PUC

Series: Monographs in Computer Science
and Computer Applications
Nº 4/76

GENERATOR FUNCTIONS IN MODULAR PROGRAMMING

by

Arndt von Staa

Departamento de Informática

Pontifícia Universidade Catolica do Rio de Janeiro
Rua Marques de Sao Vicente, 209   ZC 20
Rio de Janeiro   Brasil

Series: Monographs in Computer Science
and Computer Applications
Nº 4/76

GENERATOR FUNCTIONS IN MODULAR PROGRAMMING*

by

M 2084

Arndt von Staa

Series Editor: Sérgio E.R. Carvalho March, 1976

* This work was partially supported by the Brazilian Government Agency FINEP under contract nº 244/CT, and developed by the Information Systems Group of the Departamento de Informática-PUC/RJ

Copies may be requested from:

Rosane T.L. Castilho, Head
Setor de Documentação e Informação
Deptº de Informática - PUC-RJ
R. Marquês de São Vicente, 209 - Gávea
20000 - Rio de Janeiro - RJ - BRASIL

ABSTRACT:

Generator functions are defined as being functions which allow sequential accesses to successive elements(values) in some sequence (set). We study in this paper the basic characteristics of generator functions as well as their implementation.

KEY WORDS:

Modular programming, programming language design, program module, generator functions, coroutines, recursive coroutines, values of type *type*.

RESUMO:

Funções geradoras são funções que permitem o acesso sequencial a elementos (valores) contidos em uma sequência(conjunto). Neste trabalho nós estudamos os requisitos básicos de funções geradoras bem como a sua implementação.

PALAVRAS CHAVE:

Programação modular, projeto de linguagens de programação, módulo de programa, funções geradoras, corotinas , corotinas recursivas, valores de tipo *tipo*.

CONTENTS

1 - INTRODUCTION

Some of the desirable features of program modules are: context independence and syntactic non-interference [Dennis 73], strength and low degree of coupling [Myers 73], and transparency (property by which modules offer no insight into their implementations) [Parnas 71].

Modules should be able to implement special forms of memories such as stacks, sequences, trees, associative arrays (hash tables) etc. There must then be functions which enable users to access elements of these memories, i.e. <u>access functions</u>. These functions could access specific elements based on some sort of <u>data space identification</u> [von Staa 74, von Staa 75]. Access functions could also sequence through the elements of the set of elements contained within the memory implemented, i.e., return successive values upon each call to the function. This kind of function we call <u>generator function</u> and it will be the object of our study.

One of the prime advantages of generator functions is that we are able to reduce considerably the cases in which data space identifications (references, pointers) become necessary (or useful). Another advantages is that, together with transmission of types [von Staa 74, Liskov et alii 74] we are able to implement abstract data types, where these abstract data types contain also all the traversing algorithms necessary for its use. For instance, the type "tree" is defined containing the tree traversal algorithms. This reduces considerably the task of writing and certifying programs and increases "modularizability".

Generator functions are usually defined internally to some module, e.g. type descriptor. This has the advantage of keeping together in one module the set of valid operations on an internal data space. Thus generator functions increase the structuring of programs as well as the interchangeability of modules. Due to their being internal to other modules, generator functions know both the "structure" of the underlying set and the order (sequence) in which the elements of this set are to be delivered. This allows, then, the design of efficient generator functions without having to destroy the transparency of module implementation.

This paper characterizes generator functions and shows how to implement them. Some existing languages, e.g. SIMULA [Dahl et alii 72], implement generator functions. These implementations impose some restrictions which we attempt to eliminate.

2 - THE CONCEPT OF PROGRAM MODULE

Our characterization of the concept of a program module, or simply module, is similar to the one introduced by SIMULA with its class, or by CLU with its cluster [Liskov et alii 74].

Specifically a program module defines:

i - internal data spaces, internal type descriptors and internal functions which are not known externally to the module;

ii - visible type descriptors and functions which are made explicit to the exterior of the module, i.e. are within its outside scope.

Modules may act as type descriptors, or simply as types, e.g. integers, stacks, lists, trees. For example, the type integer can be described by a module defining an internal data space, a word, and the visible operations copy, fetch, add, multiply, etc. An integer variable declaration is then nothing more than the creation of an instance of the module describing this type, e.g. an "object" of a SIMULA class.

