Building Your Own Tools: An Oberon Industrial Case-Study

Paul Reed

Padded Cell Software Ltd, PO Box 1880, London NW6 1BQ, United Kingdom
paulreed@paddedcell.com, http://www.paddedcell.com

Summary: Our experience creating custom application software has taught us that total control over our development tools is a necessity. Project Oberon provided an excellent starting point for us to build our own cross-platform application programming environment. Our adaptation of Wirth's compiler is re-targetable at run-time via a small set of installable up-calls, enabling a single machine-specific code-generation module of typically less than a thousand lines of code. The only significant additions to the original Oberon language are floating-point binary-coded decimals and open-array variables with string concatenation (e.g. s := "Error: " + t). Accompanying run-time libraries, written in Oberon, for operating systems such as Microsoft Windows (32-bit) and MS-DOS have been developed. Several systems created using the new tools have been in use by customers for some time.

1. Introduction

We present an industrial application of Niklaus Wirth's and Jürg Gutknecht's research project [1], the programming language [2] and operating system Oberon, which aims to retain the original's simplicity, clarity and reliability. First the motivation for developing the software is explained, and the occasionally roundabout route which led to it is described in some detail. The particular design aspects of our version of the Oberon compiler and its target run-time environment are summarised, and finally we outline the steps involved in writing a new code generator for a given machine. In publishing these experiences, we hope that others may be encouraged to take Wirth's blunt attitude to 'bells and whistles', and enjoy reliable software as a natural result.

2. Motivation

Providing working software for real customers is in most cases hard work, yet despite warnings [3] the complexity involved continues to be underestimated by the industry's practitioners. Many large, high-profile software projects fail completely - and 'small' software is often unreliable. Critics often contrast this with civil engineering, where (on the whole) bridges do not tend to collapse during construction. Considering that in software we have far more control over the properties of our tools and materials than in other branches of engineering, the phenomenon is all the more surprising.

We reject the argument that computing is still in its infancy and therefore that such failures are only to be expected - consider aviation, where through great effort and the will to create a safety culture, passenger flights were made acceptably safe by the

J. Gutknecht and W. Weck (Eds.): JMLC 2000, LNCS 1897, pp. 291-298, 2000.

1930's and fatality rates were improved again by an order of magnitude by the late 1950's.

In our experience, the biggest obstacle to providing finished software is finding the time to write it. Analysing our past projects, which used the complex development tools available commercially, it dawned on us that a frightening proportion of intellectual effort and time was being wasted on installation, configuration and debugging of these tools - we estimated in one extreme case as much as eighty percent of the entire project time.

We feel that it is partly *because* software offers the freedom to design ridiculously over-featured compilers and other development tools (most of which are far from bug-free) that the complexity of projects can get out of hand. Wrestling with the unpredictable properties of a fault-ridden programmer's tool may be stimulating, but it is distracting, and leaves less of our mental capacity available to solve the problems in the customers' domain.

Creating simple, single-purpose tools is unfashionable because it restricts scope and flexibility; but this can sometimes actually be beneficial, reducing the sheer number of decisions to be made along the way. An example of this is the software running the tiny computer fitted to the Voyager 2 space probe, whose outstandingly successful mission inspired Wirth to begin the original Oberon project [1].

The concept of providing just enough is expounded repeatedly in Project Oberon. But what makes it unique is the presentation of the entire source code of the operating system and compiler, and extremely lucid descriptions of their design and construction, all in a single compact volume. Many excellent expositions of compiler design and compiler-writing toolkits exist, for example [4], [5]; but the emphasis is often on a general approach. Attention in the literature tends to be focused on writing a 'front-end', where statements formulated in the source language are parsed, whereas actual code-generation (the 'back end') is sometimes even left as an exercise to the reader.

Knowing that we could fall back on the source code and explanations in the book, at the beginning of 1997 we decided to start a project to build some new in-house development tools based around the hand-crafted, single-pass, recursive-descent Oberon compiler designed by Wirth.

3. Objectives

The goal of our project was a simple, portable application programming environment gained in a reasonably short time scale.

Simplicity was essential in order to ensure a complete understanding of the tool on which we were totally reliant. This avoids being in the position, when developing software systems for customers, of blaming tools like bad workmen. There is also usually a need to maintain and adapt software long after it has been created, and our compiler tool would not be exempt from this.

Several different machines and operating systems, including at least Microsoft Windows (32-bit) and Apple Macintosh, would need to be supported. So the compiler would need to be portable, generating code for more than one processor chip (at least Intel 80386 and Motorola 68000 respectively).

The third requirement was that the programming of the tools should not be an undue distraction - either during their construction or afterwards. We had no intention of becoming a development tools company, spending large amounts of time on the maintenance of our compiler for each machine platform - we wished actually to *use* the tools.

