

Compiler Optimizations Should Pay for Themselves

Applying the *Spirit of Oberon* to Code Optimization by Compiler

Michael Franz

Institut für Computersysteme, ETH Zürich, CH-8092 Zürich, Switzerland

Abstract

Optimizing compilers tend to be much larger and much slower than their straightforward counterparts. Their designers usually do not follow Oberon's maxim of making things "as simple as possible", but are inclined to completely disregard cost (in terms of compiler size, compilation speed, and maintainability) in favor of code-quality benefits that often turn out to be relatively marginal. Trying to make an optimizing compiler *as simple as possible* and yet *as powerful as necessary* requires, before all else, a measurement standard, by which both simplicity and power can be judged.

For a compiler that is written in the language it compiles, two such standards are easily found by considering first the time required for self-compilation, and then the size of the resulting object program. With the help of these benchmarks, one may pit simplicity against power, requiring that every new capability added to the compiler "pays its own way" by creating more benefit than cost on account of at least one of the measures.

This paper reports of a project for adding an optimization phase to an existing Oberon compiler, resulting in a higher object-code density and an improved run-time performance of compiled code. The beneficial effects of the optimization are greater than the costs, so that the improved compiler is smaller than its predecessor when compiled by itself, while it runs almost as fast.

Keywords: Software Engineering, Compilers, Optimization, Programming Languages, Oberon.

1. Introduction

The *Spirit of Oberon* can be summarized by the guiding principle of "making it as simple as possible, but not simpler" [Wir88, WG89]. Optimizing compilers, on the other hand, are characterized by an enormous and ever-increasing complexity. As their authors keep on adding functionality for handling even rarely-occurring constructs, these compilers grow to immense sizes while being slowed to appalling levels of performance. Were it not for the simultaneous advances in the speed of hardware, many optimizing compilers would long have collapsed under their own weight.

At first sight, therefore, optimizing compilers seem to exist in a realm to which the Spirit of Oberon has no admittance. Nevertheless, this paper reports of a modestly optimizing compiler that is

claimed to be inspired by Oberon. In the context of compiler construction, the principle of "making it as simple as possible, but not simpler" mandates that only those optimizations are incorporated into a compiler that actually benefit a "model program" regarded as being typical for the problem domain. Considering that compilers are substantial and non-trivial pieces of software in their own right, and that many of them are written in the language they compile, one might then suggest that each compiler itself should serve as its referential "model program". This approach introduces the practical benchmarks of *self-compilation time* and *self-compilation object-size*.

Both of these benchmarks pit a compiler's complexity against its performance. By imposing an upper limit on either or both of the measures, any attempt to introduce badly chosen or even superfluous optimizations is warded off effectively. Only those optimizations that bring more gains than expenses will be able to pass. For example, the addition of optimization routines is likely to increase the source-code size of the compiler, but this doesn't automatically mean that the object-code size after self-compilation will grow as well. The situation may very well arise that an optimization increases the object-code density by so much that this effect completely counteracts the expansion due to greater functionality.

It is exactly this result that has been achieved in the construction of an optimizing variant of the original Oberon compiler for the Apple Macintosh. While the object-code improvements effected by the new compiler are admittedly modest in comparison with some commercial offerings, they are non-trivial and deliver some astounding performance increases at a limited expense. The remainder of this paper will illuminate the principles behind these optimizations in more detail and give some performance data.

2. Foundation

The optimizing compiler described in this paper is based on the original Oberon compiler for the Apple Macintosh that is part of the *MacOberon* system [Fra93]. This earlier compiler can justly be called "as simple as possible, but not simpler", as it generates code in a straightforward manner, and does so reliably and quickly. Somewhat surprisingly, the quality of the code that this simple compiler produces is already quite acceptable and need not shun comparisons with that of commercial C compilers much larger and slower.

The original Oberon compiler is straightforward in the sense that it generates code on a statement-by-statement basis and performs no optimizations across statement boundaries. It allocates all variables in memory and releases all registers at the end of every statement. The compiler is integrated into the *MacOberon* environment, which provides a facility for inspecting the values of global variables interactively, and a mechanism for displaying a symbolic stack dump upon a user-initiated break signal and whenever a run-time test fails. In order to remain backward-compatible with the symbolic debugging features of this environment, it was decided not to allocate variables in registers in the new compiler either. Consequently, the new compiler *also* relinquishes all registers to the free-register pool at the end of every program statement.