Type descriptors are information and, consequently, may be considered values of the type type. It is conceivable, then, to allow the existence of variables and parameters of this type. Such variables and parameters allow the practice of modular programming where the implementation characteristics of a given type descriptor are effectively hidden from its users [Liskov et alii 74, von Staa 74]. In most cases, type checking will be performed statically. The conditions under which this is possible are examined in [von Staa 74].

```
type: queue of (type: user_type) is
   begin queue;
      type: node is struct(ref node: next; user_type: info);
      ref node: head, tail;
      head:=tail:=null
      outside_scope functions;
         user_type: function dequeue is ...
         function enqueue(user_type data) is ...
      end functions;
   end queue;
```

FIGURE 1. Example of a type descriptor with a parameter of type type.

Figure 1 shows an example of a type descriptor which uses a parameter of type type. Due to this parameter we may implement the module "queue" without having to define completely the information stored within the elements of a queue.

It should be clear from this example that "user_type" does not need any information about "queue" and, conversely, "queue" does not need any information about "user_type" except for the size of the space occupied by an instance of "user_type".

3 - GENERATOR FUNCTIONS

We will study generator functions in a broader scope than just as access functions. That is, a function which computes the successive elements of some sequence for each successive call will also be considered a generator function. For example, a random number generator is considered a generator function, although it is not necessarily implemented as such.

Generator functions usually possess internal parameters whose values must be kept from activation to activation. By means of these parameters the current (or next) element to be accessed is determined. These internal parameters are updated for every activation of the generator function. That is, generator functions usually produce internal side_effects delivering different values for successive activations.

In traditional languages the internal parameters could be declared as global variables, consequently being kept from activation to activation. In modular programming this is not acceptable, though, since it exposes the implementation of the function, increases data coupling and may cause syntactic interference.

In order to produce the current (or next) element of the sequence to be processed, the generator function may have to resume elaboration where it last went off. We may conclude then:

Property 3.1 - In order to be able to implement generator functions in their fullest generality,coroutine handling facilities must be provided.

We understand coroutines [Conway 63, Gentleman 71] as being program modules which are capable of being suspended and later resumed at the same spot. An immediate consequence of this definition is that activation records of coroutine instances must be kept even when these instances are inactive.

More on coroutines will be given later in this paper.

Generator functions are further distinguished from conventional functions in that they usually define the following three entry points:

i - <u>initialization entry</u> - which prepares the generator function to deliver the first element of the set, or actually delivers it;

ii - <u>successor entry</u> - which advances the element "cursor" by one element. That is, by repeatedly activating the generator function through the successor entry, we are able to effectively scan the elements in the ordered set (sequence);

iii - <u>termination entry</u> - this is a predicate which determines whether all elements of the set have already been examined.

Notice that all of these entry points do not have to be provided for each generator function. For example, a random number generator could be initialized whenever an instance of it is created. Furthermore, termination predicates are usually absent from random number generators. Thus, a random number generator may be implemented as a conventional function with static storage, although it is considered to be a generator function.

From what has been said so far we may conclude:

<u>Property 3.2</u> - Generator functions are modules possessing their own storage requirements and making one or more manipulative operations available to the exterior.

Observe that type descriptors have the same properties. Thus the mechanisms developed for type descriptors in [Liskov 74, Dahl 72, von Staa 74] could also apply to generator functions.

Consider now the following problem. Given some set S, produce a listing containing all ordered pairs of elements in S. This could be achieved by a program similar to that

```
set: S(some_type);
generate A:=S.elem_gen.first by S.elem_gen.successor
                    until S.elem_gen.last do;
   generate B:=S.elem_gen.first by S.elem_gen.successor
                       until S.elem_gen.last do;
      output A, B;
   od;
od;
```

FIGURE 2 - Ordered pair generator. First version.

of figure 2. We are not concerned here with what these sets represent, e.g. data base records bearing a given property, nodes of a tree or a graph. What we want to point out though, is that successive elements of such sets cannot be obtained by simple addition, e.g. indexing.

The construct:

```
generate <control_var>:=<origin> by <successor>
               until <termination> do ... od
```

is similar to the ALGOL 60 for statement. It is used with respect to generator functions, though. Thus, <origin>, <successor> and <termination> are, respectively, the initialization, successor and termination entries of the generator function. <control_var> may be implemented as an access typed [von Staa 76] variable and refers to the element of the set which is currently being processed.

Within the program of figure 2, "set" is a type which, besides storing elements of some type ("some_type") also makes the generator function "elem_gen" available to the exterior. This generator function scans all elements in the set "S". "elem_gen" provides the following entry points:

i - "first" which resets "elem_gen" and provides an access typed value refering to the first element in the set;

ii - "successor" which advances access to the next element in the set;

iii - "last" is a predicate which returns true iff all elements of the set have been examined.

