OP2: A PORTABLE OBERON–2 COMPILER

Presented at the 2nd International Modula–2 Conference, Loughborough, Sept 91

Régis CRELIER
Institut für Computersysteme, ETH Zürich

ABSTRACT

A portable compiler for the language Oberon–2 is presented. Most related works pay for portability with low compilation speed or poor code quality. Portability and efficiency have been given the same importance in our approach. Hence, an automated retargetable code generation has not been considered.

The compiler consists of a front–end and a back–end. The front–end does the lexical and syntactic analysis, including type checking. It builds a machine–independent structure representing the program. This structure is made up of a symbol table and an abstract syntax tree, rather than a stream of pseudo–instructions coded in an "intermediate language". If no errors are found, control is passed to the back–end which generates code from this intermediate structure. This structure clearly separates the front–end which is machine–independent from the back–end which is machine–dependent. While the front–end can remain unchanged, the back–end has to be reprogrammed, when the compiler is retargeted to a new machine.

This compiler has been successfully used to port the Oberon System onto different computers. Code generators have been implemented both for CISC and RISC processors. Differences in processor architectures are reflected in the complexity of the back–end, the generated code density and performance. The compiler is written in Oberon. New compilers have therefore to be first compiled on an already working Oberon System. If such a system is not available, a version of the compiler whose back–end produces C code may be used for the bootstrap.

The compilation techniques presented here are not restricted to Oberon compilers, but could be used for other programming languages too. Nevertheless, Oberon and OP2 tend to the same ideal: simplicity, flexibility, efficiency and elegance.

INTRODUCTION

Portability is an important criterion for program quality. A compiler is a program as well, and it may be ported. If it should produce the same code as before on the new machine (cross compiler), then it is not more difficult to port it than any other program also written in a higher programming language. But if the produced code must run on the new machine, the compiler has to be rewritten and it is not the same program any more. In that sense, the compiler is not, and cannot be, portable. By the term *portable compiler*, we refer here to a compiler that needs reasonably small effort to be adapted to a new machine and/or to be modified to produce new code.

Most related work attempt to reduce this adaptation cost to a minimum, compromising the compilation speed and the code quality. A classification of such automated retargetable code generation techniques and a survey of the works on those techniques are presented in [1]. The basic idea is to produce code for a virtual machine. This code is then expanded into real machine instructions. The expansion can be done by hand–written translators [2] or by a machine–independent code generation algorithm, in which case each intermediate language instruction [3] or each recognized pattern of these [4] is expanded into a fixed pattern of target machine instructions recorded in tables. Trees may replace linear code to feed the pattern matching algorithm [5, 6, 7]. These techniques usually yield poor code quality, making a peephole optimization phase necessary, which further increases the compilation time.

In our approach, we tried to find the right balance between code quality, compilation speed and portability. We think it is worth–while investing, say three man–months, for a port, if the resulting compiler is very fast and produces efficient code. Thus, a pattern matching or table–driven code generation has not been considered. Instead, we looked at more conventional and faster techniques, such as single–pass compilation [8, 9]. In a single–pass compiler, the compilation phases are executed simultaneously. No intermediate representation of the source text is needed between the different phases, making the compiler compact and efficient, but not very portable. Indeed, since machine–dependent and machine–independent phases are mixed up, it is very difficult to modify the compiler for a new machine.

One solution to the problem is to clearly separate the compilation phases into two groups: the *front–end* consisting of the machine–independent phases (lexical and syntactic analysis, type checking) and the *back–end* consisting of the machine–dependent phases (storage allocation, code generation). Only the back–end must be modified when the compiler is ported. The front–end enters declarations in a symbol table and builds an intermediate representation of the program statements, an abstract syntax tree. If no errors were found, control is passed to the back–end, which generates code from the syntax tree. Since this structure is guaranteed to be free of errors, type checking or error recovery are not part of the back–end, which is a noteworthy advantage. Only implementation restrictions must be checked. Another advantage of the intermediate structure is that optional passes may be inserted to optimize the code. Such an optimization phase cannot be easily embedded in a conventional single–pass compiler. The front–end and the back–end are implemented separately as a set of modules.

MODULE STRUCTURE

Originally, OP2 has been designed to compile Oberon programs [10] and has been slightly modified later to compile Oberon–2 programs [11, 12]. It consists of nine modules (see figure 1) all written in Oberon.

