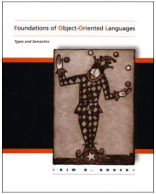
Fundamentals of Object-Oriented Languages

UNIVERSIDADE DA CORUÑA

GRAO EN ENXEÑERÍA INFORMÁTICA
DESEÑO DAS LINGUAXES DE PROGRAMACIÓN

Based on chapters 2 and 5 of: Kim B. Bruce, *Foundations of Object-Oriented Languages*. The MIT Press, 2002



Outline

Objects

- Objects: encapsulate both state and behavior. Contains:
    - Instance variables (aka fields): representing the state of the object
    - Methods: representing the behavior that the object is capable of performing
- When a message is sent to an object, the corresponding method of the object is executed
- Usually sharing semantics is used: all objects are implicitly references (pointers):
    - $o := o'$ will result in o referring to the same object as o'
    - $o = o'$ if and only if they both have the same reference
- Usually by default the instance variables of an object are not accesible from outside of that object's methods but methods are publicly accesible

Classes

- Classes are extensible templates for creating objects, providing initial values for instance variables and the bodies for methods
- All objects generated from the same class share the same methods but contain separate copies of the instance variables
- New objects can be created from a class by applying the `new` operator to the name of the class

- In many OO languages class names are used for
  - a name for the class
  - a name for a constructor of the class
  - a name for the type of objects generated from the class

Example in a language-independent notation

```
class CellClass {
  x:Integer := 0;

  function get(): Integer is
  { return self.x }

  function set(nuVal: Integer): Void is
  { self.x := nuVal }

  function bump(): Void is
  { self<=set(self<=get()+1) }
}
```

- self (`this` in C++ and Java) is used in method bodies to indicate the object currently executing the method
- <= (Smalltalk notation) to represent sending a messsage to an object
- . (dot) to get access to instance variables of the current object

Object types

- Object types should only reveal the names and types (signatures) of the messages that may be sent to them
- The should not carry implementation information
- Example: the type of objects **generated** by class `CellClass` is

```
CellType = ObjectType { get: Void -> Integer
                        set: Integer -> Void
                        bump: Void -> Void}
```

- Two classes can generate objects of the same type even if the methods result in different behaviors
- Dynamic method invocation is the mechanism by which the object receving a message is responsible for knowing which method body to execute

Subclasses

- One of the important features of object-oriented languages is the ability to make incremental changes to a class by creating a subclass (aka *derived class* in C++)
- A subclass may be defined from a class by either **adding** to or **modifying** the methods or instance variables of the original class (its superclass).
- super is used to refer to the methods (not instance variables) of the superclass

Defining a subclass in a language-independent notation

```
class ClrCellClass inherits CellClass modifies set {

  color:ColorType := blue;

  function getColor():ColorType is
  { return self.color}

  function set(nuVal:Integer); Void is
  { self.x := nuVal;
    self.color := red }
}
```

- The type of the objects generaed by `ClrCellClass` is

```
ClrCellType =
  objectType { get:Void -> INteger;
               set:INteger -> Void;
               bump:Void -> Void;
               getColor:Void -> ColorType}
```

Subtype polymorphism

- **We say type T is a subtype of U**

$$T <: U$$

  if a value of type T can be used in any context in which a value of type U is expected
- A value of type T can *masquerade* as an element of type U in all contexts if $T <: U$
- U is a supertype of T if T is a subtype of U
- It is not neccesary to restrict subtypes to those relationships that arise from subclasses (although most languages do).

- Note: other kinds of polymorphism: ad-hoc (overloading), generic or true polymorphism (e.g. generic classes in Java)

Covariant and contravariant changes in types

- If the types in a class C are replaces by sybtypes in a subclass SC, the the changes as referred to as covariant
- Replacing a type by a supertype in a subclass is referred to as a contravariant change

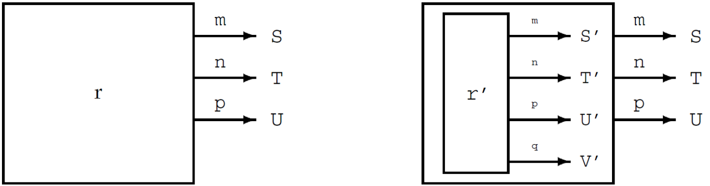
```
class C {
  v:T1 := ...;
  function m(p:T2): T3 is {...}
}

class SC inherits C modifies v, m {
  v:T1' := ...;
  function m(p:T2'): T3' is {...}
}
```

- Type safety is preserved in subclasses if we allow only:
    - covariant changes to the return types ($T3' <: T3$)
    - contravariant changes to parameter types ($T2 <: T2'$)
    - no changes at all to types of instance variables ($T1 = T1'$)

Subtyping for record types

A record r, and another record r' masquerading as an element of the same type as r:

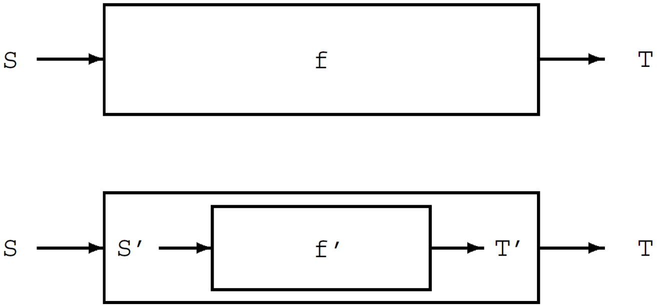


$$\{|\ l_1 : T_1'; l_2 : T_2'; l_3 : T_3'; l_4 : T_4'\ |\} <: \{|\ l_1 : T_1; l_2 : T_2; l_3 : T_3\ |\}$$

$$T_1' <: T_1 \qquad T_2' <: T_2 \qquad T_3' <: T_3$$

Subtyping for function types

A function $f : S \to T$, and another function $F' : S' \to T'$ masquerading as having type $S \to T$



