CSc 453

Compilers and Systems Software

11 : Semantic Analysis III

Department of Computer Science
University of Arizona

collberg@gmail.com

Basic and Structured Types

Basic Types

- Which basic types does the language have? In Pascal `boolean`, `real`, `integer`, `char` are basic types.
- Integers: may come in different sizes and signed/unsigned.
- Reals: may come in different sizes. Some languages allow programmer control over precision.
- Some languages have fix-point numbers, complex numbers, rational numbers,. . . .
- Does the language automatically convert from one type to another? Can I add a complex number and an integer?

Basic Types...

Enumeration types

```
TYPE E1 = (white,blue,yellow,green,red);
TYPE E2 = (apple=4,pear=9,kumquat=99);
```

- Pascal, Ada, Modula-2, C have some variant of enumeration types.

Subrange types

```
TYPE S1 = [0..10];
TYPE S2 = ['a'..'z'];
TYPE S3 = [blue..green];
```

- Subranges can be used to force additional runtime checks. Some languages use them as array index types.

Structured Types: Arrays

- Are they static or dynamic? I.e. do I create them at compile-time (C) or run-time (Java)?
- Do I check for out-of-bounds errors (Java) or not (C)?
- Are they 0-based (C) or 1-based (Icon)?
- Can the user define both the lower and upper bounds (Pascal)?
- Must the index type be integer (C,Java) or any enumerable type (Pascal)?

Structured Types: Arrays. . .

```
TYPE A1 = ARRAY 100 OF CHAR;
TYPE A2 = ARRAY [5..99] OF INTEGER;
TYPE A3 = ARRAY CHAR OF INTEGER;
TYPE A4 = ARRAY OF INTEGER;
VAR a3 :  A3;
VAR a4 :  A4;
BEGIN
   a3['X'] := 55;
   a4 := NEW ARRAY 99 OF INTEGER;
END
```

- Most languages lay out arrays in row-major order. FORTRAN uses column-major.

A[1,1]	A[1,2]
A[2,1]	A[2,2]
A[3,1]	A[3,2]
A[4,1]	A[4,2]

Matrix

0	A[1,1]
1	A[1,2]
2	A[2,1]
3	A[2,2]
4	A[3,1]
5	A[3,2]
6	A[4,1]
7	A[4,2]

Row Major

0	A[1,1]
1	A[2,1]
2	A[3,1]
3	A[4,1]
4	A[1,2]
5	A[2,2]
6	A[3,2]
7	A[4,2]

Column Major

Array Indexing – 1 Dimensions

- How do we compute the address (*L*-value) of the *n*:th element of a 1-dimensional array?
- $\mathtt{A}_{\mathtt{elsz}}$ is *A*'s element-size, $\mathtt{A}_{\mathtt{addr}}$ is its base address.

VAR `A` : **ARRAY** `[l .. h]` **OF** `T;`

$$
\begin{aligned}
\mathtt{L\text{-}VAL}(A[i]) &\equiv \mathtt{A}_{\mathtt{addr}} + (i - l) * \mathtt{A}_{\mathtt{elsz}} \\
&\equiv \mathtt{A}_{\mathtt{addr}} + (l * \mathtt{A}_{\mathtt{elsz}}) + i * \mathtt{A}_{\mathtt{elsz}} \\
C &\equiv \mathtt{A}_{\mathtt{addr}} + (l * \mathtt{A}_{\mathtt{elsz}}) \\
\mathtt{L\text{-}VAL}(A[i]) &\equiv C + i * \mathtt{A}_{\mathtt{elsz}}
\end{aligned}
$$

- Note that C can be computed at compile-time.

Array Indexing – 2 Dimensions

VAR `A` :**ARRAY** $[l_1 \,..\, h_1][l_2 \,..\, h_2]$ **OF** `T`;

$$
\begin{array}{rcl}
w_1 & \equiv & h_1 - l_1 + 1 \\
w_2 & \equiv & h_2 - l_2 + 1 \\
\mathtt{L\text{-}VAL}(A[i_1, i_2]) & \equiv & \mathtt{A}_{\mathtt{addr}} + ((i_1 - l_1) * w_2 + i_2 + l_2) * \mathtt{A}_{\mathtt{elsz}} \\
 & \equiv & \mathtt{A}_{\mathtt{addr}} + (i_1 * w_2 + i_2) * \mathtt{A}_{\mathtt{elsz}} - \\
 & & (l_1 * w_2 - l_2) * \mathtt{A}_{\mathtt{elsz}} \\
C & \equiv & \mathtt{A}_{\mathtt{addr}} - (l_1 * w_2 - l_2) * \mathtt{A}_{\mathtt{elsz}} \\
\mathtt{L\text{-}VAL}(A[i_1, i_2]) & \equiv & (i_1 * w_2 + i_2) * \mathtt{A}_{\mathtt{elsz}} + C
\end{array}
$$

- C can be computed at compile-time.

