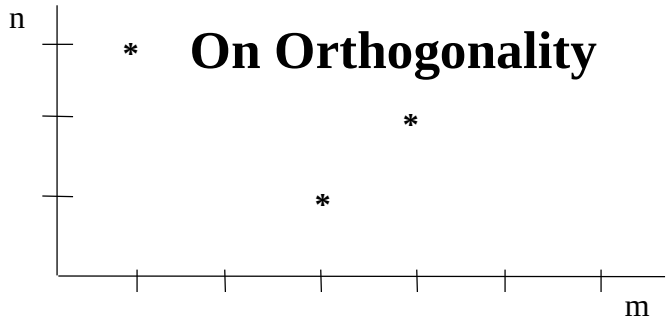


On Orthogonality

- What makes orthogonality work?
  - by remembering only m + n things, we get m * n capabilities.
- Orthogonality says that no point should be definable by more than one XY pair.
- Orthogonality advantageous only if:

  m+n + e < m*n - e

ALGOL68

ALGOL68: Goals & History

Thesis: *It is good practice in programming language design to abstain from exceptions.*

- Design goals:
  - gen purpose, rigorously-defined language
  - Clear up trouble spots in ALGOL60
    - (but, Pascal more like A60 than A68 is)
  - orthogonality, extensibility
- ALGOL68 - development started in mid-60's.
  - Revised report (SIGPLAN Notices, May 1977) cleared up many ambiguities.

Key Ideas in ALGOL68

- User type declarations (modes)
- Orthogonal design (modes, structures, ops)
- Reference mode (pointers of a sort)
- United modes (predecessor to variant records)
- Auto declaration of FOR LOOP index
- User-specified operator overloading
- Notion of "elaboration" on context entry

More Key Ideas

- Mode requirement for formals
- Casting: user-spec'd mode conversion
- Redefinition of operator precedence
- Collateral actions
- Semaphores
- W-grammars - two-level grammar
- Contexts (strong, firm, meek, weak, soft)
  - WRT coercion

ALGOL68 Structure

- ALGOL68 is block structured w/ static scope rules
  - Monolithic programming, as in ALGOL60 (and later in Pascal)
- ALGOL68's model of computation:
  - static
  - stack: block/procedure AR's; local data objects
  - heap: “heap” -- dynamic-- data objects
- ALGOL68 is an *expression*-oriented language
  - (note influence on C/C++)

ALGOL68: Organization

- Declarations:
  - Must be given (FOR LOOP index only exception)
  - Can name new types (modes)
- Imperatives (units)
  - 15 major unit types
  - Assignment is allowable side-effect of units
    - c.f. C

Data types (primitive modes)

- Int }
- Real }
- Char } primitives
- Bool }
- Void }

- Modes created from primitives --defined in "prelude"
  - String
  - Compl
  - Bits - Word full of bits

More Primitive Modes

- Bytes - Word full of chars
- Sema - Semaphore
- Format- I/O
- File - I/O

- User defined modes allowed:

  Mode largeint = long INT
  - and its attendant advantages

Non-primitive ("non-plain") modes

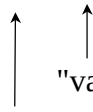
- references *
- multiples (arrays, rows)
- structures
- unions *
- procedures *

* - unusual

--can be applied to primitives or other constucted modes

References

- Variable X has two attributes of concern:
  - its value
  - reference to storage where value is kept
- Most languages don't distinguish
  - e.g. x := x + 2



"value of x"

"ref to place where value is stored"

- "The type of x is integer" and "The type of values assigned to x is integer" get combined in this case.
  - ALGOL68 made the distinction (as do e.g. C & C++).

References

- INT x -- means x is a ref to objects of type INT
- In general, a variable stands for reference to data object

so, for:

x := x + 2

"dereferenced" to yield int, so + operation is meaningful

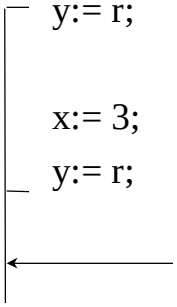
- In general, for V := E
  - type of V should be ref (type of E)
- Thus, if we declare: REF INT PNTTOX
  - mode of PNTTOX is REF REF INT and

PNTTOX:= X -- assigns X's address to PNTTOX

- action not obvious from syntax

Consider

```
INT x,y;    -- x&y are REFs to objects of  type INT
REF INT r; -- r is REF to REF INTs
x:= 2;      -- no deref necessary
r:= x;      -- ditto - pointer assignment
y:= r;      -- assigns 2 as value of y
                        --two derefs required
x:= 3;      -- no deref necessary;
y:= r;      -- assigns 3 to y.  Two derefs req'd
```



No visual clue that y's value could be affected by assignment to x.

