stichting
mathematisch
centrum

MC

AFDELING INFORMATICA (DEPARTMENT OF COMPUTER SCIENCE) IW 148/80 OKTOBER

H.B.M. JONKERS

DESIGNING A MACHINE INDEPENDENT STORAGE MANAGEMENT SYSTEM

Preprint

kruislaan 413 1098 SJ amsterdam

1980 Mathematics subject classification: 68B20

ACM-Conputing Reviews category: 4.12, 4.49

Designing a machine independent storage management system[*)]

by

H.B.M. Jonkers

ABSTRACT

A systematic method of designing a storage management system for a machine independent language implementation is described. It is based on constructing an abstract model, which contains exactly the information relevant to storage management and no more than that. The model allows the problem to be approached in a rigorous and transparent way, up to a level of formality where proofs of correctness are possible. The effectiveness of the method is demonstrated in the design of a storage management system for a machine independent ALGOL 68 implementation.

KEY WORDS & PHRASES: storage management, machine independence, abstract machine, abstract model, invariant

[*)] This report will be submitted for publication elsewhere.

1. INTRODUCTION

A usual way to obtain a portable implementation of a programming language L is to construct a compiler C, which translates programs in L into code for an "abstract machine" M [3]. The latter is a hypothetical machine, which is designed in such a way that it is easily implementable on a large class E of existing machines. Given the compiler C, the job of an implementer is then, apart from installing C, to implement M on his particular machine M^*. If $M^* \in E$, this involves only a minor overhead. The price to be paid for portability is thus kept to a minimum.

In each implementation of a programming language the problem of storage management must be solved. Let us consider this problem in the context of the above approach to programming language implementation. A first way to "solve" the problem is to shift it off to the implementer of the abstract machine M. This implies that the operations which have to do with storage management are kept abstract in M, much like the way they are kept abstract in the programming language L. The advantage of this approach is twofold. First, the problems of code generation and storage management are separated entirely. The designer of the code generator need not engage in the details and intricacies of a storage management system. This greatly simplifies the design of the code generator. Second, each implementer can design a storage management system of his own. Since he can tune this storage management system to his particular machine, it will probably be quite efficient. On the other hand the design and implementation of a storage management system may involve a considerable overhead in the implementation of the abstract machine M. This, of course, is in contradiction with the requirement that M should be easily implementable.

The way out is not to change the abstract machine M, but instead provide it with a standard storage management system written in a subset of the instruction code of M. Such a storage management system can be viewed as an implementation of those instructions of M, which relate to storage management, in terms of simpler instructions of M. The two advantages mentioned above are retained this way. Code generation and storage management remain separated, and each implementer is still free to design his own storage management system. If the overhead of designing and implementing a storage management system is considered to be too large, however, the standard storage management system can be used. The only remaining disadvantage is that, because the standard storage management system is machine independent, it is probably not optimally efficient on each existing machine. A careful design of the system may remove a great deal of this objection.

This paper addresses the problem of designing a (standard) storage management system as described above. It will be demonstrated by means of an example how this problem can be tackled in a systematic way. The example is not artificial. It is taken from the construction of the ALGOL 68 [11] compiler which is currently being developed at the Mathematical Centre. In this compiler the above approach is pursued. The abstract machine used in this compiler is called the "MIAM" ("Machine Independent Abstract Machine") [9]. The treatment will be such that no knowledge of either ALGOL 68 or the MIAM is required.

Let us first discuss the problem in general terms. The nature of a storage management system to be designed for an abstract machine depends to a large extent on the operations which are performed by the abstract machine. The first thing to do therefore is to investigate which requirements are imposed by the abstract machine on a storage management system and also which properties of the abstract machine can be used to make the storage management system more efficient. For any but simple abstract machines this is a complicated job. The point is that one easily gets mixed up in all kinds of details of the abstract machine, which are completely irrelevant to the storage management problem. The only way to avoid this is to bring about a "separation of concerns". That is, an _abstraction_ of the abstract machine should be made, which contains only those details of the abstract machine which are or may be relevant to the storage management problem. In such a (usually rather rudimentary) _model_ of the abstract machine the problem of storage management can be studied in isolation, which makes the problem much more transparent.

Apart from the latter, there are a number of additional advantages. First of all a storage management system designed this way is in a sense generally applicable. It cannot only be used with the abstract machine it was designed for, it can be used with any other machine that "fits" the model. So it is machine independent in a double sense. The second advantage is that it aids to a modularization of the process of compiler construction. Through the model the storage management problem can be presented to someone (e.g. a specialist in designing storage management systems), who need not know anything of the programming language or abstract machine in question. Finally it allows a nontrivial storage management system to be discussed as in this paper, without perishing in a host of implementation details.

As mentioned before the method will be demonstrated through the design of a machine independent storage management system for the abstract machine MIAM, which is used in a machine independent ALGOL 68 implementation. In the next section the model of the MIAM will be described (after which one can forget about the MIAM completely). Then the storage management problem will be formulated in section 3. Finally the design of an efficient machine independent storage management system will be described in section 4.

2. MODEL

Let us first look at the data structures which MIAM programs operate upon. Considered at the lowest level these data structures are merely pieces of storage. Here a more abstract look will be taken at them. They will be considered as abstract objects, which are called _areas_. Different areas correspond to disjoint pieces of storage. There are two kinds of areas, called _locales_ and _blocks_:

An area is either a locale or a block.

Speaking in technical terms a locale corresponds to an "activation record" and a block corresponds to a "data area". That is, if during the execution of an ALGOL 68 program the range S showed in Fig. 1 is entered, a locale L will be created in the MIAM, after which we say that "control resides in

L". At arrival at the declaration of the array A a block B (for the elements of A) will be created. We shall say that B is "created in L".

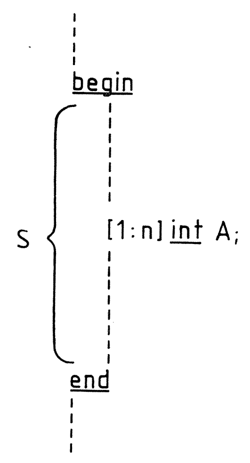

Fig. 1

Areas have a number of entities associated to them. First consider locales:

Each locale L has:

- status(L): variable status,
- type(L): constant type,
- scope(L): constant integer,
- establisher(L): constant locale.

Here the "status" of L indicates whether L is "alive" or "dead":

A status is an element from the set {alive, dead}.

Intuitively speaking a locale is alive if control resides in it or if control will ever return in it. Otherwise the locale is dead. The "type" of L is a value, the exact nature of which is completely irrelevant here. The only thing we need to know is that the (machine independent) type of L determines the (machine dependent) size of the piece of storage corresponding to L. The "scope" of L is the ALGOL 68 scope of the range corresponding to L. The latter is an integer which indicates the lifetime of the locale (the larger the scope, the shorter the locale will live). The "establisher" of L corresponds to what is usually called a "dynamic link". It is the locale where control resided immediately before control was transferred to L. For instance, if in Fig. 1 prior to entering the range S control resides in the locale M and entry of the range S results in the creation of a locale L, then establisher(L) = M.