Array Indexing – n Dimensions

VAR `A` : **ARRAY** $[l_1 \,..\, h_1]$ ... $[l_n \,..\, h_n]$ **OF** `T;`

$$w_k \equiv h_k - l_k + 1$$

$$C \equiv \mathtt{A}_{\mathtt{addr}} - ((\cdots(l_1 * w_2 + l_2) * w_3 + l_3)\cdots) * w_n + l_n) * \mathtt{A}_{\mathtt{elsz}}$$

$$\mathtt{L\text{-}VAL}(A[i_1, i_2, ..., i_n]) \equiv ((\cdots(i_1 * w_2 + i_2) * w_3 + i_3)\cdots) * w_n + i_n) * \mathtt{A}_{\mathtt{elsz}} + C$$

Record Types

- Pascal, C, Modula-2, Ada and other languages have variant records (C's union type):

```
TYPE R1 = RECORD tag :  (red,blue,green);
             CASE tag OF
                 red :  r :  REAL; |
                 blue :  i :  INTEGER; |
                 ELSE c :  CHAR;
             END;
          END;
```

  Depending on the tag value `R1` has a real, integer, or char field.

- The size of a variant part is the max of the sizes of its constituent fields.

Record Types...

- Oberon has extensible record types:

```
TYPE R3 = RECORD
             a :  INTEGER;
          END;
TYPE R4 = (R3) RECORD
             b :  REAL;
          END;
```

  R4 has both the a and the b field.
- Extensible records are similar to classes in other languages.

Pointer Types

- In order to build recursive structures, most languages allow some way of declaring recursive types. These are necessary in order to construct linked structures such as lists and trees:

```
TYPE P = POINTER TO R;
TYPE R = RECORD
         data :  INTEGER;
         next :  P;
      END;
```

- Note that `P` is declared before its use. Languages such as Pascal and C don't allow forward declarations, but make an exception for pointers.

Procedure Types

- C, Modula-2, and other languages support procedure types. You can treat the address of a procedure like any other object:

```
TYPE P = PROCEDURE(x:INTEGER; VAR Y:CHAR):REAL;
VAR z : P; VAR c : CHAR; VAR r : REAL;
PROCEDURE M (x:INTEGER; VAR Y:CHAR):REAL; BEGIN···END;
BEGIN
   z := M; /* z holds the address of M. */
   r := z(44,c);
END.
```

- Languages differen in whether they allow procedures whose address is taken to be nested or not. (Why?)

Class Types

- Java's classes are just record types. Some languages (Object Pascal, Oberon, MODULA-3) define classes just like records:

```
TYPE C1 = CLASS
            x :  INTEGER;
            void M() { ··· };
            void N() { ··· };
          END;
TYPE C2 = CLASS EXTENDS C1
            r :  REAL; // Add another field.
            void M() { ··· }; // Overrides C1.M
            void Q() { ··· }; // Add another method.
          END;
```

Type Constructors

Type Expressions (TE)

- To reason about types we build up an algebra of TEs:

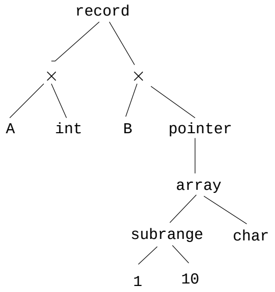
$$
\begin{aligned}
TE &= \texttt{int}, \texttt{string}, \texttt{real}, \cdots, \texttt{type\_error}, \texttt{void} \\
&= \texttt{subrange}(\texttt{from}, \texttt{to}) \\
&= \texttt{array}(\texttt{idx}, \texttt{eltype}) \\
&= \texttt{record}((f_1 \times t_1) \times \cdots \times (f_n \times t_n)) \\
&= \texttt{pointer}(\texttt{type}) \\
&= d_1 \times \cdots \times d_n \rightarrow r
\end{aligned}
$$

