Functional Programming

2022-2023

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 het left-hand side of equation)
  - expression style (emphasis on het 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 …`**

case-expressions

- cases can also be analysed using a *case-expression*
- eg

```
day :: Integer ⟶ String
day d = case d of
          1 ⟶ "Saturday"
          2 ⟶ "Sunday"
          _ ⟶ "Weekday"

null :: [a]⟶Bool
null xs = case xs of
            []    ⟶ True
            (_:_) ⟶ False
```

- declaration style: equations using patterns; expression style: case-expression using patterns

Lambda expressions

- notation for anonymous functions
- e.g. `\x ⟶ x * x` as another way of writing `square`
- `x` is the formal parameter; `x * x` is the body of the function
- symbol \ 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 (at least) two arguments can be converted into an infix operator, and vise 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 ⊛ is an operator, then expressions of the form `(a ⊛)`, and `(⊛ b)` for arguments `a` and `b` are called *sections.*
- more formally using lambda expressions:

```
(⊛)   = \x -> (\y -> x ⊛ y)
(x ⊛) = \y -> x ⊛ y
(⊛ y) = \x -> x ⊛ y
```

```
map :: (a⟶b)⟶[a]⟶[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 /= 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 :: Bool ⟶ S`
- step 1: solve the problem for `False`

```
f False = …
```

- step 2: solve the problem for `True`

```
f False = …
f True  = …
```

- (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. `Char` ⟶ `Int`
- `X` ⟶ `Y` ⟶ `Z` is shorthand for `X` ⟶ (`Y` ⟶ `Z`)
- values in a similar syntax: `\c ⟶ ord c - ord '0'`
  - recall: `c` is the formal parameter; `ord c - ord '0'` is the body of the function
- `\x y ⟶ e` is shorthand for `\x ⟶ \y ⟶ e = \x ⟶ (\y ⟶ 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
  - `Int ⟶ Int`
  - `a ⟶ a`
  - `(Int,Int) ⟶ Int`
  - `(a,a) ⟶ a`
  - `(a,b) ⟶ a`
  - `Int ⟶ (Int ⟶ Int)`
  - `(Int ⟶ Int) ⟶ Int`
  - `a ⟶ (a ⟶ a)`
  - `(a ⟶ a) ⟶ 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?

  `(+) :: Integer ⟶ Integer ⟶ Integer`

  `(+) :: Float ⟶ Float ⟶ Float`

- cannot have `(+) :: a ⟶ a ⟶ 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 `(+) :: (Num a)⇒(a ⟶ a ⟶ 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
Eq	==, /=	Integer, Int, Double, String etc
Ord	< etc, min etc	Integer, Int, Double, String etc
Enum	succ etc	Integer, Int, Char, Double etc
Bounded	minBound, maxBound	Int, Char, Bool etc
Show	show	Integer, Int, Char, Bool, [a] etc
Read	read	Integer, Int, Char, Bool, [a] etc
Num	+, * etc	Integer, Int, Double, Float
Integral	div etc	Integer, Int
Fractional	/ etc	Double, Float
Floating	exp, log, sin etc	Double, Float

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

```
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
```

```
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

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

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

Beware: these are not the real types of these functions

```
pyramid xs = pyramid' 0 xs

spaces :: ❶ ⟶ ❷
spaces n = replicate n ' '

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

```
spaces
❶ = ?
❷ = ?
```

Type inference

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

```
pyramid xs = pyramid' 0 xs

spaces :: Int ⟶ ❷
spaces n = replicate n ' '

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

```
spaces
❶ = Int
❷ = ?
```

Type inference

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

```
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))
```

```
spaces
❶ = Int
❷ = String
```

Type inference

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

```
pyramid xs = pyramid' 0 xs

spaces :: Int ⟶ String
spaces n = replicate n ' '

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

```
pyramid'
❶ = ?
❷ = ?
❸ = ?
```

Type inference

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

```
pyramid xs = pyramid' 0 xs

spaces :: Int ⟶ String
spaces n = replicate n ' '

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

```
pyramid'
❶ = ?
❷ = String
❸ = ?
```

Type inference

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

```
pyramid xs = pyramid' 0 xs

spaces :: Int ⟶ String
spaces n = replicate n ' '

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

```
pyramid'
❶ = Int
❷ = String
❸ = ?
```

Type inference

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

```
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))
```

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

Type inference

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

```
pyramid :: ❶ ⟶ ❷
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))
```

pyramid
❶ = ?
❷ = ?

Type inference

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

