1.2 Higher-Order Functions

Mittwoch, 27. April 2016 10:00

We have introduced the 4 main components of Haskell (declarations, expressions, patterns, types).

Now: functional programming techniques

1.2 • Higher-Order Functions

1.3 • Lazy Programming with Infinite Data Objects

1.4 • Programming with Monads (IO)

Higher-Order Functions: functions that have functions as arguments or as result

square :: Int → Int   first-order function

plus :: $\underset{\text{argument}}{\text{Int}}$ → $\underbrace{\text{(Int → Int)}}_{\text{result}}$ higher-order function

Function Composition (" . ")

in mathematics: $f \circ g$ stands for the composition of the functions f and g

in Haskell: pre-defined as an operator (.):

```
infixr 9 .
(.) :: (b->c) -> (a->b) -> (a->c)      higher-order function
```

$$\underset{\text{type } b\to c}{f} \;.\; \underset{\text{type } a\to b}{g} = \backslash \underset{\text{type } a}{x} \to \underbrace{f\ (g\ x)}_{\text{type } c}$$

So: (half.square) 4 results in $\frac{4^2}{2} = 8$

$((\backslash x \to x+1) . \text{square})\ 5$ results in 26

The function "map" (Slide 21)

Idea for functional programming with higher-order functions:

- Many algorithms on a data structure have similar recursion structure.
- Instead of implementing each of these algorithms from scratch, identify those parts that are equal and represent them by a higher-order function that implements this recursion structure.
- Then actual algorithms can be implemented by re-using this higher-order function again and again.

```
suc :: Int -> Int
suc = plus 1

suclist :: [Int] -> [Int]
suclist [] = []
suclist (x:xs) = suc x : suclist xs
```

Thus: suclist $[x_1, .., x_n]$ = $[\text{suc } x_1, \ldots, \text{suc } x_n]$

```
sqrtlist :: [Float] -> [Float]
sqrtlist [] = []
sqrtlist (x:xs) = sqrt x : sqrtlist xs
```

Thus: $\text{sqrtlist } [x_1, \dots, x_n] = [\text{sqrt } x_1, \dots, \text{sqrt } x_n]$

Try to abstract from the differences between suclist and sqrtlist in order to find their common recursion structure:

- Abstract from the type of the list elements
  (Int resp. Float are replaced by a type variable).
  This is only possible in prog. languages with parametric polymorphism.
- Abstract from the auxiliary function that is applied to each list element.
  (suc resp. sqrt are replaced by a variable that stands for a function).
  This is only possible in prog. languages with higher-order functions.

So in our example, some function g is applied to all elements in a list.

$\Rightarrow$ we obtain a function f such that

$$f\ [x_1, \dots, x_n] = [g\ x_1, \dots, g\ x_n].$$

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

g has type a -> b

Since the variable g occurs on the right-hand side, it should be one of f's arguments

```
map :: (a -> b) -> [a] -> [b]
```

```
map g [ ] = [ ]
map g (x:xs) = g x : map g xs
```

map is a higher-order function that implements the recursion structure: "traverse a list and apply a function to each element in the list". ← is pre-defined in Haskell

Now functions like suclist or sqrtlist should not be implemented from scratch, but they should be implemented using "map":

- shorter
- more readable

```
suclist :: [Int] -> [Int]          sqrtlist :: [Float] -> [Float]
suclist = map suc                  sqrtlist = map sqrt
```

"map" can also be defined for user-defined data structures like trees, graphs, ... : traverse the data structure and apply a function to each component

The function "filter" (Slide 22)

```
dropEven :: [Int] -> [Int]
dropEven [ ] = [ ]
dropEven (x:xs) | odd x     = x : dropEven xs
                | otherwise = dropEven xs
```

Thus:
dropEven [1,2,3,4] results in [1,3]

```
dropUpper :: [Char] -> [Char]
dropUpper [ ] = [ ]
dropUpper (x:xs) | isLower x = x : dropUpper xs
                 | otherwise = dropUpper xs
```

Thus:
dropUpper "GmbH" results in "mb"

Thus:
dropEven [1,2,3,4] results in [1,3] | dropUpper "GmbH" results in "mb"

Haskell has a library organized in modules.
By default, Haskell imports pre-defined functions from the module "Prelude".
Other modules have to be imported explicitly: To use "isLower", one has to add "import Char" at the beginning of the file.