- The f_i:s are field names and t_i:s are field types (TEs).
- $d_1 \times \cdots \times d_n$ is the domain and r is the range of a function type. d_i and r are TEs.

```
TYPE T = RECORD
          A : INTEGER;
          B : POINTER TO ARRAY [1..10] OF CHAR;
     END
```

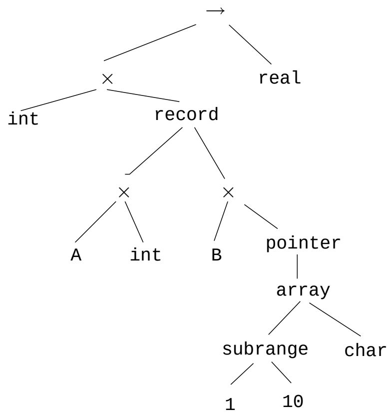
```
record(
   A × int,
   B × pointer(
      array(subrange(1,10),char))))
```

Type Expression



Type Graph

TYPE `P = ` **PROCEDURE** `(A:`**INTEGER**`; B:T) : REAL;`



$\texttt{int} \times \texttt{T} \longrightarrow \texttt{real}$

Typechecking

Static/Dynamic Typechecking

- Semantic checking can be done both at compile-time (static checking) and run-time (dynamic checking).
- Some translators also do some checking at link-time and load-time. Java, for example, verifies the correctness of class-files at class load time.
- A language has a Sound Type System if no dynamic typechecking necessary.
- In a Strongly Typed Language there are no type errors at run time.

Static/Dynamic Typechecking...

```
VAR V : REAL;
VAR S = [1 .. 10];
BEGIN
   V := V + 3.14; // Static check
   S := READ; // Dynamic check
END
```

Type Equivalence

Equivalence Types

- Equivalence types are used to create type aliases:

```
TYPE Flag = (red,white,blue);
TYPE Q = Flag;
VAR x :  Flag;
VAR y :  Q;
BEGIN
x := y; /* Legal?  */ END;
```

- But, when are two types equivalent? I.e. when can we compare two variables of "different" types?
- Some languages use structural type equivalence, others name equivalence, others a mixture of the two.

Structural Equivalence

```
PROCEDURE Equiv(s, t) : BOOLEAN
    IF basic(s) & basic(t) & s = t THEN
        RETURN TRUE
    ELSIF s = array(i1, t1) & t = array(i2, t2) THEN
        RETURN Equiv(i1, i2) & Equiv(t1, t2)
    ELSIF s = l1 × r1 & t = l2 × r2 THEN
        RETURN Equiv(l1, l2) & Equiv(r1, r2)
    ELSIF s = pointer(p1) & t = pointer(p2) THEN
        RETURN Equiv(p1, p2)
    ELSIF s = d1 → r1 & t = d2 → r2 THEN
        RETURN Equiv(d1, d2) & Equiv(r1, r2)
    ELSE RETURN FALSE
END
```

Structural Equivalence...

```
class Square {void move(){···}; void draw(){···};}
class Cowboy {void move(){···}; void draw(){···};}
void main(){Square s=new Square(); Cowboy c=s;} // Legal?
```

- Structural type equivalence will sometimes get us in trouble.
- Structural type equivalence make sense in distributed systems — what type does an object have after I have packed it into a bit-string and sent it over the net to another process?
- In MODULA-3 (which uses structural type equivalence) you can tag a type with a unique string to make sure it's not equivalent to other types by chance.

Modula-2 Type Equivalence

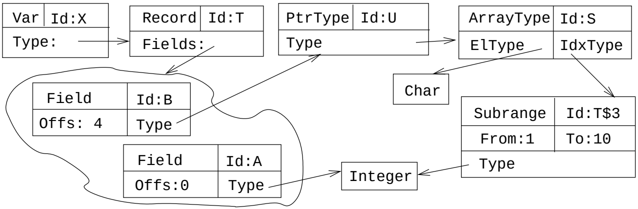
```
PROCEDURE Equiv(s, t) :  BOOLEAN
    IF s = t THEN
        RETURN TRUE
    ELSIF s = subrange(t1, l1, h1) &
              t = subrange(t2, l2, h2) THEN
        RETURN Equiv(t1, t2)
    ELSIF s = l1 × r1 & t = l2 × r2 THEN
        RETURN Equiv(l1, l2) & Equiv(r1, r2)
    ELSIF s = d1 → r1 & t = d2 → r2 THEN
        RETURN Equiv(d1, d2) & Equiv(r1, r2)
    ELSE
        RETURN FALSE
END
```

