Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Dissolving a half century old problem about the implementation of procedures

Gauthier van den Hove

CWI, SWAT, Science Park 123, 1098 XG Amsterdam, The Netherlands

ARTICLE INFO

Article history:
Received 21 June 2014
Received in revised form 11 July 2017
Accepted 14 July 2017
Available online xxxx

Keywords:
Static link
Block
Closure
Lexical scope
Procedure

ABSTRACT

We investigate the semantics of the procedure concept, and of one of the main techniques introduced by E. W. Dijkstra in his article *Recursive Programming* to implement it, namely the "static link," sometimes also called "access link" or "lexical link." We show that a confusion about that technique persists, even in recent textbooks. Our analysis is meant to clarify the meaning of that technique, and of the procedure concept. Our main contribution is to propose a better characterization of the "static link."

1. Introduction

One of the first concepts introduced in the development of programming languages is the procedure concept. Its first precise definition appears in the ALGOL 60 *Report*, together with the static (or lexical) scope rule. Its implementation poses a number of difficulties, and the now classical solution to solve these difficulties, for ALGOL-like languages, was proposed by Dijkstra in his article *Recursive Programming* [1]. The central elements of this solution are the execution stack, and what is now known as the "static link"; they were embodied a few months later in the first ALGOL 60 system, designed and implemented by Dijkstra and J. A. Zonneveld. A close study of the *Report*, of that article, and of that system, revealed a common misunderstanding, on one specific aspect, of that article.

Specifically, we show that the "static link" cannot be described independently of the program execution, and we propose a definition of the "static link" that is better than those found in the literature (§ 4). We also show that the procedure concept was originally dynamic and was not identified with the subroutine concept (§ 7). A subroutine can be defined as a program text, that can be called with a number of parameters, and that eventually returns to the point immediately following that from which it was called; the procedure concept includes an additional dynamic element, and will be defined later (in § 7). These results are certainly known to experts in the field, and perhaps even totally obvious for some of them. We believe, however, that they should be more largely known, and we observe that they were often presented ambiguously in the literature (§ 3 and § 6). In fact, we do not pretend to solve a real problem, but rather to dissolve a false one.

The article is organized in six parts. We start by presenting an apparently anecdotal problem (§ 2), we show that it has led to unclear or incorrect explanations, and sometimes to incorrect implementations (§ 3), and we give its solution (§ 4). We then examine how this solution can be implemented (§ 5), and we show that this problem is not at all

E-mail address: ghe@cwi.nl.

http://dx.doi.org/10.1016/j.scico.2017.07.007
0167-6423/

Please cite this article in press as: G. van den Hove, Dissolving a half century old problem about the implementation of procedures, Sci. Comput. Program. (2017), http://dx.doi.org/10.1016/j.scico.2017.07.007

anecdotal, and is actually ubiquitous in ALGOL 60 (§ 6). We conclude by discussing the meaning of the procedure concept (§ 7).

2. A problem

It is well-known that ALGOL 60 introduced the "static scope" of identifiers, and that it differs from the "dynamic scope" of identifiers used, for example, in LISP. The difference between these two scoping mechanisms can be illustrated with the following program:

```
begin                                                                    (1)
  real procedure sqrt (r); value r; real r; sqrt := r ↑ (1/2);
  integer procedure fibonacci (n); value n; integer n;
  fibonacci := ((1 + sqrt (5))/2) ↑ n/sqrt (5);
  begin
    real procedure sqrt (r); value r; real r; sqrt := exp(ln(r)/2);
    outinteger (1, fibonacci (3))
  end
end
```

In ALGOL 60 and its descendants, the identifier *sqrt* in the body of *fibonacci* refers to the procedure defined with the exponentiation operator, whereas in LISP the execution of *fibonacci* would use the procedure *sqrt* defined with the *exp* and *ln* procedures. This difference can, at first sight, be explained as follows. With static scoping, one only needs to read the program text to find the declaration that corresponds to an identifier at a given program point: one first looks at the declarations in the block in which that program point resides, then at the declarations in its parent block, *et cætera*. With dynamic scoping, one needs to take the program execution into account: the relevant declaration is the last one that was met, that is, the last one through which the execution flow passed. This is, however, not a completely correct explanation of static scoping: if a declaration appears in a procedure that is activated recursively, then it gives rise to multiple instances of the declared object, and the above statement does not specify to which of these instances the identifier refers.

Dijkstra's article gives an indication that seems to answer that question. After having introduced the concept of an execution stack, divided into frames, each frame corresponding to an activation of a subroutine, he notes that each procedure of an ALGOL 60 program text can be translated into a subroutine, and points out that in that case it is necessary to record an additional frame pointer value in the link [1, pp. 317–318]:

> When a subroutine is called in, the link contains *two* [frame] pointer values [...]. Firstly, the youngest [frame] pointer value corresponding to the block in which the *call* occurs [...], secondly, the value of the [frame] pointer corresponding to the most recent, not yet completed, activation of the first block that lexicographically encloses the *block* of the subroutine called in.

In this explanation, the word "block" is used to mean generically a procedure block or a non-procedure block; we will follow that convention. One could conclude from this reading that the object instance to which an identifier refers is always the one that was created in the "most recent activation" of its parent block, given that the "static link," that is used to locate global objects, points to that "most recent activation." This understanding is indeed correct, even with the following quite complex program, in which all identifiers always refer to their declaration in the "most recent activation" of *fibonacci*:

```
begin                                                                    (2)
  integer procedure fibonacci (n); value n; integer n;
  begin integer fm;
    integer procedure fa;
        fa := fm := if fm > 0 then fm else fibonacci (n ÷ 2);
    integer procedure fb; fb := fibonacci (n ÷ 2 − 1);
    integer procedure fc; fc := fa × fb;
    integer procedure fd; fd := fa + fb;
    fm := 0;
    fibonacci := if n < 2 then n else fa↑2 + ( if even (n) then fc × 2 else fd↑2)
  end ;
  outinteger (1, fibonacci (3))
end
```

Please cite this article in press as: G. van den Hove, Dissolving a half century old problem about the implementation of procedures, Sci. Comput. Program. (2017), http://dx.doi.org/10.1016/j.scico.2017.07.007