ALGOL68 References

- Note: can't do:

```
r:= 3;  -- r is REF REF INT and 3 is INT
        -- no coercion possible
(ref int) r:= 3  -- will work.  It assigns 3 to the last
                    variable r referred to (i.e. x).
```

- Note: can create REF REF REF ... INT, etc if so inclined.

Syntactic consistency? Manifest interface?

Structuring Primitives

- ARRAYs (rows) -- 1D: ROW; 2D: ROW ROW;
- STRUCTURES
  - e.g.

    [1:12] INT MONTH -- vector of 12 integers

- On equivalence of arrays:
  - Objects of different dimensions -> different modes
  - Bounds are *not* part of the mode (c.f. Pascal)

    [1:10, 1:n] REAL time } equivalent

    [1:100, 7:11] REAL thing } modes.

More Structured Types

- Aggregate Assignment

  month:= (31,28,31,30,31,30,31,31,30,31,30,31)

  --adopted in Ada and later languages
- Dynamic arrays:

  [m:n] INT obj

  -- When encountered, array with n-m+1 locations created.

Continue Structuring Primitives

- FLEX ARRAYs -- change size on the fly.
  - e.g.

    FLEX [1:0] INT obj -- a row with no integers.

    obj:= (5,5,5) -- changes bounds to 1:3 on the fly.

    --bounds change only by assignment to whole array
- Aside on strings:

  mode string = FLEX[1:0] CHAR -- done in prelude declaration

  string greetings;
  greetings:= "Greetings and salutations"
  -- creates vector exact length of string.

Structures:

- e.g.

  mode bin_tree =
  struct(INT data,
  REF bin_tree l_child, r_child)
  ^ note recursive definition
  (illegal definition w/o REF) -- Why?
- Other standard modes built up from structs:
  - e.g.

    mode compl = struct (REAL re, im)
    mode bits = struct ([1:bits_width] BOOL x)
    mode bytes = struct ([1:bytes_width] CHAR x)
    mode sema = struct (REF INT x)
    -- all in ALGOL68 prelude

Unions

- e.g.

```
mode combine = UNION ( INT, BOOL )
  . . .
combine x -- x can take on INT or BOOL values but
          -- only under controlled conditions.
```

- assignment is OK:

```
x:= 5
x:= TRUE
```

More Unions

- Using x in an expression requires:

```
CASE x IN  -- "conformity clause"
 (INT x1):  ... <use x1>
 (BOOL x2): ... <use x2>
ESAC
```

- Note:

```
UNION (t1, t2, ..., tn) -- ti can be any mode.
   -- Only limitation: can't have ti and REF ti in same union.
   -- "incestuous union"
   -- creates ambiguity in cases like:
        UNION (INT, REF INT) x;
        INT y;
        . . .
        x:= y;  -- Can't determine how many deREFs to do on y;
                --  0: if x is ref ref int;  1: if x is ref int
```

Procedures

- Procedure units have mode and value;
  - mode determined by arg modes and ret mode.
- ALGOL68 supports procedure-valued variables:

```
mode Pr = PROC ( vector, matrix ) matrix;
          ...
Pr  P1, P2;  -- two instances of generic Pr
     ...
P1 = PROC (vector a, matrix b) matrix:
        {procedure definition}
         ...
P2 = P1 -- P2 now has same def as P1
          -- implemented using pointers
```

- Procedure modes can be used as parameters
  - (routine texts)
- Formals and actuals must have same type!

Coercion

- six kinds (see Tannenbaum):
  - dereferencing
  - deproceduring
  - widening
  - rowing
  - uniting
  - voiding

More Coercion

```
int i;  real r;  [1:1] int rowi;  ref int refi;
union(int, real) ir;  proc int p;

r:= i/r          -- i gets widened
ir:= i;          -- uniting
ir:= r;          -- uniting
i:= p;           -- deproceduring;
i:= refi;        -- dereferencing (twice)
p;               -- deproceduring; voiding
rowi:= 5;        -- rowing
```

CASE Clauses

```
CASE i IN
    <action1>,
    <action2>,
    <action3>,
    <action4>,
     . . .
ESAC
```

- Pro(s):
  - Enforced structure
    - (as compared to FTN computed goto and ALGOL60 switch)
- Cons:
  - CASE expression restricted to INT -- a bother
  - If for, say, i = 2, 4, and 6 we want to perform the same action, that action would have to be repeated in place all three times.

