8

Type Checking

1. What is a type system? List the major functions performed by the type systems.

Ans: A **type system** is a tractable syntactic framework to categorize different phrases according to their behaviors and the kind of values they compute. It uses logical rules to understand the behavior of a program and associates types with each compound value and then it tries to prove that no type errors can occur by analyzing the flow of these values. A type system attempts to guarantee that only value-specific operations (that can match with the type of value used) are performed on the values. For example, the floating-point numbers in C uses floating-point specific operations to be performed over these numbers such as floating-point addition, subtraction, multiplication, etc.

The language design principle ensures that every expression must have a type that is known (at the latest, at run time) and a type system has a set of rules for associating a type to an expression. Type system allows one to determine whether the operators in an expression are appropriately used or not. An implementation of type system is called **type checker**.

There are two type systems, namely, *basic type system* and *constructed type system*.

- **Basic type system:** Basic type system contains atomic types and has no internal structure. They contain integer, real, character, and Boolean. However, in some languages like Pascal, they can have subranges like 1 . . . 10 and enumeration types like orange, green, yellow, amber, etc.
- **Constructed type system:** Constructed type system contains arrays, records, sets, and structure types constructed from basic types and/or from other constructed types. They also include pointers and functions.

Type system provides some functions that include:

- **Safety:** A type system allows a compiler to detect meaningless or invalid code which does not make a sense; by doing this it offers more strong typing safety. For example, an expression `5/"Hi John"` is treated as invalid because arithmetic rules do not specify how to divide an integer by a string.
- **Optimization:** For optimization, a type system can use static and/or dynamic type checking, where static type checking provides useful compile-time information and dynamic type checking verifies and enforces the constraints at runtime.
- **Documentation:** The more expressive type systems can use types as a form of documentation to show the intention of the programmer.

- **Abstraction (or modularity):** Types can help programmers to consider programs as a higher level of representation than bit or byte by hiding lower level implementation.

2. Define type checking. Also explain the rules used for type checking.

Ans: Type checking is a process of verifying the type correctness of the input program by using logical rules to check the behavior of a program either at compile time or at runtime. It allows the programmers to limit the types that can be used for semantic aspects of compilation. It assigns types to values and also verifies whether these values are consistent with their definition in the program.

Type checking can also be used for detecting errors in programs. Though errors can be checked dynamically (at runtime) if the target program contains both the type of an element and its value, but a sound type system eliminates the need for dynamic checking for type errors by ensuring that these errors would not arise when the target program runs.

If the rules for type checking are applied strongly (that is, allowing only those automatic type conversions which do not result in loss of information), then the implementation of the language is said to be *strongly typed*; otherwise, it is said to be *weakly typed*. A strongly typed language implementation guarantees that the target program will run without any type errors.

Rules for type checking: Type checking uses syntax-directed definitions to compute the type of the derived object from the types of its syntactical components. It can be in two forms, namely, *type synthesis* and *type inference*.

- **Type synthesis:** Type synthesis is used to build the type of an expression from the types of its sub-expressions. For type synthesis, the names must be declared before they are used. For example, the type of expression E_1 * E_2 depends on the types of its sub-expressions E_1 and E_2. A typical rule is used to perform type synthesis and has the following form:

```
if expression f has a type s → t and expression x has a type s,
then expression f(x) will be of type t
```

  Here, s → t represents a function from s to t. This rule can be applied to all functions with one or more arguments. This rule considers the expression E_1 * E_2 as a function, mul(E_1,E_2) and uses E_1 and E_2 to build the type of E_1 * E_2.

- **Type inference:** Type inference is the analysis of a program to determine the types of some or all of the expressions from the way they are used. For example,

```
public int mul(int E1, int E2)
    return E1 * E2;
```

  Here, E_1 and E_2 are defined as integers. So by type inference, we just need definition of E_1 and E_2. Since the resulting expression E_1 * E_2 uses * operation, which would be taken as integer because it is performed on two integers E_1 and E_2. Therefore, the return type of mul must be an integer. A typical rule is used to perform type inference and has the following form:

```
if f(x) is an expression,
then for some type variables α and β, f is of type α → β and
x is of type α
```

3. Explain type expressions.

Ans: Type expressions are used to represent structure of types and can be considered as a textual representation for types. A type expression can either be of basic type or can be created by applying an operator (known as **type constructor**) to a type expression.