Suppose now that there were only one instance of the generator function "elem_gen". When the internal loop, i.e. "generate B=...", terminates, the generator function instance is necessarily in a state where the until test yields true. Now, when terminating the internal loop, the external loop is resumed, i.e. the next left hand element of the ordered pair is generated. Since, by assumption, there is only one instance of the generator function, it follows immediately that the external loop is also terminated. Consequently the program shown would be in error, since only those ordered pairs are listed for which the left hand element is the first element of the set being traversed. Generalizing we have:

<u>Property 3.3</u> - There may be several instances (activations) of a generator function, each at a different stage of elaboration and each possibly related to the same data space or module instance.

Observe that even when the generator function is a simple addition operation the above is true. In this case, the multiple instances are usually embedded into the program's code, e.g. by multiple expansions of the for loop "macro".

We have already mentioned that generator functions and type descriptors are quite similar from the implementation point of view. Thus, instances of generator functions could be created in the same way as data spaces of a given type are created, i.e. by means of a declaration. We will assume then, that all generator function instances are declared.

In figure 3 we show an example of generator function instance declarations. Since "outer" and "inner" are different instances of the generator function "elem_gen", it follows immediately that there is no interference between the two generate statements in figure 3.

```
set: S(some_type);
with outer = S.elem_gen generate A:=outer.first by
          outer.successor until outer.last do;
   with inner = S.elem_gen generate B:=inner.first
             by inner.successor until inner.last do;
      output A, B;
   od;
od;
```

FIGURE 3 - Ordered pair generator. Second version.

In figure 4 we show the type descriptor "set". As mentioned earlier, the type "set" makes available the generator function "elem_gen". The example shown should be clear, since it merely reflects exactly all that has been said about generator functions so far.

A generator function instance is created whenever control passes through the program section elaborating the declaration made explicit by the with portion. Thus, the internal space of a generator function, i.e. the activation record, is allocated and remain allocated up to the end of the with block even if none of the operations defined by the generator function are actually elaborated.

There are several creation steps associated with generator functions. One step is the creation of the code sections and occurs usually at compile (load) time. A second creation step occurs when creating an object of the type described by the type descriptor module. This step may occur at compile time, and does so in many present day languages, e.g.

ALGOL, FORTRAN, SIMULA. A third creation step occurs when a generator function instance is created during the elaboration of the with portion. A fourth creation step occurs when an internal function of the generator function is started to be elaborated (e.g. outer.first)

```
type: set of (type: user_type) is begin set;
   type: element is struct(ref element: next; user_type: info);
   ref element: head:=null;
   outside_scope set_operations;
      generator elem_gen is begin element_generator;
         ref element: current:=head;
         boolean: end_reached:=head=null;
         outside_scope entry_points;
            user_type: access function first is
               if end_reached then null; else head->info; fi;
            user type: access function successor is
               if end_reached then null;
                  else head:=head->next;
                     end_reached:=head=null;
                     if end_reached then null;
                        else head->info;
                     fi;
               fi;
            boolean: function last is end_reached;
         end entry_points;
      end element_generator;
   ...
end set;
```

FIGURE 4 - Definition of the type "set".

4 - COROUTINES

Let us consider now a generator function which performs the infix traversal of a binary tree. Whenever a node to be visited is found, the infix generator function must relinquish control to the calling procedure. Furthermore, there is a "past history", e.g. a stack, associated with the traversal algorithm. This past history must be preserved from activation to activation, possibly by the coroutine itself [Weizenbaum 63, Smith 73]. Thus, the generator function must be implemented as a coroutine. Now, the past history could be maintained in an implicit way by means of a recursive procedure. Thus this infix generator function could be implemented as a recursive coroutine.

Before proceding let us examine first the mechanism of passing control between modules. We refer the interested reader to [Johnston 71] for a detailed discussion about run-time configurations.

A module may receive control only through well defined entry points. There may be several entry points to one module. Every entry point defines an effective entry value which determines where elaboration has to begin (or resume) when control is transferred through this particular entry point. Effective entry values may be variable. For example, a coroutine usually allows elaboration to resume at one of several predefined points within the text. Whenever a coroutine deactivates, it sets the corresponding effective entry to refer to one of the elaboration resumption points. We need thus a deactivation construct which also sets the value of the effective entry to refer to the resumption points.