The lowest module of the hierarchy is OP**M**, where **M** stands for **m**achine. We must distinguish between the host machine on which the compiler is running, and the target machine for which the compiler is generating code. Most of the time, the two machines are the same, except during a bootstrap or in case of a cross–compiler. The module OPM defines and exports several constants used to parametrize the front–end. Some of these constants reflect target machine characteristics or implementation restrictions. For example, these values are used in the front–end to check the evaluation of constant expressions on overflow. But OPM has a second function. It works as interface between the compiler and the host machine. This interface includes procedures to read the text to be compiled, to read and write data in symbol files [13], and to display text (error messages e.g.) onto the screen. All these input and output operations are strongly dependent on the operating system. If the compiler resides in the Oberon System environment [14, 15], the host–dependent part of OPM remains unchanged.

The topmost module (OP2) is very short. It is the interface to the user, and therefore host machine–dependent. It first calls the front–end with the source text to be compiled as parameter. If no error is detected, it then calls the back–end with the root of the tree that was returned by the front–end as parameter.

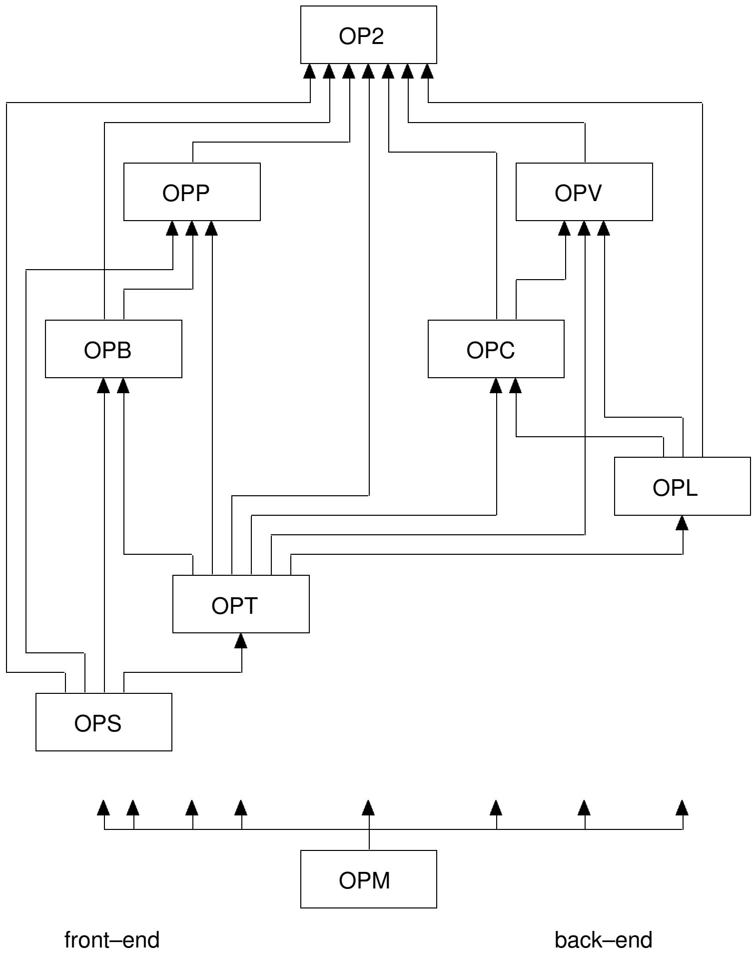

Figure 1. Module import graph (an arrow from A to B means *B imports A*)

Between the highest and the lowest module, one finds the front–end and the back–end, which consist of four, respectively three modules. There is no interaction during compilation between these two sets of modules. The symbol table and the syntax tree are defined in module OPT and are used by both the front–end and the back–end. This explains the presence of import arrows from OPT to back–end modules visible in the import graph (figure 1). But there is no transfer of control, such as procedure calls.

The front–end is controlled by the module OP**P**, a recursive–descent **p**arser. Its main task is to check syntax and to call procedures to construct the symbol table and the syntax tree. The parser requests lexical symbols from the **s**canner (OP**S**) and calls procedures of OP**T**, the symbol **t**able handler, and of OP**B**, the syntax tree **b**uilder. OPB also checks type compatibility.

The back–end is controlled by OP**V**, the tree tra**v**erser. It first traverses the symbol table to enter machine–dependent data (using OPM constants), such as the size of types, the address of variables or the offset of record fields. It then traverses the syntax tree and calls procedures of OP**C** (**c**ode emitter), which in turn synthesizes machine instructions using procedures of OP**L** (**l**ow–level code generator).

