Introduction to Haskell

John Mitchell
(earlier slides by Kathleen Fisher)

Announcements

• Homework 1
  – Handed out today (on web this evening), due next Wed
• Course Graders
  – Send email to cs242@cs.stanford.edu
• Some problems with CourseGraders discussion
  – Apparently some permission issues that prevent some
    posts, and/or prevent some menu display
  – Significant toward working these out
• Reading on web site:
  – “Concepts in Programming Languages” Revised Chapter 5
  – “Real World Haskell”, Chapter 0
    (http://book.realworldhaskell.org)

Language Evolution

Lisp
     Algol 60
          Algol 68
              Pascal
                  C
                  Smalltalk
              ML
              Modula
              C++
              Java

Many others: Algol 58, Algol W, Scheme, EL1, Mesa (PARC), Modula-2, Oberon, Modula-3, Fortran, Ada, Perl, Python, Ruby, C#, Javascript, F#…

C Programming Language

Dennis Ritchie, ACM Turing Award for Unix

• Statically typed, general purpose systems programming language
• Computational model reflects underlying machine
• Relationship between arrays and pointers
  – An array is treated as a pointer to first element
  – E1[E2] is equivalent to ptr dereference: *(E1)+(E2))
  – Pointer arithmetic is not common in other languages
• Not statically type safe
  – If variable has type float, no guarantee value is floating pt
• Ritchie quote
  – “C is quirky, flawed, and a tremendous success”

ML programming language

• Statically typed, general-purpose programming language
  – “Meta-Language” of the LCF theorem proving system
• Type safe, with formal semantics
• Compiled language, but intended for interactive use
• Combination of Lisp and Algol-like features
  – Expression-oriented
  – Higher-order functions
  – Garbage collection
  – Abstract data types
  – Module system
  – Exceptions
• Used in printed textbook as example language

Haskell

• Haskell programming language is
  – Similar to ML: general-purpose, strongly typed, higher-order
    functional, supports type inference, interactive and compiled use
  – Different from ML: lazy evaluation, purely functional core, rapidly
    evolving type system
• Designed by committee in 80’s and 90’s to unify research
efforts in lazy languages
  – Haskell 1.0 in 1990, Haskell ’98, Haskell ongoing
  – “A History of Haskell: Being Lazy with Class” HOPL 3

Robin Milner, ACM Turing Award for ML, LCF Theorem Prover, …
Haskell B Curry

• Combinatory logic
  – Influenced by Russell and Whitehead
  – Developed combinators to represent substitution
  – Alternate form of lambda calculus that has been used in implementation structures
• Type inference
  – Devised by Curry and Feys
  – Extended by Hindley, Milner

Although “Currying” and “Curried functions” are named after Curry, the idea was invented by Schoenfinkel earlier.

Why Study Haskell?

• Good vehicle for studying language concepts
• Types and type checking
  – General issues in static and dynamic typing
  – Type inference
  – Parametric polymorphism
  – Ad hoc polymorphism (aka, overloading)
• Control
  – Lazy vs. eager evaluation
  – Tail recursion and continuations
  – Precise management of effects

Why Study Haskell?

• Functional programming will make you think differently about programming.
  – Mainstream languages are all about state
  – Functional programming is all about values
• Haskell is “cutting edge”
  – A lot of current research is done using Haskell
  – Rise of multi-core, parallel programming likely to make minimizing state much more important
• New ideas can help make you a better programmer, in any language

Most Research Languages

Successful Research Languages
C++, Java, Perl, Ruby

Committee languages

Practitioners

Geeks

Threshold of immortality

The complete absence of death

1yr 5yr 10yr 15yr

Practitioners

Geeks

The slow death

1yr 5yr 10yr 15yr

Haskell

The second life?


Practitioners

Geeks

Function Types in Haskell

In Haskell, $f : A \to B$ means for every $x \in A$,

$$f(x) = \begin{cases} 
\text{some element } y = f(x) \in B \\
\text{run forever}
\end{cases}$$

In words, “if $f(x)$ terminates, then $f(x) \in B$.”

Higher-Order Functions

- Functions that take other functions as arguments or return as a result are higher-order functions.
- Common Examples:
  - Map: applies argument function to each element in a collection.
  - Reduce: takes a collection, an initial value, and a function, and combines the elements in the collection according to the function.

```
list = [1,2,3]
g = foldl (\accumulator i -> i + accumulator) 0 \ list
```
- Google uses Map/Reduce to parallelize and distribute massive data processing tasks.

