Spring 2012

Introduction to Haskell

Mooly Sagiv
(original slides by Kathleen Fisher & John Mitchell)

Lambda Calculus

Computation Models

- Turing Machines
- Wang Machines
- Lambda Calculus

Untyped Lambda Calculus

Chapter 5

Benjamin Pierce

Types and Programming Languages

Basics

- Repetitive expressions can be compactly represented using functional abstraction
- Example:
  - (5* 4* 3 * 2 * 1) + (7 * 6 * 5 * 4 * 3 * 2 *1) =
  - factorial(5) + factorial(7)
  - factorial(n) = if n = 0 then 1 else n * factorial(n-1)
  - factorial= λn. if n = 0 then 0 else n factorial(n-1)

Untyped Lambda Calculus

t ::=	terms
x	variable
λ x. t	abstraction
t t	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 λx. t
  - otherwise it is bound
  - λx is a binder
- Examples
  - λz. λx. λy. x (y z)
  - (λx. x) x
- Terms w/o free variables are combinators
  - Identify function: id = λ x. x

Operational Semantics

$(\lambda\ x.\ t_{12})\ t_2 \to [x \mapsto t_2]\ t_{12}$ (β-reduction)

redex

$(\lambda\ x.\ x)\ y \to \quad y$

$(\lambda\ x.\ x\ (\lambda\ x.\ x)\ )\ (u\ r) \to u\ r\ (\lambda\ x.x)$

$(\lambda\ x\ (\lambda w.\ x\ w))\ (y\ z) \to \quad \lambda w.\ y\ z\ w$

Evaluation Orders

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

$(\lambda\ x.\ x)\ ((\lambda x.\ x)\ (\lambda z.\ (\lambda x.\ x)\ z))$ $\quad$ id (id (λz. id z))

Normal order

id (id (λz. id z)) $\rightarrow$

id (λz. id z) $\rightarrow$

λz. id z $\rightarrow$

λz. z $\not\rightarrow$

call-by-name

id (id (λz. id z)) $\rightarrow$

id (λz. id z) $\rightarrow$

λz. id z $\not\rightarrow$

call-by-value

id (id (λz. id z)) $\rightarrow$

id (λz. id z) $\rightarrow$

λz. id z $\not\rightarrow$

Lambda Calculus vs. JavaScript

$(\lambda\ x.\ x)\ y$

`(function (x) {return x;}) y`

Programming in the Lambda Calculus
Multiple arguments

- $f= \lambda(x, y). s$
- Currying
- $f= \lambda x. \lambda y.s$

$f\ v\ w =$

$(f\ v)\ w =$

$(\lambda x. \lambda y.s\ v)\ w \rightarrow$

$\lambda y.[x \mapsto v]s)\ w) \rightarrow$

$[x \mapsto v]\ [y \mapsto w]\ s$

Programming in the Lambda Calculus
Church Booleans

- tru = λt. λf. t
- fls =λt. λf. f
- test =λl. λm. λn. l m n
- and = λb. λc. b c fls

Programming in the Lambda Calculus
Pairs

- pair = λf. λb. λs. b f s
- fst = λp. p tru
- snd = λp. p fls

Programming in the Lambda Calculus Numerals

- $c_0 = \lambda f.\ \lambda z.\ z$
- $c_1 = \lambda f.\ \lambda z.\ s\ z$
- $c_2 = \lambda f.\ \lambda z.\ s\ (s\ z)$
- $c_3 = \lambda f.\ \lambda z.\ s\ (s\ (s\ z))$
- $scc = \lambda n.\ \lambda s.\ \lambda z.\ s\ (n\ s\ z)$
- $plus = \lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ m\ s\ (n\ s\ z)$
- $times = \lambda m.\ \lambda n.\ m\ (plus\ n)\ c_0$
- Turing Complete

Divergence in Lambda Calculus

- omega= $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$
- fix = $\lambda f.\ (\lambda x.\ f\ (\lambda\ y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda\ y.\ x\ x\ y))$

Operational Semantics

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

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

$FV(x) = \{x\}$
$FV(\lambda\ x.\ t) = FV(t) - \{x\}$
$FV\ (t_1\ t_2) = FV(t_1) \cup FV(t_2)$