This module structure achieves to make the front–end target–independent as well as host–independent, and to make the back–end host–independent.

SYMBOL TABLE

The symbol table contains information about declared constants, variables, types and procedures. It is built by the front–end. The front–end uses it to check the context conditions of the language and the back–end retrieves type information from it. The symbol table is a dynamically allocated data structure with three different element types:

```
TYPE
    Const = POINTER TO ConstDesc;
    Object = POINTER TO ObjDesc;
    Struct = POINTER TO StrDesc;
```

An *Object* is a record (more exactly a pointer to a record), which represents a declared, named object. The object declaration in the compiler is the following:

```
ObjDesc = RECORD
    left, right, link, scope: Object;
    name: OPS.Name;          (* key *)
    leaf: BOOLEAN;           (* procedure: leaf; variable: candidate to be allocated in register *)
    mode: SHORTINT;          (* constant, type, variable, procedure or module *)
    mnolev: SHORTINT;        (* imported from module –mnolev, or local at procedure nesting level mnolev *)
    vis: SHORTINT;           (* not exported, exported, read–only exported *)
    typ: Struct;(* object type *)
    conval: Const;           (* numeric attributes *)
    adr, linkadr: LONGINT    (* storage allocation *)
END ;
```

The name of the object stored in the object itself (field *name*) is used as key to retrieve the object in its scope. Each scope is organized as a sorted binary tree of objects (fields *left* and *right*) and is anchored (field *scope*) to the owner procedure, which in turn belongs as object to the enclosing scope. Parameters of the same procedure, fields of the same record and variables of the same scope are additionally linked together (field *link*) to maintain the declaration order. The flag *leaf* indicates whether a procedure is a leaf procedure or whether a variable is a candidate to be allocated permanently in a register. The back–end may or may not use this information – Note that this information could not be available in a single–pass compiler without intermediate representation of the program. An object has always a type (field *typ*), which is described by a record named *StrDesc*:

```
StrDesc = RECORD
    form, comp: SHORTINT; (* basic or composite type, type class *)
    mno: SHORTINT;           (* imported from module mno *)
    extlev: SHORTINT;        (* record extension level *)
    ref, sysflag: INTEGER;   (* export reference *)
    n, size: LONGINT;        (* number of elements and allocation size *)
    tdadr, offset: LONGINT;  (* address of type descriptor *)
    txtpos: LONGINT;         (* text position *)
    BaseTyp: Struct;         (* base record type or array element type *)
    link: Object;            (* record fields or formal parameters of procedure type *)
    strobj: Object           (* named declaration of this type *)
END ;
```

There are several classes of types: basic types like character, integer or set, and composite types like array, open array or record (fields *form* and *comp*). The third element type of the symbol table is *ConstDesc*. This record contains numeric attributes of objects, like values of declared or anonymous constants:

```
ConstDesc = RECORD
    ext: ConstExt;              (* extension for string constant *)
    intval: LONGINT;
    intval2: LONGINT;
    setval: SET;
    realval: LONGREAL
END ;
```

An example is shown in figure 2 below:

