Functional Programming

Fall 2017

Funkcinis Programavimas

2017 Ruduo

Functional Programming: Course introduction

Advanced level course for students in Computer Science

- Course web page and lecture slides will be available in Moodle
- Lecturer: Linas Laibinis, e-mail: linas.laibinis@mif.vu.lt
- Office: MIF Building, Didlaukio 47, room 504

Functional Programming: Course introduction

- Lectures: Wednesdays 14–16, 101 MIF-Didlaukio
- Exercise sessions: Mondays 12–14, 14–16, Wednesdays 16–18, 18–20, every second (odd) week, 321 MIF-Didlaukio
- Exercises are given in advance and should be submitted by e-mail or in Moodle before the given deadline

Functional Programming: Course goals

- Learning the key concepts and principles of the functional programming (e.g., functional composition, recursion and induction, higher-order functions, pattern matching, polymorphism, etc.) using the Haskell language
- Solving problems and write programs in a functional style
- Building inductive user-defined data types and write efficient functional programs for them
- Overviewing practical applications of functional programming

Functional Programming: How to pass the course

- **Final exam** at the end of the course
- Exam: to solve a number of tasks of various difficulty in Haskell
- **Exercise sessions** are used for consulting and presenting your solutions
- 5-6 different exercise sets will be given; 2-3 personal meetings explaining and “defending” your solutions are expected
- Exercise marks will give you 40% of exam points

Functional Programming: Literature

- S. Thompson. Haskell: The Craft of Functional Programming. Addison Wesley, 2011
- M. Lipovača. Learn You a Haskell for Greater Good! (Available online). No Starch Press, 2010
- B. Sullivan et al. Real World Haskell (Available online). O'Reilly, 2008

Functional Programming: What is it?

- Based on application and evaluation of (mathematical) functions, expressing the functional relationships between the data
- In short, in functional programming (FP), we compute by evaluating expressions which use functions from our area of interest
- The task of programmer is to write the functions modelling the problem area
- A functional program = a collection of definitions of functions and other values

Functional Programming: What is it? (Once more)

- A computer programming paradigm (or style) that relies on the notion of functions modelled on mathematical functions
- Programs are combinations of expressions, which can be concrete values, variables, and functions
- Functions are expressions that can be applied to an argument or input, and once applied, can be *reduced* or *evaluated*
- Functions are *first-class citizens*: they can be used as values or passed as arguments to other functions
- Essential property – *referential transparency*: the same function, given the same values to evaluate, always will return the same result

Functional Programming: What is it?

- Theoretical basis: lambda calculus
- Different programming paradigm (comparing to imperative and object-oriented), different way to express and solve problems
- Focuses primarily on values, their relationships and transformations
- Example of declarative programming (functions are also data, although more high level)
- Many programming languages: Lisp, ML family (Standard ML, Moscow ML, PolyML, Ocaml), Scheme, Erlang, F#

Functional Programming: What are the advantages?

- Different approach to solving problems (the involved concepts, relationships, ...)
- Side-effect free (functions always return the same results for the same inputs, no internal state involved); Immutable data structures
- Precise and concise description of iterative and recursive calculations; For-loop disappears and is replaced by more flexible and powerful ways to perform iterative tasks
- Functional composition and higher-order functions; Functions can be passed as parameters or created as results

Functional Programming: What are the advantages?

- Pattern matching by using data constructors
- Generics (polymorphic types, type classes)
- Lazy evaluation – computation is deferred until it is actually needed; Working with infinite data structures
- Easy and effective parallelism (mostly because of side-effect freeness)
- Close to mathematical definition; Easy to check/verify correctness;
- Based on strong formalism. The proof is the code. There is a saying "If your code compiles, you're 99% done"

Why learn functional programming?

- Important to learn many languages over your career; Different perspective, different way to approach the problem
- Functional languages/ techniques become increasingly important in industry
- Operate on data structure as *a whole* rather than *piecemeal*
- Good for concurrency, which is very important nowadays

Functional programming – history

- 1950s – the invention of *lambda calculus* as a tool for investigating the foundations of mathematics (Alonzo Church, Haskell Curry)
- 1958 – the first functional language (Lisp)
- 1970s – Robin Milner creates a more rigorous FP language Standard ML to help with automated proofs of mathematical theorems, however, it starts to be used for more general computing and data manipulation tasks
- 1970-80s – many more FP languages appear (Scheme, Miranda, ...)
- 1990 – introduction of the Haskell language

Functional programming – history

- 1990-2000s – influences of FP on development of Python, Perl, Ruby
- 2000s – many new FP languages (Closure, Mathematica, Erlang, Ocaml, F# ...)
- 1990s-2000s – started to be actively used in statistics, data analytics, business mathematics
- 2010s – symbiosis of programming styles: the mainstream languages (Java, C#) started to add features from FP; Emergence of Scala = improved Java + functional programming
- 2010s – FP techniques are used for data analysis and transformations in the cloud (MapReduce)

Java vs Scala

Finding out whether a given string contains an upper case character.