In order to pass control back to the caller, we need an effective return value. This value is also associated with the entry point through which control has been passed to the module. Thus, an entry point could be viewed as the name of a composite data space containing the following kind of values <effective entry; effective return>. It should be noted here that an effective return value could be a more complicated structure than just a return label, e.g. address. For instance, an effective return value could name the actual return point, as well as the entry points of several error recovery entries. In some cases the data space containing the effective returns may be shared by several entry points. This is the case, for example, in subroutines (or functions) which posses multiple entry points, e.g. the sine/cosine function.

A module instance is an "elaboratable" copy of a program module. Usually a module instance consists of a (shared) portion of code and a (non-shared) portion of working storage, i.e. an activation record. Module instances are created when a creation section associated with the module is elaborated. Usually there is only one creation section per module and this creation section is provided in an implicit form by the language processor. Observe that, when a module is a macro, the creation of a module instance corresponds to a macro expansion.

A creation section which is implied by the language processor is called a prologue and its elaboration precedes the elaboration of the first syntactically visible statement of the module. Thus, the effective entry value can easily be initialized to refer to an appropriate prologue in which case it will be called a creation entry.

Creation may occur all at once, e.g. as in FORTRAN IV, or it could occur in several steps, e.g. in ALGOL 60 the shared portion is created at compile (load) time, whereas activation records are created and destroyed during execution. Termination is the operation of destroying a module instance. It should be clear that, if a retention mechanism is implemented [Berry et alii 73] actual termination could be delayed with respect to the instant when the termination operation is performed.

A module instance may be activated only if it has already been created and has not yet been terminated. Thus, in order to activate a non existing module instance, first some creation section of this module must be elaborated. In order to prevent incorrect execution, the effective entry values must be initialized to refer to either a creation section or to **abort**. Observe that if control is passed to **abort**, an error condition is flagged and the program execution may reach a premature end. By a similar argument we may conclude that all effective return values must be initialized to **abort**.

Consider now recursive coroutines. With reapect to deactivations, a recursive coroutine may deactivate and return control to the exterior, or it may deactivate (terminate) and resume elaboration of another instance of this coroutine. This implies then the existence of at least two effective return values and, also, of two entry points. It implies furthermore, the necessity of deciding which effective return is to be used when deactivating. Since this decision depends on the algorithm to be implemented, it must be provided by the programmer. It follows then, that deactivations must be able to name the entry point containing the return point to be used.

In SIMULA, we are able to create module instances by means of the **new** operator and being, thereafter, able to deactivate (**detach**) and activate specific instances (**resume**) at will. However, from the point of view of modular programming, this has the inconvenience that we must tell the user that the generator function is a **class** and, thus, a different kind of module than other functions. For this reason we took the more elaborate approach of considering coroutines as being program modules rather than named objects of a given class.

Deactivating through specific entry points may cause some unusual control flow interactions between modules, since, in some cases, the effective return value to be used is not the one associated with the entry point used to activate the deactivating module. This is, some deactivations could be relative to an entry point associated with another sub module of the coroutine.

By initializing the effective entry value to become a creation entry, we may create and activate a coroutine with the first transfer of control (activation) to it. Subsequent activations of this coroutine can be prevented from creating a new instance, simply by setting the effective entry values to refer to some portion of the code which is not a creation section. It follows then, that coroutine instances need not be created by an explicit construct external to the coroutine.

In figure 5 we show a definition of the type "binary_tree". Within "binary_tree" we define the infix traversal generator function "infix". The purpose and implementation of this generator function has been described earlier.

Within "infix" two procedures (modules)are defined. The recursive procedure (co_function) "start_subtree" which performs the actual traversal, and the coroutine (co_function)

```
type:binary_tree of(type: user_type)is begin binary_tree;
   type: node is struct(ref node: left, right; user_type: info);
   ref node: root:=null;
   outside_scope operations;
      generator infix is begin infix
      co_function start_subtree(value ref node: pointer)is
         begin subtree;
            if pointer≠null
               then call start_subtree(pointer->left);
A:                  deactivate next(pointer->info);
B:                  call start_subtree(pointer->right);
            fi;
         end subtree;
         outside_scope only_known_entry;
            user_type: access co_function next is begin next;
C:                  call start_subtree(root);
D:                  repeat deactivate next(null);
            end next;
         end only_known_entry;
      end infix; ...
```

FIGURE 5 - A recursive coroutine as generator function

"next" which serves as a communications link with the exterior of "infix". That is, control can be passed to "infix" only through "next". It follows then, that "start_subtree" must deactivate through "next" whenever a node to be visited has been found. Observe that deactivation through "next" requires also the passing of an access typed value naming the node to be visited.

