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

\[
\text{square} :: \text{Int} \rightarrow \text{Int} \quad \text{first-order function}
\]

\[
\text{plus} :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \quad \text{higher-order function}
\]

Function Composition (\( \circ \))

In mathematics: \( f \circ g \) stands for the composition of the functions \( f \) and \( g \)

In Haskell: pre-defined as an operator (\( . \)):

\[
\text{infixr } 9 \ .
\]

\[
(\ .\ ) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)
\]

\[
\left\{ \begin{array}{l}
\text{\( f \cdot g \) = \( \lambda x \rightarrow f \ (g \ x) \)} \\
\text{\( f \) \text{ is of type } b \rightarrow c \)} \\
\text{\( g \) \text{ is of type } a \rightarrow b \)} \\
\text{\( f \cdot g \) \text{ is of type } a \rightarrow c}
\end{array} \right.
\]

So: \((\text{half}\cdot\text{square})\) 4 results in \(\frac{4^2}{2} = 8\)
The function "map" (Slide 24)

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.

\[ ((\lambda x \rightarrow x+1). \text{square}) 5 \text{ results in } 26 \]

\[ \text{suc} :: \text{Int} \rightarrow \text{Int} \]
\[ \text{suc} = \text{plus} \ 1 \]

\[ \text{sucList} :: [\text{Int}] \rightarrow [\text{Int}] \]
\[ \text{sucList} \ [3] = [3] \]
\[ \text{sucList} \ (x : xs) = \text{suc} \ x \ : \ \text{sucList} \ xs \]

Thus: \[ \text{sucList} \ [x_n, \ldots, x_0] = [\text{suc} \ x_n, \ldots, \text{suc} \ x_0] \]

\[ \text{sqrtList} :: [\text{Float}] \rightarrow [\text{Float}] \]
\[ \text{sqrtList} \ [3] = [3] \]
\[ \text{sqrtList} \ (x : xs) = \text{sqrt} \ x \ : \ \text{sqrtList} \ xs \]
Thus: \( \text{sqrtlist }[x_1, ..., x_n] = [\text{sqrt } x_1, ..., \text{sqrt } x_n] \)

Try to abstract from the differences between suc and list 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

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

\( f :: [a] \to [b] \)

\( f [] = [] \)

\( f (x : xs) = g x : f xs \)

\( \text{has type } a \to b \)

\( \text{map} :: (a \to b) \to [a] \to [b] \)

Since the variable \( g \) occurs on the right-hand side, it should be one of \( f \)'s arguments.
map \ g \ \texttt{[x] = [x]}

map \ g \ (x \cdot xs) = g \cdot x : \text{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". \texttt{\_\_} is pre-defined in Haskell.

New functions like \texttt{suc\_list} or \texttt{sqrt\_list} should not be implemented from scratch, but they should be implemented using "map":

\begin{itemize}
  \item shorter
  \item more readable
\end{itemize}

\texttt{suc\_list :: [Int] \rightarrow [Int]}

\texttt{suc\_list = map suc}

\texttt{sqrt\_list :: [Float] \rightarrow [Float]}

\texttt{sqrt\_list = 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.

\underline{The function "filter" (Slide 22)}

\texttt{drop\_even :: [Int] \rightarrow [Int]}

\texttt{drop\_even \ [x] = [x]}

\texttt{drop\_even \ (x:x:s) | odd \ x = x : drop\_even \ xs}
\texttt{| otherwise = drop\_even \ xs}

Thus: \texttt{drop\_even \ [1,2,3,4] results in \ [1,3]}.

\texttt{drop\_upper :: [Char] \rightarrow [Char]}

\texttt{drop\_upper \ [x] = [x]}

\texttt{drop\_upper \ (x:x:s) | islower \ x = x : drop\_upper \ xs}
\texttt{| otherwise = drop\_upper \ xs}

Thus: \texttt{drop\_upper \ "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 α
- replace the functions odd resp. isLower by a function variable g

\[ f :: [\alpha] \rightarrow [\alpha] \]

\[ f \ [\ ] = [\ ] \]

\[ f (x : xs) \mid g x = x : f xs \]
\[ \quad \text{otherwise} = f xs \]

\[ \text{filter} :: (\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow [\alpha] \]

\[ \text{filter} \ g \ [\ ] = [\ ] \]

\[ \text{filter} \ g \ (x : xs) \mid g x = x : \text{filter} \ g \ xs \]
\[ \quad \text{otherwise} = \text{filter} \ g \ xs \]

