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/)

Thanks to John Mitchell and Simon Peyton Jones for some of these slides.

Algol 60

- Basic Language of 1960
  - Simple imperative language + functions
  - Successful syntax, BNF -- used by many successors
    - statement oriented
    - begin ... end blocks (like C { ... })
    - if .. 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

`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];`
`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:
  ```algol60```
  ```
  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

- Problems with parameter passing mechanisms
  - Pass-by-name "Copy rule" duplicates code,
    interacting badly with side effects
  - Pass-by-value expensive for arrays

- Some awkward control issues
  - goto out of block requires memory management
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]);
```

Is this what you expected?

Algol 68

- 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
  - Fixed some problems of Algol 60
  - Eliminated pass-by-name
  - Not widely adopted

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 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);
  ```
- 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!
- 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, rapidly evolving type system.
- Designed by committee in 80's and 90's to unify research efforts in lazy languages.
  - Haskell 0.1 in 1990, Haskell '98, "Haskell" ongoing.
  - "A History of Haskell: Being Lazy with Class" HOPL 3
- 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

<table>
<thead>
<tr>
<th>Year</th>
<th>Geeks</th>
<th>Hackers</th>
</tr>
</thead>
<tbody>
<tr>
<td>5yr</td>
<td>1</td>
<td>1,000,000</td>
</tr>
<tr>
<td>10yr</td>
<td>100</td>
<td>10,000</td>
</tr>
<tr>
<td>15yr</td>
<td>1,000</td>
<td>1,000</td>
</tr>
</tbody>
</table>

The quick death
**Successful Research Languages**

![Graph showing the slow death of Geeks vs Practitioners](image1)

**C++, Java, Perl, Ruby**

![Graph showing the complete absence of death](image2)

**Haskell**

![Graph showing the second life?](image3)

**Function Types in Haskell**

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

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

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

In ML, functions with type \( A \rightarrow B \) can throw an exception, 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.
  - Google uses Map/Reduce to parallelize and distribute massive data processing tasks.

**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
  16 :: Integer
  Prelude> if 5>3 then "Harry" else "Hermione"
  "Harry" :: [Char]
  Prelude> 5==4
  False
  Prelude> 14 :: Bool
  ```
Overview by Type

- **Booleans**
  
  `True, False :: Bool`

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

- **Strings**
  
  "Ron Weasley"

- **Floats**
  
  `1.0, 2, 3.14159, ...`

Simple Compound Types

- **Tuples**
  
  `[4, 5, "Griffendor"] :: (Integer, Integer, String)`

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

- **Records**
  
  `data Person = Person {firstName :: String, lastName :: String}`

Patterns and Declarations

- **Patterns** can be used in place of variables

  `(x, y) :: Cons | tuple | cons | records ...`

- **Value declarations**

  - General form
    
    `<pat> ::= <var> | <tuple> | <cons> | <record> ...`

  - Examples

    - `myTuple = ("Flitwick", "Snape")`
    - `myList = [1, 2, 3, 4]`
    - `xs = myList`

- **Local declarations**

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

Functions and Pattern Matching

- **Anonymous function**

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

- **Declaration form**

  - `\<name><pat1> = <exp1> ...`
  
  - `\<name><pat2> = <exp2> ...`
  
  - `\<name><patn> = <expn> ...`

  - `f (x, y) = x+y`  --- actual parameter must match pattern `(x, y)`

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`

    ```lisp`
    (lambda (f xs)
      (if (null? xs) ()
        (cons (f (car xs)) (map f (cdr xs)))))
    ```

More Functions on Lists

- **Append lists**

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

- **Reverse a list**

  - `reverse [] = []`
  
  - `reverse (xs:ys) = (reverse ys) ++ [xs]`

Questions

- **How efficient is `reverse`?**
- **Can it be done with only one pass through list?**
More Efficient Reverse

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

Examples

```haskell
data Color = Red | Yellow | Blue
data Atom = Atom String | Number Int
data List = Nil | Cons (Atom, List)
data <name> = <clause> | … | <clause>
    <clause> ::= <constructor> | <constructor> <type>

Define datatype of expressions

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

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", …
data List = Nil | Cons (Atom, List)
  elements are Nil, Cons (Atom "A", Nil), …
```

Examples

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

Datatypes and Pattern Matching

```haskell
data Tree = Leaf Int | Node (Int, Tree, Tree)
Node(4, Node(3, Leaf 1, Leaf 2), Node(5, Leaf 6, Leaf 7))
```

Recursive function

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

Recursive defined data structure

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

Example: Evaluating Expressions

Define datatype of expressions

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

```haskell
Write (x+3)+y as
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
```

Case Expression

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

```haskell
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)
```

```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
Ev(Plus(Var 1, Plus(Var 2, Var 3))) -> …
```
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.

Using Laziness

isSubString :: String -> String -> Bool
isSubString s t = or [ x isPrefixOf t | t <- suffixes s ]

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

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

- Download:
  - ghc: [http://haskell.org/ghc](http://haskell.org/ghc)
  - Hugs: [http://haskell.org/hugs](http://haskell.org/hugs)
- Interactive:
  - ghci intro.hs
  - hugs intro.hs
- Compiled:
  - ghc --make intro.hs

Testing

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

```
reverse xs = foldr (\x y -> y : x) [] xs

-- Write properties in Haskell
prop_Reversal :: T5 -> Bool
prop_Reversal ls = reverse (reverse ls) == ls
```
**Test Interactively**

bash$ ghci intro.hs

Prelude> :m +Test.QuickCheck

Prelude Test.QuickCheck> quickCheck prop_RevRev

+++ OK, passed 100 tests

-----

**Things to Notice**

No side effects. At all.

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

  ```haskell
  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.

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

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

- Takes a list; may modify it; may modify other persistent state; may do I/O.

---

**Things to Notice**

Pure functions are easy to test.

- 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 postcondition

---

**Things to Notice**

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.

---

**More Info: haskell.org**

- The Haskell wikibook

- All the Haskell bloggers, sorted by topic
  - [http://haskell.org/haskellwiki/Category:Bloggers](http://haskell.org/haskellwiki/Category:Bloggers)

- Collected research papers about Haskell
  - [http://haskell.org/haskellwiki/Research_papers](http://haskell.org/haskellwiki/Research_papers)

- Wiki articles, by category
  - [http://haskell.org/haskellwiki/Category:Haskell](http://haskell.org/haskellwiki/Category:Haskell)

- Books and tutorials
  - [http://haskell.org/haskellwiki/Books_and_tutorials](http://haskell.org/haskellwiki/Books_and_tutorials)