Coroutines and Processes in Block Structured Languages.

A. Kreczmar
Institute of Informatics
University of Warsaw, PKiN
VIII floor, 00901 Warsaw, Poland

T. Müldner
School of Computer Science
Acadia University, Wolfville
Nova Scotia, BOP IXO, Canada

ABSTRACT.

This paper considers the semantics of coroutines and processes in block structured languages; in particular, the problem of existence of static and dynamic environments. It is shown that a definition of inaccessible module instances may result in an inconsistent meaning of some operations. Both an Algol-like language and a SIMULA-like language, (with pointers yet without coroutines), are proven to have well-defined semantics. The examples provided in this paper show that some coroutine and concurrent operations may, however, destroy the static environment.

1. INTRODUCTION.

The problem of the existence of the static and dynamic environments in block structured languages with coroutines and processes seems to be up-to-date, see e.g. ADA, [10]. The literature on coroutines and processes is rich and diverse see e.g. [7], [8] [9].

Should the structure of the module instances reflect an actual storage management system, it has to include an operation which deallocates inaccessible instances. The deletion of an instance, however, may perturb the normal program execution since the structure of static and dynamic connections could be destroyed. For completeness, Section 2 quotes the well-known results concerning the semantics of Algol-like languages. The following section comprises the corresponding analysis of language with pointers yet without coroutines. Section 4 introduces coroutine and semi-coroutine operations, and examines their semantics. The last section extends the analysis to the languages with concurrent processes. A conclusion is that the languages with coroutines and processes do not satisfy the basic requirement of an existence of the static environment. Therefore, a new approach to storage management and referencing mechanisms is needed.

2. BLOCK STRUCTURED LANGUAGES.

This section will introduce some basic concepts and recall the main properties of Algol-like languages (see [2,3,11]). Let Y be any syntactic entity, such as a variable or a module. We write Y **decl** M for Y is declared in the module M. For any program, the set T of all its modules with the relation **decl** forms a tree denoted **T[decl]** = <T,decl> with the main block (MB) as its root. For any binary

relation R, denote by R+ (R*) the transitive (and reflexive) closure of R.

A module N is a static container (cf [2]) for the occurrence of the identifer X in a module M, N = SC(X,M), if (i) X **decl** N, (ii) M **decl*** N, (iii) there is no module N' for which M **decl*** N' **decl+** N and X **decl** N'. #

The instances of a module M will be denoted by P(M), Q(M), etc. (with indexes, if necessary), or simply by P, Q, etc. A state of the program execution is considered as a finite set of instances that exist when a snap-shot of the execution is taken. The states will be denoted by S, S', etc.

The changes of states will now be considered. According to the syntactic structure for an occurrence of the identifier X in the module M, the instance P(N) of the module N = SC(X,M) is accessed. The instance P is called the dynamic container for the occurrence of X in M, F = DC(X, M) (compare [2]). For any instance P (except of MB) another instance Q, called the syntactic father of P, will be uniquely defined (see def. 2). The relation between P and Q will be denoted by P => Q. The main property of => is

(2.1) if P(M) => Q(N) then M **decl** N. #

The relation P => Q will sometimes be denoted by P.SL = Q, because P's **S**tatic **L**ink points to Q.

Definition 1.

The sequence $P_k,...,P_1$ is the static chain of the instance P_k, if P_{i+1} => Pi for i = k-1,..,1; and P_1 is the instance of the main block. #

The existence of the static chain of the currently executed instance will be proved later (see 2.5). From (2.1) and the above definition, it follows:

(2.2) If $P_k(M_k)$, $P_{k-1}(M_{k-1}),..,P_1(M_1)$ is a static chain of the instance P_1 then $M_k,..,M_1$ forms a path from M_k to the root MB, in the tree **T[decl]**. #

(2.3) If $P_k(M_k)$, $P_{k-1}(M_{k-1}),..,$ $P_1(M_1)$ is the static chain of the instance P_k then for any occurrence of an identifier Y, such that the static container $SC(Y,M_k)$ exists, there is a unique j, $1 \leq j \leq k$, for which $M_j = SC(Y,M_k)$. #