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

\[ \text{dropEven} :: [\text{Int}] \rightarrow [\text{Int}] \]
\[ \text{dropEven} = \text{filter} \ \text{odd} \]

\[ \text{dropUpper} :: [\text{Char}] \rightarrow [\text{Char}] \]
\[ \text{dropUpper} = \text{filter} \ \text{isLower} \]
"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.

\[
\begin{align*}
\text{plus} & : \mathbb{Int} \to \mathbb{Int} \to \mathbb{Int} \\
\text{times} & : \mathbb{Int} \to \mathbb{Int} \to \mathbb{Int} \\
\text{plus} \times y & = x + y \\
\text{times} \times y & = x \times y \\
\text{add} & : (\text{List} \ \mathbb{Int}) \to \mathbb{Int} \\
\text{prod} & : (\text{List} \ \mathbb{Int}) \to \mathbb{Int} \\
\text{add} \ \text{Nil} & = 0 \\
\text{prod} \ \text{Nil} & = 1 \\
\text{add} \ (\text{Cons} \times x_0) & = \text{plus} \times (\text{add} \times x_u) \\
\text{prod} \ (\text{Cons} \times x_0) & = \text{times} \times (\text{prod} \times x_u)
\end{align*}
\]

Thus, if we call the function with the argument \(\text{Cons} \ x_1 \ (\text{Cons} \ x_2 \ (\ldots \ (\text{Cons} \ x_n \ \text{Nil}) \ldots))\) we obtain:

\[
\begin{align*}
\text{plus} \times_1 (\text{plus} \times_2 (\ldots (\text{plus} \times_n 0) \ldots)) \\
\text{times} \times_1 (\text{times} \times_2 (\ldots (\text{times} \times_n 1) \ldots))
\end{align*}
\]

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 \(g\):

\[
\begin{align*}
f & : (\text{List} \ \mathbb{a}) \to \mathbb{b} \\
f \ \text{Nil} & = e \\
f \ (\text{Cons} \times x_0) & = g \times (f \times x_u)
\end{align*}
\]

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

Type of \(e\) is \(\mathbb{b}\)

Type of \(g\) is \(\mathbb{a} \to \mathbb{b} \to \mathbb{b}\)
\[
\text{fold} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow (\text{List } a) \rightarrow b
\]
fold \(g \in \text{Nil} = e\)
fold \(g \in (\text{Cons } x \times xs) = g \times (\text{fold } g \in xs)\)

Thus: \(\text{fold } g \in (\text{Cons } x_1 (\text{Cons } x_2 (\ldots (\text{Cons } x_n \text{ Nil}) \ldots)))\) results in
\(g \times x_1 (g \times x_2 (\ldots (g \times x_n e) \ldots))\)

Now \(\text{add}\) and \(\text{prod}\) can be implemented in a much shorter way:
\(\text{add} :: (\text{List } \text{Int}) \rightarrow \text{Int}\)
\(\text{prod} :: (\text{List } \text{Int}) \rightarrow \text{Int}\)
\(\text{add} = \text{fold} \text{ plus } 0\)
\(\text{prod} = \text{fold} \text{ times } 1\)

Another example that can be simplified using \text{fold}:
\(\text{Concat} \ (\text{Slide})\) appends all elements in a list of lists

\(\text{(for pre-defined lists, \text{Concat} is pre-defined in Haskell, e.g.:} \text{Concat [1,2,3,1,2,3] = [1,2,3]}\)

\(\text{Concat can be implemented in a short way:}\)
\(\text{Concat} :: \text{List (List } a) \rightarrow \text{List } a\)
\(\text{Concat} = \text{fold} \text{ append } \text{Nil}\)

On pre-defined lists, \(\text{fold}\) is also pre-defined under the name "\text{foldr}" (Slide 24).

