DIVISION OF COMPUTER SCIENCE

The Oberon-2 Language and Environment

Audrey Mayes
Mary Buchanan

Technical Report No.190

March 1994

The Oberon-2 Language and Environment

Audrey Mayes and Mary Buchanan

March 25, 1994

Contents

1 Introduction

This report provides an introduction to the Oberon-2 language. The language is supplied as part of the Oberon-2 system which consists of a compiler and some library modules embedded in their own environment. The embedding of a language in its own environment means that it is necessary for programmers to learn the environment as well as the language. For this reason, a brief introduction to the environment is given in section 3.

The Oberon-2 language was originally designed as part of a project to develop an object-oriented operating system and as such permits object-oriented programs to be implemented. The main features of the language are described in section 2.

2 The Oberon-2 language

Oberon-2 is a general purpose, procedural language which evolved from Modula-2 via Oberon. In common with its ancestor languages Oberon-2 has strong data typing and separately compilable modules.

The process of evolution from Modula-2 involved adding facilities to permit an object-oriented style of programming as well as the removal of some of the less essential features [1]. The most significant additional feature present in Oberon is type extension which permits the construction of new types from existing ones. Oberon-2 extends the original Oberon facilities with type-bound procedures. These are used to associate procedures with record types. Both of these new features are explained in sections 2.5.3 and 2.8.3.

A major objective, during the development of Oberon, was to keep the language simple, concise and free from superfluous constructs which would add to the complexity of either the language or the compiler. In order to reduce complexity some Modula-2 features were omitted. The omitted features include the separately compilable definition module, opaque types, subranges, enumerated types and variant records. The functionality of the variant record has been retained by the introduction of type extension.

The remainder of this section explains the constructs and facilities of the Oberon-2 language. The language is described with reference to Appendix B. This contains two modules which illustrate the basic constructs of Oberon-2. The lines of the code in the appendix have been numbered for ease of reference in the text. Line numbers are not used in Oberon-2. More information can be obtained from the reference guide for programmers which is available in the Oberon-2 system.

2.1 Program structure and system execution

A program in Oberon-2 consists of a group of interacting modules. The system does not have a main driver module or main procedure. Any procedure which does not require parameters can be used to drive the program. This makes it possible to have several points of access into a system. It is also possible for one program to instantiate other systems by calling a procedure.

2.2 Modules

The unit of compilation is the module. A module consists of

- an import list
- a collection of type, constant, variable and procedure declarations
- a sequence of statements.

The module construct is also used for information hiding. The module defines the scope of global variables and type names. Variable and type names must be exported explicitly if they are to be imported by other modules. The exported names form the interface of the module. A module's interface can contain the interface for more than one type. The types Person and Customer could be declared in a single module, in which case, the module interface would encompass the interface to both these types.

2.3 Import list

The IMPORT list names all the modules which contain procedures or types used by the current module. For example:

```
IMPORT Out, In;
```

The list shows that the whole module is imported not just specific features from a module. The required features are accessed by using the dot notation as shown below.

```
Out.String('' the name assigned to person is'');
```

The reason for importing the whole module is said to be that it is preferable to qualify each occurrence of an identifier explicitly [1]. A side effect of the explicit qualification is that the compiler is simplified.

It is possible to declare an alias for a module name in the import list. In the above example the imported modules could be declared as:

```
IMPORT O:= Out,I := In;
```

The features from the module Out would then referenced as O.featurename. For example `Out.String` would be referenced as `O.String`. This facility might be useful to emphasise the name of a procedure or type rather than its declaring module. Modules are said to be clients of modules named in the import list.

2.4 Exported features

Features which are to be exported are marked with symbols. An asterisk after a type name, procedure name or variable name marks that feature for export.

```
PROCEDURE (p : Person) AssignName* (n : ARRAY OF CHAR);
```

This export mark means that other modules have read and write access to the feature. Read only access can be granted by the - mark:

```
      name -      : ARRAY 32 OF CHAR;
```

The export marks are required because Oberon-2 does not have a separate definition module.

2.5 Types

Oberon-2 provides eight basic types and several constructs to allow the programmer to define structured types.

2.5.1 Basic types

Five of the basic types are numeric types. These numeric types are related by a set hierarchy.

$$LONGREAL \supseteq REAL \supseteq LONGINT \supseteq INTEGER \supseteq SHORTINT$$

This set hierarchy gives compatibility of a smaller type with a larger type. For example, an integer variable can be assigned to a variable of type real. The other basic types are CHAR, BOOLEAN and SET. The type SET is restricted to sets of integers.

2.5.2 Structured types

A structured type can be implemented as an array, a record or a pointer to an array or record. Pointer types in Oberon-2 can only be used with records and arrays. Records and pointers to records are used to implement the object-oriented concept of a class. Classes have procedures (methods) as well as data associated with them. The Oberon-2 mechanisms for associating procedures with data are explained in sections 2.8.3, 2.8.2 and 2.9.

The elements of an array must all be the same type but the fields of a record may be different types. The declaration of arrays, records and pointers in Oberon-2 is very similar to their declaration in Modula-2. In Oberon-2, arrays always have a lower bound of 0 so array declarations only need to specify the number of elements in an array. For example,

```
s : ARRAY 32 OF CHAR;
```

Record declarations have to mark the fields to be exported. In order to export any fields of a record, the record itself must be exported, see scope rules in section 2.6. For example, in the declaration of PersonDesc, the record type is exported even if a client module is not expected to declare variables of this type. It is usual to declare pointers to records rather than instances of the record itself. The reasons for this are explained in sections 2.7 and 2.8.1. In this example, client modules would be expected to declare variables of type Person.

```
PersonDesc* = RECORD
          name - : ARRAY 32 OF CHAR;
          address -: ARRAY 32 OF CHAR;
          telNo -   : INTEGER;
END(*RECORD*);
```

Record declarations can be empty. For example,

```
Message = RECORD END.
```

The empty record declaration is used in conjunction with type extensions which are described in the next section.

