CS536
Type Translation and Flow Control

A Sahu

CSE, IIT Guwahati

Outline

- Type and Declaration
- Type expression and equivalence
- Control flow

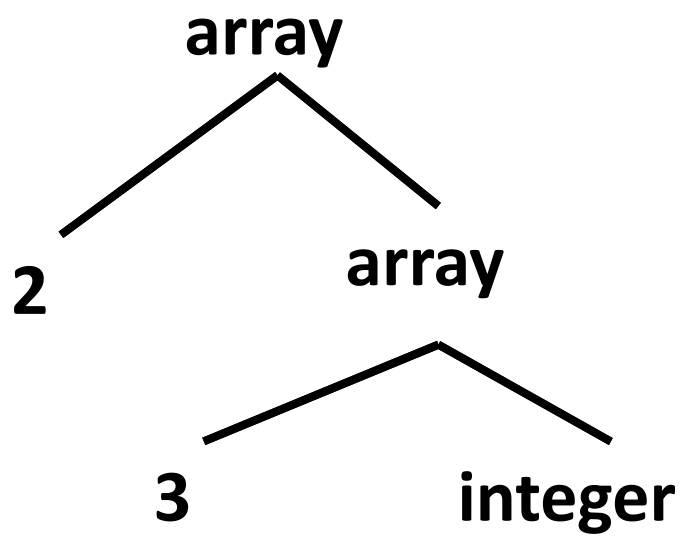
Types and Declaration

- Type Checking
  - Uses logical rules to reason about the behavior of a program at run time
  - Specifically, it ensure the types of operand matches the type expected by an operator
- Translation Application
  - From type of a name, compiler determine the storage needed for the name.
  - Needed for calculate the address denoted by an array reference : **to insert explicit type conversion** and choose the **right version of arithmetic operator**

Type Expressions

Example: int[2][3]

array(2,array(3,integer))



Type Expressions

- A basic type is a type expression
  - Typical basic types: bool, char, int, float, void, etc
- A type name is a type expression
- A type expression can be formed by applying the array type constructor to a number and a type expression.
- A record is a data structure with named field
  - A type expression can be formed by applying types in the record type constructor to the filed names and their types

Type Expressions

- A type expression can be formed by using the type constructor → for function types
  - s→t is function from type s to type t
  - ***See Meta Lang. (ML) function online***
- If s and t are type expressions, then their Cartesian product s*t is a type expression
  - Product are introduced for completeness
  - They can be represent as list / tuple of types; e.g. function parameters
- Type expressions may contain variables whose values are type expressions

Type Equivalence

When two type expr are equivalent?

When type expressions are represented by graph, two types are structurally equivalent iff

- They are the same basic type.
- They are formed by applying the same constructor to structurally equivalent types.
- One is a type name that denotes the other.

