Types-2

- Type equivalence
  - Structural
  - Name
- Algol68
  - Arrays
    - Array addressing
  - User-defined modes
  - Discriminated unions
  - Dereferencing as type conversion

Type Equivalence

Structural equivalence

- SE1: a type name is structurally equivalent to itself
- SE2: two types are structurally equivalent if they are formed by applying the same type constructor to two structurally equivalent types
- SE3: after type declaration `type n=T` the type name `n` is structurally equivalent to `T`

Structural Equivalence

- **Implies "same shape" type in a user-defined type**

```
type S = array [0..99] of char
type T = array [0..99] of char

typedef struct{
 int j, int k, int *ptr}cell;
typedef struct{
 int n, int m, int *p}element;
```


Other Def'ns of Equivalence

```
T: type array [1..20] of int;
x,y: array [1..20] of int;
w,z : T;
v: T;
```

x,y,w,z,v are all the same type by structural equivalence

Name equivalence:

Types are equivalent which have the same name or are formed by the same type expression.

`x` and `y` are of same type, `w, z, v` are of same type, but `x` and `w` are of diferent types!

Fine Points of Type Equiv.

- **C/C++ use structural equivalence for all types except structs**

```
type A = record
   x,y : real;
end;
type B = record
  z,w : real
end;
```

In C and Algol68, A and B not equivalent types because the fieldname is part of the type.

Algol68

- **Explicitly typed, statically checkable PL**
- **Block structured, extensible type system**
- ***Objects* - entities stored in memory during execution of a program**
  - ***Mode of an object* - analogous to type**
    - **1 is of *mode int***
- ***Variable* - considered an L-value declared using the type of objects that can be assigned to it**
  - ***Mode of a variable* - int k means that k is of *mode ref int* and k can store int values.**

A.Tanenbaum, Tutorial on Algol68, ACM Computing Surveys, June 1976, Vol 8 No 2 in SERC Reading Room

Algol68

- **Examples**
  - **Integer constant, *mode int* (1)**
  - **Integer-valued variable, *mode ref int* (j)**
  - **Well-typed assignment statement *lhs = rhs***
    - **where *lhs* is convertible to an object of *mode ref int* and *rhs* is convertible to an object of *mode int***

    `int j,k,l;` -- **means all variables on list are *mode ref int***

    `j := 2;` -- `j` **can store *mode int* objects**

    `k := j+l;` -- **+ operator on *mode int* objects**

    `k := j;` --**dereference** `j` **to obtain 2 of *mode int***
- **Strings are primitive**

Algol68 - Arrays

- **Vector (array) - one dimensional sequence of objects all having same mode**

  `[ ] int` ***row of int***

  `[ , ] real` ***row row of real***
- **Array type only includes dimensionality, not bounds**

  `[1:12] int month;[1:7] int day;` ***mode row int***

  `[0:10,0:10]real matrix;`

  `[-4:10,6:9]real table` ***mode row row int***

  Note `table` **and** `matrix` **are type equivalent!**

Algol68 - Arrays

[1:10] [1:5,1:5] int kinglear;

/kinglear is a vector of 10 elements each of which is a *row row int* array of 25 elements.

kinglear[j] is legal wherever *[,]int* mode is legal

kinglear[j][1,2] is legal wherever *int* mode is legal

kinglear[1,2,3] is ILLEGAL!

Array Operations

- ***Trimming:* yields some cross section of an original Algol68 array (slicing an array into subarrays)**
- ***Subscripting:* limiting 1 dimension to a single index value**

[1:10]int a,b; [1:20] real x; [1:20,1:20] real xx;

b[1:4] := a[1:4] -- assigns 4 elements

b:= a -- assigns all of a to b, same as b[1:10]:=a[1:10]

xx[4,1:20]:=x --assigns 20 elements to row 4 of xx

xx[8:9,7] := x[1:2] --assigns x[1] to xx[8,7] and x[2] to xx[9,7]

Array Addressing

- **X[low:high] of E bytes each data item. What's the address of X[j]?**

  addr(X) + (j-low) *E <= addr(X) +(high-low)*E

  - **Note: addr(X)-low*E is a compile-time constant**
  - **X[ ] row real (4 bytes each);**
  - **X[3] is addr(X[0]) + (3-0)*4 = addr(X) + 12**
  - **X[0], X[1] is at address X[0]+4, X[2] is at address X[0]+8, etc**