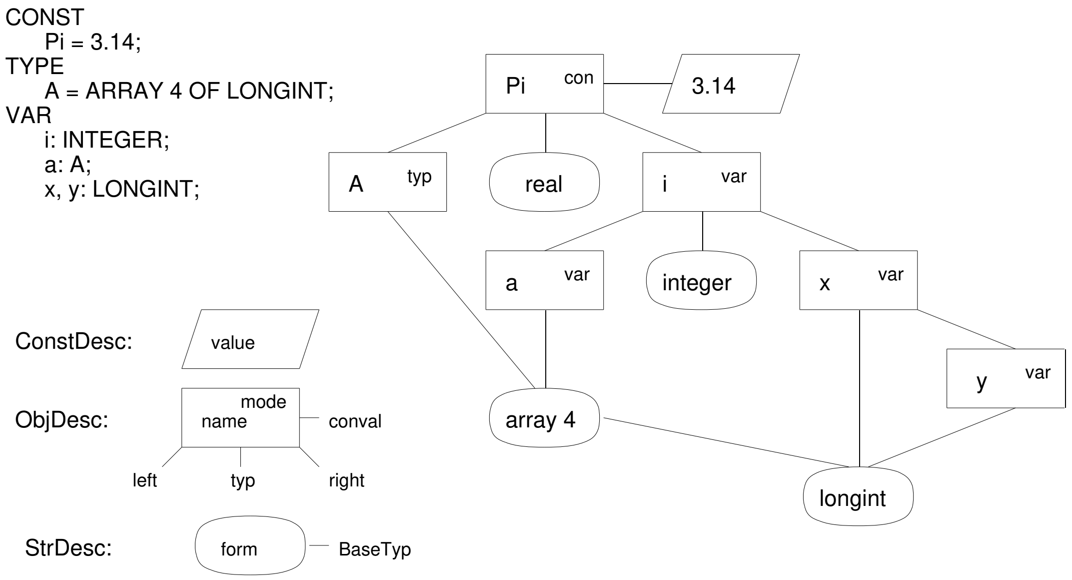

Figure 2. Declarations and corresponding symbol table

SYNTAX TREE

The front–end builds an abstract syntax tree representing all statements of the program. The Oberon syntax is mapped into a binary tree of elements called *NodeDesc*:

```
Node = POINTER TO NodeDesc;

NodeDesc = RECORD
    left, right, link: Node;
    class, subcl: SHORTINT;
    readonly: BOOLEAN;
    typ: Struct;
    obj: Object;
    conval: Const
END ;
```

A binary tree has been chosen because each Oberon construct can be decomposed into a root element identifying the construct and two subtrees representing its components: an operator has a left and a right operand, an assignment has a left and a right side, a While statement has a condition and a sequence of statements, and so on. Some Oberon constructs are organized sequentially: there are lists of actual parameters in procedure calls and sequences of statements in structured statements. It would be expensive to insert dummy nodes to link these subtrees; an additional *link* field in the node is much cheaper.

Each node has a class (field *class*) and possibly a subclass (field *subcl*) identifying the represented Oberon construct. Each node has a type, which is a pointer (field *typ*) to a *StrDesc* of the symbol table. Similarly, a leaf node representing a declared object contains a pointer (field *obj*) to the corresponding *ObjDesc* of the symbol table. A *ConstDesc* may be attached (field *conval*) to a node to describe a numeric attribute, such as the value of an anonymous constant. A *ConstDesc* denoting the position in the source text is anchored to the root node of each statement. This allows locating compilation errors reported by the back–end.

Figure 3 shows the representation of two statements manipulating variables declared in Figure 2.

