From Modula to Oberon

N. Wirth

Abstract

The programming language Oberon is the result of a concentrated effort to increase the power of Modula-2 and simultaneously to reduce its complexity. Several features were eliminated, and a few were added in order to increase the expressive power and flexibility of the language. This paper describes and motivates the changes. The language is defined in a concise report.

Introduction

The programming language Oberon evolved from a project whose goal was the design of a modern, flexible, and efficient operating system for a single-user workstation. A principal guideline was to concentrate on properties that are genuinely essential and - as a consequence - to omit ephemeral issues. It is the best way to keep a system in hand, to make it understandable, explicable, reliable, and efficiently implementable.

Initially, it was planned to express the system in Modula-2 [1] (subsequently called Modula), as that language supports the notion of modular design quite effectively, and because an operating system has to be designed in terms of separately compilable parts with conscientiously chosen interfaces. In fact, an operating system should be no more than a set of basic modules, and the design of an application must be considered as a goal-oriented extension of that basic set: Programming is always extending a given system.

Whereas modern languages, such as Modula, support the notion of extensibility in the procedural realm, the notion is less well established in the domain of data types. In particular, Modula does not allow the definition of new data types as extensions of other, programmer-defined types in an adequate manner. An additional feature was called for, thereby giving rise to an *extension* of Modula.

The concept of the planned operating system also called for a highly dynamic, centralized storage management relying on the technique of garbage collection. Although Modula does not prevent the incorporation of a garbage collector in principle, its variant record feature constitutes a genuine obstacle. As the new facility for extending types would make the variant record feature superfluous, the removal of this stumbling block was a logical decision. This step, however, gave rise to a *restriction* (subset) of Modula.

It soon became clear that the rule to concentrate on the essential and to eliminate the inessential should not only be applied to the design of the new system, but equally stringently to the language in which the system is formulated. The application of the principle thus led from Modula to a new language. However, the adjective "new" has to be understood in proper context: Oberon evolved from Modula by very few additions and several subtractions. In relying on evolution rather than revolution we remain in the tradition of a long development that led from Algol to Pascal, then to Modula-2, and eventually to Oberon. The common traits of these languages are their procedural rather than functional model and the strict typing of data. Even more fundamental, perhaps, is the idea of abstraction: the language must be defined in terms of mathematical, abstract concepts without reference to any computing mechanism. Only if a language satisfies this criterion, can it be called "higher-level". No syntactic coating whatsoever can earn a language this attribute alone.

The definition of a language must be coherent and concise. This can only be achieved by a careful choice of the underlying abstractions and an appropriate structure combining them. The language manual must be reasonably short, avoiding the explanation of individual cases derivable from the general rules. The power of a formalism must not be measured by the length of its description. To the contrary, an overly lengthy definition is a sure symptom of inadequacy. In this respect, not complexity but simplicity must be the goal.

In spite of its brevity, a description must be complete. Completeness is to be achieved within the framework of the chosen abstractions. Limitations imposed by particular implementations do not belong to a language definition proper. Examples of such restrictions are the maximum values of numbers, rounding and truncation errors in arithmetic, and actions taken when a program violates the stated rules. It should not be necessary to supplement a language definition with a voluminous standards document to cover "unforeseen" cases.

But neither should a programming language be a mathematical theory only. It must be a practical tool. This imposes certain limits on the terseness of the formalism. Several features of Oberon are superfluous from a purely theoretical point of view. They are nevertheless retained for practical reasons, either for programmers' convenience or to allow for efficient code generation without the necessity of complex, "optimizing" pattern matching algorithms in compilers. Examples of such features are the presence of several forms of repetitive statements, and of standard procedures such as INC, DEC, and ODD. They complicate neither the language conceptually nor the compiler to any significant degree.

These underlying premises must be kept in mind when comparing Oberon with other languages. Neither the language nor its defining document reach the ideal; but Oberon approximates these goals much better than its predecessors.

A compiler for Oberon has been implemented for the NS32000 processor family and is embedded in the Oberon operating environment [8]. The compiler requires less than 50 KByte of memory, consists of 6 modules with a total of about 4000 lines of source code, and compiles itself in about 15 seconds on a workstation with a 25MHz NS32532 processor.

After extensive experience in programming with Oberon, a revision was defined and implemented. The differences between the two versions are summarised towards the end of the paper. Subsequently, we present a brief introduction to (revised) Oberon assuming familiarity with Modula (or Pascal), concentrating on the added features and listing the eliminated ones. In order to be able to start with a clean slate, the latter are taken first.

