Detection of Side-Effects in Function Procedures

Robert Griesemer
International Computer Science Institute, Berkeley
gri@icsi.berkeley.edu

TR–94–032

August 1994

Abstract

Procedural programming languages usually do not support side-effect free functions but merely a form of function procedures. We argue that functions should be free of (non-local) side-effects, if they are considered as abstraction mechanism for expressions. While it is easy to statically detect side-effects in functions that do not dynamically allocate variables, this is no longer the case for functions that do create new data structures. After giving a classification of different levels of side-effects, we describe a simple and efficient method that allows for their dynamic detection while retaining assignments, i.e., without referring to a pure functional implementation. The method has been implemented for an experimental subset of Oberon.

1 Introduction

Procedural programming languages usually provide two different concepts for specifying computation, namely *expressions* and *statements*. In a first step, we consider only expressions that do not call functions, and only statements that do not call procedures. We call them *simple expressions* and *simple statements*, respectively.

A simple expression specifies the computation of a new *value* by applying operations on other values, its operands. The result of an operation depends only on its operands and the operation is expected to do nothing else besides computing and returning a result. Thus, we regard these operations as functions in a mathematical sense. Note that in programming languages operations are frequently denoted by *operators*, but usually not all operators denote operations in our sense.

A simple statement in turn specifies computation of new *state* by modifying previous state via *actions* (e.g., assignments). The current state of a program is constituted by the current set of values of all its variables. The repeated execution of a (simple) statement leads to a path in the state space [1] of a program whereas the repeated evaluation of a simple expression does not change the current state and always yields the same result. Thus, while the *effect* of executing a (simple) statement is the modification of state, evaluating a simple expression affects nothing besides computing a value, i.e., simple expressions are *free of side-effects*. This property distinguishes them semantically from being statements.

Programmer-defined *functions* and *procedures* raise the abstraction level of expressions and statements, respectively. Both methods may be used recursively as building blocks for even higher abstractions, i.e., functions may be called within expressions contained in other functions, and procedures may be called within statement sequences in the body of other procedures. Generally, it is important that an abstraction does not weaken the essential properties of what it abstracts from. Consequently, if functions are used to abstract from simple expressions, they should be free of side-effects.

Unfortunately most procedural programming languages do not support side-effect free functions. In these languages, a function is nothing else but a procedure that specifies a return value. Therefore, sometimes the term *function procedure* is used [7]. Because procedures can have side-effects (it is their very purpose), they are usually allowed in function procedures, too. Thus, expressions are not always side-effect free in these languages. On the other hand, if functions are generally side-effect free, expressions are always side-effect free as well, regardless of how complex they are and which functions they call. Then, the only way to produce a side-effect is via assignments or, indirectly, via procedures. If expressions and statements are syntactically distinct, the side-effect producing spots in a program can be located easily. This significantly simplifies correctness proofs and thus enhances the readability of programs. Furthermore, a compiler can produce better code by exploiting expressions being side-effect free.

There are various approaches for the detection of side-effects in functions. Typically, assignments are restricted (or completely disallowed), which permits to detect side-effects statically, i.e., at compile-time. These approaches have the disadvantage of restricting a programming language more than necessary. In this paper, we describe a hybrid (i.e., partially static and partially dynamic) method that does not suffer from such restrictions. Especially, it allows functions to create and return new data structures, as long as the functions remain side-effect free. In the following, we assume that all functions must be side-effect free, although a programming language could support both, function procedures and side-effect free functions. In Section 2, we introduce some terminology and specify the problem. In Section 3,

the hybrid method is presented. Variants of our scheme and applications are discussed in Section 4. Finally, we conclude in Section 5.