For example, a type expression for the array type `int[3][5]` considers it as an "array of 3 arrays having 5 integers in each of them", and its type expression can be written as "array (3, array (5, integer))". The type expression uses a tree to represent the type structure. The tree representation of array type `int[3][5]` is shown in Figure 8.1, where an operator array takes two arguments: a number and a type.

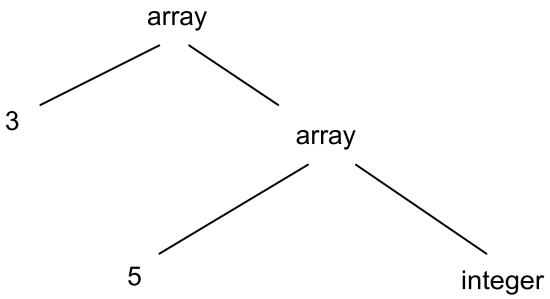

Figure 8.1 Type Expression of Int[3][5]

Type expressions can be defined as follows:

- **Basic types:** Every basic types such as Boolean, char, integer, float, void, etc., is a type expression.
- **Type names:** Every type name is a type expression.
- **Constructed types:** A constructed type applies constructors to the type expressions, which can be:
  - **Arrays:** A type expression can be constructed by applying an array type constructor to a number and a type expression. It can be represented as array `(I,T)`, where `I` is an index type and `T` is the type of array elements. For example, array `(1...10, integer)`.
  - **Cartesian product:** For any type expressions T_1 and T_2, the Cartesian product $T_1 \times T_2$ is also a type expression.
  - **Record with field names:** A record type constructor is applied to the field names to form a type expression. For example, `record{float a, int b}` → `X`;
  - **Function types:** A type constructor → is used to form a type expression for function types. For example, `A` → `B` denotes a function from type `A` to type `B`. For example, `real` × `real` → `real`.
  - **Pointers:** A type expression pointer (T_1) represents a pointer to an object of type T_1.

4. Define these terms: static type checking, dynamic type checking, and strong typing.

Ans: Static type checking: In static type checking, most of the properties are verified at compile time before the execution of the program. The languages C, C++, Pascal, Java, Ada, FORTRAN, and many more allow static type checking. It is preferred because of the following reasons:

- As the compiler uses type declarations and determines all types at compile time, hence catches most of the common errors at compile time.
- The execution of output program becomes fast because it does not require any type checking during execution.

The main disadvantage of this method is that it does not provide flexibility to perform type conversions at runtime. Moreover, the static type checking is conservative, that is, it will reject some programs that may behave properly at runtime, but that cannot be statically determined to be well-typed.

Dynamic type checking: Dynamic type checking is performed during the execution of the program and it checks the type at runtime before the operations are performed on data. Some languages that support dynamic type checking are Lisp, Java Script, Smalltalk, PHP, etc. Some advantages of the dynamic type checking are as follows:

- It can determine the type of any data at runtime.
- It gives some freedom to the programmer as it is less concerned about types.
- In dynamic typing, the variables do not have any types associated with them, that is, they can refer to a value of any type at runtime.

- It checks the values of all the data types during execution which results in more robust code.
- It is more flexible and can support union types, where the user can convert one type to another at runtime.

The main disadvantage of this method is that it makes the execution of the program slower by performing repeated type checks.

Strong typing: A type checking which guarantees that no type errors can occur at runtime is called **strong type checking** and the system is called **strongly typed**. The strong typing has certain disadvantages such as:

- There are some checks like array bounds checking which require dynamic checking.
- It can result into performance degradation.
- Generally, these type systems have holes in the type systems, for example, variant records in Pascal.

