Functional Programming

2022-2023

Sjaak Smetsers

Twan van Laarhoven

(Marc Schoolderman)

Introduction: the functional way of problem solving

Lecture 1

Outline

- Organization
- Literature
- What’s it all about?

Organisation

- Lecture (Monday – 8:30-10:15, on campus, recorded):
  - introduction and explanation of concepts
  - lecture slides available immediately on Brightspace after lecture
- Assignments:
  - Exercise sets (Brightspace)
  - assignment available immediately after lecture (on Brightspace)
  - deadline Sunday 17:30
- Tutorial (Wednesday – 8:30-10:15, on campus, not recorded):
  - discuss any questions you have about lecture / assignments, making exercises from the assignment
- Lab (Friday – 8:30-12:15):
  - work on assignments, student assistants are available

Organisation

- Preparatory assignments:
  - exercises intended to familiarize yourself with the concepts
  - do them before the tutorial to see how well you understand the new stuff
- Mandatory assignments:
  - upload before deadline: only last upload is stored on Brightspace!
  - exercises intended to apply your skills with new concepts
- Exam Regulation
  - at most 3 assignments can be skipped
    - submit a (simple text) file in which you indicate why you did not make the assignment
  - you cannot retake assignments (not during the course and not at the end)
  - the course is concluded with a written (digital) exam. If you do not meet the above requirement, you may take the written exam, but your final result for the course will be a maximum of 5

Exam

- Exam: closed book, digital with Cirrus (only)

Literature

- Miran Lipovaca, *Learn You a Haskell for Great Good!: A Beginner's Guide*, No Starch Press, 2011.
- Richard Bird, *Thinking Functionally with Haskell*, Cambridge University Press, 2015.
- Will Kurt, *Get Programming with Haskell*, Manning Publications, 2018.
- Graham Hutton, *Programming in Haskell (2nd Edition)*, Cambridge University Press, 2016.
- Bryan O'Sullivan, John Goerzen, Don Stewart, *Real World Haskell*, O'Reilly Media, 2008.
- Simon Thompson, *Haskell: The Craft of Functional Programming (3rd Edition)*, Addison-Wesley Professional, 2011.

```
square x = x * x
```

FP and OOP

Imperative

- assignment
- iteration
- arrays
- eager evaluation
- Von Neumann architecture

Functional

- pure, no side-effects
- recursion
- (infinite) lists
- lazy evaluation
- reduction

Object oriented programming

- classes, interfaces, inheritance
- generic classes

- overloading and type classes
- polymorphism

- higher order functions

Expressions vs statements

- in ordinary programming languages the world is divided into a world of *statements* and a world of *expressions*
- statements:
  - `x := e,  s1 ; s2, while e do s`
  - execution order is important

    `i := i + 1 ; a := a * i` $\neq$ `a := a * i ; i := i + 1`
- expressions:
  - `a + b * c, a and not b`
  - evaluation order is unimportant (*referential transparency*): in

    `(2 * a * y + b) * (2 * a * y + c)`
  - evaluate either parenthesis first (or both simultaneously!)
  - assumes no side-effects: order matters in `++x + x--`

Comparison with ‘ordinary’ programming

- insertion sort
- q
- (binary search trees)

Insertion sort: Modula-2

```
PROCEDURE InsertionSort ( VAR a : ArrayT ) ;
 VAR i , j : CARDINAL ;
     t : ElementT ;
BEGIN
 FOR i := 2 TO Size DO
  (* a[1..i-1] already sorted *)
  t := a [ i ] ;
  j := i ;
  WHILE ( j > 1 ) AND ( a [ j-1 ] > t ) DO
   a [ j ] := a [ j-1 ]; j := j-1
  END ;
  a [ j ] := t
 END
END InsertionSort ;
```

Insertion sort: Haskell

```
insertionSort [ ]      = [ ]
insertionSort (x : xs) = insert x (insertionSort xs)

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

q: C