2 Terminology and Problem Specification

In the following, a *function* refers to a procedure that specifies a result value (a function procedure). A *variable* serves to hold *values* of particular type(s), one value at any time. We distinguish between *static* (not to be confused with the C storage class *static*!) and *dynamic* variables ("objects"). A static variable is introduced via a *variable declaration* that specifies the type(s) of the variable and binds it to a *variable name*. Later on, the static variable is referred to via its name. Dynamic variables are allocated at program run-time and are referred to via a special identifying value, a *pointer*. A pointer is held in a *pointer variable*. A dynamic variable is accessed by dynamically *dereferencing* such a pointer variable. A particular pointer variable may point to different dynamic variables at run-time. A variable that is statically declared local to a function and that exists only for a particular instantiation of the function is called *local variable* of that function. All other variables are called *non-local variables*. Note that call-by-value parameters are considered as local variables while call-by-reference parameters merely provide local *names* for non-local variables and therefore are considered non-local also. Note that non-local does not necessarily mean global, since a non-local call-by-reference parameter may well be local to another procedure. Altogether, we can divide all variables into three classes (Figure 1).

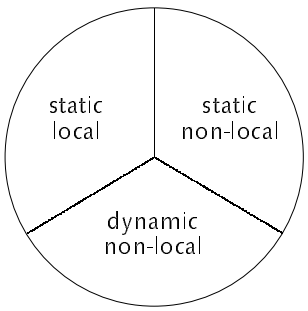

Figure 1 Variable classes

We make use of a simplified assignment syntax: in general, a *designator* is used to refer to a variable, it consists at least of a *variable name* followed by an arbitrary long sequence of *selectors* (such as record field selectors or array indices). Dereferenciation is denoted by an arrow (↑). Using this syntax, dynamic variables are referred to via designators that contain at least a single arrow. Thus, references to dynamic variables can be isolated syntactically. Note that even if dereferenciation is not explicitly visible (such as in Oberon, where p↑.f can be abbreviated to p.f), usually a compiler has the required information anyway; i.e., our notation is not a technical necessity but a means to clarify a point.

```
Assignment = Designator ":=" expression.
Designator = variableName {selector | "↑"}.
```

A *side-effect* is any assignment during evaluation of an expression, or to be more precise, "anything a (function) procedure has done that persists after it returns a value" [6]. Because a particular function instantiation comes with its own local state and any modification of this state does not persist after returning the result, assignments to local variables cause local side-effects but the function can still be considered as being side-effect free on the level of the expression calling it. Consequently, only state changes that are *observable* from outside a function are relevant as side-effects (pathological cases such as functions that actually modify global variables but then reset them to their previous values are considered as not being side-effect free). Using this observability criterion, we can construct the following tower of function categories (see also [4]):

- *R-functions* are not allowed to modify any non-local variables. These functions are perfectly side-effect free because no non-local state change is possible. In particular, for a given global state and given arguments, the same value is always returned. Therefore R-functions are *referencially transparent*.

- *S-functions* are not allowed to modify any non-local variable that existed *before* the function has been called. These functions do not modify the existing state, but may extend it by allocating and returning new dynamic variables. Therefore, it is assumed that dynamic variable allocation is an intrinsic operation of the programming language and that its mechanisms are *not visible* to a program. Then, no side-effects can be observed. However, for a given state and given arguments a function may return different pointer values. If the language features an equality test on pointers, S-functions are not referencially transparent anymore. They are always *structurally transparent*, though.

- *N-functions* are allowed to modify any variable. They allow for arbitrary side-effects without restrictions and hence are *not transparent*.

We can further distinguish between *static* and *dynamic side-effect freeness*. Of course, any statically side-effect free function is also dynamically side-effect free, but it may well be that a function contains an assignment to a non-local variable (a static side-effect) which is never executed at run-time (a dynamic side-effect). The presence or absence of dynamic side-effects depends not only on the structure of a function but also on its parameters and possibly non-local variables, while static side-effect freeness garanties that no side-effect will ever occur independently of any input data.

