V
Type Checking

Shyamalendu Kandar
Assistant Professor,
Department of Information Technology
IIEST, Shibpur

Introduction

Compiler must check that the source program will follows both the syntactic and semantic conventions of the source language.

Static checks - reported during the compilation phase.

Dynamic checks - occur during the execution of the program.

Here we will discuss static checks.

- Why does it require?

Verify that a type of a construct matches that expected by its context.

Example:

i) For MOD operation : Integer operands

ii) For pointer: *

iii) For function f(id1, id2.....idn) → require correct argument.

These information are required during code generation.

Static Checks

Different Types of Static Checks:

Type Checks: Operator is applied to an incompatible operands. *Example: Mod in Pascal requires integers operands.*

Flow of control: Statement that cause flow of control to leave a construct must have some place to which to transfer the flow of control. *Example: Break in C causes control to leave the smallest enclosing while, for or switch statement.*

Uniqueness checks: Situation in which an object must be declared uniquely. *Example: labels in case statement.*

Name Related checks: Some times same name must appear two or more times. *Example: in Ada a loop or block may have a name that appear at the beginning and end of construct.*

This chapter discusses only type checking.

Type System

- A collection of rules for assigning type expressions to the various parts of a program.(such as variables, expressions, functions or modules)

Based on:

- ✓ Syntactic constructs in the language
- ✓ notion of a type
- ✓ the rules for assigning types to the language construct.

Purpose:

- to reduce bugs in computer programs by defining interfaces between different parts of a computer program.
- then checking that the parts have been connected in a consistent way.

Example:

- Both operands of an arithmetic operators of '+', '-', '*' are integer. --→Result is of type integer.
- If type of the operand for unary operator & is <type> then the type of the result is 'pointer to <type>'

<type> may be integer, float, char etc.

Strongly typed language: is one in which compiler can verify that program will execute without any type error. All checks are made static. Eliminates the necessity of dynamic type checking.

Type Expression

- Type of a language construct is denoted as 'Type expression'.
- either a basic type or formed by applying an operator called a type constructor to other type expression.

The following are different kinds of type expression.

I. Basic Type:

integer, real, char, Boolean that do not have internal structure.

void is a basic type denotes absence of a value

To indicate some type violation <type-error> is used.

II. Type Constructors:

Array, pointer, function.

Type Expression

II. A type constructor applied to a type expression is a type expression.

i) Array: array(I, T) is a type expression. I is the index set. Type of an array same with elements of type T. I may be a range of integer.

ii) Product: If T_1 and T_2 are type expression then their Cartesian product $T_1 \times T_2$ is a type expression.

iii) Records: record type constructor will be applied to a tuple formed from field name and field types. The difference between record and products is that fields of a records have names.

var table: array [1-100] of row ----- Variable table to be an array of records of this type.

iv) Pointer: If T is type expression then pointer(T) is a type expression denoting the type "Pointer to an object of type T"

```
int  p
int c=*p;
```

c has type pointer(p).

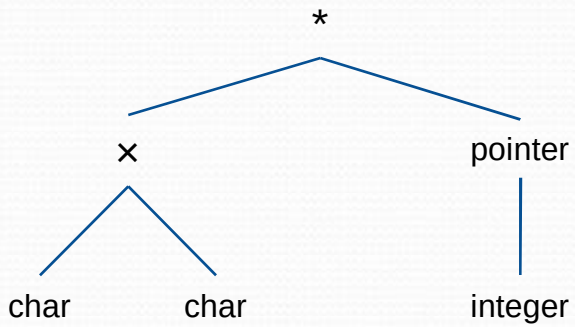
Type Expression

v) Functions: Function maps elements of one set (the domain) to another set (the range). Type of such function is denoted by D →R [D is the domain, R is the range]

int p;

function(char a, char b)= *p

char × char →pointer(integer).



Grammar for type expression

Type → int | float| ……|void|
error| variable|
array(size, type)|
pointer(type)|
function(type, type….)

