Functional Programming
an overview

(211205)

J. Kuper

j.kuper@ewi.utwente.nl

September 13, 2011

Preliminary remark. This text is *not* intended as a textbook on functional programming, but as an accompanying text to the lectures. It mentions rather then explains several essential aspects of a functional programming language. I am convinced that *the learning is in the doing*, and it is my hope that this text can serve as a readable reference guide when doing exercises.

I do intend, though, to extend this text into a more self contained introduction to functional programming, so I will highly appreciate all comments on this text.

Note. This text was originally written for the functional language Amanda[1]. It is now in the process of being rewritten for Haskell; there still follow some sections on I/O and monads. Thanks go to Philip Hölzenspies.

1 Expressions, definitions

The activity of programming in a functional programming language consists of defining functions, and evaluating their effect on given arguments. Definitions are collected in a *script file*, which can be *loaded* into the *interactive environment* or *compiled* into an *executable*. Script files may import other script files (modules, see section 6).

In this course we will use Haskell[2], and script files typically have the extension "`.hs`". When a line in a Haskell script file contains the symbol `--`, the rest of that line is considered as *comment*. Multi-line comments may be included in `{-` and `-}`.

Expressions. We start with a warning: a functional programming language *only* has expressions, there are *no* statements as in imperative or object-oriented languages. In particular, there is no assignment statement. A functional program thus does not "work" by explicitly storing and changing values in computer memory, but by *evaluating* expressions, in order to calculate their values.

As usual, expressions are built up from identifiers, constants, variables, and operations. *The* basic operation in a functional language is *application*

[1]Amanda is written by Dick Bruin and can be downloaded from his home page: `http://www.engineering.tech.nhl.nl/engineering/personeel/bruin/`

[2]More specifically, we will use the Glasgow Haskell Compiler (GHC), available from `www.haskell.org/ghc`, also available as interactive interpreter (GHCi, version 7.0.x).

of a function `f` to an argument `a`, written as `f a`. That is, the operation of function application is denoted by juxtaposition.

We stress that application (often called "function application") is an operation just like many other well-known operations, like `+`, `*` for operations on numerical values, `>`, `>=` for comparisons, etc. Function application has priority over all other operations, for example: `f 3+5` means `(f 3)+5` and not `f (3+5)`. Parentheses are used to overrule the standard priority rules.

In appendix A a list of standard operations available in Haskell is mentioned (see the documentation on `www.haskell.org` for more)[3].

Definitions. To define `a` as a name for a certain value, include in the script file a definition of the form

```
a = ...
```

where "`...`" stands for some expression. The following definition is a very simple example

```
a = 25+17
```

Functions are also defined in a standard way. For example, a straightforward polynomial function like:w

$$f(x) = 3x^2 + 5x - 4$$

is defined by a line

```
f x = 3*x^2 + 5*x - 4
```

in a script file. After loading this script file into the evaluator, expressions like `f 7` can be evaluated. As expected, the *actual argument* `7` is *substituted* for the *formal parameter* `x` in the definition of `f`, after which the prescribed calculation is performed.

In definitions we have the possibility of *pattern matching*: in the definition of a function the formal parameter may—by its form alone—determine which argument fits into the parameter. For example, let `g` be defined by a definition consisting of three *clauses*:

```
g 0 = 10
g 1 = 20
g x = 3*x
```

[3]It's worthwhile to check these, it can save you a lot of trouble in defining already existing functions.

When g is used in some concrete context, the patterns 0, 1, x in this definition determine which clause is chosen. For example, a call g 1 will choose the second clause, whereas a call g 5 will choose the third clause. Naturally, only the actual argument 0 fits in pattern 0, likewise for 1. Since x is a variable, every actual argument fits in pattern x. Clauses are tried in-order (from the first line downwards), so changing the order of the clauses in this definition may make a certain clause unreachable.

An alternative formulation of g, using *guards*, would be:

```
g x | x == 0     = 10
    | x == 1     = 20
    | otherwise = 3*x
```

Mixtures of these two forms are also possible:

```
g x | x == 0     = 10
    | x == 1     = 20
g x = 3 * x
```

Here, if neither of the guards x==0 and x==1 evaluates to True, GHC continues with the final clause.

Where clauses. Definitions may contain where clauses, as in

```
h x = y + y + y
    where
      y = x^2
```

Clearly, the application h 5 evaluates to 75.

The definition y=x^2 in the where clause is only valid inside the definition of h, and y can not be used from outside this definition, i.e. y is not "visible" from outside the definition. Would we write

```
h 1 = y + y + y      where y = 10
h x = y + y + y      where y = x^2
```

the y in the second clause is completely unrelated to the y in the first, i.e. we could have given them different names altogether.

A where clause is important for readability and for efficiency: it assures that x^2 is evaluated only once, whereas using the functionally equivalent definition

```
h x = x^2 + x^2 + x^2
```

the same subexpression would be evaluated three times[4].

Lay-out. The *lay-out* of definitions is important for Haskell to recognize where a definition begins and ends: the first character of a definition determines the top-left corner of a block that is ended by a character in the same column. Subsequent lines that are part of the same definition should start to the right of this character.

The following code is *wrong*, because the definition of `z` is not part of the where clause in the definition of `h` and, thus, `x` is unknown in the definition of `z`.

```
h x = y + z
  where
   y = 3 * x
z = 5 * x
```

The correct code would be

```
h x = y + z
  where
   y = 3 * x
   z = 5 * x
```

In the last definition there are three blocks: `h` indicates the top-left corner of a block containing the whole definition. The keyword `where` creates the possibility for nested definitions in which local names are defined. In the nested definitions, the position of `y` marks the top-left corner of a nested block, which is ended by the definition of `z`, since `z` is in the same column as `y`. Likewise, `z` starts a new block as well.