However, twelve years after Dijkstra's article, C. L. McGowan published an article entitled *The "most recent" error*, claiming that the description given by Dijkstra is not correct. He calls Dijkstra's frames "activation records" and the two frame pointer values the "dynamic link" and the "static link" of these activation records [2, pp. 193 and 194]:

> Activation records in the stack must be linked both for purposes of control and for purposes of accessing. [...] For accessing, the *static link* of the program block B is a pointer to *a* stack activation record allocated for the block B' in which B is immediately (textually) enclosed. [...] To achieve accessing of identifiers non-local to [a procedure P declared in the immediately enclosing block B], the [activation record] is (statically) linked to *an* activation record corresponding to the enclosing block B in the program. It is tempting to [statically] link [the activation record] to the "most recent" record corresponding to B in the stack. This strategy is usually correct [...], but not always. This error in stack implementation of block structured programming languages is a persistent and recurrent one. [...] Dijkstra in his classic paper incorrectly describes the static link of an activation record for a procedure to be a pointer "to the most recent, not yet completed, activation of the first block that lexicographically encloses the block of the subroutine called in."

The author claims that it is not correct to describe the "static link" as a pointer to the "most recent activation" of the enclosing block, and he proposes to describe it as pointing to "*an* activation record corresponding to the enclosing block in the program" (his emphasis), which is of course correct, but only by lack of saying anything specific. McGowan then gives a two page long semi-formal description of an abstract interpreter for a simplified ALGOL-like language, based on the contour model introduced earlier by J. B. Johnston [3], to characterize the values taken by the "static links" during execution of a program written in that language. Finally, he identifies and proves the correctness of two complex "compile-time decidable conditions" under which Dijkstra's description, which has seemingly the advantage of being easy to understand and to implement, is correct [2, pp. 199][1]:

> [*Definitions:*] If procedure Q contains within its body a call statement with procedure name R, then Q *can call* R. If R is a formal parameter of Q and [the program] contains a call of Q with identifier F in the R-position of the argument list, then Q *can call* F. Let *can call* $-^*$ be the transitive closure of *can call*. We say a procedure Q is *potentially recursive* if Q *can call* $-^* Q$. [...]
>
> *Conditions:* A program must have [1] no potentially recursive procedure R which contains (within its body) the declaration of a procedure P *and* [2] a call of some procedure Q having P as one of its arguments, where Q *can call* $-^* R$.

It is already clear from the above that the problem identified by McGowan is related to procedural parameters in the presence of nested procedure declarations and of recursion, but the precise meaning of his claim, and of these conditions, can be best explained with an example ALGOL 60 program that does not fulfill them:

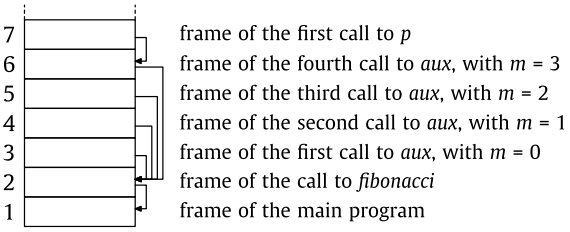
```
begin                                                                    (3)
  integer procedure fail; fail := fail;
  integer procedure fibonacci (n); integer n;
  begin
    integer procedure aux (m, n1, n2);
        integer m; integer procedure n1, n2;
    begin
      integer procedure p; p := if m < 2 then m else n1 + n2;
      aux := if m = n then p else aux (m + 1, n2, p)
    end ;
    fibonacci := aux (0, fail, fail)
  end ;
  outinteger (1, fibonacci (3))
end
```

After four calls to *aux*, procedure p has been declared four times (as, say, p_0, p_1, p_2, p_3), each time in terms of the two previous definitions of p or of *fail*; for instance, p_3 is defined in terms of p_1 and p_2. When procedure p_3 is finally called, in the fourth call to *aux*, it recursively uses the three other procedures p_0, p_1 and p_2 to perform its calculations and to yield the correct result.

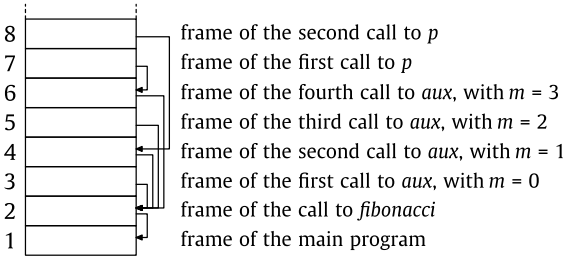
This program indeed does not fulfill the conditions mentioned earlier, because procedure *aux* is recursive and contains within its body both the declaration of a procedure p and a call of procedure *aux* having p as one of its parameters. Dynamically this implies the following. When p is called, the execution stack can be depicted as follows:

[1] These conditions have been studied in detail in [4], [5], and [6]: they are decidable but PSPACE-complete, and necessary but not sufficient for the problem to actually show up (in the example program (3), if the argument of the call to *fibonacci* is 0, the problem does not arise).

Please cite this article in press as: G. van den Hove, Dissolving a half century old problem about the implementation of procedures, Sci. Comput. Program. (2017), http://dx.doi.org/10.1016/j.scico.2017.07.007



The values of the second frame pointers saved in the links are figured on the right of the stack. As one sees, the second frame pointer in the link of this activation of p points to the sixth frame on the stack, which is evidently the "most recent" activation of the block in which p is declared in the program text, namely, procedure *aux*. To evaluate p (that is, p_3), it is necessary to consult the value of m, which is global to that procedure. In this case, the contents of the sixth frame indicates that $m = 3$, and p therefore asks for the evaluation of procedural parameter *n1*, that is, for the evaluation of the procedure p that has been passed by the second call of *aux* to the third call of *aux*, that is, for the evaluation of p_1. When p_1 is called, the execution stack can be depicted as follows:



In this case, the second frame pointer in the link of the activation of p must point to the fourth frame of the stack, which is evidently not the "most recent" activation of the block in which p is declared in the program text, which would have been again the sixth frame on the stack. Had it pointed to the sixth frame, then evaluation of p_1 would again find the value $m = 3$, and the program would again ask for evaluation of p_1, that is to say, it would enter an infinite recursion loop.

