UNIVERSITY OF MANCHESTER INSTITUTE OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTATION

Object Oriented Specification, Design and Implementation

Lecture 1: Types and Type Constructors

Introduction

"[Programming]... is easier to learn than piano

playing but more difficult than tic-tac-toe"

Schneiderman

The lecture has been written such that it assumes only a knowledge of a block-structured high-level programming language[1]. The style of presentation seeks to identify fundamental and general concepts and principles, and to explain these by simple examples. Where necessary, footnotes, appendices and references are included in order that terms used, and concepts and principles introduced, are clarified and may subsequently be further studied.

This lecture is concerned, first, with a notion fundamental to the development of descriptions of *computations*, i.e. the notion of *type*. Secondly, in the context of program language design, we will examine the notion of *abstraction*, in particular, how a *module construct* provides a basis for developing realisations of *types*. Finally, the notion of *abstract data types* will be examined and used in later lectures as a basis for furthering our understanding of the notions of *class* and *class instance* (or *object*.).

Background

The notion of object orientation is now widely used as a basis for developing a variety of different kinds of *description*, including specifications, designs and also implementations of software systems written in programming languages. The notion of object orientation can be placed in a variety of different contexts[2], however, one particular context enables us to see how object orientation seeks to address limitations of existing programming languages, i.e. the context of *programming language design*.

Programming languages reflect or embody a *paradigm*, i.e. a "model" against which they can be compared and contrasted. Fundamental and general paradigms include the *imperative* paradigm, the *functional* paradigm, the *declarative* paradigm and also the *concurrent* paradigm[3].

A particular programming language will embody notions drawn from at least one paradigm, for example, it may be essentially a "functional" programming language yet also embody the notion of *assignment* drawn from the imperative paradigm.

Object-oriented programming languages may be extensions or revisions of existing languages, or they may have been expressly designed "from scratch" to embody the object-oriented paradigm, i.e. the notions of *class*, *class instance* (*object*), *inclusion polymorphism* (*inheritance*), *parametric polymorphism* (*type parameterisation*), *ad-hoc* or *intersection polymorphism,* etc.

In addition to object oriented programming languages, a variety of different means of developing *software designs,* have become widely used. These design techniques seek to embody notions drawn from the object oriented paradigm. There are also object oriented *specification notation techniques* which enable software components to be given a description about which certain properties can be proved using mathematics and logic. Finally, object orientation has been employed as a means of organising the description, storage and manipulation of *persistent data*, e.g. object oriented database systems.

We will examine, next, the notion of *type* and how types are provided by programming languages and may be constructed using programming languages.

Types and Programming Languages

When we say that v is a *value* of *type* T we imply that v ? T, i.e. that a type T is a set of values. We also imply *constraints* upon the operations which may be applied to values of the type, i.e. we insist that all the values of the type exhibit uniform behaviour under operations associated with the type. Thus, *{0..9}* is a type because its values exhibit uniform behaviour under the operations +, -, DIV and MOD, but *{true, maybe, 42, fred}* is not a type in this context.

Typically, programming languages separate types into two kinds: *primitive types*, i.e. types which cannot be decomposed, and *composite* or *structured types*. Thus, a typical general-purpose programming language will provide *type constructors* for well-understood mathematical constructs including *cartesian products*, *disjoint unions*, *mappings*, *powersets* and *recursive types*, for example, the programming language Pascal supports *record*, *variant record*, *array*, *set* and *pointer* types, together with 'special purpose' types, e.g. a *file* type.

We will examine, next, the notion of *abstraction* because it is inherently a basis for the notion of one particular kind of abstraction which is of concern to programming language designers, i.e. *data abstraction*.

Abstraction and Data Abstraction

Programming languages provide *control abstractions*, e.g. *skips*, *assignments*, *procedure calls*, *sequential commands*, *collateral commands*, *conditional commands*, *iterative commands* and *block commands*, and *data abstractions*., e.g. *modules*, *objects* and *classes*.

A *module construct* is a feature of many imperative languages which seek to provide support for the systematic development or "engineering" of large-scale modular *software systems* as compositions of named, separately compilable components. A module in such languages *encapsulates* its components (*constants*, *types*, *variables*, *procedures* and *functions*) enabling a (syntactic) decomposition of a procedure hierarchy into sets of related components with a well-defined (module) interface.