The major difference between the old and the new compiler, however, is that the new compiler is able to *re-activate* registers from the unused-register pool, i.e., utilize the values stored in them multiple times, without having to re-load them from memory. It does this by monitoring specific values as they flow through a program, and by remembering which of these values is currently stored

in what register. This sort of optimization is by no means original, nor does its optimization yield come even close to that of more modern techniques. The point is, however, that with a few well-chosen simplifications, this approach strikes an excellent balance between effort and effect and is able to "pay its own way".

3. Value Numbers

The optimization built into the new compiler is a modest one. It works by tracking the flow of *certain* values through *certain* parts of a program, but constitutes neither a global data-flow analysis, nor does it even do as much as eliminating all the common subexpressions occurring entirely within basic blocks. Nevertheless, the implemented technique based on *value numbering* performs remarkably well in practice.

A value number can be seen as a version stamp that identifies the current contents of a variable uniquely; each time that the variable is changed, it will receive a new value number. Likewise, variables that are equal to each other for certain extents of a program can be assigned the same value number for the range of their equality. This simplifies the task of removing redundant computations from a program later, as it can be based on a comparison of value numbers, instead of having to inspect parts of the program over and over. The problem lies in identifying the equalities in the first place – unfortunately, the problem of deciding whether two computations are equivalent is undecidable in general (it is *NP*-complete), so that practical implementations allow only incomplete solutions.

Value numbering is an old technique in the field of compiler construction [CS70]. Many variants of the method are similar to a *symbolic execution* of the program being compiled, during which the compiler collects information about individual program-expressions and compares this with data it has collected about other expressions occurring elsewhere in the program. In recent years, value-numbering schemes based on transforming the program to Static Single Assignment form (SSA) have become popular [AWZ88, RWZ88]. These are used in compilers of high sophistication, mainly for RISC architectures.

The technique used in the implemented compiler is patterned after the symbolic execution model. The new compiler interposes an additional processing step between the parsing and code-generation passes. During this extra pass, the abstract-syntax-tree representation of the program, which has been generated by the compiler's front-end, is taken and some of its nodes are annotated by value numbers. The structure of the abstract syntax-tree itself is left completely unchanged in this process; the only additional information passing to the back-end in the new compiler is two additional data items in each syntax-tree node: an integer field that represents a value number and a flag that indicates whether run-time tests can be suppressed (as will be explained below).

Rather than stamping every *expression* in the abstract syntax-tree with a value number, the implemented compiler concerns itself only with *designators*. This seems to be a sensible compromise between effort and effect, considering that in order to perform value numbering, information about all expressions to be stamped need not only be stored, but also *searched repeatedly*. The drawback is that some not-so-uncommon program patterns cannot be optimized in this manner; for example, if the expression "x+1" were to appear fifteen times in a basic block, only the "x" part of each would be handled by the implemented optimizer.

4. Data Structure and Algorithm

During the value-numbering phase, the abstract syntax-tree is traversed recursively. Each node is visited exactly once, in the same order in which the code generator will eventually repeat this process. In the course of the traversal, information about designators and their associated value numbers is collected in *value sets*. At any point during the traversal, the current value set contains those designators for which the compiler can be sure it knows their value number. Value sets are preserved across certain kinds of branches, but not across procedure calls and not upon entering or leaving loops. This greatly simplifies the implementation.

Value sets are implemented simply as linked lists of designator descriptors. The designators in them are hierarchic; for example, a designator representing a record can have descendants representing the individual fields. The descendants of a designator are kept together also in a linked list that is anchored in the ancestor. As an illustration, a value set representing the designators p , $p\uparrow$, $p\uparrow.f$, $p\uparrow.g$, r , $r.f$, and $r.g$ would have the structure depicted in Figure 1 (r being of a record type with fields f and g , and p being of an associated pointer type).

