CS 613 – Design and Implementation of Functional Programming Languages

Course Contents

1. **Motivation**

   - Why functional programming matters?
     - Tries to pin down the reason why functional programs should be taken seriously.
     - Facilitates the process of abstraction.
     - Abstraction - is the process of taking away or removing characteristics from something in order to reduce it to a set of essential characteristics
     - Example from imperative language – procedural abstraction, type abstraction.
     - Examples of abstractions in FP – higher order functions, polymorphic types, laziness.
     - Abstraction leads to reuse. The pyramid diagram.
   - A real world application of functional programming - software prototyping.
     - An immediate use of FP.
     - Experiment conducted by US Navy. Language experts invited to write a prototype for a small but important software. Haskell came winner. Why?

2. **Guided tour of Haskell, illustrating**

   - Functions as first class objects.
     - Functions can be passed as arguments and returned as results. The difference between functions and data is abstracted away.
     - Useful first order functions can be crafted as instances of a single higher order function.
     - Example of higher order function - twice:

```
twice f x = f (f x)
main = (twice twice sq) 2

(twice twice sq) 2
=> (twice (twice sq)) 2
=> twice (twice sq) 2
=> (twice sq) ((twice sq) 2)
=> (twice sq) (twice sq 2)
=> (twice sq) (sq (sq 2))
=> sq (sq (sq (sq 2)))

twice :: (b -> b) -> b -> b
```

- Referential transparency.
  - Identical expressions in the same lexical scope should evaluate to the same value.
  - Useful for reasoning about programs.
  - The expression in imperative languages. A larger area where values bound to variables do not change.

```
t1 = a + b              x = (a+b)*c + (a+b)*c*d
t2 = t1 * c
t1 = t2 * d
t2 = t1 + t2
x = t2
```

  - Commutativity. Does not hold in Java. `i + (i++)` is not the same as `(i++) + i`.
  - Optimization - CSE in imperative languages cannot be done in presence of side effects.

```
x = a + f(x)        y = a + f(x)
...                 ...
        ... = a + f(x)
```

cannot be replaced by

```
t = a + f(x)        t = a + f(x)
x = t                  y = t
      ... = t
```

in the presence of side effects.

- Laziness.
  - A spreadsheet example:

s			
	a	b	c
1	c3+c2		
2	a3*b2	2	a2+b2
3	7	a2+a3	

r			
	a	b	c
1	37		
2	14	2	16
3	7	21	

A Haskell solution:

```
r = array (bounds s) [(i,j) := eval r (s ! (i,j)) | (i,j) <- indices s]
```

The value of r(i,j) *depends on the value of* r *at some indices which are dictated by* s(i,j).

The corresponding C solution does not work

```
for (i=1, i < dim1(s), i++)
        for (j=1, j < dim2(s), j++)
                r[i,j] = eval r s[i,j];
```

The value of r(i,j) *is computed using the current value of* r *at some indices which are dictated by* s(i,j).

  - A functional program captures essential dependences.
  - Abstracts out restriction of control flow.
  - Uses: clean separation of concerns in generate and test situations.

- Data types and pattern matching.

- data Nat = Zero | Succ Nat

  Zero, Succ – data constructors. The corresponding Pascal Definition is

  ```
  type Nattype = (Zero, Succ)
  type Nat = case tag:nattype of
                  Zero: ()
                  Succ: (pred: Nat)
              end
  ```

- Pattern matching in function definition:

  ```
  add Zero n = n
  add (Succ n) m = add n (Succ m)
  ```

  In Pascal

  ```
  function add(n,m : Nat): Nat
  var i : Nat
  begin
      if m.tag = Zero then add := m
      else begin
              i.tag := Succ;
              i.pred := m;
              add := add(n.pred, i)
          end
  end;
  ```

• Type variables and parametric polymorphism.

```
List a = Nil | Cons a (List a)

length [ ] = 0
length (x:xs) = 1 + length xs

length :: List a -> Int
```

- Classes and overloading.

  ```
  elem x [ ] = False
  elem x (y:ys) = (x == y) || (x < y) && (elem x ys)

  elem :: (Ord a) => a -> List a -> Bool
  ```

  Enables us to say: The type is a -> List a -> Bool, but only for those a's on which an order is defined (which belong to the class Ord)

- IO in Haskell.
  - What is the problem with IO?

    ```
    inputLine (is) ++ inputLine (is)
    ```

    The values of the two inputLine (is) are likely to be different.

  - Haskell has side-effect free IO. Manages to model the above not as

    inputLine :: InputStream -> String, but as

    ```
    inputLine :: InputStream -> World -> (String, World)
    let
       w = initial world
       (l1,w1) = inputLine(is,w)
       (l2,w2) = inputLine(is,w1)
    in
      l1 ++ l2
    ```

- **Lambda Calculus**
  - Motivation
    * A minimal but complete language
    * Originally introduced to capture the notion of computbility.

        * Used to give semantics of FP. Sematics of a PL is usually given in terms of
            · mathemetical notions - away from operational intutions
            · compilers - non-standard
            · in terms of a language which is very small in size; the language becomes standardised through usage.
        * Lambda Calculus is convenient, because of its size and rewrite model of execution.
        * can be used to implement FP
    - Syntax of Lambda terms
    - Alpha, beta and delta conversions
    - Normal forms, applicative order and normal order reductions
    - Church-Rosser theorems
    - Computing using Lambda Calculus
    - Y combinator and recursion.
- **Type Inference**
    - Hindley-Milner type checking algorithm and its extension to handle type classes.
- **Monads and IO**
    - Introduction to Monads
    - Side effect free IO using Monads

3. **Implementation of FP Languages**

- Backend issues in implementation of FP languages.

