OBJECT-ORIENTED PROGRAMMING IN OBERON-2

Hanspeter Mössenböck
ETH Zürich, Institut für Computersysteme

ABSTRACT

Oberon-2 is a refined version of Oberon developed at ETH. It introduces type-bound procedures, read-only export of data, and open array variables. The For statement is reintroduced. This paper concentrates on type-bound procedures which make Oberon-2 an object-oriented language with dynamically-bound messages and strong type checking at compile time. Messages can also be sent as data packets (extensible records) that are passed to a handler procedure and are interpreted at run time. This is as flexible as the Smalltalk message dispatching mechanism. Objects carry type information at run time which allows dynamic binding of messages, run time type tests, and the implementation of persistent objects. Oberon-2 is available on various machines.

OVERVIEW

In 1987, Wirth defined the language Oberon [1]. Compared with its predecessor Modula-2, Oberon is smaller and cleaner, and it supports type extension which is a prerequisite for object-oriented programming. Type extension allows the programmer to extend an existing record type by adding new fields while preserving the compatibility between the old and the new type. Operations on a type, however, have to be implemented as ordinary procedures without syntactic relation to that type. They cannot be redefined for an extended type. Therefore dynamically-bound messages (which are vital for object-oriented programming) are not directly supported by Oberon, although they can be implemented via message records (see below).

Compared to Oberon, Oberon-2 [2] provides type-bound procedures (methods), read-only export of data, and open array variables. The For statement is reintroduced after having been eliminated in the step from Modula-2 to Oberon. This paper concentrates on type-bound procedures and the use of Oberon-2 for object-oriented programming. The other facilities are described in the Oberon-2 language report.

Type-bound procedures are operations applicable to variables of a record or pointer type. They are syntactically associated with that type and can therefore easily be identified as its operations. They can be redefined for an extended type and are invoked using dynamic binding. Type-bound procedures together with type extension make Oberon-2 a true object-oriented language with dynamically-bound messages and strong type checking at compile time. Oberon-2 is the result of three years experience of using Oberon and its experimental offspring Object Oberon [3]. Object-oriented concepts were integrated smoothly into Oberon without sacrificing the conceptual simplicity of the language.

Object-oriented programming is based on three concepts: *data abstraction*, *type extension* and *dynamic binding* of a message to the procedure that implements it. All these concepts are supported by Oberon-2. We first discuss type extension since this is perhaps the most important of the three notions, and then turn to type-bound procedures, which allow data abstraction and dynamic binding.

TYPE EXTENSION

Type extension was introduced by Wirth in Oberon. It allows the programmer to derive a new type from an existing one by adding data fields to it. Consider the declarations

```
TYPE
    PointDesc      = RECORD x, y: INTEGER END;
    PointDesc3D    = RECORD (PointDesc) z: INTEGER END;

    Point          = POINTER TO PointDesc;
    Point3D        = POINTER TO PointDesc3D;

    PointXYZ       = POINTER TO PointDescXYZ;
    PointDescXYZ   = RECORD x, y, z: INTEGER END;
```

PointDesc3D is an extension of *PointDesc* (specified by the type name in parentheses that follows the symbol RECORD). It starts with the same fields as *PointDesc* but contains an additional field *z*. Conversely, *PointDesc* is called the base type of *PointDesc3D*. The notion of type extension also applies to pointers. *Point3D* is an extension of *Point* and *Point* is the base type of *Point3D*. Type extension is also called inheritance because one can think of *PointDesc3D* as "inheriting" the fields *x* and *y* from *PointDesc*.

The crucial point about type extension is that *Point3D* is compatible with *Point*, while *PointXYZ* is not (though it also points to a record with the fields *x* and *y*). If *p* is of type *Point* and *p3* is of type *Point3D* the assignment

```
p := p3
```

is legal since *p3* is an (extended) *Point* and therefore assignment compatible with *p*, which is a *Point*. The reverse assignment *p3 := p* is illegal since *p* is only a *Point* but not a *Point3D* like *p3*. The same compatibility rules apply to records.