The effective entry value of "next" is initialized to refer to the prologue of the procedure body of co_function "next". Thus, "next" is a creation entry for a newly created instance of "infix". Traversal of the tree is initiated due to the statement "C" which invokes "start_subtree". The effective entry value of "next" is set to refer to statement "B" whenever deactivation occurs due to elaborating statement "A", thus avoiding the reelaboration of the prologue of the co_function "next". Thus, "next" becomes a simple activation entry when control passes through the statement "A" the first time. Once the traversal has been completed, "start_subtree" returns (deactivates and terminates) to the original caller, i.e. statement "C". Elaboration proceds then to statement "D" which will deliver a null value for this and all subsequent activations through "next". Thus, every instance of the "infix" generator function traverses the underlying tree once and only once.

It should be noted here that we are unable to implement generator functions as recursive coroutines in SIMULA. This follows from the fact that when the execution of an object is deactivated (detached) we do not have the recursion hierarchy to resume execution of the external caller. Instead execution preceeds at the next lower level within the recursion hierarchy. Thus, in SIMULA we must keep explicit track of recursion by means of a user defined stack. We will not discuss the consequences fo this "restriction"; instead we refer the interested reader to [Knuth 74].

5 - CONCLUSIONS

In this paper we have reintroduced the concept of generator functions [Newell et alii 65, Knuth 64]. We have shown their basic characteristics and also described how to implement them. By means of generator functions we are able to hide sequencing algorithms and thus allow explicit linear constructs (for statement) to access successive elements of some sequence.

Several problems must still be investigated as for instance the ability to prove the correctness of generator functions [Clint 73] and the definition of an elegant syntax.

This paper introduces no new concepts. Its merits, however, lies in the approach taken and, also, in the attempt to reestablish the utility of generator function which, with few exceptions seems to have been forgotten.

REFERENCES

BERRY, D.M.; CHIRICA, L.; JOHSTON, J.B.; MARTIN, D.F.; SORKIN,A. On the time for retention. Los Angeles, UCLA, Computer Science Department, 1973. N - 20.

CLINT, M. Program proving: coroutines. Acta Informatica, 2 (1): 50-63, 1973.

CONWAY, M.E. Design of a separable transition diagram compiler. Comm. ACM, 6 (7): 396 - 408, July 1963.

DAHL, O.J.; MYHRHAUG, B.; NYGAARD, K. Simula 67 common base language. Oslo, Norway, Norwegian Computing Centre, 1972. S - 22.

DENNIS, J.B. Modularity. In: BAUER, F.L., ed. Advanced course on software engineering. Berlin, Springer Verlag, 1973. p. 128 - 82.

GENTLEMAN, W.M. A portable coroutine system. In: INTERNATIONAL FEDERATION FOR INFORMATION PROCESSING. IFIP congress. Ljubljana, 1971, Computer Software Booklet TA-3. Amsterdam, North Holland, 1971.

JOHNSTON, J.B. The contour model of block structured processes. SIGPLAN Notices, New York, 6 (2): 55 - 82, Feb. 1971.

KNUTH, D.E. et alii. A proposal for input/output conventions in ALGOL 60. Comm. of the ACM, New York, 7 (5): 273 - 83, May 1964.

KNUTH, D.E. - Structured programming with go to statements. ACM computing surveys, New York, 6 (4): 261 - 302, dec 1974.

LISKOV, B. et alii. CLU design notes. Boston, M.I.T., Project MAC, 1974.

MYERS, G.J. Characteristics of composite design. Datamation, Barrington, 19 (9): 100-2, sept. 1973.

NEWELL, A. et alli. Information processing language V manual. 2 ed. Englewood Cliffs, N.J., Prentice Hall, 1965.

PARNAS, D.F. Information distribution aspects of programming languages. In: INTERNATIONAL FEDERATION FOR INFORMATION PROCESSING. IFIP congress. Ljubljana,1971, Computer Software Booklet TA-3. Amsterdam, North-Holland, 1971.

SMITH, D.C. et alii. ML ISP2. Stanford, Stanford University, Computer Science Department, 1973. STAN-CS-73-356.

VON STAA, A. Data transmission and modularity aspects of programming languages. Waterloo, University of Waterloo, Department of Computer Science, 1974. CS-74-17.

VON STAA, A. On the design of languages for a disciplined use of references. Rio de Janeiro, Pontifícia Universidade Católica, Departamento de Informática, 1975. MCSCA - nº 12/75.

WEIZENBAUM, J. Symetric list processor. Comm. of the ACM, New York, 6 (9): 524 - 44, Sept. 1963.