UNIVERSITY OF MANCHESTER INSTITUTE OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTATION

Object Oriented Specification, Design and Implementation

Lecture 6: Polymorphism[1]

Introduction

Earlier lectures have examined the notion of *abstract data types* and have also introduced the notions of *inclusion* and *parametric polymorphism*, i.e. inheritance and type parameterisation. This lecture will identify, first, different forms of polymorphism. Secondly, ways in which different forms of polymorphism can be exploited will be examined. Finally, the notions of *type parameters* and *polymorphic types* will be described via examples written in a "model" object oriented programming language.

Inheritance and Abstract Inheritance

In our "everyday" world we are familiar with the notion of *inheritance* in a variety of different contexts, for example, *artefacts* can be inherited in the sense that the legal right to their ownership may be assigned to someone else by a Will. In addition, we are also (hopefully) familiar with the notion of biological inheritance where physical characteristics are passed from one generation to another via genes.[2]

The notion of *abstract inheritance* may be less familiar. Consider, for example, how *inheritance hierarchies* play a rôle in "knowledge representation systems" and also in object oriented programming languages. Notions such as *frames* and *is-a hierarchies*[3] play a vital rôle in knowledge representation, whilst the notions of *classes* and *inheritance* in object oriented programming languages have influenced one another.

In object oriented programming languages the notions of *classes* and *inheritance* are closely related to the notions of *type* and *polymorphism*[4] and are used to support the construction of (hopefully "more") reliable software systems[5].

Kinds of Polymorphism

As we have seen in earlier lectures *polymorphism* may be supported by a variety of different programming language paradigms. Consider, for example, the operations supported by conventional imperative programming languages for converting between simple types. Such operations are a form of polymorphism in that they allow *typing constraints* to be "relaxed", i.e. they permit *exceptions* to monomorphic typing[6]. We shall refer to support for common exceptions to monomorphic typing, e.g. type coercion, overloading[7], value sharing between types[8], as *ad-hoc polymorphism* in order to distinguish it from *universal polymorphism*, i.e. support for theoretically sound forms of polymorphism, e.g. *inclusion* and *parametric polymorphism*.

Inheritance

Inheritance (inclusion polymorphism) provides a means of organising "things", e.g. the notion of a *class* supports the description of such organisations by the "mechanism" of inheritance[9]. In the context of (software) design, inheritance provides a means of determining the *structure* of a system. Each component in such a system defines in the abstract a collection of *objects* with similar *behaviour*. Such a collection includes the collections of objects defined by its specialisations (subclasses or subtypes[10]). Subclasses (subtypes) will usually represent "smaller" collections of more precisely defined objects.

From a practical standpoint *inheritance* is a means of supporting *incremental modification*, i.e. systematic or stepwise refinement. Consider, first, the example below written in the "model" object oriented programming language. In this example, the types `true` and `false` are subtypes of the abstract (super)type `Boolean`:-

```
TYPE Boolean;
END;

TYPE true;
  SUPERTYPES Boolean;
```

```
END;

TYPE false;
  SUPERTYPES Boolean;
END;
```

1. Refining an Initial Method

The above definition contains no methods. A simple method we might consider adding by refinement is a "display" method, e.g.

```
TYPE Boolean;
END;

TYPE true;
  SUPERTYPES Boolean;
  PROCEDURE display; output.write('true');
END;

TYPE false;
  SUPERTYPES Boolean;
  PROCEDURE display; output.write(false);
END;
```

In the above example, the method `display` is only defined for objects of type `true` and `false`. Typically, we will want to manipulate objects of type `Boolean` rather than its subtypes. Given the above description, we need only define a *signature* for the `display` method in the `Boolean` type[11], e.g.

```
TYPE Boolean;
  PROCEDURE display; SKIP;
END;
```

