

Functional Programming

2021-2022

Sjaak Smetsers

Types and polymorphism
Lecture 2

Outline

- defining functions
- strong typing
- simple types and Enumerations
- functions
- tuples
- polymorphism
- type synonyms
- type classes
- programming example
- type inference
- type-driven program development

The anatomy of functions

```
insert :: Ord a => a -> [a] -> [a]
insert a [ ]      = [a]
insert a (b : xs)
  | a ≤ b         = a : b : xs
  | otherwise     = b : insert a xs
```

type

equations

- function definition consists of
 - a type (optional)
 - one or more equations (alternatives)
- each equation consists of:
 - the function name
 - zero or more parameters
 - variable, wildcard, or a pattern
 - guard (optional)
 - an expression that constitutes the result

Defining functions

- Haskell supports two different programming styles
 - declaration style (emphasis on the left-hand side of equation)
 - expression style (emphasis on the right-hand side of equation)
- expression style is often more flexible
- declaration style often improves the readability of your program
- experienced programmers use both simultaneously
 - also a matter of taste...

Conditional definitions

- definition of *smaller* using *conditional expression* (expression style):

`smallest :: Integer → Integer → Integer`

`smallest x y = if x ≤ y then x else y`

- could also use *guarded equations* (declaration style):

`smallest :: Integer → Integer → Integer`

`smallest x y`

`| x ≤ y = x`

`| otherwise = y`

- each *clause* has a *guard* and an *expression* separated by =
- last guard can be *otherwise* (synonym for *True*)

Pattern matching

- define function by several equations
- arguments on lhs not just variables, but *patterns*
 - patterns may be *variables* or *constants* (or *constructors*, later)
- e.g.

```
day :: Integer → String
day 1 = "Saturday"
day 2 = "Sunday"
day _ = "Weekday"
```

- also *wild-card pattern* _
- a function with 2 arguments

```
(&&) :: Bool → Bool → Bool
False && False = False
False && True  = False
True  && False = False
True  && True  = True
```

```
(&&) :: Bool → Bool → Bool
(&&) False False = False
(&&) False True  = False
(&&) True  False = False
(&&) True  True  = True
```

Local definitions

- repeated sub-expression can be captured in a *local definition*

```
sqroots :: (Float,Float,Float) → (Float,Float)
```

```
sqroots (a,b,c) = ((-b-sd)/(2*a), (-b+sd)/(2*a))
```

```
  where sd = sqrt (b*b - 4*a*c)
```

- scope of **where** clause extends over whole right-hand side
- multiple local definitions can be made:

```
demo :: Integer → Integer → Integer
```

```
demo x y = (a + 1) * (b + 2)
```

```
  where a = x - y
```

```
        b = x + y
```

layout rule for multiple definitions: new definition must begin in the same column

let-expressions

- definitions local to an expression

`demo` :: `Integer` → `Integer` → `Integer`

`demo` x y = **let** a = x - y

 b = x + y

in (a + 1) * (b + 2)

- *declaration style: **where**; expression style: **let ... in ...***

Lambda expressions

- notation for anonymous functions
- e.g. $\lambda x \rightarrow x * x$ as another way of writing **square**
- x is the formal parameter; $x * x$ is the body of the function
- symbol λ represents the Greek letter *lambda*, written as λ
- can be used in the same way as other functions

```
>>> (\x -> x + x) 2  
4
```

- declaration style:

```
quad :: Integer → Integer  
quad x = square x * square x
```

- expression style using lambda expressions

```
quad :: Integer → Integer  
quad = \x → square x * square x
```