3. Its permanence

McGowan states that the error he identified is a "persistent and recurrent one," but gives a single example, namely an "early version of PL/I from the IBM Hursley Laboratory." It seems thus necessary to give more substance to his claim.

Before Dijkstra's solution became standard, a number of other implementation strategies were proposed and implemented. For example, the solution presented by E. T. Irons and W. Feurzeig resembles a typical early LISP implementation. It does not use stack frames and frame pointers, and therefore does not use a "static chain." However, as D. E. Knuth and J. N. Merner already indicated [7, p. 269], their method to handle label parameters suffers from the problem identified above: their "go to interpreter" repeatedly performs the operations to return from a block or procedure until an activation of the block in which the label is declared has been found and restored [8, p. 69].

Likewise, those who followed the principles laid down by Dijkstra sometimes considered that the second frame pointer values saved in the links were an unnecessary optimization in his solution: it was thought that their values could be recalculated during the execution of the program, with the help of the chain of the first frame pointer values saved in the links. For instance, in their classical work *ALGOL 60 Implementation*, B. Randell and L. J. Russell present the two frame pointer values as follows (our emphasis) [9, p. 65]:

> The record of previous values of [the frame pointer (that is, the first frame pointer values saved in the links)] is also used to find the actual address [...] of a variable in the stack. However, within a block the only variables that can be used are those which are currently accessible. Thus only the values of [the frame pointer] for blocks which are currently accessible will be needed to find the addresses of variables. It is therefore *preferable* to keep two separate records of values of [the frame pointer]. The first record gives the start of the stacked working storage of each block that has been activated and not yet left. It is this record that would be used to reset [the frame pointer value] each time a block is left. The second record contains only the values of [the frame pointer] which can be used, in the current block, to find the actual address of a variable.

Please cite this article in press as: G. van den Hove, Dissolving a half century old problem about the implementation of procedures, Sci. Comput. Program. (2017), http://dx.doi.org/10.1016/j.scico.2017.07.007

This is stated even more explicitly a few pages later [9, p. 74]:

> [The static link of a block with a block number of n] is the value of [the frame pointer] for the last activation of a block with [a block number of $n-1$]. This could be found by working back down the dynamic chain [provided that, as it is the case in the implementation strategy they present, block numbers are stored in the links] [...].

This faulty scheme was indeed used in some implementations of ALGOL 60, for instance by P. A. Samet [10]. His short article concentrates on the handling of label parameters, which are essentially parameterless procedural parameters. His solution, namely to follow the "dynamic chain" until the "most recent activation" of the block in which the label is declared has been found, clearly suffers from the problem identified above.

These three additional examples are, however, half a century old, and could probably be dismissed as being, at best, part of the minutiæ of programming languages history. It is also true that this error is not anymore present in any major programming language implementation, but for a long time, imperative programming languages have often ruled out either nested procedure declarations or procedural parameters. For instance, C, C++ (before its 2011 revision), and FORTRAN 77 allow procedural parameters but do not allow nested procedure declarations, and Ada 83 allows nested procedure declarations but not procedural parameters. In Wirth Pascal (as opposed to the later Standard Pascal), nested procedure declarations and procedural parameters are both allowed, with the restriction that a procedural parameter may not itself have a procedural parameter; however, most of the following Pascal implementations (for instance, P1 to P4, Pascal-S and UCSD Pascal) did not support procedural parameters at all. Standard Pascal and Ada 95 both allow nested procedure declarations and procedural parameters, without restrictions, but again, most subsequent Pascal implementations (for instance, Borland Pascal and Free Pascal) did not support them.

However, we claim that this error still affects the contemporary literature. On this specific point, most current textbooks on programming languages fall in one of the following three categories: (a) those who remain unspecific on the value of the "static link," (b) those who use Dijkstra's description, and (c) those who use McGowan's description.

(a) Some authors present the static link by explaining its purpose, but are not explicit about its value. For example, W. M. Waite and L. R. Carter (1993) explain that "each activation record contains the address of the memory used for local storage by the activation of the containing routine" [11, p. 205], or that the second frame pointer value saved in the link is "the address of the activation record for the enclosing procedure" [11, p. 250]. Likewise, N. Wirth (1996) writes that "the static link of a procedure p points to the activation record of the procedure which contains p" [12, p. 92]. These descriptions are not incorrect, but they are clearly insufficient to understand what happens in cases like the example program (3) above.

(b) Many textbooks still use Dijkstra's original description of the second frame pointer value, which stated that it points to the "most recent [...] activation of the first block that lexicographically encloses the block of the subroutine called in." For example, D. Watson (1989) writes: "The static chain pointer in any stack frame points back down the stack to the frame corresponding to the latest invocation of the next textually enclosing block or procedure. In other words, by following the static chain down the stack, the stack frames of all blocks or procedures containing variables in scope are identified, unless they are hidden by more recent declarations of the same name" [13, p. 224]. Although he discusses parameters passed by name [13, p. 24 and p. 233], he does not explain how procedural parameters should be implemented. T. W. Parsons (1992) writes that each static pointer "is aimed at the most recent activation" or "topmost activation" of the enclosing block [14, p. 295], and leaves the implementation of procedural parameters as an exercise, without further indications [14, p. 306]. S. S. Muchnick (1997) explains that "to set the static link in a stack frame, we need a mechanism for finding the nearest invocation of the procedure that the current procedure is statically nested in" [15, p. 114]. Admittedly, he later presents a particular (correct) technique to implement procedural parameters, but without explaining if it is a "mechanism for finding the nearest invocation" or not, and without giving an example [15, p. 126]. A. W. Appel (1999, corrected first edition) writes that the static link of a procedure with static nesting depth $i+1$ is "a pointer to the frame of the most recently entered procedure whose static nesting depth is i" and that "whenever a function f is called, it is passed a pointer to the stack frame of the "current" (most recently entered) activation of the function g that *immediately encloses* f in the text of the program" [16, p. 133 (and p. 146)]. He later presents the same particular technique to implement procedural parameters, without linking it with his earlier presentation [16, p. 312]. The exact same words are used by Appel and M. Ginsburg (1999, corrected first edition) [17, p. 134 (and p. 148), p. 318], and by Appel and J. Palsberg (2002, second edition) [18, p. 125 (and p. 134), p. 301]. A. V. Aho et al. (2006, second edition) explain that "if procedure p is nested immediately within procedure q in the source code, then the access link in any activation of p points to the most recent activation of q" and that, when a certain object defined in q is to be accessed, the activation of q in which it is located "will always be the most recent (highest) activation [...] for q that currently appears on the stack" [19, pp. 445–446 (and p. 449)]. They also present, a few pages later, the same technique to implement procedural parameters, but they do not explain if this technique is an application of the general rule stated earlier or not. The example program they use to demonstrate the technique is not very helpful to answer that question: four procedures are declared, and each one has only one activation [19, pp. 448–449]. M. L. Scott (2009, third edition) explains that the "static link in each frame [...] points to [...] the frame of the most recent invocation of the lexically surrounding subroutine" [20, pp. 126–127 (and p. 389)]. He also presents the same technique to implement procedural parameters, this time with an example program that breaks the rule stated earlier, but without saying so explicitly [20, pp. 154–155]. R. W. Sebesta (2012, tenth edition) writes that "the most recent activation record instance of the parent scope must be found at the time of the call. This can be done by looking at activation record instances on the