The type declaration below shows that person has been implemented as a pointer type. In this example, a record of type personDesc is bound to the pointer.

```
Person* = POINTER TO PersonDesc;
```

It is necessary to dereference a pointer in order to access the variable. The usual method is to use

```
pointerName^.field.
```

Oberon-2 allows this to be abbreviated to

```
pointerName.field
```

unless the field contains a pointer type. In this latter case, the field must be dereferenced by ^.

2.5.3 Extended types

Extended types were introduced into the language to allow new types to be based on existing types. Such types implement the object-oriented concept of inheritance. Any record type can be extended. An example of type extension can be seen below. The name of the record type being extended is declared after the keyword RECORD and is enclosed by round brackets. If, as is shown in the example, the record type being extended is declared in a different module, the type name must be prefixed with the module name.

```
Customer*  = POINTER TO CustomerDesc;
CustomerDesc* = RECORD (Persons.PersonDesc)
                  overdraftCeiling - : REAL;
END (*RECORD*);
```

In the example modules, both type Person and type Customer are implemented as pointers. Type Person points to a record of type PersonDesc. Type Customer points to a record which is an extension of type PersonDesc. The relationship between the record types pointed to is extended to the pointer types. Type Customer is therefore an extension of type Person.

If, as in the example modules, an extended record is declared in a separate module, it inherits all the fields of the base type but can access only the exported features. It cannot access the hidden fields or change read-only features. This is because a module has only one interface - defined by its exported features. Therefore, all client modules have the same access rights to a module's features. The inability of an extended type to change a read-only variable declared in its base type has been shown to cause difficulties [2].

2.6 Scope Rules

Identifiers are either predefined or are declared within a program. An identifier can refer to its associated object in those parts of the program for which its declaration is in scope.

2.6.1 General scope rule

The scope of a declaration extends from the point of declaration to the end of the block to which the declaration belongs. A block may be a record, a procedure or a module. For example, a local variable declared within a procedure is in scope for that procedure alone and a global variable declared within a module is in scope for all procedures in the module. If a procedure declares a local variable with the same identifier as a global variable, then the global variable is *not* in scope within the procedure. This use of local identifiers has the advantage that it is not necessary to know the names of all global identifiers. Local variables are only in existence while a procedure is being executed whereas a module's global variables exist until the module is unloaded from the system.

Procedures can be nested and the same scope rules apply. Hence if a nested procedure declares a variable with the same identifier as a surrounding procedure, then only the nested variable is in scope in the nested procedure. A nested procedure can use a variable declared in an outer procedure provided the nested procedure does not redeclare the variable.

There are no scope rules for nested modules since these are not permitted in Oberon-2.

2.6.2 Additional scope rules

1. Record types

   Field identifiers, for example *name: ARRAY 32 OF CHAR*, are in scope within:

   - the record definition itself,
   - field designators referring to elements of record variables.

     For example, since p: PersonDesc designates a record, p.name is a field designator which denotes the name field of p. The field identifier, name, is in scope wherever p is in scope in its declaring module.

2. Pointer types

   If an identifier is declared as T: POINTER TO T1, then the identifier T1 can be declared after the declaration of T provided that they are both declared within the same block. This is known as forward referencing.

3. Type-bound procedures

   Type-bound procedures, see section 2.8.3, are globally visible within the declaring module but from a client module perspective are considered to be part of the record to which they are bound.

4. Exported identifiers

   Identifiers within the global scope of a module may be exported. Exported identifiers may be used in any module which imports the declaring module. To use an exported identifier it is necessary to qualify the identifier by prefixing the identifier with the name of the declaring module. For example, Persons.Person denotes the identifier Person in a module importing the Persons module.

   If an exported identifier is declared as a pointer to a record, then the scope rules for record fields require that the record is also exported. For example, in the Persons module the export of Person necessitates the export of PersonDesc

TYPE Person* = POINTER TO PersonDesc;

PersonDesc* = RECORD ...

2.7 Type compatibility and assignment

Oberon-2 bases its type compatibility rules for structured types on name equivalence with the exception of open arrays and procedure types. In both these cases, a form of structural equivalence is used. Examples of this include, for open arrays, that any array of characters is compatible with an open array of characters and, for procedure types, that procedure types with the same formal parameter list and return type are compatible. Procedure types are described in section 2.8.2.

The set hierarchy of numeric types gives an alternative basis for type compatibility. This is based on inclusion, for example

SHORTINT is type compatible with INTEGER, LONGINT etc.

Type compatibility is used to determine assignment compatibility. In general, for a variable of type T1 to be assignment compatible with a variable of type T2, the types must either be the same or type T1 must be included in T2.

The general type compatibility rules must be modified to allow assignment of extended records. The modification ensures that an extended type is type compatible with its base type. For example, Customer is an extension of type Person so Customer is type compatible with Person. Type compatibility governs the assignment compatibility of variables so the following sequence of statements is valid.

```
VAR p : PersonDesc;
    c : CustomerDesc;

BEGIN
    ...

    p:=c;

    ...
END.
```

The effect of such an assignment depends on whether the variables being assigned are records or pointers. The difference is explained in the following section and is one of the reasons for using a pointer to a record for implementing objects instead of using the record type itself, as mentioned in section 2.5.2.