Features omitted from Modula

Data types

Variant records are eliminated, because they constitute a genuine difficulty for the implementation of a reliable storage management system based on automatic garbage collection. The functionality of variant records is preserved by the introduction of extensible data types.

Opaque types cater for the concept of abstract data type and information hiding. They are eliminated as such, because again the concept is covered by the new facility of extended record types.

Enumeration types appear to be a simple enough feature to be uncontroversial. However, they defy extensibility over module boundaries. Either a facility to extend given enumeration types has to be introduced, or they have to be dropped. A reason in favour of the latter, radical solution was the observation that in a growing number of programs the indiscriminate use of enumerations (and subranges) had led to a type explosion that contributed not to program clarity but rather to verbosity. In connection with import and export, enumerations give rise to the exceptional rule that the import of a type identifier also causes the (automatic) import of all associated constant identifiers. This exceptional rule defies conceptual simplicity and causes unpleasant problems for the implementor.

Subrange types were introduced in Pascal (and adopted in Modula) for two reasons: (1) to indicate that a variable accepts a limited range of values of the base type and to allow a compiler to generate appropriate guards for assignments, and (2) to allow a compiler to allocate the minimal storage space needed to store values of the indicated subrange. This appeared desirable in connection with packed records. Very few implementations have taken advantage of this space saving facility, because the additional compiler complexity is very considerable. Reason 1 alone, however, did not appear to provide sufficient justification to retain the subrange facility in Oberon.

With the absence of enumeration and subrange types, the general possibility of defining *set types* based on given element types appeared as redundant. Instead, a single, basic type SET is introduced, whose values are sets of integers from 0 to an implementation-defined maximum.

The basic type *CARDINAL* had been introduced in Modula in order to allow address arithmetic with values from 0 to 2^{16} on 16-bit computers. With the prevalence of 32-bit addresses in modern processors, the need for unsigned arithmetic has practically vanished, and therefore the type CARDINAL has been eliminated. With it, the bothersome incompatibilities of operands of types CARDINAL and INTEGER have disappeared.

Pointer types are restricted to be bound to a record type or to an array type.

The notion of a definable index type of arrays has also been abandoned: All indices are by default integers. Furthermore, the lower bound is fixed to 0; array declarations specify a number of elements (length) rather than a pair of bounds. This break with a long standing tradition since Algol 60 clearly demonstrates the principle of eliminating the inessential. The specification of an arbitrary lower bound hardly provides any additional expressive power. It represents a rather limited kind of mapping of indices which introduces a hidden computational effort that is incommensurate with the supposed gain in convenience. This effort is particularly heavy in connection with bound checking and with dynamic arrays.

Modules and import/export rules

Experience with Modula over the last eight years has shown that *local modules* were rarely used. Considering the additional complexity of the compiler required to handle them, and the additional complications in the visiblity rules of the language definition, the elimination of local modules appears justified.

The *qualification* of an imported object's identifier x by the exporting module's name M, viz. M.x, can be circumvented in Modula by the use of the import clause FROM M IMPORT x. This facility has also been discarded. Experience in programming systems involving many modules has taught that the explicit qualification of each occurrence of x is actually preferable. A simplification of the compiler is a welcome side-effect.

The dual role of the main module in Modula is conceptually confusing. It constitutes a *module* in the sense of a package of data and procedures enclosed by a scope of visibility, and at the same time it constitutes a single *procedure* called main program. A module is composed of two textual pieces, called the definition part and the implementation part. The former is missing in the case of a main program module.

By contrast, a module in Oberon is in itself complete and constitutes a unit of compilation. Definition and implementation parts are merged; names to be visible in client modules, i.e. exported identifiers, are marked, and they typically precede the declarations of objects not exported. A compilation generates in general a changed object file and a new symbol file. The latter contains information about exported objects for use in the compilation of client modules. The generation of a new symbol file must, however, be specifically enabled by a compiler option, because it will invalidate previous compilations of clients.

The notion of a main program has been abandoned. Instead, the set of modules linked through imports typically contains (parameterless) procedures. They are to be considered as individually activatable, and they are called *commands.* Such an activation has the form M.P, where P denotes the command and M the module containing it. The effect of a command is considered - not like that of a main program as accepting input and transforming it to output - as a change of state represented by global data.

Statements

The *with statement* of Modula has been discarded. Like in the case of imported identifiers, the explicit qualification of field identifiers is to be preferred. Another form of with statement is introduced; it has a different function and is called a regional guard (see below).

The elimination of the for statement constitutes a break with another long standing tradition. The baroque mechanism of Algol 60's for statement had been trimmed significantly in Pascal (and Modula). Its marginal value in practice has led to its absence from Oberon.