(Dean & Ghemawat, OSDI 2004)
Basic Overview of Haskell

- Interactive Interpreter (ghci): read-eval-print
  - ghci infers type before compiling or executing
  - Type system does not allow casts or other loopholes!
- Examples

```haskell
Prelude> (5+3)-2
6
it :: Integer
Prelude> if 5>3 then "Harry" else "Hermione"
"Harry"
it :: [Char] -- String is equivalent to [Char]
Prelude> 5==4
False
it :: Bool
```

Overview by Type

- Booleans
  ```haskell
  True, False :: Bool
  if ... then ... else ...
  -- types must match
  ```
- Integers
  ```haskell
  0, 1, 2, ... :: Integer
  (+, *, ...) :: Integer -> Integer -> Integer
  ```
- Strings
  ```haskell
  "Ron Weasley"
  ```
- Floats
  ```haskell
  1.0, 2, 3.14159, ... -- type classes to disambiguate
  ```

Simple Compound Types

- Tuples
  ```haskell
  (4, 5, "Griffendor") :: (Integer, Integer, String)
  ```
- Lists
  ```haskell
  [] :: [a] -- polymorphic type
  1 : [2, 3, 4] :: [Integer] -- infix cons notation
  ```
- Records
  ```haskell
  data Person = Person {firstName :: String,
    lastName :: String}
  hg = Person { firstName = "Hermione",
    lastName = "Granger"}
  ```

Patterns and Declarations

- Patterns can be used in place of variables
  ```haskell
  <pat> ::= <var> | <tuple> | <cons> | <record> ...
  ```
- Value declarations
  -- General form: <pat> = <exp>
  -- Examples
  ```haskell
  myTuple = ("Flitwick", "Snape")
  (x,y) = myTuple
  myList = [1, 2, 3, 4]
  z:zs = myList
  let (x,y) = (2, "Snape") in x * 4
  ```

Functions and Pattern Matching

- Anonymous function
  ```haskell
  \x -> x+1
  -- like Lisp lambda, function (\x) in JS
  ```
- Function declaration form
  ```haskell
  <name> <pat1>  = <exp1>
  <name> <pat2>  = <exp2> ...
  <name> <patn>  = <expn> ...
  ```
- Examples
  ```haskell
  f (x,y) = x+y
  length [] = 0
  length (xs) = 1 + length(xs)
  ```

Map Function on Lists

- Apply function to every element of list
  ```haskell
  map f [] = []
  map f (x:xs) = f x : map f xs
  ```
  ```haskell
  map (\x -> x+1) [1,2,3]       [2,3,4]
  ```
- Compare to Lisp
  ```haskell
  (define map
    (lambda (f xs)
      (if (eq? xs ())  ()
        (cons (f (car xs))  (map f (cdr xs)))))
  ```

Patterns and Pattern Matching

- Tuples
  ```haskell
  (4, 5, "Griffendor") :: (Integer, Integer, String)
  ```
- Lists
  ```haskell
  [] :: [a] -- polymorphic type
  1 : [2, 3, 4] :: [Integer] -- infix cons notation
  ```
- Records
  ```haskell
  data Person = Person {firstName :: String,
    lastName :: String}
  hg = Person { firstName = "Hermione",
    lastName = "Granger"}
  ```

Patterns and Declarations

- Patterns can be used in place of variables
  ```haskell
  <pat> ::= <var> | <tuple> | <cons> | <record> ...
  ```
- Value declarations
  -- General form: <pat> = <exp>
  -- Examples
  ```haskell
  myTuple = ("Flitwick", "Snape")
  (x,y) = myTuple
  myList = [1, 2, 3, 4]
  z:zs = myList
  let (x,y) = (2, "Snape") in x * 4
  ```

Functions and Pattern Matching

- Anonymous function
  ```haskell
  \x -> x+1
  -- like Lisp lambda, function (\x) in JS
  ```
- Function declaration form
  ```haskell
  <name> <pat1>  = <exp1>
  <name> <pat2>  = <exp2> ...
  <name> <patn>  = <expn> ...
  ```
- Examples
  ```haskell
  f (x,y) = x+y
  length [] = 0
  length (xs) = 1 + length(xs)
  ```

Map Function on Lists

- Apply function to every element of list
  ```haskell
  map f [] = []
  map f (x:xs) = f x : map f xs
  ```
  ```haskell
  map (\x -> x+1) [1,2,3]       [2,3,4]
  ```
- Compare to Lisp
  ```haskell
  (define map
    (lambda (f xs)
      (if (eq? xs ())  ()
        (cons (f (car xs))  (map f (cdr xs)))))
  ```
More Functions on Lists

• Append lists

\[
\text{append } \{\}, \text{ys} = \text{ys} \\
\text{append } (\text{x:xs}, \text{ys}) = \text{x : append } \text{xs} \text{, ys}
\]

• Reverse a list

\[
\text{reverse } \{\} = \{\} \\
\text{reverse } (\text{x:xs}) = \text{reverse } \text{xs} ++ [\text{x}]
\]

Questions

– How efficient is reverse?
– Can it be done with only one pass through list?