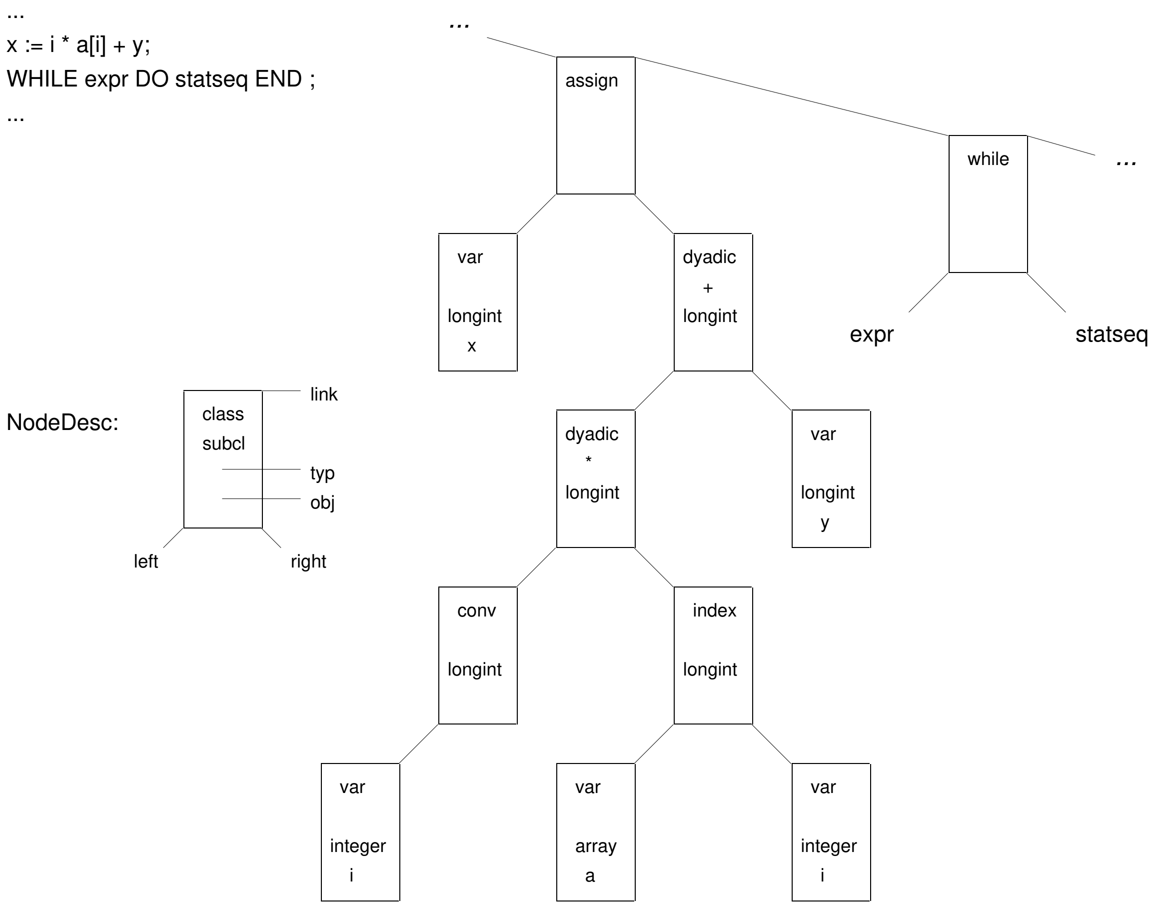

Figure 3.   Statements and corresponding syntax tree

The rules of numeric type compatibility are flexible in Oberon; for example, it is possible to multiply integers with long integers, as shown in Figure 3. Note the conversion operator inserted in the tree, freeing the back–end from type checking.

While generating code for a node, one typically has to recursively evaluate left and right subtrees, then the node itself, and finally the linked successors if any. A traversal of the tree looks like this:

```
Traverse(node: Node):
  WHILE node # NIL DO
    Traverse(node^.left);
    Traverse(node^.right);
    Do something with node;
    node := node^.link
  END
```

The intermediate representation could be a stream of instructions for a virtual machine; we have preferred an internal abstract syntax tree for different reasons. A virtual machine instruction set should be defined without knowing anything about the future target machines. Perhaps the mapping of this instruction set to a real instruction set would not be easy, the virtual and real machines being very different (RISC vs. CISC e.g.). Generating these pseudo–instructions already needs a code generator, whereas building the syntax tree is a trivial recursive task easily embedded in the recursive–descent parser. Since the tree is a natural mapping of the Oberon syntax, each procedure of the parser returns as parameter the root of the subtree corresponding to the construct just parsed by this procedure. Furthermore, the tree keeps the program structure intact, so that a control–flow analysis necessary for an optimization phase can be done easily. Without a tree, this analysis would be expensive, since basic blocks would have been dissolved in the linear code. The reordering of program pieces is easier to perform in a tree than in an instruction stream; for example, the conditional expression of a While statement may be evaluated at the end of the loop, while first generating code for the right subtree of the While node, and then for its left subtree.

The syntax tree has nevertheless one drawback over an intermediate linear code: it needs quite a large heap space, but, nowadays, this is not a real problem any more.

CODE GENERATION

Although an extensive experience in compiler development is not necessary to write a new back–end, knowledge of the subject is nevertheless an advantage. The front–end, which doesn't need any modifications, has only to be adapted to the new target machine by editing constants exported from module OPM. Then, the storage allocation strategy must be adapted to the data alignment requirements of the new processor. This allocation is done in module OPV, where a procedure traverses the symbol table and distributes addresses to variables, offsets to record fields, sizes to structured types, and so on. The last thing to do, but not least, is to rewrite the code generator.

After storage allocation, code generation takes place. Between the two phases, OPT shortly gets control in order to produce a symbol file. The whole process of code generation can be viewed as the process of computing attribute values for each node of the tree. Attribute values of a node depend on the attribute values of the children nodes and on the node's class and type. Hence, module OPV recursively traverses the syntax tree and calls, in a post–order fashion, procedures of underlying modules computing attribute values for each traversed node, or subtree of nodes. These procedures act similarly to an *attribute evaluator* of an *attribute grammar*. The emission of code is actually a side effect of computing these attributes. Since the side effect (code) is more important than the computed attribute values, and since these values will only be reused later to compute the attribute values of parent nodes, they don't need to be stored in the tree. Instead, they are passed as procedure parameters named *Item* during the recursive traversal of the tree.

