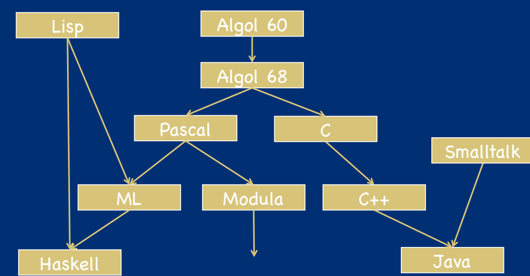
cs242

THE ALGOL FAMILY AND HASKELL

Kathleen Fisher

Reading: "Concepts in Programming Languages" Chapter 5 except 5.4.5
"Real World Haskell", Chapter 0 and Chapter 1
(http://book.realworldhaskell.org/)

Language Evolution



Many other languages: Algol 58, Algol W, Euclid, EL1, Mesa (PARC), Modula-2, Oberon, Modula-3 (DEC), …

Algol 60

- Basic Language of 1960
  - Simple imperative language + functions
  - Successful syntax, BNF -- used by many successors
    - statement oriented
    - begin … end blocks (like `C/Java { … }`)
    - if … then … else
  - Recursive functions and stack storage allocation
  - Fewer ad hoc restrictions than Fortran
    - General array references: `A[ x + B[3] * y ]`
  - Type discipline was improved by later languages
  - Very influential but not widely used in US
- Tony Hoare: "Here is a language so far ahead of its time that it was not only an improvement on its predecessors but also on nearly all of its successors."

Algol 60 Sample

```
real procedure average(A,n);
  real array A; integer n;
  begin
    real sum; sum := 0;
    for i = 1 step 1 until n
  do
    sum := sum + A[i];
    average := sum/n
  end;
```

No array bounds.

No ";" here.

Set procedure return value by assignment.

Algol Oddity

- Question:
  - Is `x := x` equivalent to doing nothing?
- Interesting answer in Algol:

```
integer procedure p;
begin
   ….
   p := p
   ….
end;
```

Assignment here is actually a recursive call!

Some trouble spots in Algol 60

- Holes in type discipline
  - Parameter types can be arrays, but
    - No array bounds
  - Parameter types can be procedures, but
    - No argument or return types for procedure parameters
- Some awkward control issues
  - goto out of block requires memory management
- Problems with parameter passing mechanisms
  - Pass-by-name "Copy rule" duplicates code, interacting badly with side effects
  - Pass-by-value expensive for arrays

Algol 60 Pass-by-name

- Substitute text of actual parameter
  - Unpredictable with side effects!
- Example

```
procedure inc2(i, j);
  integer i, j;
  begin
    i := i+1;
    j := j+1
  end;

inc2 (k, A[k]);
```

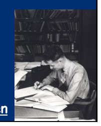
→

```
begin
  k := k+1;
  A[k] := A[k] +1
end;
```

Is this what you expected?

Algol 68

- Fixed some problems of Algol 60
  - Eliminated pass-by-name
- Considered difficult to understand
  - Idiosyncratic terminology
    - Types were called "modes"
    - Arrays were called "multiple values"
  - Used vW grammars instead of BNF
    - Context-sensitive grammar invented by van Wijngaarden
  - Elaborate type system
  - Complicated type conversions
- Not widely adopted



Adriaan van Wijngaarden

Algol 68 "Modes"

- Primitive modes
  - int
  - real
  - char
  - bool
  - string
  - compl (complex)
  - bits
  - bytes
  - sema (semaphore)
  - format (I/O)
  - file
- Compound modes
  - arrays
  - structures
  - procedures
  - sets
  - pointers

Rich, structured, and orthogonal type system is a major contribution of Algol 68.

Other Features of Algol 68

- Storage management
  - Local storage on stack
  - Heap storage, explicit alloc, and garbage collection
- Parameter passing
  - Pass-by-value
  - Use pointer types to obtain pass-by-reference
- Assignable procedure variables
  - Follow "orthogonality" principle rigorously

A Tutorial on Algol 68 by Andrew S. Tanenbaum

Pascal

- Designed by Niklaus Wirth (Turing Award)
- Revised the type system of Algol
  - Good data-structuring concepts
    - records, variants, subranges
  - More restrictive than Algol 60/68
    - Procedure parameters cannot have higher-order procedure parameters
- Popular teaching language
- Simple one-pass compiler

Niklaus Wirth

Limitations of Pascal

- Array bounds part of type

illegal

```
procedure p(a : array [1..10] of integer)
procedure p(n: integer, a : array [1..n] of integer)
```

- Attempt at orthogonal design backfires
  - Parameter must be given a type
  - Type cannot contain variables

How could this have happened? Emphasis on teaching?

