Safely Extending Procedure Types to Allow Nested Procedures as Values (Corrected Version)[1]

Christian Heinlein

Dept. of Computer Structures, University of Ulm, Germany
`heinlein@informatik.uni-ulm.de`

Abstract

The concept of nested procedure values, i. e., the possibility of using nested procedures as values of procedure types, is a useful and powerful concept. Nevertheless, it is not allowed in languages such as Modula-2 and Oberon(-2), because it creates a serious security hole when used inappropriately. To prevent such misuse while at the same time retaining the benefits of the concept, alternative language rules as well as a small language extension for Oberon-2 are suggested, which allow nested procedures to be safely used as values of procedure types and especially to pass them as parameters to other procedures.

1. Introduction

Nested procedures, i. e., procedures declared local to another procedure, are a useful concept for structuring and decomposing large procedures into smaller and more comprehensible pieces in a natural way, without needing to introduce artificial global procedures to achieve that aim. Furthermore, the fact that nested procedures can directly access the local variables and parameters of their enclosing procedures helps to keep their parameter lists short, without needing to introduce artificial global variables for that purpose.

Procedure types, i. e., types possessing procedures as values, are another useful concept for program development that allows algorithms (e. g., for sorting) to be decoupled from their basic operations (e. g., comparing objects) and by that means increasing their generality and applicability.

To give a simple example for both of these concepts, Fig. 1 shows a global Oberon-2 procedure `QuickSort` containing nested procedures `Swap` and `SortRange` which directly access the parameter `a` of their enclosing procedure. Furthermore, to keep the sorting algorithm independent of any particular ordering criterion, a procedure `cmp` comparing two objects of type `T` (and returning an integer value smaller than resp. equal to resp. greater than zero if the first object is smaller than resp. equal to resp. greater than the second) is passed as a parameter of the procedure type `CmpProc`.

The *combination* of nested procedures and procedure types, i. e., the possibility of using not only global, but also nested procedures as actual parameters of other procedures, would be an even more useful and powerful concept, as the examples given in Sec. 2 will demonstrate. Unfortunately, however, languages such as Modula-2 [13] and Oberon(-2) [14, 8] do *not* allow this combination, i. e., they require procedure values (i. e., values of procedure types) to be global procedures. After explaining in Sec. 3 the reasons for this apparently strange restriction, in particular the problem of *dangling procedure values*, Sec. 4 suggests alternative language rules which *do* allow nested procedure values (i. e., nested procedures as procedure values) without running into this problem. Since there remains at least

[1] This is a corrected and extended version of [4] and a replacement for [5].
The correction concerns definitions L1 and L3 in Sec. 4, which have been generalized to cover `VAR` parameters, and rule R3 dealing explicitly with `VAR` parameters, which has been removed since it contained a loophole for the creation of dangling procedure values.

```
TYPE T = ...;   (* An arbitrary type. *)

TYPE CmpProc = PROCEDURE (x, y: T) : INTEGER;

PROCEDURE QuickSort (VAR a: ARRAY OF T; cmp: CmpProc);
  PROCEDURE Swap (i, j: LONGINT);
    VAR x: T;
  BEGIN x := a[i]; a[i] := a[j]; a[j] := x
  END Swap;

  PROCEDURE SortRange (i, j: LONGINT);
  BEGIN
    (* Recursively sort a[i..j]. *)
    (* Repeatedly calls Swap and cmp. *)
  END SortRange;
BEGIN
  SortRange(0, LEN(a) - 1)
END QuickSort;
```

Figure 1: Simple example of nested procedures and procedure types in Oberon-2

one important application of procedure values that is permitted by the original rule, but not by the new ones, a simple language extension is suggested in Sec. 5 to overcome this limitation, too. The paper closes with a brief sketch of implementation ideas in Sec. 6 and a concluding discussion in Sec. 7.

2. Examples of Nested Procedure Values

If the procedure `QuickSort` of Fig. 1 would be a nested procedure – for instance, because the element type `T` is a local type –, and its enclosing procedure wants to call it with two or more different comparison procedures, the latter obviously must be nested procedures, too.

As another example, Fig. 2 shows a procedure `Trav` that recursively traverses in infix order a binary tree `t` containing integer values, executing a callback procedure `cb` for every node's value. In many applications of this procedure, it would be natural to use a nested procedure as callback procedure because of its ability to access local variables of its enclosing procedure. For example, Fig. 3 shows a procedure calculating the sum of all values stored in the tree `t` by calling procedure `Trav` with the nested procedure `Add` as callback procedure.