$[x \mapsto s]x = s$

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

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

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

Call-by-value Operational Semantics

t ::=	terms	v ::=	values
x	variable	λ x.	abstraction values
λ x. t	abstraction		
t t	application		

$(\lambda\ x.\ t_{12})\ v_2 \to [x \mapsto v_2]\ t_{12}$ (E-AppAbs)

$$\frac{t_1 \to t'_1}{t_1\ t_2 \to t'_1\ t_2}$$ (E-APPL1)

$$\frac{t_2 \to t'_2}{v_1\ t_2 \to v_1\ t'_2}$$ (E-APPL2)

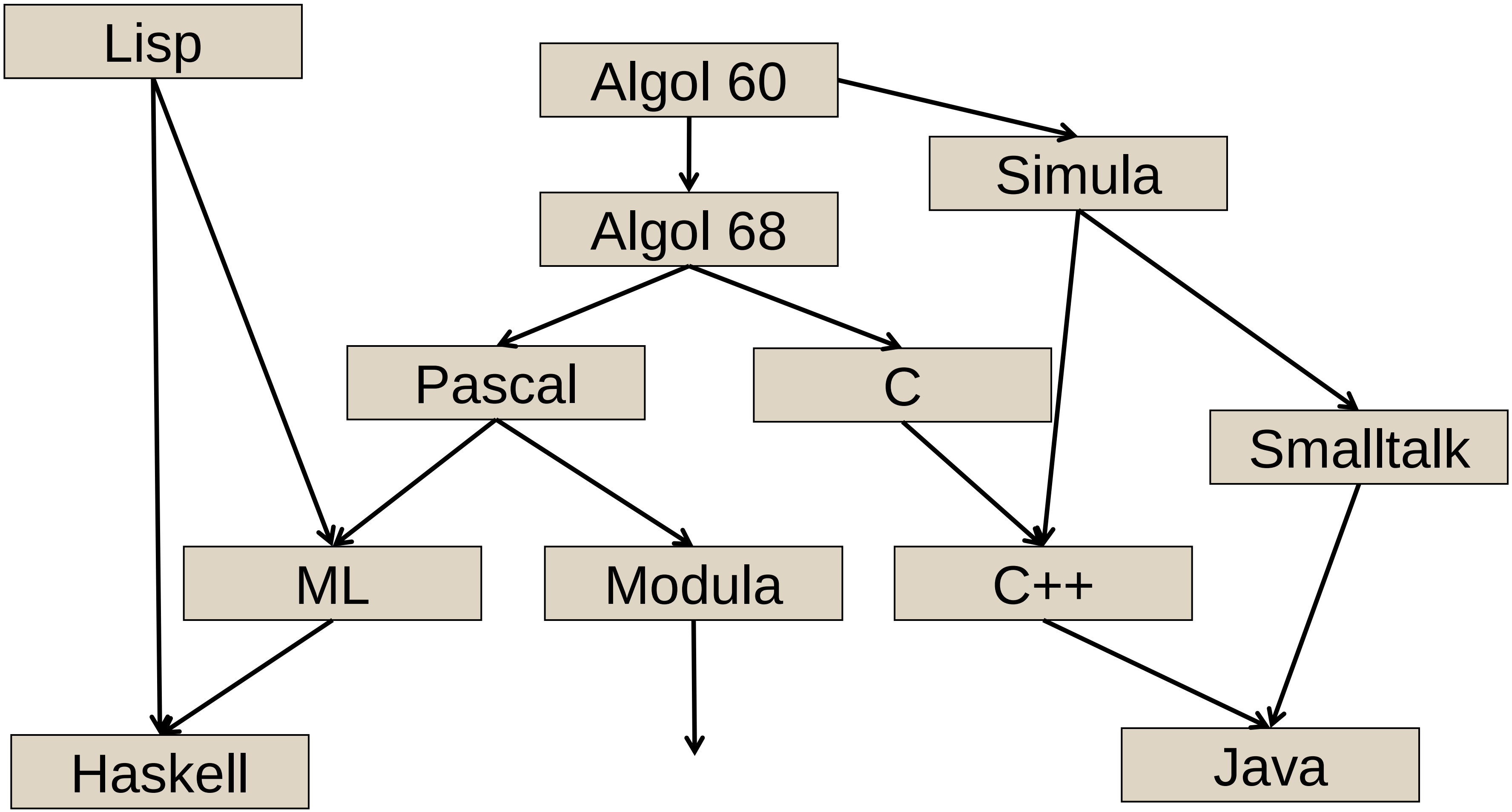
Extending the Lambda Calculus