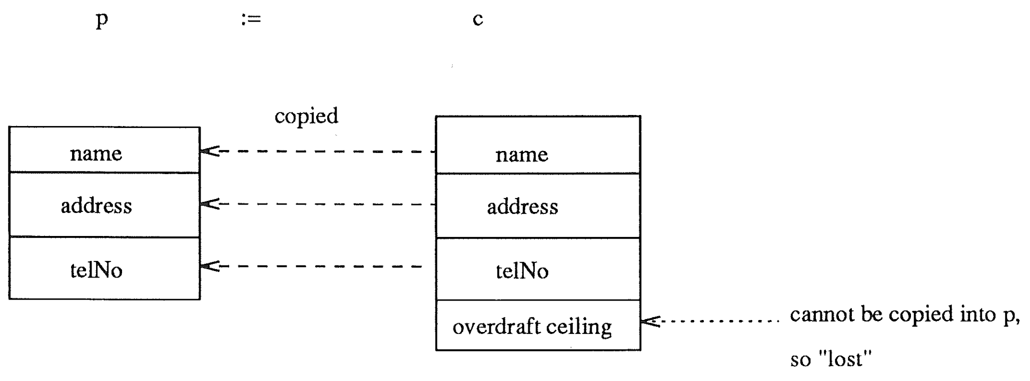

Figure 1: Assignment of Records

2.7.1 Record assignment

When the structures involved in the assignment, `p := c`, are records, the values of the fields are copied from c into p. The variable c is an instance of type CustomerDesc so contains one more field than p which is an instance of type PersonDesc. The extra field cannot be copied and so is 'lost' in the assignment to p. The original variable, c, still contains the value. This is shown in Figure 1.

When a record type is used as a variable (VAR) parameter to a procedure, the address of the variable is passed not the value. The effect of an assignment is then the same as pointer assignment.

2.7.2 Pointer assignment

Pointer variables contain the address of the actual structure. It is the address value that is changed by the assignment statement. If the assignment `p : = c` refers to pointer types, any extra fields are not lost. This is shown in figure 2. The extra fields are not accessible via the base variable because the language has static type checking. The variable p is declared to be of type Person. This means that its static type is Person. Consider the following statement sequence.

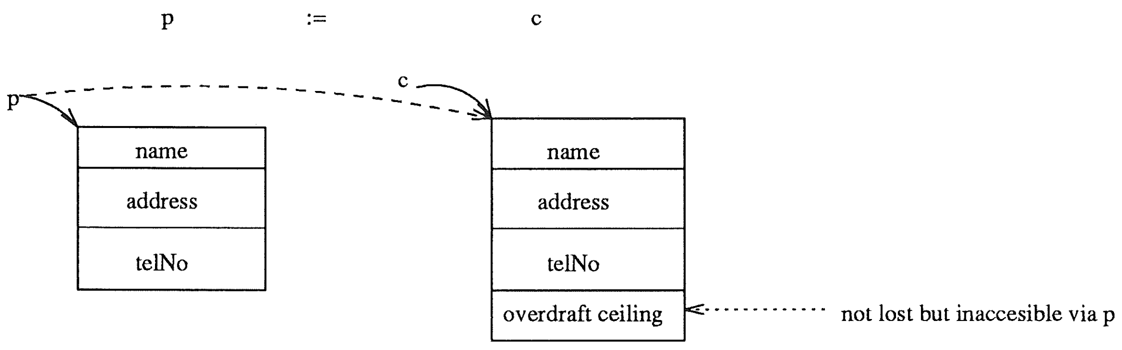

Figure 2: Assignment of Pointers

```
VAR
  p : Person;
  c : Customer;
  n : REAL;

BEGIN

  ...
  p := c    (*a valid assignment*)
  n := p.overdraftCeiling; (* invalid - type Person does not
                              contain this field*)

END.
```

The sequence is invalid because p has a static type Person so, during compilation, it does not have a field called overdraftCeiling. The assignment statement `p := c` results in p pointing to a variable of type Customer giving p the dynamic type Customer. Therefore, at run time, the variable p has the extra field of its extended type. Oberon-2 provides a method to make use of this fact. The method uses type tests or type guards to provide dynamic type checking.

2.7.3 Type tests and type guards

Type tests and type guards are used to check the dynamic type of a pointer variable or of a formal VAR parameter of a record type. A type test for the above example takes the form

```
p IS Customer
```

and asserts that p has the dynamic type Customer or an extension of type Customer. After this type test, it is possible to access the fields added by type Customer or to assign a base type variable to an extended type variable. If, at run time, p does not have the dynamic type Customer, the result is undefined, so the type test should be part of an IF statement. For example:

```
IF p IS Customer THEN
     n := p.overdraftCeiling;
END(*IF*);
```

A type guard performs a similar function. The required dynamic type is named in round brackets after the variable name. For example,

```
n := p(Customer).overdraftCeiling;
```

The two dynamic type checking mechanisms described above apply to small regions of the program. A regional type guard was added to the language to make the code clearer to read. For example,

```
WITH
     p : Person  DO (* some lines of code specific to
                       Person variables*)
   | p : Customer  DO (* some lines of code specific
                       to Customer variables*)
ELSE (* code to deal with unexpected type.*)

END
```

Type tests and guards are used during the processing of heterogeneous data structures in order to discriminate between the various types of element. They are also used to discriminate between different types of messages. Messages are described in section 2.9.

Type guards cannot be used to test the type of a record field to discriminate between various extensions of the field type.

2.8 Procedures

Procedures can take several forms in Oberon-2. These forms are proper procedures, function procedures and type-bound procedures. It is also possible to declare procedure types. Instances of procedure types are procedures. Procedure types and type-bound procedures can both be used to associate procedures with records in order to implement the objects required for object-oriented programming.

2.8.1 Proper procedures and function procedures

These types of procedure have exactly the same form and options as their Modula-2 counterparts. These procedures can have local variables, a return type and be nested inside other procedures. The return type of a function procedure must be a simple type or a pointer type. It cannot be an array or record type. This is another reason for using pointers instead of record types for implementing as mentioned in section 2.5.2.

Proper procedures and functions are bound to the module in which they are declared. They are not syntactically related to the record type on which they operate.

The following code gives an example of a proper procedure declaration.

```
PROCEDURE AssignTelNo*(p : Person; i : INTEGER);

BEGIN
  p.telNo := i;
END AssignTelNo;
```

Procedures declared in this way are statically type checked and statically bound to instances of the type. It is not possible to redefine a function or proper procedure.

The following line of code shows how to make a call to such a procedure.

```
      AssignTelNo(a,4763);
```

If this procedure is used by a client module, the procedure name must be prefixed by the module name. The above procedure call would then take the form:

```
Persons.AssignTelNo(a,4763);
```

The type compatibility rules allow this procedure to be called with a variable of type Customer as the actual parameter instead of a variable of type Person. The call takes the form:

```
Persons.AssignTelNo(b,4763);
```

The developer needs to know the name of the module which declared the base type in order to used this type of procedure. The programmer may have difficulties locating the procedure declaration when a deep hierarchy of types is developed.