Please cite this article in press as: G. van den Hove, Dissolving a half century old problem about the implementation of procedures, Sci. Comput. Program. (2017), http://dx.doi.org/10.1016/j.scico.2017.07.007

dynamic chain until the first one of the parent scope is found" [21, p. 459]. He mentions that this method works "except when parameters that are subprograms are involved," but he does not explain how the "static link" should be calculated in the latter case, although he elsewhere presents the general principle behind the technique described by the previous authors [21, p. 405 and p. 431].

These explanations are partly correct, especially if one looks benevolently at their context, but they are misleading: (1) the "most recent activation" statements are not correct in general, (2) it is not easy to see that they are not correct in general, (3) they are inadequate to understand the purpose of "static links," even if procedural parameters are excluded, (4) they are also inadequate to understand how they can be implemented, and finally (5) the words "most recent activation," which can be traced back to Dijkstra's article, had originally a different meaning than what they now evoke, as will be shown in § 4.

(c) Finally, a few authors follow McGowan's suggestion and describe the "static link" with an expression derived from his proposal [22]. For instance, R. Wilhelm and D. Maurer (1995) explain, to introduce the concept, that "the immediate [static] predecessor of a [...] procedure incarnation in [the call] tree is the last [or, a few lines earlier, "most recent"] incarnation of the immediately surrounding procedure" [23, p. 36], but their presentation of the general rule is much clearer (our emphasis): "In every frame created in the stack for the incarnation of a procedure *p*, the static link points to the stack frame of the *correct* incarnation of the program unit immediately surrounding *p*. In programs without [...] procedures as parameters, this is the youngest living incarnation of the program unit immediately surrounding *p*" [23, p. 41 (and p. 53)]. Of course, this general description of the "static link" as pointing to the "correct incarnation of the immediately surrounding program unit" is as vague as McGowan's proposed description and is true only by lack of being precise.

4. Its solution

We now show that, in spite of the evidence of the example program (3), McGowan's claim is the result of a misinterpretation of the meaning of the procedure concept, and specifically of the semantics of procedure calls, as they are defined in the ALGOL 60 *Report*. Let us reread Dijkstra's description of the two frame pointer values saved in the link when a subroutine is entered, paying attention to the words "not yet completed" instead of the words "most recent" [1, pp. 317–318]:

> Firstly, the youngest [frame] pointer value corresponding to the block in which the *call* occurs [...], secondly, the value of the [frame] pointer corresponding to the most recent, not yet completed, activation of the first block that lexicographically encloses the *block* of the subroutine called in.

These three words may seem, at first, to be useless: it is obvious that the second frame pointer value cannot point to an activation of a block that is already completed. However, these words could also mean, reciprocally, that the other activations of the lexicographically (or lexically) enclosing block are already completed. While this interpretation is grammatically possible, it seems absurd: in the two previous pictures of the execution stack, there are clearly four activations of procedure *aux*. Yet it corresponds to the meaning of the procedure concept as it is defined in the *Report*: the frames numbered 3 to 6 in these pictures are actually activations of four *different* blocks.

To understand this, it is necessary to take a closer look at the semantics of procedure calls. When a procedure is called through a (non-formal or formal) identifier, the *Report* specifies that "the procedure body [...] is inserted in place of the procedure statement and executed" [24, § 4.7.3.3]. (Before this insertion happens, the identifiers that appear in the procedure heading must be replaced by the text of the actual parameters throughout the procedure body, and the identifiers that appear in the procedure body must, if necessary, be renamed to avoid unintended identifications of originally distinct identifiers [24, § 4.7.3.2].) The consequence of this rule, for our example program (3), is that the procedure body of *aux* is dynamically inserted four times in place of the occurrences of the identifier *aux*, as if the procedure statements gave rise to a macro expansion. This dynamically creates four different blocks, say, aux_0, aux_1, aux_2 and aux_3. Block aux_1 is then nested dynamically in block aux_0, and lexically, like block aux_0, in block *fibonacci*. The same holds for blocks aux_2 and aux_3: they are nested dynamically in blocks aux_1 and aux_2, and lexically in block *fibonacci*.

The consequence of this is that four different procedures p_0 to p_3 are declared, dynamically, in four different blocks: p_0 is declared in aux_0, p_1 in aux_1, etc. The first block that lexically encloses block p_3 is, therefore, aux_3, and when p_3 is called, the second frame pointer to be saved in the link should point to the "not yet completed," or current, activation of aux_3. Likewise, the first block that lexically encloses block p_1 is aux_1, and when p_1 is called, after having moved around through procedural parameters, the second frame pointer to be saved in the link should point to the current activation of aux_1.

It should now be clear that the words "most recent activation" do not mean "*n*th activation among the *n* current activations," because each block is, at any moment, either activated or not activated, but rather "*n*th activation among the *n* successive activations." With the procedure of example (3), in the loop '**for** $i := 3, 3$ **do** *fibonacci* (i)' for instance, the same four blocks aux_0 to aux_3 would be activated twice, successively. The two words "most recent" could therefore be replaced by the word "current," like the word "youngest" in the description of the first frame pointer value saved in the link; they could also be omitted altogether.