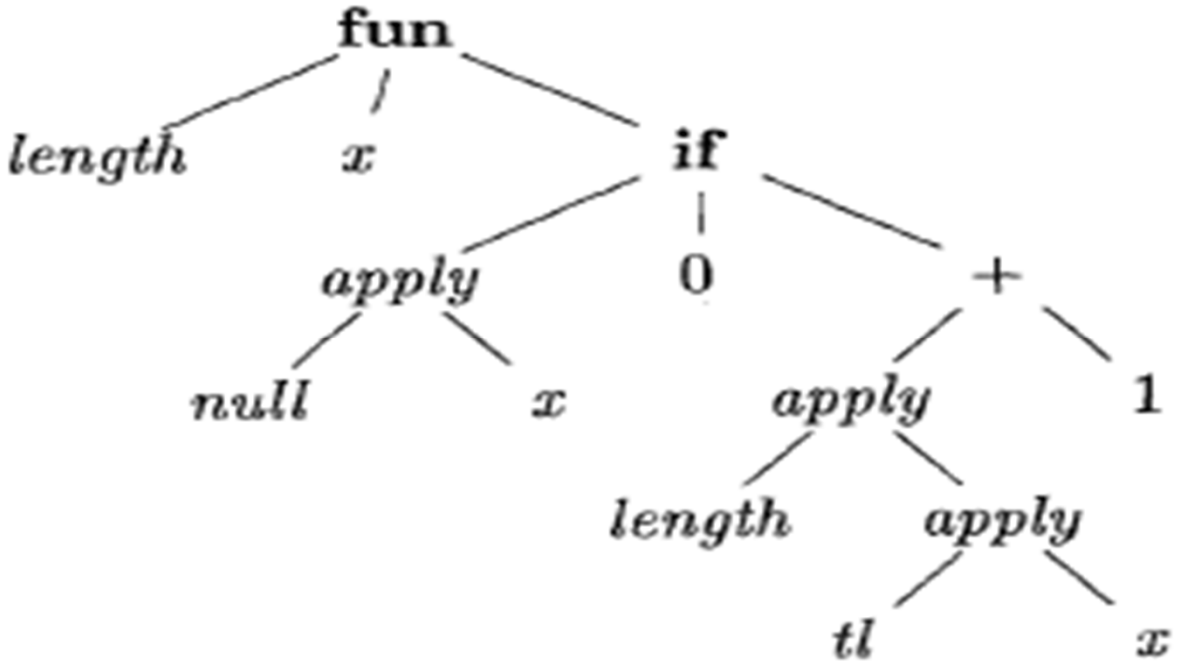
Unification algorithm : will see later

Rule for type Checking

- If f has a type s→t then x has type s and the expression f(x) has a type t
- If f(x) is an expression then
  - For some α and β, f has a type α → β and x has a type α

Abstract syntax tree for the function definition

fun length(x) =
if null(x) then 0 else length(tl(x)+1)



This is a polymorphic function
in ML language

Abstract syntax tree for the function definition

```
fun length(x) =
   if null(x) then 0 else length(tl(x)+1)
```

- This is a polymorphic function in ML language
- Example
  - length([‘a’,’b’])+length([1,2,8])
- Universal qualifier
  - Forall α. list(α)→ integer
  - Forall β. list(β)→integer

Unification of type: function

A : s -> t B : t -> u

AB : s -> u

A : s -> t B : s -> t

A+B : s -> t

A : s -> s

A^2: s -> s

Unification of type: function

Now given some expression, $(AB + CD)^2$,

A : s -> t B : u -> v AB : a -> b

C : w -> x D : y -> z CD : c -> d

AB + CD : e -> f $(AB + CD)^2$: g -> h

==========

For AB: t = u, a = s, b = v **for CD:** x = y, c = w, d = z

for AB + CD : a = c = e, b = d = f

for $(AB + CD)^2$: e = f = g = h

Equivalence classes:

(a = b = c = d = e = f = g = h = s = v = w = z),

(t = u), (x = y)

Inferring a type for the function *length*

fun length(x) = if null(x) then 0 else length(tl(x)+1)

Line	Expression : type	Unify
1	length :β → γ	
2	x :β	
3	if :bool x α_i x α_i = α_i	
4	null :list(α_n) → bool	list(α_n)=β
5	null(x) : bool	
6	0 :int	α_i =int
7	+ : int x int → int	
8	tl : list(α_t) → list(α_t)	
9	tl(x) : list(α_t)	list(α_t) =list(α_n)
10	length(tl(x): γ	γ=int
11	1 : int	
12	length((tl(x))+1: int	
13	If(…) :int	

Suppose we know null take list and give a Boolean and tl take list and produce list

Final infer: Forall α. list(α)→ integer

Unification algorithm

```
bool unify (Node m, Node n) {
        s = find(m); t = find(n);
        if ( s = t ) return true;
    else if ( nodes s and t represent the same basic type )
            return true;
    else if (s is an op-node with children s1 and s2 and
        t is an op-node with children t1 and t2) {
        union(s , t) ;
        return unify(s1, t1) and unify(s2, t2);
       }
    else if s or t represents a variable {
        union(s, t) ; return true;
    }
    else return false;
}
```

Declarations

- Grammar for simplified declaration : one name at a time

  D→T id; D | ϵ

  T→ B C | record ‘{‘ D ‘}’

  B→ int | float

  C→ ϵ | [num] C

- C is component types: generate string of zero or more integer surrounded by bracket []