4. Getting Started

The initial plan was to take the (1996) standard ETH distribution of Oberon System 3 with a view to using it as-is, modifying it only slightly for our needs. Ports existed for many different processors, including the Intel and Motorola code generators which we required. Our idea was for one person to spend a year (part-time) building a usable integrated development environment (program source code editor, compiler, and linker) for the two platforms, Windows and Macintosh. We had learned from previous cross-platform projects that developing for more than one platform *at the same time* leads to better software: when it is discovered that a bug is either platform-specific or application-specific, this becomes the first step in the divide-and-conquer approach to correcting the error.

Unfortunately, although Regis Crelier's portable OP2 compiler [6] on which the System 3 Oberon distributions are based is an elegant and simple design, we found the actual implementations on each machine platform to be rather inscrutable. This is partly due to the vagaries of each machine's instruction set and run-time environment (particularly Intel) but also born from the desire to 'optimise' code-generation to produce high-performance machine code as the result. We were more concerned with maintainability and reliability of the tools - we had been 'bitten' in the past by optimising compilers which, when optimisation was turned on, generated the *wrong* code.

Reluctantly, we embarked on the one-man development of our own Oberon compiler version, using Project Oberon and Wirth's little Compiler Construction book [7] as our guides. A 'throw-away' MS-DOS compiler for Oberon-0 (a subset of the Oberon language) was developed, based on the Oberon-0 compiler described in Compiler Construction. This had the dual role of allowing us to see whether we were up to the task, and also gave us a rudimentary estimate of how long a full project might take. A compiler capable of compiling a demonstration of the Syracuse or "3x + 1" problem [8] was finished after three months. It was realised that one man-year had been somewhat optimistic, and the time-scale was re-set to three years (part-time). This meant that completion neatly coincided with the end of the century, a useful psychological deadline.

After the completion of our Oberon-0 compiler, we made what in hindsight we consider to be our biggest mistake of the project. The decision was made to define a subset of the language, which we called 'Oberon Enough', which was enough to implement the compiler. The thinking behind this was that we could reduce the amount of the compiler that we needed to get working, before the compiler would compile both itself and the final compiler. The problem was, what we were able to leave out (for example LONGINTs and REALs) did not really decrease the complexity of the compiler. The result was a six-month-long detour while we implemented another compiler which we have now discarded. Nevertheless, the experience gained certainly helped us implement the final compiler.

The Oberon Enough compiler, based heavily on the Project Oberon design, compiled itself successfully and reliably in MS-DOS by the end of October 1997, the ten-month mark. This meant that we were now self-sufficient in the tools we were using to create our tools - indeed Oberon Enough was also used for some minor customer projects. In the two years before it was eventually replaced by our final portable compiler, we discovered only three or four bugs, each of which was fixed in a matter of minutes. The benefits of building our own tools were already becoming clear.

5. Building a Portable Compiler

Having gained experience with Oberon-0 and Oberon Enough, we now embarked on the design of our ultimate, portable compiler.

The central idea became clear - to keep it simple we would not attempt to output optimal code. Twenty- to fifty-fold improvements in the speed of semiconductor technology between 1987 and 1997 had released us from the obligation to squeeze every ounce of performance out of a target machine. We even discovered, for example, that choosing a simpler register-allocation and spilling scheme than in Oberon Enough actually generated smaller and faster code in all but the most complex expressions. Wirth reports similar discoveries [1] - where a far less complex design turns out in practice to perform almost as well as a 'clever' one.

As a result, we made many machine-independent decisions about how code was to be generated. This enabled us, starting again with the Project Oberon design, to migrate all of the machine-specific parts of the compiler into a single module, of around a thousand lines of source code. A few machine-dependent variables (pointer size, stack alignment size) persist, which are set initially and are accessible by the rest of the compiler where necessary.

We then realised that by having the machine-dependent code generation module install itself into the rest of the compiler via a small set of up-calls, or procedure variables (function pointers in other languages), we could make the compiler re-targetable at run time. These up-calls (e.g. Move, Addr, BinOp etc.) are precisely the procedures which need to be written for a new processor. By constraining the interface of the code generator, it had become much easier and quicker to port the compiler to new architectures.

It is often tempting when developing a compiler to add features to the language. We kept these to a minimum, implementing only two significant extensions which were absolutely essential to us: a floating-point binary-coded decimal type for quantities such as money, and open-array variables supporting string concatenation. String handling in the original Oberon language was not sufficient for us efficiently to develop commercial applications, where character strings are extensively manipulated. A way was found to expand on Oberon's idea of an open-array parameter, and also provide an infix "+" operator for strings, without radically changing the Project Oberon design.

