UNIVERSITY OF MANCHESTER INSTITUTE OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTATION

Object Oriented Specification, Design and Implementation

Lecture 9: Object Oriented Software Specification

Introduction

This lecture is concerned with an approach to specifying the desired properties of objects. First, the notions of Abstract Data Types and their Equational (Algebraic) Specification are considered in detail, in particular how the types of operations may be classified, how certain "styles" of definition may be adopted and also some common problems encountered when developing such descriptions. Secondly, this algebraic approach is "extended" to support the notions of objects with attributes, parametric and also inclusion polymorphism. Finally, it is shown how such specifications can be translated into implementations written in the "model" object oriented programming language.

Abstract Data Types and Equational (Algebraic) Specification

Recall, first, 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, e.g. the "ADT MODULE" which defines the type `list` shown below:-

```
ADT MODULE list_definition;

    INTERFACE

      TYPE list;

      FUNCTION empty: list;
      FUNCTION is_empty(l: list): Boolean;
      FUNCTION cons(x: integer; l: list): list;
      FUNCTION head(l: list): integer;
      FUNCTION tail(l: list): list;

    SPECIFICATION

      VAR x: integer;
          l: list;

      EQUATIONS
          is_empty(empty)      = true;
          is_empty(cons(x, l)) = false;
          head(cons(x, l))     = x;
          tail(cons(x, l))     = l;

  END.
```

The equations in the specification of the type list indicate that their left-hand-sides are equal to their right-hand-sides, they 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 (school algebra). This fact may be exploited either to prove the equivalence of two expressions, or to evaluate one. This is an application of *equational logic*:-

```
EXAMPLE PROOF OF

head(tail(cons(a, cons(b, c)))) = head(cons(b, d))
WHERE a and b are integers and c and d are lists

1.     head(tail(cons(a, cons(b, c)))) = head(cons(b, c))          4th list
axiom
2.     head(cons(b, c)) = b                                        3rd list
axiom
3.     head(tail(cons(a, cons(b, c)))) = b                         from 1 and 2
and
transitivity of
                                                               =

4.     head(cons(b, d)) = b                                        3rd list
axiom
5.     head(tail(cons(a, cons(b, c)))) = head(cons(b, d))          from 3 and 4
and
transitivity of =

EXAMPLE EVALUATION OF

head(tail(cons(a, cons(head(cons(b, d)), c))))
WHERE a, b and d are integers and c and d are lists

1.     head(tail(cons(a, cons(head(cons(b, d)), c))))
        = head(cons(head(cons(b, d)), c))                          4th list
axiom
2.     head(cons(head(cons(b, d)), c)) = head(cons(b, d))          3rd list
axiom
3.     head(cons(b, d)) = b                                        3rd list
axiom
```

The facilities provided by modules like the list module are a subset of those found in specification languages such as OBJ[1] and ASF[2].

Constructor Functions

In the definition of the list ADT, "empty" and "cons" are pure constructor functions and, as such, are not defined by any equations. They represent the means of constructing instances of the type with which they are associated. Any valid expression defined in terms of ADT operations can be written in a *reduced form* which only contains constructor functions, e.g. all lists can be represented in terms of "empty" and "cons" and a set of element values.

Predicates

Predicates allow questions to be asked about the basic structure of an ADT value. As such they usually take single arguments of the type with which they are associated, and return Boolean results. The "is_empty" operation on lists is a predicate which enables an "empty" list to be distinguished from a non-empty list, i.e. a "cons" list.

Destructor Functions (or Access Operators)

These are the inverse of the constructor functions, i.e. they allow an ADT value to be dismantled into its component parts. The "head" and "tail" functions fulfil this rôle for lists.

Auxiliary Functions

Auxiliary functions are used to define calculations on ADTs. They do not add new values to the type with which they are associated, nor do they provide information about its structure which cannot be obtained by the use of the predicates and destructors, e.g. a "sort" operation on lists.

Designing New ADTs

When designing a new ADT, it is normal to define its constructor functions first to represent the different kinds of value which are included in the type, e.g. lists are either "empty" lists with no attributes, or "cons" lists with two attributes, a "head" of some type, and a "tail" of type list. The necessary predicates and destructor functions can then be defined in terms of the constructor functions. Last of all, any required auxiliary functions can be defined.

