The Algol Family and ML

John Mitchell

Reading: Chapter 5

Language Sequence

Algol 60
Algol 68
Pascal
ML
Modula

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 { ... } )
  – if ... then ... else
• Recursive functions and stack storage allocation
• Fewer ad hoc restrictions than Fortran
  – General array references: A[x + [I]^y]
• Type discipline was improved by later languages
• Very influential but not widely used in US

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;

Algol Joke

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

Type discipline improved by later languages
• parameter types can be array
  – no array bounds
• parameter type can be procedure
  – no argument or return types for procedure parameter

Parameter passing methods
• Pass-by-name had various anomalies
  – "Copy rule" based on substitution, interacts 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"
  - vW grammars instead of BNF
    - context-sensitive grammar invented by A. van Wijngaarden
  - Elaborate type system
  - Complicated type conversions
- Fixed some problems of Algol 60
  - Eliminated pass-by-name
  - Not widely adopted

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 and structured 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

Source: Tanenbaum, Computing Surveys

Pascal

- Revised type system of Algol
  - Good data-structuring concepts
  - 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
- 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 for writing Unix by Dennis Ritchie
- Evolved from B, which was based on BCPL
  - B was an untyped language; C adds some checking
- Relation 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

- Typed programming language
- Intended for interactive use
- Combination of Lisp and Algol-like features
  - Expression-oriented
  - Higher-order functions
  - Garbage collection
  - Abstract data types
  - Module system
  - Exceptions
  - General purpose non-C-like, not OO language
  - Related languages: Haskell, OCAML, ...

Why study ML?

- Learn an important language that’s different
- Discuss general programming languages issues
  - Types and type checking
    - General issues in static/dynamic typing
    - Type inference
    - Polymorphism and Generic Programming
  - Memory management
    - Static scope and block structure
    - Function activation records, higher-order functions
  - Control
    - Force and delay
    - Exceptions
    - Tail recursion and continuations

History of ML

- Robin Milner
- Logic for Computable Functions
  - Stanford 1970-71
  - Edinburgh 1972-1995
- Meta-Language of the LCF system
  - Theorem proving
  - Type system
  - Higher-order functions

Logic for Computable Functions

- Dana Scott, 1969
  - Formulate logic for proving properties of typed functional programs
- Milner
  - Project to automate logic
  - Notation for programs
  - Notation for assertions and proofs
  - Need to write programs that find proofs
    - Too much work to construct full formal proof by hand
  - Make sure proofs are correct

LCF proof search

- Tactic: function that tries to find proof
  
  tactic(formula) =
  
  succeed and return proof
  search forever
  fail

- Express tactics in the Meta-Language (ML)
- Use type system to facilitate correctness
Tactics in ML type system

- Tactic has a functional type
  \[ \text{tactic : formula} \rightarrow \text{proof} \]
- Type system must allow "failure"

\[ \text{tactic(formula)} = \begin{cases} 
\text{succeed and return proof} \\
\text{search forever} \\
\text{fail and raise exception}
\end{cases} \]

Function types in ML

\[ f : A \rightarrow B \text{ means} \]

for every \( x \in A \),

\[ f(x) = \begin{cases} 
\text{some element } y = f(x) \in B \\
\text{run forever} \\
\text{terminate by raising an exception}
\end{cases} \]

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

Addition never occurs in \( f(x) + 3 \) if \( f(x) \) raises exception.

This form of function type arises directly from motivating application for ML. Integration of type system and exception mechanism mentioned in Milner's 1991 Turing Award.

Higher-Order Functions

- Tactic is a function
- Method for combining tactics is a function on functions
- Example:

\[ f(\text{tactic}_1, \text{tactic}_2) = \lambda \text{formula}. \begin{cases} 
\text{try tactic}_1(\text{formula}) \\
\text{else tactic}_2(\text{formula})
\end{cases} \]

Basic Overview of ML

- Interactive compiler: read-eval-print
  - Compiler infers type before compiling or executing
  - Type system does not allow casts or other loopholes.
- Examples

\[ \begin{align*}
- (5+3)-2; \\
> \text{val it} = 6 : \text{int}
\end{align*} \]

\[ \begin{align*}
- \text{if } 5>3 \text{ then } "\text{Bob}" \text{ else } "\text{Fido}"; \\
> \text{val it} = "\text{Bob}" : \text{string}
\end{align*} \]

\[ \begin{align*}
- 5=4; \\
> \text{val it} = \text{false} : \text{bool}
\end{align*} \]

Overview by Type

- Booleans
  - true, false : bool
  - if ... then ... else ... \hspace{1em} \text{(types must match)}
- Integers
  - 0, 1, 2, ... : int
  - +, *, ... : int * int \rightarrow int \hspace{1em} \text{and so on ...}
- Strings
  - "Austin Powers"
- Reals
  - 1.0, 2.2, 3.14159, ... \hspace{1em} \text{decimal point used to disambiguate}

Compound Types

- Tuples
  - (4, 5, "noxious") : int * int * string
- Lists
  - nil
  - 1 :: [2, 3, 4] \hspace{1em} \text{infix cons notation}
- Records
  - \{ name = "Fido", hungry=true \}
  - \{ name : string, hungry : bool \}
Patterns and Declarations

