Programming Language Concepts

Programming Language Theory

Topics

- **Type System**
- **Type Equivalence**
- **Type Compatibility**

Data Type

- A ***Data Type*** is a homogeneous collection of values and a set of operations applicable to the values for manipulation.
  - ***Homogeneous***: Same or similar kind(↔*heterogeneous*).
  - ***Values + Operations***: Data type is not only about the values, but also includes operations.
    - e.g.) We need different operations for integers, strings and arrays.

Type System

- A programming language has its own ***Type System*** - Information and Rules to manage data types.
- A type system usually consists of
    - A set of ***predefined types***,
    - Mechanisms to support ***definition of new types***,
    - Mechanisms to control types such as ***equivalence*** rules, ***compatibility*** rules and ***type inference***.

Denotable, Expressible, Storable

- Values are,
    - ***Denotable***, if we can put a name on them.
        - Variable (names), Function (names).
    - ***Expressible***, if we can get them from a complex expression.
        - Numbers, strings, even memory locations in C, which can be appeared in an expression.
    - ***Storable (or updatable)***, if we can store them in a variable.
        - Variable vs. Function - although code fragments of functions are stored in a disk, we cannot update them in a program.

Static and Dynamic Type Checking

- ***Dynamic Type Checking***: type constraints are checked during runtime.
- ***Static Type Checking***: checking of type constraints are conducted at compile time.
  - No runtime type checking - execution is more efficient.
  - Design of static type checking is more complex, and compilation takes longer.
    - But compilation happens only a few times, while executions are frequent.

Static and Dynamic Type Checking

```
int x = 0;
if(x > 0)
  x = "PL";
x = 1+2;
```

- Static type checking requires *conservative* type constraints.
  - Type checking is too excessive, so that it reports some errors which won't happen at runtime.
- In the example code, `x = "PL"` violates type constraints.
  - However, this code won't be executed during runtime.
- Because determining whether a program causes a type error is ***undecidable***.

Necessity of Combination

- Almost every high-level programming languages doing both static and dynamic type checking.
- Even though a language employs static type checking, it requires dynamic checking for some cases.
  - e.g.) Array index bound check.
  - If an array's size is dynamically decided, then the check for its index boundaries must be dynamic too.

Scalar and Composite Types

- ***Scalar Type***: no aggregation of different values.
  - Booleans, characters, integers, real numbers.
  - Enumerations
    - `type days = { Mon,Tue,Wed,Thu,Fri,Sat,Sun }`
  - Intervals: 1...10
- **Composite Types**: Non-scalar types.
  - Record (or structure), Array (or vector), Set, Pointer, Functions, Recursive Types, etc.
  - These types may have different operations.

Type Equivalence

- **Name Equivalence** - two types are equivalent if their names are identical.

- **Structural Equivalence** - two types are equivalent if their structures are identical.

- **Declaration Equivalence** - two types are equivalent if they are declared together.

- *Referential Transparency:* Two equivalent types can be substituted each other in any context, without change the meaning of programs.

- Modern languages are often using one rule with exceptions.

Name Equivalence

- Let's use a pseudo language for type definition.

- `type <type_name> = <expression>;`

- ```
  type Type1 = int;
  type Type2 = int;
  type Type3 = 1..100;
  type Type4 = 1..100;
  ```

- Name equivalence is very restrictive rule - all the types above are different.

- Java, C++ use name equivalence for most of their types.
  ```

Structural Equivalence

- Two types are equivalent if they have the same structure.

- More loose constraints.

- ```
  type Type1 = int;
  type Type2 = int;
  type Type3 = 1..100;
  type Type4 = 1..100;
  ```

- `Type1, Type2` are equivalent, and `Type3, Type4` are equivalent.

- Java arrays, C arrays and `typedef`.
  ```

Structural Equivalence

- There are some ambiguous cases.

- Different field names.

  - ```
    type Type1 = struct {
        int a;
        int b;
    }
    type Type2 = struct {
        int n;
        int m;
    }
    ```

  - Are `Type1, Type2` equivalent? - it depends on the language, but often types with different field names considered different.
    ```

Structural Equivalence

- Recursive Types.
  - ```
    type Type1 = struct {
        int a;
        Type2 b;
    }
    type Type2 = struct {
        int a;
        Type1 b;
    }
    ```
  - Are `Type1, Type2` equivalent? - Type check cannot solve such mutual recursion, hence they are considered not equivalent.
    ```

Declaration Equivalence

- In the middle of name and structural equivalence.

- *Weak* name equivalence: Consider equivalent for simple renaming or declared together (e.g. Pascal).

    - ```
      type Type1 = int;
      type Type2 = Type1;
      type Type3 = 1..100;
      type Type4 = 1..100;
      ```

    - `Type1`, `Type2` are equivalent, but `Type3`, `Type4` are still different.
      ```

Type Compatibility

- *Type T is compatible with type S, if a value of type T can be used in any context where a value of type S is used.*

- More specifically, type T and S are compatible when,

1. Types T and S are equivalent.

   - Referential transparency.

2. T's values are the subset of S's values.

   - intervals 1..10, 1..100.

Type Compatibility

3. All the operations of S can be applicable to T.

    - `type S = struct { int a; }`
      `type T = struct { int a; char b; }`

    - Only possible operation of S is accessing the field a.

    - $T \not\subset S$, but we can apply operations of S by taking only a of T.

4. T's values are correspond to values of S, in a *canonical fashion*.

    - T: `int` - S: `float`. T is not subset of S, but we can use `int` for `float` (e.g. 2 for 2.0).

5. T's values can be converted to values of S with transformation.

    - `float` can be converted to `int` by rounding (e.g., rounding down in C).

Type Conversion

- ***Implicit Conversion (coercion)***: also called forced conversion. Type conversions are done by the compiler.
    - When types T and S are compatible, conversions are automatically done even programmers didn't specify.
- ***Explicit Conversion (cast)***: Programmers explicitly indicate the type conversion.
    - `S s = (S) t;`

Type Checking and Inference

- *Type Checking*: when an expression E and type T are given, verify whether E is of type T.
  - `int f(int a) { return a+1; }`
  - `a+1` should be type T.
- *Type Inference*: only an expression E is given, derive the type of E.
  - `def f(a): return a+1`
  - 1 is `int` and + takes two integers, hence `a` is `int`, and function `f()` is `int->int`.

Type Safety

- All these type checking and inference are to secure *type safety* of a language.
- A type system (or a language) is ***type safe***, when no program can violate the distinction of types defined in the language.
- Theoretically, type safety is more restrict than you think.
    - *Unsafe Languages*: like C, C++, languages with pointers to access memory directly (memory safety issue).
    - *Locally Safe Languages*: some languages (e.g.Pascal) contain some unsafe parts.
    - *Safe Languages*: in theory, these languages don't generate any hidden type errors (e.g. Scheme, ML, Java).

Summary

- Type System
- Type Equivalence and Compatibility
- Type Checking, Inference and Safety