Typechecking Designators

Semantic analysis of Structured Types

- Declarations of structured types (arrays, records, pointers) become a **type graph** of type dependencies in the symbol table:

TYPE S = **ARRAY** [1..10] **OF CHAR**;
U = **POINTER TO** S;
T = **RECORD** A:**INTEGER**1 B:U; **END**



Typechecking Designators

- A designator is any part of an expression that references a memory location.
  1. Simplest case: `X`.
  2. Structured types complicate things: `X^^.V[5][7]^.P`.
- Designators are typechecked using the symbol table type graph.
- An attribute $\Updownarrow$`Type:`**TypeT**`.TypeIn` stores the type of partially processed designator.
- A synthesized attribute `TypeOut` returns the type of the complete designator.

Typechecking Designators. . .

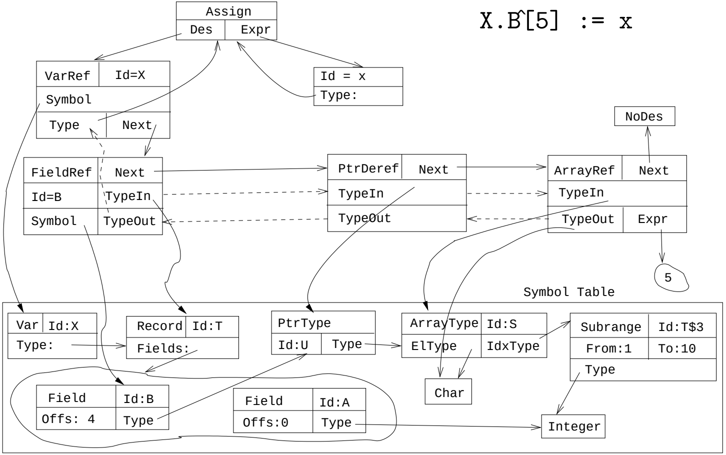
```
TYPE    S = ARRAY [1..10] OF CHAR;
TYPE    U = POINTER TO S;
TYPE    T = RECORD A:INTEGER B:U; END

PROCEDURE P (VAR X : T); ···
VAR X : T; C : CHAR;
BEGIN
   P(X.B^[5]);      (* L-Value *)
   X.B^[5] := "x";(* L-Value *)
   C := X.B^[5];   (* R-Value *)
END
```

```
PROCEDURE Des (n :  Node);
  IF n.Kind = VarRef THEN
    Symbol := Lookup(n.Id,n.Env);
    n.Next.TypeIn := GetType(Symbol);
    Des(n.Next); n.Type:=n.Next.TypeOut;
  ELSIF n.Kind = FieldRef THEN
    IF TypeKind(n.TypeIn) ≠ Record THEN
      PRINT "Record Type Expected"
    ENDIF;
    Symbol := FindField(n.Id,n.TypeIn);
    n.Next.TypeOut := FieldType(Symbol);
    Des(n.Next);
    n.TypeOut:=n.Next.TypeOut;
            .........
```

```
ELSIF n.Kind = ArrayRef THEN
  IF TypeKind(n.TypeIn) ≠ Array THEN
    PRINT "Array Type Expected"
  ENDIF;
  Expr(n.Expr);
  IdxType := ArrayIndexType(n.TypeIn);
  IF n.Expr.Type ≠ IdxType THEN
    PRINT "Wrong Index Type"
  ENDIF;
  n.Next.TypeIn:=ArrayType(n.TypeIn);
  Des(n.Next);
  n.TypeOut:=n.Next.TypeOut;
          .........
```

```
ELSIF n.Kind = PointerRef THEN
   IF TypeKind(n.TypeIn) ≠ Pointer THEN
      PRINT "Pointer Type Expected"
   ENDIF;
   n.Next.TypeIn := PtrType(n.TypeIn);
   Des(n.Next);
   n.TypeOut:=n.Next.TypeOut;
ELSIF n.Kind = NoDes THEN
   n.TypeOut := n.TypeIn;
END;
```

