advanced programming seminar 1

September 14 2018

Pieter Koopman

Radboud University

preliminary planning

date	lecture
9/10/18	recap FP
9/17/18	generic programming cons kind
9/24/18	generic programming Clean style
10/1/18	iTask intro
10/8/18	iTask combinators
10/15/18	iTask advanced
10/22/18	*Break*
10/29/18	*Break*
11/5/18	state in FP
11/12/18	DSL deep embedding
11/19/18	DSL shallow embedding
11/26/18	DSL with GADT
12/3/18	tagless DSL
12/10/18	Model Based Testing
12/17/18	dependent types
12/24/18	*Break*

Planning is subject to change

Teachers:

- Pieter Koopman M1.1.07
- Mart Lubbers
- Rinus Plasmeijer M1.1.06

Talk, mail or visit us in case of problems/questions

{pieter, mart, rinus}@cs.ru.nl

assignments

- the only way to master programming is by doing it
  - reading examples is great
  - but, there is a big difference between understanding a programming technique and being able to apply it
  - to pass the exam you have to be able to apply the techniques introduced
  - **you'd better practice !**
- weekly assignment to master topic of the week
  - make them together with partner: you learn more by discussing, collaboration is less work for you, collaboration is less work for us
- 2 larger assignments to use techniques introduced
  - these can increase your final result for the course if the exam result ≥ 5

seminars

- help you to make the assignment
  - raise questions
  - be sure that you know the assignment
- discuss the previous assignment
  - tell us if you want more or less details
  - tell us if there are things unclear

Programming language used in this course

- Clean
- http://clean.cs.ru.nl/
  - download the version from https://ftp.cs.ru.nl/Clean/builds/
  - there is a new version (almost) every day, the basis should be rock stable
  - later we will need a new version, but that is not 100% stable
- available for Windows, Mac OS X, Linux
  - only the Windows version has a simple IDE, much better than nothing

- Haskell ?
  - Clean and Haskell are similar
  - we will use libraries that are not available in Haskell
  - the digital exam will provide help for Clean, not for Haskell
  - you better should get used to Clean

Clean initial expression

- any Clean program starts evaluating `Start`
- the result is printed on the console
  - unless you specify something else
- e.g. file prog.icl

```
module prog

import StdEnv

fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)

Start :: Int
Start = fac 7
```

module name must match file name

standard library

it is encouraged to specify types

your definitions

program evaluates and prints this

modules

```
implementation module Bin
import StdEnv

ins :: a (Bin a) -> Bin a | < a
ins a Leaf = Bin Leaf a Leaf
ins a (Bin l b r) | a < b
    = Bin (ins a l) b r
    = Bin l b (ins a r)

inorder :: (Bin a) -> [a]
inorder Leaf = []
inorder (Bin l a r) = inorder l ++ [a:
inorder r]
```

```
definition module Bin
import StdEnv
:: Bin a = Leaf | Bin (Bin a) a (Bin a)
ins :: a (Bin a) -> Bin a | < a
inorder :: (Bin a) -> [a]
```

defined in StdEnv

definition of Bin is not repeated here

```
module BinDemo
import Bin

mysort :: [a] -> [a] | < a
mysort l = inorder (foldr ins Leaf l)

Start = mysort ['u','o'..'a']
```

StdEnv imported by Bin

clean files

- prog.icl implementation module, the actual function definitions
- prog.dcl definition module, the exported definitions (data types + functions)

  list only the type of functions
- prog.prj project file of main module, project settings, paths, …
- prog.abc generated abstract machine code
- prog.o object code, generated machine code
- prog.exe windows executable

- the main module does not need a .dcl file
  - the first line is: `module` *`filename`*
  - not `implementation module` *`filename`*

infix operators

- infix operators are just binary functions

```
(o) infixr 9 ::  (a -> b)  (c -> a) -> c -> b
(o) f g = \ x . f (g x)

twice f = f o f
```

argument count is reflected in the type
Haskell: (a -> b) -> (c -> a) -> (c -> b)

- you can add your own operators
  - specify binding direction and priority

Currying is fine

- many infix operators are classes
  - e.g. ==, +, -, *, ..
  - you can define your own instances

```
class (+) infixl 6 a :: !a !a -> a
class (*) infixl 7 a :: !a !a -> a
```

arguments are strict,
compiler uses eager
evaluation for efficiency

macro

- a macro is a definition expanded at compile time
  - function types are not allowed here

```
(o) infixr 9 //:: (a -> b) (c -> a) -> c -> b
(o) f g :== \ x . f (g x)
```

```
One :== 1
```