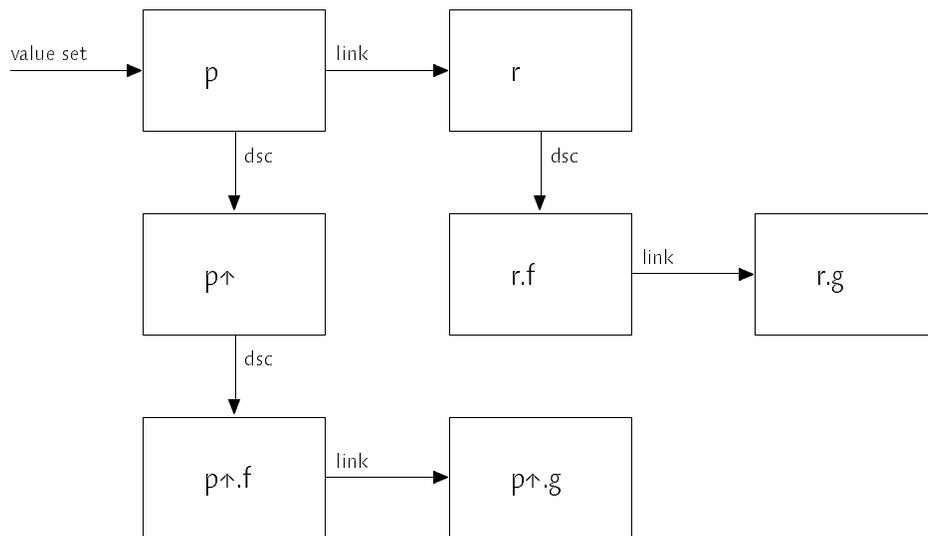


Figure 1: Sample Value-Set Data Structure

The basic algorithm then becomes straightforward: starting with an empty value set, statement sequences are processed on a statement-by-statement basis. Each time that a designator is encountered in one of the subexpressions, the current value set is searched to see if the designator is already present. If it is, then this signifies that the value of the designator hasn't changed since its last use and the syntax-tree node corresponding to the designator can be stamped with the value number of the designator.

If the designator cannot be found in the current value set, then there are two possibilities: the designator may not have shown up in the program so far, or its value-set entry may have been

invalidated because the optimizer could no longer be sure that the corresponding value hadn't changed. In either case, a new designator is constructed and added to the value set, a new unique value number is generated, and both the new designator and the syntax-tree node corresponding to it are stamped with it.

When an assignment is come across, there are again two possibilities. The expression being assigned may in fact be a designator. In this case, one can copy the value number and all descendants from the source to the destination designator. Otherwise, the destination designator needs to receive a completely new value number and all of its existing descendants have to be thrown away.

5. Aliases

Unfortunately, value numbering is not as simple as the previous section suggests, due to the existence of *aliases*. In most programming languages, there are several situations that allow the same storage location to be accessed by way of more than one designator. For example, consider the following procedure in the programming language Oberon:

```
PROCEDURE P(VAR x, y: INTEGER);
BEGIN  x:=2*x; y:=3*y
END P;
```

In the program fragment above, it may happen that the VAR-parameters x and y refer to the same variable, for example, if the procedure is called passing the same actual parameter twice, as in "P(v, v)". Consequently, when compiling the statement "y:=3*y" above, one must not assume that the y required on the right side of the assignment is the same one that was required on the right side of the previous statement.

At this point, a very clever compiler would probably examine all call locations of P , determining at each call site whether the actual parameters could be aliases of each other (note how the alias property can propagate). It could then create two different versions of procedure P , one in which x and y are assumed not to be aliases of each other and another variant in which one cannot be so sure, and route each of the different calls of P to the appropriate compiled version.

Needless to say that such an approach is *not* as simple as possible. In keeping with Oberon's motto, the implemented compiler uses a conservative strategy and simply leaves unoptimized what cannot be optimized safely at first sight. In order to guarantee correctness of compilation for any legal Oberon program, it is therefore necessary to assume always the worst and treat all possible aliases as if they were indeed present.

Aliases can occur not only when using VAR-parameters, but also with pointers and arrays. Two pointer variables p and q of the same type may point to the same address, and two different index-expressions $a[i]$ and $a[j]$ refer to the identical array element when i and j have the same value. Worse still, the individual alias-effects are independent of each other and can occur in combination, so that even expressions such as $a[i]$ and $b[j]$ might turn out to be aliases, for example if a and b are aliases of the same array, and i and j have the same value.