  FPg `->` Syntax and parsing `->` Type-checking `->` Overloading resolution `->` Conversion to lambda term `->` Conversion to graph `->` Interpretation (direct graph reduction) `->` Compilation (generation of code to do graph reduction)
- The G-Machine

4. **Selected topics in Functional Programming**

- Bird-Meertens Formalism
- Abstract Interpretation
- Functional Programming using Monads.

Grading

- 50% for end semester performance
- 15% for mid semester performance
- 20% for two programming assignments
- 15% for quizzes (maybe three of them)

References

1. John Hughes, *Why Functional Programming Matters?*, Technical Memo of Institutionen for Datavetenskap, Chalmers Tekniska Hogskola, Goteborg, Sweden.**(Course web page)**

2. Paul Hudak and Mark. P. Jones, *Haskell vs. Ada vs. C++ vs. Awk vs.... An experiment in Software Prototyping Productivity* , Technical report, Department of Computer Science, Yale University, New Haven, CT 06518. **(Course web page)**

3. Philip Wadler, *Why no one uses functional languages.* **(Course web page)**

4. Paul Hudak, John Peterson and Joseph Fasel. *A Gentle Introduction to Haskell Version 98.* **(Course web page)**

5. S. Peyton Jones and John Hughes (eds), *Haskell 98: A Non-strict Purely Functional Language* **(Course web page)**

6. R.Bird and P.Wadler. *Introduction to Functional Programming*, Prentice Hall, New York, 1988. **(Central Library)**

7. H.P. Barendregt. *The impact of lambda calculus in logic and computer science.* **(Course web page)**

8. H.P. Barendregt. *The lambda calculus*, its syntax and semantics, Elsevier Science Publishers B.V. Amsterdam, 1984. **(Central Library)**

9. J.R.Hindley and J.P. Seldin. *Introduction to Lambda Calculus and Combinators.* Cambridge University Press, 1984.

10. Cordellia Hall, Kevin Hammond, Simon Peyton-Jones & Phil Wadler. *Type classes in Haskell* **(Course web page)**

11. Henk Barendregt. Lambda Calculus with Types, From Handbook of Logic in Computer Science, Eds. S. Abramsky, D. Gabbay, T.S.L. Maibaum, Oxford University Press. **(Course web page, Departmental library)**

12. Michael I. Schwartzbach. *Polymorphic Type Inference*, BRICS Lecture Series Technical Report, LS-95-3, June 1995. **(Course web page)**

13. S.Peyton Jones. *The Implementation of Functional Programming languages*, Prentice Hall, New York, 1987. **(Central Library)**

14. Simon Peyton Jones and David R. Lester, *Implementing functional languages: A Tutorial.*

15. Philip Wadler, *How to declare an Imperative*, **(Course web page)**

16. Philip Wadler, *The essence of functional programming.* **(Course web page)**

17. Jeremy Gibbons, *Calculating Functional Programs.* **(Course web page)**

18. Graham Hutton, *A Tutorial on the Universality and Expressiveness of fold.* **(Course web page)**

Type constructor

Single line comment

Multi line comment

Data Constructor

Datatype declaration (Bintree a)

```haskell
--data List a = Cons a (List a) | Nil

{- Does not have to be declared. Haskell standard
   prelude already declares it -}

data BinTree a = Node a (BinTree a) (BinTree a)
               | Nulltree
               deriving Show
```

Operator

Function definition (inOrder)

Patterns

```haskell
inOrder Nulltree = []
inOrder (Node i t1 t2) = inOrder t1 ++ [i]  ++ inOrder t2
```

Type declaration

Context

```haskell
insert :: Ord a => a -> BinTree a -> BinTree a
insert i Nulltree = Node  i  Nulltree  Nulltree
insert i (Node i' t1 t2) = if ( i > i') then Node i' t1 ( insert  i  t2)
                           else Node  i' (insert  i  t1)  t2


t1 = Node 4 t2 t3
t2 = Node 2 (Node 1 Nulltree Nulltree) (Node 5 Nulltree Nulltree)

t3 = Node 3 Nulltree   (Node  6  Nulltree  Nulltree)
y = inOrder z
z = insert 1 t1
```

```
--data List a = Cons a (List a) | Nil

{- Does not have to be declared. Haskell standard
   prelude already declares it -}

data BinTree a = Node a (BinTree a) (BinTree a)
                | Nulltree
                deriving Show

inOrder Nulltree = []
inOrder (Node i t1 t2) = inOrder t1 ++ [i]  ++ inOrder t2

insert :: Ord a => a -> BinTree a -> BinTree a
insert i Nulltree = Node  i  Nulltree  Nulltree
insert i (Node i' t1 t2) = if ( i > i') then Node i' t1 ( insert  i  t2)
                                else Node  i' (insert  i  t1)  t2

t1 = Node 4 t2 t3
t2 = Node 2 (Node 1 Nulltree Nulltree) (Node 5 Nulltree Nulltree)
t3 = Node 3 Nulltree   (Node  6  Nulltree  Nulltree)
y = inOrder z
z = insert 1 t1
```