Each function may be associated with zero or more equations, each of which defines an individual case to which the function applies. Functions may be *partial*, i.e. they do not need to apply to all possible values of the type. The "head" and "tail" functions on lists are both partial functions, as neither applies to "empty" lists. The result of applying a function to arguments which do not match any specified in its equations, e.g. head(empty), cannot be simplified and is said to be *undefined*. Such a function application can only be evaluated to itself, i.e. head(empty) evaluates to head(empty). Notice that this is consistent with the definition of constructor functions. As constructor functions are not specified by any equations, they cannot be simplified, no matter what arguments they are applied to.

Enumerating Function Definitions

The definition of the "is empty" predicate for lists demonstrates that functions may be defined by a set of rules covering each of the special cases to which a function applies. Taken to the extreme, this amounts to enumerating each of the possible argument values of a function alongside the required function result. The sequence of pairs shown below defines the successor function for a small subset of the natural numbers by enumeration.

```
DEFINE succ(x: natural): natural
    x:         0   1   2   3   4   5   6   7   ...
    succ(x):   1   2   3   4   5   6   7   8   ...
```

Definition by enumeration is the basis for tables of mathematical functions, the techniques used by mathematicians to describe function values as sets of ordered pairs, and the "truth tables" used by hardware designers to document logic functions. The following example demonstrates how the type "Boolean" might be defined as an ADT using the equivalent of a truth table.

```
ADT_MODULE Boolean_definition;

  INTERFACE

    TYPE Boolean;

    FUNCTION false: Boolean;
    FUNCTION true : Boolean;
    FUNCTION not(x: Boolean): Boolean;
    FUNCTION or (x, y: Boolean): Boolean;
    FUNCTION and(x, y: Boolean): Boolean;

  SPECIFICATION

    EQUATIONS

      not(false)        = true;
      not(true)         = false;
      or (false, false) = false;
      or (false, true ) = true;
      or (true,  false) = true;
      or (true,  true ) = true;
      and(false, false) = false;
      and(false, true ) = false;
      and(true,  false) = false;
      and(true,  true ) = true;

END.
```

Defining functions in this way is similar in many ways to defining languages by enumerating the strings they contain. For all but the simplest examples, it is tedious and precludes the definition of functions on infinite types, e.g. "succ" on naturals as above. Variables are often used in equational specifications to overcome both of these problems.

The definition of the type "Boolean" may be abbreviated from 10 rules to 6 rules by using a variable of type "Boolean" to denote those argument values which, in particular rules, may be either "true" or "false". A revised specification for the "Boolean" ADT module, which makes use of a variable, is shown below.

```
VAR x: Boolean;

not(false)     = true;
not(true)      = false;
or  (false, x) = x;
or  (true,  x) = true;
and(false, x)  = false;
and(true,  x)  = x;
```

In general, the use of a variable in the left hand side of a rule is equivalent to prefixing the rule with a universal quantifier, e.g. " x: Boolean, or(false, x) = x. This is consistent with the definition of a type as a set of values. **Note:** Only variables which appear in the left hand side of a rule may appear in its right hand side.

Circular and Ambiguous Specifications

It is just as easy to write incorrect specifications, as it is to write incorrect implementations. The primary difference, is that a specification can only be incorrect with respect to the original requirements analysis, whereas an implementation can only be incorrect with respect to its specification. In addition to being incorrect, a specification may be flawed by circularity or ambiguity. A simple illustration of a circular definition is as follows.

```
or(false, x) = or(false, x);
```

Such definitions do not provide any useful information, and may cause the execution of an ADT operation not to terminate. A more subtle example involves the commutativity of "or", i.e. the fact that or(x, y) = or(y, x) for all x and y of type "Boolean". Our requirements for "Booleans" might include a statement that "or" should be commutative. If so, we might be tempted to include the following rule in the specification of the type "Boolean".

```
VAR x, y: Boolean;

or(x, y) = or(y, x);
```

Unfortunately, this kind of rule might also lead to non-termination of the evaluation of the "or" operation if the rule evaluator chose to apply the rule repeatedly, i.e. two successive substitutions based on this rule cancel each other out. This example demonstrates a distinction between the specification of the properties of a type, as might be found in a requirements definition, and the specification of the results of applying the operations of a type. Requirements definitions are usually non-executable, and there are many good reasons for thinking that they should never be executable[3]. The direct definition of the commutativity of "or" is not required to define the "Boolean" operators. Indeed, this particular property of "or" can be formally proved from the existing rules, i.e. the specification can be shown to be consistent with a requirements definition which states that "or" must be commutative.

