Introduction to Haskell

Shachar Itzhaky & Mooly Sagiv
(original slides by Kathleen Fisher & John Mitchell)
Part I: Lambda Calculus
Computation Models

- Turing Machines
- Wang Machines
- Counter Programs
- Lambda Calculus

\[ \lambda f. (\lambda x. (f (x x))) (\lambda x. (f (x x))) \]
Untyped Lambda Calculus

Chapter 5
Benjamin Pierce
Types and Programming Languages
Basics

• Repetitive expressions can be compactly represented using functional abstraction

• Example:

  \[(5 \times 4 \times 3 \times 2 \times 1) + (7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1) =\]

  \[\text{factorial}(5) + \text{factorial}(7)\]

  \[\text{factorial}(n) = \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{factorial}(n-1)\]

  \[\text{factorial} = \lambda n. \text{if } n = 0 \text{ then } 0 \text{ else } n \text{ \text{factorial}(n-1)}\]
Untyped Lambda Calculus

\[ t ::= \]

\[ \text{terms} \]

\[ x \]

\[ \text{variable} \]

\[ \lambda x . \ t \]

\[ \text{abstraction} \]

\[ t \ t \]

\[ \text{application} \]

Terms can be represented as abstract syntax trees

Syntactic Conventions

• Applications associates to left
  \[ e_1 \ e_2 \ e_3 \equiv (e_1 \ e_2) \ e_3 \]

• The body of abstraction extends as far as possible
  • \[ \lambda x . \ (\lambda y . \ x \ y) \ x \equiv \lambda x . \ (\lambda y . \ (x \ y) \ x) \]
Free vs. Bound Variables

• An occurrence of x is free in a term t if it is not in the body on an abstraction $\lambda x. t$
  – otherwise it is bound
  – $\lambda x$ is a binder

• Examples
  – $\lambda z. \lambda x. \lambda y. x (y z)$
  – $(\lambda x. x) x$

• Terms w/o free variables are combiners
  – Identify function: id = $\lambda x. x$
Operational Semantics

$$(\lambda x. t_{12}) t_2 \rightarrow [x \mapsto t_2] t_{12} \quad (\beta\text{-reduction})$$

$FV: t \rightarrow P(\text{Var})$ is the set free variables of $t$

$FV(x) = \{x\}$

$FV(\lambda x. t) = FV(t) - \{x\}$

$FV(t_1 t_2) = FV(t_1) \cup FV(t_2)$

$[x \mapsto s] x = s$

$[x \mapsto s] y = y$ \quad if $y \neq x$

$[x \mapsto s] (\lambda y. t_1) = \lambda y. [x \mapsto s] t_1$ \quad if $y \neq x$ and $y \notin FV(s)$

$[x \mapsto s] (t_1 t_2) = ([x \mapsto s] t_1) ( [x \mapsto s] t_2)$
Operational Semantics

$$(\lambda \mathbf{x}. t_{12}) \ t_2 \rightarrow [\mathbf{x} \mapsto t_2] \ t_{12} \quad (\beta\text{-reduction})$$

redex

$$\quad (\lambda \mathbf{x}. \mathbf{x}) \ \mathbf{y} \rightarrow \ \mathbf{y}$$

$$\quad (\lambda \mathbf{x}. \mathbf{x} (\lambda \mathbf{x}. \mathbf{x}) ) \ (\mathbf{u} \ \mathbf{r}) \rightarrow \ \mathbf{u} \ \mathbf{r} \ (\lambda \mathbf{x}.\mathbf{x})$$

$$\quad (\lambda \mathbf{x} (\lambda \mathbf{w}. \mathbf{x} \ \mathbf{w})) \ (\mathbf{y} \ \mathbf{z}) \rightarrow \ \lambda \mathbf{w}. \ \mathbf{y} \ \mathbf{z} \ \mathbf{w}$$
Lambda Calculus vs. JavaScript

\((\lambda \ x. \ x) \ y\) \quad \text{(function } (x) \ {\text{return}} \ x;{\}) \ y\)
Evaluation Orders

$$(\lambda x. t_{12}) t_2 \rightarrow [x \mapsto t_2] t_{12} \quad (\beta\text{-reduction})$$

$$(\lambda x. x) ((\lambda x. x) (\lambda z. (\lambda x. x) z)) \equiv id (id (\lambda z. id z))$$