3. Reasons for Disallowing Nested Procedure Values

Unfortunately, nested procedures are *not* allowed as values of procedure types in languages such as Modula-2 and Oberon(-2) causing the above examples to be actually *illegal*. When considering the usefulness of the concept, this appears to be a completely unreasonable restriction at first glance. In the tree traversing example, for instance, it would be extremely unnatural to declare `Add` as a global procedure because this would require to declare the variable `sum` globally, too. However, there are two reasons justifying this restriction, although they are rarely explained in language reports or textbooks.

First, due to the fact that nested procedures can directly access variables of their enclosing procedures, it is more difficult and possibly less efficient to implement procedure types whose values might be nested procedures. While global procedures can be simply identified by the starting address of their code block, nested procedures usually need additional context information, e. g., a *display* or a *static*

```
TYPE
  Tree = POINTER TO Node;
  Node = RECORD
    val: INTEGER;
    left, right: Tree;
  END;
  CallbackProc = PROCEDURE (x: INTEGER);

PROCEDURE Trav (t: Tree; cb: CallbackProc);
BEGIN
  IF t # NIL THEN
    Trav(t.left, cb);
    cb(t.val);
    Trav(t.right, cb);
  END
END Trav;
```

Figure 2: Traversing a binary tree

```
PROCEDURE Sum (t: Tree) : INTEGER;
  VAR sum: INTEGER;

  PROCEDURE Add (x: INTEGER);
  BEGIN sum := sum + x
  END Add;
BEGIN
  sum := 0;
  Trav(t, Add);
  RETURN sum;
END Sum;
```

Figure 3: Application of procedure `Trav`

link chain [1, 15]. It has been shown, however, that this problem can of course be solved in principle, and that the resulting code is sufficiently efficient in practice [3].

The second reason for disallowing nested procedure values is the danger of creating *dangling procedure values* by assigning a procedure to a variable whose lifetime extends that of the procedure (cf. Sec. 4 for precise definitions of terminology). While the well-known problem of *dangling pointers* has been removed from Oberon(-2) by restricting pointers to refer to dynamically allocated storage which cannot be explicitly deallocated by the program, allowing nested procedure values would introduce similar and comparably serious security holes, as the example of Fig. 4 shows. Here, the nested procedure `B`, which accesses the local variable `a` of its enclosing procedure `A`, can be called via the global procedure variable `g` after its enclosing procedure has exited. Because the local variable `a` will no longer exist at that time, i. e., the location on the procedure stack where `B` expects that variable will contain some other data (e. g., a variable of another procedure or, even more seriously, crucial runtime information such as the return address of a procedure), the execution of `B` would erroneously overwrite that data, resulting in completely undefined program behaviour afterwards.

```
MODULE DanglingProcedureValue;
  VAR g: PROCEDURE;

  PROCEDURE A;
    VAR a: INTEGER;

    PROCEDURE B;
    BEGIN a := 1
    END B;
  BEGIN
    g := B
  END A;
BEGIN
  A;
  g;
END DanglingProcedureValue.
```

Figure 4: Example of a dangling procedure value

4. Alternative Language Rules for Oberon-2

In order to retain the benefits of nested procedure values without creating the danger of dangling procedure values, assignments of procedure values to "more global" variables must be forbidden. This can be achieved by replacing the original rule:

R0: Procedure values must be global procedures.

with the following set of definitions (L1 to L3) and rules (R1 and R2):

L1: As usual, the *lifetime of a variable* is defined as the execution time of its directly enclosing procedure. To simplify terminology, a module is treated as a top-level procedure for that purpose.
Of course, the lifetime of an *array element* or *record field* is identical to the lifetime of the enclosing array or record.
The lifetime of an explicitly or implicitly *dereferenced pointer* is defined as the execution time of the program, because a pointer always refers to dynamically allocated storage whose lifetime extends to the end of the program as long as it is referenced by at least one pointer.
The lifetime of a *VAR parameter* is also defined as the execution time of the program, because it might refer to a global or dynamically allocated variable.

L2: Likewise, the *lifetime of a procedure name* is defined as the execution time of its directly enclosing procedure.
That means in particular, that the lifetime of a procedure name is quite different from the execution time of a particular activation of this procedure. The former actually represents the time where the procedure can be correctly and safely invoked.

L3: A *procedure value* is either (i) a procedure name or (ii) the value of a procedure variable or (iii) the result of calling a procedure whose result type is a procedure type, either directly by its name or indirectly via a procedure variable.
In all these cases, the *lifetime of a procedure value* is defined as the lifetime of the procedure name or variable used to obtain the value.
In contrast to definition L1, which defines the lifetime of a `VAR` parameter used as a *variable*, the *value* of a `VAR` parameter is treated like the value of a regular local variable here, because it might actually refer to any variable.