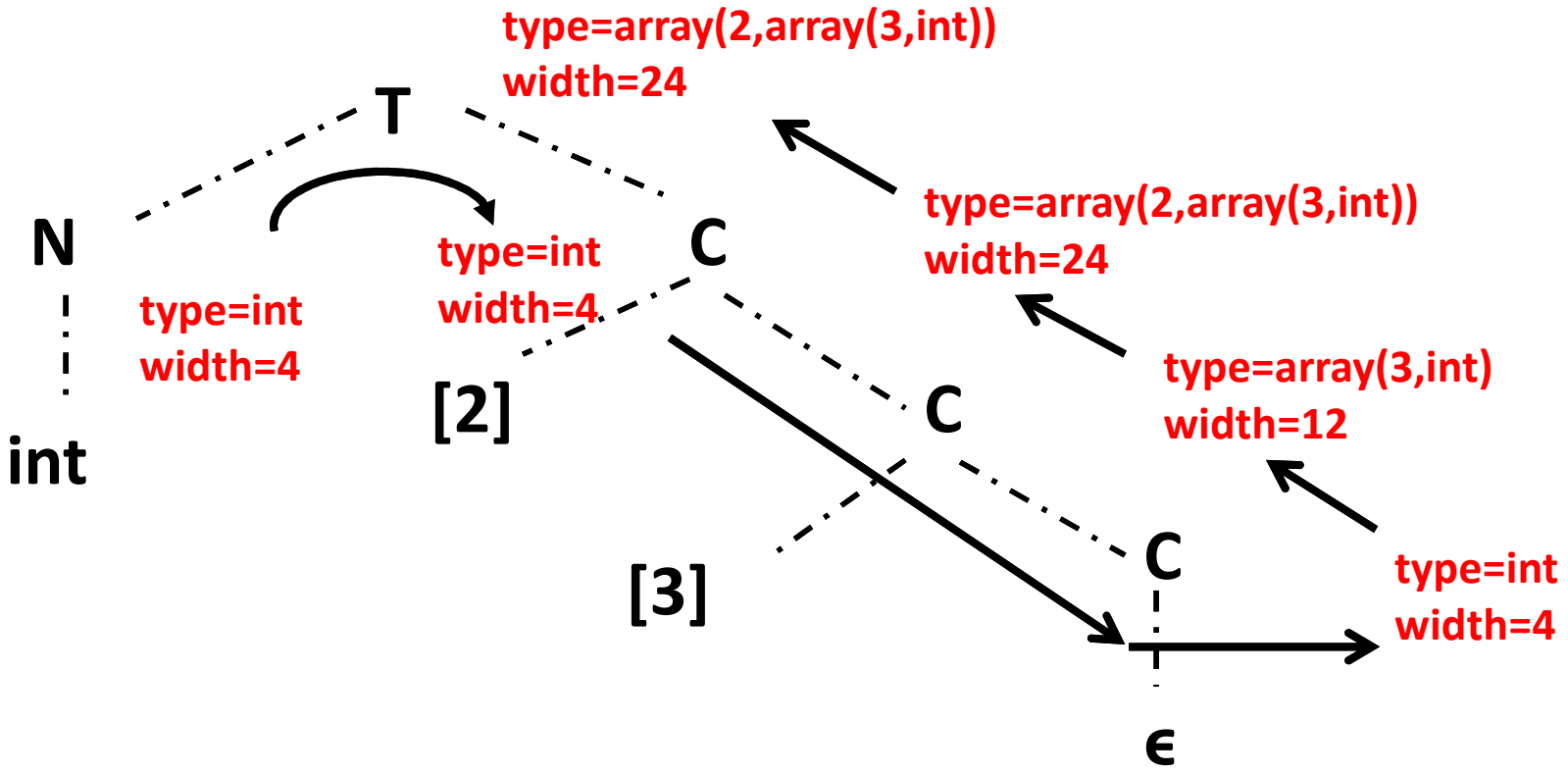
Storage Layout for Local Names

- **Computing types and their widths**

T→B	{t=B.type; w=B.width;}
C	
B→ int	{B.type=integer, B.width=4;}
B→ float	{B.type=float, B.width=8;}
C→ ϵ	{ C.type=t, C.width=w;}
C→ [num] C1	{ array(num.value, C1.type);
	C.width=num.value x C1.width;}

Storage Layout for Local Names

- Syntax-directed translation of array types



Sequences of Declarations in a Procedure

- Java/C allow all the declaration in a single procedure to be processed as group
- Declarations may be distributed with in procedure but can still be processed when it is analyzed
- Variable ***offset***: to keep tract of next available relative address
  - Before the first declaration ***offset*** set to 0
  - As each new name x is seen, x is entered to ST with relative address set to current value of offset and offset incremented by width of the type x;

Sequences of Declarations

P→ M D
M→ ϵ {offset =0}
D→ T id ; { top.put(id.lexme, T.type, offset);
offset=offset+T.width;}
D1
D→ ϵ

//M is marker and D is set of declaration
P→ M D
M→ ϵ { offset=0;}

Fields in Records and Classes

```
float x;
struct {float x ; float y; } p;
struct {int tag, float x, float y} q;
```

```
T→struct `{`   {Env.push(top); top=new Env();
                Stack=push(offset);offset=0;}

          D'}' {     T.type=struct(top);
                     T.width=offset;
                     top=Env.pop();
                     offset=Stack.pop();}
```