2. Refining Further Methods

Consider, next, the addition of an additional method, `not`, to the type Boolean.

```
TYPE Boolean;

  FUNCTION not: Boolean = nil;

END;

TYPE true;

  SUPERTYPES Boolean;

  PROCEDURE display; output.write('true');

  FUNCTION not: Boolean = false;

END;

TYPE false;

  SUPERTYPES Boolean;

  PROCEDURE display; output.write(false);

  FUNCTION not: Boolea  = true;

END;
```

Again, we define a signature for the method in the abstract type in order to satisfy *static* type constraints[12]. Our model language requires all *function methods* to return a *value*, in the same way that all *procedural methods* must define a (possibly empty) *body*. The type *nil* is the functional equivalent of *SKIP*, and can be used to generate an *object* suitable for *assignment* to any function result.

3. Augmenting a subtype of an abstract supertype

In the previous examples, we augmented the type Boolean with further definitions, whose *meaning* was given in the subtypes `true` and `false`. In the example below, the additional methods `and` and `or` apply only to objects of types `true` and `false`. If the method `and` is applied to an object of type `true`, e.g.

```
true.and(x)
```

the result of the method application need only determine if the actual argument `x` was true or false, since:

and(true, x) = x

The result of the method application:-

```
false.and(x)
```

is always false, since:

and(false, x) = false

In the example below, the definition of `or` in the type `false` returns `self`, i.e. `self` denotes the *object* to which the method was applied.

```
TYPE true;

  SUPERTYPES Boolean;

  FUNCTION and(b: Boolean): Boolean = b;

  FUNCTION or(b: Boolean): Boolean = self;

END;

TYPE false;

  SUPERTYPES Boolean;

  FUNCTION and(b: Boolean): Boolean = self;

  FUNCTION or(b: Boolean): Boolean = b;

END;
```

Multiple Inheritance[13]

In principle, there is no reason why a class cannot be derived from more than one other class by inheritance. Indeed, it is possible to argue that the full "power" of object orientation cannot be exploited without support for multiple inheritance since it provides a means of combining some of the functionality (and also some of the implementation) of the classes that are inherited from.

The fundamental problem introduced by multiple inheritance is that the inheriting class has access to its superclasses without any need to *qualify* such access. Consider, for example, the type `multiple` defined below:-

```
TYPE multiple;

  SUPERTYPES Boolean, Logical;

END;
```

If the types `Boolean` and `Logical` both support a method `not` then there is a "clash of names" in the sense that it is not possible to determine which of the two "versions" of `not` is being referred to. As with other forms of ambiguity, it is necessary to resolve this situation. One means of achieving a systematic resolution is to employ the notion of *scope*[14].

Within the context of object oriented programming languages we must consider both *static* and *dynamic* binding (or *static* and *dynamic* scopes). Consider, for example, the "model" object oriented language in which *static scope* is associated with declarations at compile-time and dynamic scope is associated with method calling at run-time.

In the model language there are three kinds of named "elements", i.e. *types*, *methods* and *attributes*. Whenever named items are referenced, rules are used to determine how they are retrieved.

When elements are searched in "parent" types, there is the problem of the order in which parent types (and their parents etc.) are searched. An ordering is defined on parent types such that a a depth-first search of the supertype tree is made, searching the supertype lists in the order in which they are defined.

This is shown in the example below:

[pic]

The search order for this structure is (T0, T1, T1.1, T1.2, T2, T2.1, T3). These rules apply for any named element. An attempt to define operations on attributes, types or methods which have the same name in two or more supertype branches will be trapped by the compiler - the way only to apply operations to these language components is through renaming.

Types are searched in the following order:

1. Local Type arguments
2. Inherited Type arguments
3. Global types and variants

Attributes are searched in the following order:-

1. Local Method arguments
2. Local Method attributes
3. Local variant attributes (if any)
4. Local Type attributes
5. Inherited Type attributes
6. If no attribute is found, then the attribute is assumed to be a type generator, and hence the type scoping rules described above come into operation.