Purely functional programming languages ensure static side-effect freeness by disallowing assignments [3]. In procedural languages, static side-effect freeness can be guaranteed by restricting assignments within functions to *local variables* only. Such a restriction can be enforced easily by controlling the left-hand side of assignments and by disallowing procedure calls from within a function. This method has been used in Euclid [5], a descendant of Pascal, and in the experimental language Oberon-V [2]. Unfortunately it restricts these languages to R-functions, which do not allow new data structures as results, quite a useful feature. For instance, it is not possible to implement an R-function add(x, y: ↑T): ↑T that takes pointer arguments x and y, allocates an "add" node that refers to x and y and then returns a pointer to this node. Using a procedure add(x, y: ↑T; VAR res: ↑T) that returns the result as a call-by-reference parameter would do the job but makes it impossible to chain several add operations without using temporary variables (e.g., add(add(x, y), z)).

The problem is thus to detect whether an S-function generates a side-effect or not. While

almost the same static technique can be used as for R-functions, now assignments to dynamic variables are allowed sometimes. They are allowed whenever the dynamic variable did not exist before the function has been called; i.e., when the variable is not *older* than the function. Unfortunately there is no simple method to detect this statically (without restricting assignments more than desired) because a designator p↑ may denote *different* dynamic variables during run-time, while a designator for a static variable always denotes the same variable (instance). Instead of relying on (conservative) data-flow analysis techniques, we strive for a more direct method in the following.

3 A Hybrid Approach: The Birthdate Method

Detecting whether something is new or old is usually done by looking at the age of the object in question. Similarly, by attributing dynamically allocated variables as well as function instantiations with a *time-stamp* representing their *birthdates*, a simple comparison of birthdates enables side-effect detection: whenever an assignment to a dynamic variable is done within a function, additionally its birthdate must be compared with the birthdate of the function. If the variable is older than the function, the function has generated a side-effect, and an exception should be raised.

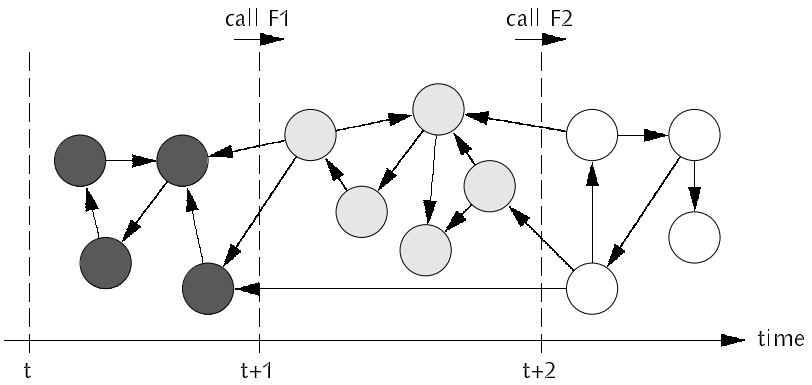

Figure 2 Variables of different age

Figure 2 illustrates the birthdate method with two function calls F1 and F2. The dark grey variables have been allocated at some time t, i.e., they all have birthdate t. When function F1 is called, the global clock is advanced by one and hence all the variables allocated by F1 (the light grey ones) as well as the current instantiation of F1 get the birthdate t+1. Within F1 it is legal to modify variables that are not older than t+1 and it is especially allowed to assign pointers referring to older variables to them. Thus, the light grey variables may refer to the dark grey ones but not vice versa as long as F1 is active. After calling F2 and advancing the clock to t+2 the same story holds for the F2 variables: they may refer to everything allocated before F2 has been called but not the other way round as long as F2 is active. After leaving F2 and returning to F1, F1 may also modify white variables because they are younger than F1, but F1 is still not allowed to modify a dark grey variable. Note that it does not suffice to compare the birthdate of a variable with the current time but it must be compared with the birthdate of the function instantiation in which the assignment is done. Otherwise it would not be possible to modify variables allocated by a particular function after termination of a locally called function. For

instance, in Figure 1 it would not be possible for F1 to modify previously allocated (and modified!) light grey variables after F2 has returned.