The simplest example of an ambiguous specification involves two mutually contradictory rules as in the following example.

```
not(false) = true;
not(true)  = true;
```

Again however, more subtle variations can occur. Suppose we wish to define an auxiliary function on integers called "factorial", such that for x > 0, factorial(x) = the product of all the natural numbers between 1 and x inclusive, and factorial(0) = 1. This definition suggests the two rules shown below.

```
VAR x: integer;
factorial(0) = 1;
factorial(x) = x * factorial(x - 1);
```

The problem with this, is that both rules apply to the case where factorial is applied to 0. The intention is clearly to define factorial(0) directly, and the factorial of all other values by the second recursive rule. However, these two rules only define factorial if they are always applied in the order they are written, i.e. the least general rule first. If the rule evaluator chooses to apply the more general rule first in the evaluation of factorial(0), the evaluation will not terminate. It would therefore appear, that "factorial" cannot be specified in this way, and that some other method, which avoids non disjoint left hand sides in rules, is required. This can be accomplished by using a single rule with a conditional right hand side. Conditional expressions have a syntax which is similar to that of Pascal conditional statements, but the components following THEN and ELSE are expressions rather than statements, and the ELSE part must always be present. The condition following IF determines whether the value of the whole conditional expression is represented by the expression following THEN or the expression following ELSE. Using this notation, factorial can be defined as follows.

```
VAR x: integer;
factorial(x) = IF x = 0 THEN 1 ELSE x * factorial(x - 1);
```

However, this technique does not work very well for functions with multiple arguments. In such cases the complexity of the required conditions might well result in a loss of clarity. To overcome this problem, many specification languages allow the order in which the rule evaluator attempts to apply rules to be specified by the order in which they are written. This shorthand is adopted in the evaluation of operations specified by ADT modules. Consequently, the original two rule definition of factorial will result in satisfactory evaluation. This implies that the set of values mentioned in the implied universal quantifier, associated with each variable used in a rule, is reduced by the elements which have already been specified in earlier rules. The following definition of an auxiliary function for "greatest common divisor" exploits this shorthand notation. The implied prefix for the second rule is "" x, y: integer such that x <> y".

```
VAR x, y: integer;
gcd(x, x) = x;
gcd(x, y) = IF x > y THEN gcd(x - y, y) ELSE gcd(y, x);
```

Specifying Classes

The kinds of data types which can be specified in a language like that used in the previous sections possess no *attributes*, i.e. the types are *purely applicative* in that applications of operations over the type have no "side-effects".

Consider, next, the definition of an `account` type which describes what it "means" to be an `account` object with a single attribute `balance` shown below:-

```
CLASS account_def;
  TYPE account;
  ATTRIBUTE FUNCTIONS
    balance_of(account) -> money;
  METHOD FUNCTIONS
    credit(account X maney) -> account;
    debit (account X maney) -> account;
  AXIOMS
    VAR m: money;
    INITIAL balance_of(self) = 0 END;
    balance_of(credit(self, m)) = balance_of(self) + m;
    balance_of(debit (self, m)) = balance_of(self) - m;
```

```
END.
```

In this example, operations on objects of type `account` are separated into those which permit the state of an `account` object to be examined, i.e. attribute functions, and those which permit the state of an `account` object to be maintained, i.e. method functions. Thus, applying the attribute function `balance_of` to an object of type `account` returns the current balance "on" or "in" that account. Two method functions `credit` and `debit` provide a means of crediting and debiting amounts from an `account` (we could of course have used a single method function `credit` and permitted negative amounts to be "credited").

The attribute function `balance_of` takes an argument of type `account` and has its meaning further defined by axioms in terms of its application to some "current" object denoted by `self` within an application of the method functions `credit` and `debit`, i.e. `balance_of(credit(self, m)) = balance_of(self) + m;` and `balance_of(debit (self, m)) = balance_of(self) - m;` The variable `m` merely provides an argument of the correct type for use in these axioms.

