Control Abstraction + Data Types

Programming Language Theory

Topics

- Control Abstraction
  - Subprogram
- Data Types
  - Type System
  - Type Equivalence & Compatibility

Control Abstraction

Control Abstraction

- One of the two most important concepts of programming language, along with Data Abstraction.
- For a complex, large software, the main goal or requirement can be achieved,
  - by satisfying many smaller requirements.
  - Concept of Divide and Conquer.

Control Abstraction

- Suppose we're developing a mobile shopping app.
- We need the following functionalities.
  - Read product data.
  - Display products in the app.
  - Search for products.
  - Manage Customer information.
  - Product Reviews
  - Payment.
  - Manage Purchases.
  - Manage Delivery options.

Control Abstraction

- It is not a good idea to implement all these things in one big program.
- Instead, we can implement ***subprograms*** provide each function.
    - We can ***hide implementation details*** and separate them from design.
        - Easily switch to different implementation.
    - We only need to know ***how to use*** such subprograms.

Subprogram

- Also called ***procedure*** or ***function***.
- As we talked about before, we will use subprogram, procedure, and function together as synonyms.
- Although *subprogram* is the most general term, let's use function for this lecture.
    - Since it is more familiar, and more similar to what we will describe later in this lecture.

Function

- A function is a piece of code,
  - which is identified by its *name,*
  - is given a *local environment of its own*,
  - and exchanges information with the other parts of code using *parameters, return value and non-local environment.*
- We can ***define (or declare)*** a function and ***use (or call)*** the function.

Parameter Passing

- Parameter passing is one of the important ways for a subprogram to communicate with the other parts of a program.
- We can consider three kinds of parameters in subprogram's viewpoint.
  - IN parameters: communicate *from the caller to callee*.
  - OUT parameters: *from the callee to the caller*.
  - IN/OUT parameters: *bidirectional* communication.

Parameter Passing

- There are various *parameter passing modes*, which we will discuss in this lecture.

- For further explanation, we need to consider two terms.

    - ***Formal parameters***: these parameters are the ones appeared in the declaration of a function.

    - ***Actual parameters***: also known as ***arguments***. These are the ones passed to a function in a function call.

Call by Value

- It is a mode which corresponds to an IN parameter.
- Actual parameters can be expressions.
  - At the time of a call, actual parameters are evaluated and their ***r-values*** are associated with formal parameters.
  - When the function terminates, formal parameters are destroyed, hence these values are lost.
- Call by value (or pass by value) is the most simple and popular mode.
  - It is the only parameter passing mode in C and Java.

Call by Reference

- In this mode, parameters can be used both input and output (IN/OUT).
- Actual parameters must be expressions with ***l-values***.
- When a function is called, ***actual parameters' l-values*** are associated with formal parameters - ***aliasing***.
- In C++
    - `void func(MyObject& obj);`
    - `MyObject x; func(x);`
- Java has no call by reference: its variable model is reference model, but parameter passing itself is not call by reference.

Call by Reference

- Java has no call by reference: its variable model is reference model, but parameter passing itself is call by value only.
  - `class A { int v; }     swap(x, y);`
  - ```
    void swap (A a1, A a2) {
        int tmp = a1.v;  a1.v = a2.v;  a2.v = tmp; //simulate call by ref.
        A tmp2 = a1;  a1 = a2;  a2 = tmp2; //What happens?
    }
    ```
- Two references of objects in type `A` are copied and passed to `swap()`.
- You can change the value of their member variables using reference model (`a1.v`).
- But you can't actually swap two variables `x` and `y`.
    ```

Call by Constant

- In case of a parameter is not modified in the body of a function,
  - maintain semantics of call by value, while implementing it using call by reference.
  - parameters are considered ***read-only***.
  - In C++
    - `void func(const MyObject& obj);`

Call by Result

- It is a mode implements ***output-only*** communication.
- Actual parameters must be ***expressions with l-values***.
  - ***Backward assignment***: the values of formal parameters are *copied to locations obtained using actual parameters' l-values* after a function terminates.
  - ```
    void foo(result int x) { x = 5; }
    int y = 2;
    foo(y);  //y = 5 after this call.
    ```
    ```

Call by Value-Result

- Combination of Call by Value and Call by Result.
- Implement ***bidirectional*** communication.
- Actual parameters must give ***l-values***.
- At the call, r-values of actual parameters are assigned to formal parameters (Call by Value).
- Then value of formal parameters are copied backward using l-values of actual parameters (Call by Result).
  - ```
    void foo(val-res int x) { x = x + 1; }
    int y = 2;
    foo(y);  //y = 3 after this call.
    ```
    ```

Call by Value-Result

- Difference to call by reference.