Next consider blocks:

Each block B has:

- status(B): variable status,
- type(B): constant type,
- scope(B): variable integer,
- generator(B): variable locale.

Here the "status", "type" and "scope" of B are analogous to the corresponding entities associated to locales. The "generator" of B is the locale in which B was created. For instance, if in Fig. 1 entry of the range S and execution of the declaration of A would result in the creation of a locale L and a block B respectively, then after that generator(B) = L. The reason why the scope and generator of a block are variable and not constant entities will be discussed later.

The above covers the discussion of areas. However, areas are not the only data structures which are of interest to the storage management problem. One of the more exotic features of ALGOL 68 is the possibility to specify that certain parts of a program should be executed in parallel, where synchronization can be done through "semaphores" [2]. Programs using this feature will be called "parallel programs". The other "normal" programs will be called "sequential programs". In order to model parallellism neatly the concept of a <u>process</u> must be introduced. As opposed to areas, processes do not correspond to separate pieces of storage. They are "embedded" in locales.

Let us first discuss processes informally. In general a number of processes may simultaneously be active during the execution of a program on the MIAM, where each process has its own control. Only when executing a sequential program there is only one active process. Suppose control of the active process P resides in the locale L corresponding to the range S in Fig. 2. When control arrives at the parallel clause "par(S1, S2, S3)", which specifies that S1, S2 and S3 should be executed in parallel, three new active processes P1, P2 and P3 (corresponding to S1, S2 and S3) will be created, while P becomes inactive until P1, P2 and P3 are completed. That is, P "ramifies" over P1, P2 and P3. We shall say that P1, P2 and P3 are "created by P in L".

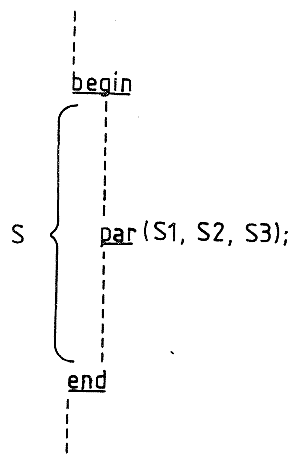

Fig. 2

The concept of a process will now be defined more precisely:

Each process P has:
- mode(P): variable mode,
- origin(P): constant locale,
- environ(P): variable locale,
- spawner(P): constant process.

Here the "mode" of P indicates whether P is "active", "spawned" (= ramified over a number of processes) or "completed":

A mode is an element from the set {active, spawned, completed}.

The "origin" of P is the locale in which P was created and the "environ" of P is the locale in which control of P currently resides. The "spawner" of P is the process which created P. For instance, in the example discussed in the previous paragraph spawner(P1) = spawner(P2) = spawner(P3) = P.

From the data structure point of view the model of the MIAM can now be regarded as a collection of four variables:

The model consists of:
- $\mathcal{L}$: variable set of locales,
- $\mathcal{B}$: variable set of blocks,
- $\mathcal{P}$: variable set of processes,
- R: variable process.

The variables $\mathcal{L}$, $\mathcal{B}$ and $\mathcal{P}$ represent the set of all locales, blocks and processes respectively which have so far been created during the execution of a program. The variable R has to do with the fact that the MIAM is a sequential machine. Only one process at a time can be executed on the MIAM, which implies that parallellism must be "serialized". The variable R indicates which (active) process is currently being executed. R will be called the "running process" and the environ of R will be called the "current environ".

Prior to the execution of a program the following holds:

Initially
- $\mathcal{L} = \{L_0\}$,
- $\mathcal{B} = \emptyset$,
- $\mathcal{P} = \{P_0\}$,
- $R = P_0$,

where L_0 is a locale such that
- status(L_0) = alive,
- type(L_0) = $\sim$,
- scope(L_0) = 0,
- establisher(L_0) = L_0,

and P_0 is a process such that
- mode(P_0) = active,
- origin(P_0) = L_0,
- environ(P_0) = L_0,
- spawner(P_0) = P_0.

The locale L_0, which will stay alive during the entire execution of a program, will be called the "initial locale". Among other things it contains the constant table. The process P_0 will be called the "initial process".

This completes the data structure part of the MIAM model. A thing one can argue about is whether the data structures described capture all information relevant to the storage management problem. An important concept that seems to be missing is that of a "reference" between areas, or more abstractly the concept of "reachability". This is an important concept because of the occurrence of "heap objects" in ALGOL 68, which correspond to areas with "infinite" lifetimes (their scope is zero). The only effective way to cope with the storage management problems caused by these objects is the use of a "garbage collector". The design of a garbage collector is a problem in its own right, which will not be discussed in this paper. Hence the concept of reachability need not be introduced in the model. Instead a garbage collection operation will be introduced as a primitive operation in the problem definition.

Let us now look at the operations performed by the MIAM. They can be modelled in terms of operations on the data structures described in the foregoing. Before doing so a few definitions will be introduced.

<u>Definition "$\leq_\Lambda$" and "$<_\Lambda$"</u>

"$\leq_\Lambda$" and "$<_\Lambda$" are relations on the set Λ of all locales, defined as follows:

$L \leq_\Lambda M \Leftrightarrow \exists\, n \geq 0\ [L = \text{establisher}^n(M)]$ $\qquad (L, M \in \Lambda)$

$L <_\Lambda M \Leftrightarrow L \leq_\Lambda M \wedge L \neq M$ $\qquad (L, M \in \Lambda)$

<u>Definition "$\leq_\Pi$" and "$<_\Pi$"</u>

"$\leq_\Pi$" and "$<_\Pi$" are relations on the set Π of all processes, defined as follows:

$P \leq_\Pi Q \Leftrightarrow \exists\, n \geq 0\ [P = \text{spawner}^n(Q)]$ $\qquad (P, Q \in \Pi)$

$P <_\Pi Q \Leftrightarrow P \leq_\Pi Q \wedge P \neq Q$ $\qquad (P, Q \in \Pi)$

Here "establisher^n(M)" and "spawner^n(Q)" denote the result of applying "establisher" and "spawner" n times to M and Q respectively. So in other words, "$\leq_\Lambda$" and "$\leq_\Pi$" are the reflexive and transitive closures of the relations "L = establisher(M)" and "P = spawner(Q)" respectively, while "$<_\Lambda$" and "$<_\Pi$" are the antireflexive contractions of "$\leq_\Lambda$" and "$\leq_\Pi$" respectively. Note that all four relations are constant. Restricted to the sets $\mathcal{L}$ and $\mathcal{P}$ they are variable, however (because $\mathcal{L}$ and $\mathcal{P}$ are variable). To illustrate these relations, consider Fig. 3 which shows a possible state of the machine. That is, it shows the locales, blocks and processes in $\mathcal{L}$, $\mathcal{B}$ and $\mathcal{P}$ respectively at a certain point of the execution of a program. In this figure among other things the following holds:

$L_0 <_\Lambda L,\ L_0 <_\Lambda E,\ \neg(L \leq_\Lambda E \vee E \leq_\Lambda L),$

$P_0 <_\Pi P,\ P_0 <_\Pi R,\ \neg(P \leq_\Pi R \vee R \leq_\Pi P).$

As can be seen from this figure the locales in $\mathcal{L}$ and the processes in $\mathcal{P}$ each constitute a tree. If L is a locale in $\mathcal{L}$, the set of all locales M in $\mathcal{L}$ with $M \leq_\Lambda L$ constitutes a list, which is usually called the "dynamic chain" emanating from L. The dynamic chain emanating from the current environ will be called the "current dynamic chain".