The consequence of the above paragraphs is that:

Please cite this article in press as: G. van den Hove, Dissolving a half century old problem about the implementation of procedures, Sci. Comput. Program. (2017), http://dx.doi.org/10.1016/j.scico.2017.07.007

Definition. If a block-structured programming language is implemented with Dijkstra's scheme, then two frame pointer values are recorded in the links:

1. the frame pointer value corresponding to the activation of the block in which the block is dynamically entered, and
2. the frame pointer value corresponding to the activation of the block in which the block entered was dynamically declared.

This definition, in which block structure implies lexical scope, makes explicit the fact that a procedure block is declared anew each time that its declaration is met during the execution of the program.

This definition is correct (a) for programming languages without procedures, (b) for programming languages with procedures but without nested procedure declarations, with or without procedural parameters, (c) for programming languages with procedures and nested procedure declarations, with or without procedural parameters, and even, as we will see later, (d) for programming languages with implicit, or anonymous, procedure declarations. (It is also correct (e) for programming languages in which procedures can be "stored" or returned by procedures. This case falls, however, outside of the scope of this article, because Dijkstra's implementation scheme cannot be used for those languages. The procedure activations cannot be stored in a stack, and although it is possible to use "static links" it is often better not to use them, for efficiency reasons.)

A non-procedure block being entered when it is declared, the two frame pointer values recorded in the link are the same. When a non-formal procedure block is entered, the activation of the block in which that procedure was declared is either the current block or one of the blocks in which it is lexically nested, and the frame pointer value sought can therefore easily be found by following the chain of the second frame pointer values saved in the links; in other words, the block in which it was declared is still present in the current environment. When a formal procedure block is entered however, the activation of the block in which that procedure was declared may not be present in the current environment anymore, either because it is entered in a lexically enclosing block of the block in which it was declared or because the block in which it was declared may have been shadowed by another insertion of the procedure body in which it is declared in the program text. In this last case, it is only under apparently exceptional conditions, namely, those indicated by McGowan, that the activation of the block in which that procedure was declared is not present in the current environment anymore.

5. Possible implementations

It is, of course, possible to object that our interpretation of Dijkstra's words has the merit of saving his article, but that it is far from obvious, and not supported by other substantial evidence. This is, however, untrue. The last paragraph of the *Recursive Programming* article presents a technique which among others solves this problem in an elegant and efficient way [1, p. 318]:

> One can assign a so-called *block number* to each block, indicating the number of blocks which enclose it lexicographically: the main program therefore has block number = 0. [...] The introduction of block numbers also makes it possible that the arithmetic unit has immediate access to all the [frame] pointer values it may need. They can be stored in order of increasing block number in a so-called "display". By tracing the second chain of [frame] pointers one can, when necessary (amongst others at the return, in the call of a formal procedure, and at the beginning of the evaluation of a non-trivial formal parameter), bring the display up to date.

The words of special importance here are "when necessary [...] in the call of a formal procedure," which indicate that the execution of a formal procedure must be preceded by a specific operation that "brings the display up to date." However, except for this short mention, the handling of formal procedures is not explained further in the article, and it is necessary to look elsewhere for the details. If we assume, for a moment, that procedures can have only two types of parameters, values and procedure identifiers, then the passing of an actual parameter to a procedure called in a certain block, say b_a, can be presented as follows [25, p. 17]:

1. either it is a value, in which case it is recorded in the local variables of the newly allocated stack frame;
2. or it is a non-formal procedure identifier, in which case a triplet (address of that procedure, block number of the current block (say, bn_a), current value of the frame pointer (say, fp_a)) is recorded in the local variables of the newly allocated stack frame (such a triplet, containing both a procedure and one of the many possible representations of the dynamic environment in which it is declared, is now known, following the suggestion of P. J. Landin [26, pp. 316–317 and p. 320], as a "closure");
3. or it is an already formal procedure identifier, in which case it suffices to make a local copy of a previously recorded triplet, from either the current stack frame at the moment of the call, or from one of the stack frames pointed to by the chain of second frame pointers in the links.

When a block, say b_e, with a certain block number bn_e, is entered in a block b_o with a certain block number bn_o, a new frame is allocated on the stack, its link is filled with the current block number = bn_o, frame pointer = fp_o, and return address, the frame pointer is set to the stack pointer value, and only two cases are then possible [25, pp. 13–17]:

Please cite this article in press as: G. van den Hove, Dissolving a half century old problem about the implementation of procedures, Sci. Comput. Program. (2017), http://dx.doi.org/10.1016/j.scico.2017.07.007

1. either b_e is a non-procedure block, or a procedure block called through a non-formal identifier, in which case two operations have to be performed:
   (1) display[bn_e] ← frame pointer,
   (2) second frame pointer value in the link of the new frame ← display[$bn_e - 1$],
2. or b_e is a procedure block called through a formal identifier, in which case an additional operation has to be performed before the two of the previous case:
   (1) with the help of block number bn_a and of frame pointer fp_a saved in the triplet representing the actual parameter, restore the contents of the display that were current at the moment the procedure was passed as parameter (starting with bn_a and fp_a, which is saved into display[bn_a], the other frame pointers display[$bn_a - 1$], …, display[0] can be found in the links in the execution stack),
   (2) display[bn_e] ← frame pointer,
   (3) second frame pointer value in the link of the new frame ← display[$bn_e - 1$].

As we have seen, the first operation in this second case is explicitly mentioned in the *Recursive Programming* article, with the rule: "by tracing the second chain of [frame] pointers one can, when necessary […] in the call of a formal procedure […] bring the display up to date." And the last operation is a particular case of the rule "the arithmetic unit has immediate access to all the [frame] pointer values it may need" in the display.

Finally, as indicated by the rule "by tracing the second chain of [frame] pointers one can, when necessary […] at the return […] bring the display up to date," when a block b_e, previously entered in a block b_o, is left, the stack pointer is set to the frame pointer value, the contents of the link (block number = bn_o, frame pointer = fp_o, return address) are restored, and a single operation needs to be performed:

(1) with the help of block number bn_o and of frame pointer fp_o just restored from the link, restore the contents of the display that were current at the moment the block b_e was entered (starting with bn_o and fp_o, which is saved into display[bn_o], the other frame pointers display[$bn_o - 1$], …, display[0] can be found in the links in the execution stack).