2.8.2 Procedure types

The definition of a procedure type declares the parameter list and result type, if any, of the procedure. As stated previously (section 2.7), procedure types with the same signature, that is the same formal parameter list and result type, are compatible. Any procedure with a compatible signature can be assigned to a variable of a procedure type.

Procedure types can be assigned to the fields of a record type. The values of these fields define the operations which can be applied to variables of that record type. The Person type in Appendix B could have been implemented as follows:

```
TYPE
   Person = POINTER TO PersonDesc;
   PersonDesc  =RECORD
              name : ARRAY 32 OF CHAR;
              address : ARRAY 32 OF CHAR;
              telNo : INTEGER;

              assignName,assignAddress :
               PROCEDURE (p : Person; s : ARRAY 32 of CHAR);

              assignTelNo :
                    PROCEDURE (p : Person; n : INTEGER);

              print : PROCEDURE ( p : Person);
              END;
```

Actual procedures must be assigned to the procedure fields in the procedure which creates a new instance of the type. All fields of the record of a base type are inherited by extended types. Any extensions of type Person would inherit the print field and hence the ability to print themselves. Clearly, this print method would not suffice if extra fields need to be printed in the extended type. The print method could be redefined as follows:

- Write a procedure to print customer. This cannot be called print because this conflicts with the *print* field inherited from type Person.
- Write a procedure which creates a new instance of type Customer and assigns the new procedure to the print field.

The following example illustrates how these steps can be used to redefine the print method for type Customer. The type Customer is assumed to be the same as the one declared in Appendix B.

```
PROCEDURE printCustomer(c : Person);
BEGIN
    WITH c : Customer
               (* A type guard is needed to prevent the wrong
                class(!) of person being printed.*)

    DO

      (*Code to print the fields of a customer variable.*)

    END;

END printCustomer;


PROCEDURE newCustomer : Customer;

VAR
    c : Customer;


BEGIN
   NEW (c);
   c.print := printCustomer;
   RETURN c;
END newCustomer;
```

The redefined method for print is then invoked when a variable has the dynamic type Customer. The use of procedure types provides static type checking and dynamic binding of methods to variables but, as can be seen below, they are not as easy to use as type-bound procedures.

2.8.3 Type-bound procedures

Type-bound procedures were included in the language to provide a simple mechanism for the dynamic binding of procedures to variables. These procedures are bound to the record type not to an individual instance of the type and so are the equivalent of Eiffel routines or C++ class functions. Extensions of a type inherit all the type-bound procedures. The form of a type-bound procedure declaration is shown below.

```
10    PROCEDURE (p : Person) AssignName* ( n : ARRAY OF CHAR);

11    BEGIN
12      COPY (n,p.name);
13    END AssignName;
```

A variable of the type to which the procedure is bound is declared before the name of the procedure. Procedure parameters are enclosed in brackets after the procedure name as usual. Within the procedure, the fields of the variable are explicitly referenced using the dot notation, see line 12.

Type-bound procedures give static type checking and dynamic binding of the procedure to the specific instance of the type.

The following line of code shows how to invoke such a procedure.

```
78    b.AssignName(''Audrey'');
```

The required type-bound procedure is uniquely identified by the variable name and the procedure name because type-bound procedures are considered to be part of their record type.

Comparing this call with the call to a proper procedure, shows that the proper procedure needs to be identified by the module in which it is declared and the procedure name whereas the type-bound procedure call requires just the variable name and the procedure name.

If required, inherited type-bound procedures can be redefined to make them applicable to the new class, but the formal parameters must remain the same. It is possible to invoke a base type's version of a redefined type-bound procedure by appending ^ to the procedure call. A redefined type-bound procedure is shown below,

```
66    PROCEDURE (c : Customer) Print*;

67    BEGIN
68       c.Print^; (* accesses Print procedure in the base type*)
69       Out.Real(c.overdraftCeiling,8);
70       Out.Ln;
71    END Print;
```

Dynamic binding ensures that the correct version of a type-bound procedure is invoked if the dynamic type of a variable is an extension of its base type. Static type checking ensures that the procedure is available.

Type-bound procedures are also used to define the required behaviour in abstract classes. The following example of defining abstract type-bound procedures is taken from a book describing Oberon [3]. The example module implements a graphics editor. All figures will need to display common behaviour such as draw and move. The procedures for the common behaviour must be declared in the base module. The type-bound procedures to define the above behaviour are shown below.

```
PROCEDURE (f: Figure) Draw*; END Draw;

PROCEDURE (f: Figure) Move*(dX,dY : INTEGER); END Move;
```

The shape specific modules are expected to define the abstract behaviour inherited from the base type. The compiler does not check that these behaviours have been implemented so it is suggested that the abstract procedure should declare a call to the predeclared procedure HALT. This causes the program to terminate if a programmer has forgotten to implement an abstract procedure.

2.9 Message Records

Message records are a development of procedure types and are a third means of providing dynamic binding.

The implementation of message records involves several stages and requires a combination of three features which have been explained earlier. The three features are type-extension (section 2.5.3), procedure types (section 2.8.2) and type guards (section 2.7.3).