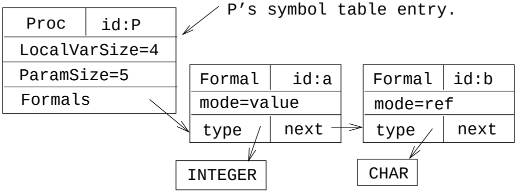

X.B^[5] := x
Assign
Des
Expr
VarRef
Id=X
Symbol
Type
Next
Id = x
Type:
NoDes
FieldRef
Next
Id=B
TypeIn
Symbol
TypeOut
PtrDeref
Next
TypeIn
TypeOut
ArrayRef
Next
TypeIn
TypeOut
Expr
5
Symbol Table
Var
Id:X
Type:
Record
Id:T
Fields:
PtrType
Id:U
Type
ArrayType
Id:S
ElType
IdxType
Subrange
Id:T$3
From:1
To:10
Type
Field
Id:B
Offs: 4
Type
Field
Id:A
Offs:0
Type
Char
Integer

Typechecking Procedure Calls

Typechecking Procedure Calls

- To typecheck procedure calls we first have to build an appropriate symbol table structure.
- This is simply a linked list of the procedure's formal parameters. For each parameter we give its **name**, **type** and **mode** (value or reference (`VAR` in Pascal)).

PROCEDURE `P (a:`**INTEGER**`;` **VAR** `b:`**CHAR**`);`
VAR `c:`**INTEGER**`;` **BEGIN** $\cdots$ **END** `P;`



Procedure Calls...

- Checking a procedure call becomes very simple: just traverse the list of actual parameters and the list of formal parameters in parallel, checking one type at a time.
- Obviously, we have to check that the lists are of the same length.
- We give each `Actual`-node an inherited attribute $\Downarrow$`Formal`:**FormalT** that points to the current formal parameter in the symbol table.

Example Procedure Call:

```
VAR x : INTEGER;
VAR y : INTEGER;
BEGIN P(5+x, y) END
```

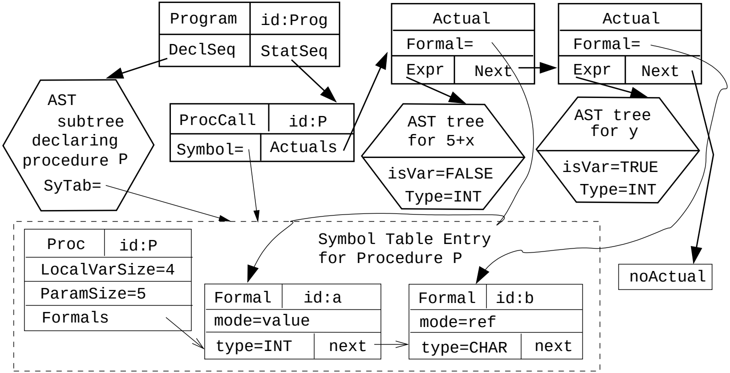
1. Look up the name of the procedure in the current environment.
2. Get a pointer to the first formal node in the symbol table. Start checking the actuals.

```
PROCEDURE Statement (n:Node);
   IF n.Kind = ProcCall THEN
      IF Member(n.Name, n.Env) THEN
         Symbol := Lookup(n.Name, n.Env);
         n.Actuals.Formal := GetFormals(Symbol);
         n.Actuals.Env := n.Env;
         CheckCall(n.Actuals);
      ELSE
         PRINT "Procedure not declared" ENDIF;
```

- The attribute $\Uparrow E$.IsVar:**BOOL** is TRUE if expression E is an L-Value.

```
PROCEDURE CheckCall(n:Node);
   IF n.Kind = Actual THEN
      n.Expr.Env := n.Env; Expr(n.Expr);
      IF n.Expr.Type ≠ n.Formal.type THEN
         PRINT "Wrong Parameter Type"
      ENDIF;
      IF n.Formal.mode = ref AND n.Expr.IsVar = FALSE THEN
         PRINT "Variable expected"
      ENDIF;
      n.Next.Formal := GetNextFormal(n.Formal);
      n.Next.Env := n.Env;
      CheckCall(n.Next);
```

Procedure Calls. . .



Homework

Homework I

- Build an AST for the program below. Show – in detail – how the assignment statements are checked for type correctness.

```
PROGRAM M;
   TYPE A = RECORD
              X : ARRAY [1..10] OF INTEGER;
           END;
           B = POINTER TO A;
           C = ARRAY [1..2] OF B;
   VAR V : C;
BEGIN
   V[1]^.X[4] := "C";
   V[2].X[4] := 5;
END.
```

Summary

Readings and References

- Read Louden: pp. 313–331.
- Or, read the Dragon book: 343–360.