Operator sections

- functions such as `+` that are written between their two arguments are called (binary) *operators*
- in Haskell any prefix function with two arguments can be converted into an infix operator, and vice versa.
- e.g. `7 `div` 2`, and `(+) 1 2`
- moreover, the latter also allows one of the arguments to be included in the parentheses if desired, as in `(1 +) 2` and `(/ 2) 4`.
- in general, if \odot is an operator, then expressions of the form $(x \odot)$, and $(\odot y)$ for arguments x and y are called *sections*.
- more formally using lambda expressions:

$$(\odot) = \lambda x \rightarrow (\lambda y \rightarrow x \odot y)$$

$$(x \odot) = \lambda y \rightarrow x \odot y$$

$$(\odot y) = \lambda x \rightarrow x \odot y$$

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

Strong typing

- Haskell is *strongly typed*: every expression has a type
- each type supports certain operations, which are meaningless on other types
- type checking guarantees that type errors cannot occur
- Haskell is *statically typed*: type checking occurs compile-time (before run-time)
- Haskell can *infer types*: determine the most general type of each expression
- wise to specify (some) types anyway, for documentation, and as starting point for definition

Simple types

- booleans
- characters
- strings
- numbers

Booleans

- type **Bool** (note: type names capitalized)
- two constants, **True** and **False** (note: constructor names capitalized)
- definition by pattern-matching

not :: **Bool** → **Bool**

not False = True

not True = False

- and **&&**, or **||** (both non-strict in second argument: $a \neq 0 \ \&\& \ b/a > 1$)

(&&) :: **Bool** → **Bool** → **Bool**

False **&&** _ = False

True **&&** b = b

Boole design pattern

- types come with a pattern of definition
- task: define a function $f :: \text{Bool} \rightarrow S$
- step 1: solve the problem for **False**
 $f \text{ False} = \dots$
- step 2: solve the problem for **True**
 $f \text{ False} = \dots$
 $f \text{ True} = \dots$
- (exercise: define your own conditional)

Characters

- type **Char**
- constants in single quotes: `'a'`, `'?'`
- special characters escaped: `'\''`, `'\n'`, `'\\'`
- ASCII coding: `ord :: Char → Int`, `chr :: Int → Char`
(defined in library module `Data.Char`)
- comparison and ordering, as usual

Strings

- type **String**
- (actually defined in terms of **Char**, see later)
- constants in double quotes: **"Hello"**
- comparison and (lexicographic) ordering
- concatenation **++**
- display function **show :: Integer → String** (actually more general than this; see later)

Numbers

- fixed-precision (in Haskell 64-bit) integers **Int**
- arbitrary-precision integers **Integer**
- single- and double-precision floats **Float**, **Double**
- others too: rationals, complex numbers, . . .
- comparisons and ordering
- **+**, **-**, *****, **^**
- **abs**, **negate**
- **/**, **div**, **mod**, **quot**, **rem**
- etc
- numeric literals and operations are overloaded (more later)

Enumerations

- mechanism for declaring new types

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

- e.g. **Bool** is not built in (although **if ... then ... else** syntax is):

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

- types may even be parameterized and recursive! (more later)

Functions

- naturally, FP is a matter of functions
- function types: e.g. $\text{Char} \rightarrow \text{Int}$
- $X \rightarrow Y \rightarrow Z$ is shorthand for $X \rightarrow (Y \rightarrow Z)$
- values in a similar syntax: $\backslash c \rightarrow \text{ord } c - \text{ord } '0'$
- recall: c is the formal parameter; $\text{ord } c - \text{ord } '0'$ is the body of the function
- $\backslash x \ y \rightarrow e$ is shorthand for $\backslash x \rightarrow \backslash y \rightarrow e = \backslash x \rightarrow (\backslash y \rightarrow e)$
- function application: $f \ x$ (“space operator”)
- $f \ x \ y$ is shorthand for $(f \ x) \ y$

Tuples

- pairing types: e.g. `(Char, Integer)`
- values in the same syntax: `('a', 440)`
- selectors `fst`, `snd`
- definition by pattern matching:
`fst (x, _) = x`
- nested tuples: `(Integer, (Char, Bool))`
- triples, etc: `(Integer, Char, Bool)`
- nullary tuple `()`; type with a single value in the same syntax: `()`
- comparisons and (lexicographic) ordering

Polymorphism

- what is the type of `fst`?
- applicable at different types: `fst (1,2)`, `fst ('a',True)`, ...
- what about strong typing?
- `fst` is *polymorphic* —it works for *any* type of pairs:
- `fst :: (a,b) → a`
- `a, b` here are *type variables* (uncapitalized)

Find the function...

