CSc 520

The Oberon Programming Language Report

Manish Swaminathan, Bharathwaj

Spring 2008

05/14/2008

Contents

1. Introduction & History of the Language: 3
2. Procedures: 3
2.1 Procedure forward declaration 4
2.2 Local procedures 4
2.3 Parameter Passing: 4
3. Types: 5
3.1 Type Safety: 6
3.2 RECORDS: 6
3.3 PROCEDURE TYPE 6
3.4 TYPE EXTENSION 7
3.5 Type bound procedures 8
3.6 Installations 8
4. Polymorphism: 9
5. Modules 9
6. Commands: 10
7. Dynamic Loading: 10
8. Comparison with other Programming Languages: 11
9. Conclusion and ongoing work: 12
10. References 12

1. Introduction & History of the Language:

Oberon was developed in 1991 at ETH Zurich by Niklaus Wirth. It was originally intended for the design of the Oberon Operating System which ran on the Ceres workstation. In syntax Oberon is very similar to Modula-2 but is considerably smaller. The entire language is specified in a page of EBNF (Extended Bakus Naur Form). The language is known for its simplicity with its report spanning only 16 pages.

The original Oberon was designed to be a 'safe' language; it employs array bounds checking, garbage collection and strong type checking. These features enable errors to be detected as early as possible (i.e. at compile-time), can significantly reduce the number of bugs occurring in a program at runtime.

The language supports both Procedural and Object Oriented Programming. It also offers support for Systems programming

2. Procedures:

Both pass by reference and pass by value are supported. The procedure declaration determines the method by which the parameters are passed. Usage of the VAR keyword marks the pass by reference method. A local variable is generated for each pass by value parameter and is visible only within the procedure. Upon the completion of the procedure, these variables go out of scope.

There are two types of procedures – Proper procedure and function procedures. Function procedures return a value, while the proper procedures do not.

Examples:

```
PROCEDURE Copy(in: ARRAY OF CHAR; VAR out ARRAY OF CHAR) ;
(* proper procedure *)
BEGIN
  (* body of procedure *)
  RETURN
END Copy

PROCEDURE Sum(x, y: REAL): REAL
(* function procedure *)
BEGIN
  (* body of procedure *)
  RETURN answer
END Sum
```

Oberon procedures can return only a pointer or a simple type. The return type cannot be compound.

2.1 Procedure forward declaration

Oberon compilers are generally one-pass, thus enabling them to be very fast. So, if a procedure is used before it is defined, it must be declared first. The definition of the procedure is indicated by the use of a caret symbol, as shown below. Note that the procedure definition and the actual procedure should have the same definition. Example:

```
PROCEDURE ^Sum(x, y: REAL): REAL

PROCEDURE MathFunction (* a procedure that calls Sum *)

(* actual declaration of Sum, as above *)
```

2.2 Local procedures

Unlike C/C++, Oberon can have procedures within procedures. If there is local and a greater scope procedure, one within the other, then the local one will take precedence. If local and the greater scope procedure have the same name, then they should have the same parameters too. And in this case, if the outer function is called, then compiler will either perform type conversion and call the inner function, or will complain that that parameter lists do not match.

2.3 Parameter Passing:

We discussed functions and procedures in the previous section. This is an extension of the previous section and goes a bit more into parameters. Procedures may be given parameters. There are 3 types of parameter passing in Oberon –

a. **Variable Parameters**
The actual parameter corresponding to a formal variable parameter (specified by the symbol VAR) must be a variable. The formal identifier then stands for that variable.

```
PROCEDURE exchange(VAR x, y: INTEGER);
VAR z: INTEGER; BEGIN z := x; x := y; y := z
END exchange
```

The procedure calls : exchange(a, b); exchange(A[i], A[i+1])
then have the effect of the above three assignments, each with appropriate substitutions made upon the call.

- Variable parameters may be used to make a change outside the procedure
- The actual parameter cannot be an expression, and therefore not a constant either, even if no assignment to its formal correspondent is made.

- If the actual parameter contains indices, these are evaluated when the formal-actual substitution is made.