The implemented Oberon compiler is able to detect when the target of an assignment can have aliases. In these cases, it removes from the current value set all other designators that could possibly

be aliases of the variable being modified in the assignment. Designators that can be aliases of each other in this manner are VAR-parameters, variables from outer scopes, values accessed via pointers, and values accessed through array references. However, in a strongly-typed language such as Oberon, aliases need to have *the same type* as the variable that has been modified, which reduces the number of designators in the value set that are actually invalidated on such occasions.

6. Run-Time Test Suppression

As a side-effect of value numbering, the implemented compiler is also able to *suppress* many of the *run-time tests* that would otherwise be necessary for ensuring program integrity. For example, dereferencing a pointer is possible only if the pointer is different from *NIL*. In the absence of memory protection on the Macintosh, this test must be performed explicitly at run-time before dereferencing. Similar run-time checks are required to ascertain that index expressions lie within the permissible index range of the referenced array, and for supporting the type guards of the Oberon language.

Recall now that the new compiler keeps track of designators in value sets. If, at the time of dereferencing a pointer variable p , the designator $p\uparrow$ is present in the value set, then that means that the same p has been dereferenced earlier on and need not be tested for *NIL* again. It might very well be the case during code generation that $p\uparrow$ is no longer present in any register and needs to be re-loaded from memory, but at least the *NIL*-test need not be repeated. Index checks and type guards can be avoided in an analogous way.

In some cases, this strategy can even be improved upon, by *adding designators tentatively* to the value set at specific points. For example, the standard procedure *NEW()* in Oberon never returns *NIL*, but generates a run-time trap if there is not sufficient memory to fulfil the storage allocation request. Consequently, the new compiler is able to add a designator $p\uparrow$ with a new value number to the current value set directly following each *NEW(p)* statement, as it can be sure that dereferencing p will not fail at that point.

7. Code Generation

The code generator of the original Oberon compiler underwent major modifications only in the routines responsible for register management, while most other parts remained virtually untouched. The only *code pattern* that was changed was the one used for comparing expressions. Here, the *TST* instruction, which compares the contents of a register or memory location with zero, was replaced by a *load instruction* that sets condition-codes as a side-effect. The change was based on the observation that designators occurring in comparisons are frequently referenced again elsewhere in the same program. Since the *TST* instruction costs the same as the corresponding load instruction, both in instruction length and in execution time, the modification has no adverse effects, yet it often enables a following load instruction to be eliminated completely.

The register-management mechanism was completely redesigned for the new compiler. Apart from monitoring which registers are in use, the new register allocator also keeps track of value numbers that correspond to the current contents of registers. This is not as straightforward as it may

look at first sight, since it requires that the value-number entry corresponding to a register is invalidated whenever an instruction is generated that modifies the register's contents.

In the new compiler, even "unused" registers may still contain potentially valuable data that should be preserved for future re-activations. It is therefore desirable to utilize as few registers as possible when compiling expressions, so that a minimum of cached information is destroyed. The implemented code-generator minimizes the number of registers required for evaluating an expression. It is capable of distinguishing between *source* and *destination* registers when emitting individual instructions, and releases no-longer-required source registers to the unused-register pool for immediate re-use. This is in contrast to many other simple compilers that release registers only after compiling complete statements.

For the reservation of new registers, a *priority scheme* is used to reduce the overwriting of registers that still have the potential of being re-activated. Under this scheme, registers that might still contain a cached value are used in the compilation of expressions only when there are no longer any unused registers with invalidated value numbers left.

8. Measurements

The tables in this section compare the new Oberon compiler using value numbering ("new") to the latest generation of the MacOberon compiler before the introduction of this optimization ("old"), and to a commercial C compiler. Apart from the value-numbering-related changes, the two Oberon compilers are identical, meaning that the "old" compiler incorporates already some improvements over the last MacOberon compiler for which performance data has been published previously (in [Fra94]).

Table 1 sets the object-code sizes of the two Oberon compilers against each other. After the introduction of value numbering, the back-end of the compiler grew by almost 20% in object-code size when cross-compiled using the old compiler. However, the new compiler is not only more complex, but also generates denser object code. When the new compiler is compiled by itself, the resulting binary code turns out to be smaller than that of the old compiler.