- here is a little game: I give you a type, you give me a function of that type
 - $\text{Int} \rightarrow \text{Int}$
 - $a \rightarrow a$
 - $(\text{Int}, \text{Int}) \rightarrow \text{Int}$
 - $(a, a) \rightarrow a$
 - $(a, b) \rightarrow a$
 - $\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$
 - $(\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int}$
 - $a \rightarrow (a \rightarrow a)$
 - $(a \rightarrow a) \rightarrow a$
- polymorphic functions: flexible to use, “hard” to define
- polymorphism is a property of an algorithm: same code for all types

Type synonyms

- alternative names for types
- brevity, clarity, documentation
- e.g.

```
type Card    = (Rank,Suit)
type Name    = String
type Date    = (Int,Int,Int)
type Person  = (Name,Date)
```

- cannot be recursive
- just a 'macro': no new type

Record types

- like tuples, a **record type** collects expressions that do not have to have the same type
- a record type identifies its components via a **field name**
- a record type introduces a new type (which can be recursive)

```
data Date      = Date      { year :: Int, month :: Int, day :: Int }  
data Parents   = Parents   { mother :: Person, father :: Person }  
data Person    = Person    { name :: Name, birth :: Date, parents :: Parents }
```

- fields can be used as selectors, i.e.

```
year :: Date → Int
```

- creating a record:

```
>>> date = Date 2020 9 9  
>>> date  
Date {year = 2020, month = 9, day = 9}
```

- updating a record:

```
>>> date { day = 11 }  
Date {year = 2020, month = 9, day = 11}
```

Type classes

- what about numeric operations?
 $(+) :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer}$
 $(+) :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float}$
- cannot have $(+) :: a \rightarrow a \rightarrow a$ (too general)
- the solution is *type classes* (sets of types)
- e.g. the type class **Num** is a set of numeric types; includes **Integer**, **Float**, etc
- now $(+) :: (\text{Num } a) \Rightarrow (a \rightarrow a \rightarrow a)$
- *ad-hoc polymorphism* (different code for different types), as opposed to *parametric polymorphism* (same code for all types)

Some standard type classes

type class	functions/operators	instances
<code>Eq</code>	<code>==</code> , <code>/=</code>	<code>Integer</code> , <code>Int</code> , <code>Double</code> , <code>String</code> etc
<code>Ord</code>	<code><</code> etc, <code>min</code> etc	<code>Integer</code> , <code>Int</code> , <code>Double</code> , <code>String</code> etc
<code>Enum</code>	<code>succ</code> etc	<code>Integer</code> , <code>Int</code> , <code>Char</code> , <code>Double</code> etc
<code>Bounded</code>	<code>minBound</code> , <code>maxBound</code>	<code>Int</code> , <code>Char</code> , <code>Bool</code> etc
<code>Show</code>	<code>show</code>	<code>Integer</code> , <code>Int</code> , <code>Char</code> , <code>Bool</code> , <code>[a]</code> etc
<code>Read</code>	<code>read</code>	<code>Integer</code> , <code>Int</code> , <code>Char</code> , <code>Bool</code> , <code>[a]</code> etc
<code>Num</code>	<code>+</code> , <code>*</code> etc	<code>Integer</code> , <code>Int</code> , <code>Double</code> , <code>Float</code>
<code>Integral</code>	<code>div</code> etc	<code>Integer</code> , <code>Int</code>
<code>Fractional</code>	<code>/</code> etc	<code>Double</code> , <code>Float</code>
<code>Floating</code>	<code>exp</code> , <code>log</code> , <code>sin</code> etc	<code>Double</code> , <code>Float</code>

more later

Programming example: a pyramid of strings

write a function

`pyramid :: String → String`

such the application

`pyramid "Functional programming is fun"`

produces the output as shown on the left

```
Functional programming is fun
unctional programming is fu
nctional programming is f
ctional programming is
tional programming is
ional programming i
onal programming
nal programming
al programmin
l programmi
programm
program
rogra
ogr
g
```

