Typing in Object-0riented Languages

Achieving predictable and type safe semantics for covariant specialisation.

Overview

- Types
- Some Type Rules
- Class Based Object-oriented languages
- Subclasses and Subtypes
- Method override
- Type rules for method override
- Summary/ Conclusions

What is a Type?

- Traditional Answers:
  - A set of values
  - A set together with specified operations
- Syntactic Answer:
  - Type expression
  - Introduction and Elimination Rules
  - Equations relating Introduction and Elimination Rules

Example:

- Type expression: A x B
- Introduction Rule:

$$\frac{x : A \ , \ y : B}{< x \ , \ y > : AxB}$$

- Elimination Rule:

$$\frac{p : AxB}{fst(p) : A \quad snd(p) : B}$$

- Equations:
  - intro; elim = identity
  - fst <x, y> = x, snd<x,y> = y
  - <fst(p), snd(p)> = p

Subtyping & Subsumption

- Variable types may be assigned values of the same type e.g. int x; x = 3;
- **Subtype rule:** If S <: T then a variable of type T may be assigned a value of type S. However, it is not permitted to assign a variable of type S, a value of type T as it is possible that some operations are allowed on S which are not permitted on T.

Covariance

- The product operator, x , is a **covariant** operator in both arguments as A x B varies in the same sense as A or as B.
- The covariance subtype relation may be stated as:
  - A x B <: A1 x B1 if A <:A1 and B <:B1.
    - i.e. if A <:A1, B <:B1, a:A and b:B, then by subsumption a: A1 and b:B.
    - Hence, the pair (a, b) has type A x B and by subsumption it also has the type A1 x B1.
    - Therefore, any type A x B also has the type A1 x B1 whenever A <:A1 and B <:B1.

Contravariance

- If A --> B is a function type with argument type A and result type B then the contravariant rule may be stated as follows:

  A --> B <: A1 --> B1 provided that A1 <:A and B <:B1.

- The function type A-->B may be substituted for A1 --> B1 without causing a typing error if the parameter types vary contravariantly and the result types vary covariantly.
- Using a contravariant policy, in the case of multiple arguments, all arguments types are contravariant.

Invariance

- If an operator doesn't enjoy any covariance or contravariance properties then it is said to be Invariant.
- A # B <:A1 # B1 if A=A1 and B = B1 where # is an operator applied to types A and B.
- A lot of controversy has arisen in the object-oriented community with respect to the use of covariance or contravariance of argument types when defining methods in a class hierarchy.

Object-Oriented Languages

- Object based languages
- Class based languages:
  - a class is a template which specifies the implementation of an object. Classes therefore provide a new type in our programming language e.g. Smalltalk, C++, Java
  - Attributes & Methods
  - Variables declared of the class type, each declaration produces a new instance of the class, that is, a new object.
  - The data and the methods that may be used with the new type are wrapped up together in a single unit.

A Class Example

```
class Employee
{       private:
                string Name;
                int PRSI;
        public:
                Employee(string n, int p, int a)
                { Name := n; PRSI := p; Age := a;};
                void Display()
                {cout << Name <<  PRSI;};
}
```


Inheritance

```
class Hourly: public Employee
{       private:
        float hourlyrate;
        public:
        float Calcpay(float time)
        {return (time * hourly rate);
        }
}
```


Method Suites

- **Dynamically typed languages:**
  - Each class has a method suite which contains its own methods and attributes. When a method is executed each class is searched from subclass to superclass (parent) until the corresponding method is found.
- **Statically typed languages:**
  - Suites are collapsed at each level so that a search through the hierarchy is not necessary i.e. as if each class or subclass has its own complete method lookup table
- **Type Safety**:
  - A programming language is type safe if it ensures that operations are only applied to values which are permitted for their type.

Subtyping & Subclassing

- **Subclassing:** allows the reuse of implementations
  - Given a class P with the method m, a class C that inherits from class P, will contain the same method interface and implementation of method m.
- **Subtyping:** Allows Extension of data types
  - supports external reuse as any methods that may be applied to a data type may also be applied to a subtype of that type.
  - This is possible as the subtype relation depends on the public interface of the type not on its implementation.
- **Subtype rule:** $\frac{M : S \;,\; S <: T}{M : T}$

Subsumption

- **Subsumption:** if c1:C and C<: P then c1:P.
- The effect of this at run time is that an instance of a subclass may be used wherever an instance of a superclass is expected without error**.**
- **Example:** A method may except a parameter of type P but may receive a parameter of type C where C is a subclass of P. Subsumption permits this but may cause unexpected behaviour as the type C is more restricted than its parent type P.
- Static knowledge about the true type of an object, here a value of type C can be viewed as a value of supertype P - the value is subsumed from type C to type P.
- p.m(c) calls parents method

Is Inheritance Subtyping?