More Efficient Reverse

\[
\text{reverse } \text{xs} = \\
\text{let } \text{rev } (\{\}, \text{accum} ) = \text{accum} \\
\text{rev } (\text{y:ys}, \text{accum} ) = \text{rev } \text{ys}, \text{y:accum} \\
\text{in } \text{rev } (\text{xs}, \{\})
\]

List Comprehensions

• Notation for constructing new lists from old:

\[
\text{myData} = \{1,2,3,4,5,6,7\} \\
\text{twiceData} = \{2 * \text{x} | \text{x} \in \text{myData}\} \\
\text{twiceEvenData} = \{2 * \text{x} | \text{x} \in \text{myData}, \text{x} \text{ mod } 2 == 0\}
\]

• Similar to “set comprehension”

\[
\{ \text{x} | \text{x} \in \text{Odd} \land \text{x} > 6\}
\]

Datatype Declarations

• Examples

\[
\text{data Color} = \text{Red} \mid \text{Yellow} \mid \text{Blue} \\
\text{data Atom} = \text{Atom String} \mid \text{Number Int} \\
\text{data List} = \text{Nil} \mid \text{Cons (Atom, List)}
\]

• General form

\[
\text{data } \text{name} = <\text{clause}> | … | <\text{clause}> \\
<\text{clause}> ::= <\text{constructor}> \mid <\text{constructor}> <\text{type}>
\]

– Type name and constructors must be Capitalized.

Datatypes and Pattern Matching

• Recursively defined data structure

\[
\text{data Tree} = \text{Leaf Int} \mid \text{Node (Int, Tree, Tree)}
\]

\[
\text{Node}(4, \text{Node}(3, \text{Leaf } 1, \text{Leaf } 2), \text{Node}(5, \text{Leaf } 6, \text{Leaf } 7))
\]

• Recursive function

\[
\text{sum } (\text{Leaf } n) = n \\
\text{sum } (\text{Node}(\text{n:tl, t2})) = n + \text{sum}(\text{tl}) + \text{sum}(\text{t2})
\]

Example: Evaluating Expressions

• Define datatype of expressions

\[
\text{data Exp} = \text{Var Int} \mid \text{Const Int} \mid \text{Plus (Exp, Exp)}
\]

\[
\text{write } (x+3)+ y \text{ as Plus(Plus(Var 1, Const 3), Var 2)}
\]

• Evaluation function

\[
\text{ev}(\text{Var } n) = \text{Var } n \\
\text{ev}(\text{Const } n) = \text{Const } n \\
\text{ev}(\text{Plus}(\text{e1, e2})) = \\
\]

• Examples

\[
\text{ev}(\text{Plus(\text{Const } 3, \text{Const } 2))} \rightarrow \text{Const } 5 \\
\text{ev}(\text{Plus(Var 1, Plus(\text{Const } 2, \text{Const } 3))}) \rightarrow \text{Plus(Var } 1, \text{Const } 5)\]
Case Expression

- Datatype

```haskell
data Exp = Var Int | Const Int | Plus (Exp, Exp)
```

- Case expression

```haskell
case e of
    Var n -> ...
    Const n -> ...
    Plus(e1,e2) -> ...
```

Indentation matters in case statements in Haskell.

Evaluation by Cases

```haskell
data Exp = Var Int | Const Int | Plus (Exp, Exp)

eval ( Var n) = Var n
eval ( Const n ) = Const n
eval ( Plus ( e1,e2 ) ) =
    case eval e1 of
        Var n -> Plus( Var n, eval e2)
        Const n -> case eval e2 of
            Var m -> Plus( Const n, Var m)
            Const m -> Const (n+m)
            Plus(e3,e4) -> Plus ( Const n, Plus ( e3, e4 ))
        Plus(e3, e4) -> Plus( Plus ( e3, e4 ), eval e2)
```

Laziness

- Haskell is a lazy language
- Functions and data constructors don’t evaluate their arguments until they need them
- Programmers can write control-flow operators that have to be built-in in eager languages

```haskell
cond :: Bool -> a -> a -> a
cond True  t e = t
cond False t e = e
```