Try to abstract from the differences between dropEven and dropUpper in order to obtain a general function that implements their common recursion structure:

- replace the types Int resp. Char by a type variable a
- replace the functions odd resp. isLower by a function variable g

```
f :: [a] -> [a]
f [] = []
f (x:xs) | g x       = x : f xs
         | otherwise = f xs
```

Since g occurs on the rhs, it should also be one of the arguments of the function

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

Now dropEven and dropUpper can be implemented in a much shorter way:

```
dropEven :: [Int] -> [Int]
dropEven = filter odd
```

```
dropUpper :: [Char] -> [Char]
dropUpper = filter isLower
```

"filter" is pre-defined on lists, but can also implemented on user-defined data structures: traverse a data structure and drop all those components that do not satisfy a certain Boolean function.

The function "fold" (Slide 23)

We first illustrate this function with user-defined lists.

```
plus :: Int -> Int -> Int          times :: Int -> Int -> Int
plus x y = x + y                   times x y = x * y

add :: (List Int) -> Int           prod :: (List Int) -> Int
add Nil = 0                        prod Nil = 1
add (Cons x xs) = plus x (add xs)  prod (Cons x xs) = times x (prod xs)
```

Thus: if we call the function with the argument Cons x_1 (Cons x_2 (...(Cons x_n Nil)))
we obtain

plus x_1 (plus x_2 (... (plus x_n 0)...)) | times x_1 (times x_2 (... (times x_n 1)...))

Both functions take a data object and replace each data constructor by a new function:

- add replaces Nil by 0, Cons by plus
- prod replaces Nil by 1, Cons by times

To abstract from their differences, we
replace Nil by a variable e, Cons by a variable g

```
f :: (List a) -> b
f Nil = e
f (Cons x xs) = g x (f xs)
```

$$\texttt{f (Cons x xs) = g}\ \underbrace{\texttt{x}}_{\text{Type a}}\ \underbrace{\texttt{(f xs)}}_{\text{Type b}}$$

Since the variables e and g occur on the rhs, they should also be arguments of the function

Type of e is b

Type of g is a -> b -> b

$$\underbrace{\;\;\text{Type } a \quad \text{Type } b\;\;}_{\text{Type } b} \qquad \text{Type of } g \text{ is } a \to b \to b$$

$$fold :: (a \to b \to b) \to b \to (List\ a) \to b$$
$$fold\ g\ e\ Nil = e$$
$$fold\ g\ e\ (Cons\ x\ xs) = g\ x\ (fold\ g\ e\ xs)$$

Thus: $fold\ g\ e\ (Cons\ x_1\ (Cons\ x_2 (\ldots (Cons\ x_n\ Nil) \ldots)))$ results in

$$g\ x_1 (g\ x_2 (\ldots (g\ x_n\ e) \ldots))$$

Now add and prod can be implemented in a much shorter way:

add :: (List Int) -> Int
add = fold plus 0

prod :: (List Int) -> Int
prod = fold times 1

Another example that can be simplified using fold:

Concat (Slide) appends all elements in a list of lists

(for pre-defined lists, concat is pre-defined in Haskell, e.g.: concat [[1,2],[ ], [3]] = [1,2,3])

Concat can be implemented in a short way:

concat :: List (List a) -> List a
concat = fold append Nil

On pre-defined lists, fold is also pre-defined under the name "foldr" (Slide 24).
right

Fold-functions can also be implemented on user-defined data structures: replace every data constructor by a new function.

List Comprehensions (Slide 25)

Mathematics: $\{x * x \mid x \in \{1,\dots,5\}, \text{odd}(x)\}$

Haskell : $[\, x * x \mid x \leftarrow \underbrace{[1\,..\,5]}, \text{odd } x \,]$

Result is $[1, 9, 25]$

$[a\,..\,b]$ computes $[a, a+1, \dots, b]$

List comprehensions have the following form:

$[\, \underline{exp} \mid \underline{qual_1}, \dots, \underline{qual_n} \,]$

qualifiers, which are

- generators (like $x \leftarrow [1\,..\,5]$) or
- guards (like odd x)

Meaning of a generator $\underline{var} \leftarrow \underline{exp'}$:

The variable $\underline{var}$ should take all values in the list $\underline{exp'}$.

