Science of Computer Programming 150 (2017) 75–86

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 18 August 2017

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/

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
```