Using Laziness

```haskell
isSubString :: String -> String -> Bool
isSubString' s = or [ s `isPrefixOf` t | t <- suffixes s ]

suffixes :: String -> [String]
-- All suffixes of s
suffixes [] = [[]]
suffixes(x:xs) = (x:xs) : suffixes xs

or :: [Bool] -> Bool
-- (or bs) returns True if any of the bs is True
or [] = False
or (b:bs) = b || or bs
```

Core Haskell

- Basic Types
  - Unit
  - Booleans
  - Integers
  - Strings
  - Reals
  - Tuples
  - Lists
  - Records
- Patterns
- Declarations
- Functions
- Polymorphism
- Type declarations
- Type Classes
- Monads
- Exceptions
Running Haskell

• Look for instructions on web site
  – Available on Stanford pod cluster (soon?)
• Or, download: [http://haskell.org/ghc](http://haskell.org/ghc)
• Interactive:
  – ghci intro.hs
• Compiled:
  – ghc -make HaskellIntro.hs

Testing

• It’s good to write tests as you write code
• E.g. `reverse` undoes itself, etc.

```
reverse xs = 
  let rev ( [], z ) = z
      rev ( y:ys, z ) = rev( ys, y:z )
  in rev( xs, [] )

-- Write properties in Haskell

prop_RevRev :: [Int] -> Bool
prop_RevRev xs = reverse(reverse xs) == xs
```

QuickCheck

• Generate random input based on type
  – Generators for values of type `a` has type `Gen a`
  – Have generators for many types
• Conditional properties
  – Have form `<condition> ==> <property>`
  – Example:
    ordered xs = and (zipWith (<=) xs (drop 1 xs))
    insert x xs = takeWhile (<x) xs++[x]++dropWhile (<x) xs
    prop_Insert x xs =
      ordered xs ==> ordered (insert x xs)
    where types = xs::Int

```
prop_RevId xs = reverse xs == xs
```

```
quickCheck prop_RevId
Falsifiable, after 1 tests:
[ 3,15]
```

Things to Notice

No side effects. At all.

```
reverse :: [w] -> [w]
```

• A call to `reverse` returns a new list; the old one is unaffected.

```
prop_RevRev 1 = reverse(reverse 1) == 1
```

• A variable ‘l’ stands for an immutable value, not for a location whose value can change.

• Laziness forces this purity.

Test Interactively

```
bash$ ghci intro.hs
Prelude> :m +Test.QuickCheck
Prelude Test.QuickCheck> quickCheck prop_RevRev
+++ OK, passed 100 tests
```

```
quickCheck :: Testable prop => prop -> IO ()
```

```
Prelude Test.QuickCheck> :t quickCheck
quickCheck :: Testable prop => prop -> IO ()
```

QuickCheck

```
-- Test case generation continues until
-- 100 cases which do satisfy the condition have been found, or
-- until an overall limit on the number of test cases is reached (to
-- avoid looping if the condition never holds).
```

```
see : [http://www.cse.chalmers.se/~rjmh/QuickCheck/manual.html](http://www.cse.chalmers.se/~rjmh/QuickCheck/manual.html)
```
Things to Notice

- Purity makes the interface explicit.
  \[ \text{reverse} : [w] \to [w] \quad \text{-- Haskell} \]
- Takes a list, and returns a list; that’s all.
  \[ \text{void reverse} ( \text{list } l ) \qquad \text{C} \]
- Takes a list; may modify it; may modify other persistent state; may do I/O.

Things to Notice

- Pure functions are easy to test.
  \[ \text{prop_RevRev} l = \text{reverse}(\text{reverse } l) == l \]
- In an imperative or OO language, you have to
  – set up the state of the object and the external state it reads or writes
  – make the call
  – inspect the state of the object and the external state
  – perhaps copy part of the object or global state, so that you can use it in the post condition

Things to Notice

Types are everywhere.

- Usual static-typing panegyric omitted...
- In Haskell, \textit{types express high-level design}, in the same way that UML diagrams do, with the advantage that the type signatures are machine-checked.
- Types are (almost always) optional: type inference fills them in if you leave them out.

More Info: haskell.org

- The Haskell wikibook
  – \url{http://en.wikibooks.org/wiki/Haskell}
- All the Haskell bloggers, sorted by topic
  – \url{http://haskell.org/haskellwiki/Blog_articles}
- Collected research papers about Haskell
  – \url{http://haskell.org/haskellwiki/Research_papers}
- Wiki articles, by category
  – \url{http://haskell.org/haskellwiki/Category:Haskell}
- Books and tutorials
  – \url{http://haskell.org/haskellwiki/Books_and_tutorials}