	Front-End	Back-End	Total
Self-Compilation of Old Compiler	69978	43264	113242
Cross-Compilation of New using Old	69978	51184	121162
Self-Compilation of New Compiler	62038	47444	109482

Table 1: Object-Code Sizes (Bytes)

Table 2 compares compilation times on two different machines, namely a *Macintosh IIfx* (MC68030 Processor running at 40MHz) and a *Macintosh Quadra 840AV* computer (MC68040 Processor, also running at 40MHz). The measurements show that value numbering slows down compilation only on the older of the two machines. On the Macintosh Quadra 840AV, cross-compiling the new compiler takes equally long as its self-compilation, although it still takes longer than self-compilation of the old

compiler due to a larger source-code size. Nevertheless, this small self-compilation-time penalty seems justified in light of better object code.

	Macintosh IIfx	Quadra 840AV
Self-Compilation of Old Compiler	18.1	8.3
Cross-Compilation of New using Old	19.0	8.9
Self-Compilation of New Compiler	19.8	8.9

Table 2: Compilation Times (Seconds)

The last two tables compare the code quality of the two Oberon compilers with each other, and with that of the *Apple MPW C compiler* for the Macintosh (Version 3.2.4) with all possible optimizations for speed enabled ("–m –mc68020 –mc68881 –opt full –opt speed"). Both tables list execution times in milliseconds (less is better). Due to processor-cache effects, these timings can vary by as much as 15% when executed repeatedly; the figures give the best of three executions. The C version of the *Treesort* benchmark exceeded all meaningful time bounds, due to apparent limitations of the standard Macintosh operating system storage-allocator. The two Oberon compilers are not bound by these limitations, as they generate calls to the storage allocator of MacOberon, which includes an independent memory-management subsystem.

Considering that the two Oberon compilers are faster than the C compiler by a factor of more than ten, and furthermore require no separate linking step, which for C takes yet again as long as compilation, these results can be called respectable. The optimization based on value numbering is able to further boost the code quality of an already adequate MacOberon compiler, while, on the more modern of the machines surveyed, it doesn't even increase compilation time.

Benchmark	Old	New	Change	C
Permutation	367	367	none	403
Towers of Hanoi	434	350	–19%	493
Eight Queens	284	250	–12%	136
Integer Matrix Multiplication	484	483	none	446
Real Matrix Multiplication	667	650	–3%	650
Puzzle	2867	2316	–19%	2450
Quicksort	316	267	–16%	195
Bubblesort	600	383	–36%	358
Treesort	317	250	–21%	> 1000
Fast Fourier Transform	1050	817	–22%	720

Table 3: Execution Times (Milliseconds) on a Macintosh IIfx (MC68030)

Benchmark	Old	New	Change	C
Permutation	83	83	none	113
Towers of Hanoi	83	83	none	121
Eight Queens	50	50	none	43
Integer Matrix Multiplication	150	150	none	173
Real Matrix Multiplication	133	133	none	173
Puzzle	633	566	-11%	803
Quicksort	66	66	none	61
Bubblesort	117	100	-15%	88
Treesort	83	66	-20%	> 1000
Fast Fourier Transform	133	116	-13%	125

Table 4: Execution Times (Milliseconds) on a Macintosh Quadra 840AV (MC68040)

9. Code Sample

In the benchmarks in Tables 3 and 4 above, the *Bubblesort* program emerges as a great beneficiary of the value-numbering optimization. Nevertheless, even the new Oberon compiler is still outperformed by the C compiler in this benchmark. The following example illustrates the gains of the value-numbering method, but also the drawbacks of not being able to optimize arbitrary index expressions, such as "a[i+1]". Consider this program fragment:

```

VAR
  top: LONGINT;
  sort: ARRAY N+1 OF LONGINT;

PROCEDURE Bubble;
  VAR i, j: LONGINT;
BEGIN
  top:=N;
  WHILE top > 1 DO
    i:=1;
    WHILE i < top DO
      IF sort[i] > sort[i+1] THEN
        j:=sort[i]; sort[i]:=sort[i+1]; sort[i+1]:=j;
      END;
      INC(i)
    END;
    DEC(top)
  END
END Bubble;

```

The assembler listings below represent the respective outputs of the two Oberon compilers (with array index checking disabled). It is clearly apparent that the differences between the two code sequences concern only the avoidance of memory accesses in the new compiler. It is also obvious that the code quality of the new compiler could be further improved by optimizing the expressions "i+1" and "sort[i+1]" in the same manner. However, optimizing arbitrary expressions by the value-numbering method would lead to much longer compilation times, as it would result in much larger value sets that would furthermore have to be searched much more often.