  - ```
    void foo(val-res/ref int x, val-res/ref int y) {
        x = 1; y = 3;
        if(x == y) y = 1; //x == y ➞ true if call by ref. - aliasing.
    }
    ```

  - ```
    int a = 2;
    foo(a, a); //a is 3 after the call by value-result, 1 after call by ref.
    ```

Call by Name

- This mode is no longer used by modern programming languages.
- Although it is conceptually important, we will not discuss details of the method in this lecture.
- Simply speaking, it is a method to replace formal parameter names in the function body with actual parameter names.
  - ```
    void foo(name int x) { x = 1; }
    int y = 2;
    foo(y); → void foo(int y) { y = 1; }
    ```
    ```

Call by Name

- However, simple replacements may cause a problem.

  - ```
    int x = 0;
    int foo(name int y) {
      int x = 2;
      return x + y;
    }
    int a = foo(x + 1);
    ```

  - $a = x + x + 1 = 5$

  - ```
    int x = 0;
    int foo(name int y) {
      int z = 2;
      return z + y;
    }
    int a = foo(x + 1);
    ```

  - $a = z + x + 1 = 3$

- Hence it is necessary to ***pass actual parameters as well as their evaluation environment***.

Higher-Order Functions

- A function is considered ***higher order***, if
  - it accepts functions as parameters,
  - or it returns a function.
- This mechanism is supported by many programming languages, especially in *functional* languages.

Functions as Parameters

- On the right, there is an example C code using a function as a parameter.
- Function f is passed as a parameter to g.
- Variable x is defined multiple times.
- In function g, which binding of x should be used?
    - Which environment should be checked for name x?

```
int x = 1;
int f(int y) {
    return x+y;
}
int g(function<int(int)> h) {
    int x = 2;
    return h(3)+x;
}

int main(){
    //Functions as parameters
    int x = 4;
    int z = g(f);
```

Deep and Shallow Binding

- ***Deep Binding***: uses the environment active when the ***link between f and h*** **are made**.

- ***Shallow Binding***: uses the environment active when the ***call to f (using h) occurs***.

```
int x = 1;
int f(int y) {
    return x+y;
}
int g(function<int(int)> h) {
    int x = 2;
    return h(3)+x;
}

int main(){
    //Functions as parameters
    int x = 4;
    int z = g(f);
```

Deep and Shallow Binding

- **Static Scope + Deep Binding**
  - h(3) = 4, g(f) = 6
- **Dynamic Scope + Deep Binding**
  - h(3) = 7, g(f) = 9
- **Dynamic Scope + Shallow Binding**
  - h(3) = 5, g(f) = 7

Static + Deep
Dynamic + Deep
Dynamic + Shallow

```
int x = 1;
int f(int y) {
    return x+y; x = 1
}
int g(function<int(int)> h) {
    int x = 2;
    return h(3)+x; x = 2
}


int main(){
    //Functions as parameters
    int x = 4;
    int z = g(f); x = 4
```

What defines the Environment?

- Visibility Rules.
- Exceptions in Visibility Rules - need to consider re-defined names, use names after declaration.
- Scope Rules.
- Parameter Passing Modes.
- Binding Policy.

Functions as Results

- Function can return another function as a result.
- With static scope, the call `k()` is actually calling `s`, and x = 1 in `s`.
    - Environment is also considered in the returned function.
- Hence the result of `calling_s()` is actually a ***closure***.

```
int x = 1;
int s() {
    return x+1;
}
function<int()> calling_s() {
    return s;
}

int main(){
    //Functions as results
    int x = 4;
    function<int()> k = calling_s();
    int y = k();
```

Closure

- A ***Closure*** is a pair of (expression, environment),
  - which the environment includes all the free variables in the expression.
- ***Free variables*** are the variables used in the expression, but not declared in the environment.
  - In Python, they have a special distinction, so that global variables are not free variables.
  - But usually, global variables are free variables, unless they are declared again in the local environment.

Data Types

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

- Static type checking requires *conservative* type constraints.
  - Type checking is often 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.

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

- Because determining whether a program causes a type error is ***undecidable***.

Necessity of Combination

- Almost all high-level programming languages are doing both static and dynamic type checking.
- Although 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 types are equivalent for simple renaming or if they are 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, types 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 a 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.