Referential transparancy. As in mathematical expressions the value of a variable is the same at all occurrences of that variable inside the clause of a definition, no matter how many lines a clause contains. This is different from imperative programming languages, where the value of a variable may depend on the place where it occurs. Referential transparancy is an important feature when proving that a program is correct and to enable advanced program transformations by the compiler.

[4]Of course, the compiler will try and eliminate common subexpressions, but it can not always do this.

2 Types

Every (correct) expression e has a *type* T, written as

e `::` T

A type can be seen as a *set of values* on which certain *operations* are defined. Then $e::T$ means that the value of e belongs to the set T, and all operations defined on T may be applied to e.

Basic types. There are six basic types in Haskell:

`Bool`	for truth values (`True` and `False`)
`Char`	for character values (`'a'`, `'A'`, `'7'`, `'*'`), including non-printable characters, such as tab (`'\t'`) and escape (`'\esc'`).
`Int`	for 32-bit *signed* integer values (everything between -2147483648, or $-(2^{31})$, and 2147483647, or $2^{31}-1$)
`Integer`	for arbitrarily large integer values
`Float`	for 32-bit floating point values (3.1415927)
`Double`	for 64-bit, i.e. double precision, floating point values (3.141592653589793)

Types may be combined into *compound* types. In this section we discuss three ways to combine types: *tuple types*, *list types*, and *function types.* Later on we discuss other possibilities to define new types (section 5)

Haskell can deal with *type variables* which denote arbitrary types. A type variable is an identifier starting with a lower case letter. Names for specific types should start with a capital letter.

Tuple types. Let `a`, `b`, `c` be arbitrary types, then `(a,b)`, `(a,b,c)`, etc, are the types of *n-tuples*. Expressions of these types are written (respectively) as `(x,y)`, `(x,y,z)`. Thus,

```
(x,y)   :: (a,b)
(x,y,z) :: (a,b,c)
```

where `x::a`, `y::b`, etc. Expressions of this form may be used as patterns.

There are two standard *projection* functions `fst` and `snd` in Haskell, which select the first and the second element of a 2-tuple (respectively)[5]. Thus:

[5]Haskell has no standard projection functions for tuples of more than two arguments.

```
fst (x,y) = x
snd (x,y) = y
```

List types. Let `a` be an arbitrary type, then `[a]` is the type of all *lists* of elements of type `a`. Examples of list expressions of type `[a]` are `[]` (the empty list), and `[x,y,z]` (where `x`, `y` and `z` must all have the same type `a`). Lists may have any length (including infinity), and they are *ordered* from left to right. The same element may occur more than once in a list.

The primitive operation for lists is "`:`" by which an element can be added to the front-end of a list: `x:[y,z] = [x,y,z]`. In fact, `[x,y,z]` is shorthand notation for `x:y:z:[]`, i.e. lists are constructed by "`:`", starting from the empty list[6].

Given an index `i` and a list `xs`, the expression `xs!!i` denotes the `i`-th element of the list `xs`[7]. The first element of a list has index `0`. Of course, `i` should be smaller than the length of `xs`, but not negative.

Lists can be defined by means of *list comprehension*, a set-theory-like way to denote lists. For example, if `xs` is a list of numbers, then

```
[ x^2 | x<-xs , x>0 ]
```

is the list of squares of all positive numbers from `xs`. The expression "`x<-xs`" is called a *generator*, and "`x>0`" a *qualifier*. There may be several of them in a list comprehension, and they are evaluated from left to right (exercise: check the value of `[ (x,y) | x<-xs , y<-ys ]`).

Generators may use patterns. For example, when `ps` is a list of pairs of type `(Int,Int)`, then

```
[ x+y | (x,y) <- ps ]
```

is the list of all totals of all pairs in `ps`.

Lists of characters are called *strings*. There is a special notation for strings, for example, the list `['a','Z','8',' ','$','#']` may be written as `"aZ8␣$#"`.

A final remark about a specific notation for lists:

```
[1..5]     = [1,2,3,4,5]
[10,7..0]  = [10,7,4,1]
[1..]      = [1,2,3,4,5,...
```

[6]As shown in this example, "`:`" is right-associative, i.e. `x : y : []` is the same as `x :( y :[])`

[7]The notation "`xs`" is mnemonic for lists: it denotes English plural (of "x"), and is pronounced as "x-es."

The last expression denotes an infinite list.

This notation works also for `Char` (in general for all types in the class `Enum`[8]):

```
['a'..'e']     = "abcde"
['k','i'..'a'] = "kigeca"
```

List expressions of the form `[]`, `[x,y,z]`, `x:xs` and `"abc"`, may be used as patterns in function definitions.

There are two basic functions for lists: `head` and `tail`. They yield the first element and everything but the first element of the list respectively. That is:

```
head (x:xs) = x
tail (x:xs) = xs
```

Both yield an error when applied to the empty list.

Some further standard operations for lists are `!!` for list indexing (list indexes start at 0), `++` for concatenation and `length` for the length of a list:

```
[5,8,7,4,3]!!2   = 7
[1,2] ++ [4,5,6] = [1,2,4,5,6]
length [5,2,5,3] = 4
```

Lists may be defined *recursively*, i.e. the defined list may be used in the definition itself. For example:

```
ones = 1 : ones
```

yields the infinite list

```
[1,1,1,...
```

We remark that because of lazy evaluation (see section 7) it is possible to include infinite lists into your program. For example,

```
head ones = 1
```

Here, the list `ones` will be evaluated only as far as necessary.

Function types. The type of *functions* from arguments of type `a` to results of type `b` is written as `a->b`. If `f::a->b` and `x::a`, then `f x ::b`. So, for some standard functions mentioned above we have:

[8]See `http://www.haskell.org/hoogle/`

```
fst  :: (a,b) -> a
snd  :: (a,b) -> b
head :: [a] -> a
tail :: [a] -> [a]
```

Notice, that these functions are *polymorphic*, they work for all types `a`, `b`.