Normal order

$$(\lambda z. id z) \rightarrow id (\lambda z. id z)$$

call-by-name

$$(\lambda z. id z) \rightarrow \lambda z. id z$$

call-by-value

$$(\lambda z. id z) \rightarrow \lambda z. z$$
Call-by-value Operational Semantics

\[ (\lambda x. t_{12}) \ v_2 \rightarrow [x \mapsto v_2] \ t_{12} \quad \text{(E-AppAbs)} \]

\[ t_1 \rightarrow t'_1 \]

\[ t_1 \ t_2 \rightarrow t'_1 \ t_2 \quad \text{(E-APPL1)} \]

\[ t_2 \rightarrow t'_2 \]

\[ v_1 \ t_2 \rightarrow v_1 \ t'_2 \quad \text{(E-APPL2)} \]
Programming in the Lambda Calculus

Multiple arguments

\[ f = \lambda(x, y). s \]

-> Currying

\[ f = \lambda x. \lambda y. s \]

\[
\begin{align*}
  f \ v \ w &= \\
  (f \ v) \ w &= \\
  (\lambda x. \lambda y. s \ v) \ w &\rightarrow \\
  \lambda y.[x \mapsto v] s \ w &\rightarrow \\
  [x \mapsto v][y \mapsto w] s
\end{align*}
\]
Booleans

• \text{tru} = \lambda t. \lambda f. t
• \text{fls} = \lambda t. \lambda f. f
• \text{test} = \lambda l. \lambda m. \lambda n. l m n
• \text{and} = \lambda b. \lambda c. b c \text{fls}
Programming in the Lambda Calculus
Pairs

• pair = \(\lambda f. \lambda s. \lambda b. b\ f\ s\)

• fst = \(\lambda p. p\ \text{tru}\)

• snd = \(\lambda p. p\ \text{fls}\)
Programming in the Lambda Calculus
Numerals

• $c_0 = \lambda s. \lambda z. z$
• $c_1 = \lambda s. \lambda z. s\ z$
• $c_2 = \lambda s. \lambda z. s\ (s\ z)$
• $c_3 = \lambda s. \lambda z. s\ (s\ (s\ z))$
• $\text{succ} = \lambda n. \lambda s. \lambda z. s\ (n\ s\ z)$
• $\text{plus} = \lambda m. \lambda n. \lambda s. \lambda z. m\ s\ (n\ s\ z)$
• $\text{times} = \lambda m. \lambda n. m\ (\text{plus}\ n)\ c_0$

> Turing Complete
Divergence in Lambda Calculus

• $\text{omega}= (\lambda x. x \ x) \ (\lambda x. x \ x)$

• $\text{fix} = \lambda f. \ (\lambda x. f \ (\lambda y. x \ x \ y)) \ (\lambda x. f \ (\lambda y. x \ x \ y))$
Extending the Lambda Calculus

• Primitive values
• Exceptions
• References
Summary: Lambda Calculus

• Powerful
• Useful to illustrate ideas
• But can be counterintuitive
• Usually extended with useful syntactic sugars
• Other calculi exist
  – pi-calculus
  – object calculus
  – mobile ambients
  – ...
Part II: Some Chaps
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 earlier by Moses Schönfinkel
Haskell

• 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
Haskell

• Haskell programming language is -

  general-purpose
  strongly typed
  higher-order
  supports type inference
  interactive and compiled use

  based on lazy evaluation
  purely functional (core)
  rapidly evolving type system

Like most functional PLs

Unlike most functional PLs

Paul Hudak

John Hughes

Simon Peyton Jones

Phil Wadler
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
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
Function Types in Haskell

In Haskell, $f :: A \rightarrow B$ 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 $f(x) \in B$.”