- 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
  - …

Language Evolution



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

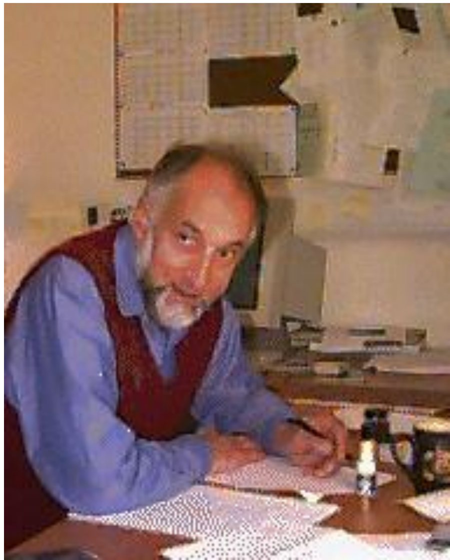
C Programming Language

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
- 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



Robin Milner, ACM Turing-Award for ML, LCF Theorem Prover, …

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

Paul Hudak

John Hughes

Simon
Peyton Jones

Phil Wadler

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

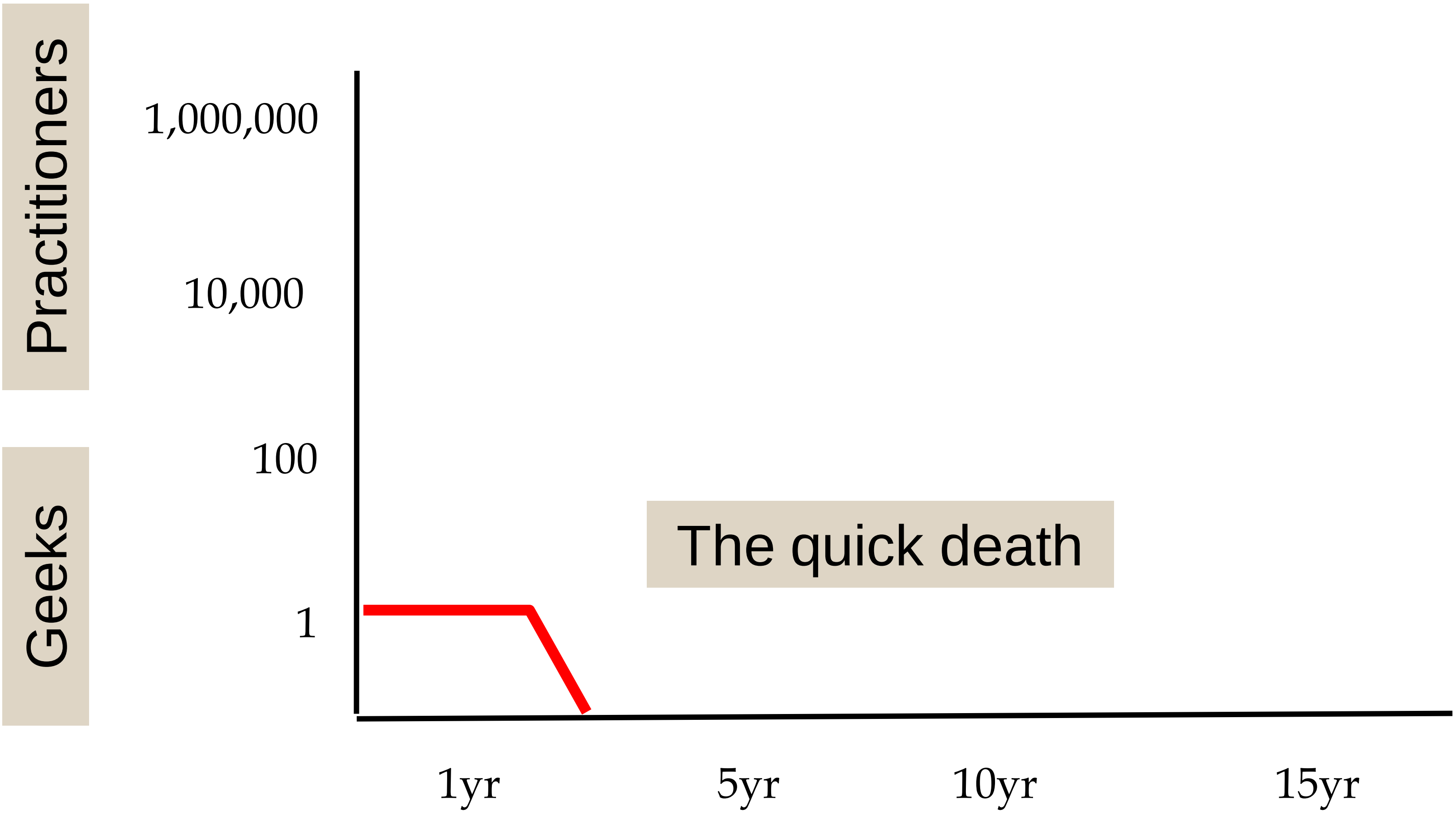
Why Study Haskell?