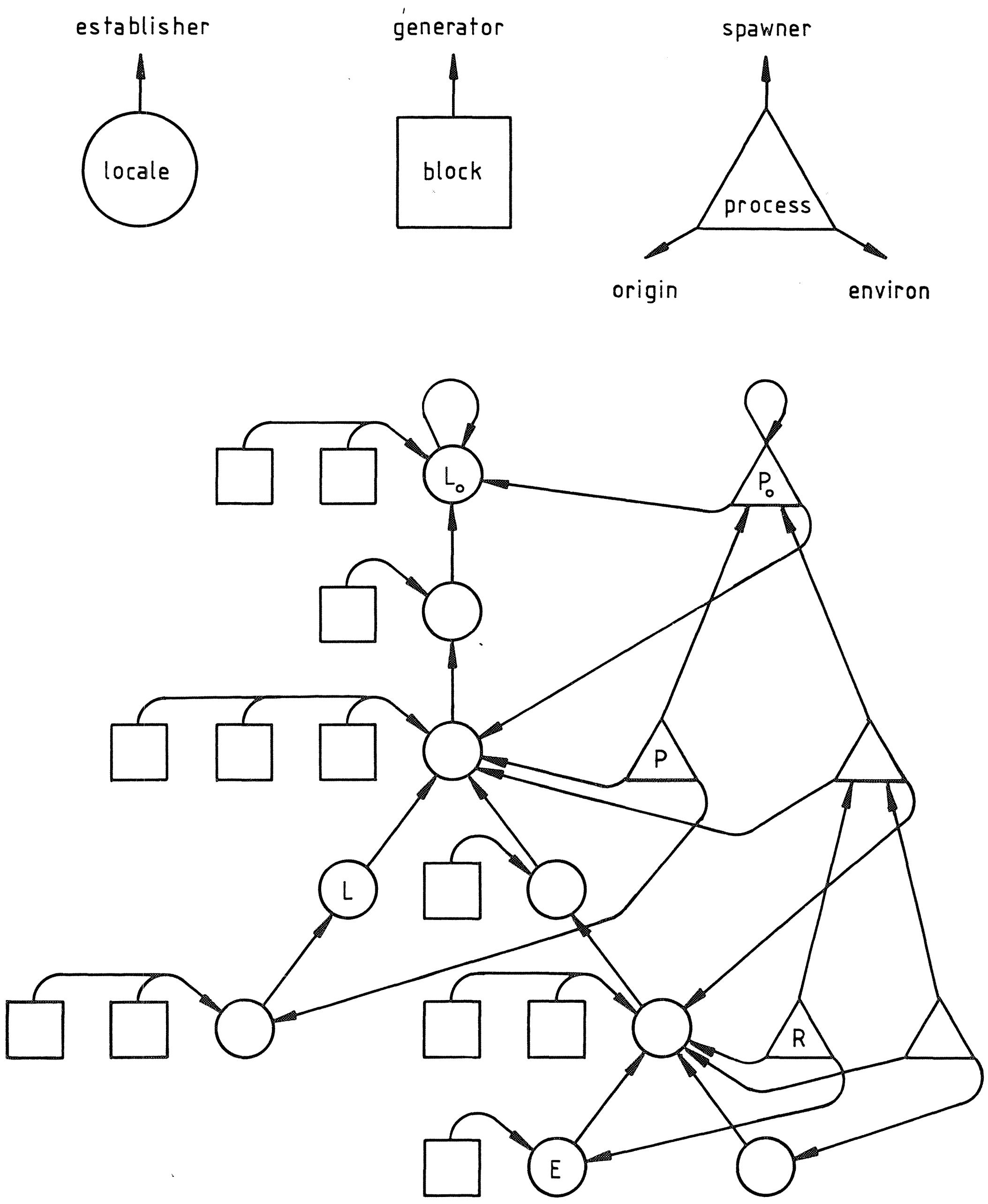
establisher
locale
generator
block
spawner
process
origin
environ
L_0
P_0
P
L
R
E

Fig. 3

The first operation which will be introduced corresponds to entering a range in ALGOL 68. It reads as follows:

```
ESTABLISH(t):
   Precondition:
      t is a type.
   Action:
      Let E = environ(R).
      Let L be a locale such that
      - $L \notin \mathcal{L}$,
      - status(L) = alive,
      - type(L) = t,
      - scope(L) = scope(E) + 1,
      - establisher(L) = E.
      $\mathcal{L} := \mathcal{L} \cup \{L\}$.
      environ(R) := L.
```

It amounts to creating a fresh, living locale L of type t. Hence $L \notin \mathcal{L}$, status(L) = alive and type(L) = t. The scope of this locale must be one larger than the scope of the current environ E (it is "newer" than E). Since control will be transferred from E to L, establisher(L) should be equal to E. L must then be added to $\mathcal{L}$ and control must be transferred to L (by making L the new current environ).

The second operation corresponds to leaving a range in ALGOL 68 (the range corresponding to the current environ):

```
FINISH:
   Precondition:
      environ(R) $\neq$ origin(R).
   Action:
      Let E = environ(R).
      environ(R) := establisher(E).
      status(E) := dead.
      For each $B \in \mathcal{B}$ with generator(B) = E
      | status(B) := dead.
```

Control of the running process R cannot be transferred beyond the locale in which R was created, which explains the precondition environ(R) $\neq$ origin(R). When leaving the range corresponding to the current environ E, control must be transferred from E to the "old" current environ. That is, environ(R) must be changed into establisher(E). This turns E into a dead area, but also all blocks created in E (the blocks B with generator(B) = E). The status of all these areas should therefore be changed into "dead".

The third operation to be discussed is concerned with the creation of blocks:

```
GENERATE(t, L):
   Precondition:
      t is a type,
      $L \in \mathcal{L}$, $L \leq_{\Lambda}$ environ(R).
   Action:
      Let B be a block such that
      - $B \notin \mathcal{B}$,
      - status(B) = alive,
      - type(B) = t,
      - scope(B) = scope(L),
      - generator(B) = L.
      $\mathcal{B} := \mathcal{B} \cup \{B\}$.
```

It describes the creation of a fresh, living block B of type t in the locale L, which should be in the current dynamic chain (the precondition L $\leq_{\Lambda}$ environ(R)). During this operation control can be thought to be temporarily transferred from the current environ to L. The block B will live as long as L, and hence the scope of B should be equal to the scope of L. Because B is created in L, the generator of B should be equal to L. The actual creation of B is accomplished by adding B to $\mathcal{B}$. A thing to be noted here is that blocks corresponding to ALGOL 68 heap objects are created in the initial locale L_0 (through "GENERATE(t, L_0)"). Consequently these blocks have scope zero.