Objects which are pointers or records have both a *static type* and a *dynamic type*. The static type is the type which the object is declared of. The dynamic type is the type which the object has at run time. It may be an extension of its static type. After the assignment *p := p3* the dynamic type of *p* is *Point3D*, while its static type is still *Point*. That means that the field *p3^.z* is still part of the block that *p* points to, but it cannot be accessed via *p* since the static type of *p* does not contain a field *p^.z* (Figure 1).

after *p := p3*

Figure 1. Assignment between the extended object and the base object

Objects like *p* are polymorphic, i.e. they may assume various types at run time. The actual type an object has at run time can be examined with a type test:

```
p IS Point3D
```

yields TRUE if the dynamic type of *p* is *Point3D* (or an extension of it) and FALSE otherwise. A type guard

```
p(Point3D)
```

asserts (i.e., tests at run time) that the dynamic type of *p* is *Point3D* (or an extension of it). If so, the designator *p(Point3D)* is regarded as having the static type *Point3D*. If not, the program is aborted. Type guards allow the treatment of *p* as a *Point3D* object. Therefore the following assignments are possible: *p(Point3D)^.z := 0; p3 := p(Point3D);*

For objects of a record type, the static and the dynamic types are usually the same. If *pd* is of type *PointDesc* and *pd3* is of type *PointDesc3D*, the assignment *pd := pd3* does not change the dynamic type of *pd*. Only the fields *pd3.x* and *pd3.y* are moved to *pd*, and the dynamic type of *pd* remains *PointDesc*. The compatibility between records is of minor importance except when *pd* is a formal variable parameter and *pd3* is its corresponding actual parameter. In this case the dynamic type of *pd* is *Point3D* and the component *pd3^.z* is not stripped off.

The motivation for type extension is that an algorithm which works with type *Point* can also work with any of its extensions. For example, the procedure

```
PROCEDURE Move (p: Point; dx, dy: INTEGER);
BEGIN INC(p.x, dx); INC(p.y, dy)
END Move;
```

can be called not only as *Move(p, dx, dy)* but also as *Move(p3, dx, dy)*.

TYPE-BOUND PROCEDURES

Type-bound procedures serve to implement abstract data types with dynamically bound operations. An abstract data type is a user-defined type which encapsulates private data together with a set of operations that can be used to manipulate this data. In Modula-2 or in Oberon an abstract data type is implemented as a record type and a set of procedures. The procedures, however, are syntactically unrelated to the record, which sometimes makes it hard to identify the data and the operations as an entity.

In Oberon-2, procedures can be connected to a data type explicitly. Such procedures are called type-bound. The interface of an abstract data type for texts may look like this:

```
TYPE
  Text = POINTER TO TextDesc;
  TextDesc = RECORD
    data: ... (*(hidden) text data*)
    PROCEDURE (t: Text) Insert (string: ARRAY OF CHAR; pos: LONGINT);
    PROCEDURE (t: Text) Delete (from, to: LONGINT);
    PROCEDURE (t: Text) Length (): LONGINT;
  END;
```

This gives a nice overview showing which operations can be applied to variables of type Text. However, it would be unwise to implement the operations directly within the record since that would clutter up the declarations with code. In fact, the above view of *Text* was extracted from the source code with a browser tool. The actual Oberon-2 program looks like this:

```
TYPE
  Text = POINTER TO TextDesc;
  TextDesc = RECORD
    data:  (*(hidden) text data*)
  END;

PROCEDURE (t: Text) Insert (string: ARRAY OF CHAR; pos: LONGINT);
BEGIN ...
END Insert;

PROCEDURE (t: Text) Delete (from, to: LONGINT);
BEGIN ...
END Delete;

PROCEDURE (t: Text) Length (): LONGINT;
BEGIN ...
END Length;
```

This notation allows the programmer to declare the procedures in arbitrary order and after the type and variable declarations, eliminating the problem of forward references. The procedures are associated with a record by the type of a special formal parameter *(t: Text)* written in front of the procedure name. This parameter denotes the operand to which the operation is applied (or the receiver of a message, as it is called in object-oriented terminology). Type-bound procedures are considered local to the record to which they are bound. In a call they must be qualified with an object of this type, e.g.

```
txt.Insert("Hello", 0)
```

We say that the message *Insert* is sent to *txt*, which is called the *receiver* of the message. The variable *txt* serves two purposes: it is passed as an actual parameter to *t* and it is used to select the procedure *Insert* bound to type *Text*.