An *Item* is a record of attributes representing the operand or the result of an operation. It indicates where the operand is located (memory, register or immediate value e.g.). *Items* make it possible to delay emission of code, so that processor addressing modes can be optimally used. There are as many *Item* modes as processor addressing modes. Depending on the mode specified by a field of the *Item*, other attributes like type, address, offset, register number, value of constant are stored in *Item* fields as well. The complexity of a processor architecture is reflected in the declaration of the *Item*. Typically, back–ends for CISC processors have *Items* with many fields and many possible modes, whereas the ones for RISC are very simple.

Since almost each procedure of the code generator has to distinguish between the different *Item* modes, the complexity of the back–end depends on the number of modes. Expression evaluation for RISC processors is therefore very easy to code. A more difficult part of RISC back–ends is the register allocation. The advantage in performance of RISC over CISC may be lost if a too simple allocation strategy is used.

An excerpt of the module OPV is listed below, giving an idea how the back–end works:

```
PROCEDURE^ expr(n: OPT.Node; VAR x: OPL.Item);     (* forward declaration *)

PROCEDURE design(n: OPT.Node; VAR x: OPL.Item);
    VAR y: OPL.Item;
BEGIN
    CASE n^.class OF
    ...
    | index: design(n^.left, x); expr(n^.right, y); OPC.Index(x, y)    (* x := x[y] *)
    ...
    END ;
    x.typ := n^.typ
END design;

PROCEDURE expr(n: OPT.Node; VAR x: OPL.Item);
    VAR y: OPL.Item;
BEGIN
    CASE n^.class OF
    ...
    | dyadic:
        expr(n^.left, x); ... expr(n^.right, y);
        CASE n^.subcl OF
        ...
        | plus: OPC.Add(x, y)      (* x := x + y *)
        ...
        END ;
    ...
    END
    x.typ := n^.typ
END expr;

PROCEDURE stat(n: OPT.Node);
    VAR x: OPL.Item; L0, L1: OPL.Label;
BEGIN
    WHILE n # NIL DO
        CASE n^.class OF
        ...
        | while:
            L0 := OPL.pc;            (* remind loop beginning *)
            expr(n^.left, x);        (* evaluate conditional expression into x *)
```

```
            OPC.CFJ(x, L1);      (* if not x then jump to L1 *)
            stat(n^.right);      (* do statement sequence *)
            OPC.BJ(L0);          (* backwards jump to L0 *)
            OPL.FixLink(L1)      (* fix–up L1 with current pc *)
        ...
        END ;
        n := n^.link
    END
END stat;
```

MEASUREMENTS AND BENCHMARKS

The front–end of OP2 (modules OPS, OPT, OPB and OPP), which remains the same for all versions of OP2, consists of less than 3500 lines of Oberon source code. The length of the back–end and the size of the machine code depend on the target processor architecture.

The very first back–end for OP2 was written by the author in less than three months for the National Semiconductor NS32532 processor used in the Ceres workstation developed at ETH [16]. The newest version (Oberon–2) of this back–end (OPL, OPC and OPV) for that machine consists of less than 2500 lines of Oberon source code, of which about 500 are portable (module OPV). The total size of the compiler (including modules OPM and OP2) is less than 6500 lines.

About 2000 lines (30% of the compiler) have to be rewritten when the compiler is ported. This number may vary slightly depending on the target architecture. For the MIPS R2000, for example, the back–end is 500 lines longer. The heap space required to store the syntax tree of a module depends only on the size of the module, but not on the target processor architecture; it is about 8 times larger than the source text of the module.

Larger variations are noticed in the size of the machine code: the whole compiler for NS32532 is 62KB, whereas the one for the MIPS R2000 is 152KB. The different architecture is not only reflected in code density, but in performance too: on a Ceres (NS32532, 25 MHz), it takes 32 seconds to recompile OP2; on a DECstation 5000 (MIPS R3000, 25 MHz), only 6.3 seconds. Note that self–compilation time is a good indicator for compiler quality, since speed, compactness and code quality are multiplicative factors contributing to the overall result.