Methods are statically searched in the following order:-

1. Local Variant methods
2. Local Type methods
3. Inherited Type methods

Dynamic Scope

Methods can always be statically found using the scoping rules described above. However, at runtime, a *method call* results in a method definition being found depending on the *type* of the *attribute* to which the method is applied. The compiler will *statically check* if an attempt to call a method which is redefined in two or more different supertype branches. Hence, the *dynamic scope rules* of a method call are the same as the static scope inheritance rules defined above.

Parametric Polymorphism[15]

One means of supporting polymorphism is to allow *types* to be defined in terms of *type variables* rather than in terms of specific type. In the example below, the function abstractions `monomorphic_second_of` and `polymorphic_second_of` are defined in terms of specific types and type variables respectively in a language with a Pascal-like syntax:-

```
FUNCTION monomorphic_second_of(first: integer; second: integer): integer;
BEGIN
  monomorphic_second_of:=second
END;
```

```
FUNCTION polymorphic_second_of(first: TYPE a = any ; second: TYPE b = any): b;
BEGIN
  polymorphic_second_of:=second
END;
```

The function abstraction `polymorphic_second_of` is polymorphic in that it accepts *arguments* which are exactly those *values* that have types of the form `a` X `b`, i.e. *pairs*[16], and returns a *value* of the same type as the second argument (`any` denotes any type definable in the language).

In general, a *parameterised type* is a type that has other type(s) are parameters. In the example below, the types `monomorphic` and `polymorphic` are defined in a language with a Pascal-like syntax:-

```
TYPE monomorphic = RECORD
                     first : integer;
                     second: integer
                   END;

     polymorphic(a: TYPE) = RECORD
                              first : a;
                              second: a
                            END;
```

In this example, `(a: TYPE)` 'a' denotes an unknown type and this ensures that the identifier (`polymorphic`) denotes a parameterised type.

Type Inference

The notion of type parameterisation exploits the property that some abstraction's type is not explicitly stated but *inferred*. Where sufficient "information" is available, *type inference* may yield a *monotype*, for example, an *expression* (in a strongly-typed language) may infer a single type and this type may be used to infer a corresponding single type for some abstraction it is defined in. In general, however, there will be insufficient information because the abstraction will be defined in terms of polymorphic "components".

In the example below, a language with a Pascal-like syntax has been used to define a polymorphic list type `poly_list`:-

```
TYPE poly_list(a: any) = LIST OF a;
```

A function `cardinality` can be defined such that it relies on type inference, e.g.

```
FUNCTION cardinality(l: poly_list) = IF empty(l) THEN 0
                                     ELSE 1 + cardinality(tail(l));
```

The result type of the function `cardinality` is integer because the type expression (`IF empty(l) THEN 0 ELSE 1 + cardinality(tail(l))`) has an implied integer type.

Type Parameters and Polymorphism

In the example below, written in the "model" object oriented programming language, a type *t* takes an unconstrained *type parameter p*:

TYPE t{p};

END.

In the second example below, the type *t* takes a constrained *type parameter q*. Actual parameters for *q* must be a *subtype* of *r*:

TYPE t{q <= r};

END.

```
VAR s: list{string};

s:=empty{string} OR

s:=cons{string}('chris', empty{string});
```

In the third example below, a *parameterised polymorphic type* is declared via the IS clause:-

TYPE string_list IS list{string};

TYPE no_strings IS empty{string};

TYPE cons_string IS cons{string};

Summary and Conclusions

This lecture has identified three kinds of polymorphism, ad-hoc, inclusion and parametric polymorphism. The notion of inheritance has been shown to be a means of organising types via the concept of a class and also to be a means of supporting the incremental refinement (or modification) of a type definition. One means of removing the potential ambiguity associated with a class that inherits from more than one other class has been identified. Finally, parametric polymorphism has been examined in the context of "conventional" imperative programming languages and also object oriented programming languages.

Chris Harrison, January 1997.

