The Algol Family and ML

John Mitchell

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[i + B[3]*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
  
  ```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

- 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
    - 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
  ```pascal
  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!!
**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

**Goals in study of ML**

- Survey a modern procedural language
- 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

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

- 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”
Function types in ML

\[ f : A \to B \] means
for every \( x \in A \),
\[
f(x) = \begin{cases} 
some element y = f(x) \in B \\
runtime forever \\
terminate by raising an exception
\end{cases}
\]

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:
\[
f(tactic_1, tactic_2) = \\
\lambda \text{formula}. \begin{cases} 
try tactic_1(\text{formula}) \\
\text{else } tactic_2(\text{formula})
\end{cases}
\]

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
    - \((5+3)-2\);
      \[> \text{val it} = 6 : \text{int} \]
    - \(5>3\) then "Bob" else "Fido";
      \[> \text{val it} = "Bob" : \text{string} \]
    - \(5=4\);
      \[> \text{val it} = false : \text{bool} \]

Overview by Type

- Booleans
  - true, false : bool
  - if ... then ... else ...
  - (types must match)

- Integers
  - \(0, 1, 2, ... : \text{int} \)
  - \(+, *, ... : \text{int} \to \text{int} \to \text{int} \) and so on ...

- Strings
  - "Austin Powers"

- Reals
  - \(1.0, 2.2, 3.14159, ... \)
    - decimal point used to disambiguate

Compound Types

- Tuples
  - \((4, 5, "noxious") : \text{int} \times \text{int} \times \text{string} \)

- Lists
  - nil
  - \(1 :: [2, 3, 4] \) \(\text{infix cons notation} \)

- Records
  - \{name = "Fido", hungry=true\}
  - \{name : string, hungry : bool\}

Patterns and Declarations

- Patterns can be used in place of variables
  - \(<\text{pat}> ::= <\text{var}> | <\text{tuple}> | <\text{cons}> | <\text{record}> \) ...

- Value declarations
  - General form
    - \(\text{val } <\text{pat}> = <\text{exp}> \)
  - Examples
    - \(\text{val myTuple } = ("Conrad", "Lorena"); \)
    - \(\text{val } (x,y) = \text{myTuple}; \)
    - \(\text{val myList } = [1, 2, 3, 4]; \)
    - \(\text{val x: rest } = \text{myList} \)
  - Local declarations
    - \(\text{let val } x = 2+3 \text{ in } x^4 \text{ end; } \)
Functions and Pattern Matching

- **Anonymous function**
  - \( \text{fn } x \mapsto x+1; \) like Lisp lambda

- **Declaration form**
  - \( \text{fun } <\text{name}> <\text{pat}_1> = <\text{exp}_1> \)
  - \( <\text{name}> <\text{pat}_2> = <\text{exp}_2> \) ... 
  - \( <\text{name}> <\text{pat}_n> = <\text{exp}_n> \) ...

- **Examples**
  - \( \text{fun } f (x,y) = x+y; \) actual par must match pattern \((x,y)\)
  - \( \text{fun length nil } = 0 \)
  - \( \text{length (x::s) } = 1 + \text{length(s)}; \)

Map function on lists

- **Apply function to every element of list**
  - \( \text{fun map } (f, \text{nil}) = \text{nil} \)
  - \( \text{map } (f, x::xs) = f(x) :: \text{map } f, xs; \)
  - \( \text{map } (\text{fn } x \mapsto x+1, \{1,2,3\}) \rightarrow \{2,3,4\} \)

- **Compare to Lisp**
  - \( \text{(define map} \)
  - \( \text{(lambda } (f \text{x}s) \)
  - \( \text{(if } (\text{eq? } \text{x}s () ) () \)
  - \( \text{(cons } f \text{(car } \text{x}s ) (\text{map } f \text{(cdr } \text{x}s ))) \)
  - \( )); \)

More functions on lists

- **Reverse a list**
  - \( \text{fun reverse nil } = \text{nil} \)
  - \( \text{reverse (x::xs) } = \text{append } ((\text{reverse } xs), \{x\}); \)

- **Append lists**
  - \( \text{fun append(nil, ys) } = ys \)
  - \( \text{append(x::xs, y::ys) } = x :: \text{append(xs, ys);} \)

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

Datatype Declarations

- **General form**
  - \( \text{datatype } \langle \text{name} \rangle = \langle \text{clause} \rangle | ... | \langle \text{clause} \rangle \)

- **Examples**
  - \( \text{datatype color = red | yellow | blue} \)
    - elements are red, yellow, blue
  - \( \text{datatype atom = atm of string | nmbr of int} \)
    - elements are atm('A'), atm('B'), ..., nmbr(0), nmbr(1), ...
  - \( \text{datatype list } = \text{nil} | \text{cons of atom*list} \)
    - elements are nil, cons(atm('A'), nil), ...

Datatype and pattern matching

- **Recursively defined data structure**
  - \( \text{datatype tree } = \text{leaf of int} | \text{node of int*tree*tree} \)
    - \text{node(4, node(3,leaf(1), leaf(2)), node(5,leaf(6), leaf(7))})

- **Recursive function**
  - \( \text{fun sum } (\text{leaf } n) = n \)
  - \( \text{sum (node(n, t1, t2)) } = n + \text{sum(t1) } + \text{sum(t2)} \)
Example: Evaluating Expressions

◆ Define datatype of expressions
  
  
  datatype exp = Var of int | Const of int | Plus of exp* exp;

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

◆ Evaluation function
  
  fun ev(Var(n)) = Var(n)
  | ev(Const(n)) = Const(n)
  | ev(Plus(e1,e2)) = ...

  Examples:
  
  \(\text{ev(Plus(Var(1),Var(2)))}\)
  \(\text{ev(Plus(Var(1),Const(3)),Var(2)))}\)

Case expression

◆ Datatype
  
  datatype exp = Var of int | Const of int | Plus of exp* exp;

◆ Case expression
  
  case e of
  Var(n) => ...
  | Const(n) => ...
  | Plus(e1,e2) => ...

Evaluation by cases

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

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) => Const(n+m)
  | Plus(e3,e4) => Plus(Plus(e3,e4),ev(e2))
  | Plus(e3,e4) => Plus(Plus(e3,e4),ev(e2))

Core ML

◆ Basic Types
  
  • Unit
  • Booleans
  • Integers
  • Strings
  • Reals
  • Tuples
  • Lists
  • Records

◆ Patterns
  
  • Unit
  • Booleans
  • Integers
  • Strings
  • Reals
  • Tuples
  • Lists
  • Records

◆ Declarations
  
  • Function declarations
  • Type 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
  
  • Basic properties
    - A variable names a storage location
    - Contents of location can be read, can be changed
  
  • ML
    - A variable 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 : int  non-assignable integer value
    - y : int ref location whose contents must be integer
    - 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:int
    - y := ly + 3 add 3 to contents of y and store in location y
ML examples

◆ Create cell and change contents
  val x = ref "Bob";
  x := "Bill";

◆ Create cell and increment
  val y = ref 0;
  y := !y + 1;

◆ While loop
  val i = ref 0;
  while !i < 10 do !i := !i + 1;
  !i;