Chapter 6: Type Checking

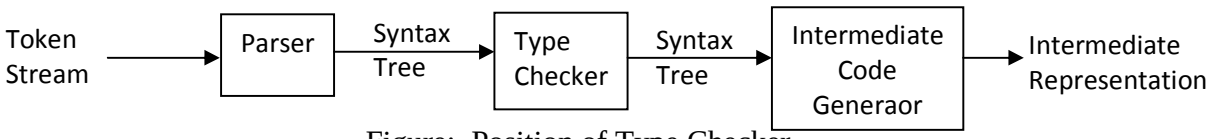

Figure: Position of Type Checker

A compiler has to do semantic checks in addition to syntax analysis. Type checking is one of the static semantic checks but some systems also use dynamic type checking.

Static checking: the compiler enforces programming language's *static semantics*

- Program properties that can be checked at compile time

Static checking examples:

- ✓ Type checks.
  - o Report an error if an operator is applied to an incompatible operand e.g

```
int op(int), op(float);
int f(float);
int a, c[10], d;
d = c+d;            // FAIL type mismatch
*d = a;             // FAIL  not a pointer type
a = op(d);          // OK: overloading (C++)
a = f(d);           // OK: coersion of d to float
vector<int> v;      // OK: template instantiation
```

- ✓ Flow-of-control checks.
  - o Statements that causes flow of control to leave a construct must have some
  - o place to which to transfer the flow of control e.g.

```
myfunc(int a)
{ cout<<a;
 break; // ERROR
//missplaced break
statement
}
```

```
myfunc()
{ …
  switch (a)
 { case 0:
          …
        break; // OK
    case 1:
          …      } }
```

```
myfunc(int a)
{
while(a)
{ ....
If(i>10)
Break;  //ok
}
}
```

- ✓ Uniqueness checks.
  - o There are situations where an object must be defined exactly once.
  - o labels, identifiers e.g.

```
myfunc()
{ int i, j, i; // ERROR
  …
}
```

- ✓ Name-related checks.
  - o Sometimes the same names may be appeared two or more times.
  - o Beginning and end of a construct

Dynamic semantics: checked at run time

- ✓ Compiler generates verification code to enforce programming language's dynamic semantics

A type system is a collection of rules for assigning type expressions to the parts of a program. A type checker implements a type system. A sound type system eliminates run-time type checking for type errors.

A programming language is strongly typed if its compiler can guarantee that there will be no type errors during run-time.

Type Expression:

The type of a language construct is defined by a "type expression". A type expression is defined as follows:

- A Basic type is a type expression e.g. integer, char, real etc.
- A type name is a type expression e.g. if int is named by a variable x , then x is a type expression.
- A type constructor applied to a type expression is a type expression. The constructor include - array, product, pointer, function or record. e.g.
  - ✓ T array[I] or array [I,T] is a type expression with I elements of type T.
  - ✓ If T_1 and T_2 are type expressions, then Cartesian product T_1 X T_2 is type expression - product
  - ✓ If T is a type expression, then pointer(T) is a type expression.
  - ✓ Function in programming languages is a mapping a domain type D to a range type R. e.g. int x int → pointer(char) denotes a function that takes a pair of integer and returns a pointer to char.
  - ✓ A record is a structured type

Specification of a simple type checker: The simple type checker is specified by the translation scheme that saves the type information for any identifier. Following is the translation scheme for a declaration.

$P \rightarrow D;E$
$D \rightarrow D;D$
$D \rightarrow id:T$ { *addtype(id.entry,T.val)* }
$T \rightarrow char$ { *T.val =char* }
$T \rightarrow int$ { *T.val =int* }
$T \rightarrow real$ { *T.val =real* }
$T \rightarrow *T_1$ { *T.val =pointer(T_1.val)* }
$T \rightarrow array[intnum]\ of\ T_1$ { *T.val=array(1..intnum.val,T_1.val)* }

The above set of declarations describes a translation schemes that saves the type of an identifier in a language like pascal e.g. **a:integer** saves the type of id 'a' as 'integer'

The type checking expression:

Following example shows the type checking expression in translation schemes

$E \rightarrow$ ***id*** { *E.type=lookup(id.entry)* }
$E \rightarrow$ ***literal*** { *E.type=char* }
$E \rightarrow$ ***intliteral*** { *E.type=int* }
$E \rightarrow$ ***realliteral*** { *E.type=real* }
$E \rightarrow E_1 + E_2$ { *if (E_1.type=int and E_2.type=int) then E.type=int*
else if (E_1.type=int and E_2.type=real) then E.type=real
else if (E_1.type=real and E_2.type=int) then E.type=real
else if (E_1.type=real and E_2.type=real) then E.type=real
else E.type=type-error }
$E \rightarrow E_1 [E_2]$ { *if (E_2.type=int and E_1.type=array(s,t)) then E.type=t*
else E.type=type-error }
$E \rightarrow *E_1$ { *if (E_1.type=pointer(t)) then E.type=t*
else E.type=type-error }