- 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

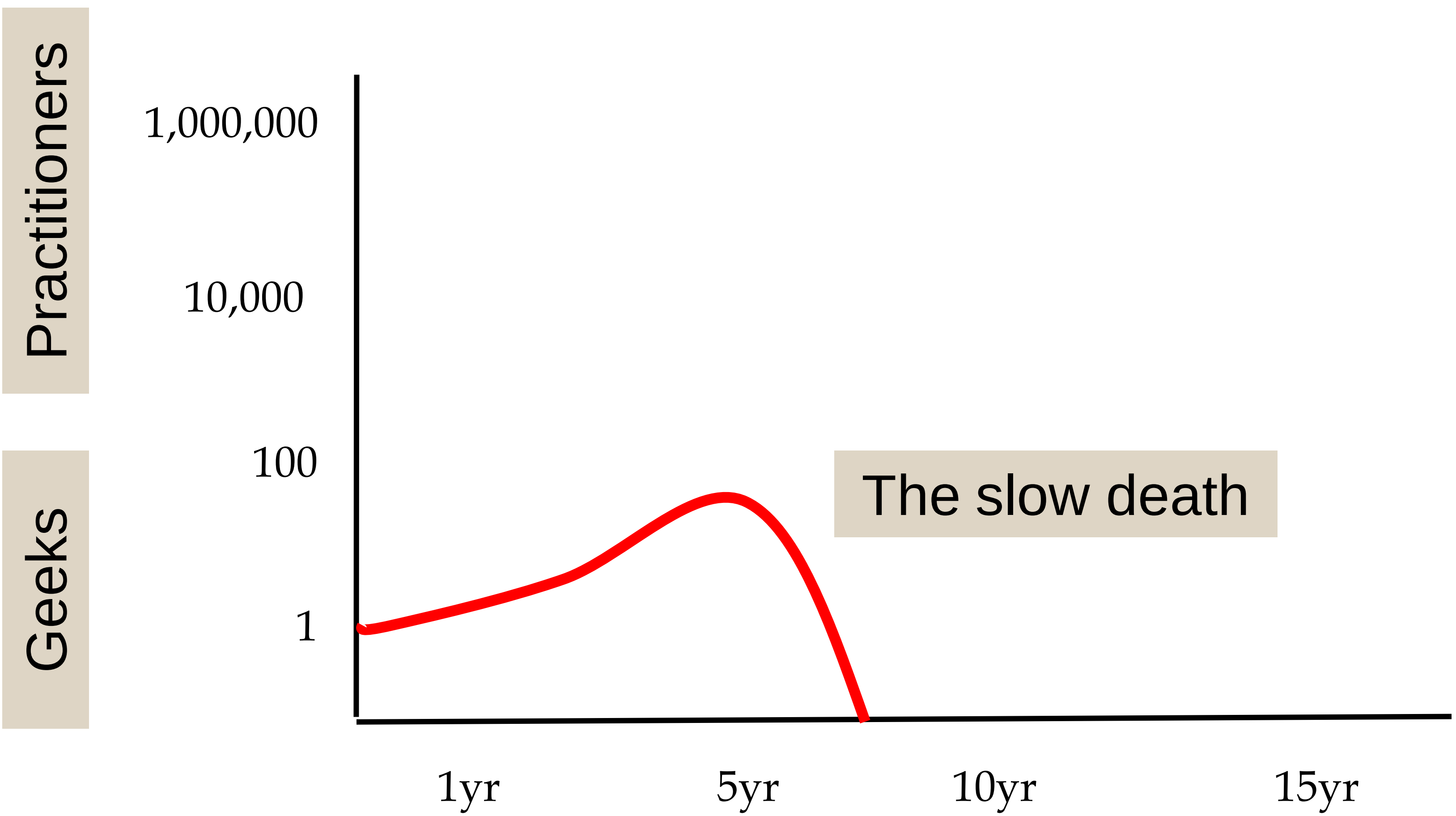
Why Study Haskell?