The fourth operation is somewhat trickier than the ones met before in the sense that it does not correspond directly to any ALGOL 68 operation.

It is an operation which is concerned with efficiency. The point is that it is sometimes useful to be able to extend the lifetime of a (large) block, e.g. to prevent an expensive copy operation. A typical example is found in passing procedure values. This lifetime extension is exactly what the following operation accomplishes:

```
KEEP(B, L):
   Precondition:
      $B \in \mathcal{B}$, generator(B) = environ(R),
      $L \in \mathcal{L}$, $L \neq L_0$, $L <_\Lambda$ environ(R).
   Action:
      generator(B) := L.
      scope(B) := scope(L).
```

It extends the lifetime of the block B to that of the locale L, with the restriction that B must have been created in the current environ and L must belong to the current dynamic chain. This amounts to changing generator(B) to L. Since the scope of an area indicates its lifetime, in addition to this the scope of B must be changed to the scope of L. This explains why the scope and the generator of a block are variable.

The fifth operation corresponds to entering an ALGOL 68 parallel clause:

```
SPAWN(n):
   Precondition:
      n > 0.
   Action:
      Let $\mathcal{Q}$ be a set of n processes such that for each P $\in \mathcal{Q}$
      - P $\notin \mathcal{P}$,
      - mode(P) = active,
      - origin(P) = environ(R),
      - environ(P) = environ(R),
      - spawner(P) = R.
      $\mathcal{P} := \mathcal{P} \cup \mathcal{Q}$.
      mode(R) := spawned.
      Let P $\in \mathcal{P}$ with mode(P) = active.
      R := P.
```

Through this operation n fresh, active processes are created. Each process P in the set $\mathcal{Q}$ of these new processes is created in the current environ, with control of P initially residing in the current environ. The creator of each P is the running process R. Hence origin(P) = environ(P) = environ(R) and spawner(P) = R. The set of new processes $\mathcal{Q}$ is then added to $\mathcal{P}$. After that the running process is made to be "spawned" and an arbitrary active process P (e.g. from $\mathcal{Q}$) is made to be the new running process.

The sixth operation relates somehow to the operation SPAWN(n) as FINISH relates to ESTABLISH(t). It corresponds to leaving a constituent statement of an ALGOL 68 parallel clause:

```
COMPLETE:
   Precondition:
      environ(R) = origin(R), $R \neq P_0$.
   Action:
      mode(R) := completed.
      Let S = spawner(R).
      Let $\mathcal{Q}$ = {P ∈ $\mathcal{P}$ | spawner(P) = S}.
      If $\forall$ P ∈ $\mathcal{Q}$ [mode(P) = completed]
      | mode(S) := active.
      Let P ∈ $\mathcal{P}$ with mode(P) = active.
      R := P.
```

This operation "completes" the current running process R. The precondition is that control of R has returned in the locale in which R was created and that R is not the initial process. After having changed the mode of R to "completed", the process S which created R is determined. This is a spawned process, which should be made active if all processes created by it (all processes in the set $\mathcal{Q}$) are completed. Then, an arbitrary active process P must be selected and made to be the new running process.

The seventh operation is concerned with the situation that the running process runs into an impassable semaphore:

```
SWITCH:
   Precondition:
      $\exists$ P ∈ $\mathcal{P}$ [$P \neq R$, mode(P) = active].
   Action:
      Let P ∈ $\mathcal{P}$ with $P \neq R$ and mode(P) = active.
      R := P.
```

If the running process is halted by an impassable semaphore, R must be changed to an active process which is not. The operation SWITCH models this change of running process by selecting an arbitrary active process $P \neq R$ and assigning P to R. The precondition of SWITCH takes care that the choice of P is always well-defined. Of course, even if the precondition of SWITCH is satisfied, there may in reality not exist an active process P which is not waiting for an impassable semaphore ("deadlock"). Instead of the operation SWITCH the program is then supposed to be aborted.

The ALGOL 68 equivalent of the eighth and final operation is a jump to some global label. Its definition reveals the disruptive nature of the "goto":

```
JUMP(L, P):
   Precondition:
      $L \in \mathcal{L}$, $L \leq_\Lambda$ environ(R),
      $P \in \mathcal{P}$, $P \leq_\Pi R$,
      origin(P) $\leq_\Lambda L \leq_\Lambda$ environ(P).
   Action:
      R := P.
      mode(R) := active.
      environ(R) := L.
      For each $M \in \mathcal{L}$ with $L <_\Lambda M$
      |  status(M) := dead.
      For each $B \in \mathcal{B}$ with $L <_\Lambda$ generator(B)
      |  status(B) := dead.
      For each $Q \in \mathcal{P}$ with $P <_\Pi Q$
      |  mode(Q) := completed.
```

L is the locale corresponding to the range where the label jumped to occurs. P is the process which takes over control by jumping to the label. The fact that the label jumped to must be "visible" implies that $L \leq_\Lambda$ environ(R) and $P \leq_\Pi R$. Furthermore, L should be a locale to which control of P has access: origin(P) $\leq_\Lambda L \leq_\Lambda$ environ(P). The jump is accomplished by making P the running process, changing the mode of R (= P) to "active" and then transferring control to L. Through this jump to the locale L of process P the lifes of all locales and blocks which were created "after" L (the locales M with $L <_\Lambda M$ and blocks B with $L <_\Lambda$ generator(B)) are aborted. The status of these areas must therefore be changed into "dead". Also all processes which were started "after" P (the processes Q with $P <_\Pi Q$) are aborted, which amounts to changing their mode into "completed".

The entire model of the MIAM has now been introduced. From a storage management point of view the execution of any program on the MIAM can be modelled by a sequence of the operations described above. Not bothered by irrelevant details, the job is, to design a storage management system for this model. In doing so it should be assumed that any sequence of the above operations not violating the preconditions is allowed.

Before going deeper into the problem of storage management it is worthwhile to take a closer look at the model. The model satisfies a number of invariants, which are listed below. They can be proved by showing that they hold initially and by checking that each operation, assuming its precondition holds, does not affect them. This is a simple job, which is left to the reader.

Invariants for L_0

K1. $L_0 \in \mathcal{L}$.

K2. status(L_0) = alive.

K3. scope(L_0) = 0.

K4. establisher(L_0) = L_0.

Invariants for P_0

O1. $P_0 \in \mathcal{P}$.

O2. mode(P_0) $\neq$ completed.

O3. origin(P_0) = L_0.

O4. spawner(P_0) = P_0.

Invariants for R

R1. $R \in \mathcal{P}$.

R2. mode(R) = active.