```
void    q    ( int a [ ] , int l , int r ) {
  if ( r > l ) {
    int i = l ; int j = r ;
    int p = a [ ( l + r ) / 2 ] ;
    for ( ; ; ) {
      while ( a [ i ] < p ) i++;
      while ( a [ j ] > p ) j--;
      if ( i > j ) break ;
      swap(&a[i++] , &a[j--]);
    } ;
        q    ( a , l , j ) ;
        q    ( a , i , r ) ;
  }
}
```

Quicksort: Haskell

```
quickSort [ ] = [ ]
quickSort (x : xs) = quickSort littles ++ [x] ++ quickSort bigs
  where littles = [a | a ← xs, a < x]
        bigs    = [a | a ← xs, x ≤ a]
```

Editors, IDEs, Interpreters, Compilers, Scripts

- *Haskell Platform*
  - Compiler + interpreter, no editor/IDE
  - Available on the PCs in the computer labs (not on Windows)
  - Install on our own PC/laptop
- we will use *GHCi*, an interactive version of the *Glasgow Haskell Compiler*, a popular implementation of *Haskell*

- a program is a collection of *modules*
- a module is a collection of (functions and data) *definitions* (aka a *script*)
- a standalone program includes a ‘main’ expression

My First Haskell Script

- When developing a Haskell script, it is useful to keep two windows open, one running an editor for the script, and the other running GHCi.

- Start an editor, type in the following two function definitions, and save the script as `mfhs.hs`:

```
double x = x + x


quadruple x = double (double x)
```

My Second Haskell Script

- Define a function that adds the numbers 1 to 10
  - You cannot use loops because there are none in Haskell
  - You also cannot use variables

```
sum1to10 = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
```

- And now more general: Define a function that adds the numbers n to m.
- Use recursion (think recursively)
  - Base case (no more numbers left): result is 0
  - Recursive case (still numbers left): result is the current number added to the result of the recursive call for the remaining numbers.

```
sumNtoM n m = if n > m then 0 else n + sumNtoM (n+1) m
```

Evaluation

- interpreter evaluates expression by reducing to simplest possible form
- *reduction* is rewriting using meaning-preserving simplifications: *replacing equals by equals*

```
square x = x * x
```

```
square (3 + 4)
```

⟹ { definition of + }

```
square 7
```

⟹ { definition of `square` }

```
7 * 7
```

⟹ { definition of * }

```
49
```

- expression 49 cannot be reduced any further: *normal form*
- *applicative order* evaluation: reduce arguments before expanding function definition (call by value, eager evaluation)

Alternative evaluation orders

```
square x = x * x
```

- other evaluation orders are possible:

  `square (3 + 4)`

  ⟹ { definition of *square* }

  `(3 + 4) * (3 + 4)`

  ⟹ { definition of + }

  `7 * (3 + 4)`

  ⟹ { definition of + }

  `7 * 7`

  ⟹ { definition of * }

  `49`

- final result is the same: if two evaluation orders terminate, both yield the same result (*confluence*)
- *normal order* evaluation: expand function definition before reducing arguments (call by need, lazy evaluation)

Values

- in FP, as in maths, the sole purpose of an expression is to denote a value
- values may be of various kinds: numbers, truth values, characters, tuples, lists, functions, etc
- important to distinguish *abstract value* (the number 42) from *concrete representation* (the characters '4' and '2', the string "XLII", the bit-sequence 0000000000101010)
- evaluator prints *canonical representation* of a value
- some values have no canonical representation (e.g. functions), some have only infinite ones (e.g. π)

Functions

- naturally, FP is a matter of functions
- script defines *functions* (`square`, `insert`)
- function transforms (one or more) arguments into result
- *deterministic*: same arguments always give same result
- may be *partial*: result may sometimes be undefined
- e.g. cosine, square root; distance between two cities; compiler; text formatter; process controller

Function types

- *type declaration* in a script specifies the type of function
- e.g. `square :: Integer → Integer`
- in general, `f` :: $A \longrightarrow B$ indicates that function `f` takes arguments of type A and returns results of type B
- *apply* function to argument: `f x`
- sometimes parentheses are necessary: *square* (3 + 4) (function application is an operator, binding more tightly than the operator +)

Operators

- functions with alphabetic names are *prefix*: `f 3 4`
- functions with symbolic names are *infix*: `3 + 4`
- make an alphabetic name infix by enclosing in back-quotes: ``17 `mod` 10``
- make symbolic operator prefix by enclosing it in parentheses: `(+) 3 4`

Definitions

- we’ve seen some simple definitions of functions so far
- can also define other kinds of values:

```
name :: String
name = "Sjaak"
```

- all definitions so far have had an identifier (and perhaps formal parameters) on the left, and an expression on the right
- other forms possible: conditional, pattern-matching, and local definitions
- also recursive definitions (later)

Conditional definitions

- definition of *smallest* using a *conditional expression*:

```
smallest :: Integer ⟶ Integer ⟶ Integer
smallest x y = if x ≤ y then x else y
```

- could also use *guarded equations*:

```
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*)
- especially convenient with three or more clauses

Declaration vs expression style

- Haskell supports two different programming styles
- *declaration style*: using (guarded) equations, patterns and expressions

```
smallest :: Integer ⟶ Integer ⟶ Integer
smallest x y
    | x ≤ y     = x
    | otherwise = y
```

- *expression style*: emphasizing the use of expressions

```
smallest :: Integer ⟶ Integer ⟶ Integer
smallest x y = if x ≤ y then x else y
```

- expression style is often more flexible
- experienced programmers use both simultaneously

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* _
- evaluate by reducing argument to normal form, then applying first matching equation
- result is undefined if argument has no normal form, or no equation matches

Local definitions

- repeated sub-expressions 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
```

- in conjunction with guarded equations, the scope of a **where** clause covers all guard clauses

Layout

- structure obtained by layout, not punctuation
  - all definitions in same scope must start in the same column
  - indentation from start of definition implies continuation

```
demo :: Integer ⟶ Integer ⟶ Integer
demo x y = (a + 1) * (b + 2) where
    a = x – y
    b = x + y
```

  - use spaces, not tabs!

`let`-expressions

- **`where`** clause is syntactically attached to an equation
- also: 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 …`**
- **`let`**-expressions are more flexible than **`where`** clauses

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
```

spaces

string

newline

recursion

Pyramid: Java

```
public class Main {

   public static void main(String[] args) {
      System.out.print( pyramid("Functional Programming is fun") );
   }

   private static String pyramid(String str) {
      StringBuilder sb = new StringBuilder();
      for (int n = 0; n < (str.length() + 1) / 2; n++) {
         sb.append(" ".repeat(n));
         sb.append(str.substring(n, str.length() - n));
         sb.append("\n");
      }
      return sb.toString();
   }
}
```

Pyramid: Haskell

```haskell
module Pyramid where

pyramid :: String -> String
pyramid str = pyram 0 str where
   pyram :: Int -> String -> String
   pyram n str
      | length str <= 2 = spaces ++ str ++ newline
      | otherwise       = spaces ++ str ++ newline ++ pyram (n+1) substr

      where spaces  = replicate n ' '
            newline = "\n"
            substr  = tail (init str)


main = putStr (pyramid "Functional programming is fun")
```

The art of functional programming

- a problem is given by an expression
- a solution is a value
- a solution is obtained by evaluating an expression to a value
- a program introduces vocabulary to express problems and specifies rules for evaluating expressions
- the art of functional programming: finding rules
- Haskell has a very simple computational model
- . . . as in primary school: replacing equals by equals
- we can calculate not only with numbers, but also with lists, trees, grammars, pictures, music . . .