UNIVERSITY OF MANCHESTER INSTITUTE OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTATION

Object Oriented Specification, Design and Implementation

Lecture 2: Abstract Data Types And Object Orientation

Introduction

'It is not our business,' he said, 'to help students to think for themselves. Surely this is the very last thing which one who wishes them well should encourage them to do. Our duty is to ensure that they shall think as we do, or at any rate, as we hold it expedient to say we do.'

The Colleges of Unreason, Chapter 22 of Erewhon by Samuel Butler.

Lecture 1 examined how a type can be described solely in terms of *operations* over the type, i.e. as an *abstract data type*. This lecture expands our understanding of the module construct used in lecture 1 by describing, first, how modules support a variety of different "styles" of use.

One particular style of use introduced in Lecture 1, i.e. modules which are parameterised by types, is shown to be the basis for object oriented programming when this style of description is combined with the notion of *inheritance*.

Styles of Module Use

Three different styles of module use are shown in the examples below. The first example demonstrates how a module can be used to form a "library" of pre-developed components which may be used by other modules or programs on the same basis as pre-defined "language" components, e.g. the type *Boolean*, I/O operations, etc.

```
MODULE example_1;

  INTERFACE

    CONST x_min = 0; x_max = 79; y_min = 0; y_max = 23;

    TYPE  abscissa = x_min..x_max; ordinate = y_min..y_max;

    PROCEDURE clear_screen;

    PROCEDURE print_string(s: string; x: abscissa; y: ordinate);

  IMPLEMENTATION

    USES system_io;

    PROCEDURE clear_screen;
    BEGIN
      (* statements to clear screen *)
    END;

    PROCEDURE print_string(* s: string; x: abscissa; y: ordinate *);
    BEGIN
      (* statements to display string *)
    END;

END.
```

Access to the components defined by the module's interface is gained by including a `USES` declaration in a program which uses the module, e.g.

```
PROGRAM module_user;
   USES example_1;
BEGIN
  .........
```

```
END.
```

Notice that the implementation part of this module is itself dependent upon another module called "system_io". A `USES` declaration placed in the implementation part in this way allows a module to access the facilities provided by "system_io" but hides those facilities from other modules or programs which use the components defined in module "example_1".

The next example demonstrates how a module can be used to *encapsulate*[1] the *state* of an "object"[2]. Variables declared in the implementation part of a module store information which can only be accessed via calls to procedures and functions defined in the interface. The designer of a module can use this technique to ensure that users only access stored information in the way that he or she intends, i.e. they cannot "break" the module by gaining direct access to its stored data.

```
MODULE memory;

  INTERFACE

    USES words;

    PROCEDURE read_memory(address  : word;
                          VAR value: word
                         );

    PROCEDURE write_memory(address: word;
                           value  : word
                          );

  IMPLEMENTATION

    USES word_operations;

    CONST max_address = 2047;

    TYPE  address     = 0..max_address;

    VAR   store       : ARRAY[address] OF word;

    PROCEDURE read_memory(* address  : word;
                            VAR value: word
                         *);
        VAR   index: integer;
    BEGIN
      index:=word_to_integer(address);
      IF index IN [0..max_address] THEN value:=store[index]
      ELSE value:=integer_to_word(0)
    END;

    PROCEDURE write_memory(* address: word;
                             value  : word
                          *);
            VAR   index: integer;
    BEGIN
      index:=word_to_integer(address);
      IF index IN [0..max_address] THEN store[index]:=value
    END;

END.
```

In this example, both the interface and the implementation depend upon resources provided by other modules. The interface is defined in terms of the type "word" which is provided by the module called "words". The implementation also makes use of "words" and a module called "word_operations" which provides a function `word_to_integer` for converting words into integers.

In lecture 1, we considered how a module may be used to provide a realisation of an abstract type.

```
MODULE stack_definition{item_type = integer};

  INTERFACE

    TYPE stack = HIDDEN;

    PROCEDURE empty;
    FUNCTION  is_empty(s: stack): Boolean;
    PROCEDURE push(item: item_type);
    FUNCTION  top: item_type;
    PROCEDURE pop;

  IMPLEMENTATION
    CONST max_cardinality = 10;
    TYPE  stack   = ^element;
          element = RECORD
                      this: item_type;
                      next: stack
                    END;
    .
    .
END.
```

This particular module, written in a hypothetical language with a Pascal-like syntax[3], defines a type `stack` and is also parameterised by a type, i.e. `item_type = integer`.