It is clear that this technique is a correct and general solution to the three possible cases mentioned at the end of § 4. A non-procedure block b_e is entered when it is declared, in b_o, and in that case display[$bn_e - 1$] contains the value of the frame pointer that was current in b_o. When a non-formal procedure block b_e is entered, the value of the frame pointer pointing to the current activation of the block in which that procedure was declared is also contained in display[$bn_e - 1$]. Finally, when a formal procedure block b_e is entered, display[$bn_e - 1$] contains the value of the frame pointer pointing to the current activation of the block in which that procedure was declared at the moment it was passed as parameter, which, as implied by the previous case, corresponds to the activation of the block in which block b_e was declared.

This technique can easily be extended to support implicit, or anonymous, procedure declarations, referred to as the "non-trivial formal parameters" in the previous quotation. In example program (3), the parameter $m + 1$ passed to procedure *aux* is a very simple example of one such anonymous procedure: when parameter m is evaluated in procedure p, its value is actually not immediately found in the corresponding activation of *aux* but is recursively recalculated, starting with the value 0 passed by *fibonacci* to the first call of *aux*. In other words, execution proceeds as if a procedure *m plus 1* had been defined inside of procedure *aux*, that is, as if procedure *aux* had been defined as follows:

```
integer procedure aux (m, n1, n2);                                    (4)
    integer procedure m, n1, n2;
begin
  integer procedure p; p := if m < 2 then m else n1 + n2;
  integer procedure m plus 1; m plus 1 := m + 1;
  aux := if m = n then p else aux (m plus 1, n2, p)
end ;
```

This implies that the block number of procedure *m plus 1* is one higher than the block number of procedure *aux*, and, more generally, that the parentheses in which the actual parameters of a procedure are enclosed start and end a new block level. In other words, when the implicit procedure $m + 1$ is called, the value of the second frame pointer to be saved in the link is also to be found in display[$bn_e - 1$], but bn_e is one higher than the block number of the block in which the procedure statement that contains the declaration of that implicit procedure resides. This should, of course, be understood recursively, because nothing forbids, in ALGOL 60, that an implicit procedure contains a call to some procedure with a number of implicit procedures as parameters.

It is of course possible, should one believe that the display is an inefficient mechanism, to avoid using it. The immediate consequence of the rules given above is that the simplest way to pass an actual (explicit or implicit) procedure p as parameter is to pass a pair (address of procedure p, current frame pointer value corresponding to $bn_p - 1$), where bn_p is the block number of the body of the procedure p, instead of a triplet (address of procedure p, current block number, current value of the frame pointer). When the formal procedure is later called, the second value of that pair can then be used immediately as the value of the second frame pointer to be recorded in the link. This solution, known as "fat pointers," is presented in

Please cite this article in press as: G. van den Hove, Dissolving a half century old problem about the implementation of procedures, Sci. Comput. Program. (2017), http://dx.doi.org/10.1016/j.scico.2017.07.007

the literature for instance as "when procedures are used as parameters, the caller needs to pass, along with the name of the procedure-parameter, the proper access link for that parameter" [19, pp. 448], or as "the standard approach is for the caller to compute the callee's static link and to pass it as an extra, hidden parameter" [20, p. 387 (and p. 154)]. The P5 implementation of Pascal for example, completed in 2009, more than thirty years after the P4 implementation, essentially uses that technique, implemented with three new specific instructions.

More complicated schemes, based on the rules given above, can be used to handle procedural parameters. One of them, proposed by T. M. Breuel [27], has been implemented in the second version of the GCC compiler suite. It requires dynamic generation of a short segment of code in the stack, which contains the pair (address of procedure p, current frame pointer value corresponding to $bn_p - 1$) mentioned in the previous paragraph and records the value contained in the second element of that pair in the link of the newly allocated frame, before jumping to the address indicated in the first element of that pair. The procedural parameter can then be moved around through a pointer to that segment of code. The advantage of this more complex technique, nicknamed "trampoline," is that it is compatible with C pointers, which makes interfacing between multiple languages easier; its main disadvantage is that it requires that the elements in the stack be executable. The GNU Pascal compiler, which uses the GCC compiler back-end, correctly handles procedural parameters using that technique.

6. The extent of the problem

It is also possible to object that our interpretation of Dijkstra's words is not correct by arguing that the above scheme solved the problem identified by McGowan only by chance. Our example program (3) is indeed rather complex, and the example given by McGowan, in PL/I, is even more intricate than ours; it is derived from an example program used by Johnston [3, p. 61] and can be translated as follows into ALGOL 60 [2, p. 196]:

```
begin                                                          (5)
  integer b; procedure q; begin end ;
  procedure a(f); procedure f;
  begin
    integer x; procedure p; outinteger (1, x);
    b := if b = 0 then 1 else 0; x := 5 × b;
    if b = 1 then a(p) else f
  end ;
  b := 0; a(q)
end
```

This program should output the integer value "5"; with a faulty implementation it would print "0". This example program, or slight variations or complications thereof, has become the canonical example to demonstrate the problem.[2] The other classical example to check whether an implementation is correct with respect to this problem is the "Man or boy" test program designed by Knuth long before McGowan's article [32], which was executed correctly on the ALGOL 60 system designed by Dijkstra and Zonneveld [33]:

```
begin                                                          (6)
  real procedure A (k, x1, x2, x3, x4, x5); value k; integer k;
  begin
    real procedure B;
    begin k := k − 1; B := A := A (k, B, x1, x2, x3, x4) end ;
    if k ≦ 0 then A := x4 + x5 else B
  end ;
  outreal (1, A (10, 1, −1, −1, 1, 0))
end
```

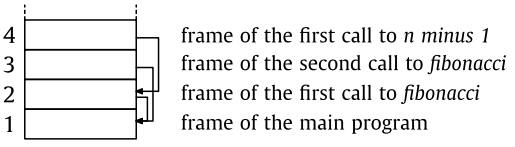
This program should print the value "−67"; like our example program (3), it would enter an infinite recursion loop with a faulty implementation. Looking at these examples, one might be tempted to conclude that the error arises only in exceptional circumstances, and most probably not in an ALGOL 60 program that was not specifically designed to test this corner case. If this were true, it would still be imaginable that Dijkstra missed the problem. A much simpler and clearer example could, however, be used:

[2] Cf. for instance [20, p. 155], [28, p. 116], or [29, p. 37] (itself also based on [30, pp. 108–109]). They are collected and discussed in [31].