Low-level facilities

Modula makes access to machine-specific facilities possible through low-level constructs, such as the data types ADDRESS and WORD, absolute addressing of variables, and type casting functions. Most of them are packaged in a module called SYSTEM. These features were supposed to be rarely used and easily visible through the presence of the identifier SYSTEM in a module's import list. Experience has revealed, however, that a significant number of programmers import this module quite indiscriminately. A particularly seductive trap are Modula's type transfer functions.

It appears preferable to drop the pretense of portability of programs that import a "standard", yet system-specific module. *Type transfer functions* denoted by type identifiers are therefore eliminated, and the module SYSTEM is restricted to providing a few machine-specific functions that typically are compiled into inline code. The types *ADDRESS* and *WORD* are replaced by the type BYTE, for which type compatibility rules are relaxed. Individual implementations are free to provide additional facilities in their module SYSTEM. The use of SYSTEM declares a program to be patently implementation-specific and thereby non-portable.

Concurrency

The system Oberon does not require any language facilities for expressing concurrent processes. The pertinent rudimentary features of Modula, in particular the coroutine, were therefore not retained. This exclusion is merely a reflection of our actual needs within the concrete project, but not on the general relevance of concurrency in programming.

Features introduced in Oberon

In contrast to the number of eliminated features, there are only a few new ones. The important new concepts are type extension and type inclusion. Furthermore, open arrays may have several dimensions (indices), whereas in Modula they were confined to a single dimension.

Type extension

The most important addition is the facility of extended record types. It permits the construction of new types on the basis of existing types, and establishes a certain degree of compatibility between the new and old types. Assuming a given type

```
T    = RECORD x, y: INTEGER END
```

extensions may be defined which contain certain fields in addition to the existing ones. For example

```
T0   = RECORD (T)  z: REAL END
T1   = RECORD (T)  w: LONGREAL END
```

define types with fields x, y, z and x, y, w respectively. We define a type declared by

```
T'   = RECORD (T) <field definitions> END
```

to be a *(direct) extension* of T, and conversely T to be the *(direct) base type* of T'. Extended types may be extended again, giving rise to the following definitions:

A type T' is an *extension* of T, if T' = T or T' is a direct extension of an extension of T. Conversely, T is a *base type* of T', if T = T' or T is the direct base type of a base type of T'. We denote this relationship by T' → T.

The rule of assignment compatibility states that values of an extended type are assignable to variables of their base types. For example, a record of type T0 can be assigned to a variable of the base type T. This assignment involves the fields x and y only, and in fact constitutes a *projection* of the value onto the space spanned by the base type.

It is important to allow modules which import a base type to be able to declare extended types. In fact, this is probably the normal usage.

This concept of extensible data type gains importance when extended to pointers. It is appropriate to say that a pointer type P' bound to T' extends a pointer type P, if P is bound to a base type T of T', and to extend the assignment rule to cover this case. It is now possible to form data structures whose nodes are of different types, i.e. inhomogeneous data structures. The inhomogeneity is automatically (and most sensibly) bounded by the fact that the nodes are linked by pointers of a common base type.

Typically, the pointer fields establishing the structure are contained in the base type T, and the procedures manipulating the structure are defined in the same (base) module as T. Individual extensions (variants) are defined in client modules together with procedures operating on nodes of the extended type. This scheme is in full accordance with the notion of system extensibility: new modules defining new extensions may be added to a system without requiring a change of the base modules, not even their recompilation.

As access to an individual node via a pointer bound to a base type provides a projected view of the node data only, a facility to widen the view is necessary. It depends on the ability to determine the actual type of the referenced node. This is achieved by a *type test,* a Boolean expression of the form

```
t IS T'        (or   p IS P')
```

If the test is affirmative, an assignment t' := t (t' of type T') or p' := p (p' of type P') should be possible. The static view of types, however, prohibits this. Note that both assignments violate the rule of assignment compatibility. The desired assignment is made possible by providing a *type guard* of the form

```
t' := t(T')     (p' := p(P'))
```

and by the same token access to the field z of a T0 (see previous examples) is made possible by a type guard in the designator t(T0).z. Here the guard asserts that t is (currently) of type T0. In analogy to array bound checks and case selectors, a failing guard leads to program abortion.

Whereas a guard of the form t(T) asserts that t is of type T for the designator (starting with) t only, a *regional type guard* maintains the assertion over an entire sequence of statements. It has the form

```
WITH t: T DO StatementSequence END
```

and specifies that t is to be regarded as of type T within the entire statement sequence. Typically, T is an extension of the declared type of t. Note that assignments to t within the region therefore require the assigned value to be (an extension) of type T. The regional guard serves to reduce the number of guard evaluations.

