CSE 3302
Programming Languages
Lecture 8: Functional Programming

(based on the slides by Tim Sheard)
Leonidas Fegaras
University of Texas at Arlington

Functional Programming Languages

- Programs consist of functions with no side-effects
- Functions are first class values
- Build modular programs using function composition
- No accidental coupling between components
- No assignments, statements, for-loops, while-loops, etc
- Supports higher-level, declarative programming style
- Automatic memory management (garbage collection)
- Emphasis on types and type inference
  - Built-in support for lists and other recursive data types
  - Type inference is like type checking but no type declarations are required
    * Types of variables and expressions can be inferred from context
  - Parametric data types and polymorphic type inference
- Strict vs lazy functional programming languages

Lambda Calculus

- The theoretical foundation of functional languages is lambda calculus
  - Formalized by Church in 1941
  - Minimal in form
  - Computationally complete
- Syntax: if e₁, e₂, and e are expressions in lambda calculus, so are
  - Constant: c
  - Variable: v
  - Application: e₁ e₂
  - Abstraction: λv. e
- Bound vs free variables
- Beta reduction:
  - (λv. e₁) e₂ → e₁[e₂/v]
  - (e₁ but with all free occurrences of v in e₁ replaced by e₂)
  - need to be careful to avoid the variable capturing problem (name clashes)

Examples

λx . + 1 x
In Lisp: (lambda (x) (+ 1 x))
In Haskell: \x -> x+1

Beta reduction:
(λ x . + 1 x) 2 => (+1 2) => 3
(λ x . (+((λ y . ((λ x . * x y) 2)) x) y)
=> (λ x . (+((λ y . (* 2 y)) x) y)
=> (λ x . (* 2 x) y)
Reductions

- REDucible EXpression (redex)
  - an application expression is a redex
  - abstractions and variables are not redexes
- Use beta reduction to reduce
  - \((\lambda x . x) x\) is reduced to \(x\)
- Normal form = no more reductions are available
- Reduction is confluent (has the Church-Rosser property)
  - normal forms are unique regardless of the order of reduction
- Weak normal forms (WNF)
  - no redexes outside of abstraction bodies
- Call by value (eager evaluation): WNF + leftmost innermost reductions
- Call by name: WNF + leftmost outermost reductions (normal order)
- Call by need (lazy evaluation): call by name, but each redex is evaluated at most once
  - terms are represented by graphs and reductions make shared subgraphs

Polymorphic Lambda Calculus

- Typed lambda calculus:
  - Constant: \(c\)
  - Variable: \(v\)
  - Application: \(e_1, e_2\)
  - Abstraction: \(\lambda v . e\)
- Types are:
  - Base type: \(\text{int}\)
  - Function: \(t_1 \rightarrow t_2\)
- Polymorphic lambda calculus:
  - Constant: \(c\)
  - Variable: \(v\)
  - Application: \(e_1, e_2\)
  - Abstraction: \(\lambda v . e\)
  - Type abstraction: \(\forall v . e\)
  - Type instantiation: \(e[t]\)
- Types are:
  - Base type: \(\text{int}\)
  - Function: \(t_1 \rightarrow t_2\)

Functional Languages

- Functional languages = polymorphic lambda calculus + syntactic sugar
- Functional languages support parametric (generic) data types
  - data List a = Nil
    - Cons a (List a)
  - data Tree a b = Leaf a
    - Node b (Tree a b) (Tree a b)
  - Cons 1 (Cons 2 Nil)
  - Cons "a" (Cons "b" Nil)
- Polymorphic functions:
  - append (Cons x r) s = Cons x (append r s)
  - append Nil s = s
  - The type of append is \(\forall a . (\text{List} a) \rightarrow (\text{List} a) \rightarrow (\text{List} a)\)

Type Inference

- Functional languages need type inference rather than type checking
  - \(\lambda v . e\) requires type checking
  - \(\forall v . e\) requires type inference (need to infer the type of \(v\))
- Type inference is undecidable in general
- Solution: type schemes (shallow types):
  - \(\forall a, \forall b, \ldots \forall a, t\) no other universal quantification in \(t\)
  - \((\forall b . b \rightarrow \text{int}) \rightarrow (\forall b . b \rightarrow \text{int})\) is not shallow
- When a type is missing, then a fresh type variable is used
- Type checking is based on type equality; type inference is based on type unification
  - A type variable can be unified with any type
- Example in Haskell:
  - let \(f = \lambda x . x\) in \((f 5, f "a")\)
  - \(\lambda x . x\) has type \(\forall a . a \rightarrow a\)
- Cost of polymorphism: polymorphic values must be boxed (pointers to heap)
Haskell