In Java:

```
boolean nameUpperCase = false;
for (int i = 0; i < name.length(); ++i) {
  if (Character.isUpperCase(name.charAt(i))) {
    nameUpperCase = true;
    break;
  }
}
```

In Scala:

```
val nameHasUpperCase = name.exists(_.isUpper)
```

Here `_.isUpper` is a function that takes a character argument (represented by the underscore character), and tests whether it is an upper case letter.

Functional programming is the new new thing

Erlang, F#, Scala attracting a lot of interest from developers

Features from functional languages are appearing in other languages:

- Garbage collection Java, C#, Python, Perl, Ruby, Javascript
- Higher-order functions Java, C#, Python, Perl, Ruby, Javascript
- Generics Java, C#
- List comprehensions C#, Python, Perl 6, Javascript
- Type classes C++ "concepts"

Why Haskell?

- One of the most popular FP languages; Stood test of time
- General-purpose programming language
- Concise, precise and yet (comparatively) easy to understand and learn; Many syntactic definitions have migrated to other languages
- Good documentation, literature, and support community (via www.haskell.org)

Why Haskell?

- Very efficient implementation of FP (especially lazy evaluation, parallel computing)
- Statically typed, yet very flexible via generics and type classes
- The most popular free compiler – ghc (Glasgow Haskell Compiler), and its interactive intepreter – ghci
- Recommended setting – the Haskell Platform (ghc, ghci – interpreter, main standard packages/libraries)
- Can be freely downloaded and installed (in Linux, Windows, and MacOS) from www.haskell.org

Who uses Haskell?

- AT&T – automate form processing
- Bank of America Merril Lynch – data transformation and loading
- Facebook – manipulating PHP code base
- Google – internal IT infrastructure
- MITRE – cryptographic protocol analysis
- NVIDIA – in-house tools
- ...

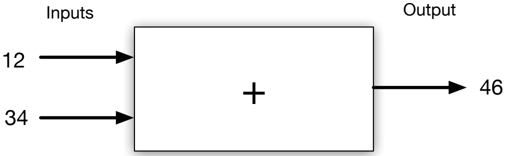
What is a function?

- a recipe for generating an output from inputs, e.g., "Multiply a number by itself":
- an equation, e.g., $f\ x = x^2$
- (for numbers), a graph relating inputs to some output
- In general, a kind of relationship between function input and output data, satisfying the functionality constraint – no more than one output for the same inputs.

What is a function?

A function can be seen as a box that for some given inputs (parameters, arguments) produces the output (result)

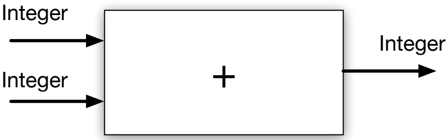
An example: addition



Operators (like +) are all treated as ordinary functions (only infix by default)

Types

A type is collection of values. Each function is defined by giving the intended types of its inputs and outputs:



Haskell is statically typed (never confusion about types)

Function types (*type signature*) in Haskell are given using the following syntax, for instance,

```
square ::   Integer -> Integer
(+) ::   Integer -> Integer -> Integer
```

Types (continued)

In general, using Haskell syntax, function types (type signature) is given as

`name ::` T_1 −> T_2 −> ... −> T_n −> T_o

where `name` is the function name being defined, $T_1, ..., T_n$ are the types of function parameters, and T_o is the type of the function result

If `name` is an operator, it is surrounded by parentheses, e.g., `(+)`

Haskell definitions

- A Haskell definition = Type signature + Data or function declaration
- A Haskell module (file) typically consists of a collection of such definitions
- If a type signature is omitted, Haskell tries to infer the (most general) type itself
- In general, a type signature is recommended to ensure strict intended typing

Haskell definitions

- Simple value declaration:

  name :: Type
  name = expression

  size :: Integer
  size = 3 ˆ12

- Functional declaration:

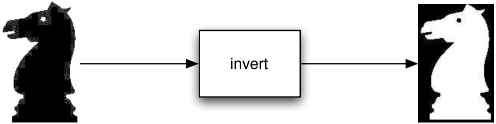
  fname :: T_1 –> T_2 –> ... –> T_n –> T_o
  fname p1 p2 ... pn = expression

Examples of Haskell types

- Integers: 42, -69
- Floats: 3.14
- Characters : 'h'
- Strings (lists of characters): "hello"
- Booleans: True, False
- User defined/implemented types, e.g., Pictures: 