Programming example: a pyramid of strings

```
pyramid :: String → String
pyramid xs = pyramid' 0 xs
```

```
spaces :: Int → String
spaces n = replicate n ' '
```

```
pyramid' :: Int → String → String
pyramid' n xs
  | length xs < 2 = spaces n ++ xs
  | otherwise     = spaces n ++ xs ++ "\n" ++ pyramid' (n+1) (init (tail xs))
```

predefined

Functional programming is fun
functional programming is fu
nctional programming is f
ctional programming is

functional programming is

functional programming i

functional programming

functional programming

functional programming

functional programming

functional programming

functional programming

functional programming

functional programming

functional programming

Type inference

- Type inference is a kind of logical puzzle
- Rules:
 - every (sub)expression has a type
 - in a function the same argument has the same type everywhere
 - all alternatives of a function have the same type

```
pyramid xs = pyramid' 0 xs
```

```
spaces n = replicate n ' '
```

```
pyramid' n xs
```

```
| length xs < 2 = spaces n ++ xs
```

```
| otherwise     = spaces n ++ xs ++ "\n" ++ pyramid' (n+1) (init (tail xs))
```

Type inference (cont.)

- How do you solve this puzzle?
 - start with a general type for each function
 - use patterns, guards and right-hand sides to derive more specific type information
 - Before typing a function `f`, examine all functions used by `f` first.

```
pyramid xs = pyramid' 0 xs
```

```
spaces n = replicate n ' '
```

```
pyramid' n xs  
| length xs < 2 = spaces n ++ xs  
| otherwise     = spaces n ++ xs ++ "\n" ++ pyramid' (n+1) (init (tail xs))
```

Type inference

```
pyramid xs = pyramid' 0 xs
```

```
spaces :: ① → ②
```

```
spaces n = replicate n ' '
```

```
pyramid' n xs
```

```
| length xs < 2 = spaces n ++ xs
```

```
| otherwise     = spaces n ++ xs ++ "\n" ++  
                  pyramid' (n+1) (init (tail xs))
```

Beware: these are
not the real types of
these functions

```
replicate :: Int → Char → String
```

```
init, tail :: String → String
```

```
length :: String → Int
```

```
(++) :: String → String → String
```

```
(+) :: Int → Int → Int
```

spaces

① = ?

② = ?

Type inference

```
pyramid xs = pyramid' 0 xs
```

```
spaces :: Int → ❷
```

```
spaces n = replicate ❶ ' '
```

```
pyramid' n xs
```

```
| length xs < 2 = spaces n ++ xs
```

```
| otherwise     = spaces n ++ xs ++ "\n" ++  
                  pyramid' (n+1) (init (tail xs))
```

```
replicate :: Int → Char → String
```

```
init, tail :: String → String
```

```
length :: String → Int
```

```
(++) :: String → String → String
```

```
(+) :: Int → Int → Int
```

spaces

❶ = Int

❷ = ?

Type inference

```
pyramid xs = pyramid' 0 xs
```

```
spaces :: Int → String
```

```
spaces n = replicate n ' '
```

```
pyramid' n xs
```

```
| length xs < 2 = spaces n ++ xs
```

```
| otherwise     = spaces n ++ xs ++ "\n" ++  
                  pyramid' (n+1) (init (tail xs))
```

```
replicate :: Int → Char → String
```

```
init, tail :: String → String
```

```
length :: String → Int
```

```
(++) :: String → String → String
```

```
(+) :: Int → Int → Int
```

spaces

① = Int

② = String

Type inference

```
pyramid xs = pyramid' 0 xs
```

```
spaces :: Int → String  
spaces n = replicate n ' '
```

```
pyramid' :: ① → ② → ③
```

```
pyramid' n xs
```

```
| length xs < 2 = spaces n ++ xs
```

```
| otherwise     = spaces n ++ xs ++ "\n" ++  
                  pyramid' (n+1) (init (tail xs))
```

```
replicate :: Int → Char → String  
init, tail :: String → String  
length :: String → Int  
(++) :: String → String → String  
(+) :: Int → Int → Int
```

```
pyramid'  
① = ?  
② = ?  
③ = ?
```