Meaning of a guard : boolean expression to restricts the values of $\underline{var}$.

Haskell translates list comprehensions into expressions with higher-order functions:

A list comprehension $[\, \underline{exp} \mid Q \,]$ is translated as follows:

list of qualifiers. If Q is empty, then $[\, \underline{exp} \mid Q \,]$ stands for $[\, \underline{exp} \,]$.

$[\underline{exp} \mid \underline{var} \leftarrow \underline{exp}', Q] = concat\ (map\ f\ \underline{exp}')$ where $f\ \underline{var} = [\underline{exp} \mid Q]$

Concat concatenates all elements of a list of lists,
e.g. $concat\ [[1,2,3],[\,],[4]] = [1,2,3,4]$

Thus:

$[\underline{exp} \mid \underline{var} \leftarrow [a_1,\dots,a_n], Q] =$

$concat\ (\underbrace{map\ f\ [a_1,\dots,a_n]}_{[f\,a_1,\, f\,a_2,\dots,\, f\,a_n]})$ where $f\ \underline{var} = [\underline{exp} \mid Q]$ $=$

$f\,a_1 \mathbin{++} f\,a_2 \mathbin{++} \dots \mathbin{++} f\,a_n$ where $f\ \underline{var} = [\underline{exp} \mid Q]$ $=$

$[\underline{exp} \mid Q]\ \underbrace{[\underline{var}/a_1]}_{\text{substitution that replaces } \underline{var} \text{ by } a_1} \mathbin{++} [\underline{exp} \mid Q]\ [\underline{var}/a_2] \mathbin{++} \dots \mathbin{++} [\underline{exp} \mid Q]\ [\underline{var}/a_n]$.

Meaning of guards:

$[\underline{exp} \mid \underline{exp}', Q] =$ if $\underline{exp}'$ then $[\underline{exp} \mid Q]$ else $[\,]$

$\uparrow$ guard of type Bool

Example:

$[x * x \mid x \leftarrow [1\,..\,5],\ odd\ x]$

$= concat\ (map\ f\ [1\,..\,5])$ where $\underbrace{f\ x = [x * x \mid odd\ x]}$

$= concat\ [f\,1, f\,2, f\,3, f\,4, f\,5]$ where $\underbrace{\quad}_{''}$

$= f\,1 \mathbin{++} f\,2 \mathbin{++} f\,3 \mathbin{++} f\,4 \mathbin{++} f\,5$ where $\underbrace{\quad}_{''}$

$=$ —— '' —— where $f\ x =$ if $odd\ x$ then $[x * x]$ else $[\,]$

$= [1] \mathbin{++} [\,] \mathbin{++} [9] \mathbin{++} [\,] \mathbin{++} [25]$

$= [1, 9, 25]$

Example to show that the order of qualifiers is important:

Example to show that the order of qualifiers is important:

$[(a, b) \mid a \leftarrow [1..3], b \leftarrow [1..2]]$

$= [(1,1), (1,2), (2,1), (2,2), (3,1), (3,2)]$

$[(a,b) \mid b \leftarrow [1..2], a \leftarrow [1..3]]$

$= [(1,1), (2,1), (3,1), (1,2), (2,2), (3,2)]$

Later qualifiers can depend on earlier qualifiers:

$[(a,b) \mid a \leftarrow [1..4], b \leftarrow [a+1..4]]$

$= [(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)]$

Guards and generators can be mixed:

$[(a,b) \mid a \leftarrow [1..4], \text{even } a, b \leftarrow [a+1..4], \text{odd } b]$

$= [(2,3)]$

With list comprehensions, one can implement list algorithms like map:

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

One can also define many other useful list algorithms like quicksort:

```
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort l1 ++ [x] ++ qsort l2
    where l1 = [y | y <- xs, y < x]
```

$l_2 = [y \mid y \leftarrow xs,\ y >= x]$

Much shorter + simpler than in imperative languages!!
(Slide 26)