- Subtyping is concerned with the class interface only and not with the implementation. Objects from several classes may be of the same type - therefore there need be no connection between subclass and subtype.
- However, most object-oriented languages do identify type and class as well as subtype and subclass - ie a class is identified with the type of objects it generates, while only subclasses generate subtypes
- C++ private inheritance => subclassing is not subtyping
- Java Interfaces => separates classes from types

Overloading & Overriding

Overloading:

- A method definition is **overloaded** if its interface changes.
- There may be more than one version of a method permitted in a class
- The version of the method executed depends on the parameters/types supplied by the programmer during execution.

Overriding:

- A if a derived class redefines the base class method definition is **overridden** method implementation.
- A different behaviour is associated with the method of a derived class than that associated with the same method in the base class.

Overriding Display

```
class Hourly: public Employee{
  private:
     float hourlyrate;
     public:
     float Calcpay(float time)
     {return (time * hourly rate);}
     void Display()
     {cout<< name << PRSI  <<age;
      cout << "Rate:" << Hourlyrate;}
};
```


Parameters?

- No parameters => overridden versions should also have no parameters.
- With parameters => types permitted for the parameters of the overridden methods depend on the typing rules used : Covariant, Contravariant, Invariant
- If a method is overridden incorrectly, the compiler will interpret the new method as an overloading of the original method.

Rule 1: Invariant Subtyping

- The method interface must be the same as that of the inherited method.
- When the method is executed, the type of the object calling the method is determined.
- If the method definition is not present in the corresponding class then the parent classes are searched in turn until the first definition of the method is found.
- If the derived class tries to override a method using different parameters/ return types then the method will be overloaded instead of overridden.

Example:

```
class Parent
{ private: …
  public:
  void m()
 { cout << "Parent Method Called";}
}
class Child : public Parent
{ private: …
  public:
  void m()
  {cout <<"Child Method Called";}
};
```


Method Execution

- `Parent p = new Parent;`
- `Child c = new Child;`

- Method Execution p.m();results in the message "Parent Method Called" being displayed.
- Method Execution c.m();results in the message ”Child Method Called" being displayed.

Adding parameters

- The same result happens when the base class method being overridden takes **parameters** and **these parameter types are not changed in the derived class** when the method is overridden.
- Example: p.m(p); c.m(p);

  The result is the same for both Java and C++ as the Child method m overrides the Parent method m.
  - p.m(p) calls the parents version
  - c.m(c) calls the childs version

Changing return types in subclasses

```
class point
{protected:    int x; int y;
 public:       point() {x = 0; y = 0};
               point move(int dx, int dy)
               {point p; p.x =p.x + dx;
                 p.y = p.y + dy; return p;}
}
class colourpoint: public point
{ private:     colourtype colour ;
  public:      colourpoint move(int dx, int dy)
               {colourpoint cp; cp.x =cp.x + dx;
                 cp.y = cp.y + dy; return cp;}
}
```


- Invariant type systems do not allow the subtype to be written as a return type of the declaration of the method in Colour point.

- They require the return type to remain as type point, and then require the programmer to perform a type cast to convince the system to treat the returned value as a colour point.

Changing parameter and attribute types in subclasses

- Methods with self as parameter: Methods which have a parameter whose type is intended to be the same as the receiver of the message e.g. equal, lessthan …
- In these method it is often convenient to change both return parameters and parameter types of methods.

- Nat <: Real
- Nat x Nat <: Real x Real
- Real x Real -> Bool <: Nat x Nat -> Bool
  - e.g. Equal (Real, Real) : Bool, Equal(Nat x Nat) : Bool
- Contravariant Rule=> Type Safe
- If the covariant rule was used in this case it would be easy to write down well typed terms that produce run time type errors.
  - **It would be possible to use methods of type**
  - **Equal(Nat,Nat) : Bool instead of Equal (Real, Real) : Bool. This is well typed due to the subtying rule but may cause a runtime error i.e. type safety is lost.**

Why we need covariance ...

- C++ and Eiffel give up type safety and use the covariant subtyping rule for specialisation
- Given a class node with
  - attributes int data and node * next
  - methods to include getNext which returns a Node, and setNext which takes a node as its parameter.
  - If we try to define a class DoubleNode we need to redefine the entire class instead of using inheritance as invariant class typing doesn't allow us to override the type of the attribute **next** which should now point to a doublenode and the parameters and return types of the above method which should now be of type Doublenode.

Rule 2:Covariant Subtyping

- A method is overridden by a method of the same name with a subclass of the original parameter as its new parameter.
- Problem: Combining covariant specialisation and inheritance based subtyping in a type safe manner.
- When overriding a method in a language which uses covariance specialisation =>
  - it must have the same number of parameters as the method it overrides
  - The type of each parameter must be a subtype of the type of the corresponding parameter in the overridden method
  - The type of the result must be smaller than or equal to the result of each overridden method.

Overriding with parameters

```
class Parent
{ private:...
  public:
  void m(Parent p)
  {cout <<"Parent Method Called";};
}
class Child : public Parent
{private: …
  public:
  void m(child c)
  {cout <<"Child Method Called";
};
```

- **p.m(p);** The **Parent's method** is called (in both Java and C++) as the Parent class contains a method which takes an instance of Parent as its parameter.
- **c.m(c);** The **Child's method** is called as the method is overridden through the **covariant** typing rule.