	Old Compiler	New Compiler
	MOVE.L #N, (top)	MOVE.L #N, (top)
L0	CMPI.L #1, (top)	CMPI.L #1, (top)
	BLE L5	BLE L5
	MOVE.L #1, (-4, A6)	MOVE.L #1, (-4, A6)
L1	MOVE.L (-4, A6), D7	MOVE.L (-4, A6), D7
	CMP.L (top), D7	CMP.L (top), D7
	BGE L4	BGE L4
L2	MOVE.L (-4, A6), D7	
	LEA (sort), A4	LEA (sort), A4
	MOVE.L (-4, A6), D6	MOVE.L D7 , D6
	ADDQ.L #1, D6	ADDQ.L #1, D6
	LEA (sort), A3	
	MOVE.L (0, A4, D7.L*4), D5	MOVE.L (0, A4, D7.L*4), D5
	CMP.L (0, A4, D6.L*4), D5	CMP.L (0, A4, D6.L*4), D5
	BLE L3	BLE L3
	MOVE.L (-4, A6), D7	
	LEA (sort), A4	
	MOVE.L (0, A4, D7.L*4), (-8, A6)	MOVE.L D5 , (-8, A6)
	MOVE.L (-4, A6), D7	
	ADDQ.L #1, D7	ADDQ.L #1, D7
	LEA (sort), A4	
	MOVE.L (-4, A6), D6	MOVE.L (-4, A6), D4
	LEA (sort), A3	
	MOVE.L (0, A4, D7.L*4), (0, A3, D6.L*4)	MOVE.L (0, A4, D7.L*4), (0, A4, D4.L*4)
	MOVE.L (-4, A6), D7	
	ADDQ.L #1, D7	ADDQ.L #1, D4
	LEA (sort), A4	
	MOVE.L (-8, A6), (0, A4, D7.L*4)	MOVE.L D5 , (0, A4, D4.L*4)
L3	ADDQ.L #1, (-4, A6)	ADDQ.L #1, (-4, A6)
	BRA L1	BRA L1
L4	SUBQ.L #1, (top)	SUBQ.L #1, (top)
	BRA L0	BRA L0
L5		

Summary and Conclusion

A simple Oberon compiler has been augmented by a simple optimization step. Unlike other optimizing compilers, the implemented one allocates all variables in memory and never delays any write operations to them, guaranteeing compatibility with existing debugging tools. The added optimization has the sole effect of avoiding the re-loading of information that has been preserved in registers across statement boundaries. Surprisingly, this innocuously-looking optimization has a profound effect on program length and moreover benefits execution speed. This enables the optimization to pay its own way: The *self-compilation object-size* of the new compiler is less than that of its predecessor, although the source text is larger. The *self-compilation time*, on the other hand, is longer because of a longer compiler source-text, but on a modern target machine, the speed of compilation is comparable to that of the old compiler.

References

- [AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck; "Detecting Equality of Variables in Programs"; *Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, San Diego, California*, 1–11; 1988.
- [CS70] J. Cocke and J. T. Schwartz; *Programming Languages and Their Compilers: Preliminary Notes*, Second Revised Version, Courant Institute of Mathematical Sciences, New York University; 1970.
- [Fra93] M. Franz; "Emulating an Operating System on Top of Another"; *Software–Practice and Experience*, 23:6, 677–692; 1993.
- [Fra94] M. Franz; "Technological Steps toward a Software Component Industry"; in J. Gutknecht (Ed.), *Programming Languages and System Architectures*, Springer Lecture Notes in Computer Science, No. 782, 259–281; 1994.
- [RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck; "Global Value Numbers and Redundant Computations"; *Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, San Diego, California*, 12–27; 1988.
- [Wir88] N. Wirth; "The Programming Language Oberon"; *Software–Practice and Experience*, 18:7, 671–690; 1988.
- [WG89] N. Wirth and J. Gutknecht; "The Oberon System"; *Software–Practice and Experience*, 19:9, 857–893; 1989.