different values stands for possibly different values

```
und True y = y              | und False True = False
und _ _ = False             |
```

- integer, float, char, string : these patterns only match themselves

```
is_5 :: Int -> Bool

is_5 5 = True
is_5 _ = False
```

- (constr pat_1 ... pat_n), $n \geq 0$ — matches all expressions (constr exp_1 ... exp_n) where pat_i matches exp_i

  ↑ data constructor (e.g. True, [ ], :, ...)

  e.g. len $\underbrace{(x:xs)}_{\text{pattern}}$ = 1 + len xs

- [pat_1, ..., pat_n], $n \geq 0$ matches all lists [exp_1, ..., exp_n] where pat_i matches exp_i

```
has_length_three :: [Int] -> Bool
has_length_three [_,_,_] = True
has_length_three _       = False
```

- (pat_1, ..., pat_n) where $n \geq 0$ matches all tuples (exp_1, ..., exp_n) where pat_i matches exp_i.

```haskell
maxi :: (Int, Int) -> Int
maxi (0, y) = y
maxi (x, 0) = x
maxi (x, y) = 1 + maxi (x-1, y-1)
```