```
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))
```

```
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
1 = String
2 = String

Polymorphic type inference

```
dupl :: …
dupl x = (x,x)
```

Polymorphic type inference

```
dupl :: ❶ → ❷
dupl x = (x,x)
```

❶ = ?
❷ = ?
x :: ❶

Polymorphic type inference

```
dupl :: ❶ → (❸, ❹)
dupl x = (x,x)
```

```
❶ = ?
❷ = (❸, ❹)
x :: ❶
❸ = ?
❹ = ?
```

Polymorphic type inference

```
dupl :: ❶ ⟶ (❶, ❹)
dupl x = (x,x)
```

❶ = ?
❷ = (❶, ❹)
x :: ❶
❸ = ❶
❹ = ?

Polymorphic type inference

```
dupl :: ❶ ⟶ (❶, ❶)
dupl x = (x,x)
```

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

```
dupl :: a ⟶(a,a)
dupl x = (x,x)
```

```
❶ = ?
❷ = (❶, ❶)
x :: ❶
❸ = ❶
❹ = ❶
```

Polymorphic type inference

```
fst :: …
fst (x,y) = x

swap :: …
swap (x,y) = (y,x)
```

Polymorphic type inference: `fst`

```
fst :: ❶ ⟶ ❷
fst (x,y) = x

swap :: …
swap (x,y) = (y,x)
```

```
fst
❶ = ?
❷ = ?
```

Polymorphic type inference: `fst`

```
fst :: (❸,❹) ⟶ ❷
fst (x,y) = x

swap :: …
swap (x,y) = (y,x)
```

```
fst
❶ = (❸,❹)
❷ = ?
❸ = ?
❹ = ?
x :: ❸
y :: ❹
```

Polymorphic type inference: fst

```
fst :: (❸,❹) ⟶ ❸
fst (x,y) = x

swap :: …
swap (x,y) = (y,x)
```

```
fst
❶ = (❸,❹)
❷ = ❸
❸ = ?
❹ = ?
x :: ❸
y :: ❹
```

Polymorphic type inference: `swap`

```
fst :: (a,b) ⟶ a
fst (x,y) = x
```

```
swap :: ❶ ⟶ ❷
swap (x,y) = (y,x)
```

```
swap
❶ = ?
❷ = ?
```

Polymorphic type inference: swap

```
fst :: (a,b) ⟶ a
fst (x,y) = x

swap :: (❸,❹) ⟶ ❷
swap (x,y) = (y,x)
```

```
swap
❶ = (❸,❹)
❷ = ?
❸ = ?
❹ = ?
x :: ❸
y :: ❹
```

Polymorphic type inference: swap

```
fst :: (a,b) ⟶ a
fst (x,y) = x

swap :: (❸,❹) ⟶ (❺,❻)
swap (x,y) = (y,x)
```

```
swap
❶ = (❸,❹)
❷ = (❺,❻)
❸ = ?
❹ = ?
x :: ❸
y :: ❹
❺ = ?
❻ = ?
```

Polymorphic type inference: `swap`

```
fst :: (a,b) ⟶ a
fst (x,y) = x

swap :: (❸,❹) ⟶ (❹,❻)
swap (x,y) = (y,x)
```

```
swap
❶ = (❸,❹)
❷ = (❹,❻)
❸ = ?
❹ = ?
x :: ❸
y :: ❹
❺ = ❹
❻ = ?
```

Polymorphic type inference: swap

```
fst :: (a,b) ⟶ a
fst (x,y) = x

swap :: (❸,❹) ⟶ (❹,❸)
swap (x,y) = (y,x)
```

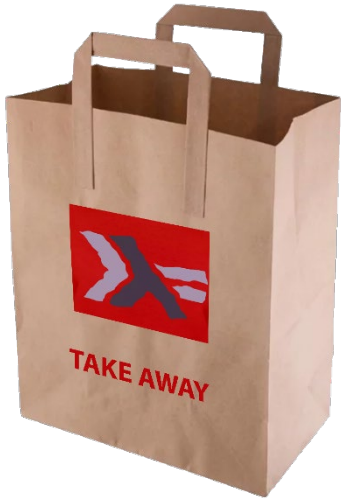
```
swap
❶ = (❸,❹)
❷ = (❹,❸)
❸ = ?
❹ = ?
x :: ❸
y :: ❹
❺ = ❹
❻ = ❸
```

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 ⟶ U
```

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