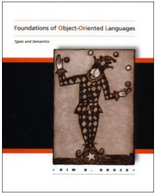
Type Problems in Object-Oriented Languages

UNIVERSIDADE DA CORUÑA

GRAO EN ENXEÑERÍA INFORMÁTICA

DESEÑO DAS LINGUAXES DE PROGRAMACIÓN

Based on chapter 3 of: Kim B. Bruce, *Foundations of Object-Oriented Languages*. The MIT Press, 2002



Outline

Type checking of OO languages is difficult

- Subtyping and inheritance create dificulties in type-checking
- There was great confusion over what is the proper subtyping rule for functions (remind: contravatiant subtyping for the types of parameters)
- Modifying existing methods can create problems: if a method `m` being modified was used in a second method `n` of the superclass, then changes in types in `m` may destroy the type correctness of `n` when it is inherited in the subclass

Strenghts and weakness of the type-checking sytems of popular OO languages

- Some show little or no regard for static typing (e.g. Smalltalk)
- Some have relatively inflexible static type systems, requiring typecasts to overcome deficiencies (unchecked in C++ and Object Pascal, checked at run time in Java)
- Some provide mechanisms like "`typecase`" statements (e.g. Modula-3, SImula 67, Beta)
- Some allow "reverse" assignments from superclasses to subclasses, which require run-time checks (e.g., Beta, Eiffel)
- Inflexibility in changing the types of parameters of methods overriden (e.g., Object Pascal, Modula-3; earlier versions of C++ and Java)
- Too much flexibility in changing the types of parameters of methods overriden or instance variables, requiring extra run-time or link-time checks to catch the remaining type errors (e.g., Eiffel, Beta)

Simple type systems are lacking in flexibility

- Languages like Object Pascal, Modula-3, C++ arose as OO extensions of imperative languages (we could include Java as well).
- They inherits relatively simple and straightforward type systems, in which the programmer has little flexibility in redefining methods in subclasses: a redefined method and variable instance cannot change type when overriden
- These type systems are called *invariant* type systems
- The programmer must use mechanisms as typecasting when he/she is able to deduce more refined types for methods than the language allows to be written

The need to change return types in subclasses

- What should be the type of `clone`?
- If we clone an object of type `AType`, we would like `clone` to return an object of type `AType`...
- ... but in the invariant type systems, the return type of clone is a top `ObjectType`, even though the method actually return a value of type `AType`!

Example (1)

```
class C {
   ...
   function deepClone():CType is
      { self <= clone(); ... }
}

class SC inherits C modifies deepClone {
   newVar: newObjType := nil;

   function newMeth():Void is
      { ...}

   function setNewVar (newVarVal:newObjType):Void is
      {self.newVar := newVarVal }

   function deepClone():SCType is { // illegal return type change!!
                                    // must be CType instead
      var newClone: SCType := nil   // local variable

      newClone := super <= deepClone(); // (*) another problem!!
      newClone <= setNewVar(newVar <= deepClone());
      return newClone
   }
}
```

- Object Pascal, C++ and Java programmers would be forced to perform type cast to tell the compiler that the clones object has type SCType

Example (2)

- We could try to solve the probleam by adding a method `SCdeepClone` to class SC:

```
function SCdeepClone():SCType is {
    ...
    }
```

- But suppose we add a method `m` to class C:

```
function m();Void is{
      ...
      self <= deepClone();
      ...
    }
```

- Given a variable `sc` of type `SCType`, the execution of `sc <= m()` will result in the execution of the method `deepClone` from the superclass rather than the newly defined `SCdeepClone`

Example (3)

- Even if in modern versions of languages it is possible to specialize the return type of methods in subclasses, this does not solve all of our problems:

```
function deepClone():SCType is {
   var newClone: SCType := nil   // local variable

   newClone := super <= deepClone(); // (*) another problem!
   newClone <= setNewVar(newVar <= deepClone());
   return newClone
}
```

- The right side of the assignment on line (*) returns a value of type `CType` but the type of the variable on the left side is a subtype of `CType`, thus the assignment is illegal!
- A Type cast would have to be inserted to make the assignment legal
- The issue gets worse and worse as deeper subclasses are defined

Binary methods

- Binary methods are methods that have a parameter whose type is intended to be the same as the receiver of the message
- Messages involving comparisons, such as `eq`, `lt`, `gt` or other binary relations are common examples of binary methods
- The problems arise with subclasses

Example of problem with binary method

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

class SC inherits C modifies equals {
    ...
    function equals(other:CType):Boolean is
         // Want parameter type to be SCType instead
    { super <=equals(other);
    ...  //Can not access SC-only features in other
    }
    ...
}
```

- We can not make a covariant change in the type of parameters of method equals (this will break the correctness of the type system) even though may be what is desired here

Typecasting