The type declaration given in section 2.8.2 is repeated here and is used to explain the implementation of message records.

```
TYPE
   Person = POINTER TO PersonDesc;
   PersonDesc  =RECORD
              name : ARRAY 32 OF CHAR;
              address : ARRAY 32 OF CHAR;
              telNo : INTEGER;

              assignName,assignAddress :
               PROCEDURE (p : Person; s : ARRAY 32 of CHAR);

              assignTelNo :
                    PROCEDURE (p : Person; n : INTEGER);

              print : PROCEDURE (p : Person);
              END;
```

It can be seen from the above example that the signatures of the procedure types all declare an instance of the type Person as a formal parameter. This commonality is exploited by message records. Message records are implemented as procedures which take two formal parameters. The first parameter is a variable of the type receiving the message, in this example p of type Person. The second parameter is a record. The fields of the record contain the data (extra parameters) required by the procedure. All the above procedure fields can be replaced by one procedure field which is usually called handler. The signature of the procedure is:

`PROCEDURE (p: Person; m: Message)`

The following code shows the resulting record declaration.

```
TYPE
    PersonDesc = RECORD
           name :ARRAY 32 OF CHAR;
           address :ARRAY 32 OF CHAR;
           telNo : INTEGER;
           handler : PROCEDURE ( p : Person; m : Message);
           END;
```

The type Message is declared as an empty record. Any operation which is required to be available on a particular type is declared as an extension of this base type. The parameters required by the operation are declared as fields of the extended record. Using Person as an example, the declarations would be:

```
TYPE   Message = RECORD END;
       assignName = RECORD(Message)
                   n : ARRAY 32 OF CHAR;
                  END;

       assignTelNo = RECORD(Message)
                    i : INTEGER;
                   END;

       assignAddress = RECORD(Message)
                    a : ARRAY 32 OF CHAR;
                    END;

       print    = RECORD (Message)
                    END;
```

A procedure with the same parameters as the handler field is written to implement the various messages. For the above example the procedure might be :

```
PROCEDURE personHandler (p :Person; VAR m : Message);

BEGIN
  WITH     (*regional type guard*)
       m(assignName) DO  ....
     | m(assignTelNo) DO ....
       (* plus any other message types*)
   END;
END personHandler;
```

The final stage is to write the procedure required to create a variable of type person with the correct handler installed. For example,

```
PROCEDURE newPerson : Person;

VAR    p : Person;
BEGIN
  NEW (p);
  p.handler := personHandler;
  RETURN p;
END newPerson;
```

The personHandler procedure can be declared in a *different* module to the type Person. The static type checking system checks that only procedures with the correct signature are assigned to the handler and that all the implemented messages are valid extensions of the declared base type, *Message*. However, it is not possible for the type checking mechanism to ensure that only messages which appear in the handler implementation are sent to a variable. It is possible therefore that a message to display overdraftCeiling could be sent to a variable of type Person above. This could cause the system to crash unless an ELSE statement is included in the handler procedure.

It can be seen above that this method of implementing methods is more complicated than using type-bound procedures or procedure types. Another disadvantage of this method of implementation is the incomplete static type checking available for messages. The main advantage of implementing methods by message records is the flexibility gained by declaring the handler in a different module to the type.

3 The Oberon-2 Environment

The dynamic model of the Oberon-2 system consists of a single process which repetitively interprets *commands*. Commands are parameterless procedures; they are the executable components of the language and are considered to be atomic actions. If a parameterless procedure P is exported from a module M, the command denoted by $M.P$ can be activated from the shell of the operating system. When $M.P$ is invoked, the module M is dynamically loaded (unless it is already in memory), P is executed and when it terminates, M remains loaded thereby saving the global state of M. When P or any other command of M is subsequently invoked, it acts on the saved state. In order to unload M from memory, deliberate action must be taken as described in section 3.11.

Commands cover a wide spectrum of actions including quitting the system and compiling a module. The system has been designed to enable users to write their own commands thereby tailoring the system to their own requirements.

In contrast to many interactive programs, commands avoid direct dialogs with the system user. Since the original Oberon language was designed for the implementation of an operating system, user input was not initially required. Consequently, Oberon does not have library modules providing procedures for input from the keyboard apart from the module Input which enables characters to be read from the keyboard. In order to input strings, it is necessary to write one's own procedures which can use the read character procedure. Data can be read from files and this is the more usual way of inputting data. The Input module also enables mouse clicks to be detected and, somewhat surprisingly perhaps, it enables the current system time to be accessed.

It might have been expected that Oberon-2, intended as a general purpose language, would provide more extensive library modules for user interaction than the original Oberon but this is not the case.

3.1 Accessing the system

Oberon-2 is available at the University of Hertfordshire on the SPARC workstations. SPARC-Oberon (TM) 2.7 (copyright ETH Zurich) covers both the programming language and the system.

In order to access the system, it is necessary to have an account on the work stations with access to X11-Windows. The following two lines should be added to the .login file:

```
set path = (/usr/local/Oberon $path)
setenv OBERON .:/usr/local/Oberon/X11:/usr/local/Oberon
```

The system can now be accessed via X-Windows by typing in the command *oberon*. To run Oberon in the background, postfix the *oberon* command with an ampersand.

The screen will be divided into two vertical columns or tracks. The left hand track is the user track and the right hand one is the system track. Both tracks may contain viewers. A viewer is a thin-framed rectangular area on the screen which can display any kind of data. At the top of the system track is the **System.Log** viewer. When a command is executed its success or failure is reported in the **System.Log** viewer. At the bottom of the system track is the **System.Tool** viewer which contains edit, compile, browser and system commands.

3.2 Viewers

Viewers have a header in the form of a title-bar. The title-bar contains the viewer title and a menu in the form of commands. The title of a viewer is the name of the opened file. The menu commands automatically refer to their own viewer. For text viewers, menu commands are included from the System tool package and the Edit tool package. For example, *System.Copy* opens a new viewer displaying the same text, *System.Grow* enables the viewer to grow to the full size of the screen and *Edit.Store* stores the text on file. *System.Close* removes the viewer, without saving the contents, or shrinks the viewer to its normal size if invoked after a call to *System.Grow*. To execute a command, place the cursor over the command and press the middle mouse button.

Viewers may be moved by pointing to the title-bar with the cursor, holding down the left mouse button and moving the cursor up or down.

Viewers have scroll bars up their left hand sides. To scan up and down a viewer, move the cursor into the scroll bar. To scroll forward, press the left mouse button, move the cursor down and release the button when the line of text which you wish to be the top line is underlined. Alternatively, move the cursor to the line of text which you want at the top and click the left mouse button.