In ML, functions with type $A \rightarrow B$ can throw an exception or have other effects, but not in Haskell.
Basic Overview of Haskell

• Anonymous functions

\x \rightarrow \ x + 1 \quad --\text{like Lisp lambda, function (\ldots) in JS}

• Named functions

\begin{verbatim}
inc \ x = x + 1
f \ (x, y) = x + y
length \ [] = 0
length \ (x:s) = 1 + length(s)
\end{verbatim}
Overview by Type

- **Booleans**
  ```
  True, False :: Bool
  if ... then ... else ... --types must match
  ```

- **Integers**
  ```
  0, 1, 2, ... :: Integer
  +, *, ... :: Integer -> Integer -> Integer
  ```

- **Strings**
  ```
  "Ron Weasley"
  ```

- **Floats**
  ```
  1.0, 2, 3.14159, ... --type classes to disambiguate
  ```
Simple Compound Types

- **Tuples**

  \[(4, 5, \text{"Griffindor"}) :: (\text{Integer}, \text{Integer}, \text{String})\]

- **Lists**

  \[[] :: [a] \quad -- \text{polymorphic type}\]

  \[1 : [2, 3, 4] :: [\text{Integer}] \quad -- \text{infix cons notation}\]

- **Records**

  ```haskell
  data Person = Person {firstName :: String,
                         lastName  :: String}
  
  hg = Person { firstName = "Hermione",
                lastName  = "Granger"}
  ```
List Comprehensions

• Notation for constructing new lists from existing ones:

```
myData = [1,2,3,4,5,6,7]
twiceData = [2 * x | x <- myData]
          -- [2,4,6,8,10,12,14]
twiceEvenData = [2 * x | x <- myData, x `mod` 2 == 0]
                 -- [4,8,12]
```

• In the spirit of mathematical “set comprehension”
  
  \[ \{ x | x \in \text{Odd} \land x > 6 \} \]
Interactive Interpreter (ghci)

• REPL: read-eval-print loop
  – ghci infers type before compiling or executing
  – Type system does not allow casts or other loopholes!
• Examples

```ghci
% ghci
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
```
Higher Order Functions

• Functions are first class objects
  – Passed as parameters
  – Returned as results

• Practical examples
  – Google map/reduce
Example: Differentiate

- The differential operator
  \[ f'(x) = \lim_{h \to 0} \frac{(f(x+h)-f(x))}{h} \]

- In Haskell:

```
diff f = f_prime
  where
    f_prime x = (f (x + h) - f x) / h
    h = 0.0001
```

- \( \text{diff} :: \text{float} \to \text{float} \to \text{float} \)
- \((\text{diff} \ \text{square}) \ 0 = 0.0001\)
- \((\text{diff} \ \text{square}) \ 0.0001 = 0.0003\)
- \((\text{diff} \ (\text{diff} \ \text{square})) \ 0 = 2\)
Pattern Matching

- Patterns can be used in place of variable names
  
  $$\text{<pat> ::= <var> | <tuple> | <cons> | <record> ...}$$

- Value declarations
  
  - General form: $$\text{<pat> = <exp>}$$
  - In global declarations
    
    ```
    myTuple = ("Flitwick", "Snape")
    (x,y) = myTuple
    myList = [1, 2, 3, 4]
    z:zs = myList
    ```

  - In local declarations
    
    ```
    let (x,y) = (2, "Snape") in x * 4
    ```
Pattern Matching

• Explicit case expression

```haskell
myTuple = ("Flitwick", "Snape")
v = case myTuple of
    (x, "Snape") -> x ++ "?"
    ("Flitwick", y) -> y ++ "!
    _ -> "?!"
```
Map Function on Lists

• Apply function to every element of list

\[
\begin{align*}
\text{map } f \; [] &= [] \\
\text{map } f \; (x:xs) &= f \; x \; : \; \text{map } f \; xs
\end{align*}
\]

\[
\text{map } \left( \lambda x \to x+1 \right) \; [1,2,3] \quad \rightarrow \quad [2,3,4]
\]

• Compare to Lisp

```
(define map
  (lambda (f xs)
    (if (eq? xs ()) ()
      (cons (f (car xs)) (map f (cdr xs))))))
```
More Functions on Lists