If *Text* is extended, the procedures bound to it are automatically also bound to the extended type. However, they can be redefined by a new procedure with the same name (and the same parameter list), which is explicitly bound to the extended type. Let's assume that we want to have a more sophisticated type *StyledText* which not only maintains ASCII text but also style information. The operations Insert and *Delete* have to be redefined since they now also have to update the style data, whereas the operation *Length* is not affected by styles and can be inherited from *Text* without redefinition.

```
TYPE
  StyledText = POINTER TO StyledTextDesc;
  StyledTextDesc = RECORD (TextDesc)
    style:  ... (*(hidden) style data*)
  END;

PROCEDURE (st: StyledText) Insert (string: ARRAY OF CHAR; pos: LONGINT);
BEGIN
  ... update style data ...
  st.Insert^ (string, pos)
END Insert;

PROCEDURE (st: StyledText) Delete (from, to: LONGINT);
BEGIN
  ... update style data ...
  st.Delete^ (from, to)
END Delete;
```

We do not want to rewrite *Insert* and *Delete* completely but only want to update the style data and let the original procedures bound to *Text* do the rest of the work. In a procedure bound to type *T*, a procedure bound to the base type of *T* is called by appending the symbol ^ to the procedure name in a call (*st.Insert^ (string, pos)*).

Dynamic binding

Because of the compatibility between a type and its extensions, a variable *st* of type *StyledText* can be assigned to a variable *t* of type *Text*. The message *t.Insert* then invokes the procedure Insert which is bound to *StyledText*, although *t* has been declared of type *Text*. This is called dynamic binding: the called procedure is the one which is bound to the dynamic type of the receiver.

Polymorphism and dynamic binding are the cornerstones of object-oriented programming. They allow viewing an object as an abstract entity which may assume various concrete shapes at run time. In order to apply an operation to such an object, one does not have to distinguish between its variants. One rather sends a message telling the object what to do. The object responds to the message by invoking that procedure which implements the operation for the dynamic type of the receiver.

In Oberon-2, all type-bound procedures are called using dynamic binding. If static binding is desired (which is slightly more efficient), ordinary procedures can be used. However, one must be aware that statically-bound procedures cannot be redefined.

Information hiding

One important property of abstract data types is information hiding, i.e. the implementation of private data should not be visible to clients. It seems as if information hiding is violated in Oberon-2 since all record components can be accessed if they are qualified with an object of that record type. However, hiding components is not the business of records; it is the business of modules. A module can export record fields (and type-bound procedures) selectively. In client modules only the exported components are visible. If none of the record fields is exported the private data of the record is hidden to clients.

Notation

Object-oriented languages differ in the notations they use for classes and type-bound procedures. We want to explain why we chose the above notation in Oberon-2.

Classes. We refrained from introducing classes but rather expressed them by the well-known concept of records. Classes are record types with procedures bound to them.

Methods. Methods are expressed by type-bound procedures. The fact that their invocation is driven by the dynamic type of an object is reflected by the qualification with an explicit receiver parameter. In a call, the actual receiver is written in front of the message name (*x.P*); therefore the formal receiver is also declared in front of the procedure name (*PROCEDURE (x: T) P (...)*).

We refrained from duplicating the headers of bound procedures in record declarations as it is done in C++ [6] and Object-Pascal [8]. This keeps declarations short and avoids unpleasant redundancy. Changes to a procedure header would otherwise have to be made at two places and the compiler would have to check the equality of the headers. If the programmer wants to see the record together with all its procedures, he uses a browser to obtain the information. We believe that the working style of programmers has changed in recent years. Programs are written more interactively and high performance tools can be used to collect information that had to be written down explicitly in former days.

The procedures bound to a type can be declared in any order. They can even be mixed with procedures bound to a different type. If procedures had to be declared within a type declaration, indirect recursion between procedures bound to different types would make awkward forward declarations necessary for one-pass compilation.