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

**List Comprehensions (Slide 25)**

Mathematics: \[ \{ x \mid x \in \{1, \ldots, 5\}, \text{odd}(x) \} \]

Haskell: \[ [x \mid x \in [1..5], \text{odd } x] \]

Result is \[ [1, 9, 25] \]

\[ [a .. 6] \text{ computes } [9, 9+1, \ldots, 6] \]

List comprehensions have the following form:

\[ [\exp_1 | \text{qual}_1, \ldots, \text{qual}_n] \]

Qualifiers, which are

- generators (like \( x \in [1..5] \)) or
- guards (like \( \text{odd } x \))

Meaning of a generator: \( \text{var} \gets \exp' \): The variable \( \text{var} \) should take all values in the list \( \exp' \).

Meaning of a guard: boolean expression to restricts the values of \( \text{var} \).

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

A list comprehension \( [\exp_1 | Q] \) is translated as follows:

\( [\exp] \).
\[
[\text{exp} | \text{var} \leftarrow \text{exp'}, Q] = \text{concat} \left( \text{map} f \text{exp'} \right) \text{ where } f \text{ var} = [\text{exp} | Q]
\]

Concat concatenates all elements of a list of lists, e.g. \[\text{concat} \left[ [1,2,3], [7], [4] \right] = [1,2,3,7,4] \]

Thus:
\[
[\text{exp} | \text{var} \leftarrow [a_n, ..., a_n], Q] = \text{concat} \left( \text{map} f [a_n, ..., a_n] \right) \text{ where } f \text{ var} = [\text{exp} | Q]
\]
\[= \left[ a_n, f_{a_2}, ..., f_{a_n} \right] \]
\[
\text{ where } f \text{ var} = [\text{exp} | Q]
\]
\[= \left[ \text{exp} | Q \right] \left[ \text{var}/a_n \right] ++ \left[\text{exp} | Q \right] \left[ \text{var}/a_2 \right] ++ ... ++ \left[\text{exp} | Q \right] \left[ \text{var}/a_n \right]
\]

Substitution that replaces \text{var} by \text{an}

Meaning of guards:
\[\text{[exp | exp', Q]} = \text{if exp' then [exp | Q] else []} \]

guard of type \text{Bool}

Example:
\[\left[ x \times x \mid x \leftarrow [1..5], \text{odd } x \right]\]
\[= \text{concat} \left( \text{map} f [1..5] \right) \text{ where } f \text{ x} = \left[ x \times x \mid \text{odd } x \right]
\]
\[= \text{concat} \left[ f_1, f_2, f_3, f_4, f_5 \right] \text{ where }
\]
\[= f_1 ++ f_2 ++ f_3 ++ f_4 ++ f_5 \text{ where } f \text{ x} = \text{if odd } x \text{ then } [x \times x] \text{ else []}
\]
\[= [7] ++ [9] ++ [13] ++ [25]
\]
\[= [7, 9, 13, 25]
\]

Example to show that the order of qualifiers is important:
Example to show that the order of qualifiers is important:
\[ (a, b) \mid a \in [1..3], b \in [1..2] \]
\[ = \{ (1,1), (1,2), (2,1), (2,2), (3,1), (3,2) \} \]
\[ (a, b) \mid b \in [1..2], a \in [1..3] \]
\[ = \{ (1,1), (2,1), (3,1), (1,2), (2,2), (3,2) \} \]
Later qualifiers can depend on earlier qualifiers:
\[ (a, b) \mid a \in [1..4), b \in [a+1..4] \]
\[ = \{ (1,2), (1,3), (1,4), (2,3), (2,4), (3,4) \} \]
Guarded and generators can be mixed:
\[ (a, b) \mid a \in [1..4], \text{even } a, b \in [a+1..4], \text{odd } b \]
\[ = \{ (2,3) \} \]

With list comprehensions, one can implement list algorithms like \texttt{map}:
\[
\text{map} :: (a \to b) \to [a] \to [b] \\
\text{map } f \ x s = \{ f \ x \mid x \in x s \}
\]
One can also define many other useful list algorithms like \texttt{quicksort}:
\[
\text{qsort} :: \text{Ord } a \Rightarrow [a] \to [a] \\
\text{qsort } [ ] = [ ] \\
\text{qsort } (x : x s) = \text{qsort } l_1 ++ [x] ++ \text{qsort } l_2
\]
where \( l_1 = [y \mid y \in x s, y < x] \)
\[ l_2 = \{ y \mid y \in x, \; y \geq x \} \]

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