Complementary to the compiler code generator is the run-time environment provided, around which application software can be created. In the Oberon system, this includes a kernel, containing low-level functions such as memory-allocation, and also higher-level libraries like user-interface operations. Conversely, in the case of application software like ours, the run-time library provides a machine-independent

layer, which in turn calls on the services of the target operating system such as Microsoft Windows.

The Windows 32-bit environment was targeted as the first test of our portable compiler, and a simple kernel and user-interface library was written. This allowed us to create new versions of some of the data-entry applications which we had previously provided to our customers as MS-DOS programs. It was even possible to postpone the implementation of run-time garbage-collection (reclamation of unused dynamic or 'heap' memory); our first few programs simply terminated and restarted themselves with a clean heap at key points transparent to the user.

One of the first applications completed with our new tools was a Windows-based library management system for the National Gallery in London. This has already proved extremely reliable in use over more than eighteen months, and a version is now also used at the Tate Gallery. Development and maintenance of these systems has been straightforward, and the impact (on our application) of major systems changes at both sites has been slight. We believe that all this is due in no small part to the simplicity of the tools used to construct the application software itself.

Once a nearly feature-complete compiler was being used in-house to generate programs for customers, the next step was to complete the 16-bit code generator in order to compile the compiler, obviating the need for our separate Oberon Enough compiler. This was completed by coincidence on the second anniversary of our finishing Oberon Enough, at month thirty-four.

Finally a Motorola 68000 code generator module was added to the compiler. This was written by one person in 52 hours (in 15 days over one month). Around five bugs in other parts of the compiler were uncovered, but no changes to the working design were necessary.

6. Compiler Design Summary

Most design decisions concerning our implementation of the compiler were influenced by simplicity and reliability, rather than performance. First, for example, taking a lead from Java [9], we too 'stick a stake in the ground' and force all basic types to be the same size on each machine. This reduces the number of machine-dependent variables needed by parts of the compiler other than the code generator module; and it also avoids the irritating experience (found for example in the C language) where the numeric range handled by an integer variable differs from machine to machine.

The run-time memory layout is standardised across all machines, using a fixed *arena pointer* (sometimes called static base) for each module, which is usually loaded into an address register and is saved and restored when calling across module boundaries. Module variables and type tags are accessed at negative offsets from this pointer, and other data, such as the arena pointers of imported modules, are accessed at positive offsets.

As explained above, no attempt is made to make the very best use of the resources on any particular processor: controversially, for example, we do not always use all available machine registers, instead providing standard 'spill' locations reserved in each module's arena for use in complex expressions.

Unsupported operations on particular processors, such as long integer division, as well as certain other library routines (some type conversions, floating-point decimal arithmetic, string concatenation) are called through a *trap vector* stored at offset zero in

the arena. Constant data, such as strings, are accessed through a *constant pointer* located in each module's arena immediately after the trap vector.

Encapsulating the interface to the code generator using procedure variables as described above means that the compiler can contain code generators for several machines, selectable by a command-line option or other method. In addition to the convenience of having only one compiler to use, this enhances the maintainability across all platforms.

The approximate size of each module of the compiler is indicated by the figures given below, for a particular platform (MS-DOS, 16-bit Intel 8086):

Module	source lines	object file (bytes)
Scanner	402	5955
Symbol table	671	14111
Object file writer	529	9765
Expressions	983	26025
Statements	404	9276
Main parser	979	19838
Command-line reader	199	4248
Code generators:		
16- and 32-bit Intel	1644	25935
Motorola	825	13324
Total:	6636	128477

7. Writing a New Code Generator

The general method which we found extremely helpful when developing a new code generator is the same as that explained in Project Oberon [1]. Before describing his compiler in detail, Wirth first lists fifteen sample 'code patterns' (we added a sixteenth for our open-array operations), showing what machine code should be output for a particular set of sample Oberon statements and expressions.

These patterns are far from working programs, but as Wirth points out, the discipline of deciding *which* code should be output (usually with a paper and pencil in our case) is a prerequisite, before discovering *how* best to design the generator. Working our way through the patterns kept the complexity of the task in check.

Each code-generation procedure has a standardised interface, and there are 14 procedures in all:

```
Released()  Move()  Addr()  Index()  MonOp()  BinOp()  Branch()
Jump()  Trap()  Case()  PrepCall()  Call()  Enter()  Return()
```

The procedures usually take as parameters one or more variables of type *Item*, a data structure (described in Project Oberon and Compiler Construction) containing information used by the code generator to select addressing modes and sizes of operands:

```
Item = RECORD
  mode, lev: INTEGER; (*addressing mode and scope*)
  a0, a1, a2: INTEGER; (*various*)
  typ: Struct; (*type, e.g. Integer, Record*)
  ...
END;
```

For example, when compiling code for a procedure call, a variable may need to be pushed onto the stack as a parameter. The procedure

```
Move(VAR x, y: Item)
```

will in this case be called with

`x.mode = Stk` (stack), `y.mode = Var` (a static or local variable).