The types of corresponding formal and actual parameters must be the same.

b. **Value Parameters:**

Are used to transfer values from caller to callee. The corresponding actual parameter could be a variable, expression or a constant. As an illustration, we formulate a program to compute the power z = xi as a procedure.

```
PROCEDURE ComputePower(VAR z: REAL; x: REAL; i: INTEGER);
BEGIN z := 1.0;
WHILE i > 0 DO
IF ODD(i) THEN z := z*x END;
x := x*x; i := i DIV 2
END
END ComputePower
```

Possible calls are, for example

```
ComputePower(u, 2.5, 3) for u := 2.53
ComputePower(A[i], B[i], 2) for Ai := Bi, 2
```

An important point to keep in mind here is that the formal parameter is treated as a local variable and needs storage.

c. **Open Array Parameters:**

If the formal parameter is an array, then the corresponding actual parameter must also be an array of the same type. This automatically implies that the formal and actual parameters must have the same bounds of index range. Oberon provides this feature by using the concept of open arrays, which just requires the formal and the actual parameters of the array to be of the same type. Nothing needs to be said of the index range, and arrays of any size can be used as parameters. An open array is specified by the element type preceded by "ARRAY OF". For example, a procedure declared as

```
PROCEDURE P(s: ARRAY OF CHAR)
```

allows calls with character arrays of arbitrary length. The actual length is obtained by calling the standard function LEN(s).

3. Types:

Oberon supports a variety of types. This is summarized below.

Type=primitive | ArrayType | RecordType | PointerType | ProcedureType

Oberon includes a wide variety of types. The primary types include INTEGER, REAL, BOOLEAN, CHAR, SET, constants and variable declaration.

3.1 Type Safety:

Type Safety was one of the most of important considerations of the language design. In spite of its extensible features which rely on dynamically allocated data structures, Oberon is absolutely safe. Safety is achieved through Static Type Checking. Static types are checked at compile time (also across module boundaries). It also supports run time type checking for extensible types (effect of Polymorphism). Pointer Safety is facilitated by garbage collection.

3.2 RECORDS:

Non homogeneous data structures are supported in the form of Records. The record lets us declare a collection of items as an unit, even if they are all of different types. Example –

```
Date = RECORD day, month, year: INTEGER END

Person = RECORD
            firstName, lastName: ARRAY 32 OF CHAR;
            male: BOOLEAN;
            idno: INTEGER;
            birth: Date
         END
```

3.3 PROCEDURE TYPE

Oberon offers an option of allowing procedures themselves as a special kind of constant declaration, the value of this constant being the procedure itself. These are called procedure types. A procedure type declaration specifies the number and the type of parameters and, if applicable, the type of return value. Example – a procedure with an INTEGER ARRAY argument and a BOOLEAN return type is declared by:

```
TYPE Function = PROCEDURE(i:ARRAY OF INTEGER):BOOLEAN ;
```

If we now declare *f:Function*, then the assignment of a function with the same definition as above to f becomes possible. This is illustrated below. Search is a procedure which searches list integers using the algorithm that is defined in the function parameter.

```
Search: PROCEDURE( f:Function; a:ARRAY OF INTEGER) ;
```

This procedure could then be invoked as follows. BinarySearch and LinearSearch are procedures which search as the name suggests.

```
Search( BinarySearch, arr ) ;   Search( LinearSearch, arr ) .
```

This idea is similar to the concept of function pointers used in C/C++.

3.4 TYPE EXTENSION

Type extension is one of the most important features of Oberon. It allows the programmer to derive a new type from an existing one by just adding on to it. For example, consider the following declarations –

```
Base  = RECORD x, y : INTEGER END ;

Child = RECORD( Base ) z : INTEGER END ;
```

Child is an extension of the Base record. This is indicated by its declaration of the record where the base type's name is specified within parentheses. The Base record contains two integers – x and y. The Child record, which is an extension to Base, contains the 2 integers of Base – x and y, as well as its own integer – z. The notion of type extension also applies to pointers.

The crucial point about type extension is that if we have two pointers – B pointing to Base and C pointing to Child, then assignment B := C is legal, since C is an extended B and therefore, completely compatible with each other. The reverse assignment C := B is not possible since B is only a Base and not a Child like C. The same compatibility rules apply to records.

The concept of type extension directly lends itself to the idea of Classes and Objects.

CLASSES AND OBJECTS

There's no separate construct for classes. The **Record** type is itself used for class declaration. Methods are just added to the Record declaration. These methods are called **Type bound procedures**. They are equivalent to instance methods

Below is an example for Type extension:
Consider an Array for instance. An array can be an array of Ints or an array of Tuples of the form <x,y>. Hence the Array becomes an Abstract data type which can have different implementations (sub classes).