Two limitations of a module construct in such languages can be identified:-

? The inability of such a construct to provide a means of ensuring that the decomposition satisfies certain formal criteria, for example, the *operations* defined in the interface are

sufficient to define a *type*

? The need to support a concrete *realisation* of a type such that *instances* of the type are created in a suitably initialised "state" and with "protection" for this encapsulated "state"

Types and Abstract Types

The notion of type as a set of values is suitable for most purposes, however, problems can arise when a "new" type is to be defined in terms of existing types since we must choose a *representation* for *values* of the new type. In some cases, the representation type may have values that do not correspond to any values of the desired type, or the representation type may have several values that correspond to the same value of the desired type. Consider, for example, the definition of the type `speed` (written in a language with a Pascal-like syntax) shown below:-

```
TYPE speed = RECORD
               distance: integer;
               time    : integer
             END;
```

This definition might usefully embody some *constraint,* for example, one which excludes those values of `distance` and `time` which have a common factor. We might represent such a constraint as:-

```
{speed(m, n) | m, n ? integer AND n > 0 AND m, n have no common factor}
```

In the example above a representation type for `speed` was defined directly in terms of a type, i.e. a set of values. Consider, next, how a type can be described indirectly by a group of *operations* rather than directly as set of values, i.e. as an *abstract type*. The definition of a "new" type as an abstract type enables undesirable properties of a representation type to be excluded. Given the realisation of the abstract type `speed` shown below, a "user" of this type can generate *values* of the type by evaluating *expressions* involving the constants `zero` and `one` and the functions `cons_speed` and `add_speed`, and two values of the type `speed` can be compared by a call to the function `speed_eq`. The implementation of the function `cons_speed` is used to constrain the value of its second argument (`time`) to be non-zero:-

```
MODULE speed_def;
  INTERFACE
    TYPE speed = HIDDEN;
    FUNCTION zero: speed;
    FUNCTION one : speed;
    FUNCTION cons_speed(m: integer; n: integer): speed;
    FUNCTION add_speed (r: speed; s: speed    ): speed;
    FUNCTION speed_eq  (r: speed; s: speed    ): Boolean;
  IMPLEMENTATION
    TYPE speed = RECORD
                   distance: integer;
                   time    : integer
                 END;
    FUNCTION zero: speed;
    BEGIN
      .
      .
    END;
    FUNCTION one: speed;
    BEGIN
      .
      .
    END;
    .
    .
    FUNCTION cons_speed(m: integer; n: integer): speed;
        VAR  r: speed;
    BEGIN
      IF n <> 0 THEN
        WITH r^ DO
        BEGIN distance:=m; time:=n END
      ELSE ...{not a speed}
    END;
    .
    .
END.
```

It is important to note that the *module construct* used above to realise the type `speed` provides a means of specifying *types* and not *type constructors*[4]. If we wish, for example, to support a type whose *elements* may be of different types a <u>separate</u> module must be provided for each. Where a whole *class* of similar objects is to be created a "generic module" is needed, i.e. a module construct which permits an *object class* to be defined such that it captures a whole class of behaviourally related types.

In the example below, a language whose syntax "resembles" the syntax of a modular Pascal-like language is used to describe the <u>class</u> **stack**[5]:-

```
CLASS_MODULE stack(item_type            : TYPE;
                   CONST max_cardinality: natural
                  );
  INTERFACE
    PROCEDURE empty;
```

```
    FUNCTION  is_empty(s: stack): Boolean;
    PROCEDURE push(item: item_type);
    FUNCTION  top: item_type;
    PROCEDURE pop;
  IMPLEMENTATION
    .
    .
END.
```

Given the above definition of a **stack** an implicit *subtyping relation*[6] enables a stack "user" to *assign* the value of a "subtype" where an instance of the "supertype" is expected. i.e. **s1:=s2** is "type safe" in the example below, but **s2:=s1** is not.

```
PROGRAM stack_user;
    .
    USES stack;
    TYPE natural = 0..maxint;
    VAR  s1      : stack[integer];
         s2      : stack[natural];
         n       : natural;
    PROCEDURE local_typed_procedure;
       VAR s: stack[boolean];
    BEGIN
      s:=s.push(true, s.empty_stack);
      .
      .
    END;
BEGIN
  s1:=s1.push(3);
  n:=3;
  s2:=s2.push(n);
  .
  .
  s1:=s2
END.
```

Although the concepts of *abstract type* and *object class* are subtly different[7] both enable the creation of *values* of a type whose *representation* is *hidden* and whose *state* can be changed only by *operations* with exclusive access.