Type synonyms. Synonyms for structured types may be introduced by using the keyword `type`. A first example from the standard Prelude is

```
type String = [Char]
```

A somewhat more elaborated example is

```
type Persons = [(String,Int)]
```

which denotes the type of lists of persons, where a person is (implicitly) supposed to be represented by (e.g.) his/her name (`String`) and age (`Int`). Equivalently, we might have written

```
type Person  = (String,Int)
type Persons = [Person]
```

3 More on functions

Recursion. Functions may be defined *recursively*, i.e. the function to be defined may be used in the definition itself. The standard example of a recursive function definition is the factorial function[9] (using pattern matching[10]):

```
fac :: Int -> Int
fac 0     = 1
fac (n+1) = (n+1) * fac n
```

One can look at recursive definitions in two ways: *operationally* and *equationally*. The first way imagines all the computational steps which are performed when calculating a specific application. For example, imagine the actual calculation of `fac 3` in your mind, and compute the intermediate

[9]Since types are of great help in debugging a program, it is good practice to start a function definition with the type of the function.

[10]The pattern `n+1` matches only positive integers, such that the patterns `0` and `n+1` are mutually disjoint. Note that these patterns are not exhaustive: applying `fac` to a negative integer will give an error message.

results `fac 0`, `fac 1`, `fac 2` by multiplying the previous result with the corresponding number. This approach is preferably left to the computer.

The second way of looking is based on the observation that the factorial of $n+1$ simply *is* the product of $n+1$ and the factorial of n. Since in the definition of `fac` above we have that

- there is a clause for the base case `0`,
- in the recursive clause the argument `n` on the right hand side is "simpler" than `n+1` on the left hand side,

and since we may *assume* that `fac n` yields the correct result (compare the *induction hypothesis* from proofs by induction), we conclude that `fac (n+1)` yields the correct result as well[11]. Hence, we may leave the actual *execution* of the computation to the computer, and the programmer may restrict himself to looking at the definition as a set of *equations*.

Functions on lists may also be defined recursively. For example, a function which computes the length of a list (i.e. the number of elements in it), can be defined as follows:

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

Currying. In daily practice, definitions of functions with more than one argument (say three) usually take the form

$$f(x, y, z) = \cdots$$

Using n-tuples and pattern matching, such a definition can be directly translated into a functional language:

```
f (x,y,z) = ...
```

Notice that the type of such an `f` (for an appropriate choice of types `a`, `b`, `c`, `d`) is

```
f :: (a,b,c) -> d
```

In functional programming, functions are often *curried* (after the logician Haskell B. Curry, who is also the language's namesake). The above schema then gets the form

[11]The branch of mathematics that deals with properties like this is called *Recursion Theory*.

```
f x y z = ...
```

and the type of `f` then is

```
f :: a -> b -> c -> d
```

Function application is *left associative*, i.e. `f x y z` means `((f x) y) z`. Because of currying, functions can be applied to fewer arguments than one would expect, e.g. the expression `f x y` denotes a *function* which still has to be applied to an argument `z` before yielding a result of type `d`.

Correspondingly, "`->`" is *right* associative, i.e. `a -> b -> c -> d` stands for `a->(b->(c->d))`. So, for `f` we have (let `x::a`, `y::b`, `z::c`)

```
f       :: a -> b -> c -> d
f x     :: b -> c -> d
f x y   :: c -> d
f x y z :: d
```

For example, define the function `add` as

```
add :: Int -> Int -> Int
add x y = x + y
```

Now,

```
g = add 3
```

is the function which adds 3 to its argument, e.g. `g 39 = 42`. It follows that

```
g :: Int -> Int
```

Equivalently, we might have defined `g` by the following equation:

```
g x = add 3 x
```

However, the first definition is on a higher, and more "abstract" level than the second, and expresses more clearly that a function is an object (or value) in itself.

Sectioning. Functions are written in prefix notation. However, binary functions may be "infixed" by writing it between backwards single quotes (`‘`). For example, with the function `add` from above, this gives

```
40 ‘add‘ 2 = 42
```

Inversely, binary infix operations may be turned into functions (in prefix notation) by *sectioning*, written by parentheses. Consider the expression `3+5`. Now, `(3+)`, `(+5)`, `(+)` are functions and the following expressions all evaluate to `3+5`:

```
(3+) 5
(+5) 3
(+) 3 5
```

A special case is `-`, which denotes both a unary operation (negation) and a binary operation (subtraction). To disambiguate between these two, Haskell's default choice is the binary operation.

Lambda abstraction. Consider the function definition (in mathematical notation)

$$f(x) = 3x^2 + 5x - 4$$

An equivalent definition, again expressing more clearly that a function is an object in its own right, is by so-called *lambda abstraction*[12]:

$$f = \lambda x.\, 3x^2 + 5x - 4$$

Applying f to an argument, e.g. 2, then is evaluated as follows:

$$\begin{aligned} f\ 2 &= (\lambda x.\, 3x^2 + 5x - 4)\ 2 \\ &= 3 * 2^2 + 5 * 2 - 4 \\ &= 18 \end{aligned}$$

Here, the important step is the substitution of 2 for x in the second line. Clearly, this is exactly what we do intuitively when we have to calculate $f(2)$ using the first definition of f above. The advantage of lambda abstraction is that we can use functions in expressions without first introducing names for them.

Lambda abstraction is also possible in Haskell, though the syntax differs slightly[13]. The above lambda term is expressed as

```
f =  \x -> 3*x^2 + 5*x - 4
```

The variable `x` is called the *formal parameter* of the lambda term, the part on the right hand side of the arrow is the *body* of the lambda term, the variable `x` in the body is *bound* by the introduction of `x` on the left hand

[12]In the 1930's the logician Alonzo Church founded the *lambda calculus* as a formal theory about functions. It may be considered as an abstract formulation of the essence of a functional programming language.