Type inference

```
pyramid xs = pyramid' 0 xs
```

```
spaces :: Int → String
```

```
spaces n = replicate n ' '
```

```
pyramid' :: ① → String → ③
```

```
pyramid' n xs
```

```
| length xs < 2 = spaces n ++ xs
```

```
| otherwise     = spaces n ++ xs ++ "\n" ++  
                  pyramid' (n+1) (init (tail xs))
```

```
replicate :: Int → Char → String
```

```
init, tail :: String → String
```

```
length :: String → Int
```

```
(++) :: String → String → String
```

```
(+) :: Int → Int → Int
```

```
pyramid'
```

```
① = ?
```

```
② = String
```

```
③ = ?
```

Type inference

```
pyramid xs = pyramid' 0 xs
```

```
spaces :: Int → String  
spaces n = replicate n ' '
```

```
pyramid' :: Int → String → ③
```

```
pyramid' n xs  
  | length xs < 2 = spaces n ++ xs  
  | otherwise     = spaces n ++ xs ++ "\n" ++  
                    pyramid' (n+1) (init (tail xs))
```

```
replicate :: Int → Char → String  
init, tail :: String → String  
length :: String → Int  
(++) :: String → String → String  
(+) :: Int → Int → Int
```

```
pyramid'  
① = Int  
② = String  
③ = ?
```

Type inference

```
pyramid xs = pyramid' 0 xs
```

```
spaces :: Int → String  
spaces n = replicate n ' '
```

```
pyramid' :: Int → String → String
```

```
pyramid' n xs  
  | length xs < 2 = spaces n ++ xs  
  | otherwise     = spaces n ++ xs ++ "\n" ++  
                    pyramid' (n+1) (init (tail xs))
```

```
replicate :: Int → Char → String  
init, tail :: String → String  
length :: String → Int  
(++) :: String → String → String  
(+) :: Int → Int → Int
```

```
pyramid'  
① = Int  
② = String  
③ = String
```

Type inference

pyramid :: ① → ②

pyramid xs = pyramid' 0 xs

spaces :: Int → String

spaces n = replicate n ' '

pyramid' :: Int → String → String

pyramid' n xs

| length xs < 2 = spaces n ++ xs

| otherwise = spaces n ++ xs ++ "\n" ++
pyramid' (n+1) (init (tail xs))

replicate :: Int → Char → String

init, tail :: String → String

length :: String → Int

(++) :: String → String → String

(+) :: Int → Int → Int

pyramid

① = ?

② = ?

Type inference

```
pyramid :: ❶ → String  
pyramid xs = pyramid' 0 xs
```

```
spaces :: Int → String  
spaces n = replicate n ' '
```

```
pyramid' :: Int → String → String  
pyramid' n xs  
  | length xs < 2 = spaces n ++ xs  
  | otherwise     = spaces n ++ xs ++ "\n" ++  
                    pyramid' (n+1) (init (tail xs))
```

```
replicate :: Int → Char → String  
init, tail :: String → String  
length :: String → Int  
(++) :: String → String → String  
(+) :: Int → Int → Int
```

```
pyramid  
❶ = ?  
❷ = String
```


Type inference

```
pyramid :: String → String
pyramid xs = pyramid' 0 xs
```

```
spaces :: Int → String
spaces n = replicate n ' '
```

```
pyramid' :: Int → String → String
pyramid' n xs
  | length xs < 2 = spaces n ++ xs
  | otherwise     = spaces n ++ xs ++ "\n" ++
                    pyramid' (n+1) (init (tail xs))
```

```
replicate :: Int → Char → String
init, tail :: String → String
length :: String → Int
(++ ) :: String → String → String
(+) :: Int → Int → Int
```

```
pyramid
❶ = String
❷ = String
```

Polymorphic type inference

`dupl :: ...`

`dupl x = (x,x)`

Polymorphic type inference

`dupl` :: ① \rightarrow ②

`dupl` `x` = (`x`, `x`)

① = ?

② = ?

Polymorphic type inference

`dupl` :: ① \rightarrow (③, ④)

`dupl` x = (x, x)

① = ?
② = (③, ④)
③ = ?
④ = ?

