Types (2^{nd} Part)

Francesco Nidito

Programmazione Avanzata AA 2005/06

Outline

Reference: Micheal L. Scott, “Programming Languages Pragmatics”, Chapter 10

Types, type systems and type checking

- A type *type* is an abstraction system that defines
  - the memory layout of data
  - a set of operations that can be performed on value belonging to that type
- A *type system* consists of
  - a mechanism for *defining* types and *associating* them to language structures
  - a set of rules for: type equivalence, type compatibility and type inference
- *Type checking* is the process of ensuring that a program obeys the language's type compatibility rules

Class type

- Class is a *type constructor* like struct or array
- A class combines
  - Data (like struts)
  - Methods (operations on the data)
- A class has two special operations to provide
  - Initialization
  - Finalization

Inheritance

- A class B inherits from class A ($B <: A$) when an object of class A is expected an object of class B can be used instead
  Student $<:$ Person - a student can do everything a person can do
- Inheritance expresses the idea of adding features to an existing type
  Student $<:$ Person - a Student is a Person that follows PA lessons
- Inheritance can be *single* or *multiple*

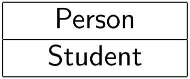
Single inheritance

- In single inheritance a class can extend only one class
- Very restrictive condition, it can't represent complex systems
- Single inheritance doesn't help expressivity

Single inheritance implementation

- When class *Student* inherits from class *Person* the memory layout of *Person* is stacked over *Student*
- From *Student* we can access *Person* using a pointer to the super class (`super`)

Student <: Person



Single inheritance (Example)


```
class Person {
  string Name;
  int SayHello() {
     print ''Hello my name is ''.Name ;
  }
}

class Sudent : Person {
  string Course;
  int SayHello() {
     super.SayHello();
     print ``I am a ``.Course.`` student'';
  }
}
```

Multiple inheritance

- In multiple inheritance a class can extend as many classes as it likes
- Very expressive, it can represent complex systems
- Multiple inheritance has both conceptual and implementation issues

Multiple inheritance implementation

- When class *JetCar* inherits from class *Car* and class *Jet* the memory layout of the *JetCar* is stacked with the ones of *Jet* and *Car*
- From *JetCar* we can access *Car* of *Jet* using a pointer to the super desired class
- We must specify the particular super class if both super classes have a method with the same name

$$\text{JetCar} <: \text{Jet}\ ,\ \text{JetCar} <: \text{Car}$$

Jet
Car
JetCar

Multiple inheritance (Example 1)

```
class Jet {
  void Fly() { /*...*/ }
}

class Car {
  void Drive() { /*..*/ }
}

class JetCar: Jet, Car {
  void FlyAndDrive() {
    Fly();
    Drive();
  }
}
```

Multiple inheritance (Example 2)

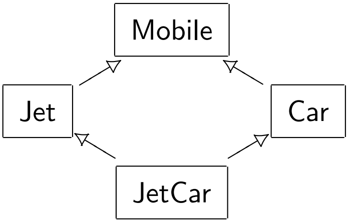
```
class Jet {
  void Stop() { /*...*/ }
}

class Car {
  void Stop() { /*..*/ }
}

class JetCar: Jet, Car {
  void Stop() {
    Jet::Stop();
    Car::Stop();
  }
}
```

Multiple inheritance (diamond problem)

- Suppose:
  JetCar <: *Jet* and *JetCar* <: *Car* and *Jet* <: *Mobile* and *Car* <: *Mobile*
- Mobile has a method called `MoveTo(x,y,z)`



Multiple inheritance (diamond problem)

- We can have two problems:
  - Memory layout is bigger (2 copies of *Mobile*)
  - If we have two copies of *Mobile*, when in *JetCar* we write `Mobile::MoveTo(x,y,z)` what happens?

$$\text{JetCar} <: \text{Jet} \text{ , } \text{JetCar} <: \text{Car}$$
$$\text{Jet} <: \text{Mobile}, \text{ Car} <: \text{Mobile}$$

Mobile(Jet)
Mobile(Car)
Jet
Car
JetCar

Multiple inheritance solution to the diamond problem

- C++ by default follows each inheritance path separately, so a *JetCar* object would actually contain two separate *Mobile* objects. But if the inheritance from *Mobile* is *virtual* C++ takes care to have only one copy of *Mobile*
- Eiffel handles this situation by a select and rename directives: *Jet* can have part of the interface of *Mobile* and *Car* another part, or we can rename the method `MoveTo(x,y,z)` as `MoveJetTo(x,y,z)` in *Jet* and `MoveCarTo(x,y,z)` in *Car*
- Perl handles this by specifying the inheritance classes as an ordered list. If *Jet* is specified before *Car* we have only *Mobile*(*Jet*) in memory
- Python creates a classes tree that would be searched in left-first depth-first order and then removes all but the last occurrence of any repeated classes

Mix-in inheritance

- In some languages (Java, C#), it is not possible to inherit from multiple classes
- Mix-in allows only to inherit from one class but to implement multiple interfaces
- No diamond problem!
- Partial code reuse and low expressivity (higher than single inheritance)

Mix-in inheritance "problems"