Applying a function

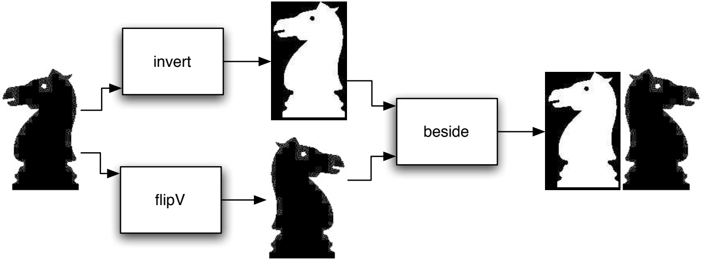
```
invert :: Picture -> Picture
invert p = ...

knight :: Picture
knight = ...

invert knight
```



Composing functions

```
beside :: Picture -> Picture -> Picture
flipV :: Picture -> Picture
invert :: Picture -> Picture
knight :: Picture
```

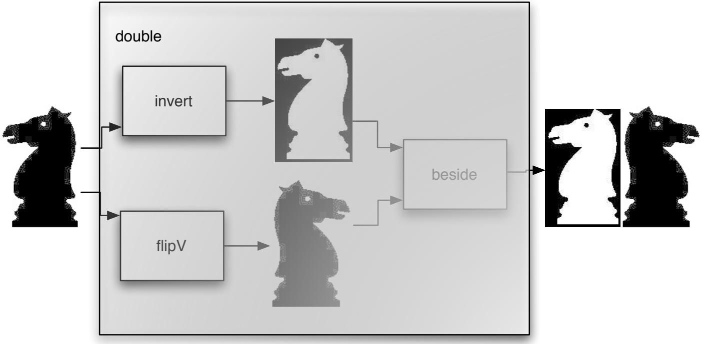
```
beside (invert knight) (flipV knight)
```



Defining a new function

double :: Picture −> Picture
double p = beside (invert p) (flipV p)

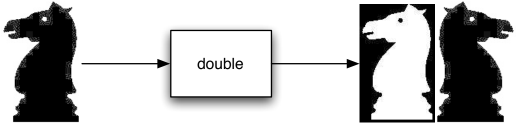
double knight



Defining a new function

```
double :: Picture -> Picture
double p = beside (invert p) (flipV p)
```

double knight



Terminology

Type signature

```
double :: Picture -> Picture
```

Function declaration

$$\underbrace{\texttt{double}}_{\text{function name}}\ \texttt{p}\ =\ \underbrace{\texttt{beside (invert p) (flipV p)}}_{\text{function body}}$$

Polymorphic Types and Type Classes

Sometimes function can be applied to parameters of any type. To express that, Haskell introduces polymorphic types, for example

```
id :: t -> t
id x = x
```

where t is a type variable, standing for arbitrary type

If we want to constrain a collection of types, the notion of type classes is used. Then the function type declaration becomes

```
name :: Class t => t -> t
```

where *Class* denotes a specific type class t belongs to

A simple module in Haskell

```haskell
{-
FirstScript.hs
-}

module FirstScript where

-- The value size is an integer
size :: Integer
size = 25

-- The function to square an integer
square :: Integer -> Integer
square n = n*n
```

A simple module in Haskell (continued)

```
-- The function to double an integer
double :: Integer -> Integer
double n = 2*n

-- An example using double, square, and size
example :: Integer
example = double (size - square (2+2))
```

Exercise sessions the next week (11.09 and 13.09)

- No exercises to solve this time
- The first set of exercises will be given the next week (with the deadline in three weeks)
- The goal of the first exercise session is to get accustomed with the ghci interpreter of Haskell
- Small pieces of code to try and experiment on will be given during the session

(Some) ghci interpreter commands

:type ⟨*expr*⟩ (or :t ⟨*expr*⟩)	– the type of data expression or function
:load ⟨*file*⟩ (or :l ⟨*file*⟩)	– load a Haskell module from *file*
:reload (or :r)	– repeat the last load command
:info ⟨*name*⟩ (or :i ⟨*name*⟩)	– information about the identifier *name*
:browse ⟨*name*⟩	– all definitions from the module *name*
:help (or :h)	– the list of all interpreter commands
:!⟨*shell_command*⟩	– run a shell command
:quit (or :q)	– quit the system

Typical ghci error messages

- Parsing/syntax errors "Parse error – possibly incorrect ..."
- Wrong or undefined name "Variable ... not in scope"
- Typing errors "No instance for (Type1, Type2) arising at ..."
- Typing errors "Could not match expected type Type1 against inferred type Type2"