Receiver. In most object-oriented languages the receiver of a message is passed as an implicit parameter that can be accessed within a method by a predeclared name such as *self* or *this*. The data of a class can be accessed in a method without qualification. For example, in C++ the method *Delete* would look like this:

```
void Text::Delete (int from, to) {
    length = length - (to-from);     // field length of the receiver is accessed without qualification
    ... NotifyViews(this) ...     // receiver is accessed with the predeclared name this
}
```

We believe that it is better to declare the receiver explicitly, which allows the programmer to choose a meaningful name for it (not just "this"). The implicit passing of the receiver seems to be a little bit mysterious. We also believe that it contributes to the clarity of programs if fields of the receiver must always be qualified with the receiver's name. This is especially helpful if fields are accessed which are declared in the receiver's base type. In Oberon-2, *Delete* is therefore written in the following way:

```
PROCEDURE (t: Text) Delete (from, to: LONGINT);
BEGIN
    t^.length := t^.length - (to-from);     (* length is explicitly qualified with t *)
    ... NotifyViews(t) ...     (* receiver has the user-defined name t *)
END Delete;
```

MESSAGE RECORDS

Type-bound procedures are one way to implement messages. Another way is to take the phrase "sending a message" literally and to view a message as a packet of data that is sent to an object. This requires message records of various kinds and lengths and a handler per object that accepts all these message records. Type-extension provides these two mechanisms. Messages are extensible records and the handler is a procedure which takes a message as a parameter and interprets it according to the dynamic type of the message.

Consider a graphics editor. The objects in this application are various kinds of figures (rectangles, circles, lines, etc.) and the operations are drawing, moving, and filling the figures. For every operation a message record is declared which contains the arguments of the message as record fields:

```
TYPE
    Message     = RECORD END;
    DrawMsg     = RECORD (Message) END;
    MoveMsg     = RECORD (Message) dx, dy: INTEGER END;
    FillMsg     = RECORD (Message) pat: Pattern END;
```

Next, the type *Figure* is declared, which contains the handler as a procedure variable:

```
TYPE
    Figure = POINTER TO FigureDesc;
    FigureDesc = RECORD
        x, y, width, height: INTEGER;
        handle: PROCEDURE (f: Figure; VAR m: Message)
    END;
```

The handler has two parameters: the receiver of the message (which is a *Figure* here) and the message itself. Since *m* is of type *Message*, all message types that are derived from it (*DrawMsg*, *MoveMsg*, etc.) are compatible. Note, that *m* is a variable parameter, so it may have a dynamic type which is an extension of its static type *Message*. Every extension of *Figure* (i.e., *Rectangle*, *Circle*, *Line*) has its own handler that is installed in objects of this type. The handler for rectangle objects might look like this:

```
PROCEDURE HandleRect (f: Figure; VAR m: Message);
BEGIN
    WITH
        m(DrawMsg) DO ... draw the rectangle f ...
    |   m(MoveMsg) DO ... move the rectangle f by (m.dx, m.dy) ...
    |   m(FillMsg) DO ... fill the rectangle f with m.pat ...
    ELSE (* ignore the message *)
    END
END HandleRect;
```

The With statement is a regional type guard. It has been extended in Oberon-2 to accept multiple variants. The above With statement should be read as follows: if the dynamic type of *m* is *DrawMsg*, then the statement sequence following the first DO symbol is executed and a type guard *m(DrawMsg)* is implicitly applied to every occurrence of *m*; else if the dynamic type of *m* is *MoveMsg*, then the statement sequence following the second DO symbol is executed where every occurrence of *m* is regarded as a *MoveMsg*; and so on. If no variant matches and if no else part is specified program execution is aborted. Using objects of type *Figure* requires the following actions:

```
VAR f: Figure; r: Rectangle; move: MoveMsg;

NEW(r); r^.handle := HandleRect;     (*initialize the object by installing the rectangle handler*)
... f := r ...
move.dx := ...; move.dy := ...;    (*set up the message record*)
f.handle(f, move);     (*send the message*)
        (*possibly retrieve output arguments from the message record*)
```

The use of message records has both advantages and disadvantages.