Array Addressing

- **Assume arrays are stored in row major order y[0,0], y[0,1], y[0,2], …, y[1,*], y[2,*],…**
- **Consider memory a sequence of locations**
- **Then if have y[low1:hi1,low2:hi2] in Algol68, location y[j,k] is**

  addr(y[low1,low2]) + (hi2-low2+1)*E*(j-low1)
  +(k-low2)*E
  #locs per row
  #rows in front of row j
  # elements in row j in front of element [j,k]

Example

y[0:2, 0:5] in Algol68, an int array. Assume row major storage and find address of y[1,3].

address of y[1,3] = addr(y[0,0])+(5-0+1)*4*(1-0)+(3-0)*4

6 elements per row

1 row before row 1

3 elements in row 1 before 3

= addr(y[0,0])+24+12

= addr(y[0,0])+36

- **Analogous formula holds for column major order.**

Algol68 - User-defined Modes

- **Similar to typedefs in C, but without pointers**

```
mode vector = [1:n] real;
mode person = struct (string initials, int age, bool
  married);

mode family = struct(person mom,dad; [1:2]person
  child); //last is a row of person structs, child[1]
  and child[2] are two different person structs.

mode tree = struct(int value, ref tree right, left);
//recursive data structure is well-defined; couldn't
  put tree field in tree mode, but ref tree field is
  okay.
```

Example

```
person tom := (“tj”, 40, true);
person mary := (“mah”, 37, true);
family jones := (mary,tom,(skip,skip));
initials of mom of jones := “mhj”;
if (age of mom of jones = age of data of jones)….
```

- Can define new operators on user-defined modes
- Can initialize such variables
- Can use component selection
- Equality comparisons use structural equivalence which includes the fieldnames

Type Unions

- **Idea: allow a variable to contain values of different types during execution**
  - **Pascal: variant records; C, Algol68: unions**
- **Problem: with unions, how can we assure type-safe programs?**
  - **Pascal and C are not safe.**
  - **Algol68 is type-safe!! Uses *discriminated unions***
- **Usually all versions of union use same storage**

Example

- **C:**

```
union{double f;int j} fi;
…fi.f =3.14159;…printf("%d\n", fi.j); ….
```

  - **Means that `fi` sometimes contains int values and sometimes double values**
  - **Can check all uses of union variables by runtime check of current type tag, as in PLs with implicit typing**
- **Pascal: type tag tells which type value the variant contains but checking it is *optional!***

Discriminated Unions (Algol68)

```
union(int, real,bool) kitchensink;
kitchensink := 3;
kitchensink := 3.14;
kitchensink := true;
if random <.5 then kitchensink := 1
              else kitchensink := 2.78
fi;
case kitchensink in
(int j): print (("integer",j));
(real r): print (("real",r));
(bool b): print (("bool",b));
esac;
```

Problems with Unions

- What is meaning of assignment to tag field without assignment to variant fields?
  - Ada: must change both value and tag together
- If tag not kept in record itself (Pascal), how can its value be checked?
- Should tag fields be required to be initialized?
- Component selection has to be runtime checked

Why use Unions?

- Simulate a logical shift on sequence of bits using a multiply or divide by 2 on an integer
- To allow structs with an initial portion that has essential information, followed by properties that vary per person represented (e.g., has_children, #children, married, spouse_name)

Ref Mode

- **Algol68 integer variables are of *mode ref int (reference to integer)***
- **Assignment statement**

  ***ref int mode* object *:= mode int* object**
  - **Either side of assignment may need dereferencing to obtain right mode. (However, implicit dereferencing of the rhs is not performed in Algol68)**
  - **Algol68 does allow more than one automatic dereference on rhs, whereas Pascal and C each allow only 1.**
  - **`int j,k; j:=k;` dereference `k` to get *mode int* object**

Legal Assignments

- **Let level be #refs in mode of a variable**
  - **declarations: `int j; ref int p;`**
    - **j is *ref int mode*, p is *ref ref int mode***
    - **level(3) = 0; level(j) = 1; level(p) = 2**
- **Consider an assignment lhs := rhs**

