WS 2020/2021

Functional Programming

Part 2 – Introduction

René Thiemann Benedikt Dornauer Maximilian W. Haslbeck
Bart Keulen Fabian Mitterwallner Christian Sternagel
Vincent van Oostrom Xiang Zhang Philipp Zech

Department of Computer Science

(Functional) Programming

- task: solve problems
  - sort a list
  - generate a website
  - navigate from Innsbruck to Cologne
- distinguish between data ...
  - input `[1,5,2]` and output `[1,2,5]`
  - query "search for 'functional programming'" and resulting website
  - map of Europe, two locations and route
- ... and programs
  - control over how data should be processed
  - mostly written by humans
- usually computers are used for executing a program on some input, but computation can also be done by hand or in mind

Learning Programming

- \+ read, study and write programs (many)
- \+ actively attend lecture and proseminar
- \+ try to solve exercises (alone or discuss in small teams)
- − copy solutions from other students or from the internet

Algorithms and Programs

story (language agnostic)

text (language dependent)

- Tom and Paul were struggling until ...
- Thomas und Paul rauften solange bis ...
- 토마스와 파울은 싸우고 있었는데...

algorithm (prog. language agn.)

- task: determine the maximum of m and a list of numbers
- if list is empty, result is m
- otherwise, change m to max. of head of list and m
- continue with tail of list

program (language dependent)

- ```
  maxlist m [] = m
  maxlist m (x : xs) =
    maxlist (max m x) xs
  ```
- ```
  while (list != null) {
    m = max(m, list.num);
    list = list.next; }
  return m;
  ```

programs (in programming languages) have to respect specific syntax

Different Programming Styles

- Imperative Programming (VO Introduction to Programming)
  - state is assignment of variables to data
  - assignments instruct computer to update state
  - example
    - if `x` stores value 7
    - then after assignment `x := x + 5`
    - `x` stores value 12
- Functional Programming (this lecture)
  - define functions (mathematical: same input implies same output), no assignments
  - computer evaluates these functions via equational reduction
  - example
    - define new function `increase5 x = x + 5`
    - ask computer to evaluate expression `increase5 7`
    - evaluation: `increase5 7 = 7 + 5 = 12`
- Logic Programming, Object Oriented Programming, ...

Different Programming Styles

- fact: most programming languages are of equal power
- demand for different styles still reasonable
  - each style has its own distinguishing features and limitations (like in real languages: translate "Ohrwurm" or "Internetbrowser")
  - good programmer should know about alternatives: choose suitable style and language depending on problem and context
- advantages of functional programming
  - intuitive evaluation mechanism
  - suitable for verification
  - expressive language features
  - suitable for parallelization
- disadvantages of functional programming
  - more difficult to model state, side-effects, and I/O
  - not main-stream in industry, but getting more popular

Different Functional Programming Languages

- combinatory logic (Moses Schönfinkel 1924, Haskell Curry 1930): foundation of FP
- λ-calculus (Alonzo Church 1936): foundation of FP
- LISP (John McCarthy, 1958): List Processing
- ML (Robin Milner, 1973): Meta Language, several dialects
- Erlang (Ericsson, 1987): distributed computing
- Haskell (Paul Hudak and Philip Wadler, 1990): language in this course
- F# (Microsoft, 2002) and Scala (Martin Odersky, 2003): combine different programming styles, including FP

Syntax and Semantics

- syntax of a (programming) language defines what are valid sentences (programs)
  - "This is a proper English sentence."
  - "this one not propper"
  - computers refuse programs that contain syntactical errors!
- semantics defines the meaning of valid sentences / programs
  - "Clean your room!" ✔
  - `let xs = 1 : 1 : zipWith (+) xs (tail xs) in take 9 xs` ✘
- we will learn both syntax and semantics of Haskell

Haskell Scripts

```
-- This script is stored in file script.hs
square :: Integer -> Integer
square x = x * x
```