As an example of the use of type tests and guards, consider the following types Node and Object defined in a module M:

```
TYPE Node =   POINTER TO Object;
     Object = RECORD key, x, y: INTEGER;
                left, right: Node
              END
```

Elements in a tree structure anchored in a variable called root (of type Node) are searched by the procedure *element* defined in M.

```
PROCEDURE element(k: INTEGER): Node;
  VAR p: Node;
BEGIN p := root;
  WHILE (p # NIL) & (p.key # k) DO
    IF p.key < k THEN p := p.left ELSE p := p.right END
  END ;
  RETURN p
END element
```

Let extensions of the type Object be defined (together with their pointer types) in a module M1 which is a client of M:

```
TYPE Rectangle =    POINTER TO RectObject;
     RectObject =   RECORD (Object) w, h: REAL END ;
     Circle =       POINTER TO CircleObject;
     CircleObject = RECORD (Object) rad: REAL; shaded: BOOLEAN END
```

After the search of an element, the type test is used to discriminate between the different extensions, and the type guard to access extension fields. For example:

```
p := M.element(K);
IF p # NIL THEN
    IF p IS Rectangle THEN  ...  p(Rectangle).w  ...
    ELSIF (p IS Circle) & ~p(Circle).shaded THEN  ...  p(Circle).rad  ...
    ELSIF  ...
```

The extensibility of a system rests upon the premise that new modules defining new extensions may be added without requiring adaptations nor even recompilation of the existing parts, although components of the new types are included in already existing data structures.

The type extension facility not only replaces Modula's variant records, but represents a type-safe alternative. Equally important is its effect of relating types in a type hierarchy. We compare, for example, the Modula types

```
T0'  =   RECORD t: T;  z: REAL END ;
T1'  =   RECORD t: T;  w: LONGREAL END
```

which refer to the definition of T given above, with the extended Oberon types T0 and T1 defined above. First, the Oberon types refrain from introducing a new naming scope. Given a variable r0 of type T0, we write r0.x instead of r0.t.x as in Modula. Second, the types T, T0', and T1' are distinct and unrelated. In contrast, T0 and T1 are related to T as extensions. This becomes manifest through the type test, which asserts that variable r0 is not only of type T0, but also of base type T.

The declaration of extended record types, the type test, and the type guard are the only additional features introduced in this context. A more extensive discussion is provided in [2]. The concept is very similar to the class notion of Simula 67 [3], Smalltalk [4], Object Pascal [5], C++ [6], and others, where the properties of the base class are said to be *inherited* by the derived classes. The class facility stipulates that all procedures applicable to objects of the class be defined together with the data definition. This dogma stems from the notion of abstract data type, but it is a serious obstacle in the development of large systems, where the possibility to add further procedures defined in additional modules is highly desirable. It is awkward to be obliged to redefine a class solely because a method (procedure) has been added or changed, particularly when this change requires a recompilation of the class definition and of all its client modules.

We emphasise that the type extension facility - although gaining its major role in connection with pointers to build heterogeneous, dynamic data structures as shown in the example above - also applies to statically declared objects used as variable parameters. Such objects are allocated in a workspace organized as a stack of procedure activation records, and therefore take advantage of an extremely efficient allocation and deallocation scheme.

In Oberon, procedure *types* rather than procedures (methods) are connected with objects in the program text. The binding of actual methods (specific procedures) to objects (instances) is delayed until the program is executed. The association of a procedure type with a data type occurs through the declaration of a record field. This field is given a procedure type. The association of a method - to use Smalltalk terminology - with an object occurs through the assignment of a specific procedure as value to the field, and not through a static declaration in the extended type's definition which then "overrides" the declaration given in the base type. Such a procedure is called a *handler.* Using type tests, the handler is capable of discriminating among different extensions of the record's (object's) base type. In Smalltalk, the compatibility rules between a class and its subclasses are confined to pointers, thereby intertwining the concepts of access method and data type in an undesirable way. In Oberon, the relationship between a type and its extensions is based on the established mathematical concept of projection.

In Modula, it is possible to declare a pointer type within an implementation module, and to export it as an opaque type by listing the same identifier in the corresponding definition module. The net effect is that the type is exported while all its properties remain hidden (invisible to clients). In Oberon, this facility is generalized in the sense that the selection of the record fields to be exported is arbitrary and includes the cases all and none. The collection of exported fields defines a partial view - a *public projection* - to clients.

In client modules as well as in the module itself, it is possible to define extensions of the base type (e.g. TextViewers or GraphViewers). Of importance is also the fact that non-exported components (fields) may have types that are not exported either. Hence, it is possible to hide certain data types effectively, although components of (opaquely) exported types refer to them.