R1: The assignment of a procedure value to a procedure variable (i. e., a variable of a procedure type) is forbidden, if the variable's lifetime exceeds the value's lifetime.
Passing a procedure value as an actual parameter is treated like an assignment of the value to the corresponding formal parameter, i. e., formal parameters are treated like variables.

R2: Returning a procedure value from a procedure is forbidden, if the lifetime of the returning procedure's name (not the execution time of the current procedure activation!) exceeds the value's lifetime. In particular, a procedure must not return a local procedure name or the value of a local procedure variable.
This rule is in accordance with the above definition of the lifetime of a procedure value that is obtained from a procedure call (L3).

Since procedure variables and values might be embedded in records or arrays, the above rules must be applied to these accordingly. Furthermore, it should be noted, that the above rules do not replace, but rather augment the other rules of the language. For example, in addition to rule R2, the general rule that the value returned by a procedure must be assignment compatible with the procedure's result type, has to be obeyed.

Normally, the relative lifetimes of two "objects" (procedure names or variables) are determined by lexical scoping: Objects declared in the same procedure obviously have equal lifetimes, while the lifetime of an object declared in a more global procedure exceeds the lifetime of an object declared in a more local procedure. (The lifetimes of objects declared in unrelated procedures never have to be compared.) As a special additional case, however, the lifetime of an actual parameter value always exceeds the lifetime of the corresponding formal parameter (which is identical to the execution time of the called procedure). Together with rule R1, this observation implies the important corollary that procedure values can be passed as parameters *without any restriction*.

5. An Additional Language Extension

Under these rules, the examples given in Sec. 2 are correct, because procedure values are only passed as parameters there and never stored in any other variables. On the other hand, the example shown in Sec. 3 will be rejected since the assignment of the nested procedure `B` to the global variable `g` violates rule R1.

Unfortunately, rule R1 also forbids the assignment of the parameter `handler` to the global array element `handlers[sig]` in Fig. 5, even if clients would call the procedure `Register` with global pro-

```
MODULE Signals;
  CONST Max = ...;
  TYPE Handler = PROCEDURE (sig: INTEGER);
  VAR handlers: ARRAY Max OF Handler;

  PROCEDURE Register* (sig: INTEGER; handler: Handler);
  BEGIN handlers[sig] := handler (* Forbidden by rule R1! *)
  END Register;

  PROCEDURE Signal* (sig: INTEGER);
  BEGIN handlers[sig](sig)
  END Signal;
END Signals.
```

Figure 5: A simple signal handling module

cedure values only. On the other hand, if module `Signals` would violate the strict information hiding principle by directly exporting the array variable `handlers` (which is used to associate with each signal number `sig` a signal handling procedure `handlers[sig]` that is executed when `Signal(sig)` is called), then clients would be allowed to assign global procedures to its elements. So the problem actually results from the fact that the *real* lifetime of the actual parameter value passed to `Register` is lost when it is assigned to the formal parameter `handler`, which is treated like a local variable of procedure `Register`.

To remedy this particular problem, the programmer needs a language construct to express the fact that the actual parameter values passed to `Register` shall always be global procedures. More generally, it must be possible to enhance a procedure variable with a *lifetime guarantee* expressing the minimum lifetime of its values.

Because the lifetime of an object is defined as the execution time of its directly enclosing procedure (definitions L1 and L2), the names of directly or indirectly enclosing procedures are well suited to express such lifetime guarantees. Therefore, the definition of a procedure type is extended with an optional *lifetime guarantee clause* `OF` *`name`* after the keyword `PROCEDURE` to express that variables of that type must not contain procedure values whose lifetime is shorter than the execution time of procedure *`name`*. In other words, only procedures declared in procedure *`name`* or in more global procedures are allowed as values of such variables. As a special case, it is also possible to use the keyword `MODULE` instead of the module's name to express that the value of a variable must be a global procedure.

In particular, replacing the declaration of type `Handler` in Fig. 5 with:

```
TYPE Handler = PROCEDURE OF MODULE (sig: INTEGER);
```

would cause the example to become correct, while at the same time forcing clients of module `Signals` to actually pass global procedures – or variables of type `Handler` – to procedure `Register`.