[13]GHC does allow unicode input, but it was specifically chosen not to represent lambda abstraction with a λ, because Greek programmers would lose a letter for their variable names.

side of the arrow. A variable in the body which is not bound by the lambda abstraction is said to be *free.*

A lambda term may have a pattern as formal parameter, for example

```
f =  \(x,y) -> x+y
```

defines a function which adds the two elements of an ordered pair. The type of `f` is

```
f :: (Int,Int) -> Int
```

To avoid parentheses, it is agreed that the body of a lambda term is extended to the right as far as possible. Thus, in the following example, all parentheses may be left out without changing the meaning:

```
(\x -> (\y -> (x+y)))
```

However, when used, some brackets may be necessary:

```
(\x -> \y -> x+y) 2 3
```

Note that the type of the outer lambda term is

```
Int -> Int -> Int
```

and the value of this last term is `5`. Finally, Haskell allows a shorthand for this lambda abstraction, by allowing a single lambda to bind multiple variables. In other words, the above lambda abstraction may also be written as

```
(\x y -> x+y) 2 3
```

Higher order functions. Because of currying, functions can have functions as results. There can also be functions which have functions as *parameters.* Furthermore, there can be *lists* of functions. One says that functions are "first class citizens." Functions with functions as parameters or as results, are called *higher order functions.* Two important higher order functions are `map` and `filter` (more are mentioned in appendix A):

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

That is, `map` applies the function `f` to all elements of the list `xs`. For example, two equivalent formulations of adding `5` to all elements of a list are

```
map (add 5)   [1,7,3,10] = [6,12,8,15]
map (\x->x+5) [1,7,3,10] = [6,12,8,15]
```

The first uses the curried function `add`, defined before, the second uses lambda abstraction.

For `filter` we have:

```
filter :: (a->Bool) -> [a] -> [a]
filter p xs = [ x | x<-xs, p x ]
```

That is, `filter` yields the list of all elements of `xs` which satisfy property `p`, i.e. for which `p x = True`. For example, two equivalent formulations of selecting all numbers greater than 5 from a list are

```
filter (>5)        [1,7,3,10] = [7,10]
filter (\x-> x>5) [1,7,3,10] = [7,10]
```

The first uses sectioning, the second uses lambda abstraction.

Function composition. A very important higher order operation for functions is *function composition*, denoted "`.`". Its mathematical definition is as follows:

$$(g.f)\,x = g(f\,x).$$

Since function composition operates on functions without first having to apply these functions to an argument, it offers the possibility to formulate definitions on a higher level of abstraction. For example, let the function `swap` be defined as follows:

```
swap (x,y) = (y,x)
```

Then the following function `sort'` sorts a list of 2-tuples on their second element:

```
sort' = map swap . sort . map swap
```

Note that the definition of `sort'` is purely functional, without mentioning an argument for `sort'`. For example:

```
sort' [(3,5), (3,6), (3,2)]
 =1=>  (map swap . sort . map swap) [(3,5),(3,6),(3,2)]
 =2=>  map swap (sort (map swap [(3,5),(3,6),(3,2)]))
 =3=>  map swap (sort [(5,3),(6,3),(2,3)])
 =4=>  map swap [(2,3),(5,3),(6,3)]
 =5=>  [(3,2),(3,5),(3,6)]
```

Here, on step `1` the definition of `sort'` is applied, and on step 2 the definition of function composition. On steps `3` and `5` the function `swap` is mapped over all elements in the list, and on step `4` the list is sorted by the predefined function `sort`, using the standard ordering on tuples.

Operations and functions. Both operations (`+`, `:`, `.`, etc) and functions (`head`, `div`, etc) can be applied to arguments to yield some result. However, even though we may turn operations into functions (by sectioning), and binary functions into operations (by back quoting), there are important differences between the two. First of all, a function is a value of some (function) type, and an operation is not. To check this, compare the reactions of GHCi to "`:t +`" and "`:t (+)`". Likewise for "`:t div`" and "`:t ‘div‘`".

In general one might say that an operation "glues" expressions together into a bigger expression and determines how the resulting expression has to be evaluated. Remember that also between a function and its argument there is an operation: function application, though that operation is not written explicitly.

In Haskell we may not only define functions, but also operations, including higher order operations. For example, function composition is predefined in Haskell, but we might have defined it ourselves as

```
 f . g = \x -> f (g x)
```

As an example to combine several of the above issues, we write a function that selects from a list of 2-tuples of numbers those pairs whose first element is even, and those pairs whose second element is between 5 and 10, using only one mentioning of `filter`.

First we define two operations that combine properties (i.e. boolean valued functions):

```
p & q = \x -> p x && q x
p # q = \x -> p x || q x
```

Now consider the expression:

```
filter  (((==0).(‘mod‘ 2).fst)  #
                 (((>5).snd) & ((<10).snd)))
```

Exercise. Analyze what the type of this expression is, design an appropiate testing expression for it, predcit what the result of that expression will be, and evaluate it in GHCi.

Recursors. The previously given examples of recursive function definitions have the form of *primitive recursion*, i.e. a function is defined for a basic value (such as `0`, `[]`), and for the "next" value(s) (`n+1`, `x:xs`)[14]. This

[14]There are other forms of recursion, such as partial recursion, or general recursion.

form of recursion can be expressed in a general way by so-called *recursors*. For natural numbers, the (primitive) recursor `recr` is defined as follows:

```
recr f a  0     = a
recr f a (n+1) = f (n+1) (recr f a n)
```

Note that the definition of `recr` is itself primitive recursive.

Using `recr` the factorial function `fac` can be defined as follows:

```
fac = recr (*) 1
```

Now `fac 4` evaluates to `4*(3*(2*(1*1)))` , which evaluates to 24. That is to say, the computation is built up from the right. There is also a left variant, `recl`:

```
recl f a  0     = a
recl f a (n+1) = recl f (f a (n+1)) n
```

Check that, since multiplication is associative and commutative, the definition

```
fac = recl (*) 1
```

leads to the same result, though the order of computation is different.