• Append lists

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

• Reverse a list

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

• Questions
  – How efficient is reverse?
  – Can it be done with only one pass through list?
More Efficient Reverse

reverse xs =
    let rev ( [], accum ) = accum
        rev ( y:ys, accum ) = rev ( ys, y:accum )
    in rev ( xs, [] )
Datatype Declarations

• Examples

```haskell
data Color = Red | Yellow | Blue

  elements are Red, Yellow, Blue

data Atom = Atom String | Number Int

  elements are Atom “A”, Atom “B”, ..., Number 0, ...

data AtomList = Nil | Cons Atom AtomList

  elements are Nil, Cons (Atom “A”) Nil, ...
  Cons (Number 2) (Cons (Atom “Bill”)) Nil, ...
```

• General form

```haskell
data <name> = <clause> | ... | <clause>
<clause> ::= <constructor> | <constructor> <type>
```

  – Type name and constructors must be Capitalized
Datatypes and Pattern Matching

- Recursively defined data structure

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

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

- Recursive function

\[
\text{sum (Leaf } n) = n \\
\text{sum (Node}(n, t1, t2)) = n + \text{sum}(t1) + \text{sum}(t2)
\]
Example: Evaluating Expressions

- Define datatype of expressions

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

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

- Evaluation function

```haskell
ev(Var n) = Var n
ev(Const n) = Const n
ev(Plus(e1, e2)) = ... 
```

- Examples

```haskell
ev(Plus(Const 3, Const 2)) ➞ Const 5
```

```haskell
ev(Plus(Var 1, Plus(Const 2, Const 3))) ➞ Plus(Var 1, Const 5)
```
Use the 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
Offside rule

- Layout characters matter to parsing

```plaintext
x = 6 / 5
y = let a = x
    b = 2 * x
    in a * b
```

- Everything below and right of = in equations defines a new scope

- Applied recursively

```plaintext
fac n = if (n == 0) then 1 else prod n (n-1)
    where
    prod acc n = if (n == 0) then acc
                  else prod (acc * n) (n-1)
```

- Lexical analyzer maintains a stack
Evaluation by Cases

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

ev ( Var n) = Var n
ev ( Const n ) = Const n
ev ( Plus ( e1,e2 ) ) =

    case ev e1 of
          Var n  -> Plus(Var n, ev e2)
          Const n -> case ev 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 ), ev e2)
Polymorphic Typing

• Polymorphic expression has many types
• Benefits:
  – Code reuse
  – Guarantee consistency
• The compiler infers that in
  length [] = 0
  length (x: xs) = 1 + length xs
  – length has the type [a] -> int
    length :: [a] -> int
• Example expressions
  – length [1, 2, 3] + length [“red”, “yellow”, “green”]
  – length [1, 2, “green” ] // invalid list
• The user can optionally declare types
• Every expression has the most general type
• “boxed” implementations
Laziness

- Haskell is a **lazy** language
- Functions and data constructors don’t evaluate their arguments until they need them

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

- Programmers can write control-flow operators that have to be built-in in eager languages

```haskell
(||) :: Bool -> Bool -> Bool
True  || x = True
False || x = x
```
Using Laziness

isSubString :: String -> String -> Bool
x `isSubString` s = or [ x `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
A Lazy Paradigm

• Generate all solutions (an enormous tree)
• Walk the tree to find the solution you want

```
nextMove :: Board -> Move
nextMove b = selectMove allMoves
    where
        allMoves = allMovesFrom b
```

A gigantic (perhaps infinite) tree of possible moves
Benefits of Lazy Evaluation

- Define streams:
  
  ```haskell
  main = take 100 [1 .. ]
  ```

- Lower asymptotic complexity
- Language extensibility
  - Domain specific languages
- But some costs

```
deriv f x = lim [(f (x + h) - f x) / h | h <- [1/2^n | n <- [1..]]]
  where lim (a:b:lst) = if abs(a/b-1) < eps then b
                                 else lim (b: lst)
  
  eps = 1.0 e-6
```
Core Haskell

- Basic Types
  - Unit
  - Booleans
  - Integers
  - Strings
  - Reals
  - Tuples
  - Lists
  - Records

- Patterns

- Declarations

- Functions

- Polymorphism

- Type declarations
  - Type Classes
  - Monads
  - Exceptions
## Functional Programming Languages