----

[1] The term *polymorphism* pervades object-oriented programming and is defined in a variety of ways see, for example, Strachey, Cardelli and Wegner, Booch, Meyer, Stroustrup, and Rumbaugh. Polymorphism is usually considered the most "powerful" abstraction mechanism in an object oriented programming language. Essentially, the term *polymorphism* is used to denote the ability of an object (or reference) to assume , e.g. be "replaced by" or "become", some form other than its current form. Inheritance, in this context, specifies slightly different or additional structure or behaviour for, or of, an object. When some object of a class assumes or becomes an object of a subclass its specialised or additional attributes characterise object-oriented polymorphism.

Confusingly, inheritance as a form of polymorphism can be reasoned about as a "special case" of parametric polymorphism where an object (or reference) may assume or become any object (hopefully subject to some implicit or explicit type constraints defined by the parametric type) whose common (type) structure is abstracted over by subclass and subtype polymorphism!

[2] Arguably one of the greatest discoveries of the 20th century was the actual mechanism by which such inheritance occurs.

[3] *Inheritance networks* in knowledge representation systems are often non-monotonic, i.e. some properties may not be preserved by the introduction of additional terms. This is a consequence of combining *is_not* relations with *is_a* relations. A classic example of an inconsistency which can arise as a result of non-monotonic reasoning is an inheritance netwrok represented by the so-called "Nixon-diamond" whose interpretation is ambiguous in the sense that we cannot state definitively if Nixon is or is not a pacifist.

```
          /      \
         /        \
    Quaker      Republican
        \        /
         \      /
          Nixon
```

The meaning of the *is_a* and *is_not* relations in the knowledge representation inheritance graph shown above may also be expressed as statements in perdicate logic, e.g.

[pic]

express the relation between *Quaker* and *Republican* respectively to the predicate *Human*, and the statements:-

[pic]

introduce the predicate *Pacifist* that results in the inconsistency.

[4] In knowledge representation the "goal" is similar in the sense that semantically consistent descriptions of some "real world" domain are constructed and used to reason about properties of the "things" in that domain.

[5] The notion of *reliability* is inextricably bound up with the more problematic notion of "software quality". The ISO standard 9126 defines reliability as:-

"*A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a period of time...*"

The difficulty with such definitions is the lack of a *dimension* that enables reliable measurements to be made. One example of such a measurement is *defect density* which is defined as:-

```
                  number of known defects
defect density = _______________________

                     product size
```

In practice such measures are rather imprecise, for example, is *product size* measured in "lines of code" or in terms of some financial property, e.g. *function points* ?

On a more philosophical note, we might ask ourselves, first, what exactly software is ? Regardless of its representation, as far as I understand it, software is essentially a product of human reasoning in the sense that it embodies concious choices. If we had a precise measure of the quality of human resoning then we might have the beginnings of a similarly precise measure of so-called "software quality" ! Those students with an interest in the abstract notion of quality and hence untimately with its interpretation in the context of the development of software systems might care to read "Zen and the Art of Motorcycle Maintenance" by Robert M. Pirsig (if they have not already done so) where the notion of quality is given a truly philosophical treatment !!

[6] The notion of *strong typing* provides a means of detecting a whole class of "errors" automatically, i.e. statically at compile time, and hence runs contrary to the notion of "relaxing" typing constraints. It can be argued that the notion of string typing is one of the most important developments in programming language design.

[7] An identifier is said to be overloaded when it has different meanings in different contexts, i.e. when it has a different meaning when used with objects of different types. The arithmetic operators in programming languages are overloaded for simple types, e.g. the meaning of the expression a + b depends upon the types of the arguments a and b. Similarly, two or more function definitions may be given the same name provided that their signatures are distinct, e.g. in the number and/or types of their arguments. Consider the simple example definitions shown below:-

```
void print(int);
```

```
void print(double);
```

The two calls to the function print shown below reslove to print(int) and print(double) respectively:-

```
print(42);
print(3.14159);
```

In practice two aspects of overloaded functions must be considered, namely, how such a function is defined and how it is resolved such that the correct function is invoked. Consider, for example, how other interpretations of a function name declared twice in the same program may be construed, namely, that if the type of the value returned and the function signatures are identical all but the first definition are redeclarations.