- 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

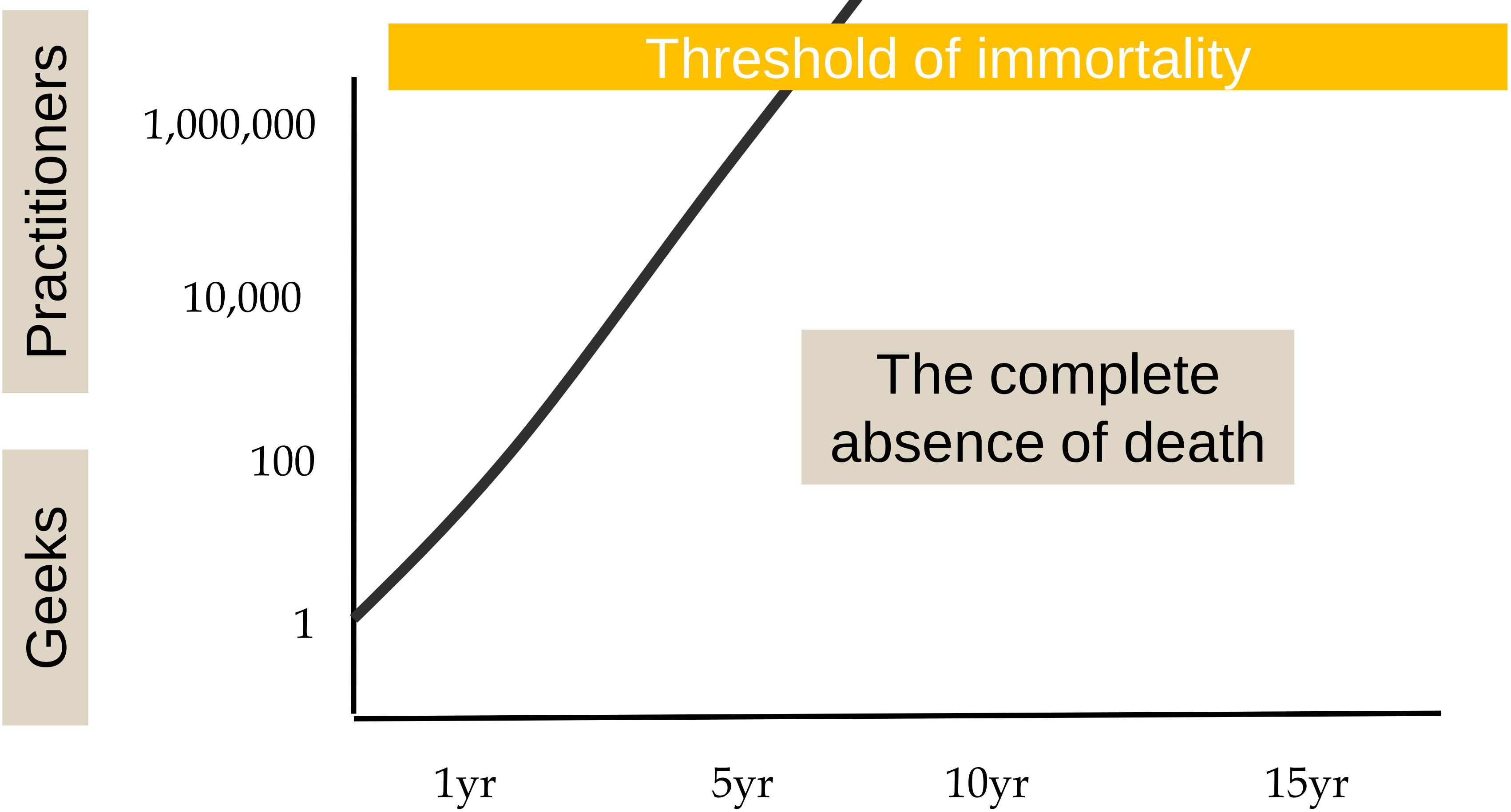
Most Research Languages



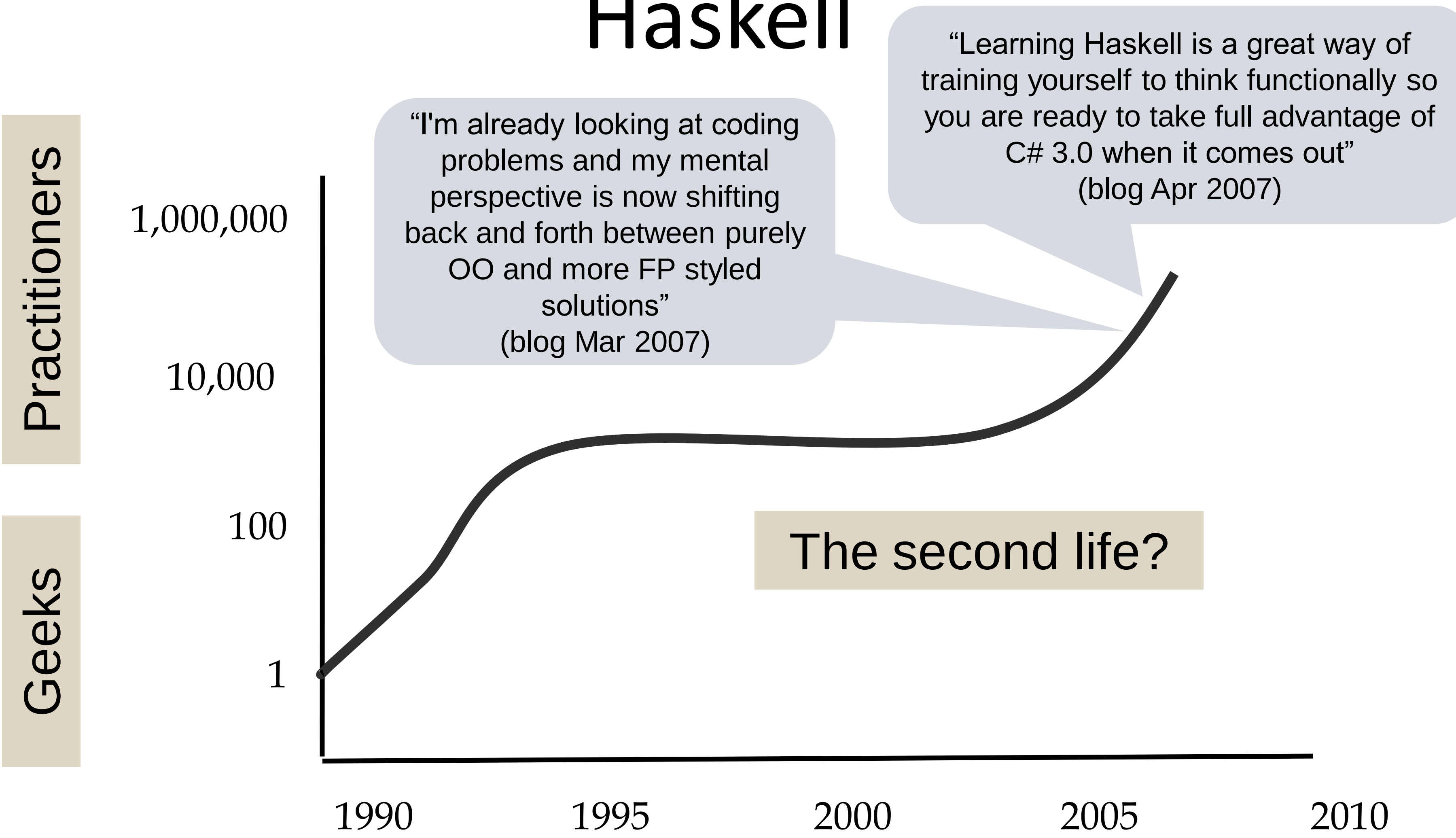
Successful Research Languages



C++, Java, Perl, Ruby



Haskell



Practitioners
Geeks
1,000,000
10,000
100
1
“I'm already looking at coding problems and my mental perspective is now shifting back and forth between purely OO and more FP styled solutions”
(blog Mar 2007)
“Learning Haskell is a great way of training yourself to think functionally so you are ready to take full advantage of C# 3.0 when it comes out”
(blog Apr 2007)
The second life?
1990
1995
2000
2005
2010