For any state S of a program execution we define a structure **S[syn]** = <S, =>>. This structure reflects the syntactic structure of the program. When a control enters a module, say M, a new instance P(M) is generated. Therefore the structure **S[mem]** with the operation **insert**(P) is introduced. The control structure of the program will be described by means of another structure **S[dyn]** = <S,->> where the relation -> determines a dynamic father. If P -> Q we shall also write P.DL = Q because P' S **D**ynamic **L**ink points to Q. An active instance at state S is the instance which is being executed at S. In all sections, except the last, we consider sequential languages for which at most one instance is active at a given state. Below, the relations =>, and -> are defined, and moreover, the transitions between states are determined:

Definition 2.

Consider a state S at which an instruction **call** F is executed in the active

instance P(N). Suppose that the static container M = SC(F,N) and the static chain of M exist. Then, by (2.2), a unique instance R(M) of the module M belongs to this static chain. The generation of a new instance Q(F) results in the following actions: (i) **insert**(Q) for **S[mem]**; (ii) add an edge Q => R for **S[syn]**; (iii) add an edge Q -> P for **S[dyn]**; (iv) the instance Q becomes active. The termination of the instance Q with the dynamic father P (Q->P) results in the following actions:
(i) delete the edge Q -> P for **S[dyn]**;
(ii) the instance Q becomes active. #

The following propositions describe the properties of the structures **S[syn]**, **S[dyn]**. The proofs are straightforward and are therefore omitted.

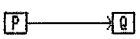
(2.4) If P => Q then non Q ->* P. #

(2.5) The structure **S[syn]** is a tree; the static chain of the active instance P forms a path from P to the root (so this chain always exists). #

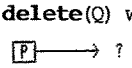
(2.6) The structure **S[dyn]** consists of a chain, (called operational chain) with the active instance as its leaf, and a number of isolated nodes. #

Clearly, any real memory management system cannot afford allocating more and more memory fields without any garbage collection. Therefore, the structure **S[mem]** will have an additional operation **delete**(P) which deallocates the instance P. This operation, however, may cause the structures **S[syn]** and **S[dyn]** to no longer be graphs.

If



then after **delete**(Q) we could obtain



We shall call such an edge a pseudo-edge, and a structure with nodes and pseudo-edges, a pseudo-graph.

The termination effect (def. 2) is redefined by adding the action.

(iii) **delete**(Q) for **S[mem]**. #

We shall investigate the following questions:

(2.7) When an instance becomes inaccessible, and what does it really mean?

(2.8) When the inaccessible instances should be deallocated?

(2.9) What are the consequences of the deallocation for the semantics of the language?

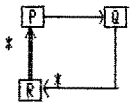
For an Algol-like language, an instance Q, is said to be accessible from the instance P at a state S iff P ->* R =>* Q (for some instance R.) The instance Q is said to be inaccessible in a state S, iff Q is not accessible from the active instance.

The following proposition answers the question 2.7:

(2.10) A terminated instance is inaccessible at any state.

The proof goes by induction. Let a state S' satisfy (2.10), and state S be obtained from S' as a result of a termination of the instance P. Then P is

inaccessible from the dynamic father Q of P, because this contradicts (2.4).



If P' ≠ P is a terminated instance accessible from Q (which is active at S): P -> Q ->* R =>* P' then P' would be accessible from P at S', which contradicts the inductive assumption. #

The proofs of the following propositions are simple enough to be left to the reader:

(2.11) The structure **S[syn]** is a tree, the active instance being its leaf. #

(2.12) The structure **S[dyn]** consists of an operational chain the active instance being its leaf. #

(2.13) The syntactic environment of the active instance is always defined, i.e. the static chain of the active instance P(M) exists and contains all the dynamic containers for all occurrences of identifiers in M. #

In accordance with the above follows the well known property of standard implementation of block structured languages follows:

(2.14) A block structured language is "stack implementable", i.e. **insert** and **delete** operations of **S[mem]** are performed in the LIFO scheduling strategy. #

In the following sections we shall investigate the semantical properties of the languages which extend an Algol-like language with the following properties:

- storage management, the terminated instances are accessible;
- control structure, the instance can be re-entered;
- parallelism, more than one active instance may exist at a time.