To view the beginning of the text, click the right mouse button anywhere in the scroll bar. It may be necessary to repeat this for a large file.

A small horizontal line in the scroll bar indicates the current position of the text display. To move this, click the middle mouse button in the scroll bar at the desired location.

3.3 Basic interaction commands

Before introducing the system itself, a few comments on the use of the three button mouse and on the caret and the cursor are in order.

The cursor has the form of an arrow and is used to identify the location at which mouse clicks will be received. The caret is triangle shaped and locates the position at which text will be input.

To place the caret:

1. Move the cursor to the desired position in the text
2. Click the left mouse button.

Commands are defined as parameterless procedures and are activated by clicking the middle mouse button. Any data that is required during the execution of a command is obtained from files or viewers. If two texts are selected, the wrong one may be accessed resulting in unexpected data entry.

If a command is given a $\uparrow$ as a postfix, then the command will apply to a file which has been highlighted. For example, if a file Person.Mod is highlighted then middle clicking on the command *Edit.Open* $\uparrow$ will open the file Person.Mod.

If a command is followed by an asterisk, *, then the command will apply to a viewer which has been marked with a star marker.

To set the star marker:

1. Place the caret in the viewer by clicking the left mouse button when the cursor is in the viewer
2. Press the F1 key on the keyboard.

For example, if the Person.Mod file has been opened and the star marker set in its viewer, then the file could be printed by middle clicking on the command *Edit.Print **. Further information on printing is given below in section 3.10.

A summary of the mouse commands is shown below:

```
  Mouse button commands
  =====================

Left button    -  set caret

Middle button  -  execute commands

Right button   -  select  text
```

Sometimes it is necessary to interclick a mouse button. To do this, press and release the button while holding down another button or maybe two other buttons. Three common operations involving more than one mouse button are shown below:

```
    Interclick mouse button commands
    ================================

Hold left button              -    copies any previously
 Interclick middle button          selected text  when
                                   left button released

Hold right button             -    copies the newly
 Interclick middle button          selected text

Hold right button             -    deletes selected text
 Interclick left button
```

There is a general rule that any mouse-controlled operation that is currently underway can be cancelled by interclicking the unused mouse buttons.

3.4 Introduction to the system

At the top of the System.Tool viewer is a welcome to Oberon message followed by the information that an introduction to SPARC-Oberon is available by clicking the middle mouse button on the command *Write.Open*. This command appears to be rather fragile and sometimes results in a system violation. If this should happen, the problem can often be solved by logging in to a different SPARC workstation! Most work stations have succeeded with the *Write.Open* command at some time.

The "hello world" program

The "hello world" program can be accessed by:

1. Highlighting the words *hello.Mod* using the right mouse button
2. Clicking on the *Edit.Open* command using the middle mouse button.

The program and instructions to use it will appear in a viewer in the user track.

To set the star marker:

1. Place the caret in the hello.mod viewer by clicking the left mouse button when the cursor is in the viewer
2. Press F1 on the keyboard.

To compile the program:

1. Middle click on the command *Compiler.Compile* * \ *s* in the System.Tool viewer.

The command *Compiler.Compile* * compiles text in a viewer marked with a star marker. The extension \ s is an option which generates a new symbol file. Other options are available and can be accessed by opening the Compiler.Tool.

To run the program:

1. Middle click on *hello world*

The "hello world" message will appear in an Out.Text viewer, not in the System.Log viewer as mentioned in the instructions listed in the "hello world" program. Instruction 6 is out of date since it refers to the library module InOut. This has now been split into two modules, In and Out. All system output appears in the Out.Text viewer. If this viewer is closed, the program does not open it again. In order to view the output, the file must be reopened.

To open a new viewer:

1. Type the command *Out.Open* (anywhere)
2. Middle click on *Out.Open*.

3.5 Exiting from the system

At the bottom of the System viewer will be found the command *!System.Quit.*

To exit from the system:

1. Delete the "!"
2. Middle click on *System.Quit.*

3.6 Writing and Editing files

Editing tools are listed in the System track in the System.Tool viewer at the bottom right of the screen. A selection of commands is immediately available. To get others it is necessary either to open one of the editor tools, Edit.Tool or Write.Tool, to display the commands or to type the commands directly into a viewer.

Only a few basic editing commands are outlined below. A more extensive list of such commands can be found in the Oberon Guide which is available on the system.

To access the guide:

1. Select Write.Tool by highlighting it using the right mouse button

2. Middle click on System.Open ↑.

The file is named OberonGuide.text.

3.6.1 Creating a file

To open a new file:

1. Write a filename between the *Edit.Open* command and the ↑ symbol. (Note that upper and lower case letters are distinct.)
    - For a text file, the filename should have a .Text extension
    - For a program source file, a .Mod extension.
2. Click the middle button on the *Edit.Open* command to open the file.

A new viewer with the filename as title will appear. Place the caret before entering text; the caret is not placed automatically. It is particularly important to invoke the *Edit.Store* command before closing the viewer as there is neither automatic saving on closing nor a reminder to save. If a viewer is inadvertently closed without saving, the situation may be retrieved by invoking the command *System.Recall* which opens the most recently closed viewer. This must be done before the end of the session.

The *Edit.Store* command saves the file in the current directory. Directories can be changed without closing files so it is possible for the current dirctory not to be the one from which the file was opened. If this is the case, a file save will result in another copy of the file and the up to date version may not be in the expected directory. If the latest version of a file is not in the expected directory, it is worth searching for it in other directories.

Text viewers opened via *Edit.Open* do not provide automatic line-breaking. *Write.Open* does have automatic line-breaking but seems to have other problems associated with it such as a tendency to produce a system violation.

Once *Edit.Store* has been invoked, the saved file is in formatted form not text form. As a result, the file cannot be accessed using Unix commands such as *emacs* or *more*; all editing must be done via the Oberon system. To save files as text only, invoke the command *WriteTools.StoreAscii*. This is another fragile command.