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 or have other effects, but not in Haskell

Higher Order Functions

- Functions are first class objects
  – Passed as parameters
  – Returned as results
- Practical examples
  – Google map/reduce

Example Higher Order Function

- The differential operator
  Df = f' where f'(x) = lim $_{h\downarrow o}$ (f(x+h)-f(x))/h
- In Haskel
```
diff f = f_
         where
          f_ x = (f  (x +h) – f x) / h
          h = 0.0001
```
- diff :: (float -> float) -> (float -> float)
- (diff square) 0 = 0.0001
- (diff square) 0.0001 = 0.0003
- (diff (diff square)) 0 = 2

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
```

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
```

- Function declaration form

  <name> <pat$_1$> = <exp$_1$>

  <name> <pat$_2$> = <exp$_2$> …

  <name> <pat$_n$> = <exp$_n$> …

- Examples

```
f (x,y) = x+y     --argument 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)
```

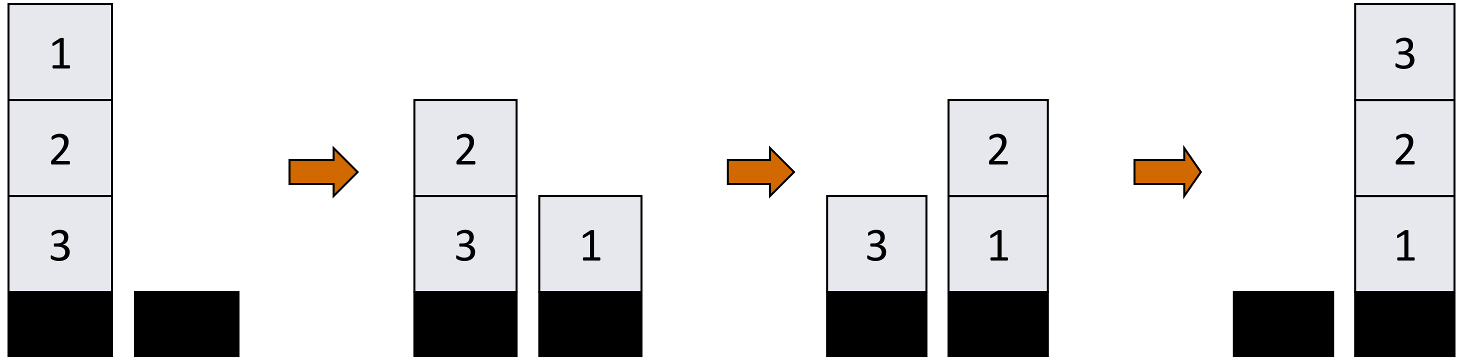
- Reverse a list

```
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]
```

- Similar to “set comprehension”

  $\{ x \mid x \in Odd \wedge x > 6 \}$

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>
```

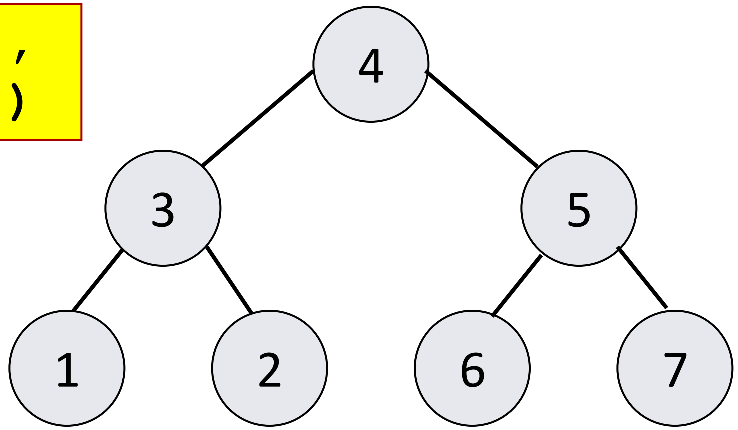
– Type name and constructors must be Capitalized

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

Offside rule

- Layout characters matter to parsing