3. POINTER LANGUAGES.

The main feature of a language with pointers is that a terminated instance can be accessed via the pointer, (i.e. reference variables) the value of which is the address of the instance. Using SIMULA notation (certify [6]) for a reference variable X of a module type M, the instruction

X: = **new** M results in a generation of memory field for the instance P(M), an execution of M's instruction (if any), and eventually, assigning the address of P(M) to X. The relation between X and P(M) will be denoted by X ->> P(M). Similarly, Q ->> P means that the instance P is pointed to by an attribute of the instance Q. The static container for dotted identifier is defined as follows: Consider the occurrence of X.W in a module N. The module N' = SC(X,N) contains the declaration of, e.g. **var** X:M. Therefore, the module N" = SC(M,N') contains the declaration of M. If M has the attribute W, then M = SC(X.W,N), otherwise the program is syntactically incorrect. Note that (2.2) from Section 2 still holds, while (2.3) is no longer true. Consider the following:

Example 1.

```
unit N: class;
   ...
   unit M: class;
      var W: integer;
   end M;
   ...
   unit N1: class;
      var X: M;
      unit N2: class;
         ...
         ... X.W ....
         ....
      end N2;
      ...
      X:= new M;
      ...
      end N2;
      ...
      X: = new M;
      ...
   end N1;
   ...
end N;
```

Then we have P(N2) =>+ P(N1) =>+ P(N) and P(M) =>+ P(N) where X points to P(M), hence P(SC(X.W,N2)) does not belong to the static chain of P(N2). #

A generation and a termination of an addressable instance are described as follows:

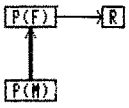
Definition 3.

The description of a generation is similar to that of Def. 2. However, if a new instance Q(N) is indirectly generated from the active instance P(M), via X.N, then the syntactic father of Q is the object pointed to by X. Now let us consider a termination. Let P be the dynamic father of the instance Q, i.e., Q-> P. Then the following actions will be performed: (i) **delete** Q -> P for **S[dyn]** (i.e. Q.DL becomes none): (ii) the instance P becomes active; (iii)if Q is an addressable instance generated by means of X:=**new** M instruction, then X points to Q. #

From this definition the analogon of (2.4) follows immediately:

(3.1) If P => Q, then **non** Q ->* P. #

Note that a structure **S[syn]** does not have to be a tree any longer: Suppose that the generation X:= **new** M takes place within the body of a procedure F, M is declared in F1 and F is called from R:



After the termination of F, R becomes active, P(F) is deleted, but P(M) remains alive and without a syntactic father:

This situation will not harm the execution of a program provided P(M) will be inaccessible. The latter notion has to be redefined in a language with pointers:

Definition 4.

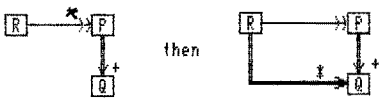
An instance Q is accessible from an instance P at a state S iff

P -> * R=>* R' ->>* Q for some instances R, R'. #

(3.2) If Q is non-addressabble, and

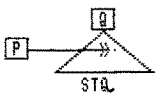
Proof. P(M) =>+ Q(M') so M is nested in M'. Moreover, R(N) ->> P(M) , so N contains the declaration of variable X of type M. Therefore, a static chain of R contains an instance Q'(M'). Our purpose is to show that Q = Q'. Suppose the contrary is true. Two different instances of the same module may communicate only via non-local variables, or via parameters. The first case is excluded, i.e. the reference to P cannot be transmitted in a remote expression via Q to Q.SL because M' is a non-addressable module (i.e. procedure /block instance). The second case is excluded because M is nested in M' and a parameter of M' has to be of a type which is visible from M'. #

(3.3) If Q is non-addressable and



#.