Please cite this article in press as: G. van den Hove, Dissolving a half century old problem about the implementation of procedures, Sci. Comput. Program. (2017), http://dx.doi.org/10.1016/j.scico.2017.07.007

```
begin                                                                    (7)
  integer procedure fibonacci (n); integer n;
  begin
    integer procedure n minus 1; n minus 1 := n − 1;
    integer procedure n minus 2; n minus 2 := n − 2;
    fibonacci := if n < 2 then n else
        fibonacci (n minus 1) + fibonacci (n minus 2)
  end ;
  outinteger (1, fibonacci (3))
end
```

This example is especially interesting because there is not a single call, in the eleven calls to *n minus 1* and in the six calls to *n minus 2*, in which the "static link" points to the current topmost activation of the procedure *fibonacci*. For instance, during the first call to *n minus 1*, the execution stack can be depicted as follows:



Had the "static link" pointed to the topmost activation of procedure *fibonacci*, then evaluating *n minus 1* would again ask for evaluation of *n minus 1*, *ad infinitum*. What makes this example even more interesting is that it is, by virtue of the explanations given above (§ 5), strictly equivalent to the following one:

```
begin                                                                    (8)
  integer procedure fibonacci (n); integer n;
  fibonacci := if n < 2 then n
      else fibonacci (n − 1) + fibonacci (n − 2);
  outinteger (1, fibonacci (3))
end
```

Passing a procedure as parameter to another procedure is thus a special case of a more general case, namely passing a complex expression by name. This equivalence between a parameter called by name and a procedure is noted by McGowan: "call-by-name can be realized by passing an implicit parameterless procedure whose body is the intended by-name argument" [2, p. 194], but apparently he did not realize that because of this it is impossible that implementers of ALGOL 60 were not already aware of the problem he identified.

Let us finally suppose that the parameter *n* in the example program (8) is a **value** parameter. Given that parameters are by default passed by name in ALGOL 60, the most straightforward method to handle parameters passed by value (and the one used in the ALGOL 60 system implemented by the team lead by Dijkstra) is to pass all parameters by name, and to assign all parameters passed by value to implicit local variables before executing the procedure body, as if the procedure had been declared as follows:

```
begin                                                                    (9)
  integer procedure fibonacci (n); integer n;
  begin
    integer nval; nval := n;
    fibonacci := if nval < 2 then nval
      else fibonacci (nval − 1) + fibonacci (nval − 2)
  end ;
  outinteger (1, fibonacci (3))
end
```

In this example program, the implicit procedures '*nval − 1*' and '*nval − 2*' are both evaluated twice, and again in these four calls the "static link" never points to the current topmost activation of the procedure *fibonacci*.

It should now be clear that the description given by Dijkstra of the "static link" and our interpretation of his words given in § 4, are, beyond any reasonable doubt, correct.

7. The procedure concept

In spite of the fact that an ALGOL 60 procedure text gives rise, during the program execution, to multiple procedure declarations, it can, as Dijkstra showed in his *Recursive Programming* article, be translated into a single subroutine and

Please cite this article in press as: G. van den Hove, Dissolving a half century old problem about the implementation of procedures, Sci. Comput. Program. (2017), http://dx.doi.org/10.1016/j.scico.2017.07.007

appear only once in memory. This is an optimization permitted by the presence of "static links," but it does not imply that the subroutine is equivalent to the procedures declared by that procedure text. In the example program (3), the declaration of procedure *p* inside procedure *aux* can be translated into a single subroutine, yet during execution of the program four different procedures are defined: p_0 and p_1 can both be described as '**if** $m = 0 \vee m = 1$ **then** m **else** *fail*', p_2 as '**if** $m = 0$ **then** 0 **else** 1', and p_3 as '**if** $m = 0 \vee m = 1$ **then** m **else** 2'. What makes these four procedures different is their dynamic environment, in which *m*, *n1* and *n2* are defined differently.

The procedure or block concept in ALGOL 60 can therefore be defined as "a program text and its dynamic environment." This is clearly a dynamic concept, different from the subroutine concept, which does not include the notion of dynamic environment; it is identical to the closure concept. When a block is declared, a dynamic environment is associated with a program text. The result of this association can then be passed around through parameters, containing the program text and the environment. When a block is entered, its program text and environment are associated with a portion of memory, for instance with a stack frame, and execution of the block starts. When the execution of the block is completed, the portion of memory is dissociated from the program text and environment. Finally, when the scope of the block declaration is left, the environment is dissociated from the program text. The program text remains in memory, at the disposal of other declarations derived from the same program text. Both non-procedure and procedure blocks can go through these five states, except the second one for non-procedure blocks, which are immediately entered when they are declared.

This dynamic meaning of the procedure concept is, by the way, in perfect accordance with the *Report* (our emphasis) [24, § 5]:

> Dynamically [the presence of declarations in the head of a block] implies the following: *at the time of an entry into a block* [...] *all identifiers* declared for the block assume the significance implied by the nature of the declarations given. If these identifiers had already been defined by other declarations outside they are *for the time being* given a new significance. [...] *At the time of an exit from a block* [...] *all identifiers* which are declared for the block lose their significance again.

This rule means, in particular, that when a procedure is dynamically declared, the procedure body depends in general on the identifiers that were dynamically declared when its lexically enclosing blocks were entered. Put another way, with lexical scoping identifier bindings depend on the program execution, as with dynamic scoping, but at the moment a procedure is declared, the identifiers it contains are bound to the current environment.

Describing the "static links" independently of the program execution, as if they were only dependent on the program text, is therefore incorrect. The purpose of the static links is to implement the dynamic implications of lexical scoping, and to allow blocks to be translated into a single subroutine. In such a subroutine, references to variables or procedures must in general depend, dynamically, on their values. It is thus wrong, even to introduce the concept, to explain that "the [dynamic] link depends on the dynamic behavior of program whereas the [static] link depends on only the static form of the program text" [34, p. 180].