- No Implementation problems *only* expressivity problems
- Suppose that we want to program class *JetCar* (without *Mobile*) in a mix-in language
- We need to inherit from *Jet* and *Car*
- We decide to make *Car* an empty interface
- The implementation of the *Car* methods is in the *JetCar* class
- If we create a new class *BoatCar* we need to implement all the *Car* methods again.
- The problem can be partially solved encapsulating all the methods in a *Car_Implementation* class and calling them as external

Mix-in inheritance: the story so far

- Mix-in is an "old" term used in various contexts
- In C++ mix-in represents the fact that one, or more, of the inherited classes is *abstract*: it's like an interface but some implemented methods are given
- Mix-in term was born with CLOS: in CLOS a mix-in is a piece of code that can be attached on-fly to an object
- Other systems using CLOS-style mix-in are: Flavours, Ruby, Perl 6, D, XOTcl, Python and ActionScript

Upcasting and downcasting

- Object systems have two useful methods to use inheritance
  - Upcast
  - Downcast

Upcast

- Upcast provides the principal the abstraction of inheritance: the inheriting class can be used in every occurrence of the inherited class
- Each *Student* can perform everything a *Person* can do.

```
Student s := new Student();
Person p := s;
```

- It is not a real cast. The runtime simply use the *Person* part of the *Student* memory

Downcast

- Downcast permit to return from an upcast "transformation"
- When a *Student* exits from school became a simple *Person* but it can return to be a *Student* when it comes back to school

```
Person p := new Student();
Student s := (Student)p;
```

- When a downcast is performed the subclass must be specified because a superclass can have a large number of subclasses
- As in *upcast* operation: this is not a real cast operation

Upcasting and downcasting internals

- When are (Up|Down)cast operations checked?
  - Upcast can be verified at *compile* time
  - Downcast must be verified at *run* time

Late binding (By example)


```
class Person {
  int SayHello() { print ''I am a Person''; }
}

class Student : Person {
  int SayHello() { print ''I am a Student''; }
}

...

Person p := new Student();
p.SayHello(); //What does it print?
```

Late binding (By example)

- Which method is called?
- It depends on the language:
  - C++ calls `Person::SayHello()`
  - Java (or C#) calls `Student::SayHello()`

Late binding

- The mechanism that associates the method `Student::SayHello()` to a *Person* object is called *late binding*
- The main advantage of late binding is that we can create *generic code* that works on classes with a common ancestor
- A typical example is given by *graphical interface* classes that inherit from a *Drawable* class with a `paint()`
- Late binding introduces time overhead because it is checked at run time
- In C++ we can use late binding only if we declare `Person::SayHello()` as a *virtual* method

Late binding implementation

- How can late binding identify the method to be invoked?
- Each class using late binding we introduce a *v-table*
  - To each virtual method is associated a slot in the v-table
  - The pointer points to the body of the method to call
- Each instance of a class, in addition to class fields, has a pointer to the v-table: this costs space
- V-tables can be used to have information on the type of the object at run time: to have all the informations we need vtables also where they are not useful

Late binding example

```
class Person {
  int SayHello() { ... }
}

class Sudent : Person {
  int SayHello() { ... }
}

Person p := new Person();
p.SayHello();
```

$p \rightarrow$ VPointer $\rightarrow$ Person VTable $\rightarrow$ $Person :: SayHello()$

Late binding example


```
class Person {
  int SayHello() { ... }
  void TakeCar() { ... }
}

class Sudent : Person {
  int SayHello() { ... }
}

Student s := new Student();
s.TakeCar();
```

$s \rightarrow$ VPointer $\rightarrow$ Student VTable $\rightarrow$ $Person :: TakeCar()$

Late binding example


```
class Person {
  int SayHello() { ... }
  void TakeCar() { ... }
}

class Sudent : Person {
  int SayHello() { ... }
}

Person p := new Student();
p.SayHello();
```

$s \rightarrow$ [VPointer] $\rightarrow$ [Student VTable] $\rightarrow$ $Student :: SayHello()$

Abstract methods and classes

- *Abstract methods* are an high expressive system
- A method is abstract if it is declared in a class but the implementation is leaved to sub classes
- A class with one (or more) abstract method is called *abstract class* and cannot be instantiate. Only sub classes can be instantiated

```
class Shape {
  abstract void Draw();
}

class Square : Shape {
  void Draw() { ... }
}
```

Overloading

- *Overloading* permits to bind more than one object to a single name
- For instance:

```
class A {
  int foo() { ... }
  int foo(int i) { ... }
}
```

- The name `foo` identifies two different methods

Overloading internals

- Overloading is a compiler trick!
- This process is called *name mangling*
- The compiler generates a different method name for each version of `foo` using the type in input (the output type must be the same!)
- For instance: `foo()` becomes `foo_v`, and `foo(int)` becomes `foo_i`
- When the method is invoked the compiler chooses the appropriate version of `foo`
- Sometime implicit conversions can lead to an ambiguity in the choice: a compiler error is raised

Operator overloading

- *Operator overloading* allows to give a different semantic to standard language operators as $+$ and $-$
- In some languages the overloading of the operators is performed in the same way of method overloading
- Conceptually the invocation of overloaded operators is rewritten as a method
- For instance we can create a *Matrix* class and define the sum operation on it (Example in C++):

```
Matrix a,b,c;
...
c = a + b; // operator=(c, operator+(a, b))
```