A terminated non-addressable instance may be a root of static sub-tree. A structure **S[syn]** is a pseudo-tree with pseudo-edges. By virtue of (3.3), if Q is a non-addressable instance and there is a reference chain between an instance P and a node of Q's subtree **STQ** then P belongs to **STQ**:

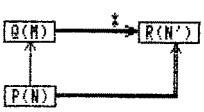


The analogons of (2.5) and (2.6) are the following:

(3.4) The structure **S[syn]** consists of a single tree **S[T]** and a number of pseudo-trees. Any instance P accessible from the active instance belongs to **S[T]**.

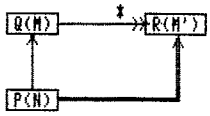
(3.5) The structure **S[dyn]** consists of a chain and a number of isolated vertices, the active instance is the leaf of the chain.

Proof. These propositions will be proved by simultaneous induction. If S' consists solely of the instance of MB, then the proof is trivial. Let S' be a state with the active instance Q(M). Consider a generation of an instance P(N). Put S = S' u (P). If P is directly generated by means of a **new** N instruction then N is visible from M; so the syntactic father R(N') of P(N) belongs to the static chain of Q(M):

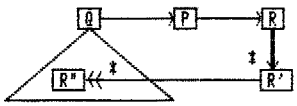


From the inductive assumption R(N') belongs to **S'[T]**, so R(N) belongs to **S[T]**. Therefore, **S[T]** consists of **S'[T]** augmented by the leaf P(N). Any instance accessible from P at state S is accessible at the state S', either from Q or from R, therefore from the inductive assumption, (3.4) holds. If P is indirectly

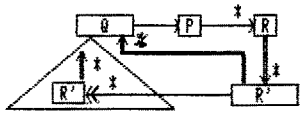
generated by means of X.**new** N instruction then



By the inductive assumption R belongs to **S'[T]** and so to **S[T]**. Therefore (3.4) holds. Now the termination of Q will be considered. Put S=S'- (Q). The only non-trivial case is that of non-addressable Q. Clearly **S'[T] = S[T] - STQ**, so we shall prove that



is not possible. Note that R $\neq$ R" = Q cannot hold because Q is non-addressable. Hence R =>* Q and from (3.3) R' =>* Q:



Therefore Q ->* R and R =>* Q which contradicts (3.1). #

The following analagon to (2.10) holds:

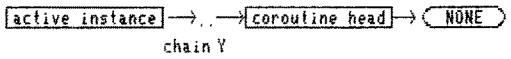
(3.6) Any accessible instance belongs to the tree **S[T]**. Hence, the syntactic and dynamic environment of the active instance is always defined as follows: the static chain of the active instance exists, and moreover, the dynamic containers (for all the occurrences of identifiers in Q) belong to the tree **S[T]**. #

Let us return now to the questions (2.7)-(2.9). A language with pointers is not "stack implementable" and we encounter one of the most difficult implementation problems. There are two well-known memory management techniques, (see [3]): - **retention** technique, which retains an instance as long as this instance is accessible (this requires expensive garbage collection to search inaccessible instances); - **deletion** technique, which deletes non-addressable instances immediately after termination. The latter technique may be fully exploited by virtue of (3.4); together with a non-addressable instance, the entire subtree of this instance may be deallocated. The reader may refer to [1], and [2] for more detailed discussion of the subject and application to the implementation of universal programming language LOGLAN.

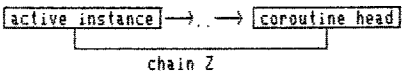
4. COROUTINE LANGUAGES.

The term "coroutines" is used for the module instances able to cooperate in a sequential fashion. This means that the execution of a coroutine instance can be suspended, and at the same time the operations of another coroutine instance are resumed. The coroutine instance may create a number of module instances that are

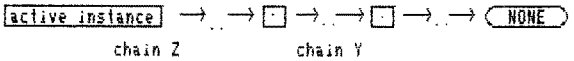
dynamically contained within it (e.g. some instances of procedures and blocks). These module instances form a coroutine chain, say Y:



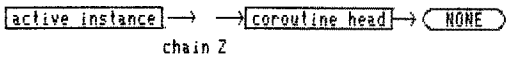
The control transfer from a coroutine chain to another one is a result of a certain coroutine operation, say **attach.** The suspended coroutine chain, say Z, will be pictured as follows:

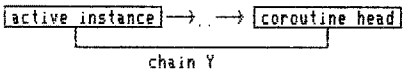


One can define the result of **attach** in the following way:



One can also define this result as follows:

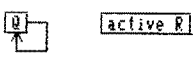




In this section we shall consider the second possibility, because we regard processes to be considered the special cases of coroutines.

