How I Teach Functional Programming

Johannes Waldmann, HTWK Leipzig

WFLP Würzburg, 22. 9. 2017

For Whom, and Why

- course *Advanced Programming* (Fortgeschrittene Programmierung)
- mandatory for 4th semester B. Sc. students of CS (Informatik and Medieninformatik)
- main thesis:
  advanced programming *is* (based on concepts from) functional programming
- for example: algebraic data types, static typing, higher order functions, laziness
- this talk: course topics *illustrated by exercises* (selected — find more in the paper)

Course Topics

- first-order data and programs
  - data $=$ terms over signature $=$ algebraic data types
  - programs $=$ term rewriting systems $=$ oriented equations, pattern matching
- higher-order data and programs (λ-calculus)
- patterns for systematic recursion
  - algebra over signature $=$ `fold`
- generic polymorphism, restricted polymorphism
  - type variables, type classes
- evaluation on demand, streams, FRP

First-Order Data

- Exercise: replace `undefined` by an expression such that `test` is `True`

```
import qualified Data.Set as S
-- imported from Prelude:
-- data Bool = False | True
data C = R | G | B          deriving (Eq, Ord,
data T = X C | Y Bool Bool deriving (Eq, Ord,
solution :: S.Set T
solution = S.fromList undefined
test :: Bool
test = S.size solution == 7
```

- automated grading by Leipzig `autotool` software for E-Learning, E-Assessment

First-Order Programs (Model)

- Example Exercise

```
for the system  TRS
   { variables = [ x, y, z]
   , rules = [ f (f (x,y),z) -> f (x,f (y,z))
             , f (x,f (y,z)) -> f (f (x,y),z)
give a sequence of steps
from  f (f (f (a , b ), f (c , d )), e )
to    f (a , f (f (b , c ), f (d , e )))
```

- Example solution (attempt):

```
[ Step { rule_number = 0 , position = [0,1]
   , substitution = listToFM
     [ (x, f(a, b)), (y, f(c, d)), (z, e) ] }
```

FO Programs — Pattern Matching

- ```
  data Bool = False | True
  data T = F T | G T T T | C
  ```
- answer for each of the following expressions:
  - is it statically correct
  - what is its result (its dynamic semantics)
  - is the pattern set complete? disjoint?

```
1. case False of { True -> C }
2. case False of { C -> True }
4. case G (F C) C (F C) of { G x y z -> F z }
5. case F C of { F (F x) -> False }
6. case F C of { F (F x) -> y }
7. case F C of { F x -> False ; True -> False
8. case True of { False -> C ; True -> F C }
```
```

FO Programs — Automated Testing

- ```
  import Prelude hiding (min)

  data N = Z | S N deriving (Show , Eq)

  plus :: N -> N -> N
  plus x y = case x of
     { Z -> y ; S x' -> S (plus x' y) }

  min :: N -> N -> N ; min x y = undefined

  spec1 = \ x y -> min x y == min y x
  spec2 = \ x y -> min (plus x y) x == x
  ```
- property-based testing (`small|lean-check`)
- specification should not give away solution
  ```

Type Inference — Eminently Useful

- types are not just for “slowing down the programmer”, or documenting code
- ```
  check :: Testable a => a -> IO ()
  class Testable a where ...
  instance Testable Bool where ...
  instance (Listable a, Testable b)
    => Testable (a -> b) where ...
  class Listable a where tiers :: [[a]]
  instance Listable Int where ...
  ```
- `check (\(x::Int)(y::Int)->x+y==y+x)`,
  given the above, the compiler statically infers
  `instance Testable (Int->(Int->Bool))`
  and generates *useful code* in each infer. step
  ```

Polymorphic Types — Data

- given

```
data () = ()
data Bool = False | True
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b

data C = R | G | B
data Pair a b = Pair a b
```

name all elememts of type

```
Either (Pair Bool (Maybe ()))
       (Maybe (Maybe C))
```

Polymorphic Types Prevent Cheating

- Exercise:

```
reverse :: List a -> List a
reverse xs = undefined
-- specification
 reverse (Cons True (Cons False Nil))
== Cons False (Cons True Nil)
```

- cheating solution:

```
reverse xs = Cons False (Cons True Ni
```

  is prevented by the type declaration
- the point is: declaring a polymorphic type *enforces* abstraction

Schematic Recursion — Folds

- principle: apply a recursion scheme = replace each constructor (function symbol) with a corresponding function.
- ⇒ each algebraic data type has *exactly one* such schema (fold), its type and implementation can be read off the `data` declaration

```
data List k
      = Nil     | Cons k (List k)
fold ::        r   -> (  k ->  r    -> r)
      -> List k -> r
```

- write down "the fold" for `Bool`, `Maybe`,. . . , look up its type in `https://www.haskell.org/hoogle/`

How To Solve “Write *f* as a Fold”

- Method:
  - draw tree for example input t
  - write $f(s)$ at root of each substree s of t
  - read off test cases for fold’s argument func.s
- Example:

```
f = \xs -> odd (length xs) = fold n c

t     = C 7 (C 4 (C 7 Nil)); f t = True
s     =      C 4 (C 7 Nil) ; f s = False

f t   = c 7 (f s1)  ;  True = c 7 False
```

- *avoid* operational reasoning (“then we go to. . . ”)
- all we need is correctness of the induction step

How To Prove that *f* is Not a Fold

- Method:
  - same as before
  - derive contradiction
- Example: `f = \ xs -> length xs >= 2`

```
f (Cons () (Cons () Nil)) = True
f          (Cons () Nil)) = False
f                   Nil   = False
==> c    () False         = True
 and        c    () False = False
```

Rel. to “standard” (i.e., OO) Topics

- data: immutable objects
  e.g., `git` data model (file system and history)
- trees: *composite* design pattern
- higher order functions: *strategy* design pattern
- recursion pattern (fold): *visitor* design pattern
- lazy stream: *iterator* design pattern
- functional reactive programming:
  (an alternative to) *observer* design pattern

. . . λ calculus is being invented over and over —
who was first?

λ Calculus — Invented in 1892 by . . .

Arthur C. Doyle: Adventure of the Blue Carbuncle

- Hotel Cosmopolitan Jewel Robbery. — John Horner, 26, plumber, was brought up upon the charge of having upon the 22nd inst., *abstracted* from the jewel-case of the Countess of Morcar the valuable gem known as the blue carbuncle.
- Found at the corner of Goodge Street, a goose and a black felt hat. Mr. Henry Baker can have the same by *applying* at 6:30 this evening at 221B, Baker Street.
  (apply = vor(an)stellen, `baker $ holmes`)

Convergence of Language Evolution?

- ```
  $ ghci   # GHCi, version 8.0.2
  Prelude> let d f x = f (f x)
  Prelude> d d d (\x -> x + 1) 0
  16
  ```
- ```
  $ node # v8.5.0, ES6
  > let d = f => x => f (f (x))
  > d(d)(d)(x => x + 1)(0)
  16
  ```
- nice: syntactic differences mostly gone. BUT …
- *We Need Static Typing*!
  Watch out for attempts to undermine, downplay, postpone, ignore it (especially in teaching).
  We teach the right thing, industry will follow — not the other way around.