Advantages

- The message can be stored in a variable and can be sent later on.
- The same message can easily be distributed to more than one object (*message broadcast*). Consider the case where all figures have to be moved. With type-bound procedures, the caller would have to traverse the list of figures and send a *Move* message to every figure:

```
f := firstFigure; WHILE f # NIL DO f.Move(dx, dy); f := f^.next END
```

The structure of the figure list must be known to the caller (which is not always the case) and the code for the list traversal is duplicated in every client. With message records one can implement the list traversal in a procedure *Broadcast* to which the message is passed as a parameter:

```
PROCEDURE (lst: List) Broadcast (VAR m: Message);
    VAR f: Figure;
BEGIN
    f := lst^.first; WHILE f # NIL DO f.handle(f, m); f := f^.next END
END Broadcast;
```

This allows hiding the list structure and keeping the code for the list traversal in a single place.
- An object can be sent a message which it does not understand. It may ignore the message or delegate it to another object. For example, a *Fill* message can be broadcast to all figures although only rectangles and circles understand it, but not lines. With type-bound procedures this is not possible because the compiler checks if a message is understood by the receiver.
- The handler can be replaced at run time, changing the behaviour of an object.
- Message records can be declared in different modules. This allows adding new messages to a type when a new module is written.

Disadvantages

- It is not immediately clear which operations belong to a type, i.e. which messages an object understands. To find that out, one has to know which handler is installed at run time and how this handler is implemented.
- The compiler cannot check if a message is understood by an object. Faulty messages can be detected only at run time and may go undetected for months.

- Messages are interpreted by the handler at run time and in sequential order. This is much slower than the dynamic binding mechanism of type-bound procedures, which requires only a table lookup with a constant offset. Message records are much like messages in Smalltalk [7], which are also interpreted at run time.
- Sending a message (i.e., filling and unfilling message records) is somewhat clumsy.

In general, type-bound procedures are clearer and type-safe, while message records are more flexible. One should use type-bound procedures whenever possible. Message records should only be used where special flexibility is needed, e.g., for broadcasting a message or for cases where it is important to add new messages to a type later without changing the module that declares the type.

PERSISTENT OBJECTS

Our implementation of Oberon-2 allows persistent objects. An object is called persistent if it outlives the program which created it. To make an object persistent, it must be possible to write it to a file and to reconstruct it from that external format. In Oberon-2, every record object carries a descriptor of its dynamic type. Among other things this descriptor contains the type name as a pair (module name, type name). It is possible to implement a procedure *GetName* which returns the type name of a given object, and a procedure *New* which creates and returns an object of a type specified by a type name.

```
DEFINITION Objects;
    PROCEDURE GetName(object: Object; VAR typeName: ARRAY OF CHAR);
    PROCEDURE New(typeName: ARRAY OF CHAR; VAR object: Object);
END Objects.
```

If *x* is an extension of *Object* and understands a *Load* and a *Store* message, procedures to externalize and internalize *x* are (a *Rider* is a position in a file and is used to read and write data)

```
PROCEDURE WriteObject(VAR r: Files.Rider; x: Object);
    VAR name: ARRAY 64 OF CHAR;
BEGIN
    Objects.GetName(x, name);
    i := -1; REPEAT INC(i); Files.Write(r, name[i]) UNTIL name[i] = 0X;
    IF x # NIL THEN x.Store(r) END  (* store fields of x to r *)
END WriteObject;

PROCEDURE ReadObject(VAR r: Files.Rider; VAR x: Object);
    VAR name: ARRAY 64 OF CHAR;
BEGIN
    i := -1; REPEAT INC(i); Files.Read(r, name[i]) UNTIL name[i] = 0X;
    Objects.New(name, x);
    IF x # NIL THEN x.Load(r) END  (* read fields of x from r *)
END ReadObject;
```

More details on persistent objects as well as on optimization aspects can be found in [5].

IMPLEMENTATION

In order to support object-oriented programming certain information about objects must be available at run time: The dynamic type of an object is needed for type tests and type guards. A table with the addresses of the type-bound procedures is needed for calling them using dynamic binding. Finally, the Oberon system has a garbage collector which needs to know the locations of pointers in dynamically allocated records. All this information is stored in so-called type descriptors of which there is one for every record type at run time.

The dynamic type of a record corresponds to the address of its type descriptor. For dynamically allocated records this address is stored in a so-called type tag which precedes the actual data and which is invisible for the programmer. If *f* is of dynamic type *Rectangle* (an extension of *Figure*), the run-time data structures are shown in Figure 2.