  – *level(lhs) <= level(rhs)+1* for a legal assignment

  `int j,k; ref int q; j:=k;k:=q;q:=j;q:=1;`

  --level(j)=level(k)=1, so level(j)=1 <= 1+1 *okay*

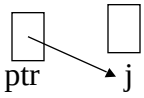
  --level(k)=1,level(q)=2, so level(k) <=2+1 *okay*

  --level(q)=2,level(j)=1, so level(q)=2 <= 1 + 1 *okay*

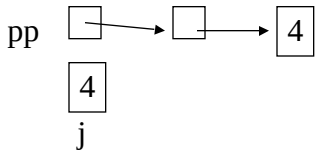
  --level(q)=2,level(1)=0, so is 2<= 0+1? *NO*

Example

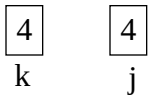
```
int j,k;
ref int ptr, sptr;
ref ref int pp;
ptr := j;     --level(ptr)=2,level(j)=1, 2=1+1 no deref
```




```
j := pp;      --level(j)=1,level(pp)=3, 1<3+1=4, dereference necessary
```

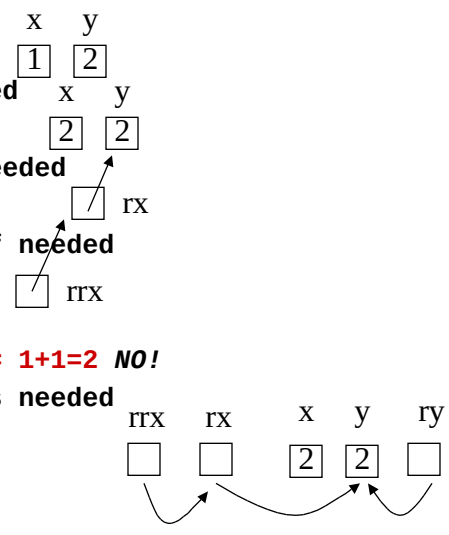



```
k := j;       --level(j)=1, level(k)=1, 1< 1+1 dereference necessary
```



Example

```
int x,y; ref int rx,ry;
ref ref int rrx,rry;
x:=1; y:=2;
x:= y; --1 deref needed
1   1
rx := y; --no deref needed
2      1
rrx := rx;  --no deref needed
3       2
rry := y; --illegal!
 3     1          3 <= 1+1=2 NO!
ry := rrx;  --2 derefs needed
 2      3
```



Ref Mode

- **What would happen in a PL which did no automatic dereferencing on the rhs of assignments?**
  - **You would need to explicitly encode all dereferences**
- **Can you think of an assignment that is legal in Algol68 but not in C?**
  - **C *int* * pointers correspond to *ref ref int* variables**
  - **& operator prevents automatic dereferencing on rhs of assignment**

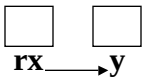
Allowing LHS Dereferences

- **In C, `int *rx;int y;*rx =y;` is legal; Can we achieve this in Algol68 without an explicit dereference operator?**

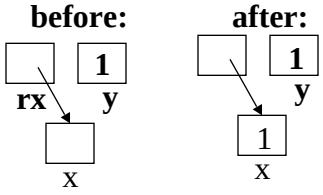
  `ref int rx; int y`

  `rx := y` **might have 2 interpretations**

  1. no dereferencing on lhs



  2. deref both lhs, rhs once



  Which is meant? Can't tell.

  Need to use an explicit cast

  `ref int (rx):=y` **forces `rx`, a *mode ref ref int* variable, to be dereferenced once to become a *mode ref int* variable**

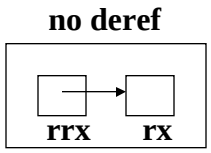
Allowing LHS Dereferences

- **Another example**

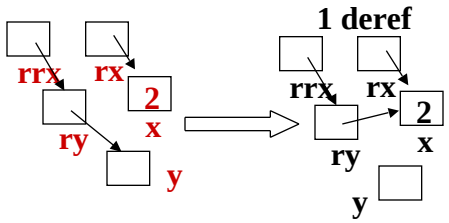
`rrx := rx`

- - 3 interpretations possible

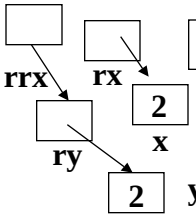
1. no dereferences



2. 1 deref of rhs and lhs



3. 2 derefs of rhs and lhs



2 derefs

ref ref int (rrx) := rx

ref int (rrx) := rx