[8] In Pascal, for example, the vlaue "nil" is universally type compatible with any pointer type and is "automatically" a value in the set of values of any pointer type.

[9] More precisely, a class provides a means of (in effect a "template" for) managing object <u>creation</u>.

[10] The terms *class* and *type* are interchangeable in the context of a language like the model language where classes are defined explicitly as type definitions. More generally, however, a class is <u>not</u> simply a type, i.e. a class is a general term denoting a specification of structure (objects), behavior (methods), and inheritance (superclasses, or recursive structure and behavior) for objects. Classes can also specify "access permissions" for "clients" and sub-classes, "visibility", etc. This is essentially a "feature-based" or "intensional" definition, emphasizing a class as a descriptor and/or constructor of objects (as opposed to a collection of objects, as with the more "classical" extensional view).

[11] The reserved word `SKIP` denotes an empty "procedure body".

[12] Statically-typed dynamic binding is found in languages such as C++, via virtual functions, and Eiffel via redefinition. In such languages, which "actual" function will be called for a "particular" function invocation at run-time must be resolved because a derived class may override the inherited function definition, in which case the overriding function must be called. Statically determining all possibilities of usage is undecidable. When the complete program is compiled, all such functions are resolved (statically) for actual objects. Object usage must have a consistent way of accessing these functions, e.g. through "tables" of function pointers in the actual objects (C++) or some equivalent mechanism, providing statically-typed dynamic binding (essentially defining simple function pointers with static type checking in the base class, and substituting these in the derived class, along with "offsets" to reset the receiver).

The run-time selection of methods is another case of dynamic binding, meaning lookup is performed or *bound* at run-time, i.e. *dynamically*. This is often desired and even required in many applications including databases, distributed programming and user interaction (e.g. GUIs).

Dynamic binding allows new objects and code to be interfaced with or added to a system without affecting existing code and reduces program complexity.

[13] In general, several distinct superclasses can declare an identical operation within a multiple inheritance hierarchy. Such ambiguity must be resolved, for example, *Eiffel* forces derived classes to rename inherited "entities". *Self* prioritizes parents. *CLOS* merges member "slots" (instance variables) with the same name into a single slot, as did the earlier *flavors*. *C++* declares an error iff a conflict arises, but a class qualifier can be used to explicitly disambiguate. Smalltalk renders same names for instance variables of subclasses illegal. Multiple-inheritance is arguably essential for modelling objects from the "real world" because such objects often belong to several classes.

[14] As you should be aware, the notion of *scope* provides a means of defining a *region of text* over which a *declaration* is effective. In Pascal, for example, the scope of each constant definition and variable declaration extends from the <u>end</u> of the definition. The scope of each type and procedure/function definition extends from the <u>beginning</u> of the group of definitions it is declared in.

[15] The term *generics* and also the term *templates*) are also used to denote types <u>parameterised</u> by types (and also functions with types). Such a facility is exploited in *parameterized classes* and *polymorphic functions* as found in languages such as Ada, C++, Eiffel, etc, although these are "syntactic" or restricted forms of the more general notion. Generics are "orthogonal" to inheritance, i.e. provide a different but

related "dimension" since types and classes may be generically parameterized.

Typically, the notion of types as parameters is constrained by the static nature of the type checking associated with their use. Functions are typically generic in statically-typed parametrically-polymorphic languages. One such popular functional language is ML, in which all functions are generic.

[16] A type of this form is called a *polytype* because it abstracts over a set of other types. In this example, "`first: TYPE a = any ; second: TYPE b = any): TYPE b`" denotes the set of types which includes all function types that accept a *pair* of values and which return, as a result, a value of the <u>same</u> type as the second argument.