If the variable is a static LONGINT, at offset -4 from the arena pointer, then

`y.a0 = -4`, `y.lev = 0`, `y.typ.size = 4` (and `x.typ.size = y.typ.size`).

Note that type checking has already been performed by the machine-independent parts of the compiler. The `Move()` procedure uses the information from the x-Item to select the addressing mode for the stack, and from the y-Item the addressing mode for the LONGINT variable. These are then combined into one or more instructions to be output as machine code.

We illustrate first how the above Item information is used when generating code for the Intel 80386 processor. On this machine, the stack addressing mode dictates a specialised push instruction which has a binary op-code value of 11111111 aa110aaa; the aa-aaa bits represent the addressing mode of the data to be pushed onto the stack.

In the case of the 80386, where we choose to store the arena pointer in the EBX register, the addressing mode selected by the y-Item will be "EBX register indirect with 8-bit displacement", 01---011. This is combined with the op-code and an 8-bit displacement argument, -4 (FC hex), so that the final result is the issue of the 3-byte instruction

```
FF 73 FC          push   dword ptr [ebx-04].
```

When targeting the Macintosh, the same `Move()` call will instead select a general 68000 "long move" op-code, 0010dddDDDSSSssss, where dddDDD is the destination addressing mode selected by the x-Item, and SSSsss is the source mode from the y-Item (capitals are used here to denote the addressing mode, lower case the register number involved).

The code generator uses the 68000 "address register indirect with predecrement" mode for the destination, using the A7 address register (stack pointer); the binary code for this is DDD=100, ddd=111. For the source (y-Item), where the arena pointer on this machine is held in address register A3, the "address register indirect with displacement" is selected: SSS=101, sss=011.

The op-code word with both sets of addressing mode bits is combined with the 16-bit displacement argument -4 (FFFC hex), culminating in the issue of the two-word instruction

```
2F2B FFFC         move.L (-4,a3), -(a7).
```

8. Conclusion

Rather than continue to tolerate the complexity and unreliability of available commercial application development tools, we decided to build our own, based around the Project Oberon compiler [1].

We have isolated and simplified the code generator interface within the compiler, reducing the amount of work required when targeting a new processor instruction set. This has been brought about by supporting the generation of good, but not 'optimal' code for each target processor, and by standardising the run-time memory layout across different machines and operating systems. Additions to the Oberon language to better support application programming have been kept to a minimum.

A satisfactory portable compiler for application development was the result of three years' part-time work by one person, which we consider to be a reasonable timescale even for a small company. However, we wish to point out that this has been achieved only by adhering to the essentials required, keeping the tools *as simple as possible*.

Acknowledgements

The author wishes to thank in particular Dr. J. M. Benson and also Dr. S. M. Ginn for their encouragement and invaluable suggestions.

References

[1] Wirth, N. and Gutknecht, J. (1992). Project Oberon: The Design of an Operating System and Compiler. Addison Wesley
[2] Reiser, M. and Wirth, N. (1992). Programming in Oberon: Steps Beyond Pascal and Modula. Addison Wesley
[3] Wirth, N. (1995). A Plea for Lean Software. IEEE Computer, vol. 28, no. 2
[4] Fischer, C. and Leblanc, R. (1988). Crafting a Compiler (1st Ed). Benjamin Cummings
[5] Tanenbaum, A. S., van Staveren, H., Keizer, E. G., and Stevenson, J. W. (1983). A Practical Tool Kit for Making Portable Compilers. Communications of the ACM, vol. 26, no. 9
[6] Crelier, R. (1991). OP2: A Portable Oberon-2 Compiler. Proceedings of the Second International Modula-2 Conference, Loughborough University of Technology (ftp://ftp.inf.ethz.ch/pub/software/Oberon/OberonV4/Docu/OP2.Paper.ps.gz)
[7] Wirth, N. (1996). Compiler Construction. Addison Wesley
[8] Lagarias, J. C. (1985). The 3x+1 Problem and its Generalizations. American Mathematical Monthly, vol. 92, no. 1 (http://www.cecm.sfu.ca/organics/papers/lagarias/)
[9] Gosling, J. and McGilton, H. (1996). The Java Language Environment: A White Paper. Sun Microsystems, Inc. (http://java.sun.com/docs/white/langenv/)