Web site: http://haskell.org/
Tutorials: http://haskell.org/tutorial/
http://haskell.org/haskellwiki/Learning_Haskell

> hugs

```
| Hugs 98: Based on the Haskell 98 standard |
| World Wide Web: http://haskell.org/hugs |
| Version: September 2006                   |
```

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type ? for help
Hugs>

Functions

- Functions are defined in files and loaded into Hugs: 
  - :! file.hs

- Functions on numbers
  - Type of numbers: Int and Float 
  - Conversion functions: fromInteger round

- Functions on Booleans
  - Relational operators < > <= => == /=
  - Combinators && || not

Examples

? 5 > 7
False
? 1==4
False

Lists in Haskell

? [1,2] ++ [4]
[1, 2, 4]

? null [2]
False

? take 3 [1,2,3,4,5,6]
[1, 2, 3]

? drop 3 [1,2,3,4,5,6]
[4, 5, 6]

Constructing Lists

- The Empty List []
- The "Cons" (: ) Constructor
  ? 3 : [3,4,5]
  [3, 3, 4, 5]

- The Dot Dot notation
  ? [1 .. 4]
  [1, 2, 3, 4]

- The Comprehension Notation
  ? [x + 1 | x <- [2..4]]
  [3, 4, 5]

  ? [(x,y) | x <- [1..2], y <- [3,5,7]]
  [(1,3), (1,5), (1,7), (2,3), (2,5), (2,7)]

  ? [ x * 2 | x <- [1..10], even x]
  [4, 8, 12, 16, 20]
Taking Lists Apart

? head [1,2,3]
1

? tail [1,2,3]
[2, 3]

? take 2 [1,2,3]
[1,2]

? drop 2 [1,2,3]
[3]

Recursive Functions

• Simple recursive functions:

  fact n
  = if n == 0 then 1 else n*(fact (n-1))

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

• Hugs session

  ? fact 3
  6

  ? len [1,2,3,4]
  4

Rules for Patterns

• All the patterns (on the left) should have compatible types
• The cases should (but are not required to) be exhaustive
• There should be no ambiguity as to which case applies
• Ordering fixes ambiguity if there is any (the first match is chosen)

• A Pattern is:
  – A variable
  – A constructor applied to patterns x:xs or Branch(x, y, z)
  – A constant
  – A tuple of patterns (x,3,y:ys)

Local Definitions

• Local Definitions: Where

  len2 [] = 0
  len2 (x:xs) = one + z
  where z = len2 xs
       one = 1

  where
  a = f x y
  b = g z

  y b = g z

• Indentation matters! Same sequence of tokens but different meaning

• Location of <newline> makes a big difference
• Rule of thumb: Definitions at the same scope should be indented equally far
Function Types & Prototyping

• Typing
  \( f \ a \ b \ c = a + b + c + 1 \)
  has type
  \( f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \)
  Read as
  \( f :: \text{Int} \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})) \)

• Prototyping
  \( \text{plus} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \)
  \( \text{plus} \ x \ y = \text{if} \ x == 0 \text{ then } y \text{ else } 1 + (\text{plus} (x-1) \ y) \)
  \( \text{myand} :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \)
  \( \text{myand True False} = \text{False} \)
  \( \text{myand True True} = \text{True} \)
  \( \text{myand False False} = \text{False} \)
  \( \text{myand False True} = \text{False} \)

Overloading and Classes

? :type difference
difference :: Num a => a -> a -> a

? difference 3 4
-1

? difference 4.5 7.8
-3.3

? The class Num is a predicate on types

Ways to Create Functions

• By defining: \( \text{pluseone} \ x = x+1 \)
  ? \text{pluseone} 3
  4

• By operator section
  ? (3+) 5
  8
  ? \text{map} (3+) [2,3,4]
  [5, 6, 7]

• By lambda expression
  ? (\(x \rightarrow x+2\)) 5
  7
  ? \text{map} (\(x \rightarrow x*2\)) [2,3,4]
  [4, 6, 8]

Creating Functions (cont.)