Type inclusion

Modern processors feature arithmetic operations on several number formats. It is desirable to have all these formats reflected in the language as basic types. Oberon features five of them:

LONGINT, INTEGER, SHORTINT (integer types)
LONGREAL, REAL (real types)

With the proliferation of basic types, a relaxation of compatibility rules among them becomes almost mandatory. (Note that in Modula the numeric types INTEGER, CARDINAL, and REAL are incompatible). To this end, the notion of *type inclusion* is introduced: a type T includes a type T', if the values of type T' are also values of type T. Oberon postulates the following hierarchy:

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

The assignment rule is relaxed accordingly: A value of type T' can be assigned to a variable of type T, if T' is included in T (or if T' extends T), i.e. if $T \supseteq T'$ or $T' \rightarrow T$. In this respect, we return to (and extend) the flexibility of Algol 60. For example, given variables

i: INTEGER; k: LONGINT; x: REAL

the assignments

k := i; x := k; x := 1; k := k+i; x := x*10 + i

conform to the rules, whereas the statements i := k; k := x are not acceptable. x := k may involve truncation.

The presence of several numeric types is evidently a concession to implementations which can allocate different amounts of storage to variables of the different types, and which thereby offer an opportunity for storage economization. This practical aspect should - with due respect for mathematical abstraction - not be ignored. The notion of type inclusion minimises the consequences for the programmer and requires only few implicit instructions for changing the data representation, such as sign extensions and integer to floating-point conversions.

Differences between Oberon and Revised Oberon

A revision of Oberon was defined after extensive experience in the use and implementation of the language. Again, it is characterized by the desire to simplify and integrate. The differences between the original version [7] and the revised version [9] are the following:

1. Definition and implementation parts of a module are merged. It appeared as desirable to have a module's specification contained in a single document, both from the view of the programmer and the compiler. A specification of its interface to clients (the definition part) can be derived automatically. Objects previously declared in the definition part (and repeated in the implementation part), are specially marked for export. The need for a structural comparison of two texts by the compiler thereby vanishes.

2. The syntax of lists of parameter types in the declaration of a procedure type is the same as that for regular procedure headings. This implies that dummy identifiers are introduced; they may be useful as comments.

3. The rule that type declarations must follow constant declarations, and that variable declarations must follow type declarations is relaxed.

4. The apostrophe is eliminated as a string delimiter.

5. The relaxed parameter compatibility rule for the formal type ARRAY OF BYTE is applicable for variable parameters only.

Summary

The language Oberon has evolved from Modula-2 and incorporates the experiences of many years of programming in Modula. A significant number of features have been eliminated. They appear to have contributed more to language and compiler complexity than to genuine power and flexibility of expression. A small number of features have been added, the most significant one being the concept of type extension.

The evolution of a new language that is smaller, yet more powerful than its ancestor is contrary to common practices and trends, but has inestimable advantages. Apart from simpler compilers, it results in a concise defining document [9], an indispensible prerequisite for any tool that must serve in the construction of sophisticated and reliable systems.

Acknowlegement

It is impossible to explicitly acknowledge all contributions of ideas that ultimately simmered down to what is now Oberon. Most came from the use or study of existing languages, such as Modula-2, Ada, Smalltalk, and Cedar, which often taught us how *not* to do it. Of particular value was the contribution of Oberon's first user, J. Gutknecht. The author is grateful for his insistence on the elimination of dead wood and on basing the remaining features on a sound mathematical foundation. And last, thanks go to the anonymous referee who very carefully read the manuscript and contributed many valuable suggestions for improvement.

References

1. N. Wirth. *Programming in Modula-2.* Springer-Verlag, 1982.
2. N. Wirth. Type Extensions. *ACM Trans. on Prog. Languages and Systems, 10,* 2 (April 1988) 204-214.
3. G. Birtwistle, et al. *Simula Begin.* Auerbach, 1973.
4. A. Goldberg, D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.
5. L. Tesler. Object Pascal Report. *Structured Language World, 9,* 3 (1985), 10-14.
6. B. Stroustrup. *The Programming Language C++.* Addison-Wesley, 1986.
7. N. Wirth. The programming language Oberon. *Software - Practice and Experience, 18,* 7 (July 1988), 671-690.
8. J. Gutknecht and N. Wirth. The Oberon System. *Software - Practice and Experience, 19,* (1989)
9. N. Wirth. The programming language Oberon (Revised Report). (companion paper)

File: ModToOberon2.Doc / NW 1.10.90