In programming languages, type checking is the process of verifying the compatibility of data types used in expressions. When comparing expressions for equivalence, type checking ensures that the types of operands on both sides of the comparison operator are compatible. The equivalence of expressions is determined based on the type rules specified by the language.

Here's an explanation of how equivalence of expressions is checked through type checking:

1. Basic Types:

- In many programming languages, basic types such as integers, floating-point numbers, characters, and booleans are compared for equivalence using the appropriate comparison operators (e.g., == for equality).
- Type checking ensures that the operands on both sides of the comparison operator are of the same type or compatible types.
- For example, in C, the equivalence of two integers is checked by comparing their values using the == operator.

2. Object Types:

- In languages that support object-oriented programming, equivalence of expressions involving objects may have different rules based on the language's type system.
- For reference types (objects), equivalence is typically determined by comparing references to objects rather than their actual contents.
- In languages like Java, the == operator checks whether two object references point to the same memory location (i.e., whether they refer to the same object).
- For content-based equivalence, such as comparing the values of two objects, the equals() method is often used.

3. Type Conversion and Promotion:

- Type checking also involves implicit or explicit type conversion or promotion to ensure compatibility when comparing different types.
- In some cases, if the types are not directly compatible, the language may attempt to convert one or both operands to a compatible type before performing the comparison.
- For example, in C, when comparing an integer with a floating-point number, the integer may be promoted to a floating-point type before the comparison.

4. Type Errors:

- If the types of operands in an equivalence comparison are incompatible or violate the language's type rules, a type error may occur.
- Type errors typically result in a compilation error or runtime exception, depending on when the type checking occurs in the language's execution model.

Type checker find whether two type expressions are equivalent or not.

This type equivalence is of two categories:

1. Structural equivalence
2. Name equivalence.

In type checking if two type expressions are equal then return a certain type else return type-error.

1. Structural equivalence:

Replace the named types by their definitions and recursively check the substituted trees. If type expressions are built from basic types and constructors then those expressions are called structurally equivalent.

Example:

S1	S2	Equivalence	Reason
Char	Char	S1 equivalent to S2	Similar basic types
Pointer (char)	Pointer (char)	S1 equivalent to S2	Similar constructor ptr to the char type

2. Name equivalence :

Two type expressions are name equivalent if and only if they are identical, that is if they can be represented by the same syntax tree, with the same labels.

Example:

```
typedef struct Node
{
int x;
} Node;
Node *first,*second;
Struct Node *last1,*last2;
```

The variables first and second are name equivalent similarly last1 and last2 are name equivalent.

Related posts:

1. Introduction to Compiler
2. Analysis and synthesis model of compilation
3. Bootstrapping and Porting
4. Lexical Analyzer: Input Buffering
5. Storage Allocation Strategies
6. Type Checking
7. Specification & Recognition of Tokens
8. Front end and back end of the compiler
9. LEX
10. Analysis synthesis model of compilation
11. Data structure in CD
12. Register allocation and assignment
13. Loops in flow graphs
14. Dead code elimination
15. Syntax analysis CFGs
16. L-attribute definition
17. Operator precedence parsing
18. Analysis of syntax directed definition
19. Recursive descent parser
20. Function and operator overloading
21. Storage allocation strategies
22. Storage organization

23. Parameter passing
24. Run time environment
25. Type checking
26. Code generation issue in design of code generator
27. Boolean expression
28. Declaration and assignment in intermediate code generation
29. Code optimization
30. Sources of optimization of basic blocks
31. Loop optimization
32. Global data flow analysis
33. Data flow analysis of structure flow graph (SFG)