The same principle can be applied to recursion for lists, leading to the important functions `foldr` ("fold-right") and `foldl` ("fold-left")

```
foldr f a []     = a
foldr f a (x:xs) = f x (foldr f a xs)

foldl f a []     = a
foldl f a (x:xs) = foldl f (f a x) xs
```

To illustrate a somewhat tricky difference between the two, consider the following two definitions of the function `length`:

```
length = foldr g 0   where  g x y = y + 1
length = foldl g 0   where  g x y = x + 1
```

The difference between `foldr` and `foldl` can be shown as follows:

```
foldr f a [b,c,d] = b ‘f‘ (c ‘f‘ (d ‘f‘ a))
foldl f a [b,c,d] = ((a ‘f‘ b) ‘f‘ c) ‘f‘ d
```

This explains the naming "right" and "left".

The variants `foldl1` and `foldr1` take the first and last (respectively) element of the list as starting value, i.e.

```
foldl1 f (x:xs) = foldl f x xs
```

Thus `foldl1`, `foldr1` both give an error message for the empty list.

Exercises. Derive the types of `foldr` and `foldl`.

Define `reverse`, `concat`, `map` and `filter` using `foldr` or `foldl`.

4 Type classes

Many functions are *polymorphic*, they work for any type. For example

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

for all types `a`, `b`. Remember that the definitions of `head` and `map` only use the structural properties of list and function types and not information from the types `a`, `b` themselves.

On the other hand, there are functions that work for a specific *class* of types. For example, relational operations such as "`<`" exist for types such as `Int`, `Char`, `[Int]`, but not for function types `a->b`. Types for which an ordering relation exist belong to the class `Ord`, and an ordering relation such as "`<`" has the following type:

```
(<) :: Ord a => a -> a -> Bool
```

This can be read as follows: "`(<)` has the type `a->a->Bool`, assuming that the type `a` belongs to the class `Ord`".

Likewise, the type of the test for equality is as follows:

```
(==) :: Eq a => a -> a -> Bool
```

These operations are called *overloaded* since they exist for several types, but their implementations may differ for the various concrete types. For example, "`<`" is implemented differently for `Int`s and for `Float`s. Assuming that the concrete definitions of "`==`" and "`<`" are given, several other operations and functions can be defined while defining the class `Ord` itself:

```
class Eq a => Ord a where
  x >  y  =  y < x
  x <= y  =  x<y || x==y
  x >= y  =  x>y || x==y
```

The operations "`>`", "`<=`", "`>=`" are called *methods* since they are valid for every type in the class `Ord`. It is left to te reader to extend this definition of the class `Ord` for methods such as `min` and `max`.

The prefix "`Eq a =>`" means that a type `a` only belongs to class `Ord`, if `a` belongs to class `Eq`. Hence, the class `Ord` is a *subclass* of the class `Eq`, where the class `Eq` itself is defined as follows:

```
class Eq a where
  x /= y  =  not (y == x)
  x == y  =  not (y /= x)
```

The methods in class `Eq` seem to be circular, but the definition of either `==` or `/=` for a concrete type will overrule the method definition in the class itself, and thus break the circularity.

If a type belongs to a class, it is called an *instance* of that class. When defining some type `A` (see section 5), a programmer may declare that `A` is an instance of class `Ord` and at the same time define the elementary relation "`<`" for type `A` as follows:

```
instance Ord A where
  x < y  =  ...
```

All other methods of the class `Ord` now automatically are valid for type `A` as well.

Remember that since `Ord` is a subclass of `Eq`, type `A` should first be declared to be an instance of class `Eq` before `A` will really belong to class `Ord`.

In several cases, methods can be derived by Haskell, and the programmer need not define them himself. For that the keyword "`deriving`" can be used. We will see examples of that below.

There exist many predefined classes in Haskell. Here we only mention the type `Show` of all types whose values can be transformed into strings such that they can be shown on an ascii screen. Most types belong to the class `Show`, but functions typically do not. This class has as a method the function

```
show :: Show a => a -> String
```

which transforms a value of any type `a` into a printable string, whenever type `a` belongs to the class `Show`.

5 Type definitions.

The above mentioned possibility of type synonyms does not introduce new types with new values, it only gives a name to types which could be constructed from given types already. In this section we discuss two possibilities to define new types whose values are also new and explicitly defined by the programmer: *algebraic types* and *record types.* These types can be defined by using the keyword `data`. Newly defined types have to be given a name, called a *type constructor*.

Algebraic types. Values of *Algebraic Data Types* (ADTs) are created by so-called *data constructors.* The basic example is the type `Bool`, which is defined as

```
data Bool = True | False
```

Thus, "`Bool`" is a *type* constructor, whereas "`True`" and "`False`" are *data* constructors. Constructors start with a capital letter. The symbol "`|`" stands for "or".

More generally, constructors may take types as arguments, for example the type

```
data BasicType = I Int | B Bool | C Char
```

contains *values* like `I 5`, `I 42`, `B True`, `C 'a'` and `C '7'`, which all have type `BasicType`. The values of this type might be called "labeled," where the data constructors `I`, `B`, `C` may be seen as the labels. Labeled values may be used as patterns in definitions.

ADTs may be *parameterized*, i.e. also a type constructor may have types as arguments:

```
data PQtype a = P a | Q a
```

For example, the types `PQtype Int` and `PQtype Char` are concrete types of this general form, and we have that

```
P 25    :: PQtype Int
P 'a'   :: PQtype Char
Q 'a'   :: PQtype Char
Q True  :: PQtype Bool
```

We remark that `P 25` and `Q 25` are different values of `PQtype Int`.