5. Write down the process to design a type checker.

Ans: A type checker is an implementation of a type system. The process to design a type checker includes the following steps:

1. **Identifying the available types in the language:** We have two types that are available in the language.
   - Base types (integer, double, Boolean, string, and so on)
   - Compound types (arrays, classes, interfaces, and so on)
2. **Identifying the language constructs with associated types:** Each programming language consists of some constructs and each of them is associated with a type as discussed below:
   - **Constants:** Every constant has an associated type. A scanner identifies the types and associated lexemes of a constant.
   - **Variables:** A variable can be global, local, or an instance of a class. Each of these variables must have a declared type, which can either be one of the base types or the supported compound types.
   - **Functions:** The functions have a return type, and the formal parameters in function definition as well as the actual arguments in the function call also have a type associated with them.
   - **Expressions:** An expression can contain a constant, variable, functional call, or some other operators (like unary or binary operators) that can be applied in an expression. Hence, the type of expression depends on the type of constant, variable, operands, function return type, and on the type of operators.
3. **Identifying the language semantic rules:** The production rules to parse variable declarations can be written as:

```
Variable Declaration → Variable
Variable → Type identifier
Type → int|double|Boolean|string|identifier|Type[]
```

The parser stores the name of an identifier lexeme as an attribute attached to the token. The name associated with the identifier symbol, and the type associated with the identifier and type symbol are used to reduce the variable production. A new variable declaration is created by declaring an identifier of that type and that variable is stored in the symbol table for further lookup.

6. What is type equivalence?

Ans: Type equivalence is used by the type checking to check whether the two type expressions are equivalent or not. It can be done by checking the equivalence between the two types. The rule used for type checking works as follows:

```
if two type expressions are equal then return a certain type else
return a type error()
```

When two type expressions are equivalent, we need a precise definition of both the expressions. When names are given to type expressions, and these names are further used in subsequent type expressions, it may result in potential ambiguities. The key issue is whether a name in a type expression stands for itself, or it is an abbreviation for another type expression.

There are two schemes to check type equivalence of expressions:

Structural equivalence: Structural equivalence needs a graph to represent the type expressions. The two type expressions are said to be structurally equivalent if and only if they hold any of the following conditions:

- They are of identical basic type.
- Same type constructor has been applied to equivalent types to construct the type expressions.
- A type name of one represents the other.

Name equivalence: If type names are treated as standing for themselves, then the first two conditions of structural equivalence lead to another equivalence of type expressions called name equivalence. In other words, name equivalence considers types to be equal if and only if the same type names are used and one of the first two conditions of structure equivalence holds.

For example, consider the following few types and variable declarations.

```
typedef double Value
...
...
Value var1, var2
Sum var3, var4
```

In these statements, `var1` and `var2` are name equivalent, so are `var3` and `var4`, because their type names are same. However, `var1` and `var3` are not name equivalent, because their type names are different.

7. Explain type conversion.

Ans: Type conversion refers to the conversion of a certain type into another by using some semantic rules. Consider an expression `a + b`, where `a` is of type `int` and `b` is of `float`. The representations of floats and integers are different within a computer, and an operation on integers and floats uses different machine instructions. Now, the primary task of the compiler is to convert one of the operands of + to make both of the operands to same type. For example, an expression `5 * 7.14` has two types, one is of `float` type and other one is of type `int`. To convert integer type constant into `float` type, we use a unary operator (`float`) as shown here:

```
x =(float)5
y = x * 7.14
```

The type conversion can be done implicitly or explicitly. The conversion from one type to another is called **implicit**, if it is automatically done by the computer. Usually, implicit conversions of constants

can be done at compile time and it results in an improvement in the execution time of the object program. Implicit type conversion is also known as **coercion**. A conversion is said to be **explicit** if the programmer must write something to cause the conversion. For example, all conversions in Ada are explicit. Explicit conversions can be considered as a function applications to a type checker. Explicit conversion is also known as **casts**.

Conversion in languages can be considered as *widening conversions* and *narrowing conversions*, as shown in Figure 8.2(a) and (b), respectively.