- Patterns can be used in place of variables
  
  ```
  <pat> ::= <var> | <tuple> | <cons> | <record> ...
  ```

- Value declarations
  
  **General form**
  
  ```
  val <pat> = <exp>
  ```

  **Examples**
  
  ```
  val myTuple = ("Conrad", "Lorenz");
  val (x,y) = myTuple;
  val myList = [1, 2, 3, 4];
  val x::rest = myList;
  ```

  **Local declarations**
  
  ```
  let val x = 2+3 in x*4 end;
  ```

Functions and Pattern Matching

- Anonymous function
  
  ```
  fn x => x+1; 
  ```

  like Lisp lambda

- Declaration form
  
  ```
  fun <name> <pat1> = <exp1>
  |     <name> <pat2> = <exp2> ...
  |     <name> <patn> = <expn> ...
  ```

  **Examples**
  
  ```
  fun f (x,y) = x+y; 
  ```

  actual par must match pattern (x,y)

  ```
  fun length nil = 0
  | length (x::s) = 1 + length(s);
  ```

Map function on lists

- Apply function to every element of list
  
  ```
  fun map (f, nil) = nil
  |     map (f, x::xs) = f(x) :: map(f,xs);
  ```

  ```
  map (fn 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

- Reverse a list
  
  ```
  fun reverse nil = nil
  | reverse (x::xs) = append ((reverse xs), [x]);
  ```

- Append lists
  
  ```
  fun append(nil, ys) = ys
  | append(x::xs, ys) = x :: append(xs, ys);
  ```

- Questions
  
  ```
  How efficient is reverse?
  Can you do this with only one pass through list?
  ```

More efficient reverse function

```text
fun reverse xs = 
  let fun rev (nil, z) = (nil, z) 
  |     rev(y::ys, z) = rev(ys, y::z)
  val (u,v) = rev(xs,nil)
  in v
  end;
```

Datatype Declarations

- General form
  
  ```
  datatype <name> = <clause> | ... | <clause>
  ```

- Declaration form
  
  ```
  <clause> ::= <constructor> |<contructor> of <type>
  ```

- Examples
  
  ```
  datatype color = red | yellow | blue
  ```

  - elements are red, yellow, blue
  
  ```
  datatype atom = atm of string | nmbr of int
  ```

  - elements are atm("A"), atm("B"), ..., nmbr(0), nmbr(1), ...

  ```
  datatype list = nil | cons of atom*list
  ```

  - elements are nil, cons(atm("A"), nil), ...

  ```
  datatype nlist = nil | cons of nlist
  ```

  - elements are nil, cons(natm", atom(ugh"), nil), ...

  ```
  cons(natm"B", cons(natm"A", nil)), ...
  ```
Datatype and pattern matching

Recursively defined data structure

```ml
datatype tree = leaf of int | node of int*tree*tree
```

Recursive function

```ml
fun sum (leaf n) = n
|    sum (node(n,t1,t2)) = n + sum(t1) + sum(t2)
```

Example: Evaluating Expressions

Define datatype of expressions

```ml
datatype exp = Var of int | Const of int | Plus of exp*exp;
```

Evaluation function

```ml
fun ev(Var(n)) = Var(n)
|    ev(Const(n)) = Const(n)
|    ev(Plus(e1,e2)) = ev(Plus(e1,e2))
```

Examples

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

Case expression

```ml
datatype exp = Var of int | Const of int | Plus of exp*exp;
```

Case expression

```ml
fun 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)) );
```

Core ML

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

Patterns
- Declarations
- Functions
- Polymorphism
- Overloading
- Type declarations
- Exceptions
- Reference Cells

Variables and assignment

General terminology: L-values and R-values
- Assignment `y := x+3`
  - Identifier on left refers to a memory location, called L-value
  - Identifier on right refers to contents, called R-value

Variables
- Most languages
  - A variable names a storage location
  - Contents of location can be read, can be changed
- ML reference cell
  - A mutable cell is another type of value
  - Explicit operations to read contents or change contents
  - Separates naming (declaration of identifiers) from "variables"
ML imperative constructs

- ML reference cells
  - Different types for location and contents
    - \( x : \text{int} \) non-assginable integer value
    - \( y : \text{int ref} \) location whose contents must be an integer
    - \( !y \) the contents of location \( y \)
    - \( \text{ref} \ x \) expression creating new cell initialized to \( x \)
  - ML assignment
    - operator := applied to memory cell and new contents
  - Examples
    - \( y := x + 3 \) place value of \( x + 3 \) in cell \( y \); requires \( x : \text{int} \)
    - \( y := !y + 3 \) add 3 to contents of \( y \) and store in location \( y \)

ML examples

- Create cell and change contents
  - \( \text{val} \ x = \text{ref} \ "\text{Bob}"; \)
  - \( x := \"\text{Bill}"; \)
- Create cell and increment
  - \( \text{val} \ y = \text{ref} \ 0; \)
  - \( y := !y + 1; \)
- While loop
  - \( \text{val} \ i = \text{ref} \ 0; \)
  - \( \text{while} \ !i < 10 \ do \ i := !i + 1; \)
  - \( !i; \)

Core ML

- Basic Types
  - Unit
  - Booleans
  - Integers
  - Strings
  - Reals
  - Tuples
  - Lists
  - Records
- Patterns
- Declarations
- Functions
- Polymorphism
- Overloading
- Type declarations
- Exceptions
- Reference Cells