One possible alternative would have been to generate a new piece of code each time a block is declared; to generate such a piece of code it would be necessary to consult the current environment, but references to variables could then be made static. This implementation scheme, which relies on the possibility for a program to modify itself during its execution, might seem improbable nowadays, but it is the basis of the implementation of ALGOL 60 by Irons and Feurzeig: whenever a parameter is passed to a recursive procedure activation, a segment of code is dynamically created, in which the references to variables are static, and it is passed to the procedure as a pointer [8, p. 68]. Breuel's solution, presented above in § 5, can be seen as an intermediate between the two approaches: procedural parameters are passed around through simple pointers to a dynamically created short segment of code, which executes a constant code after setting the "static link."

8. Conclusion

Although it was correct (albeit, in retrospect, ambiguous and misleading) for ALGOL 60, Dijkstra's description of the "static link" should be abandoned. It was correct because the semantics of procedure calls are, in ALGOL 60, defined by the copy rule, which implies that each procedure call dynamically creates a different block. In languages in which this rule is not used, this description becomes incorrect. At best, it describes one of its accidental characteristics, but it is unhelpful, both to understand its purpose and to understand how it can be implemented. A definition akin to our proposal should be favored (§ 4). Finally, the procedure concept is, as it was originally conceived, different from the subroutine concept, with which it should not be identified; it was identical to what is now known as the closure concept (§ 7).

Funding

This research was supported by the NWO (Netherlands Organisation for Scientific Research) Free Competition Grant number 612.001.003.

Acknowledgements

The author would like to thank K. R. Apt, W. Cook, M. H. van Emden, F. Heckenbach, E. Hehner, P. Klint, D. E. Knuth, F. E. J. Kruseman Aretz, H. Langmaack, B. Le Charlier, H. Leroy, P.-A. de Marneffe, P. McJones, G. Morrisett, R. D. Tennent

and R. Wilhelm for their comments, suggestions and support at various stages of the preparation of this article, and the anonymous reviewers for their insightful remarks and recommendations.

References

[1] E.W. Dijkstra, Recursive programming, Numer. Math. 2 (1960) 312–318.
[2] C.L. McGowan, The "most recent" error: its causes and correction, ACM SIGPLAN Not. 7 (1) (1972) 191–202, also published in ACM SIGACT News 14, 191–202.
[3] J.B. Johnston, The contour model of block structured processes, ACM SIGPLAN Not. 6 (2) (1971) 55–82.
[4] P. Kandzia, On the most recent property of ALGOL-like programs, LNCS 14 (1974) 97–111.
[5] K. Winklmann, On the complexity of some problems concerning the use of procedures, Acta Inform. 18 (1982) 299–318, 411–430.
[6] D. Armbruster, A polynomial determination of the most-recent property in PASCAL-like programs, Theor. Comput. Sci. 56 (1988) 3–15.
[7] D.E. Knuth, J.N. Merner, ALGOL 60 confidential, Commun. ACM 4 (6) (1961) 268–272.
[8] E.T. Irons, W. Feurzeig, Comments on the implementation of recursive procedures and blocks in ALGOL 60, Commun. ACM 4 (1) (1961) 65–69.
[9] B. Randell, L.J. Russell, ALGOL 60 Implementation, Academic Press, 1964.
[10] P.A. Samet, The efficient administration of blocks in ALGOL, Comput. J. 8 (1) (1965) 21–23.
[11] W.M. Waite, L.R. Carter, An Introduction to Compiler Construction, Harper Collins, 1993.
[12] N. Wirth, Compiler Construction, Addison-Wesley, 1996.
[13] D. Watson, High-Level Languages and Their Compilers, Addison-Wesley, 1989.
[14] T.W. Parsons, Introduction to Compiler Construction, W. H. Freeman, 1992.
[15] S.S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufmann, 1997.
[16] A.W. Appel, Modern Compiler Implementation in ML, Cambridge University Press, 1999.
[17] A.W. Appel, M. Ginsburg, Modern Compiler Implementation in C, Cambridge University Press, 1999.
[18] A.W. Appel, J. Palsberg, Modern Compiler Implementation in Java, Cambridge University Press, 2002.
[19] A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 2006.
[20] M.L. Scott, Programming Language Pragmatics, Elsevier, 2009.
[21] R.W. Sebesta, Concepts of Programming Languages, Addison-Wesley, 2012.
[22] R. Wilhelm, Personal communications (2012).
[23] R. Wilhelm, D. Maurer, Compiler Design, Addison-Wesley, 1995.
[24] J.W. Backus, et al., Report on the algorithmic language ALGOL 60, Commun. ACM 3 (5) (1960) 299–314.
[25] E.W. Dijkstra, An ALGOL 60 translator for the X1, ALGOL Bull. Suppl. 10 (1961) 1–20.
[26] P.J. Landin, The mechanical evaluation of expressions, Comput. J. 6 (4) (1964) 308–320.
[27] T.M. Breuel, Lexical closures for C++, in: USENIX C++ Conference Proceedings, 1988, pp. 293–304.
[28] R.D. Tennent, Principles of Programming Languages, Prentice-Hall, 1981.
[29] W.M. Waite, G. Goos, Compiler Construction, Springer-Verlag, 1984.
[30] A.A. Grau, U. Hill, H. Langmaack, Translation of ALGOL 60, Springer-Verlag, 1967.
[31] H. Langmaack, Dijkstras fruchtbarer, folgenreicher irrtum, Inform.-Spektrum 33 (3) (2010) 302–308, Inform.-Spektrum 33 (4) (2010) 384–392, Inform.-Spektrum 33 (6) (2010) 634–646.
[32] D.E. Knuth, Man or boy?, ALGOL Bull. 17.2.4 (1964) 7.
[33] J.A. Zonneveld, Man compiler needs big brother machine, ALGOL Bull. 18.2.5 (1964) 9.
[34] J.C. Mitchell, Concepts in Programming Languages, Cambridge University Press, 2003.

Please cite this article in press as: G. van den Hove, Dissolving a half century old problem about the implementation of procedures, Sci. Comput. Program. (2017), http://dx.doi.org/10.1016/j.scico.2017.07.007