As mentioned above, the type `PQtype a` may be declared by the programmer to belong to a given class by using the keyword `instance`. For the class `Eq` this can be done as follows:

```
instance Eq a => Eq (PQtype a) where
   P x == P y   =   x == y
   Q x == Q y   =   x == y
   _   ==   _   =   False
```

Since equality on type `PQtype a` is defined in terms of equality on the types `a`, the type `a` has to be an instance of the class `Eq` as well. Note, however, that the programmer is free to choose an alternative definition for equality.

The same can be done for the classes `Ord` and `Show`:

```
instance Ord a => Ord (PQtype a) where
   P x < P y   =   x < y
   Q x < Q y   =   x < y
   P _ < Q _   =   True
   Q _ < P _   =   False

instance Show a => Show (PQtype a) where
   show (P x)   =   "P␣" ++ show x
   show (Q x)   =   "Q␣" ++ show x
```

Here the programmer chose to let `P`-terms be smaller than `Q`-terms (since in the definition of the type `PQtype a` the data constructor `P` precedes the data constructor `Q`), but of course, here too he might have made another choice.

In practice the above schedule occurs very often, and therefore Haskell has the possibility to derive standard classes already in the definition of the type itself, using the keyword `deriving`:

```
data PQtype a = P a | Q a  deriving (Eq, Ord, Show)
```

Here it is implicitly assumed that the type `a` belongs to the classes `Eq`, `Ord` and `Show`. Haskell's choice for defining `==`, `<` and `show` is as in the above instance declarations.

Recursive types. ADTs may be used recursively, for example

```
data Tree = Leaf Int | Node Char Tree Tree
```

contains a.o. the following values:

```
Leaf 5
Node 'a' (Leaf 3) (Leaf 10)
Node 'b' (Node 'a' (Leaf 3) (Leaf 10)) (Leaf 5)
```

Intuitively, these expressions denote tree-structures (hence the name `Tree`) with numbers at the leaves, and characters at the internal nodes. Furthermore, in trees of type `Tree` every internal node has two subtrees, i.e., the type `Tree` contains binary trees.

In fact, natural numbers (`Nat`) and lists of elements of type `a` (`List a`) might also be defined as recursive types:

```
data Nat    = Zero | Succ Nat
data List a = Nil  | Cons a (List a)
```

Remember that in the final example, the type `a` stands for an arbitrary type. Clearly, the type `List a` is isomorphic to the predefined type `[a]`.

Functions on recursive types may be defined recursively, i.e., by using the recursive structure of the type definition. For example, the length of a list of type `List a` can be defined as follows:

```
length Nil         = 0
length (Cons x xs) = 1 + length xs
```

When the function `length` is used, the variable `x` in the pattern `Cons x xs` binds to a value of type `a`, whereas the variable `xs` binds to a value of type `List a`. Thus, this value of `xs` again is of the form `Nil`, or `Cons y ys`.

Exercise. Define several standard functions on `Nat` and `List a`. For example, define the arithmetical functions (addition, multiplication, etc.) for the type `Nat`, and the list functions (`head`, `reverse`, `map`, etc.) for the type `List a`.

Record types. Remember the type `Person` on page 8, defined as a type synonym for the 2-tuple `(String,Int)`. In that way of defining the type `Person` it is implicit that the string and the integer are supposed to denote the name and the age (resepctively) of the person. An alternative way to define a type for persons in which name and age are explicit, is by defining a *record type* as follows:

```
data Person  = Pers { name::String, age::Int }
```

Thus, this definition consists of the keyword `data`, mentions (`name`, `age`) and their types between curly brackets, and labels it with a data constructor (`Pers`). The data constructor may or may not be the same as the *type* constructor (here `Person`). There is no limitation on the number of fields, a field name may be any identifier (starting with a lower case letter) and types may be chosen freely.

Clearly, the list of persons still can be defined by a type synonym:

```
type Persons = [Person]
```

A concrete record of type `Person` may now be defined as

```
p0 = Pers { name="Bill", age=25 }
```

In the type `Person` the *field names* can extract the values by using them as *functions*:

```
name p0  ==>  "Bill"
age  p0  ==>  25
```

Correspondingly, their types are

```
name :: Person -> String
age  :: Person -> Int
```

Record fields may be given different values by *record updating*:

```
p0{age=35}                 ==>  Pers {name="Bill" , age=35}
p0{age=35, name="Chris"}  ==>  Pers {name="Chris", age=35}
```

Note that the order in which the field names are listed does not matter, though GHC shows them in the order as given in the type definition.

As suggested by the definition of the type Person, records indeed are an extension of algebraic data types, and we may include several records within the same type:

```
data RecTypes
      = Employee { address :: String, salary :: Int }
      | Supplier { address :: String, account :: Int }
```

Note that both record variants contain a field with name address. That is possible, as long as the content type of the field address is the same, in this case String, such that its type is well-defined:

```
address :: RecTypes -> String
```

As with algebraic data types, record types may be declared to be instances of certain classes by the keyword instance, and as before, several standard classes may be derived by the keyword deriving. Thus, after

```
data Person  = Pers { name::String, age::Int }
               deriving (Eq,Ord,Show)
```

values of type Person may be compared with equality and ordering relations, and they may be shown on an ascii screen. The result is as expected, with the remark that ordering relations look at the values of the fields in the order that these fields are introduced in the type definition.

Records may be partial and neither order, nor lay-out are important, so

```
p1 = Pers {name="Chris"}
p2 = Pers {age=35, name="Chris"}
```

both are (partial) values of type Person.

When the content of filed `age` of record `p1` is needed in a computation, an exception will arise. But as long as it is not needed, there is no problem (for an explanation, see section 7 on "lazy evaluation").

