THE ALGOL FAMILY AND HASKELL

Kathleen Fisher for cs242 at Tufts
Lightly edited with permission, Michelle Strout 4/10/15

"Real World Haskell", Chapter 0 and Chapter 1
(http://book.realworldhaskell.org/)

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
  - 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

```plaintext
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;
```

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

- **Question:**
  - Is \( x := x \) equivalent to doing nothing?
- **Interesting answer in Algol:**
  
  ```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
- Problems with parameter passing mechanisms
  - Pass-by-value expensive for arrays

**Algol 60 Pass-by-name**

- **Substitute text of actual parameter**
  - Unpredictable with side effects!
- **Example**
  
  ```algol
  procedure inc2(i, j);
  integer i, j;
  begin
    i := i+1;
    j := j+1
  end;
  inc2 (k, A[k]);
  ```

  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**
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

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

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

Limitations of Pascal
- Array bounds part of type

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

Illegal

- 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 its first element
  - `E1[E2]` is equivalent to `ptr dereference: *((E1)+(E2))`
- 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 '99, ongoing.
- "A History of Haskell: Being Lazy with Class" HOPL 3

---

**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.

Successful Research Languages

Geeks

Practitioners

1yr 5yr 10yr 15yr

The slow death

Most Research Languages

Geeks

Practitioners

1yr 5yr 10yr 15yr

The quick death

C++, Java, Perl, Ruby

Geeks

Practitioners

1yr 5yr 10yr 15yr

The complete absence of death

Threshold of immortality
Function Types in Haskell

In Haskell, \( f : A \rightarrow B \) means for every \( x \in A \),

\[
\begin{align*}
 f(x) & = \begin{cases}
 \text{some element } y = f(x) \in B & \text{run forever} \\
 \end{cases}
\end{align*}
\]

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

In ML, functions with type \( A \rightarrow B \) 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.

    ```haskell
    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)
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

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> :: <exp> | <pat> | <exp> | <recs> ..

Value declarations
General form
<pat> = <exp>

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

Local declarations
let (x,y) = (2, "Snape") in x + 4
Functions and Pattern Matching

- **Anonymous function**
  \[ x \rightarrow x + 1 \]  
  - Like Lisp lambda, Function ( ) to JS

- **Declaration form**
  `<name> <pat1> = <exp1>`, `<name> <pat2> = <exp2>`, ...

- **Examples**
  - \( f(x, y) = x + y \) --actual parameters must match pattern (x, y)
  - \( \text{length } [] = 0 \)
  - \( \text{length } (x:xs) = 1 + \text{length}(xs) \)

Map Function on Lists

- **Apply function to every element of list**
  - \( \text{map } f[] = [] \)
  - \( \text{map } f[(x:xs)] = f x : \text{map } f xs \)

- **Compare to Lisp**
  - (define map
    (lambda (f xs)
      (if (eq? xs () ) ()
        (cons (f (car xs)) (map f (cdr xs))))))

- Example
  \( \text{map } (\lambda x \rightarrow x + 1) [1,2,3] \rightarrow [2,3,4] \)

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 \text{reverse}?
  - Can it be done with only one pass through list?

More Efficient Reverse

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

- Example
  \( [1,2,3] \rightarrow [3,2,1] \rightarrow [1,2,3] \)

- Another example
  \( [a,b,c,d] \rightarrow [d,c,b,a] \rightarrow [a,b,c,d] \)
List Comprehensions

- Notation for constructing new lists from old:

  \[ \text{myData} = [1,2,3,4,5,6,7] \]
  \[ \text{twiceData} = [2 \times x \mid x \in \text{myData}] \rightarrow [2,4,6,8,10,12,14] \]
  \[ \text{twiceEvenData} = [2 \times x \mid x \in \text{myData}, x \mod 2 == 0] \rightarrow [4,8,12] \]

Examples

- elements are \text{Red}, \text{Yellow}, \text{Blue}
- elements are \text{Atom "A"}, \text{Atom "B"}, ..., \text{Number 0}, ...
- \text{Cons (Number 2, Cons (Atom "Bill"), Nil))}, ...

Datatype Declarations

- General form

  data name = \text{clause} | ... | \text{clause}
  \text{clause} ::= \text{constructor} | \text{constructor 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)}
  \[
  \text{Node(4, Node(3, Leaf 1, Leaf 2), Node(5, Leaf 6, Leaf 7))}
  \]

- 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

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

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

- Evaluation function

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

  \text{ev(Plus(Const 3, Const 2))} \rightarrow \text{Const 5}
  \text{ev(Plus(Var 1, Plus(Const 2, Const 3)))} \rightarrow \text{Plus(Var 1, Const 5)}
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.

```
(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
```

Haskell is a lazy language! Functions and data constructors don't evaluate their arguments until they need them. A programmer can write control-flow operators that have to be built-in in eager languages.
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](http://haskell.org/ghc)
- Interactive:
  - ghci intro.hs
- Compiled:
  - ghc -make AlgolAndHaskell.hs

Testing
- It's good to write tests as you write code
  - E.g. reverse undoes itself, etc.

Demo ghci
Test Interactively

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

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

Things to Notice

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

No side effects. At all.

- 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.

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

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.

Pure functions are easy to test.

prop_RevRev 1 = reverse(reverse 1) == 1

- 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
Types are everywhere.

- 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.

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

More Info: haskell.org

- The Haskell wikibook
- All the Haskell bloggers, sorted by topic
  [http://haskell.org/haskellwiki/Blog_articles](http://haskell.org/haskellwiki/Blog_articles)
- Collected research papers about Haskell
  [http://haskell.org/haskellwiki/Research_papers](http://haskell.org/haskellwiki/Research_papers)
- Wiki articles, by category
- Books and tutorials
  [http://haskell.org/haskellwiki/Books_and_tutorials](http://haskell.org/haskellwiki/Books_and_tutorials)