<u>Invariants for locales L $\in$ $\mathcal{L}$</u>

```
L1. establisher(L) $\in \mathcal{L}$.
L2. $L_0 \leq_\Lambda L$.
L3. If status(L) = alive
    | status(establisher(L)) = alive.
L4. If status(L) = alive
    | There is a P $\in \mathcal{P}$ such that
    | | mode(P) = active,
    | | L $\leq_\Lambda$ environ(P).
L5. If L $\neq L_0$
    | scope(L) = scope(establisher(L)) + 1.
```

<u>Invariants for blocks B $\in$ $\mathcal{B}$</u>

```
B1. generator(B) $\in \mathcal{L}$.
B2. status(B) = status(generator(B)).
B3. scope(B) = scope(generator(B)).
```

<u>Invariants for processes P $\in$ $\mathcal{P}$</u>

```
P1. origin(P) $\in \mathcal{L}$.
P2. environ(P) $\in \mathcal{L}$.
P3. spawner(P) $\in \mathcal{P}$.
P4. $P_0 \leq_\Pi P$.
P5. origin(P) $\leq_\Lambda$ environ(P).
P6. If mode(P) = active
    | status(environ(P)) = alive.
P7. If P $\neq P_0$, mode(P) $\neq$ completed
    | mode(spawner(P)) = spawned,
    | origin(P) = environ(spawner(P)).
P8. If mode(P) = spawned
    | There is a Q $\in \mathcal{P}$ such that
    | | spawner(Q) = P,
    | | mode(Q) $\neq$ completed.
```

From these invariants can be inferred that indeed the relations "$\leq_\Lambda$" and "$\leq_\Pi$" impose a tree structure on $\mathcal{L}$ and $\mathcal{P}$ with treetops L_0 and P_0

respectively, as was indicated in Fig. 3. The set of all living locales in $\mathcal{L}$ constitutes a subtree with treetop L_0 of the tree imposed by "$\leq_\Lambda$" on $\mathcal{L}$. The leaves of this subtree are formed by the environs of the active processes. Analogously the set of all not yet completed processes in $\mathcal{P}$ constitutes a subtree with treetop P_0 of the tree imposed by "$\leq_\Pi$" on $\mathcal{P}$. The leaves of this subtree are the active processes. Notice that the invariants imply that the operation "Let $P \in \mathcal{P}$ with mode(P) = active" in COMPLETE is well-defined. Notice also that dead areas and completed processes are really "garbage" in the model: they are not referenced or used in any other way any more.

An important special case is that of sequential programs. In sequential programs the operations SPAWN(n), COMPLETE and SWITCH will not occur. It is easy to see that no processes will be created then, which amounts to the following invariant:

Invariants for sequential programs

S1. $\mathcal{P} = \{P_0\}$.

Together with invariant L4 this invariant implies that the living locales in $\mathcal{L}$ constitute a single linear list (the "dynamic chain") as indicated in Fig. 4. The locales in $\mathcal{L}$ as a whole, however, need not constitute a linear list (but a tree).

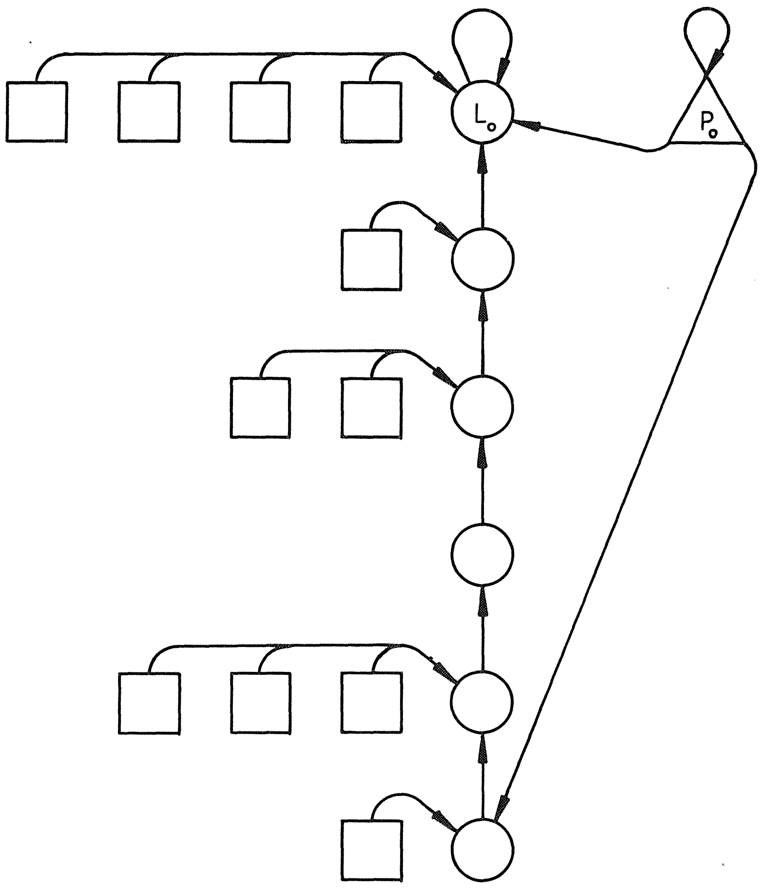

Fig. 4

3. PROBLEM

In this section the storage management problem is supposed to be defined. However, in the model as we described it there is no storage management problem. Areas which are created in the model simply fall out of the blue. The question where they come from is completely irrelevant. The storage management problem is a problem which arises only in the implementation of the abstract machine. When implementing the abstract machine on a real machine the creation of an area must be modelled by "allocating" a piece of storage to it. In contrast with the number of areas the amount of storage is limited, however. It is here where the storage management problem arises. In order to arrive at the point where the storage management problem can be formulated, we will therefore start implementing (the model of) the abstract machine described before. The method of "adding and removing variables" [6] will be used for that purpose. In a nutshell this method amounts to the following. An algorithm (or a machine) is implemented by adding extra variables and assignments to these variables to the algorithm. This creates a redundancy in the algorithm which enables certain expressions containing the "old" variables of the algorithm to be replaced by equivalent expressions containing the "new" variables. When applied in a systematic way the old variables of the algorithm can be turned into "ghost variables" this way, which may be removed from the algorithm. Thus an implementation of the algorithm in terms of the new variables is obtained. The method will be applied here by augmenting the model with an extra variable (the "allocation function"). Moreover, an abstract operation on this variable will be introduced. This operation is supposed to model or "implement" the creation of an area, which is expressed in its specification. The operation is inserted in the model at those points where areas are created. The storage management problem can then be defined as implementing this operation as efficiently as possible. In doing so a number of primitive operations on the allocation function are allowed, which may be inserted throughout the model. In particular an attempt should be made to "remove" as much abstract variables (such as e.g. the "status" of areas) from the implementation. Thus the overhead caused by the storage management system is kept to a minimum.

The first thing that needs to be done is the introduction of some model of a "store". This model should conform as closely to the store of the MIAM and the stores of existing machines as possible. We will assume the store to be a row of "cells" labelled by "addresses", which are integers 0, ..., N-1. Here N is some (large) machine dependent integer. A set of consecutive cells in the store will be called a "field" and the number of cells in a field F will be denoted as "size(F)". See Fig. 5. Though this model of a store does not cover segmented memories, it is sufficiently general to call it machine independent.