Several other code generators have been implemented at the Institut für Computersysteme. They have been used to port the Oberon System to different computers (Sun SPARCstation, Macintosh, DECstation, IBM PS/2 and S/6000) [17, 18, 19]. In most cases, the new back–ends have been developed on the Ceres workstation in about three or four months by a single person. The object files, cross–compiled on Ceres, have been then transferred to the target machine using a diskette or an RS232 line.

We also have a special back–end producing C code. The idea here is not to shirk the task of writing a code generator, but to use this Oberon–to–C translator as a preprocessor for a C compiler during the bootstrap of the compiler only. Remember that OP2 is written in Oberon; so, if there is no machine with a running Oberon compiler at immediate disposal, the new OP2 cannot be cross–compiled. So we execute the bootstrap on the target machine directly. We first translate the new OP2 to C and then compile it using an existing C compiler. After a self–compilation step of OP2, we can and should forget the C compiler.

ACKNOWLEDGEMENTS

The single–pass Oberon compiler of N. Wirth for Ceres has been a model of efficiency and compactness for this project. The advice of M. Odersky helped a lot during the design phase. J. Templ, M. Franz and H. Mössenböck made valuable comments and improvement suggestions on OP2 and on this paper. Many thanks to all of them and to the very first users who searched desperately in their programs for bugs hidden in OP2.

REFERENCES AND FURTHER READING

1. Ganapathi M., Fischer C. N., Hennessy J. L., *Retargetable Compiler Code Generation*, Computing Surveys 14:4, 573–592, 1982.
2. Amman U., Jensen K., Nägeli H., Nori K., *The Pascal 'P' Compiler: Implementation Notes*, Departement Informatik, ETH Zürich, 1974.
3. Tanenbaum A. S., Kaashoek M. F., Langendoen K. G., Jacobs C. J. H., *The Design of Very Fast Portable Compilers*, ACM SIGPLAN Notices 24:11, 125–131, 1989.
4. Glanville R. S., Graham S. L., *A New Method for Compiler Code Generation*, Fifth ACM Symposium on Principles of Programming Languages, 231–240, 1978.
5. Aho A. V., Ganapathi M., Tjiang S. W. K., *Code Generation Using Tree Matching and Dynamic Programming*, ACM Transactions on Programming Languages and Systems 11:4, 491–516, 1989.
6. Cattell R. G. G., Newcomer J. M., Leverett B. W., *Code Generation in a Machine–Independent Compiler*, ACM SIGPLAN Notices 14:8, 65–75, 1979.
7. Fraser C. W., Wendt A., *Integrating Code Generation and Optimization*, ACM SIGPLAN Notices 21:6, 242–248, 1986.
8. Wirth N., *A Fast and Compact Compiler for Modula–2*, Report 64, Departement Informatik, ETH Zürich, 1985.
9. Wirth N., *Compilerbau, Eine Einführung*, B. G. Teubner Stuttgart, 1986.
10. Wirth N., *From Modula to Oberon, The Programming Language Oberon (revised edition)*, Report 143, Departement Informatik, ETH Zürich, 1990.
11. Mössenböck H., *Differences between Oberon and Oberon–2, The Programming Language Oberon–2*, Report 160, Departement Informatik, ETH Zürich, 1991.
12. Mössenböck H., *Object–Oriented Programming in Oberon–2*, Second International Modula–2 Conference, Loughborough, 1991.
13. Gutknecht J., *Compilation of Data Structures: A New Approach to Efficient Modula–2 Symbol Files*, Report 64, Departement Informatik, ETH Zürich, 1985.
14. Wirth N., Gutknecht J., *The Oberon System*, Report 88, Departement Informatik, ETH Zürich, 1988.
15. Reiser M., *The Oberon System, User Guide and Programmer's Manual*, Addison–Wesley, 1991.
16. Eberle H., *Development and Analysis of a Workstation Computer*, Ph.D. Thesis, ETH Zürich, 1987.
17. Templ J., *SPARC Oberon – User's Guide and Implementation*, Report 133, Departement Informatik, ETH Zürich, 1990.
18. Franz M., *The Implementation of MacOberon*, Report 141, Departement Informatik, ETH Zürich, 1990.
19. Pfister C., Heeb B., Templ J., *Oberon Technical Notes*, Report 156, Departement Informatik, ETH Zürich, 1991.
20. Crelier R., *OP2: A Portable Oberon Compiler*, Report 125, Departement Informatik, ETH Zürich, 1990.