- Not successful for "industrial-strength" projects
  - Kernighan: "Why Pascal is not my favorite language"
  - Left niche for C; niche has expanded!!

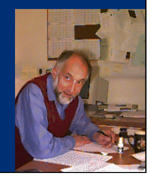
C Programming Language

Designed by Dennis Ritchie, Turing Award winner, for writing Unix

- Evolved from B, which was based on BCPL
  - B was an untyped language; C adds some checking
- 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
- Ritchie quote
  - "C is quirky, flawed, and a tremendous success."

ML

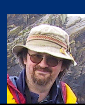
- Statically typed, general-purpose programming language
- Type safe!
- Compiled language, but intended for interactive use
- Combination of Lisp and Algol-like features
  - Expression-oriented
  - Higher-order functions
  - Garbage collection
  - Abstract data types
  - Module system
  - Exceptions
- Designed by Turing-Award winner Robin Milner for LCF Theorem Prover
- Used in textbook as example language



Haskell

- Haskell is a programming language that is
  - Similar to ML: general-purpose, strongly typed, higher-order, functional, supports type inference, supports 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

Paul Hudak



John Hughes

Simon Peyton Jones

Phil Wadler

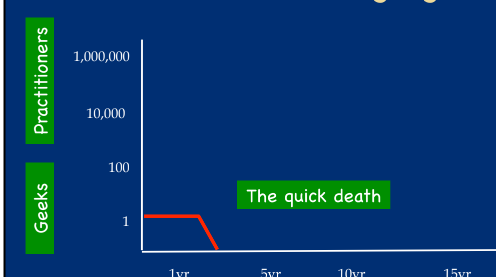
Why Study Haskell?

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

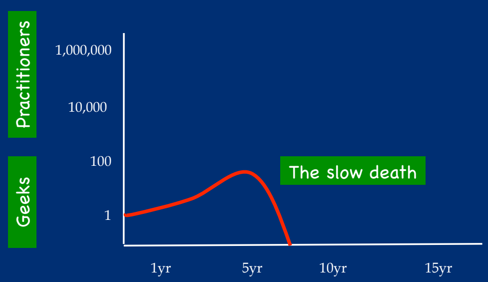
Why Study Haskell?

- Functional programming will make you think differently about programming.
  - Mainstream languages are all about state
  - Functional programming is all about values
- Ideas will make you a better programmer in whatever language you regularly use.
- Haskell is "cutting edge." A lot of current research is done in the context of Haskell.

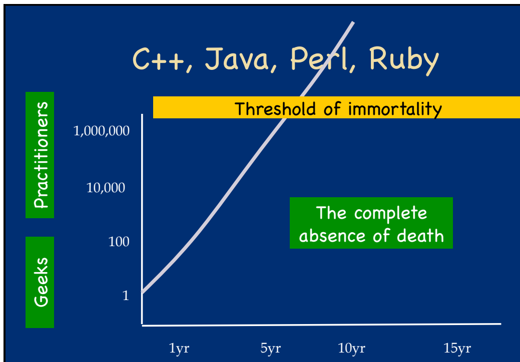
Most Research Languages



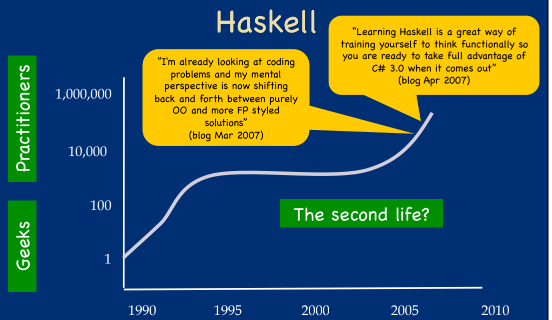
Successful Research Languages



C++, Java, Perl, Ruby



Haskell



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 $f(x) \in B$."

In ML, functions with type $A \to B$ can throw an exception, but not in Haskell.