3.6.2 Selecting and copying text

To select text:

1. Move the cursor to the beginning of the text
2. Hold the right mouse button down and drag the selection to the end.

To copy the selected text, place the caret as above but interclick the middle mouse button while still holding the left mouse button down. The selected text will be copied to the caret's position. If the caret is already in the correct position, while selecting text with the right mouse button pressed, interclick the middle mouse button. When the right button is released, the selected text will be copied to the caret position.

3.6.3 Deleting text

If the left mouse button is interclicked before releasing the right mouse button but after the whole of the required text is selected, the text will be deleted when the right button is released. (The command can be undone before releasing the right button by interclicking the middle button.)

3.6.4 Undoing commands

To undo any interclick operation, click the third button while holding down the primary button. The *Edit.Recall* command in the System.Tool viewer inserts the most recently deleted text at the position of the caret.

3.7 Compilation

To compile a program:

1. Place the cursor in the program viewer
2. Press the F1 key on the keyboard. This sets the star marker.
3. Middle click on the command *Compiler.Compile* * \ *s* in the System.Tool viewer.

The result of the compilation is shown in the log viewer in the system track at the top right of the screen. Errors are listed together with their position in the source code.

To locate the error positions:

1. Display the source program
2. Mark its viewer by placing the cursor in the viewer and pressing F1
3. Select the required position number in the system viewer
4. Invoke *Edit.Locate* in the program viewer menu bar.

   The cursor is then moved to the correct position in the source program.

A list of error codes can be obtained by accessing OberonErrors.text in System.Tool via the *Edit.Open* command.

3.8 Other system commands

A few other generally useful commands are listed below.

- A list of all the commands exported by a compiled module can be obtained by executing the command *System.ShowCommands <Module name>* . It is immaterial whether the module name inserted after *System.ShowCommands* has an extension or not.

- The global data of a module can be displayed by executing the command *System.State <Module name>*. The module name used in this command should not include the .Mod extension.

- The Oberon process can be stopped and its state preserved by invoking the command *System.Suspend*. It is quite easy to suspend the system accidentally. This can be avoided by prefixing the *System.Suspend* command with a !. The ! must be deleted before one can invoke the command.

- To execute bourne shell Unix commands, invoke *System.Execute* followed by the desired command; for example *System.Execute pwd*, *System.Execute ls*.

- The screen can be refreshed (redrawn) using Control L.

- The command *System.Quit* kills the Oberon process and removes temporary files including unsaved source files.

3.9 Directories

In the system track can be found the command *System.Directory* ↑. Below this command are file types as follows:

```
*.Mod  *.Text  *.Tool  *.Bak  *.Obj  *.Sym
```

To list all the .Mod files in the directory, highlight *.Mod using the right mouse button and middle click on *System.Directory* ↑. To open a displayed file such as Person.Mod, highlight the file name with the right button and middle click on *Edit.Open*.

The command *System.ChangeDirectory ../* can be used to go back a level in the directory hierarchy.

It is possible to customise the system viewer. For example, all your directories can be listed when entering the system. To do this, type the names of the directories you wish to access after the *System.ChangeDirectory* ↑ command and save using *Edit.Store*. To change to one of these directories, highlight it and middle click *System.ChangeDirectory* ↑. It is important that system tool changes are saved in the home directory, so that this file is accessed when the Oberon system starts up.

3.10 Printing

The print commands produce postscript output which is unsuitable for sending to the lineprinter. Postscript output is in "Adobe minimal conformat" which is useful for utility programs such as ghostview which work on postscript files.

If the work station is linked to sol, then amend the command *Edit.Print lp filename* in Edit.Tool to *Edit.Print lw filename* which directs a file to the laser printer. If the work station is not linked to sol, it is necessary to amend the *Edit.Print* command to *Edit.Print none* ↑.

There are three ways to print a file:

1. Highlight the filename

   Invoke *Edit.Print none* ↑

2. Insert the filename after *none*

   Invoke the command *Edit.Print none filename* ↑

3. Replace the ↑ with a *

   Mark the viewer with a star using the F1 key

   Click on *Edit.Print none* *.

Invoking the command *Edit.Print none* causes an Oberon.Printfile.ps to be created. This can be printed from a sol window on the workstation by typing `lpr -Plw Oberon.Printfile.ps` in the directory from which the *Edit.Print none* was invoked. Obviously this file will be overwritten by subsequent calls to the *Edit.Print* command. To save a .ps file, copy the Oberon.Printfile.ps before invoking *Edit.Print* again.

If the Write editing tool is working, then you can also print from it using *Write.Print* in the same way as *Edit.Print.*

3.11 Runtime errors

After successfully compiling a series of modules, the following run-time error may appear in the log viewer: <Module name 1> imports <Module name 2> with a bad key.

A module key represents the version of the module's interface. The above message means that Module name 1 was compiled importing a newer version of Module name 2 than the version of Module name 2 which is currently in memory. The command *System.ShowModules* can be executed to see which modules are currently loaded. An old version of a module can be removed by executing the command *System.Free <module name>*.

Whenever a module is changed and recompiled, the old version should be removed using *System.Free* since the new version is not automatically loaded.

3.12 Definition modules

Oberon-2 does not have separately compilable definition and implementation modules. However, a definition module can be generated automatically from a compiled module using the Browser tool. The command *Browser.ShowDef* ↑ can be found in the System.Tool viewer.

To obtain the definition module for a file, highlight the file name, such as Person.Mod, and middle click on *Browser.ShowDef* ↑.

The definition modules for the Person.Mod and Customer.Mod modules are shown in Appendix A.

4 Conclusions

The Oberon-2 language provides all the features required for it to be considered an object-oriented language. These features are

- encapsulation— provided by the modules and the ability to declare records with procedure fields,
- inheritance— provided by record extensions,
- polymorphism— provided by the type compatibility rules and the ability to dynamically bind code to procedure calls,
- object identity— the ability for more than one object to have the same values of state variables.

Oberon-2 is designed to be a small, simple language. The simplicity includes using conventional terms such as record and instance instead of class and object and record extension instead of inheritance.