To generalize the rules stated in Sec. 4 to variables with lifetime guarantees, the term "lifetime" has to be replaced with "lifetime guarantee," which is generally defined as follows:

G1: The lifetime guarantee of a procedure variable is defined as the maximum of the lifetime guarantee of its type, if applicable (i. e., if the type possesses a lifetime guarantee), and the variable's lifetime.
Normally, the lifetime of a variable cannot exceed the lifetime guarantee of its type, because the latter is defined as the execution time of an *enclosing* procedure. Dynamically allocated variables, however, i. e., explicitly or implicitly dereferenced pointers, possess maximum lifetime (the execution time of the program) which might indeed exceed the lifetime guarantee of their type.
Similarly, `VAR` parameters possess maximum lifetime (cf. definition L1 of Sec. 4), which might therefore exceed the lifetime of their type, too.

G2: The lifetime guarantee of a procedure value is defined as the lifetime guarantee of its type, if applicable, or otherwise as the lifetime guarantee of the procedure name or variable used to obtain the value (cf. definition L3 of Sec. 4). For that purpose, the lifetime guarantee of a procedure name is defined identical to its lifetime.

Given that, the examples of Sec. 2 as well as the example of Fig. 5 with the procedure type `Handler` modified as above are correct.

To give an artificial example which is useful to study borderline cases, Fig. 6 illustrates among other things the effect of definition G2. Since the procedure type `Q` possesses a lifetime guarantee, the procedure value obtained by executing the procedure that is currently assigned to the procedure variable `b3` (which is actually the nested procedure `D`) can be assigned to the global procedure variable `q`, which would be forbidden by the original version of rule R1 and definitions L1 and L3. On the other hand, the analogous assignment of the expression `b1()` to the global variable `p` of type `P` is still illegal with the new definitions and rules, since the type `P` does not possess a lifetime guarantee and thus the lifetime guarantee of the expression's value is identical to the lifetime guarantee of the local varia-

```
MODULE Artificial;
  TYPE P = PROCEDURE; Q = PROCEDURE OF MODULE;
  VAR p: P; q: Q;

  PROCEDURE A () : P; ...... END A;

  PROCEDURE B;
    VAR b1: PROCEDURE () : P;
    VAR b2: PROCEDURE OF MODULE () : P;
    VAR b3: PROCEDURE () : Q;

    PROCEDURE C () : P; ...... END C;

    PROCEDURE D () : Q; ...... END D;
  BEGIN
    b1 := C;     (* Correct. *)
    b2 := C;     (* Illegal. *)
    b2 := A;     (* Correct. *)
    b3 := D;     (* Correct. *)

    p := b1();   (* Illegal. *)
    p := b2();   (* Correct. *)
    q := b3();   (* Correct. *)
  END B;
END Artificial.
```

Figure 6: An artificial example

ble `b1` which is in turn identical to its lifetime. To make such an assignment correct, it is necessary to supply the local variable itself with a lifetime guarantee, which is demonstrated for the variable `b2`.

6. Implementation Ideas

To enforce the alternative language rules suggested in Sec. 4 and to support the language extension introduced in Sec. 5, the *front end* of an Oberon-2 compiler has to be changed accordingly. Furthermore, it might be necessary to modify the compiler's *back end* to generate code that is able to handle nested procedure values.

If nested procedures cannot be used as procedure values, a compiler need not maintain a static link chain [1, 15] which is used at runtime to find *intermediate variables* (i. e., local variables of enclosing procedures), but simply transform nested procedures to global procedures which receive the addresses of all intermediate variables as additional parameters. For example, the freely available `oo2c` compiler [11] transforms Oberon-2 programs to ANSI C using this technique.

If nested procedure values are permitted, however, either a static link chain has to be set up – which is a frequently used technique, even in the absence of nested procedure values, but would require major modifications to such a kind of compiler – or the addresses of the intermediate variables of a procedure have to be stored in an appropriate record (resembling a functional *closure*) when the procedure's name is used as a procedure value. This record can be statically allocated on the procedure stack, just as if it has been declared in the same procedure as the procedure name that is used as a value. Instead of just containing the address of the procedure's code block, a nested procedure value then must contain an additional pointer to this record.

A completely different strategy for adapting the `oo2c` compiler would be to use GNU C instead of ANSI C as its target resp. intermediate language, because this already allows nested functions as well as unrestricted "pointers" to them (cf. Sec. 7).

7. Related Work and Conclusion

After describing the dilemma that nested procedure values are on the one hand very useful, but on the other hand forbidden in Oberon-2 and related languages for a serious reason (dangling procedure values), alternative language rules as well as a small language extension have been proposed to retain their benefits without the danger of running into trouble.