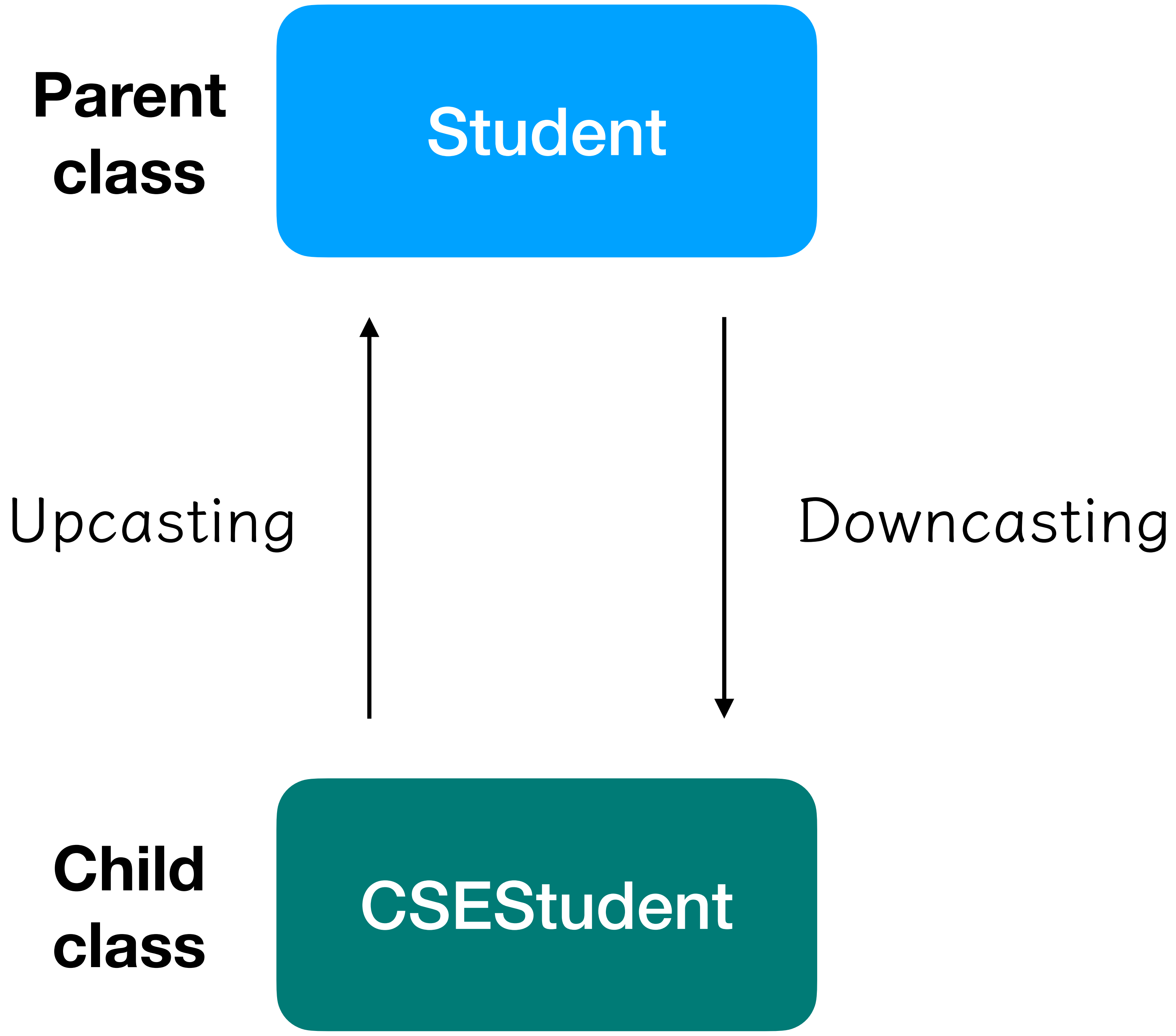
    - `float` can be converted to `int` by rounding (e.g., rounding down in C).
      ```

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;`

Upcasting vs. Downcasting

- Casting between inherited classes can be possible.
- **Upcasting**: Converting child to parent.
  - Implicit conversion is possible.
  - e.g.) `Student s = new CSEStudent();`
- **Downcasting**: Converting parent to child.
  - Need explicit conversion.
  - e.g.) `CSEStudent c = (Student)s;`



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 `int`.
- ***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 restricted 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

- Control Abstraction
  - Subprograms
  - Parameter Passing
  - Binding Policy
  - Higher-Order Functions
- Data Types
  - Type Equivalence and Compatibility
  - Type Checking, Inference and Safety