Translation of Expressions and Statements

- Translation of expression into three address code
  - An expression with more than one operator
  - Example: a + b*c
- We need to translate into instruction with at most one operator per instruction

- Notation: gen(x'=' y '+' z) represent x= y+z;

Three-address code for expressions

Production	Semantic Rules
S → id = E;	S.Code = E.code|| gen(top.get(id.lexme)'='E.addr)
E→ E1 + E2	E.addr=new Temp() E.code=E1.code||E2.code|| gen(E.addr'='E1.addr'+'E2.addr)
| -E1	E.addr=new Temp() E.code=E1.code|| gen(E.addr'='' 'minus' E1.addr)
|(E1)	E.addr=E1.addr E.Code = E1.code
|id	E.addr= top.get(id.lexme) E.code=''

Incremental Translation

- Code attribute E1.code can be long string, so they generally generated incrementally
- We can generated only TA instructions
- It can simply append earlier generated code so far
- The sequence may either be retained in memory for further processing or may be output incrementally

E→ E1 + E2 `E.addr=new Temp()`
`gen(E.addr'='E1.addr'+'E2.addr)`

Addressing Array Elements

- Accessing for a 1-dimensional array:

  x=A[i]

  x= A+i*w

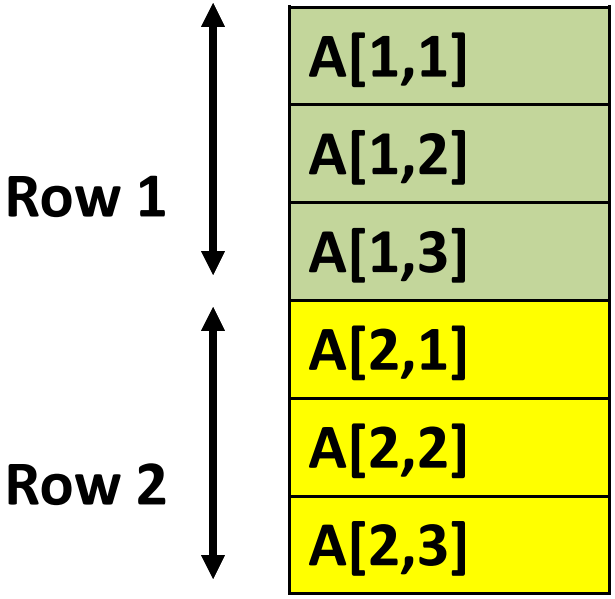
  A = &A[0]

  where w is width of the element

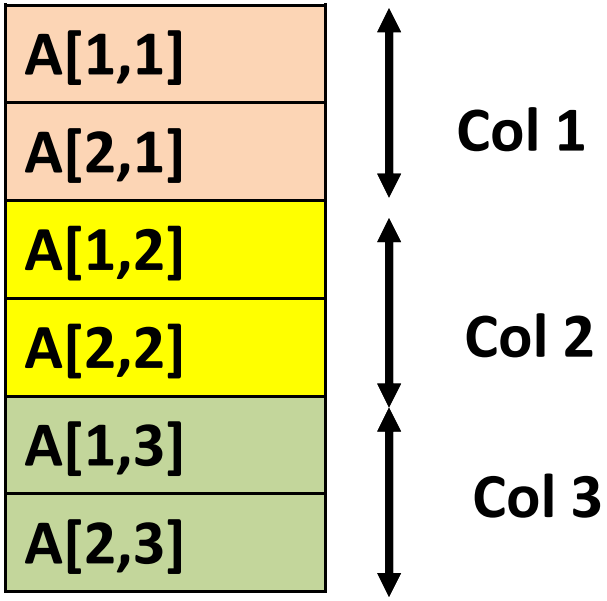
- Accessing for a 2-dimensional array:
  - Base+i1*w1+i2*w2
  - w1 is width of a row, w2 width of element of a row
- Accessing for a k-dimensional array:
  - Base+i1*w1+i2*w2+…+ik*wk

Addressing Array Elements

- Layouts for a two-dimensional array:



Row Major



Column Major

Translation of Array References

Nonterminal *L* has three synthesized attributes:

- *L.addr :* temp that is used to while computing offset for the array ref i_j x w_j
- L.array: pointer to the ST entry for an array name, ***L.array.base*** is used to determine actual L-value for an array ref
- L.type: type of syb-array generated by L

Annotated parse Tree for : c+a[i][j]