<table>
<thead>
<tr>
<th>PL</th>
<th>types</th>
<th>evaluation</th>
<th>Side-effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheme</td>
<td>Weakly typed</td>
<td>Eager</td>
<td>yes</td>
</tr>
<tr>
<td>ML OCAML F#</td>
<td>Polymorphic strongly typed</td>
<td>Eager</td>
<td>References</td>
</tr>
<tr>
<td>Haskell</td>
<td>Polymorphic strongly typed</td>
<td>Lazy</td>
<td>None</td>
</tr>
</tbody>
</table>
# Compiling Functional Programs

<table>
<thead>
<tr>
<th>Compiler Phase</th>
<th>Language Aspect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexical Analyzer</td>
<td>Offside rule</td>
</tr>
<tr>
<td>Parser</td>
<td>List notation</td>
</tr>
<tr>
<td></td>
<td>List comprehension</td>
</tr>
<tr>
<td></td>
<td>Pattern matching</td>
</tr>
<tr>
<td>Context Handling</td>
<td>Polymorphic type checking</td>
</tr>
<tr>
<td>Run-time system</td>
<td>Referential transparency</td>
</tr>
<tr>
<td></td>
<td>Higher order functions</td>
</tr>
<tr>
<td></td>
<td>Lazy evaluation</td>
</tr>
</tbody>
</table>
Structure of a functional compiler

High-level language

Polymorphic type inference

Optimizations

Functional core

Code generation

C code

Runtime system

De-sugaring:
1. Pattern matching
2. List to pairs
3. List comprehension
4. Lambda lifting
QuickCheck

• Generate random input based on type
  – Generators for values of type \( a \) have type \( \text{Gen} \ a \)
  – Have generators for many types

• Conditional properties
  – Have form \(<\text{condition}> \implies <\text{property}>\)
  – Example:

```haskell
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 = x :: Int
```
QuickCheck

• QuickCheck output
  – When property succeeds:
    quickCheck prop_RevRev OK, passed 100 tests.
  – When a property fails, QuickCheck displays a counter-example.
    prop_RevId xs = reverse xs == xs where types = xs::[Int]
    quickCheck prop_RevId
    Falsifiable, after 1 tests: [-3,15]

• Conditional testing
  – Discards test cases which do not satisfy the condition.
  – 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
Things to Notice

No side effects. At all

\[ \text{reverse:: [w] -> [w]} \]

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

\[ \text{prop_RevRev l = reverse(reverse l) == l} \]

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

- Laziness forces this purity
Things to Notice

• Purity makes the interface explicit.
  
  \texttt{reverse:: \{w\} \rightarrow \{w\} \quad -- \texttt{Haskell}}

• Takes a list, and returns a list; that’s all.
  
  \texttt{void reverse(\ list \ l \ ) \quad /* \ 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

prop_RevRev l = reverse(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.

reverse :: [w] -> [w]

• Usual static-typing panegyric omitted...
• In Haskell, 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
• All the Haskell bloggers, sorted by topic
  – http://haskell.org/haskellwiki/Blog_articles
• Collected research papers about Haskell
  – http://haskell.org/haskellwiki/Research_papers
• Wiki articles, by category
  – http://haskell.org/haskellwiki/Category:Haskell
• Books and tutorials
  – http://haskell.org/haskellwiki/Books_and_tutorials
• Compiling Haskell
  – Modern Compiler Design
Haskel in the Real World

• ABN AMRO bank measuring the counterparty risk on portfolios of financial derivatives
• Alcatel-Lucent used Haskell to prototype narrowband software radio systems, running in (soft) real-time
• Haskell is being used for backend data transformation and loading in Merril Lynch
• Barclays Capital's Quantitative Analytics group is using Haskell to develop an embedded domain-specific functional language (called FPF) which is used to specify exotic equity derivatives
Missing

- IO-Monads
Summary

• Functional programs provide concise coding
• Compiled code compares with C code
• Successfully used in some commercial applications
  – F#, ERLANG
• Ideas used in imperative programs
• Good conceptual tool
• Less popular than imperative programs
• Haskel is a well thought functional language