Polymorphic type inference

`dup1` :: ① \rightarrow (①, ④)

`dup1` x = (x, x)

① = ?
② = (①, ④)
③ = ①
④ = ?

Polymorphic type inference

`dupl` :: ① \rightarrow (①, ①)

`dupl` x = (x, x)

- No further restrictions: ① remains to be 'unknown'
 - `dupl` has a polymorphic type
- Replace unknowns with readable names

`dupl` :: a \rightarrow (a, a)

`dupl` x = (x, x)

① = ?
② = (①, ①)
③ = ①
④ = ①

Polymorphic type inference

`fst` :: ...

`fst` (x,y) = x

`swap` :: ...

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

Polymorphic type inference: `fst`

`fst` :: ① \rightarrow ②

`fst` (x,y) = x

`swap` :: ...

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

`fst`

① = ?

② = ?

Polymorphic type inference: `fst`

`fst` :: (`3`, `4`) → `2`

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

`swap` :: ...

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

`fst`

`1` = (`3`, `4`)

`2` = ?

`3` = ?

`4` = ?

Polymorphic type inference: `fst`

`fst` :: (`3`, `4`) → `3`

`fst` (x,y) = x

`swap` :: ...

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

`fst`

`1` = (`3`, `4`)

`2` = `3`

`3` = ?

`4` = ?

Polymorphic type inference: swap

`fst` :: $(a, b) \rightarrow a$

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

`swap` :: $\textcircled{1} \rightarrow \textcircled{2}$

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

`swap`

$\textcircled{1} = ?$

$\textcircled{2} = ?$

Polymorphic type inference: swap

`fst` :: $(a, b) \rightarrow a$

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

`swap` :: $(\textcircled{3}, \textcircled{4}) \rightarrow \textcircled{2}$

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

`swap`

$\textcircled{1} = (\textcircled{3}, \textcircled{4})$

$\textcircled{2} = ?$

$\textcircled{3} = ?$

$\textcircled{4} = ?$

Polymorphic type inference: swap

`fst` :: $(a, b) \rightarrow a$

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

`swap` :: $(\textcircled{3}, \textcircled{4}) \rightarrow (\textcircled{5}, \textcircled{6})$

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

`swap`

$\textcircled{1} = (\textcircled{3}, \textcircled{4})$

$\textcircled{2} = (\textcircled{5}, \textcircled{6})$

$\textcircled{3} = ?$

$\textcircled{4} = ?$

$\textcircled{5} = ?$

$\textcircled{6} = ?$

Polymorphic type inference: swap

`fst` :: $(a, b) \rightarrow a$

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

`swap` :: $(\textcircled{3}, \textcircled{4}) \rightarrow (\textcircled{4}, \textcircled{6})$

`swap` $(x, y) = (\boxed{y}, x)$

swap

$\textcircled{1} = (\textcircled{3}, \textcircled{4})$

$\textcircled{2} = (\textcircled{4}, \textcircled{6})$

$\textcircled{3} = ?$

$\textcircled{4} = ?$

$\textcircled{5} = \textcircled{4}$

$\textcircled{6} = ?$

Polymorphic type inference: swap

`fst` :: $(a, b) \rightarrow a$

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

`swap` :: $(\textcircled{3}, \textcircled{4}) \rightarrow (\textcircled{4}, \textcircled{3})$

`swap` $(x, y) = (y, \boxed{x})$

`swap`

$\textcircled{1} = (\textcircled{3}, \textcircled{4})$

$\textcircled{2} = (\textcircled{4}, \textcircled{3})$

$\textcircled{3} = ?$

$\textcircled{4} = ?$

$\textcircled{5} = \textcircled{4}$

$\textcircled{6} = \textcircled{3}$

Polymorphic type inference: swap

`fst` :: (a,b) → a

`fst` (x,y) = x

`swap` :: (a,b) → (b,a)

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

Type-driven program development

- types are a vital part of any program
 - types are not an afterthought
- first specify the type of a function
- its definition is then driven by the type

$f :: T \rightarrow U$

- f consumes a T value: suggests case analysis
- f produces a U value: suggests use of constructors