```
class SC inherits C modifies equals {
   ...
   function equals(other:CType):Boolean is
   { var otherSC:SCType := nil  // local variable

     otherSC := (SCType)other   // type cast!
     return super <=equals(other) and ...
   }
   ...
}
```

- The expression `(SCType)other` represents casting the expression `other` to type SCType
- These casts can fail at run time
- This technique requires the programmer to be disciplined in adding casts to all overriden versions of binary methods

Singly-linked nodes

```
NodeType = ObjectType{
   getValue: Void -> Integer;
   setValue: Integer -> Void;
   getNext: Void -> NodeType;
   setNext: NodeType -> Void;
}

class Node {
   value:Integer := 0;
   next:NodeType := nil;

   function getValue():Integer is { return self.value }

   function setValue(newValue:Integer):Void is { self.value := newValue }

   function getNext():NodeType is { return self.next }

   function setNext(newNext:NodeType):Void is { self.next := newNext }
}
```

Doubly-linked nodes

```
DoubleNodeType = ObjectType{
   getValue: Void -> Integer;
   setValue: Integer -> Void;
   getNext: Void -> NodeType;
   setNext: NodeType -> Void;
   getPrev: Void -> DoubleNodeType;
   setPrev: DoubleNodeType -> Void;
}

class DoubleNode inherits Node modifies setNext {
   previous:DoubleNodeType := nil;

   function getPrev(): DoubleNodeType is { return self.previous  }

   function setPrev(newPrev:DoubleNodeType):Void is { self.previous := newPrev }

   function setNext(newNext:DoubleNodeType):Void is //error - illegal change to parameter type
   { super <= setNext(newNext);
     newNext <= setPrev(self) }
}
```

- Illegal covariant change to parameter type in `setNext`
- Method `getNext` returns type `NodeType` (?!)

Type cast for legal doubly-linked nodes

```
LglDoubleNodeType = ObjectType{
   getValue: Void -> Integer;
   setValue: Integer -> Void;
   getNext: Void -> NodeType;
   setNext: NodeType -> Void;
   getPrev: Void -> LglDoubleNodeType;
   setPrev: LglDoubleNodeType -> Void;
}

class LglDoubleNode inherits Node modifies setNext {
   previous:LglDoubleNodeType := nil;

   function getPrev(): LglDoubleNodeType is { return self.previous  }

   function setPrev(newPrev:LglDoubleNodeType):Void is { self.previous := newPrev }

   function setNext(newNext:NodeType):Void is         //no change to parameter type
   { super <= setNext(newNext);
     ((LglDoubleNodeType)newNext) <= setPrev(self) }  // type cast
}
```

Problems with type cast for legal doubly-linked nodes

- But if a programmer send `setNext` to an object generated from `LglDoubleNode` with a parameter that is generated from `Node`, it will not be picked up statically as an error. Instead the cast will fail at run time.
- Even if a variable `dn` has type `LglDoubleNodeType`, the evaluation of

```
(dn <= getNext()) <= getPrev()
```

  will generate a static type error, because the type checker can only predict that the results of `dn <= getNext()` will be of type `NodeType`, not the more accurate `LglDoubleNodeType`
- Thus, even if the programmer has created a list, all of whose nodes are of type `LglDoubleNodeType`, the programmer will still be required to write type casts to get the typechecker to accept the program

Independent double-linked nodes

- A possible solution is to define a class for doubly-linked nodes independently of class Node
- But then, the type of the objects generated by this new class can not be a subtype of `NodeType`
- Therefore, methods of the class `Node` can not be sent to object generated by this new class and viceversa
- A lot of redundant code is needed for practical applications

- **These problems are not special to the `Node` example, but arise with all binary methods because of the desire for a covariant change in the parameter type of binary methods**

Other typing problems

- There are other examples where it is desirable to change a type in a subclass in a covariant way
- In these cases, the type to be changed may have no relation to the type of objects generated by the classes being defined
- Many examples of this phenomenon arise when we have objects with other objects as components

Circle and ColorCircle example

```
class CircleClass {
   center:PointType := nil;
   ...
   function getCenter():PointType is
   { return self.center }
   ...
}

class ColorCircleClass inerits CircleClass modifies getCenter {
   color:ColorType := black;
   ...
   function getCenter():ColorPointType is { ... }
                           // Illegal type change in subclass!
   ...
}
```

Summary

- In orde to guarantee tye safety in static type system:
  - Methods overriden in subclasses must have contravariant parameter types
  - Methods overriden in subclasses must have a covariant return type
  - Instance variable types are invariant in subclasses
  - We must not hide in a subclass methods that were visible in the superclass