Abstract Data Types and Equational Specification

Recall, also, a third style of module use, i.e. how a module may be used to define a type solely in terms of operations over the type and not in terms of any representation type. In the example below, the type stack is defined solely in terms of the operations over the type (empty, push, pop etc) and not in terms of a representation type, e.g. the pointer type used in the example stack definition in the previous section.

```
MODULE stack_adt;
 INTERFACE
  USES item_type;
  TYPE stack;
  FUNCTION  empty: stack;
  FUNCTION  is_empty(s: stack                ): Boolean;
  FUNCTION  push    (s: stack; item: item_type): stack;
  FUNCTION  top     (s: stack                ): item_type;
  FUNCTION  pop     (s: stack                ): stack;
 SPECIFICATION
  VAR i: item_type;
      s: stack;
  EQUATIONS
   is_empty(empty_stack) = true;
   is_empty(push(s, i))  = false;
   pop(push(s, i))       = s;
   top(push(s, i))       = i;
END.
```

The notion of simple *algebraic substitution* enables this equational specification to be given a meaning. In the description of the `stack` type above, the equations indicate that their left-hand-sides are *equal* to their right-hand-sides and may, therefore, be used as *substitution rules*, i.e. whenever an *expression* is encountered which matches the *structure* of one side of a particular equation, it may be replaced by the expression which occurs on the other side of the same equation[4].

Classifications for components in algebraic descriptions can be identified, for example, *constructor functions* can be shown to be capable of representing any *expression* defined in terms of other classifications, and hence, it can be shown that constructor functions are <u>not</u> specified by any equations and consequently cannot be simplified regardless of what arguments they are applied to. The notion of refinement can be used to demonstrate how, when *designing* a new ADT, correctness and completeness are fundamental considerations:-

? An ADT's constructor functions are defined first to represent the different kinds of value which are included in the type, e.g. stacks are either empty stacks with no attributes, or push stacks with two attributes, a top of some type, and a rest of type stack

? How each function in a specification may be associated with zero or more equations, each of which defines an individual case to which the function applies. Thus, functions *may be partial* , i.e. they do not apply to all possible values of the type.

? How the result of applying a function to arguments which do not match any specified in its equations, e.g. pop(empty_stack) cannot be simplified since it is undefined. Such a function application can only be evaluated to itself, a property which is consistent with the definition of constructor functions earlier, i.e. they are not specified by any equations and cannot be simplified regardless of the arguments they are applied to.

This style of module use can also support the notion of parametric polymorphism. In the example shown below, the type stack is again parameterised by item_type.

```
 MODULE stack_adt(item_type = integer);
  INTERFACE
    TYPE stack;
    FUNCTION  empty: stack;
    FUNCTION  is_empty(s: stack               ): Boolean;
    FUNCTION  push    (s: stack; item: item_type): stack;
    FUNCTION  top     (s: stack               ): item_type;
    FUNCTION  pop     (s: stack               ): stack;
  SPECIFICATION
    VAR i: item_type;
        s: stack;
    EQUATIONS
      is_empty(empty_stack) = true;
      is_empty(push(s, i))  = false;
      pop(push(s, i))       = s;
      top(push(s, i))       = i;
END.
```

In two of the descriptions of stack types above the expression:-

item_type = integer

has enabled the definition to exploit the notion of a type as a parameter. In practice, such expressions should permit more than one possible type to be supplied as an actual parameter.

Thus for example, the expression:-

item_type ? integer

denotes a formal parameter which must be "at least" an integer, i.e. an actual parameter would have to be the type integer or any sub-type of the type integer.

In permitting such expressions we are exploiting a relationship which such expressions implicity denote, i.e. a relationship between a type (defined as a class) and it's sub-types (defined as sub-classes) which is usually termed inclusion polymorphism or inheritance.

Inheritance

Object orientation exploits the notion of *inclusion polymorphism* or *inheritance* as a means of organising descriptions independently of any considerations of a module construct and how such a construct might support the notion of parametric polymorphism.