A coroutine is generated by **new** instruction. The generation is completed when **return** instruction is performed; if Q(C) is a coroutine instance with a dynamic father R:

then **return** will suspend Q and resume R:



The main program MB is considered to be a coroutine instance pointed to by a system variable **main.** The user is alowed to use this variable only in the instruction **attach(main).** A chain is active if control executes its active instance, otherwise it is suspended. Directly from the definitions follows:

(4.1) A coroutine chain contains neither generated nor terminated coroutine instances except of the head of a chain. #

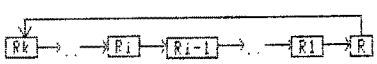
The analogon to (3.1) has the following form:

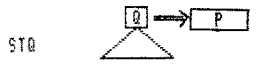
(4.2) If P => Q and Q does not belong to the dynamic chain leading to the head of a suspended coroutine chain then **non** Q - >* P. #

Lemma (3.2) is still valid while another auxiliary lemma is necessary to prove the analagons of (3.4), (3.5):

(4.3) If a suspended coroutine head R is accessible at a state S, then any instance of its chain is in **S[T].**

Proof. The only non-trivial case is that of a termination of a non-addressable instance Q. Consider the coroutine chain of R and the instance Q with the dynamic father P:





Proof goes by induction on k. Suppose that R_{i-1} belongs to **S[T]**. If $R_i = Q$ then R_i does not belong to the chain of R after the termination of Q, so (4.3) holds. Suppose that R_i =>+ Q. The instance R_i cannot be created in R_{i-1} directly, because R_{i-1} =>* R_i.SL implies R_{i-1} belongs to **STQ**. Therefore R_i is created in R_{i-1} indirectly by means of X.**new** M, so R_{i-1} =>* R'->>* R_i. SL. The instance Q is non-addressable, so $Q \neq R_i$.SL; and from (3.3)

Now R_{i-1} =>* R' =>* Q implies that R_{i-1} belongs to **STQ** which contradicts the inductive assumption. #

The analogons of (3.4), (3.5) have the following form:

(4.4) The structure **S[syn]** consists of a tree **S[T]** and a number of pseudo-trees. The active instance and any accessible instance belongs to **S[T]**. #

(4.5) The structure **S[syn]** consists of a single operating chain, a number of suspended coroutine chains, and a number of isolated vertices. The active instance is a leaf of the operating chain. #

The following example illustrates (4.3).

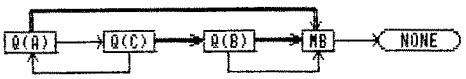
Example 2.

Consider the following the program:

```
begin
unit B: procedure;
   var X: C;
   unit C: coroutine;
      begin
         return;
         call A;
      end C;
   begin
      X: = new C:
      attach(X);
   end B;
   unit A: procedure;
      begin
         attach(main)
      end A;
begin
   call B;
end
```

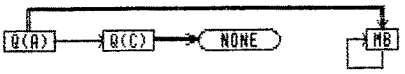
Just after the execution of **attach(main)** the state of computation looks as

follows:



Thus the head Q(C) belongs to the subtree with the root Q(B), while the instance Q(A) does not. After the termination of Q(B), the head Q(C) will be inaccessible. #

Note that in virtue of the above propositions, the proposition (3.6) still holds. For some applications semi-coroutines are necessary. There is an additional operation **detach** on semi-coroutine which returns control to the callee, i.e. the coroutine head which recently resumed this semi-coroutine. Unfortunately, such a semi-coroutine operation may destroy a syntactic environment: if one adds **detach** after the instruction "**call** B" in the example 2, then after the termination of Q(B) we have:



Hence, Q(A) is active, Q(C) is accessible from Q(A), but Q(C) does not belong to **S[T]**. This example shows that (4.4) does not fully hold. The structure **S[syn]** consists of a tree **S[T]** and a number of pseudo-trees but accessible instances need not belong to **S[T]**. So some kind of run-time checking is indispensable. Note that the dynamic structure is not harmed by the operation detach.