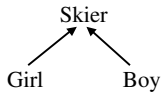
  If the **novariant** typing rule is used here (as in Java) The result is the same as if the child's method is still called but it is called due to **overloading** rather that overriding.
- **c.m(p);** This method call causes a **compile time error** if the covariant typing rule is used when overriding methods. Invariant typing => method is overloaded

Polymorphism & Covariance

Parent P = new Parent;
Child C = new Child;
P = C;

- Method m is covariantly overridden =>

  P. m(P) and P.m(C) calls the parents version of the method m as P is of type parent. P.m(c) may cause runtime errors as the type which the method is actually executed on is a child type.
- Solution: Contravariance?




```
Skier = << Skier:Roomate,
            share(Skier other){ ….Roommate = other;} >>
Girl = << Girl: Roommate,
            share(Girl other){….Roommate = other;} >>
Boy = << Boy: Roommate,
            share(Boy other){….Roommate = other;} >>

Skier S; Girl G; Boy B;                 //reference types
S = B; S.share(G);                      //Runtime error
```


- The same problem arises when a class hides an attribute in its derived classes e.g.
- The class polygon has an addvertex feature, rectangle could inherit from polygon but wouldn't want to export the addvertex feature.
- As with the covariance problem with polymorphism, it is possible to store a child type in a reference to a parent type, and to call the hidden addvertex problem on a rectangle object.

Why use Contravariance?

- We can guarantee **type safety** when forming subclasses if we restrict ourselves to overriding a method with a new one whose type is a **subtype** of the type of the original.
- This restriction on changing the types of methods in subclasses is sufficent to guarantee that the resulting types will be subtypes - this restriction means that the contravariant subtyping rule is the one used.

Rule 3: Contravariance

- A method which takes a parameter is overridden by a method which takes a parameter type that is a supertype of the original parameter type.
- The parameter becomes more generalised as the class to which the method belongs becomes more specialised.
- Smalltalk is an example of a language that uses this typing rule.
- Contravariance:

  A --> B <: A1 --> B1 provided that A1 <:A and B <:B1.

- To ensure type safety in an oo language we require that the types of the parameters of an overridding method use **supertypes** of those of the parameters of the overridden method.
- Otherwise the subclass cannot be considered a subtype of its its superclass and hence objects of the subclass cannot be used when objects of the superclass are expected.
- **Co variance Example:** Point specialised as colourpoint. Disallows a useful specialisation of **equal** in colourpoint as there is no way to compute the colour attribute of a point.
- This specialisation would mean that the colourpoint would not be a subtype of the point and hence could not be used where points are expected.

A problem

- Point = << int x, int y; equal:Point-->bool>>
- Colourpoint = << int x, int y, colour c; equal:Colourpoint--> bool>>

Assume **p: point; q: colourpoint**

- **q.equal(p)** may give a type error as p is not a subtype of q.

A related issue: Self

- Self: provides a way of referring to the object executing the method.
- Given the classes Parent and Child as defined earlier, where the overridden method m uses the novariant subtyping rule and the reference variables p, p1, c are defined as:
  - Parent p = new Parent;
  - Parent p1 = new Child;
  - Child c = new Child

Method calls ...

- **p.m(); ** Parents version executed in both C++ and in Java as self binds to the reference type of p in both cases.
- **c.m();** Child's version executed in both C++ and in Java as self binds to the reference type of c in both cases.
- **p1.m();**
  - Childs version of the method m is executed in Java
  - Parents version of the method m is executed in C++.
  - This is because the referential type self binds to the reference type (L -Value) of p1 (Parent) in C++ and it binds to the contents (R value) of the referential type p1(Child) in Java.

- If the classes as defined with the method m **covariantly** overridden then the results of executing m in all cases is that the **Parents version** of the method is called in both C++ and in Java. This is because the reference variable **self binds to the reference type** in all cases.
- p1.m(p);
- p1.m(c)

Arrays

- Arrays are covariant for read, contravariant for write hence invariant overall if both are allowed.
- If arrays allow individual components to be updated then Array (s) <: Array (t) only if S <: T and T<: S.
- In Java: Array (s) <: Array(T) if S<: T is **type safe** if the arrays are read only
  - A has type Array(s), B has type Array(T), t has type T, then b[i] = t is allowed but A[i] = t is not.

Covariance V Contravariance

- For Convariance: Covariance for method parameter types is often useful and actual runtime errors rarely result in practice
- For Contravariance: Unrestricted covariance for parameter types and instance variables results in an environment which is not typesafe
- Problem: Design static type systems for oopls with sound typing rules that support examples where covariance appears to be necessary.

Summary

- Covariance: Correct Rule for specialisation
- Contravariance: Correct Rule for substitutivity of methods
- Real-world problems often lend themselves to covariant solutions. Incorrect programs in the presence of covariant argument redefinition can cause runtime type errors.
- Contravariance solutions guarantee type safety but doesn't lend itself well to real world problems