Static and Dynamic Checking of Types

- Static: Checking done at compile time.
- Dynamic: checking done when target program runs.
- Any check can be done dynamically, if the target code carries the type of an element along with value of the element.
- Strong or sound type system eliminates the necessity of dynamic type checking.
- type-error will not generate while executing.
- some checks can be done only dynamically

```
table: array[1..255] of char;   //pascal
i : integer;
```

while executing compiler can not guarantee that during execution i will lie in the range 0 to 255.

Error Recovery

- Do something reasonable when an error is found.
- Must report the type of error and location of the error.
- Recover from the error and check rest of the input.

Specification of a Simple Type Checker

Type Checking of Expression:

Production	Semantic Rules
E → literal	E.type=char
E → num	E.type=integer
E → id	E.type=lookup(id.entry) δ
E → E_1 mod E_2	E.type=if(E_1.type=integer and E_2.type=integer) then integer else type-error
E → $E_1[E_2]$	E.type=if(E_2.type=integer^ and E_1.type=array(s,t)) then t else type-error //E_2 must be integer
E → * E_1	E.type=if E_1.type=pointer(t) then t else type-error

δ lookup(e) is used to fetch the type saved in the symbol table entry pointed by e. When an id is found in the program its declared type is fetched and assigned to the attribute type.
^ E2.type will always be an integer in array.

Specification of a Simple Type Checker

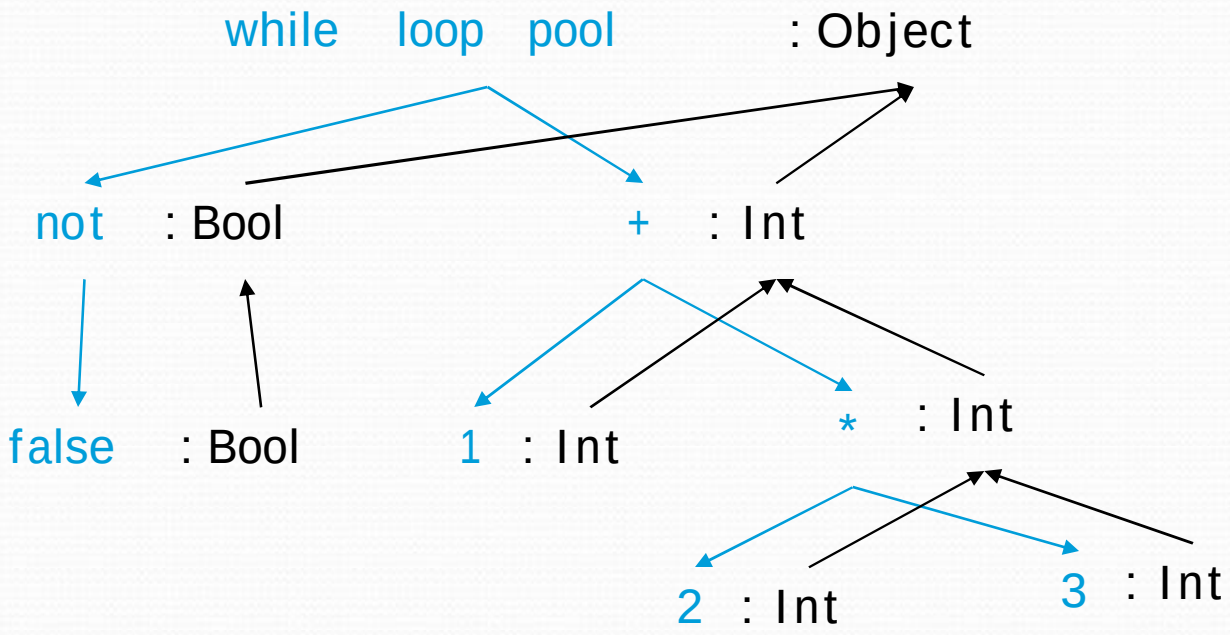
Type Checking of Statements:

Production	Semantic Rules
S → id :=E	{S.type := if id.type=E.type then void else type-error}
S → if E then S_1	{S.type :=if E.type = boolean then S_1.type else type-error}
S → while E do S_1	{S.type :=if E.type = boolean then S_1.type else type-error}
S → S_1 ; S_2	{S.type :=if S_1.type = void and S_2.type=void then void else type-error}