Continue Cons of CASE Statement

– If during program development/maintenance, an action got added or removed, programmer could miss the change, and the compiler won't complain

– very difficult kind of error to identify.

=> birth of the labeling principle (Tony Hoare came up with model Wirth included in Pascal).

- Catchall phrase (else, otherwise, etc) to catch cases not named was born later (incorporated into Ada and Modula-2)

A68 Summary...

- Coercion
  - Elaborate interactions can lead to ambiguous and difficult to read programs
  - Coercion may take place when user didn't intend it to
  - The more coercion a translator can do, the less error checking provided to the user.

==> Do you provide coercion at expense of security?

A68 Summary (cont)...

- Type compatibility
  - A68 uses structural equivalence

```
mode complex = struct (real rp; real ip);
mode weather = struct (real temp; real humid);
```

  - are equivalent
  - violates programmer's intentions

A68 Summary (cont)...

- References
  - While dangling refs are controlled in ALGOL68 they can generally only be checked at runtime.
  - Rule: in an assignment to a ref variable, the scope of the object being pointed to must be at least as large as that of the ref variable itself.
  - Dynamic data objects are reclaimed only when control leaves the scope of the associated ref variable.

A68 Summary (cont)...

- Orthogonality in general

  (real x,y; read((x,y)); if x<y then a else b fi):=

  b+ if a:=a+1; a>b then c:=c+1; +b else c:=c-1; a

  fi

  – Small set of concepts interacting in a very complex way.
  – How is simplicity best achieved?
    - Algol68: orthogonality
    - Pascal: non-rotho + "simple facilities with simple interactions."

Pascal

Pascal History

- Wirth on Design committee for ALGOL68
  - quit over differences in design philosophy
- Pascal meant to be simple, teaching language
- Target was CDC6000 family initially
  - explains functions having simple return types only (fit in one 60-bit word)
- Much of Pascal design drawn from ALGOL68 design
  - Sometimes worse! (e.g. function return types, pointer scope)

Pascal: First Impression

- A collection of irregularities
  - Files cannot be passed by value
  - Components of packed data structures cannot be passed by reference
  - Procs and funcs passed as parameters can only have by-value parameters
  - Functions can only return simple types
  - Formal param types can only be specified by a type identifier, not by its representation
  - Variables of enumerated type can only be initialized by assignment, not by input values

Discriminated & Free Union Variant Records

- Example (discriminated union):

```
Type state = (tennessee, rest);
     pi_rep = record
              case loc: state of
                 tennessee: (intpi: integer);
                 rest:      (repi: real);
              end;
```

Free union removes tag

- Assume:

```
VAR pi: pi_rep;
```

Variant Record Examples

```
CASE pi.loc of
 tennessee: pi.intpi:= 3;   --OK. compiler can
 rest: pi.repi:= 3.1415926; -- often check.
end;
```

```
pi.repi:= 3.1415926;       --error if pi.loc = tennessee

pi.repi:= 3.1415926;    -- OK if pi.loc=rest
pi.loc:= tennessee;     -- OK, but no aggregate
writeln(pi.intpi);      -- garbage
```

- w/o tags:

```
pi.repi:= 3.1415926;    -- No way to catch this
writeln(pi.intpi);      -- error, even at runtime.
```

==> verdict: variant records defeat Pascal type system.
--inconsistent with rest of language.

Name vs Structure Equivalence

- Name:
  - Types of two objects match only if they were declared using the same type name
  - Provides best protection against errors
  - Can require artificial creation of names simply to satisfy name-equiv requirements.
  - T1 = T2 does not satisfy name equiv but is often allowed.
- Structural:
  - Types of two objects match if they have the same "structure"
  - More relaxed. Prevents unnecessary creation of type names.
  - Has many logical flaws

Pascal Scope Rules...

```
Type T1 = ...
 . . .
  Procedure p1;
  Type T2 = <structure of> T1   -- ***
       T1 = . . .
```

- which T1 is ref'd at *** ?
  - (A) T2's ref to T1 is to T1 in outer level
  - (B) T2's ref to T1 is to T1 in local level
- Interpretation (B) is consistent with User Report,
- But (A) is one usually used…

Binding to Outer Level

```
Type r1 = record
           r2ptr: ^r2
          end;
     r2 = record
           r1ptr: ^r1
          end;
```

- If r2 defined in outer scope, that's what r2ptr is bound to.
- If r2 is defined in outer scope later on, meaning of program is changed!
- Wulf, Shaw: vulnerability...

C

Evaluate C

- History
- Design goals
- Contributions
- Support/violation of principles
- Interesting troublesome interactions among language features.