At first glance this scheme seems quite expensive with respect to memory and run-time requirements. Fortunately there are several opportunities for improvement. For instance, in many cases the check is required only once for a series of assignments:

... **p↑.f1 := e1**; p↑.f2 := e2; p↑.f3 := e3 ...

In this (typical) assignment sequence where different fields f1...f3 of the same dynamic variable p↑ are set, it is sufficient to do the run-time check only for the very first assignment. In a compiler that features common subexpression elimination (CSE), subsequent checks are eliminated by the CSE algorithm also. Note that even a function call in one of the expressions e1...e3 does not inhibit CSE for the whole sequence of assignments if functions cannot produce side-effects (and therefore cannot change p)! Furthermore, if the assignment sequence is preceded by an allocation statement for p – a relatively frequent program pattern – no check is required at all:

... **NEW(p)**; p↑.f1 := e1; p↑.f2 := e2 ...

In functions that do neither allocate new objects nor call other functions that return pointers, any assignment to a dynamic variable must be an assignment to a previously allocated variable and thus is a side-effect. Such functions need no time-stamp and no increment of the global clock and any side-effect can be detected statically. In fact, these functions are R-functions.

Every dynamic variable needs an additional field for its birthdate. In object-oriented environments, dynamic variables usually carry additional information already, typically a type tag, a pointer to a method table, or a combination thereof. Because the *exact* birthdate information is only required for dynamic variables that have been allocated within functions (dynamic variables allocated in procedures are older anyway) a more compact representation can be chosen: if at least a single bit of the tag field is unused, it may be used to indicate whether a birthdate field is available or not (Figure 3). The run-time check becomes slightly more expensive – assignments to dynamic variables are allowed only if the indicator bit is set and the variable is not older than the function.

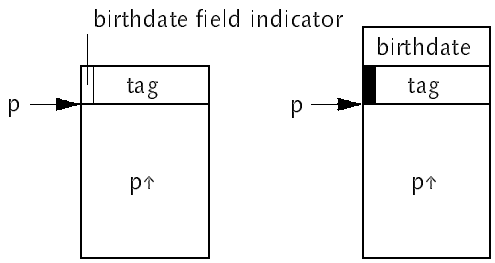

Figure 3 Layout of dynamic variables

A problem not to be ignored is that of time-stamp overflow. The birthdate check will fail if functions are called frequently, for instance, in a continously running system with a 32-bit counter. Hence, the global clock and all the time-stamps of dynamic variables must be reset periodically. Resetting the time-stamps is done via a sweep through the heap (ideally coupled with garbage collection). If resetting is necessary while functions are active, the time-stamps

must not be set to zero but decremented by the birthdate of the oldest active functions. Because the oldest active function is always a function that has been called from within a procedure, the birthdate of the oldest active function can be held in a global variable that is adjusted whenever a procedure calls a function. In an environment like the original Oberon System [8], where garbage collection occurs only between execution of commands, resetting the time-stamps is best combined with garbage collection and the time-stamps can always be reset to zero.

In systems where memory allocation is under full control of the language implementation, better solutions can be found for representing the birthdate. For instance, if dynamic variables are allocated on ascending addresses, a variable's address could be used instead of the birthdate. Ascending address placing could be achieved via a memory management unit or by using special heap allocation methods (for instance together with a stop-and-compact garbage collector that allocates variables consecutively). However, we have not yet investigated such techniques.

We have implemented a simple version of the birthdate method in a compiler for a subset of the programming language Oberon [7]. Every dynamic variable contains an additional time-stamp field for the birthdate, no redundant checks are eliminated and no clock resetting is done. The implementation effort for integrating side-effect checks into the compiler was around a dozen lines of code. The compiler first generates an intermediate program representation in form of an attributed syntax tree which is then traversed to generate ANSI C code. Figure 4 shows the syntax tree for an assignment to a dynamic variable: the last dereferenciation on the left-hand side of the assignment yields the dynamic variable in question. In our syntax tree it is the first ↑ node from top in the left branch of the assignment node. During code generation, additional side-effect check instructions are emitted for this node.