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]
r = 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

```
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

```
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
```

Haskell Libraries

Simple Compound Types

- Tuples

```
(4, 5, "Griffendor") :: (Integer, Integer, String)
```

- Lists

```
[] :: [a]                        -- polymorphic type
```

```
1 : [2, 3, 4] :: [Integer]    -- infix cons notation
```

- Records

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

Patterns and Declarations

- Patterns can be used in place of variables
  <pat> ::= <var> | <tuple> | <cons> | <record> …
- Value declarations
  - General form
    <pat> = <exp>
  - Examples

```
myTuple = ("Flitwick", "Snape")
(x,y)  = myTuple
myList = [1, 2, 3, 4]
z:zs  = myList
```

  - Local declarations

```
let (x,y) = (2, "Snape") in x * 4
```

Functions and Pattern Matching

- Anonymous function

```
\x -> x+1      --like Lisp lambda, function (…) in JS
```

- Declaration form

<name> <pat$_1$> = <exp$_1$>
<name> <pat$_2$> = <exp$_2$> …
<name> <pat$_n$> = <exp$_n$> …

- Examples

```
f (x,y) = x+y        --actual parameter must match pattern (x,y)
length [] = 0
length (x:s) = 1 + length(s)
```

Map Function on Lists

- Apply function to every element of list

```
map f [] = []
map f (x:xs) = f x : map f xs
```

```
map (\x -> x+1) [1,2,3]        ⇒        [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

```
append ([], ys) = ys
append (x:xs, ys) = x : append (xs, ys)
```

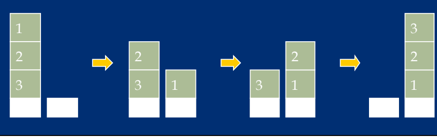
- Reverse a list

```
reverse [] = []
reverse (x:xs) = (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, [] )
```



List Comprehensions

- Notation for constructing new lists from old:

```
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]
```

Datatype Declarations

- Examples

```
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 List     = Nil    |    Cons (Atom, List)
```

  - elements are Nil, Cons(Atom "A", Nil), … Cons(Number 2, Cons(Atom("Bill"), Nil)), …

- General form

```
data <name> = <clause> | … | <clause>
<clause> ::= <constructor> | <contructor> <type>
```

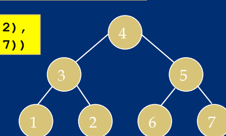
Type name and constructors must be Capitalized.

Datatypes and Pattern Matching

- Recursively defined data structure

```
data Tree = Leaf Int | Node (Int, Tree, Tree)
```

```
Node(4, Node(3, Leaf 1, Leaf 2),
        Node(5, Leaf 6, Leaf 7))
```



- Recursive function

```
sum (Leaf n) = n
sum (Node(n,t1,t2)) = n + sum(t1) + sum(t2)
```

Example: Evaluating Expressions

- Define datatype of expressions

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

Write `(x+3)+ y` as `Plus(Plus(Var 1, Const 3), Var 2)`

- Evaluation function

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

- Examples

```
ev(Plus(Const 3, Const 2))  ⇒  Const 5

ev(Plus(Var 1, Plus(Const 2, Const 3)))  ⇒
                          Plus(Var 1, Const 5)
```

Case Expression

- Datatype

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

- Case expression

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

Indentation matters in case statements in Haskell.

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)
```

Laziness

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

```
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.

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

Short-circuiting "or"

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
```

type String = [Char]

```
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

Core Haskell

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

Running Haskell

- Available on Stanford pod cluster
  - Handout on course web site on how to use.
- Or, download: http://haskell.org/ghc
- Interactive:
  - ghci intro.hs
- Compiled:
  - ghc –make AlgolAndHaskell.hs

Demo ghci

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
type TS = [Int]      -- Test at this type

prop_RevRev :: TS -> Bool
prop_RevRev ls = reverse (reverse ls) == ls
```

Test Interactively

Test.QuickCheck is simply a Haskell library (not a "tool")

```
bash$ ghci intro.hs
Prelude> :m +Test.QuickCheck

Prelude Test.QuickCheck> quickCheck prop_RevRev
+++ OK, passed 100 tests
```

...with a strange-looking type

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

Demo QuickCheck

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 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.

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

- Takes a list, and returns a list; that's all.

```
void reverse( list l )        /* 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
  - http://en.wikibooks.org/wiki/Haskell
- 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