Left and right side assignment statement have same type

Typing: Example

- Typing for while not false loop 1 + 2 * 3 pool



Specification of a Simple Type Checker

- Type Checking of Functions:

Production	Semantic Rules
productions for function declarations $T \rightarrow T1 \rightarrow T2$	{ T.type := T1.type → T2.type }
$E \rightarrow E_1(E_2)$	{E.type := if E_2.type =s and E_1.type=s→t then t else type-error} [check whether E_1 has a type which is a function that has the domain same as type of E_2. If true the type of the function call is set to the type of the return value of the function. if the function s→t, the type of the function call is set to t.]

```
divide(int i)
{
   y= i/2;
   return y;
}

x=divide(5)
```

here i is an int. y may be real (int →real). Thus x must be real.

Equivalence of Type Expression

- A. Name Equivalence: Two types are name equivalent if they have same name or label.

```
typedef struct
{
    int  a1;
    int b[10];
} Value

Value a1, b1;   //These two are name equivalent
```

```
class A

class B

A a1, a1     //concept of oop..object of class A. Thus same type.

B b1, b2
```

a1 and b1 are not name equivalent--- as they their type name are different.

Equivalence of Type Expression

- B. Structural Equivalent: Two types are equal iff they have the same structure, which can be inplemented in different way.

Example

```
typedef struct{
  int i,
  int j;
  int *p
 } cell;
```

```
typedef struct{
  int a,
  int b;
  int *p1
 } element;
```

```
boolean sequiv( s, t )
{
    if s and t are the same basic type
            return TRUE;
    else if s == array( s1, s2 ) and t == array( t1, t2 )
            return sequiv( s1, t1 ) and sequiv( s2, t2 )
    else s == s1 x s2 and t = t1 x t2 then
            return sequiv( s1, t1 ) and sequiv( s2, t2 )
    else if s == pointer( s1 ) and t == pointer( t1 )
            return sequiv( s1, t1 )
    else if s == s1 → s2 and t == t1 → t2 then
            return sequiv( s1, t1 ) and sequiv( s2, t2 )
    return false
}
```

Algorithm

```
bool AreEquivalent(struct typenode *tree1, struct typenode *tree2) {
    if (tree1 == tree2)          // if same type pointer, must be equiv!
            return true;
    if (tree1->type != tree2->type)          // check types first
            return false;
    switch (tree1->type) {
            case T_INT:
        case T_DOUBLE: ...
                        return true;          // same base type
            case T_PTR:
                        return (AreEquivalent(tree1->child[0], tree2->child[0]);
            CASE T_ARRAY:
                        return (AreEquivalent(tree1->child[0], tree2->child[0]) &&
                        (AreEquivalent(tree1->child[1], tree2->child[1]);
```

Further study: Recursive type

Type Conversion

- Also known as type casting.
- Performed in two ways implicitly or explicitly.

Example:

x+i where x is of type float and i is of type int.

CPU instructions for addition could take EITHER float OR int as operands, but not a mix.

- int to float can be automatically converted by the compiler.
- Known as implicit conversion or coercions.
- In explicit conversion programmer must write something to cause the conversion.

- When an integer is assigned to real the conversion is to the type of the left side of the assignment.
- usual conversion is to convert int to float.
- then perform a real operation on the resulting part of real operands.

With postfix as an intermediate language for expressions, we could express the conversion as follows:

x i inttoreal float+

where inttoreal operator converts I from int to float and then float+ performs real addition on its operands.

Type Conversion

Production	Semantic Rules
E → num	E.type = integer
E → num.num^	E.type=real
E → id	E.type=lookup(id.entry)
E → E_1 op E_2	E.type= if E_1.type = integer and E_2.type=integer then integer else if E_1.type = integer and E_2.type=real then real else if E_1.type = real and E_2.type=integer then real else if E_1.type = real and E_2.type=real then real else type-error

^ like 10.56 (Fractional number)

Type narrowing

Self study:

Narrowing by refinement

Narrowing by assertion

Type interface and unification

Further study:

1. Type interface and unification
2. Type checking of overloaded operations.

References

- Prof. Hilfinger Type Checking CS164 Lecture 19