Advanced Functional Programming

Tim Sheard
Dept of computer Science
Portland State University
Spring 2014

Ackknowledements

Many thanks to all those who suggested topics for discussion in this class:

- Mark Jones
- Tim Sauerwein
- Frank Taylor
- John Launchbury
- Jim Hook
- Jim Teisher

Course Mechanics

Instructor

Tim Sheard
sheard@cs.pdx.edu
Office 120-04 FAB
503-725-2410

Time: Spring Quarter 2014.

- Tuesday / Thursday 14:00 – 15:50
- classroom – FAB 40-07

Class Web page

http://web.cecs.pdx.edu/~sheard/course/AdvancedFP/DailyRecord.html

Grading Scheme

- **1 midterm Exam**
- **Programming Exercises**
  - Weekly programming assignments
  - checked off and graded
- **Final Project**
  - Chosen about 5 weeks into term
  - Due, Tuesday of finals week (June 10)
- **Tentative Grading Scheme**
  - Midterm 30%
  - Exercises 40%
  - Final Projects 30%

Materials

Required Text: None

Readings: To be assigned.

Usually a web link to papers that can be downloaded.
Sometimes handed out in class.

Resource Books:

Introductory texts in functional programming

The Haskell School of Expression
Paul Hudak, Cambridge University Press (Haskell)

Elements of Functional Programming
Chris Reade, Addisson Wesley (ML Language)

Introduction to Functional Programming
Richard Bird, Phil Wadler, Prentice Hall (Miranda-like)

Haskell: The Craft of Functional Programming
Simon Thompson, Addison Wesley (Haskell)

Other Resources

The Haskell Home Page

www.haskell.org

lots more links here!

The Haskell Report

http://www.haskell.org/onlinereport/

A gentle Introduction to Haskell

http://www.haskell.org/tutorial/

Academic Integrity

- **We follow the standard PSU guidelines for academic integrity.**
  – Discussion is good;
  – Items turned in should be your own, individual work.
- **Students are expected to be honest in their academic dealings. Dishonesty is dealt with severely.**
- **Homework. Pass in only your own work.**

Type Classes and Overloading

- **Readings for today's lecture**

  A Gentle Introduction to Haskell.

  Chapter 5: "Type classes and Overloading"
  - http://www.haskell.org/tutorial/classes.html

  Chapter 8: "Standard Haskell Classes"
  - http://www.haskell.org/tutorial/stdclasses.html

- **Research Papers about Type Classes**
  - Please skim the following 2 papers. Reachable from the background material web
  - http://web.cecs.pdx.edu/~sheard/course/AdvancedFP/2004/papers/index.html
  - A system of constructor classes: overloading and implicit higher-order polymorphism. Mark Jones
  - A theory of qualified types. Mark Jones

Other Related Papers

- **These papers can also be found in the background web page:**
  - Implementing Type Classes   Peterson and Jones
  - Type Classes with Functional Dependencies. Mark Jones
  - A Note on Functional Dependencies. Mark Jones
  - Implicit Parameters: Dynamic Scoping with Static Types. Lewis, Shields, Meijer, & Launchbury

What are type classes

Type classes are unique to Haskell

They play two (related) roles

Overloading

- A single name indicates many different functions.
- E.g. (+) might mean both integer and floating point addition.

Implicit Parameterization

- An operation is implicitly parameterized by a set of operations that are used as if they were globally available resources.

Attributes of Haskell Type Classes

Explicitly declared

- class and instance declarations

Implicit use

- Type inference is used to decide:
  - When a type class is needed.
  - What class is meant.

Uniqueness by type

- The inference mechanism must decide a unique reference to use.
- No overlapping-instances

The Haskell Class System

Think of a Qualified type as a type with a Predicate

Types which meet those predicates have "extra" functionality.

A class definition defines the type of the "extra" functionality.

An instance declarations defines the "extra" functionality for a particular type.

Example Class Definition

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y       = not (x==y)
```

Note default definition of (/=)

```
class (Eq a) => Ord a where
    compare     :: a -> a -> Ordering
    (<), (<=), (>=), (>)  :: a -> a -> Bool
    max, min         :: a -> a -> a
```

Properties of a class definition

```
class (Eq a) => Ord a where
 compare    :: a -> a -> Ordering
 (<), (<=), (>=), (>)  :: a -> a -> Bool
 max, min              :: a -> a -> a
```

- Class name is capitalized, think of this as the name of a type predicate that qualifies the type being described.
- Classes can depend on another class or in other words require another classes as a prerequisite
- The methods of a class are functions whose type can depend upon the type being qualified
- There can be more than one method.
- The methods can be ordinary (prefix) functions or infix operators.

Overloading – The Num Class

```
class (Eq a, Show a) => Num a where
    (+), (-), (*)  :: a -> a -> a
    negate         :: a -> a
    abs, signum    :: a -> a
    fromInteger    :: Integer -> a
    fromInt        :: Int -> a

    x - y           = x + negate y
    fromInt         = fromIntegral
```

Extending the Num Class with Complex

Make Complex numbers an instance of class Num.