Some of the basic operations on an Array are searching for an element in the array, printing the array, counting the number of elements in the array etc. These methods together constitute the Interface. Let's declare the base class.

```
TYPE Element = POINTER TO ArrayDesc;
        ArrayDesc = RECORD (*this is the base type*)
        print: PROCEDURE (e: Element) ;
        search: PROCEDURE (f: PROCEDURE(e:Element):BOOLEAN;e: Element) ;
END ;
```

We can have sub classes IntsDesc and TupleDesc that inherits ArrayDesc.

```
TYPE Ints = POINTER TO IntsDesc;
        IntsDesc = RECORD (ElementDesc)
        intArray : ARRAY 5 OF INTEGER ;
        searchElement: INTEGER ;
        IPrint: PROCEDURE (e: Element) ;
        ISearch: PROCEDURE( f:Function; e: Element) ;
END ;

TYPE Tuple = POINTER TO TupleDesc;
        TupleDesc = RECORD (ElementDesc)
        tupArray : ARRAY 5 OF TupleElem ;
        searchElement : TupleElem ;
        TPrint: PROCEDURE (e: Element) ;
        TSearch: PROCEDURE( f:Function; e: Element ) ;
END ;
```

3.5 Type bound procedures

Type bound procedures are the equivalent of class methods in C++. Type bound procedures are indicated by declaring the record to which the procedure in parentheses before the procedure name. Type bound procedures may be redefined for records that are extensions. This was shown in the example above. It is important to note that the redefined procedure must have the same parameters as the base class one.

3.6 Installations

In strictly object-oriented languages methods are automatically associated with every object upon its creation, in Oberon this must be achieved by explicit assignments. Such initial assignments are called **installations** (of procedures). For example:

```
VAR c: IntsDesc;
NEW(c); c.search:=ISearch ; c.print := IPrint.
```

The call of a method, i.e. of an object-bound procedure, is expressed as in the following example: c.search(f,e).

4. Polymorphism:

Oberon does not have parametric polymorphism. This is due to the strong typing present inherently in the language.

Oberon achieves inclusion polymorphism through Type Extension, which we already discussed earlier. Extended Types are accepted at run time where ever objects of corresponding base types are.

Oberon offers two constructs for Run time type testing.

- Type Test - In order to determine the type of an object at run time, the type test facility can be used. This is classified as a Boolean expression and has the form
  expression = variable “IS” identifier (similar to ISA in Luca).

- Type Guard - In order to typecast the base class object to the desired derived class object .The type guard construct can be used. If ‘P’ is of the static type “ArrayDesc” and we need to type cast it to a IntsDesc object, we can use the following syntax
  P(IntsDesc).

5. Modules

Modules are the most important difference between Oberon and its ancestor Pascal. The main building block of Oberon applications is the module. Each module has a name. Modules can be imported for use into another module. Modules can also be exported(made public) for use by other modules. Variables and functions are tagged as exported by appending a "*" at the end of the name. Read-only export is also possible, this time by appending a "-" at the end of the name. Modules can also contain the building blocks described before - type definitions, constants, variables and functions.

Comments are opened by "(*" and closed by "*)". Here is an example of the module syntax:

```
MODULE MyModule; (* MyModule is the name of this module *)
  IMPORT S := SYSTEM, OS2;
  (* Here we import 2 modules, SYSTEM for low-level Oberon functions *)
  (* and OS2, an interface to the OS/2 API *)
  (* Note "S := SYSTEM" allows us to use S as an alias of SYSTEM, so *)
  (* we can call functions like S.CC() *)

(* type definitions, constants, variables, procedures *)
BEGIN
  (* body of the module goes here *)
END MyModule.
```

Modules offer a convenient way to organize the structure of programs. It also allows the use of modules provided by other programmers. But the most important use of modules is the establishment of a hierarchy of abstractions. If the inner working of the program is protected from outside meddling, then we can make sure that the modules are working correctly and thereby limit our error searching to a limited area. Modules also offer a way to change/improve the imported modules without having to make change to the importing modules. This effective decoupling of modules is especially useful for development of large programs, especially when multiple developers are involved.

An importer of a module needs to know the properties of those objects only which are exported by the imported module. This is provided by the *definition* of a module. The definition is kind of the interface which the programmer follows while using the imported module and contains a listing of the properties that the importer can access and use.