• By currying (partial application)
  ? \text{plus} 3
  \text{plus} 3
  ? :type (\text{plus} 3)
  \text{plus} 3 :: \text{Int} \rightarrow \text{Int}
  ? \text{map} (\text{plus} 3) [3,4]
  [6, 7]

• By composition
  ? \text{map} (\text{head . tail}) [(2,3,4),(4,5,6)]
  [3, 5]

• By combinator: \( k \times y = x \)
  – Functions which return functions
  ? \text{map} (k 3) [1,2,3]
  [3, 3, 3]
Expressions, Values, and Types

- There are three types of distinguished entities in Haskell
- Expressions
  - 5 + 3
  - \texttt{len} \([1,2,3]\)
  - \texttt{rev} \([2,9]\)
  - \texttt{len}
- Values
  - 8
  - 3
  - \([9,2]\)
  - \texttt{<<function>>}
- Types
  - \texttt{Int}
  - \texttt{Int}
  - \texttt{[Int]}
  - \texttt{[a] \rightarrow Int}

Typing Tuples

```haskell
?:type (1,"x",True)
(1,"x",True): (Int,String,Bool)
?:type (1,2)
(1,2): (Int,Int)
?:type (2, ("x",3))
(2, ("x",3)): (Int, (String,Int))
- Note: \((\texttt{Int}, (\texttt{Char}, \texttt{Int})) \leftrightarrow (\texttt{Int}, (\texttt{Char}, \texttt{Int}))\)
```
- Pattern matching on tuples
  - \((\lambda (a,b) \rightarrow a) \ (2,3)\)
  - 2
  - \((\lambda (a,b) \rightarrow b + 2) \ (2,3)\)
  - 5

Tuples

- Heterogeneous Collection of a fixed width
- Tuple Expressions
  - \((5+3, \text{not True})\)
  - \((\text{tl} \ [1,2,3], [2]+[3,4], \text{“abc”})\)
- Evaluate to Tuple Values
  - \((8, \text{False})\)
  - \((\text{[2,3], [2,3,4], “abc”})\)
- And have Tuple Types
  - \((\text{Int}, \text{Bool})\)
  - \((\text{[Int], [Int], String})\)

Used when returning multiple values

- Function that splits a list into two pieces at some particular position
  - \texttt{split} 2 [1,2,3,4] \rightarrow ([1,2],[3,4])
  - \texttt{split} 3 [1,2,3,4,5,6,7] \rightarrow ([1,2,3],[4,5,6,7])
  - \texttt{split} 0 [1,2,3] \rightarrow ([],[1,2,3])
- \texttt{split} 0 x = ([],x)
- \texttt{split n} [] = ([],[n])
- \texttt{split n} (x:xs) = (x:ys,zs)
  - where \((ys,zs) = \text{split} \ (n-1) \ xs\)
Polymorphism

• Consider: `tagl x = (1,x)`
  ```haskell```
  tagl :: a -> (Int,a)
  ```haskell```
• Other functions have types like this consider `++`
  ```haskell```
  (++) :: [a] -> [a] -> [a]
  ```haskell```
• What are some other polymorphic functions and their types?
  - `id` ::
  - `reverse` ::
  - `head` ::
  - `tail` ::
  - `()` ::
  - `split` ::

Functions as arguments

• Consider:
  ```haskell```
  mymap f [] = []
  mymap f (x:xs) = (f x):(mymap f xs)
  ```haskell```
• The parameter `f` is a function
• What is the type of `mymap`?
  ```haskell```
  mymap :: (a -> b) -> [a] -> [b]
  ```haskell```
• Result?
  ```haskell```
  map add1
  ```haskell```
  map (\x -> 3) [1,2,3]

Abstracting

```haskell```
  myfoldr op e [] = e
  myfoldr op e (x:xs) =
      op x (myfoldr op e xs)
  ```haskell```
• Example on Booleans
  ```haskell```
  myand True False = False
  myand True True = True
  myand False False = False
  myand False True = False
  ```haskell```
• Pattern may contain constructors
• Constructors are always capitalized
• Order Matters
  - Variables in patterns match anything
  ```haskell```
  myand2 True True = True
  myand2 x y = False
  ```haskell```
  - What happens if we reverse the order of the two equations above?
Lazy Evaluation

- Consider:
  repeat x = x : (repeat x)
  ? repeat 't'
  "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa^C(Interrupted!)