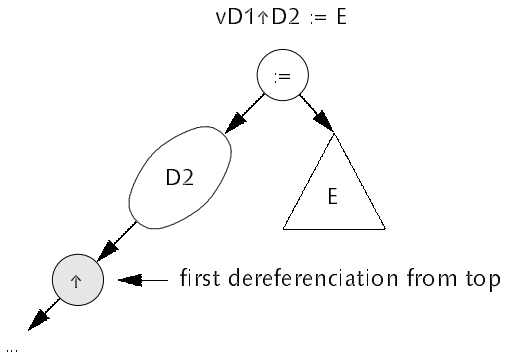

Figure 4 Syntax tree for assignment to dynamic variable

The measured time overhead introduced by a single side-effect check is 0.4 microseconds or about 10 cycles on a 25MHz SPARCstation ELC, corresponding to 10 instructions. This time includes incrementing the global timer once after entering an S-function (1 instruction) and setting the function time-stamp which resides in a register (1 instruction). Thus, currently a side-effect check costs about 8 cycles. A compiler that generates native code could of course do a better job: essentially a load instruction is used to read the variable's time-stamp into a register, a compare and a conditional trap instruction does the rest. Thus, a dynamic test should not cost much more than 3–4 instructions in average; at least as long as a more sophisticated representation for birthdates is not chosen. Furthermore, if precise exceptions are not required, the check instructions may be inserted at any appropriate position and need not to be put immediatly before or after the code for the assignment.

4 Discussion

In the previous section, we assumed that a compiler implicitly emits additional check instructions for assignments to dynamic variables. We can also think of a more explicit method, where the programmer has to use a *guard* that ensures that a dynamic variable is not older than the active function, very much like Oberon *type guards* are used to ensure that a variable is at least some given type [7]. A possible notation could be

```
p↑(NEW).f1 := e1
```

where (NEW) ensures that p↑ is not older that the currently active function. An assignment to a dynamic variable without guard would then be statically illegal (as it is the case for R-functions). While such a solution would have the advantage of making a programmer more aware of the critical assignments, it has the disadvantage of being necessary for *every* assignment to a dynamic variable. Indeed, the critical assignments are *exactly* the assignments to dynamic variables and not only a few of them. This stands in contrast to the usage of type guards in Oberon, which are required only under certain circumstances. Therefore we think that an implicit solution is perfectly adequate.

What if it seems that producing a side-effect in a function is just the right thing to do in a particular situation? We refrain from judging good from bad, but we strongly believe that in most cases either a procedure could do the same job or the program could be organized differently without sacrificing general side-effect freeness of expressions (see below). For instance, in the whole Oberon System [8], almost all function procedures are essentially S-functions. Only a few function procedures produce side effects (even visible ones – for instance, an Oberon window is opened), and they would probably better be written as proper procedures. However, sometimes an escape from the golden cage is needed: in modular languages like Modula-2, Oberon or Ada, a compiler-known module SYSTEM is available that provides low-level operations. Making use of SYSTEM is considered inherently unsafe. SYSTEM could also provide an intrinsic procedure ASSIGN(v, e), where v stands for a variable and e for an expression of a compatible type. ASSIGN would simply assign e to v without side-effect check. Because it is a compiler-known procedure, it could be legal (but of course unsafe) to call it from within a function.