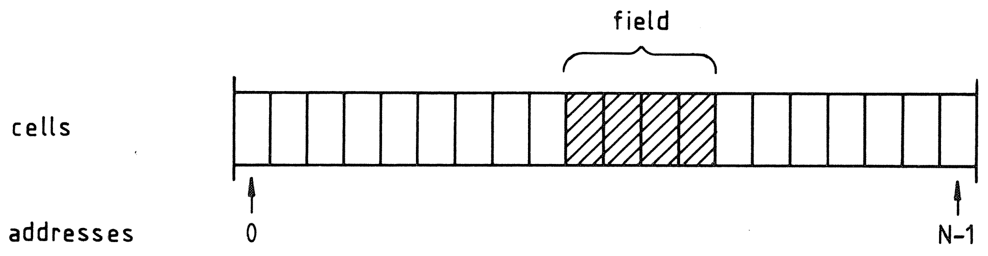

Fig.5

In an implementation of the abstract machine on a real machine the creation of an area A must be modelled by "allocating" a field in the store to it, which the area is from then on said to "occupy". This will be made more precise by introducing a new variable F, called the "allocation function", in the model:

The model is augmented with:
F: variable mapping from areas on fields.

The domain of F (which is also variable) will be denoted as "domain(F)". It contains those areas which are "located" (= occupy a field) in the store. The value of F can be changed by a number of primitive operations only, which will be discussed in the sequel. Note that the domain of F contains only locales and blocks, and no processes. Processes, as mentioned before, are "embedded" in areas. This means that the storage occupied by a process

P is part of the storage occupied by an area, to wit the origin of P.

The allocation function $\mathcal{F}$ must satisfy two obvious invariants. First of all, fields occupied by different areas may not overlap. Second, areas must occupy a field of the "proper" size. The size of the field occupied by an area will usually depend on the type of the area. The dependency need not be unique, however. It may be useful to implement certain areas of a given type different from other areas of that type. Hence areas of the same type may occupy fields of different sizes. We will therefore add an additional entity to each area A, the "size" of A, which indicates the size of the field that A should occupy. We shall assume here that the size of the field occupied by an area will not change during the execution of the program. So the size of an area is constant:

Each locale L is augmented with
- size(L): constant integer.

Each block B is augmented with
- size(B): constant integer.

The two invariants which $\mathcal{F}$ must satisfy can now be formulated as follows:

<u>Invariants for $\mathcal{F}$</u>

F1. For each area A, B $\in$ domain($\mathcal{F}$)
 | $A \neq B \Rightarrow \mathcal{F}(A) \cap \mathcal{F}(B) = \emptyset$.

F2. For each area A $\in$ domain($\mathcal{F}$)
 | size($\mathcal{F}$(A)) = size(A).

These are global invariants for $\mathcal{F}$, not to be violated by any operation on $\mathcal{F}$. In the initial situation the following should hold:

Initially
domain($\mathcal{F}$) = $\{L_0\}$,
size($\mathcal{F}(L_0)$) = size(L_0).

In other words, at the beginning of the execution of a program the initial locale should be the only area located in the store and occupy a field of the proper size (see the initial state of the model). The invariants for $\mathcal{F}$ are thus trivially satisfied in the initial situation.

The next thing to do is to introduce an abstract operation on $\mathcal{F}$, which models the creation of an area. It should allocate a field in the store to a (new) area A and will be denoted as "ALLOCATE(A)". It should do so, however, by means of the primitive operations (to be) defined on $\mathcal{F}$ exclusively. This is specified below:

ALLOCATE(A):

  Precondition:

    A is an area, $A \notin \mathcal{L} \cup \mathcal{B}$.

  Action:

    Establish the truth of the assertion $A \in \text{domain}(\mathcal{F})$ by means of the primitive operations defined on $\mathcal{F}$.

The operation ALLOCATE(A) should be inserted at those points in the model where areas are created. It should therefore be added to the operations ESTABLISH(t) and GENERATE(t, L). At the same time this gives us an opportunity to associate the proper size to an area being created:

```
ESTABLISH(t):
   Precondition:
      t is a type.
   Action:
      Let E = environ(R).
      Let L be a locale such that
      - $L \notin \mathcal{L}$,
      - status(L) = alive,
      - type(L) = t,
      - scope(L) = scope(E) + 1,
      - establisher(L) = E,
      - size(L) = $\sim$.
      ALLOCATE(L).
      $\mathcal{L} := \mathcal{L} \cup \{L\}$.
      environ(R) := L.

GENERATE(t, L):
   Precondition:
      t is a type,
      $L \in \mathcal{L}$, $L \leq_{\Lambda}$ environ(R).
   Action:
      Let B be a block such that
      - $B \notin \mathcal{B}$,
      - status(B) = alive,
      - type(B) = t,
      - scope(B) = scope(L),
      - generator(B) = L,
      - size(B) = $\sim$.
      ALLOCATE(B).
      $\mathcal{B} := \mathcal{B} \cup \{B\}$.
```

Here "$\sim$" is some implementation dependent integer, which depends on the type t.

Before formulating the problem there remains only one thing to be discussed: the set of primitive operations allowed on $\mathcal{F}$. We shall discuss these operations by investigating how ALLOCATE(A) can be implemented. The effect of ALLOCATE(A) should be that A is added to the domain of $\mathcal{F}$. So the first operation we need is an operation to extend the domain of $\mathcal{F}$ with an area. Due to the invariants for $\mathcal{F}$ and the finiteness of the store this may be impossible, however. First of all, it may be impossible to find a field F of "free cells" (= cells not occupied by areas) such that size(F) = size(A), even though the total number of free cells is more than sufficient. This is due to a phenomenon known as "fragmentation". Second, the total number of free cells may simply be insufficient ("storage overflow").

The first problem (fragmentation) can be coped with by introducing an operation to "move" areas in the store from one field to an other, i.e. change the value of $\mathcal{F}$(A) for certain A $\in$ domain($\mathcal{F}$). Thus small fields of free cells can be united into larger fields. A thing to be borne in mind with this is that in practice moving areas is an expensive operation, because all "pointers" to or into a moved area must be "updated". The second problem (storage overflow) can only be dealt with by allowing areas to be "deallocated" too, i.e. to be removed from the domain of $\mathcal{F}$. Of course only areas which are no longer used by the program should be deallocated.

What are "no longer used" areas? One thing we know for sure is that dead areas are not used any more. So dead areas can be deallocated with impunity. Yet even the deallocation of all dead areas may not help. The only escape is then to deallocate no longer used living areas too. The latter areas are considerably harder to detect than dead areas, however. The use of a "garbage collector" is required for that. The design of a garbage collector will not be discussed in this paper (but see [7]). Hence an unspecified primitive operation "COLLECT GARBAGE" on $\mathcal{F}$ is introduced. This operation is supposed to deallocate all no longer used areas (including all dead areas), while it may also move areas. It is a very expensive operation which should only be used as a last resort. As far as certain properties of COLLECT GARBAGE are important or even essential to the storage management system to be designed, these properties will be postulated in the form of "Requirements for COLLECT GARBAGE". If even a garbage collection does not help, the only way out is to abort the program.

The above accounts for the following list of primitive operations allowed on F:

<u>Primitive operations on F,</u>

1. Adding an area to domain(F).
2. Changing the value of F(A) for a number of A $\in$ domain(F).
3. Removing a number of A $\in$ domain(F) with status(A) = dead from domain(F).
4. COLLECT GARBAGE.