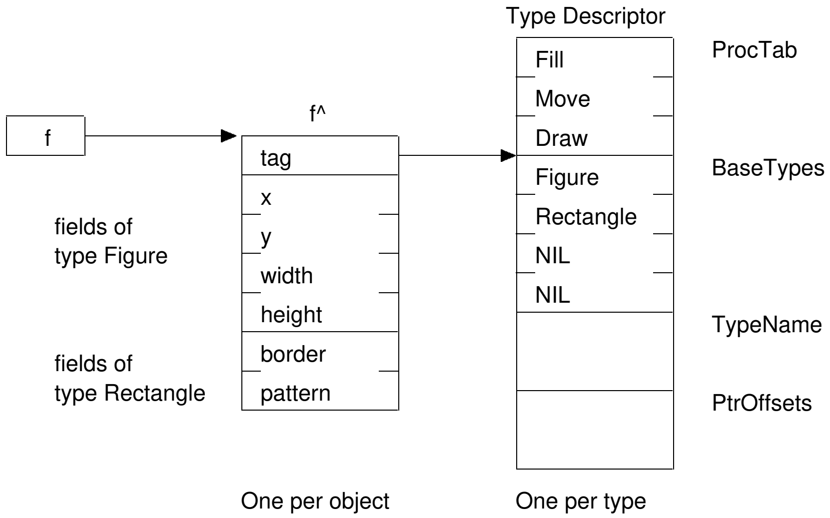

Figure 2. A variable *f* of dynamic type *Rectangle*, the record *f* points to, and its type descriptor

Since both the table of procedure addresses and the table of pointer offsets must have a fixed offset from the type descriptor address, and since both may grow when the type is extended and further procedures or pointers are added, the tables are located at the opposite ends of the type descriptor and grow in different directions.

A message *v.P* is implemented as *v^.tag^.ProcTab[Index-of-P]*. The procedure table index *Indexp* is known for every type-bound procedure *P* at compile time. A type test of the form *v IS T* is translated into *v^.tag^.BaseTypes[ExtensionLevel-of-T] = TypDescAdrT*. Both the extension level of a record type and the address of its type descriptor are known at compile time. For example, the extension level of *Figure* is 0 (it has no base type), and the extension level of *Rectangle* is 1.

Type-bound procedures cause no memory overhead in objects (the type tag was already needed in Oberon-1). They cause only minimal run-time overhead compared to ordinary procedures. On a Ceres computer (NS32532 processor) a dynamically-bound procedure call is less than 10 % slower than a statically-bound call [3]. Measured over a whole program this difference is insignificant.

More details on the implementation of Oberon, particularly on the garbage collector, can be found in [4] and [5].

AVAILABILITY

Oberon-2 was developed on the Ceres computer and has been ported to several other machines. Currently it is available on Sun's SparcStation, on Digital's DECstation, and on IBM's RS/6000. The compiler and the whole Oberon system (garbage collection, command activation, dynamic loading, etc.) is available from ETH without charge. It can be obtained via anonymous internet file transfer ftp (hostname: neptune.inf.ethz.ch, internet address: 129.132.101.33, directory: Oberon).

ACKNOWLEDGEMENTS

Oberon-2 is the result of many discussions among the members of our institute. It was particularly influenced by the ideas of N.Wirth, J.Gutknecht, and J.Templ. The compiler and the system were ported to other machines by R.Crelier, J.Templ, M.Franz, and M.Brandis.

REFERENCES

1. Wirth, N "The Programming Language Oberon" Software Practice and Experience, Vol 18, No 7, (July 1988), pp 671-690.
2. Mössenböck, H "The Programming Language Oberon-2" Computer Science Report 160, ETH Zürich (May 1991).
3. Mössenböck, H and Templ, J "Object Oberon - A Modest Object-Oriented Language" Structured Programming, Vol 10, No 4 (1989), pp 199-207.
4. Wirth, N and Gutknecht, J "The Oberon System" Computer Science Report 88, ETH Zürich (1988).
5. Pfister, C and Heeb, B and Templ, J "Oberon Technical Notes" Computer Science Report 156, ETH Zürich (March 1991).
6. Stroustrup, B "The C++ Programming Language" Addison-Wesley (1986).
7. Goldberg, A and Robson, D "Smalltalk-80, The Language and its Implementation", Addison-Wesley (1983).
8. Tesler, L "Object-Pascal" Structured Language World, Vol 9, No 3, (1985).