The above example has been "simplified" somewhat as we have assumed the definition (elsewhere) of the type `money` and the operations 0 and +, -. In practice, we would need to "import" the definitions of these "entities" into the definition of the type `account`, e.g.

```
CLASS account_def;

  USES money_spec FOR
    money;
    0 -> money;
    ~+~ (money X money) -> money;
    ~-~ (money X money) -> money
  END;

  TYPE account;

  ATTRIBUTE FUNCTIONS

    balance_of(account) -> money;

  METHOD FUNCTIONS

    credit(account X maney) -> account;
    debit (account X maney) -> account;

  AXIOMS

    VAR m: money;

    INITIAL balance_of(self) = 0 END;

    balance_of(credit(self, m)) = balance_of(self) + m;
    balance_of(debit (self, m)) = balance_of(self) - m;

END.
```

The functions **`~+~(money X money) -> money;`** and **`~-~(money X money) -> money`** take a "decorated" form, i.e. their form permits them to be used as infix-like operators in expressions on the right-hand side of equations in the axioms component.

Given implementation support for the above form of description, objects of some type defined in the above style could be used within a "conventional" imperative language which has been augmented with a little "extra" semantics, e.g.

```
PROGRAM user;

    USES object_ops, money_def, account_def;

    VAR a, b: account;
        m   : money;

BEGIN
```

```
  a:=account;                           {generate an object of type account "in"
a}
  m:=money;                             {generate an object of type money "in" m}
  m.one;                                {apply the method one to the object "in"
m)
  a.credit(self, m);                    {apply the method credit to the object
"in" a
                                         using the object "in" a, i.e. self, as
first
                                         argument and object "in" m as second
argument
                                        )
  m:=a.balance_of(self);                (generate an object of type money "in" m
                                         whose "state" is that returned by the
                                         application of balance_of method applied
                                         to the object "in" a, i.e self
                                        )
  b:=a;                                 (generate an object of type account "in" b
                                         whose type is account and whose "state"
is
                                         identical to the "state" of the object
"in" a
                                        )
END.
```

The above example serves to demonstrate the "hybrid" nature of languages which support some existing programming language paradigm together with the notion of objects, i.e. from a purely "practical" perspective we need to be able to reason about the conventional semantics associated with (in the above example) an *imperative language*, and also the *object generation* and *manipulation semantics* associated with the "use" of objects within programs written in such a language.

Combining Parametric and Inclusion Polymorphism

Consider, next, how polymorphism can (arguably) be usefully combined with the style of definition used above. First, in the example below, the type `tricky` is defined such that it parameterised by a value `b` of type `Boolean` and also a type `{t}` which is "at least" (denoted by `?`) a subtype of the type `some_other`. Note, also, how the attribute function `some_att_fun` is defined in terms of its application to an object of the type `tricky` (suitably typed, i.e. `[b X {t}]`), and how the method function `some_met_fun` is defined in terms of its application to two arguments, i.e. `some_type X tricky[b X {t}]`. The axioms component defines the semantics of the attribute and method functions, i.e. objects of type `tricky` are initially in the state "`false`" and application of the method function `some_met_fun` changes the "current" state by logical negation.

```
CLASS tricky_def;
  USES other_spec FOR some_type;
       some_other FOR some_other
  END;
  TYPE tricky(b: Boolean; {t} ? some_other);
  ATTRIBUTE FUNCTIONS
    some_att_fun(tricky[b X {t}]) -> Boolean;
  METHOD FUNCTIONS
    some_met_fun(some_type X tricky[b X {t}]) -> tricky[b X {t}];
  AXIOMS
    VAR s: some_type;
    INITIAL some_att_fun(self) = false END;
    some_att_fun(some_met_fun(s, self)) = IF some_att_fun(self) THEN false
                                          ELSE true;
              END.
```

