Principles of Compiler Design

Chapter 5

Type Checking

Type Checking

- A compiler has to do semantic checks in addition to **syntactic checks**.
- **Semantic Checks** can be
  - Static – done during compilation
  - Dynamic – done during run-time
- **Type checking** is one of these **static checking** operations ensures that errors are detected and reported.
- The following are examples of **static checks**:
  - Type Checks: A compiler should report an error if an operator is **applied** to an incompatible operand
  - Flow-of-Control Checks: Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control. For example, a break instruction in C that is not in an enclosing statement.
  - Uniqueness Checks: There are situations in which an object must be defined exactly once. For example, an identifier must be declared uniquely, labels in a case statement must be distinct.
  - Name -Related Checks: Sometimes, the same name must appear two or more times. For example, in Ada, a loop or block may have a name that appears at the beginning and end of the construct. The compiler must check that the same name is used at both places.

Contd..

- A **type checker** verifies that the type construct matches that expected by its context.
  - For example, a type checker should verify that the type value assigned to a variable is compatible with the type of the variable.
- Type information produced by the type checker may be needed when the code is generated.
- It is especially important for overloaded and polymorphic functions
  - An overloaded function represents different operations in different context (e.g., the + operator)
  - A polymorphic function can be executed with arguments of several types

Type Systems

- 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.

Type Expressions

- The type of a language construct is denoted by a *type expression*.
- A *type expression* can be:
  - **A basic type**
    - a primitive data type such as ***integer, real, char, boolean***, …
    - ***type-error*** to signal a type error
    - *void* : no type
  - **A type name**
    - a name can be used to denote a type expression.

Contd… Type Expressions

- **A type constructor applies to other type expressions.**
  - **arrays**: If T is a type expression, then *array(I,T)* is a type expression where I denotes index range. Ex: array(0..99,int)
  - **products**: If T_1 and T_2 are type expressions, then their Cartesian product $T_1 \times T_2$ is a type expression. Ex: int x int
  - **pointers**: If T is a type expression, then *pointer(T)* is a type expression.

    Ex: pointer(int)
  - **functions**: We may treat functions in a programming language as mapping from a domain type D to a range type R. So, the type of a function can be denoted by the type expression $D \rightarrow R$ where D & R are type expressions.

    Ex: int→int represents the type of a function which takes an int value as parameter, and its return type is also int.

Error Recovery

- At the very least, the compiler must report the nature and location of errors
- It is also desirable that the type checker recovers from errors and continues parsing the rest of the input
- The inclusion of error handling may result in a type system that goes beyond the one needed to specify correct programs
- For example, we may not know the type of incorrectly formed program fragments. In this case, techniques similar to those needed for languages that do not require identifiers to be declared will have to be used to cope with missing information.

Specification of Type Checker - Expressions

$E \rightarrow$ **id** { E.type=lookup(id.entry) }

$E \rightarrow$ **charliteral** { 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 }

$E \rightarrow E_1 \uparrow$ { if (E_1.type=pointer(t)) then E.type=t
else E.type=type-error }

Specification of Type Checker - Statements

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

Specification of Type Checker - Functions

$E \rightarrow E_1 (E_2)$ { if (E_2.type=s and E_1.type=s→t) then E.type=t
else E.type=type-error }

Ex: int f(double x, char y) { … }

f: double x char y → int

argument types return type

Type Conversions

- Consider expressions like x + i. Where x is of type real and i of type integer
- Of course, the machine cannot execute this operation as it involves different types of values
- However, most languages accept such expressions to be used; the compiler will be in charge of converting one of the operand into the type of the other
- The type checker can be used to insert these conversion operations into the **intermediate representation** of the source program
- For example, an operator **inttoreal** may be inserted whenever an operand need to be implicitly converted.

Endof ch5…

Thank You

?