A few more common situations deserve closer attention. As a first example we look at a random number generator:

```
VAR seed: INTEGER;

PROCEDURE Rand (max: INTEGER): INTEGER;
(* returns a random value x with 0 ≤ x < max *)
BEGIN seed := MagicFunction(seed); RETURN seed MOD max
END Rand
```

It seems quite natural to regard Rand as a function taking one argument (max). However, note that it is the very purpose of Rand not to return twice the same value in two consecutive calls with the same argument, or – to make the point even more obvious – simplifying Rand(10) + Rand(10) to 2*Rand(10) is usually wrong. While it would be trivial to replace the side-effect producing assignment by SYSTEM.ASSIGN(seed, MagicFunction(seed)), using a proper procedure instead of a function is not a burden and does not pretend functional properties that

do not exist for Rand:

```
PROCEDURE Rand (max: INTEGER; VAR x: INTEGER);
(* returns a random value x with 0 ≤ x < max *)
```

And of course, Rand(10, x); Rand(10, y); ...x+y... makes explicitly clear what is going on. At this point it might be appropriate to emphasize that a clearer and better understandable programming style frequently does not mean shorter in program length.

As a second example we consider the use of boolean-valued function procedures where the result value is used to indicate a "passed or failed" status. A common program pattern may look like this:

```
IF P1(...) & P2(...) & P3(...) THEN (* P1 and P2 and P3 successfully executed *)
ELSE (* P1 or P2 or P3 failed *)
END
```

The usual argument for this program pattern is that it is much shorter and much more readable than the following aquivalent sequence of nested IF's:

```
P1(..., ok);
IF ok THEN P2(..., ok);
   IF ok THEN P3(..., ok);
      IF ok THEN (* P1 and P2 and P3 successfully executed *)
      END
   END
END;
IF ~ok THEN  (* P1 or P2 or P3 failed *)
END
```

We present the following arguments against this claim: first of all, if proper error handling has to be done, in most cases the failing procedure and the actual error are both important. Then, the shorthand notation (P1(...) & P2(...) & P3(...)) is not satisfying anyway. Secondly, if returning an explicit error code/flag all the time is too awkward, the shorthand notation does not solve the problem of general error handling: "it relieves the symptoms but does not cure the cold". If such a program situation occurs so frequently that it becomes a problem, a proper error/exception handling concept is probably needed. However, there are ways out of the situation without additional language support. A fairly common solution is the following: every routine checks the ok flag itself. This works perfectly unless errors occur very frequently (in which case many redundant "ok" tests might be a problem):

```
ok := TRUE; P1(..., ok); P2(..., ok); P3(..., ok);
IF ok THEN (* P1 and P2 and P3 successfully executed *)
ELSE (* P1 or P2 or P3 failed *)
END

Pi (..., VAR ok: BOOLEAN);
BEGIN IF ok THEN (* procedure body *) END
END Pi
```

We have a more subtle situation in case of memoizing functions, i.e., functions that cache previously computed results in order to speed up subsequent calls with the same arguments. Such functions need to produce side-effects (otherwise they could not cache results over function calls), but no side-effects would be observable if their cache would not be visible from outside. The important property of such functions is that they yield the same results whether they memoize their results or not. Instead of using SYSTEM.ASSIGN directly in such a function F to do the caching, we can provide a library function Fm that allows for rendering a whole class of functions into memoizing ones:

```
VAR cache: CacheType;

FUNCTION Fm (x: ArgType; F: FUNCTION (x: ArgType): ResType): ResType;
(* Fm(x, F) = F(x), Fm does memoizing *)
   VAR fx: ResType;
BEGIN
   fx := Lookup(x);
   IF NotValid(fx) THEN fx := F(x); (* cache (x, fx) – side-effect! *) END;
   RETURN fx
END Fm
```

Instead of F(x) we then call Fm(x, F) (or a stub function that simply calls Fm). Only Fm needs to produce side-effects to do the caching and thus has to use SYSTEM.ASSIGN. Note that the auxiliary functions Lookup and NotValid are perfectly side-effect free functions. Ideally Fm and its cache are encapsulated in a module that protects the caching data structure from any access from outside. Thus, such a module provides the library support used to implement memoizing functions, very much like device drivers provide the support for accessing devices. In both cases low-level features of the language must be used but are sealed under a clean and safe interface.