The following simple example exhibits the essential characteristics of modules, although typically modules are considerably larger pieces of program and contain a longer list of declarations. This example exports two procedures, *put* and *get*, adding data to and fetching data from a buffer which is hidden. Effectively, the buffer can only be accessed through these two procedures; it is therefore possible to guarantee the buffer's proper functioning, i.e. sequential access without loss of elements.

```
DEFINITION Buffer;
VAR nonempty, nonfull: BOOLEAN;
PROCEDURE put(x: INTEGER);
PROCEDURE get(VAR x: INTEGER);
END Buffer.
```

6. Commands:

In Oberon, the unit of execution is a procedure unlike a program in traditional Operating Systems like Unix/Dos. A command is any parameter less procedure *P* that is exported from a module *M*. It is denoted by *M.P* and can be activated under this name from the shell of the Oberon operating system. When a command *M.P* is invoked, the module *M* is dynamically loaded unless it is already in memory and the procedure *P* is executed. When *P* terminates, *M* remains loaded. All global variables and data structures that can be reached from global pointer variables in *M* retain their values. When *P* (or another command of *M*) is invoked again, it may continue to use these values. Just as how in Unix, commands interact with each other through pipes, procedures in an Oberon system communicate through Shared Memory.

7. Dynamic Loading:

Oberon employs the lazy loading mechanism to load modules. There are no pre-linked files on the disk but every module is compiled to a separate object file. When a command from a module

M is invoked, M's object file is loaded and linked with all modules imported by M. The other dependant modules are loaded only when they are referenced in the procedure. This is very similar to how dynamic loading is done in C & Unix except that the programmer, in C needs to explicitly use the Loader APIs (dlopen() & dlclose()) to load/unload modules where as in Oberon it is abstracted from the programmer.
The dynamic loading feature facilitates extensibility at run time. Hence Oberon supports Extensibility both at the language level (through Type Extension) and at the System level (through dynamic loading). Also loading of an Oberon application/program happens much faster because only a subset of the program is loaded initially.

8. Comparison with other Programming Languages:

Data Abstraction:
Oberon like Module2 ensures Data Abstraction through Modules.

An important distinguishing feature of Oberon is that the Implementation and Definition parts are in the same file. The motivation was that they wanted to enhance the compilation time by making it a one step process as opposed to its ancestor Module-2 where it is a two step process. Oberon implementers also claim that it simplifies maintenance overhead as the Programmer can concern himself with only one document.

Classes and objects:
Other languages like C++ and Java introduce object oriented programming through special constructs such as classes and objects. Oberon is able to get the same functionality with just type extension, thus making the cross over easier. Oberon does not suggest a particular OOP style, but leaves it to the programmer to select the appropriate method for the task. Entire hierarchies of records can be created using type extension.

Type Safety:
Oberon relies on the notion of Strong Typing. All static types are checked at compile time (static type checking) .Run time checks are made for polymorphic types (feature of type extension) and pointer safety & memory consistency are taken care of by the garbage collector. Oberon also employs array bounds checking with almost zero overhead in execution time.

Genericity:
Oberon does not include any built in feature that demonstrates genericity, but one can implement a separate tool to simulate genericity (like the C++ template preprocessor).

Error Handling:
Oberon employs assertions as opposed to exceptions. They argue that features like “Exception Handling” are Application specific and should not be provided by the language. They should be provided by means of modules rather than through language constructs.

Garbage Collection:
Oberon has an in built garbage collector that ensures pointer safety.

9. Conclusion and ongoing work:

Oberon is a small and concise language and has both procedural and object oriented features. The three most important features are type extension, data abstraction and dynamic loading. Oberon was followed by Oberon 2 which adds open arrays as pointer base types, read-only field export and reintroduces the FOR loop from Modula-2.Active Oberon has been built on top of this and supports concurrency, a prominent missing feature in previous versions. It also introduces component modeling in a compact manner.

10. References

- Oberon language report - http://www.oberon.ethz.ch/oreport.html
- OBJECT-ORIENTED PROGRAMMING IN OBERON-2 http://www.statlab.uni-heidelberg.de/projects/oberon/kurs/www/Oberon2.OOP.html
- http://www.oberon.ethz.ch/
- EXTENSIBILITY IN THE OBERON SYSTEM by HANSPETER MOSSENBOCK (Institute for Computer Systems ETH Zurich)
- Erlangen's First Independent Modula_2 Journal! Nr. 9, Oct-1994
- Separate Compilation and Module Extension by Regis Bernard Joseph Crelier (Swiss Federal Institute of Technology , Zurich)
- Active Oberon language report - http://bluebottle.ethz.ch/languagereport/index.html
- Oberon vs C++ - http://www.modulaware.com/mdlt49.htm