Abstract Types and Abstract Data Types

Thus far, a type has been characterised in terms of a *set of values* and also as an *abstract type* realised by a module whose interface component defines the signature of the operations over the type and whose implementation component "hides" the definition of a representation type for the abstract type.

Where a type is defined solely in terms of operations over the type and not in terms of a representation type then it is termed an *abstract data type*. Such a type can be defined using a module construct whose implementation component is "replaced" by a specification component, for example, the type `stack` shown below:-

```
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.
```

This style of description, termed an *algebraic specification*, has the advantage that it is possible to generate an executable representation "automatically".

Conclusions

This lecture has considered, first, how a *type* can be considered solely in the context of a means of collecting together a *set* of *values*, secondly how it can be realised as an abstract type using a module construct, and thirdly as an abstract data type by specifying the meaning (semantics) of the operations over the type algebraically.

It has been shown how a *module construct* enables types to be defined but not type constructors and how, if a "class" of related types is to be defined, support for parameterisation must be provided by the module construct.

The next lecture will examine how a module construct supports a variety of different "styles" of description. One particular style of description identified in this lecture, i.e. a module which is parameterised by a type, is then 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 not unreasonable assumption since such languages have been used as a means of teaching introductory programming for many years and for good reasons.

[2] The object oriented paradigm has its conceptual basis in record structures (called *objects*) intended to be named collections of *values* (attributes) and *functions* (methods). Collections of objects form *classes* and a *subclass relation* defined on classes enables methods to work "appropriately" on all members belonging to the subclass of a given class. The first object oriented language was arguably Simula67. Many more recent languages are typed using simple extensions of the type rules for Pascal-like languages. These extensions involve principally a notion of subtyping and also more powerful type systems that elegantly incorporate *parametric polymorphism*.

[3] As stated in lecture -1, The functional and declarative paradigms seek to overcome a problem which is at the heart of Software Engineering, i.e.

"As the size of a piece of software increases the number of potential interactions between components increases exponentially until they cannot be understood, maintained or documented effectively. This effect dominates both the cost of software and also limits its applications."

Functional languages seek to overcome the problem of "doing away" with assignment (the basis of the imperative paradigm where commands update variables which may be shared) by requiring programs to be stepwise refined into hierarchies of function definitions which are ultimately used to define the expression which represents the result of the program, e.g. an integer, a file of records or a value representing an image to be displayed. Declarative languages also avoid explicit updating of variables albeit using a different technique - see Communications of ACM Vol. 28, No 12, December 1985 "Describing Prolog by its Interpretation and Compilation". Unfortunately even these "radical" attempts at changing the way the majority of programs are written have not been particularly successful in reducing the complexity of large scale software systems.

An approach which has been more successful and which is applicable to a variety of different programming language styles is to provide direct support for defining sets of constants, types, procedures and variables with well defined interfaces, i.e. *modules*.

[4] The use of the reserved word `HIDDEN` in the interface component of the type speed enables users of the module to declare *variables* of type speed and to define <u>other</u> types which have speeds as components, <u>without</u> being able to access the *implementation structure* of speeds.

[5] The type **`stack`** is parameterised by a TYPE (and also by an attribute **`CONST max_cardinality: natural`**), and the "methods", i.e. type generators and observer functions, **`push`**, **`is_empty`**, **`top`** and **`pop`** apply implicitly to an instance of the type (denoted by **`self`**).

[6] It is assumed that the type **`natural`** is a *subrange* of the discrete primitive type `integer`, i.e. that a subtype can be defined <u>as</u> a subrange.

[7] Consider, for example, how a *module* does not "create" any *variable* of the abstract type it defines, instead, it *binds* an *abstract type definition* to an encapsulated set of *bindings* such that the abstract type may be used to *declare* several distinct *variables*. Conversely, an *object class* supports <u>several</u> *objects* each of which defines several distinct *methods* which accesses a <u>distinct</u> object.