```
data Complex = C Float Float
```

An instance of Num, must first be an instance of Eq and Show and provide methods for (+), (-), and (*) (amongst others).

First provide the numeric operators

```
complex_add (C x y) (C a b)= C (x+a) (y+b)

complex_sub (C x y) (C a b)= C (x-a) (y-b)

complex_mult (C x y) (C a b)
     = C (x*a - y*b) (x*b + a*y)
```

Num Instance

Then make the instance declarations

```
instance Eq(Complex) where
    (C x y) == (C a b) = x==a && y==b

instance Show(Complex)  where
    showsPrec = error "No show for complex"
    showList = error "No show for complex"

instance Num(Complex) where
  x + y = complex_add x y
  x - y = complex_sub x y
  x * y = complex_mult x y
```

Note that the Show instance is quite imprecise, but this will cause an error only if it is ever used

Implicit Parameterization

```
-- A simple functional language

type Var = String
data Term0 =
   Add0 Term0 Term0        -- x + y
 | Const0 Int              -- 5
 | Lambda0 Var Term0       -- \ x -> x + 2
 | App0 Term0 Term0        -- f x
 | Var0 Var                -- x

data Value0 =
   Int0 Int
 | Fun0 Var Term0 Env0

data Env0 = E0 [(Var,Value0)]
```

A Simple Evaluator

```
eval0 :: Env0 -> Term0 -> Value0
eval0 (e @ (E0 xs)) t =
  case t of
   Add0 x y -> plus (eval0 e x) (eval0 e y)
   Const0 n -> Int0 n
   Var0 s -> look xs s
   Lambda0 s t -> Fun0 s t e
   App0 f x -> apply (eval0 e f) (eval0 e x)

 where plus (Int0 x) (Int0 y) = Int0 (x+y)
       look ((x,v):xs) s =
            if s==x then v else look xs s
       apply (Fun0 v t e) x = eval0 (extend e v x) t
       extend (E0 xs) v x = (E0((v,x):xs))
```

Make the environment abstract

```
data Term1 =
   Add1 Term1 Term1
 | Const1 Int
 | Lambda1 Var Term1
 | App1 Term1 Term1
 | Var1 Var

data Value1 e =
   Int1 Int
 | Fun1 Var Term1 e
```

Abstract Evaluator

```
eval1 :: e -> (e -> Var -> Value1 e) ->
        (e -> Var -> Value1 e -> e) ->
         Term1 -> Value1 e
eval1 e look extend t =
  case t of
   Add1 x y -> plus (eval e x) (eval e  y)
   Const1 n -> Int1 n
   Var1 s -> look e s
   Lambda1 s t -> Fun1 s t e
   App1 f x -> apply (eval e f) (eval e x)

 where plus (Int1 x) (Int1 y) = Int1 (x+y)
       apply (Fun1 v t e) x = eval (extend e v x) t
       eval e x = eval1 e look extend x
```

Add something new

```
data Term2 =
    Add2 Term2 Term2
  | Const2 Int
  | Lambda2 Var Term2
  | App2 Term2 Term2
  | Var2 Var
  | Pair2 Term2 Term2       -- (3, 4+5)
  | Let2 Pat Term2 Term2    -- let (x,y) = f x
   in x+y

data Value2 e =
    Int2 Int
  | Fun2 Var Term2 e
  | Prod2 (Value2 e) (Value2 e)

data Pat = Pat Var Var
```

Complex Abstract Eval

```
eval2 :: e -> (e -> Var -> Value2 e) ->
              (e -> Var -> Value2 e -> e) ->
 (e -> Pat -> Value2 e -> e) -> Term2 -> Value2 e
eval2 e look extend extpat t =
  case t of
   Add2 x y -> plus (eval e x) (eval e  y)
   Const2 n -> Int2 n
   Var2 s -> look e s
   Lambda2 s t -> Fun2 s t e
   App2 f x -> apply (eval e f) (eval e x)
   Pair2 x y -> Prod2 (eval e x) (eval e y)
   Let2 p x y -> eval (extpat e p (eval e x)) y

 where plus (Int2 x) (Int2 y) = Int2 (x+y)
       apply (Fun2 v t e) x = eval (extend e v x) t
       eval e x = eval2 e look extend extpat x
```

Using a Class