In all this it is implicitly assumed that the operations do not violate the Invariants for F.

The storage management problem now boils down to:

<u>Problem</u>

Implement ALLOCATE(A) efficiently.

The word "implement" must be taken in a broad sense here. This implies not only that ALLOCATE(A) must be expressed in terms of the primitive operations on F, but also that operations on F may be inserted anywhere in the model in order to make the implementation more efficient. The collection of all operations on F thus added to the model constitutes the "storage management system".

We require that efficiency of the storage management system to be designed should primarily be achieved for sequential programs. The rationale behind this is that ALGOL 68 was not specifically designed as a language for writing parallel programs. The majority of programs written in ALGOL 68 will be purely sequential. Hence it is reasonable if the use of parallellism costs a little extra.

The design of an efficient storage management system will be started in the next section.

4. DESIGN

A general approach to the design of a storage management system is to divide the areas in a number of classes dependent upon certain properties. For each class a special storage management strategy is used, which exploits the properties of the areas in that class. Let us assume n classes $\mathcal{C}_1, \ldots, \mathcal{C}_n$ of areas are distinguished. Then the allocation function $\mathcal{F}$ can correspondingly be written as $\mathcal{F} = \mathcal{F}_1 \cup \ldots \cup \mathcal{F}_n$, where domain($\mathcal{F}_i$) $\subset \mathcal{C}_i$ (i = 1, ..., n). Let us call the set of all cells occupied by the areas in domain($\mathcal{F}_i$) the "region" of $\mathcal{F}_i$. The job is to implement the operation "ALLOCATE(A)" efficiently in terms of operations on the $\mathcal{F}_i$. These operations may freely be chosen from the set of primitive operations defined on $\mathcal{F}$. If the operations are applied arbitrarily, however, a comprehensive bookkeeping is necessary in order to ensure the Invariants for the allocation function are satisfied. This bookkeeping can be simplified greatly if the regions of the $\mathcal{F}_i$ are kept "compact" (= constituting a field). In that case only operations may be performed on the $\mathcal{F}_i$, which do not disturb the compactness of the regions.

We shall comply with the above by abstractly modelling each $\mathcal{F}_i$ as a "pile" $\mathcal{U}_i$. A pile is a stack of areas which (apart from "push" and "pop") has a number of additional operations defined on it (to be discussed later). If a pile $\mathcal{U}$ contains the areas $A_1, \ldots, A_m$ in the order from bottom to top, this will be denoted as $\mathcal{U} = \langle A_1, \ldots, A_m \rangle$. A pile $\mathcal{U} = \langle A_1, \ldots, A_m \rangle$ can be "located" in the store in two different ways as indicated in Fig. 6. Here the areas A_i occupy contiguous fields of size(A_i) cells.

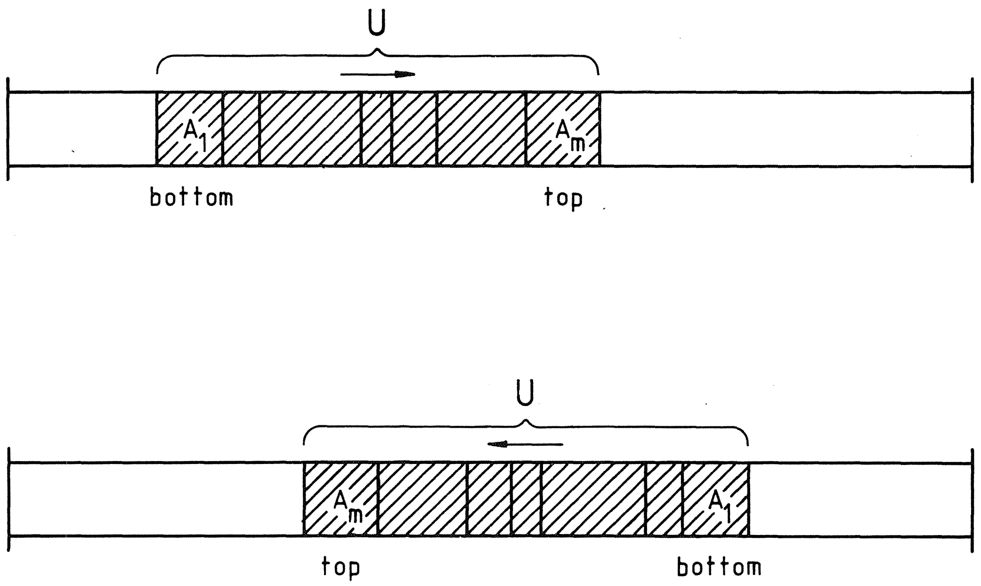

Fig. 6

It is useful to dwell briefly on what we did in the above. We represented the allocation function F as a (yet to be fixed) number of piles U_1, ..., U_n. On the one hand this can be viewed as a matter of abstraction: we abstracted from the store. This has the advantage that it makes life a lot easier. We do not need to talk in such "low level" terms as "cells", "addresses", "fields", etc. any more. A minor drawback is the fact that everything we said about F must now be translated in terms of the piles U_1, ..., U_n. Since the correspondence between F and U_1, ..., U_n is obvious, this will be omitted. Note that F can only be reconstructed from the U_1, ..., U_n after locating the latter in the store. On the other hand the things we did in the above can be viewed as a matter of concretion (the inverse of abstraction): we made a certain choice as to the structure of the allocation function. This was a design decision in order to reduce the problems caused by the Invariants for the allocation function. It also reduces the freedom of design, of course.

Up to two piles can efficiently be accommodated in the store (in the case of two piles: one at each end of the store). Though storage management systems with a larger number of piles are certainly conceivable, we will therefore limit the number of piles to two. The following are plausible choices:

1. One pile for all areas.
2. Two piles, for locales and blocks.
3. Two piles, for areas with scope > 0 and for areas with scope = 0.

The first choice does not exploit the different properties of areas. Hence it may not be expected to result in an efficient storage management system. The second choice exploits the differences between locales and blocks. This may lead to an efficient storage management scheme for locales (in the absence of parallellism locales have nested lifetimes), but for blocks (which may occupy the majority of the storage) it is just as bad as the first choice. The third choice seems the most appropriate here. It closely (but not entirely) fits in with the difference between ALGOL 68 stack and heap objects. This alternative will therefore be chosen.

The above implies that we have two piles *S* and *H* in our storage management system. *S* contains the areas with scope > 0 and *H* those with scope = 0. We assume they are located in the store as indicated in Fig. 7.