Using inheritance a "new" class can be developed from an existing class[5] such that:-

? the subclass <u>shares</u> the behaviour of the superclass

? the subclass may <u>add</u> further "data" and "operations" to those it inherits from its superclass

? the subclass may <u>modify</u> the behaviour of one or more of the "operations" it inherits from its superclass

In a "model" object_oriented programming language we consider a class to be explicitly a <u>definition</u> of a *type*. Such types are related to each other in a *type graph* a general structure for which is shown below:

[pic]

In this diagram, the thick arrows <u>specify</u> that each type is a sub-type of the type(s) to which it points. A subtype is based on its "parent" types (known as supertypes), and includes all of the operations and attributes of its "parent "types. A subtype may also provide additional attributes and operations, and may <u>redefine</u> existing operations of its supertypes. A subtype may not redefine attributes of its parent types. An aribitrary type may inherit information from a number of supertypes, and may in turn be an "ancestor" of a number of subtypes.

Examples of two simple types defined in a "model" object-oriented programming language are shown below:-

1. A type my_window which is a subtype of type window.

```
TYPE my_window;

  SUPERTYPES window;

END.
```

2. A type document_window which is a subtype of the type window <u>and</u> the type document.

```
TYPE document_window;

  SUPERTYPES window, document;

END.
```

Consider, finally, another type document_window2, which is a type defined in terms of it's inheriting from the types window and document and also in terms of two parameters, i.e. a value (paper width) of type integer and also a type checker which must be "at least" a spell_checker:-

```
TYPE document_window2(paper_width: integer){checker <= spell_checker};


  SUPERTYPES window, document;
```

VAR my_checker: checker;

" methods are defined here"

END.

Note how this example combines parametric <u>and</u> inclusion polymorphism. We will consider further examples of types defined in this "model" language in later lectures and show how, from both object oriented designs and also object oriented specifications, such descriptions can be systematically derived.

<u>Conclusions</u>

This lecture has examined, first, how modules support a variety of different "kinds" of description, in particular how modules can be used to provide "libraries" of pre-developed components, how modules can be used to encapsulate the *state* of an *object*. and hence provide <u>realisations</u> of abstract data types, and how abstract data types may be defined algebraically.

One particular style of of description introduced in Lecture 1, i.e. modules which are parameterised by *types* has been shown to be the basis for object oriented programming when this style of description is combined with the notion of *inheritance*.

Chris Harrison, January 1996.

----

[1] A module may abstract over a single declaration, however, a module usually abstracts over a set of related declarations inlcuding types, constants, variables, procedures and functions, etc. It is in this context that a module *encapsulates* the components declared within it.

[2] The term "object" is used in its most general sense here !

[3] Several "liberties" have been taken in this example to keep its representation simple. These include the notion of "type expressions" as formal parameters in a module construct, an "extended" semantics for USES declarations, etc. In general, if an abstraction, e.g. a module abstraction, is parameterised with respect to a *value* it can use the argument value even if nothing is known about the value except its type. Similarly, if an abstraction is parameterised with respect to a *variable* it can inspect and update the argument variable even if nothing is known about the variable except its type. However, type parameters are fundamenatlly different because they denote an possibly unknown argument type, i.e. nothing useful (e.g. type checking) can be done with the type parameter until *something* is known about the argument type, more specifically, what operations are applicable to values of the argument type.

[4] Such specifications may themselves adopt a variety of different "styles". See the later lecture on object oriented specification.

[5] Different languages use different terms for these notions, see for example, *heir* and *descendent* (Meyer in "Object-Oriented Software Construction"), *base class* and *derived classes* (any text on C++), etc. Irrespective of the terms used we are either then to denote inheritance as *extension* or as *specialisation.* Inheritance viewed as specialisation corresponds to the notion of types and of subtypes (more properly sets and subsets), i.e. a rectangle is a more specialised notion than a polygon and objects of type rectangle form a subset of the objects that may be accociated with objects of type polygon. Alternatively, whan a class is viewed in the context of a "provider of services" a rectangle implements the servicves (featuers) of a polygon together with its own and in this context the "subset relationship" is inverted, i.e. the features applicable to polygon objects are a subset of those applicable rectangle objects !