master course

Functional Languages

in english (also the exams)
6 credits

what will we learn

- we explore the «joys» of the functional approach using Haskell as vehicle
- learn about pure, lazy languages (but able to deal with some impure things)
- learn that types are useful for many things:
  debugging and polymorphism

- that types are inferred automatically (and we see how)
- we will also see how Haskell is compiled (project)
- learn something about run-time management

Material

- Programming in Haskell
  by Graham Hutton, Cambridge Univ. Press, 2ns edition

- Learn you a Haskell for great good
  by Miran Lipovaca

http://learnyouahaskell.com/chapters

for the project:

Implementing Functional languages: a tutorial
by Simon Payton Johns & David Lester, 2000

more material:

- Real World Haskell

by Bryan O'Sullivan, Don Stewart, and John Goerzen

http://book.realworldhaskell.org/read/

- the moodle :

elearning.studenti.math.unipd.it/labs

- for the GHC (Glasgow Haskell Compiler)

https://www.haskell.org/ghc/

- my mail: gilberto@math.unipd.it

- exam

written exam + project + homeworks

an historical perspective

in the 1930's the big problem was to define computable functions

Turing

Church,
Curry
lambda-calculus

computer model
with mutable memory

rules to define
and compose functions
all is immutable

they define the same set of functions that are therefore, the computable functions from computer models to real computers and imperative Languages

from lambda-calculus

FORTRAN
Pascal
C
Modula
C++
Java

LISP
ISWIM
FP
ML
Miranda
Haskell 87
Haskell report 2003 and 2010

SCALA, RUBY, Occaml

A function is a mapping that takes arguments and produces one result

double x = x + x

double 3

= {applying def of double}

3 + 3

= {applying +}

6

double (double 3)
= {applying <u>inner</u> double}
double (3 + 3)
= {applying +}
double 6
=
....
=
12

= {applying <u>outer</u> double}
(double 3) + (double 3)
={applying first double}
6 + (double 3)
=
6 + 6
=12

functional languages are **pure**

--don't have storable and mutable variables, they only have names that hold a constant value, instead of changing the values of variables, new values are computed

--functions don't produce side-effects

advantages: more abstraction, simplicity, correctness, enhance parallel implementation

disadvantages: difficult to model state that changes (i/o, exceptions,....)

A taste of Haskell

1. Summing lists of numbers

```
sum [] = 0
sum (n:ns)= n + sum ns

sum [1,2,3]
= {applying sum}
  1 + sum [2,3]
= {applying sum}
 1 + (2 + sum [3])
= 1 + (2 + (3 + sum [])) = 1 + (2 + (3+0))
= {applying +}
  6
```

:t sum
Num a => [a] -> a

for any type a of numbers

Haskell supports many types of numbers
integers and floating point

type is inferred automatically from 0 and +

types are useful to find errors:
sum [‘a‘, ‘b’] ➔ error

2. Sorting values

```
qsort [] = []
qsort (x:xs) = qsort smaller ++ [x] ++qsort larger
                   where
                        smaller = [a | a <-xs, a<=x]
                        larger   = [b | b <- xs, b>x]
```

++ = list concatenation

where is a keyword that introduces local definitions

[a | a <-xs, a<=x] is list comprehension qsort implements the quick-sort algorithm

----qsort [x] = [x]

qsort [x]
= {applying qsort}
qsort [] ++ [x] ++ qsort []
= {applying qsort}
[] ++ [x] ++ []
= {applying ++}
[x]

```
qsort [3,5,1,4,2]
={applying qsort}
qsort [1,2] ++ [3] ++ [5,4]
={applying qsort}
(qsort [] ++ [1] ++ qsort [2]) ++ [3]
  ++ (qsort [4] ++ [5] ++ qsort [])
={applying qsort and above property}
([] ++ [1] ++ [2]) ++ [3] ++ ([4] ++ [5] ++ [])
= {applying ++}
[1,2] ++ [3] ++ [4,5]
= {applying ++}
[1,2,3,4,5]

:t qsort
qsort :: Ord a => [a] ->[a]
```

Ord is the class of all types whose values have a total order

numbers, characters, strings are ordered types

3. Sequencing actions

```
seqn [] = return []
seqn (act: acts) = do  x <- act
                       xs <- seqn acts
                       return (x:xs)
```

```
seqn [getChar, getChar, getChar]
```

reads the next 3 char from standard input and returns their list

```
seqn :: [IO a] -> IO [a]
```

IO type is for IO operations
that have side effects

seqn :: [IO a] -> IO [a]
 shows that seqn is doing IO, i.e. operations with side effects

but seqn is more general : it can sequence
-actions that change stored value
-actions that may fail
-actions that write to a log file
-ecc

the flexibility is captured by the type class Monad

```
: t seqn
seqn :: Monad m => [m a] -> m [a]
```

Monad is a particular type class,

IO is a particular Monad

seqn is a generic function

features of Haskell

- concise programs
- powerful type system -> inference and type classes
- list comprehension ->math notation
- recursive functions
- higher order functions ->functions as 1° class objects
- effectful functions -> type classes that model effects
- generic functions -> parametric polymorphism
- lazy evaluation -> exps evaluated only when needed
- equational reasoning->functions are equations

Glasgow Haskell Compiler

www.haskell.org

GHC = compiler
GHCi = interpreter

once installed, the command «ghci» runs the interpreter

Prelude> here we can type expressions that are immediately executed
>2+4
6
Prelude is a library

Prelude contains many built-in functions

```
>head  [1,2,3]
1
>tail [1,2,3]
[2,3]
>[1,2,3] !! 1
2
>take 2 [1,2,3]
[1,2]
>drop 2 [1,2,3]
[3]
>length [1,2,3]
3
>sum [1,2,3]
6
```

>[1,2] ++ [3,4]
[1,2,3,4]
>reverse [1,2,3,4]
[4,3,2,1]

Notation

in mathematics f(a,b) +cd

in Haskell f a b + c*d

function application has higher priority than all other operations
f a + b ??

f a g x ??   f a (g x)

Haskell scripts .hs

we write in file test.hs

```
double x = x + x
quadruple x = double (double x)
```

ghci test.hs

```
>quadruple 10
40
>take (double 2) [1,2,3,4]
[1,2,3,4]
```

variables and functions start with lower-case letters, after there may be letters, digits, _, ‘

layout is very important

```
a = b + c
   where
    b=1
    c=2
d=a * 2
```

also

```
a= b + c
   where
     {b=1;
      c=2};
d=a * 2
```

comments : -- or {-......-}

Exercise 4

define a function last that, given a non-empty list, returns the last element of the list.

[1,2,3] -> 3

Exercise 5

define a function init that, given a non-empty list, removes its last element.

[1,2,3] -> [1,2]