Records (also partially) may be used as *patterns*, for example

```
isAdult (Pers {age=x})  =  x >= 18
```

Note that the *structure* of this record pattern is indicated by the part `Pers {age=...}`, whereas `x` is a variable, i.e. a "nested" pattern which "catches" the value of the corresponding field. The names of these variables can be chosen freely – *including* `age` itself[15]. That is to say, the following definition is equivalent to the above one:

```
isAdult (Pers {age=age})  =  age >= 18
```

6 Modules.

Definitions in a script file may be packed in a module, and modules may be imported in other script files. A module has to start with a line like (below some variants of this format will be discussed)

```
module ModName where
```

followed by the definitions in the module. The keyword `module` starts with a lowercase letter, whereas the name of the module starts with a capital letter. The name of the file containing the module should have the same name as the module itself.

A module may be imported in another file by including the line

```
import ModName
```

after which the functions and types from the module `ModName` may be used.

An important role for modules is to *hide* the definitions of types and/or functions. In order to make functions and types from a module known in a file where the module is imported, they have to be *exported* explicitly from the module by mentioning them in the heading of the module as follows:

```
module ModName ( Type1
               , Type2(A,B)
               , Type3(..)
```

[15]In fact, there is a wealth of possibilities in using patterns and updating fields in Haskell, but for that we refer to the GHC Manual, especially chapter 8. See `http://www.haskell.org/ghc/docs/latest/html/users_guide/index.html`

```
                  , fun1
                  , fun2
                  , ...
                  ) where
```

Here, the types `Type1`, `Type2`, `Type3`, and the functions `fun1`, `fun2` are known from outside. However, none of the constructore of `Type1` will be known outside the module, whereas from `Type2` only constructors `A` and `B` will be known. The notation "`..`" means that *all* constructors from `Type3` are exported.

Even though a type or function is exported in this way, it still can be made inaccessible from the script file where the module is imported by saying, e.g.,

```
import ModName hiding fun1
```

This is useful in case the programmer wants to define a function with the same name as `fun1`.

On the othe rhand, it is possible to indicate which function is meant by prefixing it with its module:

```
ModName.fun1
```

indicates the function `fun1` from module `ModName`.

As an example, consider the following module for sets:

```
module Set (Set, empty, add, union, member) where

data Set a = S [a]  deriving Eq

instance Show a => Show (Set a) where
  show (S x) = show x

empty :: Set a
empty = S []

add :: Eq a => a -> Set a -> Set a
add x (S xs) = S (nub (x:xs))

union :: Eq a => Set a -> Set a -> Set a
union (S xs) (S ys) = S (nubp (xs++ys))

member :: Eq a => a -> Set a -> Bool
member x (S xs) = elem x xs

nub :: Eq a => [a] -> [a]
nub  []    = []
nub (x:xs) = x : nub (filter (/=x) xs)
```

First of all, note that the only way to have access to elements of type `Set a` is by means of the exported functions. In general, a programmer will know their types and their semantics, but not their precise definitions.

Internally in the module the type `Set a` has only one constructor `S` which "packs" lists into sets. Note that the constructor `S` is not exported, so a programmer can not use it. Furthermore, since the function `show` is redefined in such a way that the constructor `S` is not shown, a programmer using this module will not see the constructor `S` at all.

Since sets do not contain duplicates, the function `nub` removes those. Note that the function `nub` is not usable from outside since it is not exported.

7 Lazy evaluation

An expression may be evaluated in various orders. For example, let

```
f x = x * x,
```

and evaluate `f (3+4)`. Then there are three possibilities:

```
f (3+4) => f 7             => 7 * 7     => 49
f (3+4) => (3+4) * (3+4) => 7 * (3+4) => 7 * 7 => 49
f (3+4) => (3+4) * (3+4) => (3+4) * 7 => 7 * 7 => 49
```

The first is called *eager evaluation* (argument first), the second *leftmost-outermost* evaluation. The third is a mixed form and has no name. Since the argument is only evaluated once, the first strategy seems to be more efficient. Nevertheless, Haskell, like many other functional languages, chooses the second possibility (extended with sharing, see below). The advantage is that the final answer will be found if there exists one. For example, define

```
from n = n : from (n+1)
```

Then

```
from 3 = [3,4,5,...
```

Evaluate

```
head (tail (from 3))
```

Then first fully evaluating the argument `from 3` would give an infinite reduction and thus no answer at all. Following the leftmost-outermost strategy one gets the answer 4:

```
head (tail (from 3))
     =1=>  head (tail (3 : from (3+1)))
     =2=>  head (from (3+1))
     =3=>  head ((3+1) : from ((3+1)+1))
     =4=>  3+1
     =5=>  4
```

Note that an argument is only evaluated insofar it is needed to evaluate an expression which contains the argument as a part. Thus, in step 1, `g 3` has to be evaluated only one step, such that the rule for `tail` can be applied in step 2. In order to apply the rule for `head` (step 4), again `g` has to be applied one step first (step 3). Note that only one addition is performed (step 5).

To repair the possible loss in efficiency by computing the same expression over and over again, *sharing* is applied, i.e. when an argument is substituted for a formal parameter, every evaluation step in one of the copies of the argument is executed in all its copies. In the first example above, this would mean that `(3+4)` would be evaluated only once:

```
f (3+4) => (3+4) * (3+4) => 7 * 7 => 49
```

Leftmost-outermost reduction with sharing is called *lazy evaluation*.

In Haskell, there is one[16] function which gives some control over the order of evaluation: `seq`. The function `seq` first evaluates its first argument, but delivers the second:

```
seq (3+4) f (3+4) => seq 7 f (3+4) => f (3+4) => ...
```

This example only shows the idea of the function `seq`, but not its usefulness.

8 Some mixed topics

Debugging. Haskell has a few standard facilities for debugging, the simplest of which is the module `Debug.Trace`, which must be imported in order to use the function

```
trace :: String -> a -> a
```

An expression like

[16]Actually, GHC has a few more, but they are non-standard.

```
trace a b
```

prints `a` on the standard output, and continues evaluates to `b`.