Secondly, in the example below, another definition of the type `tricky` defines the type as a *subtype* of another type via an `INHERITS` clause, i.e. `INHERITS yet_another;`

```
CLASS yet_another_def;
  TYPE yet_another(b: Boolean; {i} ? integer);
  METHOD FUNCTIONS
    do_it(yet_another[b X {i}])-> integer;
  AXIOMS
    INITIAL do_it(self) = 0 END;
END.
CLASS tricky_def;
  USES other_spec FOR some_type;
       some_other FOR some_other
  END;
  TYPE tricky;
  INHERITS yet_another;
  ATTRIBUTE FUNCTIONS
    some_att_fun(tricky[b X {t}])-> Boolean;
  METHOD FUNCTIONS
    some_met_fun(some_type X tricky[b X {t}]) -> tricky[b X {t}];
  AXIOMS
    VAR s: some_type;
    INITIAL some_att_fun(self) = false END;
    some_att_fun(some_met_fun(s, self)) = IF some_att_fun(self) THEN false
                                          ELSE true;
    do_it(self) = IF some_att_fun(self) THEN 1
                                        ELSE 0;
              END.
```

Note how the meaning of the inherited method function `do_it` in the type `tricky` is "refined" by "overriding" its initial (undefined) meaning. The constraints which must be imposed on the parameters to the inheriting type (given the parameters in its supertype) are beyond the scope of this lecture, suffice it to say that in this example the inheriting type is assumed to have the same parameters and does not refine its supertype by declaring further parameters. (The types `Boolean` and `integer` are "assumed types").

Deriving Implementations

Given a specification of some kind we must ultimately be able to *translate* it into some form of implementation. One possibility is to use term-rewriting to generate a "software component" written in some conventional programming language. In fact, the specification language used to define purely applicative types in the early examples in this lecture is an example of a language whose implementation uses term-rewriting to generate automatically an implementation written in a modular Pascal-like language. Thus, a user of the environment which supports this language has a choice, i.e. he/she can build implementations of ADT's directly or, alternatively, build specifications of ADT's and have them translated "automatically" into implementations which can then be compiled in the usual manner. (Of course, the derived implementation will not exploit any "optimisations" which a "hand-constructed" implementation could have included). Such an environment provides a basis for "rapidly prototyping" software systems using combinations of implementations and specifications.

The object-oriented specification language described in this lecture could also be supported by a term-rewriting system such that specifications written in the language (called **SODL** **S**imple **O**bject **D**escription **L**anguage) could be "mixed" with implementations written in the "model" object oriented programming language. Alternatively, a compiler for the SODL language could generate "code" which is directly executable on the virtual machine which supports the implementation of the "model" object oriented programming language.

A discussion of these two ways of implementing the SODL language is beyond the scope of this lecture course. However, it is incumbent upon the author of this lecture to demonstrate how SODL specifications can be "hand-translated" into implementations written in the "model" object oriented language.

In the example below, the specification of the type `account` is (partially) realised in the "model" object oriented programming language. The `account_user` user declares two *instance variables*, one of type `account` and the other of type `money`, and provides a means of "displaying" the "current balance" associated with its instance variable `"a"`. The procedure `initial` generates, first, an object of type `account` in `a` which is "guaranteed" to have its balance attribute initialised to `zero`. Next, the method function `credit` is applied with an appropriate argument (`a.credit(one);`). The current balance "in" the account is then returned by an application of the `balance_of` function (`m:=a.balance_of;`). Finally, a call to the `display` procedure displays the "current balance" by inspecting the *dynamic type* of the instance variable `m` which was assigned the result of the call to the `balance_of` function

```
(USE m AS money IN
        zero: self.show_string('zero', 20, 20);
        one : self.show_string('one', 20, 20)
 END
)
```

Note how the type `money` has been implemented as a purely applicative type (i.e. an ADT) via the notion of *variant subtypes*, i.e. the type `money` is an aggregation of two possible (variant) subtypes `zero` and `one`, and which supports a simple "arithmetic".

```
TYPE account_user;
    SUPERTYPES window;
    VAR a: account;
        m: money;
    PROCEDURE initial;
    BEGIN
      a:=account;
      a.credit(one);
      m:=a.balance_of;
      self.display
    END;
    PROCEDURE display;
    BEGIN
      USE m AS money IN
        zero: self.show_string('zero', 20, 20);
        one : self.show_string('one', 20, 20)
      END
    END;
END;
TYPE money;
    PROCEDURE initial; SKIP;
    VARIANT zero;
      FUNCTION plus(m1: money): money = IF m1 IS zero THEN zero
                                        ELSE one;
    VARIANT one;
     FUNCTION plus(m1: money): money = one;
END;
TYPE account;
    VAR balance: money;
    PROCEDURE initial;
    BEGIN
      balance:=zero
    END;
   PROCEDURE credit(m: money);
   BEGIN
     balance:=balance.plus(m);
   END;
   FUNCTION balance_of: money = balance;
END.
```

Appendix A provides implementations of the two other example used in this lecture.