Finally we mention a few possibilities for a couple of new optimizations enabled by side-effect free functions that otherwise were only possible after expensive analysis of the whole program:

- The evaluation order of (sub-)expressions – for instance the argument evaluation order of function or procedure calls – does not matter (except for shortcut evaluation of *and* and *or*, of course). This has direct consequences for a programming language and its compiler: the language does not have to specify an evaluation order for these cases or, if the evaluation order is unspecified anyway, the result will be always the same independent of the particular implementation or applied optimization. If a target machine features several processors, different functions of a single expression or argument list may be evaluated in parallel on different processors.

- If an S-function allocates new dynamic variables but does not return a pointer type, these dynamic variables must have been used for internal computations only. Thus, the space allocated on the heap could be released after leaving the function, or the variables could be allocated dynamically on the stack.

- If the same R-function or S-function that does not return a pointer occur several times with the same arguments within an expression or argument list, they are common subexpressions and may all be eliminated but one.

5 Conclusions

We have shown that it is possible to detect side-effects in functions that dynamically allocate new variables. Due to the dynamic nature of these variables a run-time check is required, analogous to other integrity-tests such as array index checks for instance. The additional implementation effort in a compiler is almost negligible and the check is fairly efficient.

It is often believed that the specification of a programming language should be as detailed as possible, i.e., should not leave things unspecified. While this is not a bad idea per se, it turns out that many details are often irrelevant for a particular program, yet there is usually no way to tell a compiler to ignore them. For instance, many programming languages specify an argument evaluation order which a programmer is forced to use, even if the order is unimportant in a particular situation. In turn, a compiler must obey the order specified by the programming language and often cannot choose another, maybe more efficient one (at least not without expensive program analysis). Hence, the more detailed a language specification, the less freedom a compiler has for optimizations. In other words: too detailed a language definition may lead to overspecified programs and overspecified programs offer less potential for optimizations.

Therefore, besides experimenting with new features, new languages should also refine and optimize well-known concepts, i.e., loose their specifications in order to better exploit their inherent potential. Side-effect free functions are a step in this direction. A programming language that restricts functions to side-effect free ones may leave things like the evaluation order of expressions unspecified. Consequently, programs become easier to prove and hence to understand and a compiler also may produce better code.

Acknowledgements

I like to thank Roberto Ierusalimschy, Noemi de la Rocque Rodriguez, Heinz W. Schmidt and David Stoutamire for various fruitful discussions on this topic, as well as Jeff Bilmes, Ralph Salomon and Christian Schwarz for commenting on earlier versions of this paper.

References

1. Dijkstra, E.W. and W.H.J. Feijen (1988). *A Method of Programming*, Addison-Wesley.
2. Griesemer R. (1993). *A Programming Language for Vector Computers*, Diss. ETH Nr. 10277, ETH Zürich, ftp ftp.inf.ethz.ch, get /doc/diss/th10277.ps.
3. Henderson P. (1980). *Functional Programming – Application and Implementation*, Prentice Hall, London.
4. Ierusalimschy, R. and N. de la Rocque Rodriguez (1993). "Side Effect Free Functions in Object-Oriented Languages", to be published.
5. Lampson, B.W., J.J. Horning, R.L. London, J.G. Mitchell and G.J. Popek (1977). "Report On The Programming Language Euclid", *ACM SIGPLAN Notices*, 12:2 (February).

6. Winston, P.H. and B.K.P. Horn (1984). *Lisp*, 2nd edition, Addison-Wesley.
7. Wirth, N. (1988). "The Programming Language Oberon", *Software – Practice and Experience*, 18:7 (July), 671–690.
8. Wirth, N. and J. Gutknecht (1992). *Project Oberon – The Design of an Operating System and Compiler*, Addison Wesley, New York.