Error. We conclude this section on functions with mentioning the special polymorphic function

```
error :: String -> a
```

The expression

```
error "this␣is␣an␣error␣message"
```

may be used in any function definition. Evaluating it causes the program to terminate after the error message is printed.

Appendix A: Some standard operators and functions

Below some important operations and functions predefined in Haskell. See the on-line help function for more, and for a more extensive description. Definitions can be found under `installation | initialisation` in Haskell-help.

`negate, abs,` `signum`	:: `Num a => a -> a`
`+, -, *`	:: `Num a => a -> a -> a`
`div, mod`	:: `Integral a => a -> a -> a`
`/`	:: `Fractional a => a -> a -> a`
`^`	:: `(Num a, Integral b) => a -> b -> a`
`abs, exp, log,` `sqrt, sin, cos`	`Floating a => a -> a`
	various arithmetical operations and functions
`min, max`	:: `Ord a => a -> a -> a`
	gives the minimum, maximum of two arguments
`not`	:: `Bool -> Bool`
`&&, ||`	:: `Bool -> Bool -> Bool`
	boolean operations negation, conjunction, disjunction
`isLower,` `isUpper`	:: `Char -> Bool`
	says whether a letter is lower-case or upper-case
`isAlpha`	:: `Char -> Bool`
	says whether a character is a letter
`isDigit`	:: `Char -> Bool`
	says whether a character is a digit
`isAlphaNum`	:: `Char -> Bool`
	says whether a character is a letter or a digit
`ord`	:: `Char -> Int`
	converts a character to its Unicode number
`chr`	:: `Int -> Char`

	converts a Unicode number to the corresponding character
`toLower,`	
`toUpper`	:: `Char -> Char`
	converts a letter to lower-case, upper-case
`==, /=`	:: `Eq a  => a -> a -> Bool`
`>, >=,`	
`<, <=`	:: `Ord a => a -> a -> Bool`
	various comparison operations
`even, odd`	:: `Integral a => a -> Bool`
	says whether a (integral) number is even or odd
`:`	:: `a -> [a] -> [a]`
	adds element to the front end of a list (*cons*)
`length`	:: `[a] -> Int`
	length of a list
`!!`	:: `[a] -> Int -> a`
	list indexing
`++, \\`	:: `[a] -> [a] -> [a]`
	list concatenation, list subtraction
`head, last`	:: `[a] -> a`
`tail, init,`	
`reverse`	:: `[a] -> [a]`
`elem`	:: `Eq a => a -> [a] -> Bool`
	tests whether a list contains a given element
`concat`	:: `[[a]] -> [a]`
	concats a list of lists into one list
`sort`	:: `Ord a => [a] -> [a]`
`sum`	:: `Num a => [a] -> a`
`minimum,`	
`maximum`	:: `Ord a => [a] -> a`
`take, drop`	:: `Int -> [a] -> [a]`
`takeWhile,`	
`dropWhile`	:: `(a->Bool) -> [a] -> [a]`
	various functions on lists

`insert`	:: `Ord a => a -> [a] -> [a]`	inserts an element into an ordered list
`and, or`	:: `[Bool] -> Bool`	yields the conjunction, disjunction of a list of booleans
`lines`	:: `String -> [String]`	breaks a string at newlines (`'\n'`) into a list of strings
`unlines`	:: `[String] -> String`	glues a list of strings with `'\n'`
`fst`	:: `(a,b) -> a`	yields the first element of a pair
`snd`	:: `(a,b) -> b`	yields the second element of a pair
`zip`	:: `[a] -> [b] -> [(a,b)]`	turns two lists into a list of pairs
`zipWith`	:: `(a->b->c) -> [a] -> [b] -> [c]`	zips two lists and applies a function to the corresponding elements
`map`	:: `(a->b) -> [a] -> [b]`	applies a function to all elements in a list
`filter`	:: `(a->Bool) -> [a] -> [a]`	selects those elements from a list which satisfy a property
`foldl`	:: `(a->b->a) -> a -> [b] -> a`	"folds" a list with a function, starting with a given value. Works from left to right through the list
`foldr`	:: `(a->b->b) -> b -> [a] -> b`	like `foldl`, but works from right to left
`foldl1`, `foldr1`	:: `(a->a->a) -> [a] -> a`	like `foldl`, `foldr`, with first, last element of the list as starting value. Error for empty list
`.`	:: `(b->c) -> (a->b) -> (a->c)`	

	function composition
`seq`	:: `a -> b -> b` partially evaluates first argument, and delivers the second
`error`	:: `String -> a` causes error with given string as error message

Literature

There are a few good websites where lots of information on functional programming can be found. For example:

```
http://www.haskell.org
http://www.cs.kun.nl/~clean
```

From these sites both tutorials and implementations can be downloaded. Some books:

Bird, R., P. Wadler, *Introduction to Functional Programming*, Prentice Hall, London, 1988

Bird, R., *Introduction to Functional Programming Using Haskell* (second edition), Prentice Hall, London, 1998

Davie, A.J.T., *An Introduction to Functional Programming Systems Using Haskell*, Cambridge UP, Cambridge, 1992

Hudak, P., *The Haskell School of Expression*, Cambridge UP, Cambridge, 2000

Hutton, G., *Programming in Haskell*, Cambridge UP, Cambridge, 2007

Koopman, P., R. Plasmeijer, M. van Eekelen, S. Smetsers, *Functional Programming in Clean*, Nijmegen, 2001 (freely available from `http:\\www.cs.kun.nl/~clean`)

Rabhi, F., G. Lapalme, *Algorithms, a Functional Programming Approach*, Addison-Wesley, Harlow, 1999

Thompson, S., *Miranda, the Craft of Functional Programming*, Addison-Wesley, Harlow, 1995

Thompson, S., *Haskell, the Craft of Functional Programming*, Addison-Wesley, Harlow, 1996