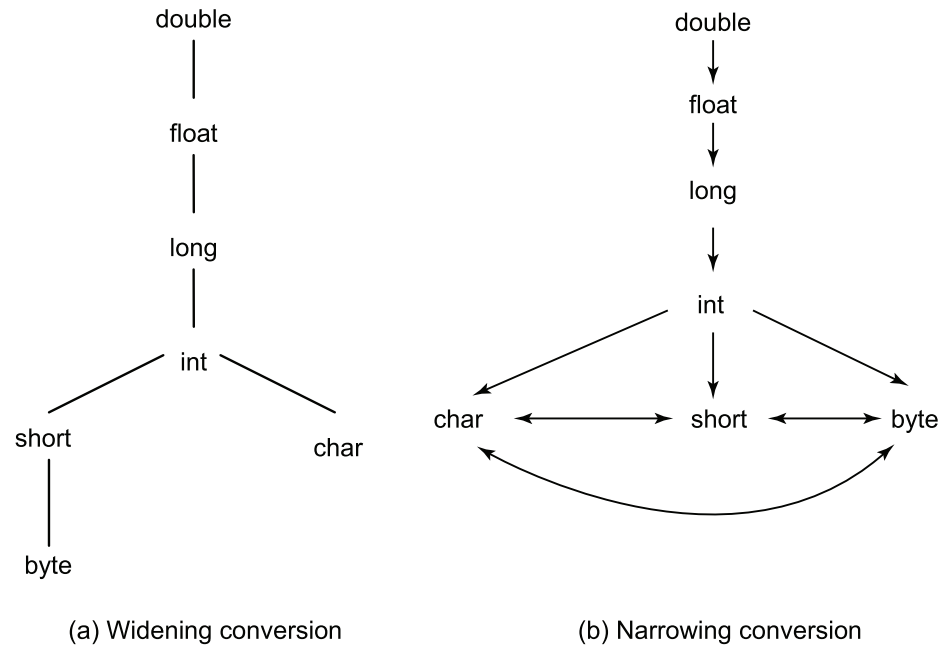

Figure 8.2 Conversion Between Primitive Types in Java

The rules used for widening is given by the hierarchy in Figure 8.2(a) and are used to preserve the information. In widening hierarchy, any lower type can be widened into a higher type like a `byte` can be widened to a `short` or to an `int` or to a `float`, but a `short` cannot be widened to a `char`.

The narrowing conversions, on the other hand, may result in loss of information. The rules used for narrowing is given by the hierarchy in Figure 8.2(b), in which a type `x` can be narrowed to type `y` if and only if there exists a path from `x` to `y`. Note that `char`, `short`, and `byte` are pairwise convertible to each other.

Multiple-Choice Questions

1. Which of the following is true for type system?
   (a) It is a tractable syntactic framework.
   (b) It uses logical rules to determine the behavior of a program.
   (c) It guarantees that only value specific operations are allowed.
   (d) All of these
2. A type system can be _____ type system or _____ type system.
   (a) Basic, constructed
   (b) Static, dynamic
   (c) Simple, compound
   (d) None of these

3. Which of the following is true for type checking?
   (a) It ensures type correctness.
   (b) It can only be done at compile time.
   (c) It can only be done at runtime.
   (d) All of these
4. A type checking is called strongly typed if _____.
   (a) It is performed at runtime.
   (b) It is performed at compile time.
   (c) The type checking rules are performed strongly.
   (d) Both (a) and (b)
5. In type synthesis, the names must be _____.
   (a) Declared after their use
   (b) Declared before their use
   (c) Need not be declared
   (d) Depends on the parent expressions
6. Why type expressions are used?
   (a) To free our program from errors
   (b) To represent structure of types
   (c) To represent textual representation for types
   (d) Both (b) and (c)
7. Which of the following is not true for static type checking?
   (a) It is performed at compile time.
   (b) It catches errors at compile time.
   (c) Most of the properties are verified at compile time.
   (d) It provides flexibility of performing type conversions at runtime.
8. A strong type checking ensures that _____.
   (a) No type errors can occur at compile time.
   (b) No type errors can occur at runtime.
   (c) Both (a) and (b)
   (d) None of these
9. Implicit type checking is also known as _____.
   (a) Casts
   (b) Explicit conversion
   (c) Manual conversion
   (d) Coercion

Answers

1. (a) 2. (a) 3. (a) 4. (c) 5. (b) 6. (d) 7. (d) 8. (b) 9. (d)