  - more efficient code
  - compile time evaluation: no recursion
- also for types

```
:: Pair x y :== (x, y)
```

uniqueness

- in order to safely update a value (file, window, array, ..) you must be the only one having access to that object
  - the type system is used to ensure that
  - a * indicates uniqueness

```
(-<<) infixl 0 :: *File x -> *File | toString x
(-<<) file x = file <<< toString x <<< "\n"

Start :: *World -> *World
Start w1 = snd (fclose f3 w2)
where   (f1, w2) = stdio w1
        f2 = f1 -<< 7 -<< "hello world"
        f3 = f2 -<< 42
```

uniqueness information is needed for files and array manipulations

let definitions

- using let definitions and special scope rules this can be written more elegantly
  - ➢ use # as the keyword let

```
Start :: *World -> *World
Start w
 # (f, w) = stdio w
 # f      = f -<< 7 -<< "hello world"
 # f      = f -<< 42
 # w      = snd (fclose f w)
 = w
```

here the previous f is the last definition

always use the last definition

assignment 1

OVERLOADING

class

- a class is a set of different functions with the same name
- types are used to distinguish those functions

```
class nat a where
   add  :: a a -> a
   null :: a
```

multi-parameter type classes are fine

- adding an instance to the class

```
instance nat Int where
   add x y = x + y
   null = 0
```

- special syntax if class has only one function

```
class (+) infixl 6 a :: !a !a -> a
```

```
instance nat N where
   add Z y = y
   add (S n) m = S (add n m)
   null = Z
:: N = Z | S N
```

type constructor class

- class variables can have any kind

```
class stack s where
   push :: a (s a)   -> s a
   pop  ::   (s a)   -> s a
   top  ::   (s a)   -> a
   empty   ::   (s a)   -> Bool
instance stack [] where
   push e stack  = [e: stack]
   pop [e: rest] = rest
   top [e: rest] = e
   empty stack   = isEmpty stack
```

`s` gets an argument: `s` has kind * -> *
hence `stack` is a type constructor class

`[a]` has kind *
`[ ]` has kind * -> *

using classes

- a single function can work form many types
  - e.g. sum works for any type a with:
    - an operator +, and
    - a constant zero

```
sum :: [a] -> a | +, zero a
sum []    = zero
sum [a:x] = a + sum x
```

works for any type a with + and zero

- definitions needed for rational numbers

```
:: Rat = {q :: !Int, n :: !Int}
instance zero Rat where zero = {q = 0, n = 1}
instance + Rat where (+) x y =
  norm {q = x.q * y.n + y.q * x.n, n = x.n * y.n}
norm :: !Rat -> Rat
norm {q, n} = {q = q / x, n = n / x}
where x = gcd q n
```

a record

pass an argument that tells to do next

CONTINUATIONS

continuations

- functions have an additional argument(s) that tells what to do next

1. more control
2. efficiency

```
divide :: Int Int -> Int
devide x 0 = abort “devide by 0”
devide x y = x / y
```

exceptions would solve the problem, but they do not mix well with lazy evaluation

```
divide :: Int Int (Int -> x) x ->
devide x 0 succ fail = fail
devide x y succ fail = succ (x / y)
```

later we will show how to hide these arguments

continuations

```
:: Bin a = Leaf | Bin (Bin a) a (Bin a)
inorder :: (Bin a) -> [a]
inorder Leaf = []
inorder (Bin l a r) = inorder l ++ [a:inorder r]
```

++ is O(N)
hence `inorder` is O(N^2)

```
inorder :: (Bin a) -> [a]
inorder tree = scan tree []
where
   scan :: (Bin a) [a] -> [a]
   scan Leaf         c = c
   scan (Bin l a r) c = scan l [a: scan r c]
```

no O(N) ++ operator
hence `inorder` is O(N)

continuations 2

```
:: Bin a = Leaf | Bin (Bin a) a (Bin a)
```

- how do we make preorder and postorder?

```
preorder :: (Bin a) -> [a]
preorder t = scan t []
where
   scan Leaf c = c
   scan (Bin l a r) c = [a: scan l (scan r c)]

postorder :: (Bin a -> [a]
postorder t = scan t []
where
   scan (Bin l a r) c = scan l (scan r [a:c])
   scan Leaf c = c
```