The rules applied to record extensions enforce the concept of strict inheritance. An extended record type is always a subtype of its base record type because programmers cannot change the parameters required by a procedure and cannot change the export status of features.

It is possible to declare many types in a single module which allows flexibility to programmers. It is also possible to declare two types in one module but only export one of them for use outside that module. This allows a type to be composed of instances of hidden types. However, the hidden type is not available for reuse which is one of the benefits of object-orientation.

The simplicity of the language imposes restrictions on the developer. Oberon-2 permits only one interface to a module which means that a type can also only have one public interface. There is no mechanism to allow derived types to access hidden fields in the base type. This restriction can also reduce reusability.

Generic types are not a necessary feature of object-oriented languages but they do provide commonly used abstractions. The absence of generic types from Oberon-2 results in the need for repetitive coding.

Oberon-2 does provide one object-oriented feature which is not available in some other languages. It is possible to have many different starting points in a single system because a main program is not needed.

The Oberon environment is not very easy to use. In particular, the need to deliberately reload modules when they are recompiled is tiresome. Other features which are not completely satisfactory are the fragility of the *Write.Open* editing command, the lack of library modules for user interaction and the absence of automatic file saving. When inputting data via highlighted text in a viewer, the need to check that unwanted text is not also highlighted can prove frustrating. In addition, the mouse commands are somewhat unusual.

To summarise, Oberon-2 provides the features necessary for object-orientation in as simple a way as possible. This simplicity is masked by the effort required to come to terms with the Oberon environment.

References

[1] N. Wirth. From Modula to Oberon. *Software - Practice and experience*, 7 1988.

[2] Libero Nigro. On the type extensions of Oberon-2. *ACM SIGPLAN Notices*, 28(2), 1993.

[3] Martin Reiser and Nicholas Wirth. *Programming in Oberon.* Addison-Wesley Publishing Company, New York, 1992.

A Definition Modules

```
DEFINITION Persons;

  TYPE
    Person = POINTER TO PersonDesc;
    PersonDesc = RECORD
      name-: ARRAY 32 OF CHAR;
      address-: ARRAY 32 OF CHAR;
      telNo-: INTEGER;
      PROCEDURE (p: Person) AssignAddress (a: ARRAY OF CHAR);
      PROCEDURE (p: Person) AssignName (n: ARRAY OF CHAR);
      PROCEDURE (p: Person) Print;
    END ;


  PROCEDURE AssignTelNo (p: Person; i: INTEGER);
  PROCEDURE Test;

END Persons.
```

```
DEFINITION Customers;

  IMPORT Persons;

  TYPE
    Customer = POINTER TO CustomerDesc;
    CustomerDesc = RECORD (Persons.PersonDesc)
      overdraftCeiling-: REAL;
      PROCEDURE (c: Customer) AssignOverdraftCeiling (i: REAL);
      PROCEDURE (c: Customer) Print;
    END ;


  PROCEDURE Test;

END Customers.
```

B Example Modules

```
(*Oberon-2 reserved words are in upper case.*)

MODULE Persons;


IMPORT Out, In;


TYPE
        Person* = POINTER TO PersonDesc;
        PersonDesc* = RECORD
            name -      : ARRAY 32 OF CHAR;
            address -   : ARRAY 32 OF CHAR;
            telNo -     : INTEGER;
        END(*RECORD*);



 PROCEDURE (p : Person) AssignName* ( n : ARRAY OF CHAR);

 BEGIN
     COPY (n,p.name);
 END AssignName;


 PROCEDURE (p : Person) AssignAddress* ( a : ARRAY OF CHAR);

 BEGIN
     COPY (a,p.address);
 END AssignAddress;


 PROCEDURE AssignTelNo* (p : Person; i :INTEGER);

 BEGIN
     p.telNo := i;
 END AssignTelNo;
```

```
PROCEDURE (p : Person) Print*;

BEGIN
    Out.String(p.name);
    Out.Ln;
    Out.String (p.address);
    Out.Ln;
    Out.Int(p.telNo,6);
    Out.Ln;
END Print;


PROCEDURE Test*;

VAR
    a     : Person;
    s     : ARRAY 32 OF CHAR;
    n     : INTEGER;

BEGIN
    In.Open;
    In.String (s);                    (* takes the name
from the screen - not from the keyboard*)
    NEW(a);
    a.AssignName(s);
    a.AssignAddress("B220");
    AssignTelNo(a,4763);

    Out.String( "the name assigned to person is   ");
    Out.String(a.name);
    Out.Ln;
    Out.String( "the address assigned to person is   ");
    Out.String(a.address );
    Out.Ln;
    Out.String( "the phone number  assigned to person is   ");
    Out.Int(a.telNo,6);
    Out.Ln;


    p.Print;

END Test;
```

```
  END Persons.
```

```
Results of running the test procedure.
Persons.Test       Audrey

the name assigned to person is   Audrey
the address assigned to person is   B220
the phone number  assigned to person is     4763
Audrey
B220
  4763
```

```
MODULE Customers;

IMPORT Persons, Out;

TYPE
    Customer*       =     POINTER TO CustomerDesc;
    CustomerDesc*   =     RECORD (Persons.PersonDesc)
                              overdraftCeiling - : REAL;
                          END (*RECORD*);


PROCEDURE (c: Customer) AssignOverdraftCeiling* (i : REAL);

BEGIN
    c.overdraftCeiling := i;
END AssignOverdraftCeiling;



PROCEDURE (c : Customer) Print*;

BEGIN
    c.Print^{};
    Out.Real(c.overdraftCeiling,8);
    Out.Ln
END Print;
```

```
PROCEDURE Test*;

VAR
    a : Persons.Person;
    b : Customer;

BEGIN
    NEW(b);
    b.AssignName("Audrey");
    b.AssignAddress("B220");
    Persons.AssignTelNo( b, 4763);
    b.AssignOverdraftCeiling(1.0);
    NEW(a);
    a := b;
    a.Print;
END Test;


END Customers.
```

```
Results of running test procedure.
Customers.Test

Audrey
B220
  4763
  1.0E+00
```