- But we can use a finite prefix
  ? take 10 (repeat 't')
  "tttttttttt"

- iterate creates infinite lists
  ? take 5 (iterate (+1) 1)
  [1, 2, 3, 4, 5]
  ? take 5 (iterate (*2) 1)
  [1, 2, 4, 8, 16]
  ? take 5 (iterate (/10) 142)
  [142, 14, 1, 0, 0]

Defining New Datatypes

- Kinds of datatypes
  - enumerated types
  - records (or products or struct)
  - variant records (or sums)
  - pointer types
  - arrays

- Haskell's `data` declaration provides many of these kinds of types in a uniform way which abstracts from their implementation details

- The `data` declaration defines new functions and constants, which provide an abstract interface to the newly defined type

The Data Declaration

- Enumeration Types
  data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
deriving Eq

- The names on right hand side are constructor constants and are the only elements of the type
  valday 0 = Sun
  valday 1 = Mon
  valday 2 = Tue
  valday 3 = Wed
  valday 4 = Thu
  valday 5 = Fri
  valday 6 = Sat

  ? valday 3
  Wed

Constructors & Patterns

- `data` defined types define new constructors and can be accessed by patterns

- Constructors without arguments are constants

- Example using `case`
  
  dayval x =
  case x of
  Sun -> 0
  Mon -> 1
  Tue -> 2
  Wed -> 3
  Thu -> 4
  Fri -> 5
  Sat -> 6
Patterns in Declarations

- In a declaration patterns can be used. Possible to have many lines for a definition if each pattern is distinct.

  ```
  data Temp = Celsius Float
  | Fahrenheit Float
  | Kelvin Float
  ```

- Use patterns to define functions over this type:

  ```
  toKelvin (Celsius c) = Kelvin(c + 273.0)
  toKelvin (Fahrenheit f) =
  Kelvin( (f - 32.0) * (5.0/9.0) + 273.0 )
  toKelvin (Kelvin k) = Kelvin k
  ```

Shape types from the Text

- `data Shape = Rectangle Float Float
  | Ellipse Float Float
  | RtTriangle Float Float
  | Polygon [ (Float,Float) ]
  deriving Show`

- Deriving Show
  - tells the system to build a `show` function for the type `Shape`

- Using Shape: functions returning shape objects

  ```
  circle radius = Ellipse radius radius
  square side = Rectangle side side
  ```

Functions over Shape

- Functions over shape can be defined using pattern matching

  ```
  area :: Shape -> Float
  area (Rectangle s1 s2) = s1 * s2
  area (Ellipse r1 r2) = pi * r1 * r2
  area (RtTriangle s1 s2) = (s1 *s2) / 2
  area (Polygon (v1:pts)) = polyArea pts
  where polyArea :: [ (Float,Float) ] -> Float
       polyArea (v2 : v3 : vs) = triArea v1 v2 v3 +
                                polyArea (v3:vs)
       polyArea [] = 0
  ```
Useful Functions on lists

• Higher Order Functions
• Let:

\[ x = [1, 2, 3, 4] \]
\[ y = ["a", "b", "c", "d"] \]
• Then:

\[
\begin{align*}
\text{map } f \ x &= [f \ 1, f \ 2, f \ 3, f \ 4] \\
\text{filter even } x &= [2, 4] \\
\text{foldr} \ (+) \ e \ x &= \\
&= 1 + (2 + (3 + (4 + e))) \\
\text{foldl} \ (+) \ e \ x &= \\
&= ((e + 1) + 2) + 3) + 4 \\
\text{iterate } f \ 1 &= \\
&= [1, f \ 1, f(f \ 1), f(f(f \ 1)), \ldots ]
\end{align*}
\]

Higher-Order Functions

• Map a function \( f \) over every element in a list

\[
\begin{align*}
\text{map: } (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
\text{map } f [] &= [] \\
\text{map } f (a:x) &= (f \ a) : \text{map } f \ x
\end{align*}
\]

e.g. \( \text{map } (x \rightarrow x+1) [1, 2, 3, 4] = [2, 3, 4, 5] \)

• Replace all cons list constructions with the function \( c \) and the nil with the value \( z \)

\[
\begin{align*}
\text{foldr: } (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \\
\text{foldr } c \ z [] &= z \\
\text{foldr } c \ z \ (a:s) &= c \ a \ (\text{foldr } c \ z \ s)
\end{align*}
\]

e.g. \( \text{foldr } (+) 0 [1, 2, 3] = 6 \)

e.g. \( \text{append } x \ y = \text{foldr } (+) \) \( y \ x \)

e.g. \( \text{map } f \ x = \text{foldr } (\text{a } \rightarrow \text{(f a)} : []) \) \( x \)

String and Char

• String is defined to be a list of char: \([\text{Char}]\)

? "abc"

abc

? ['a', 'b', 'c']

abc
• Note:

- "a" is a string (A List of Char with one element)
- 'a' is a Char
- 'a' is an operator
• Since String = \([\text{Char}]\) all the list operations work on strings

? “abc” ++ “xyz”

abcxyz

? reverse “abc”

cba