```
divide x 0 = inf
divide x y = x / y
```

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

```
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

```
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

Short-circuiting "or"

```
(||) :: 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
```

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

Benefits of Lazy Evaluation

- Define streams

  ```
  main = take 100 [1 .. ]
  ```

- ```
  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
  ```

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

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

PL	types	evaluation	Side-effect
scheme	Weakly typed	Eager	yes
ML OCAML F#	Polymorphic strongly typed	Eager	References
Haskel	Polymorphic strongly typed	Lazy	None

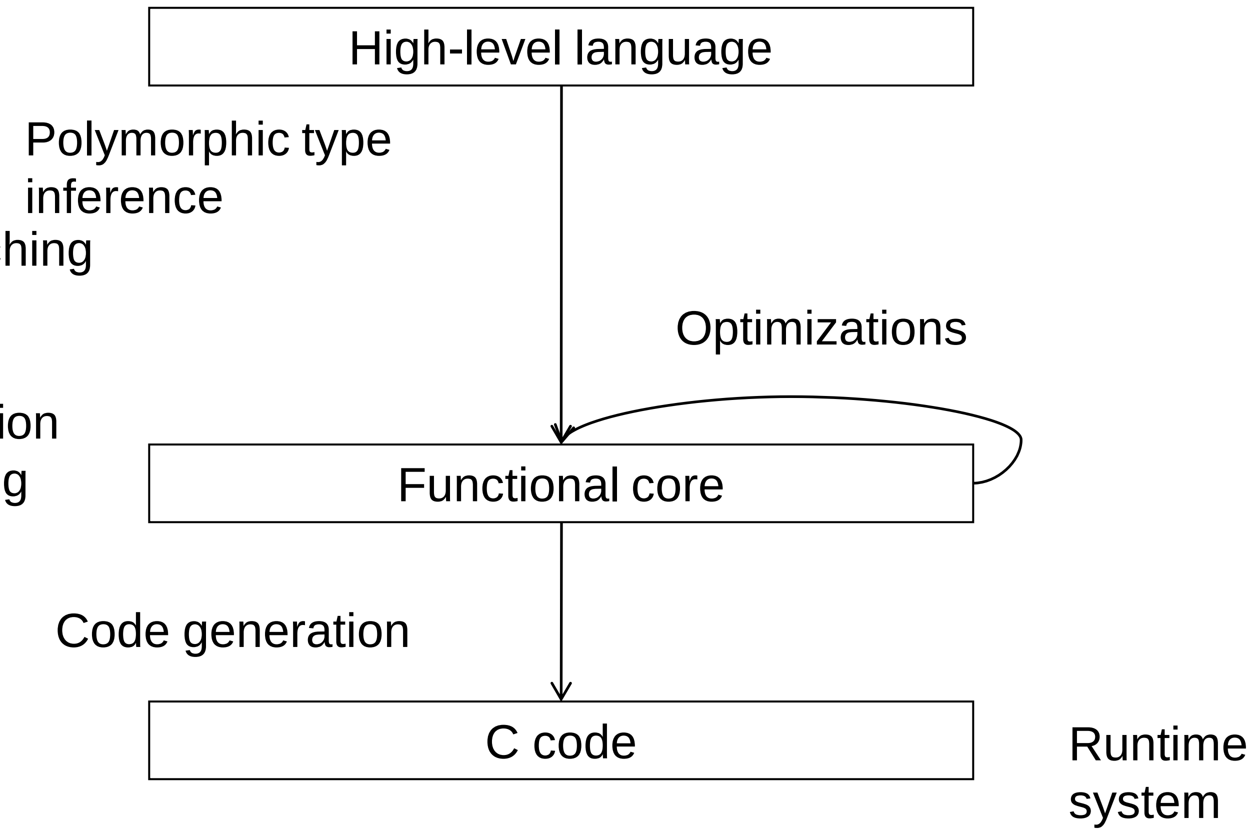
Compiling Functional Programs

Compiler Phase	Language Aspect
Lexical Analyzer	Offside rule
Parser	List notation List comprehension Pattern matching
Context Handling	Polymorphic type checking
Run-time system	Referential transparency Higher order functions Lazy evaluation

Structure of a functional compiler

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 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 = 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

```
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

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