5. PARALLEL LANGUAGES.

We consider a coroutine as a particular kind of a process; for a program with processes more than one instance may be active at a time. There is an important difference, however, between coroutines, and processes which are performed by a single multiplexed processor. In the former case the switch of the control from one coroutine chain into another one is programmable, and so it cannot happen behind the scenes. In the latter case, it is the scheduler which switches the control, so a user has no control of this, and has to consider all the active processes as the operating ones.

The operations on processes are simply extensions of those on coroutines; **stop** suspends a process; **resume**(X) resumes the process pointed to by X. It is easy to see that the basic theorems describing with syntactic and dynamic structure slightly generalize (4.3) and (4.4):

(5.1) The structure **S[syn]** consists of a tree **S[T]** with main program being its root, and a number of pseudo-trees. #

(5.2) The structure **S[dyn]** consists of a number of operating chains, and a number of suspended process chains, suspended coroutine chains, and isolated vertices. The active instances are the leafs of the operating chains. #

For parallel languages a syntactic environment may be destroyed, and so appropriate memory management systems have to be developed. A **retention** technique delays a process termination if that could destroy a syntactic environment of another process. A **deletion** technique deletes inaccessible instances, but a process may explicitly wait for the termination of his sons.

It is worthwhile to notice that one can consider a programmable deallocation technique: an instruction, say **kill**(X) deallocates a memory field pointed to by X, (in PASCAL: **dispose**, in ADA: **free**). The point is that such an operation can be "secure", i.e. the access to memory instances killed as a side-effect of these instructions results in run-time error. Because of the lack of space, further issues of memory management systems and of processes synchronization will be considered in a forthcoming paper.

BIBLIOGRAPHY

[1] Bartol, W. M., Kreczmar, A., Lao, M., Litwiniuk, A., Müldner, T., Oktaba, H., Salwicki, A., Szczepanska-Wasersztrum, D., Report on the Programming Language Loglan 79. Internal Report, University of Warsaw.

[2] Bartol, W. M., Kreczmar, A., Litwiniuk, A., Oktaba, H., Semantics and Implementation of Prefixing at many Levels. IInf UW Report NR 94 Institute of Informatics, University of Warsaw.

[3] Berry, D. M., Block Structure: Retention vs. Deletion. Proc. Third Symposium on Theory of Computation, 1971, The MIT Press, 1979.

[4] Bobrow, D. G., Wegbreit, B., A Model and Stack Implementation of Multiple Environments. BBN Report 2334, 1972.

[5] Brinch-Hansen, P., Operating System Principles. Prentice Hall, 1973.

[6] Dahl, O. J., Myhrhaug, B., Nygaard, K., Common Base Language. NCC S-22, October, 1970.

[7] Dahl, O. J., Wang, A., Coroutine Sequencing in a Block Structured Environment. BIT 1971, pp. 425-49.

[8] Lindstrom, G., Soffa, M. L. Referencing and Retention in Block Structured Coroutines. ACM TOPLAS Vol. 3, N 3, July 1981.

[9] Naur, P., (Ed.), Revised Report on the Algorithmic Language ALGOL 60. CACM 6, 1963, pp. 1-17.

[10] Organick, E. I., Computer System Organization. The B5700/6700 Series. Academic Press 1973, New York.

[11] Reference Manual for the ADA Programming Language, United States Depart. of Defense, July 1980.

[12] Wegner, P., Programming Languages - Concepts and Research Directions. In: Research Directions in Software Technology, edited by P. Wegner.

[13] Wirth, N., The Programming Language Pascal. Acta Informatica 1971, 1, pp. 35-63.