$S' \to T' <: S \to T \qquad S <: S' \qquad T' <: T$

Subtyping for variables

A variable $x : \text{Ref}\ T$, and another variable $x' : \text{Ref}\ T'$ masquerading as having type $\text{Ref}\ T$:

- In value-supplying (r-value) contexts we must have $T' <: T$
- In value-receiving (l-value) contexts we must have $T <: T'$
- Therefore, $\text{Ref}\ T' <: \text{Ref}\ T$ if and only if $T' <: T'$ and $T <: T$

Overloading

- A method **name** is overloaded in a context if it is used to represent two or more distinct methods, and where the method represented by the overloaded name is determined by the type or signature of the method

```
class Rectangle {
  ...
  function contains(pt:Point): Boolean is
    { ... }
  function contains(x, y:Integer): Boolean is
    { ... }
}
```

- For each method call, the language processor **statically** determines what method body is to be executed

Overriding

- Overriding **methods** always occur in different classes, typically when one of the classes is a subtype of the other.
- They tipically have the same signature (subtyping is allowed in some languages)
- Message sends involving overrriden methods are **resolved at run time**

Warning

- The interaction between overloaded methods names with static resolution, and overridden methods with dynamic resolution can result in great confusion

- For example:
  - in C++, overloaded methods must be defined in the same class
  - in Java, the overloading can happen when a method in a superclass is inherited in a subclass that has a method with the same name, but different signature

```
class C {
  ...
  function equals (other:CType): Boolean is
    { ... }                                   // equals 1
}

class SC inherits C modifies equals {
  ...
  function equals(other:CType): Boolean is
     { ... }                                  // equals 1

  function equals(other:SCType): Boolean is
     { ... }                                  // equals 2
}
```

- The first definition of `equals` in SC takes a parameter of type `CType`, overriding the `equals` method of class C
- The second `equals` method in class SC takes a parameter of type `SCType`: it is treated as been statically different from the others (overloading)
- `SCType` $<:$ `CType`

Example (2)

- Let c and c' variables with declared type CType
- Let sc a variable with declared type SCType
- Given the code

```
c  := new C;
sc := new SC;
c' := new SC;
```

- The variable c of type CType is assigned an object created from class C
- The variable sc of type SCType is assigned an object created from class SC
- The variable c' of type CType is assigned an object created from class SC. This is legal because SCType $<:$ CType

Example (3)

- Consider the code:

```
c <= equals(c);
c <= equals(c');
c <= equals(sc);
```

Example (3)

- Consider the code:

```
c <= equals(c);
c <= equals(c');
c <= equals(sc);
```

- All 3 messages send to c result in the execution of method **equals 1 from class C** (there is only one method with name `equals` in class C)

Example (4)

- Consider the code:

```
c' <= equals(c);
c' <= equals(c');
c' <= equals(sc);
```

Example (4)

- Consider the code:

```
c' <= equals(c);
c' <= equals(c');
c' <= equals(sc);
```

- All 3 messages send to `c'` result in the execution of method **equals 1 from class SC** (as `c'` has type `CType`, it is statically determined that the messages correspond to equals 1; as `c'` is assigned an object of type `SCType`, the method equals 1 from class SC is actually executed at run time)

Example (5)

- Consider the code:

```
sc <= equals(c);
sc <= equals(c');
sc <= equals(sc);
```

Example (5)

- Consider the code:

```
sc <= equals(c);
sc <= equals(c');
sc <= equals(sc);
```

- The first two message sends to `sc` result in the execution of method **equals 1 from class SC** (as both `c` and `c'` has type `CType`, it is statically determined that the messages correspond to equals 1)
- The last message send results in the execution of method **equals 2 from class SC** (as parameter `sc` has type `SCType`, it is statically determined that the messages correspond to equals 2)

Remember

- **The overloading of** `equals` **is resolved statically**
- That is, the selection of *equals 1* versus *equeals 2* is resolved solely on the **static types** of the receiver and parameters

Another example

```
class C {
  ...
  function equals (other:CType): Boolean is
    { ... }                                   // equals 1
}

class SC inherits C modifies equals {
  ...
  // equals 1 not overrode in class SC

  function equals(other:SCType): Boolean is
     { ... }                                  // equals 2
}
```

- Which of the two method bodies is executed in Java and C++ for each of the 9 message sends given in previous slides?

Java and C++ behave differently

- In the case of Java, the answer is like before but method *equals 1* is always from class C (this method is not overridden in class SC)
- C++ does not allow overloading methods across class boundaries. Therefore, in C++

  ```
  sc <= equals(c);
  sc <= equals(c');
  ```

  are rejected by the compiler as a result of static type checking (there not exists an `equals` method with parameter of type `CType` in class SC)
- The combination of static overloading and dynamic method invocation in OO languages is likely to result in confusion on the part of programmers