- a Haskell script (= program) has file-ending `.hs`
- a script is a collection of (several) function definitions
- each function definition consists of
  - type declaration: `square :: Integer -> Integer`
    syntax: ⟨*name*⟩ `::` ⟨*type*⟩
  - defining equation: `square x = x * x`
    syntax: ⟨*name*⟩ ⟨*vars*⟩ `=` ⟨*exp*⟩
    evaluation from left to right
  - ⟨*name*⟩s and ⟨*vars*⟩
    - always start with lower-case letters
    - consist of letters, digits and `_`
- comments are just for humans, ignored for evaluation
- single-line and multi-line comments
  - single: `-- everything right of -- is a comment`
  - multi: `{- comments can deactivate ... parts of script -}`

Editing Haskell Scripts

```haskell
-- This script is stored in file script.hs
square :: Integer -> Integer
square x = x * x
```

- coloring
  - above script is printed in color
  - when entering a Haskell script (or other computer programs), one does not add these colors in an text editor
  - editors for computer programs display scripts automatically in colors (syntax highlighting); this simplifies reading programs – identify
    - comments
    - keywords (in particular mistyped keywords)
    - identifier
    - ...
- white-space
  - in Haskell white-space matters
  - for the moment, start every new line without blanks
  - the following script is not accepted

```haskell
square :: Integer -> Integer
 square x = x * x
```

Functional Programming – Sessions

- starting a session is like activating your calculator
- we use ghci, an interpreter for Haskell

```
rene$ ghci          -- start the interpreter
Prelude> 42         -- enter a value
42
Prelude> 5 * (3 + 4)       -- evaluate an expression
35
Prelude> :load script.hs  -- load script from file script.hs
[1 of 1] Compiling Main  ( script.hs, interpreted )
Ok, 1 module loaded.       -- script was accepted
*Main> square (5 + 3)      -- expression including square
64
*Main> :quit
```

Workflow for Functional Programming

- define functions in script
- load script (will compile script or deliver error message)
  - parse error: `5 +` (argument missing)
  - type error: `5 + "five"` (cannot add number and text)
  - error-messages are sometimes cryptic
- enter expression and let it evaluate to normal form
  (read-eval-print loop, REPL)
  - normal form: (canonical representation of some) value which cannot be further simplified, e.g., `42`, `"hello"`, `[7,1,3]`, ...
  - evaluation uses
    - built-in functions (`+`, `*`, `:`, `++`, `head`, `tail`, ...), defined in Prelude
    - user-defined functions (`square`,...) from script-files

Compare FP to Calculator

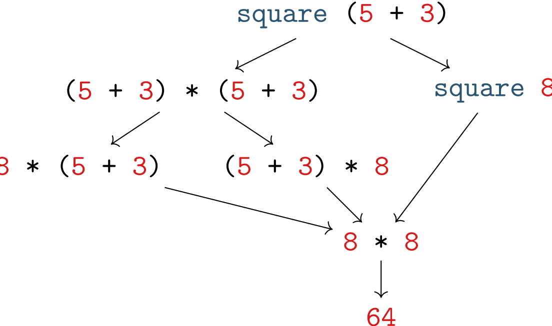
- enter expression and let it evaluate to normal form
- restricted to numbers and built-in functions

Pure Functions

- a function is pure if it always returns same result on same input
- pure functions are similar to mathematical functions
- examples of pure functions
  - addition
  - sort a list
  - ...
- examples of non-pure functions
  - roll a dice
  - current time
  - position of cursor
  - ...
- pure languages permit to define only pure functions
- Haskell is a pure language
  (workaround possible to model non-pure functions)

Evaluation Order

- there are several ways to evaluate expressions



- in pure languages, the evaluation order has no impact on resulting normal form

Theorem

Whenever there are two (different) ways to evaluate a Haskell expression to normal form, then the resulting normal forms are identical.

Not Getting A Result

- not all expressions deliver a value
  - non-terminating ones:
    load script
    ```
    inf :: Integer
    inf = 1 + inf
    ```
    and try to evaluate expression `inf`:

    `inf = 1 + inf = 1 + (1 + inf) = ...`

    computation does not finish (abort with CTRL-C in ghci)
  - underspecified ones: ``1 `div` 0``
    computation aborts with error message

  in both cases the result is $\perp$ (bottom),
  a special value indicating undefinedness
- the ability to write non-terminating programs is desirable
  - ghci should never stop while waiting for new user inputs
  - programs for controlling a vending machine should not stop
- but often non-termination is not desired
  - task such as sorting a list, computing a route, etc. should always halt
- evaluation strategy can have impact on termination behaviour