It is interesting to see in this context, that the problem solved in this paper does not even appear in many other programming languages. For example, standard C [6] and C++ [10] do not allow nested procedures at all, so the notion of nested procedure *values* is simply not applicable. (C++ *function objects* are a completely different matter; they can be used to some extent to simulate nested procedure values.) Breuel [3], however, describes a corresponding extension for C++ that is implemented in the GNU C (but interestingly not in the GNU C++) compiler. Following the typical style of C and C++, however, the problem of dangling procedure values (just as the problem of dangling pointers) is not addressed at the language level, but left to the programmer, i. e., he is responsible for using the concept correctly. Here, the approach presented in this paper could be applied in the same way as for Oberon-2 to make the language safer.

Likewise, Eiffel [7] does not provide nested procedures, while other object-oriented languages such as Java [2] do not provide procedure (resp. method) values at all, so *nested* procedure values are not appropriate either. (Eiffel *agents* are again a completely different matter, comparable to C++ function objects.)

Functional languages such as Lisp [16], Haskell [12], or ML [9] fully support nested functions including *closures*, which are just another name for nested procedure values. But since pure functional languages lack the notion of variables to which closures might be assigned, the problem of dangling closures can only appear when a function returns a locally defined function, which is forbidden in the present paper by rule R2, but typically allowed in functional languages. In such a case, the runtime system takes care to retain the environment of a function (i. e., the non-local values it accesses) as long as necessary, even after the enclosing function has exited. While this solution is quite elegant and convenient from a conceptual point of view, its implementation is rather expensive since activation records cannot be simply put on a LIFO stack, but have to be dynamically allocated and garbage-collected on a heap.

When comparing the alternative language rules suggested in this paper with the original rule R0:

- Procedure values must be global procedures.

the former appear to be much more complicated than the latter. For most practical applications, however, the simple rule of thumb:

- Procedure values must not be assigned to more global variables.

is sufficient, while the rules given in Sec. 4 are just a more precise and complete specification of that. Furthermore, the corollary mentioned at the end of Sec. 4:

- Procedure values can be passed as parameters without any restriction.

covers a majority of practically relevant cases not involving lifetime guarantees. Finally, the concept of lifetime guarantees, which has been introduced in full generality in Sec. 5 to avoid any unnecessary conceptual restrictions, is usually only needed for the special and simple case `OF MODULE` (restricting the values of a procedure variable to global procedures), which is equivalent to the original rule R0.

References

[1] A. V. Aho, R. Sethi, J. D. Ullman: *Compilers. Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] K. Arnold, J. Gosling, D. Holmes: *The Java Programming Language* (Third Edition). Addison-Wesley, Boston, 2000.

[3] T. M. Breuel: "Lexical Closures for C++." In: *Proc. USENIX C++ Technical Conferenc* (Denver, CO, October 1988).

[4] C. Heinlein: "Safely Extending Procedure Types to Allow Nested Procedures as Values." In: L. Böszörményi, P. Schojer (eds.): *Modular Programming Languages* (Joint Modular Languages Conference, JMLC 2003; Klagenfurt, Austria, August 2003; Proceedings). Lecture Notes in Computer Science 2789, Springer-Verlag, Berlin, 2003, 144−149.

[5] C. Heinlein: *Safely Extending Procedure Types to Allow Nested Procedures as Values*. Nr. 2003-03, Ulmer Informatik-Berichte, Fakultät für Informatik, Universität Ulm, June 2003.
http://www.informatik.uni-ulm.de/pw/berichte

[6] B. W. Kernighan, D. M. Ritchie: *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[7] B. Meyer: *Eiffel: The Language*. Prentice-Hall, New York, 1994.

[8] H. Mössenböck, N. Wirth: "The Programming Language Oberon-2." *Structured Programming* 12 (4) 1991, 179−195.

[9] L. C. Paulson: *ML for the Working Programmer*. Cambridge University Press, Cambridge, 1991.

[10] B. Stroustrup: *The C++ Programming Language* (Special Edition). Addison-Wesley, Reading, MA, 2000.

[11] *The OOC Project*. http://ooc.sourceforge.net/.

[12] S. Thompson: *Haskell. The Craft of Functional Programming*. Addison-Wesley, Harlow, England, 1996.

[13] N. Wirth: *Programming in Modula-2* (Fourth Edition). Springer-Verlag, Berlin, 1988.

[14] N. Wirth: "The Programming Language Oberon." *Software—Practice and Experience* 18 (7) July 1988, 671−690.

[15] R. Wilhelm, D. Maurer: *Compiler Design*. Addison-Wesley, Wokingham, England, 1995.

[16] P. H. Winston, B. K. P. Horn: *LISP* (Third Edition). Addison-Wesley, Reading, MA, 1989.