```
-- Lets capture the set of operators
-- on the abstract environments
-- as a type class

class Environment e where
  look :: e -> Var -> Value2 e
  extend:: e -> Var -> Value2 e -> e
  extpat :: e -> Pat -> Value2 e -> e
```

Simple Abstract Eval

```
eval3 :: Environment e => e -> Term2 -> Value2 e
eval3 e t =
  case t of
   Add2 x y -> plus (eval3 e x) (eval3 e  y)
   Const2 n -> Int2 n
   Var2 s -> look e s
   Lambda2 s t -> Fun2 s t e
   App2 f x -> apply (eval3 e f) (eval3 e x)
   Pair2 x y -> Prod2 (eval3 e x) (eval3 e y)
   Let2 p x y -> eval3 (extpat e p (eval3 e x)) y

 where plus (Int2 x) (Int2 y) = Int2 (x+y)
       apply (Fun2 v t e) x = eval3 (extend e v x) t
```

Instantiating the Class

```
data Env3 = E3 [(Var,Value2 Env3)]

instance Environment Env3 where
  look (E3((x,y):xs)) v = ]
      if x==v then y else look (E3 xs) v
  extend (E3 xs) v x = E3 ((v,x):xs)
  extpat (E3 xs) (Pat x y) (Prod2 a b) =
      E3 ((x,a):(y,b):xs)
```

Different Instantiation

```
data Env4 = E4 (Var -> Value2 Env4)

instance Environment Env4 where
  look (E4 f) v = f v
  extend (E4 f) v x =
      E4(\ y -> if y==v then x else f y)
  extpat (E4 f) (Pat x y) (Prod2 a b) =
    E4(\ z -> if x==z
                then a
                else if y==z
                        then b
                        else f z)
```

Using Eval

```
-- let (f,g) = (\ x -> x+1, \ y -> y + 3)
-- in f (g 5)

prog =
  Let2 (Pat "f" "g")
       (Pair2 (Lambda2 "x" (Add2 (Var2 "x") (Const2 1)))
              (Lambda2 "y" (Add2 (Var2 "y") (Const2 3))))
       (App2 (Var2 "f") (App2 (Var2 "g") (Const2 5)))

ans = eval3 (E3 []) prog

ans2 = eval3
         (E4 (\ x -> error "no such name"))
         prog
```

What do Type Classes Mean

A Type class is an implicit parameter

The parameter captures all the functionality of the class (it's methods)

The library passing transform

The type inference mechanism infers when a function needs a type class

```
eval3 :: Environment e => e -> Term2 -> Value2 e
```

The mechanism transforms the program to pass the extra parameter around

Compare

```
class Environment e where
  look :: e -> Var -> Value2 e
  extend:: e -> Var -> Value2 e -> e
  extpat :: e -> Pat -> Value2 e -> e


data EnvironmentC e =
  EnvC {lookM :: e -> Var -> Value2 e,
        extendM :: e -> Var -> Value2 e -> e,
        extpatM :: e -> Pat -> Value2 e -> e
       }
```

Explicit Library Parameter

```
eval4 :: EnvironmentC a -> a -> Term2 -> Value2 a
eval4 d e t =
  case t of
   Add2 x y -> plus (eval4 d e x) (eval4 d e  y)
   Const2 n -> Int2 n
   Var2 s -> lookM d e s
   Lambda2 s t -> Fun2 s t e
   App2 f x -> apply (eval4 d e f) (eval4 d e x)
   Pair2 x y -> Prod2 (eval4 d e x) (eval4 d e y)
   Let2 p x y -> eval4 d
                  (extpatM d e p (eval4 d e x)) y

 where plus (Int2 x) (Int2 y) = Int2 (x+y)
       apply (Fun2 v t e) x =
              eval4 d (extendM d e v x) t
```

Instances?

Note the recursion

```
e3Dict = EnvC
  { lookM = \ (E3((x,y):xs)) v ->
              if x==v then y else lookM e3Dict (E3 xs) v
  , extendM = \ (E3 xs) v x -> E3((v,x):xs)
  , extpatM = \ (E3 xs) (Pat x y) (Prod2 a b) ->
                E3((x,a):(y,b):xs) }
e4Dict = EnvC
  { lookM = \ (E4 f) v -> f v
  , extendM = \ (E4 f) v x ->
                E4(\ y -> if y==v then x else f y)
  , extpatM = \ (E4 f) (Pat x y) (Prod2 a b) ->
               E4(\ z -> if x==z
                            then a
                            else if y==z then b else f z)}

ans3 = eval4 e3Dict (E3 []) prog
ans4 = eval4 e4Dict (E4 (\ x -> error "no such name")) prog
```