Another example: A simple language type checking specification:

$E \rightarrow$ ***true*** { *E.type = boolean* }
$E \rightarrow$ ***false*** { *E.type = boolean* }
$E \rightarrow$ ***literal*** { *E.type = char* }
$E \rightarrow$ ***num*** { *E.type = integer* }
$E \rightarrow$ ***id*** { *E.type = lookup(**id**.entry)* }
$E \rightarrow E_1 + E_2$ { *E.type := **if** E_1.type = integer **and** E_2.type = integer*
***then** integer **else** type_error* }
$E \rightarrow E_1$ ***and*** E_2 { *E.type := **if** E_1.type = boolean **and** E_2.type = boolean*
***then** boolean **else** type_error* }

Type Checking expression for another grammar: Example

T → int { *T.type=int* }
T → char { *T.type=char* }
T → real { *T.type=real* }
T → array [intnum, T_1] { *T.type=array(1..intnum.val,T_1.type)* }
T → Pointer(T_1) { *T.type=pointer(T_1.type)* }

Type Checking expression of an array of pointers to real, where array index ranges from 1 to 100

T→ real { T.type = real}
E → array [100, T] { if T.Type = real then T.type=array(1..100,T) else type_error() }
E→ Pointer[E_1] { if (E_1.type=array[100,real)) then E.type=E_1.type else E.type=type-error }

Type checking expression of statements.

Assignment statement:

S → **id** = E { if (id.type=E.type then S.type=void else S.type=type-error }

If then else statement:

S → if E then S_1 { if (E.type=boolean then S.type=S_1.type else S.type=type-error }

While starement:

$S \rightarrow$ while E do S_1 { if (E.type=boolean then S.type=S_1.type else S.type=type-error }

Type Checking expression for function

$E \rightarrow E_1 (E_2)$ { if (E_2.type=s and E_1.type=s$\rightarrow$t) then E.type=t
else E.type=type-error }
Ex: int f(double x, char y) { ... }
f: double x char $\rightarrow$ int
argument types return type

Function whose domains are function from two characters and whose range is a pointer of integer.

$T \rightarrow$ int { T.type = int }
$T \rightarrow$ char { T.type = char}
$T \rightarrow$ Pointer[T_1] { T.type = Pointer(T_1.type) }
$E \rightarrow E_1[E_2]$ { if (E_2.type = (char,char) and E_1.type = (char,char) $\rightarrow$Pointer(int)
then E.type = E_1.type else type_error}

Example: *consider the following grammar for arithmetic expression using an operator 'op' to integer or real numbers*
$E \rightarrow E_1$op E_2 | num.num|num|id
Give the syntax directed definition as translation scheme to determine the type of expression when two integers are used in expression resulting type is int otherwise real

Translation scheme will be:

*$E \rightarrow$ **id*** *{ E.type=lookup(id.entry) }*
*$E \rightarrow$ **num*** *{ E.type=integer }*
*$E \rightarrow$ **num.num*** *{ E.type=real }*
$E \rightarrow E_1$ op E_2 { if (E_1.type=integer and E_2.type=integer) then E.type=integer
else if (E_1.type=integer and E_2.type=real) then E.type=real
else if (E_1.type=real and E_2.type=integer) then E.type=real
else if (E_1.type=real and E_2.type=real) then E.type=real
else E.type=type-error }

Type Conversion and Coercion

Type conversion is explicit, for example using type casts. Consider expression a + b where a is of type integer and b is real. Since the representation of integer and real in the computer system is different and different machine instructions are used for operations of integer and reals, the compiler must convert one type operand into another type operand of operator + to ensure both operands are of same type when addition is takes place.

The conversion of type of one operand to another can be explicitly using cast operators that the type checker must incorporate.

Type coercion is implicitly performed by the compiler to generate code that converts types of values at runtime (typically to *narrow* or *widen* a type)

Both require a *type system* to check and infer types from (sub)expressions

Equivalence of Type expression:

As long as type expressions are built from basic types and constructors, a notion of equivalence between two type expressions is structural equivalence. Two type expressions are structurally equivalent iff they are identical. E.g. pointer(Integer) is equivalent to pointer(Integer).

The algorithm for testing structural equivalence can be adapted to test the equivalence. If two type expressions are equal then the operation is performed otherwise it requires type conversions if possible or report type error. For example, if two operand are of int and real type, then operation can be performed by type checking and conversion but if an array and a record is operated as a+r, then there will be type error.

Following is the example of an algorithm that can be adopted to check the equivalence of the type expression.

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

If array bound s1 and t1 in s=array(s1,s2) and t=array(t1,t2) are ignored if the test for array equivalence in above then it can be re-formulated as:

```
     if s=array(s1,s2) and t=array(t1,t2) then
          return sequival(s2,t2)
```