Summary and Conclusions

This lecture has examined in detail the notions of Abstract Data Types and their Equational (Algebraic) Specification. It has been shown operations over types defined in this way may be classified; how certain "styles" of definition may be adopted; and also some common problems encountered when developing such descriptions.

This lecture has also described how an algebraic approach can be "extended" naturally[1] to support the notions of *objects* with *attributes*, *parametric* and also *inclusion polymorphism*.

Finally, this lecture has shown how such specifications can be translated into implementations written in the "model" object oriented programming language.

Chris Harrison, March 1997.

References

1. "UMIST OBJ: A Language for Executable Program Specifications", R.M. Gallimore, D. Coleman and V. Stavridou, The Computer Journal, Number 5, Volume 32, 1989.

2. "Algebraic Specification", Editors Bergstra, Heering and Klint, ACM Press Frontier Series, Addison-Wesley, 1989.

3. "An Overview of some Formal Methods for Program Design", C.A.R. Hoare, in "Essays in Computing Science", C.A.R. Hoare and C.B. Jones, Prentice Hall International series in Computer Science, 1989.

Related Reading

"Data Types and Data Structures", J.J. Martin, Prentice Hall International Series in Computer Science, 1986.

"Z in Practice", Barden, R., (et al), Prentice-Hall, 1994. in particular, pp37-41 (Object Z)

Appendix A

```
TYPE tricky_user;
   SUPERTYPES window;
   VAR t: tricky{some_type};
       b: boolean;
       s: some_type;
   PROCEDURE initial;
   BEGIN
     t:=tricky{some_type}(false);
     t.some_method(s);
     b:=t.some_attribute;
     self.display
   END;
   PROCEDURE display;
   BEGIN
     IF b THEN self.show_string('true', 20, 20)
     ELSE self.show_string('false', 20, 20)
   END;
END;
TYPE tricky{t <= some_type}(b: boolean);
    VAR state: boolean;
    PROCEDURE initial; state:=false;
    PROCEDURE some_method(s: some_type);
    BEGIN
      state:=NOT state
    END;
    FUNCTION some_attribute: boolean = state;
END;
TYPE some_type;
END.
```

```
TYPE tricky_user;
   SUPERTYPES window;
   VAR t: tricky{integer, some_type};
       b: boolean;
       s: some_type;
       i: integer;
   PROCEDURE initial;
   BEGIN
     t:=tricky{integer, some_type}(false);
     b:=t.some_attribute;
     t.some_method(s);
     i:=t.do_it;
     self.display
   END;
   PROCEDURE display;
   BEGIN
     IF i = 0 THEN self.show_string('0', 20, 20)
     ELSE self.show_string('1', 20, 20)
   END;
END;
TYPE tricky{t <= some_type}(b: boolean);
    SUPERTYPES yet_another;
    VAR state: boolean;
    PROCEDURE initial; state:=true;
    PROCEDURE some_method(s: some_type);
    BEGIN
      state:=NOT state
    END;
    FUNCTION some_attribute: boolean = state;
    FUNCTION do_it: integer =
      IF state THEN 1
      ELSE 0;
END;
TYPE yet_another{i <= integer}(b: boolean);
   VAR i: integer;
   PROCEDURE initial; i:=0;
   FUNCTION do_it: integer = 0;
END;
TYPE some_type;
END.
```

----

[1] A detailed treatment of the SODL language is beyond the scope of this lecture course as it would involve considerations of the language's *denotational*, *axiomatic* and *operational semantics*. A treatment of the

axioms permitted in the language alone is rather problematic since their *initial algebra semantics* is rather more complex than that associated with the ADT language used in earlier examples in this lecture !