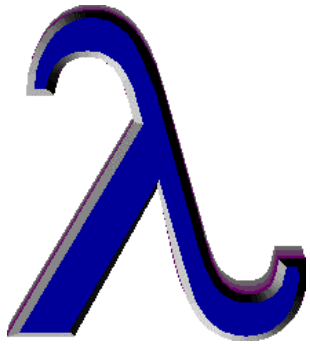
Haskell

A Purely Functional Language



Ryan Richardson

Wei Yu

Haskell

- Introduction to Haskell
- The objectives of Haskell
- Some features about Haskell
- Conclusion

Functional Languages

- What is functional Language?
  - A single expression, executed by evaluating the expression
- Some functional languages
  - LISP: the first functional language
  - Scheme: a small, static-scoped dialect of LISP
  - ML: a strongly typed functional language
  - Haskell: a purely functional language

Functional Languages (cont.)

- A comparison to imperative languages
  - Brevity
  - Ease of understanding
  - Powerful abstractions
  - More ...

Introduction to Haskell

- Where Haskell comes from?
- Why use Haskell? You can have:
  - Substantially increased programmer productivity
  - Shorter, clearer, and more maintainable code
  - Fewer errors, higher reliability
  - A smaller "semantic gap" between the programmer and the language

Objectives of Haskell

- More concise
  - 2 to 10 times shorter than C, on average
- Closely models underlying algorithms

[examples]

Features of Haskell

- Lazy evaluation
- Infinite data structures
- Functions as first-class values

Lazy Evaluation

- What is lazy evaluation?
  - no subexpression is evaluated until its value is known to be required
- Similar ideas in other languages
  - Unix pipes [example]
  - short-circuit evaluation [example]

Lazy Evaluation (cont.)

- Example of lazy evaluation [example]
- Advantages of lazy evaluation
  - frees the programmers from many concerns about evaluation order
  - uses definitions rather than the assignments
- Costs of lazy evaluation

Arithmetic Sequences

[1 . . 10] = [1, 2, 3, 4, 5, 6, 7, 8 ,9, 10]

[1, 3 . . 10] = [1, 3, 5, 7, 9]

[1, 4 . . 16] = [1, 4, 7, 10, 13, 16]

[1, 3 . .] = ?

Infinite data Structures

[1, 3 . .] = [1, 3, 5, 7, . .]
(Ad infinitum)

<u>More examples:</u>

```
squares = [n * n | n <- [0 . . ]]
fib     =
    1 : 1 : [ a + b | (a, b) <- zip fib (tail fib) ]
```

How are infinite structures useful?

take 5 squares => [0, 1, 4, 9, 16]

Be Careful!

```
member squares 16     (Okay. Returns True.)
member squares 15     (Doh! Infinite loop!)
member_smart(x : xs) n
   |  x < n                  = member_smart x n
   |  x == n                 = True
   |  otherwise              = False
```

Functions as First-class Values

```
map              (a –> b) –> [a] –> [b]
map f [ ]        = [ ]
map f (x:xs)     = f x : map f xs
```

map (add 1) [1, 2, 3] => [2, 3, 4]

Conclusion

- When to use C/C++
  - When speed is critical
  - When memory is at a premium

Conclusion (cont.)

- Advantages of Haskell
  - More concise
    (Quicksort, Binary tree, etc.)
  - Closely models algorithms
    (Infinite data structures)
  - Built-in garbage collection
  - Fewer programming errors
    (No side-effects, aliasing, or pointers)

Citations

- A Gentle Introduction to Haskell, Version 1.4
  Fasel, Hudak, and Peterson, 1997.
  http://www.haskell.org/tutorial/
- The Haskell 98 Report.
  Jones, Simon and Hughes,John, eds. 1999
  http://haskell.systemsz.cs.yale.edu/onlinereport/
- Concepts of Programming Languages, 4th ed.
  Sebesta, Robert. Reading, Mass: Addison Wesley, Longman, 1999.
- A Short Introduction to Haskell and its Advantages.
  Jones, Simon, 1998
  http://www.haskell.org/aboutHaskell.html


Questions ?

Unix Pipes

```
grep printf Quicksort.c | wc
```

```
grep printf Quicksort.c | head 5
```

[back]

Short-circuit Evaluation

(13 * A) * (B / 13 - 1)

(A >= 0) **and** (B < 10)

```
index := 1;
while (index <= listLen) and (list[index] <> key) do
        index := index + 1
```

[back]

Examples of Lazy Evaluation

a = a (non-terminating expression)

const1 x = 1 (what is the value of “const1 a” ?)

“definition” vs. “assignment”

v = 1/0

[back]

Quicksort in C

```
quicksort( a, lo, hi ) int a[], hi, lo;
{
  int h, l, p, t;
  if (lo < hi) {
    l = lo; h = hi; p = a[hi];

    do {
      while (l < h) & (a[i] <= p) do
          l = l+1;
      while (h > l) & (a[h] >= p) do
          h = h-1;
      if (l < h) {
          t = a[l]; a[l] = a[h]; a[h] = t;
    } while (l < h);

    t = a[l]; a[l] = a[hi]; a[hi] = t;

    quicksort( a, lo, l-1 );
    quicksort( a, l+1, hi );
  }
}
```

Quicksort in Haskell

```
qsort [ ]       = [ ]
qsort (x:xs)    = qsort [y | y <- xs, y < x]
                  ++ [x]
                  ++ qsort [y | y <- xs, y>=x]
```

Recursive Types

```
data Tree a         =
   Leaf  a  |  Branch  (Tree a)  (Tree a)
```

[back]