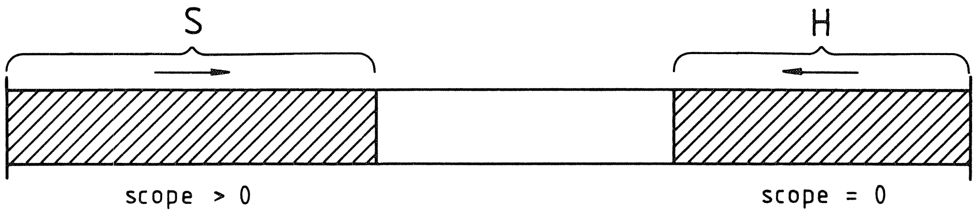

Fig.7

As with the allocation function *F* the piles *S* and *H* must satisfy a number of invariants. First, the fact that *S* and *H* correspond to (the domain of) a mapping (*F*) implies that no area may occur twice in *S* and *H*. Second, the Invariants for *F* must be translated into invariants for *S* and *H*. Invariant F1 amounts to the fact that the sum of the sizes of the areas in *S* and *H* must be less or equal to the size N of the store. Invariant F2 need nor can be expressed any more. (This invariant is incorporated in the correspondence between F and the piles S and H.) Third, S and H should contain only areas with scope > 0 and scope = 0 respectively. So we have:

<u>Invariants for S and H</u>

U1. If $S = \langle A_1, \ldots, A_m \rangle$,

$H = \langle A_{m+1}, \ldots, A_n \rangle$

| $i \neq j \Rightarrow A_i \neq A_j$ $\quad (i, j = 1, \ldots, n)$.

U2. $\sum_{A \in S \cup H} \text{size}(A) \leq N$.

U3. For each $A \in S$

| scope(A) > 0.

U4. For each $A \in H$

| scope(A) = 0.

For notational convenience the piles S and H are considered here occasionally as the sets of their elements. The translation of the initial situation for F into the initial situation for S and H leads to:

Initially

$S = \emptyset$,

$H = \langle L_0 \rangle$.

In this situation the Invariants for S and H are trivially satisfied.

During the further design of the storage management system care must be taken that Invariants U1 through U4 are not violated. These invariants could be violated in two ways. First of all the operations of the abstract machine might violate Invariants U3 and U4. Invariants U3 and U4 use the scope of areas, which is variable for blocks. However, the only operation that may affect the scope of an area is KEEP(B, L) and this operation will never change the scope of an area from > 0 into = 0 or vice versa (use Invariants L2 and L5 and the fact that $L \neq L_0$). The Invariants for S and H can therefore never be violated by any operation of the abstract machine. The second way the invariants could be violated is because of some operation on S or H that we insert in the model. It should be checked in each individual case that such an operation does not violate the Invariants for S and H.

An operation that could particularly violate the Invariants for S and H is COLLECT GARBAGE. The informal "definition" of COLLECT GARBAGE states that it removes all no longer used areas (including all dead areas) from the domain of the allocation function F. Speaking in terms of the piles S and H this implies that COLLECT GARBAGE removes all no longer used areas from $S \cup H$. In this process the remaining areas in S and H could in principle be shuffled arbitrarily. They could even be transferred from S to H or vice versa (thus violating Invariant U3 or U4). This will be prevented by the following requirements:

<u>Requirements for COLLECT GARBAGE</u>

1. No areas are added to S.
2. No areas are added to H.

It is easy to see that these two requirements are sufficient to let COLLECT GARBAGE "respect" the Invariants for S and H. Apart from these two requirements a third will be imposed which is not strictly necessary:

<u>Requirements for COLLECT GARBAGE</u>

3. The order of the remaining areas in S and H is not affected.

It says that the garbage collector must be "genetic order preserving", which is a desirable property of garbage collectors [10]. Why this is so will turn out soon. Notice that the removal of a number of areas from S and H may affect the compactness of the regions of S and H. Consequently the garbage collector must perform a "compaction" in order to restore the situation of Fig. 7. This need not be expressed in the Requirements for COLLECT GARBAGE because COLLECT GARBAGE is considered as an operation on the "abstract" piles S and H here. It follows directly from the correspondence between S and H and the allocation function F.

Let us now attempt to design a first storage management system.

4.1 The initial system

The obvious way to obtain a usable storage management system is as follows. Storage can only be allocated to an area A if there is enough room between the piles $\mathcal{S}$ and $\mathcal{H}$ in the store. The room between $\mathcal{S}$ and $\mathcal{H}$ (measured in cells) will be denoted as "FREE":

$$\text{FREE} = \text{N} - \sum_{\text{A} \in \mathcal{S} \cup \mathcal{H}} \text{size(A)}$$

If FREE < size(A) there is not enough room and a garbage collection is used to make room. If after a garbage collection there is still not enough room, the program is aborted. Otherwise storage can be allocated on $\mathcal{S}$ or $\mathcal{H}$ (using a "push" operation), dependent on the scope of A. This leads to:

```
System 1
ALLOCATE(A):
   Let s = size(A).
   If FREE < s
     | COLLECT GARBAGE.
     | If FREE < s
     |   | ABORT.
   Case
   1. scope(A) > 0
     | $\text{PUSH}_{\mathcal{S}}$(A).
   2. scope(A) = 0
     | $\text{PUSH}_{\mathcal{H}}$(A).
```

Notice that all operations performed on $\mathcal{S}$ and $\mathcal{H}$ correspond to legal (primitive) operations on the allocation function $\mathcal{F}$. Notice also that the Invariants for $\mathcal{S}$ and $\mathcal{H}$ are not violated.

The above storage management system is not very satisfactory for a number of reasons. One of them is the following. Suppose during the execution of a program the situation is reached that a garbage collection delivers only a small amount of free storage (just about sufficient to proceed). Then it will probably be necessary to perform a garbage collection very soon again, which may once more deliver only a small amount of free storage, etc.. Since a garbage collection is a time consuming operation, this may lead to the situation that the majority of the execution time of a program is spent collecting garbage before the program is finally aborted. This will be remedied in the next subsection.

4.2 Avoiding frequent garbage collections

The problem of frequent garbage collections can be solved by requiring that the garbage collector delivers a minimum number of free cells, which will be denoted as "minfree". This number should be large enough to let the program proceed undisturbedly for some time after a garbage collection. Thus we obtain:

System 2

```
ALLOCATE(A):
   Let s = size(A).
   If FREE < s
   | COLLECT GARBAGE.
   | If FREE < max(s, minfree)
   | | ABORT.
   Case
   1. scope(A) > 0
   | PUSH$_S$(A).
   2. scope(A) = 0
   | PUSH$_H$(A).
```

This removes one objection to System 1. There is another severe objection to both Systems 1 and 2, however. For the deallocation of areas both systems rely entirely on garbage collection, which does not make them very efficient. We will do something about that below.

4.3 Restraining the use of the garbage collector

Checking the list of primitive operations defined on the allocation function $\mathcal{F}$ we see that the only way to deallocate areas other than through a garbage collection, is the deallocation of dead areas. Dead areas may freely be removed from the piles $\mathcal{S}$ and $\mathcal{H}$. As far as the pile $\mathcal{H}$ is concerned this does not bring us any further, because in $\mathcal{H}$ no dead areas occur (this follows from Invariants U4, K2, L2, L5, B2, B3 and the fact that $\mathcal{H} \subset \mathcal{L} \cup \mathcal{B}$). So all dead areas in $\mathcal{S} \cup \mathcal{H}$ occur in $\mathcal{S}$. It would not be very wise, however, to allow dead areas to be deallocated arbitrarily inside $\mathcal{S}$, because that would require an expensive "compaction" in order to restore the compactness of the region of $\mathcal{S}$. Dead areas can be popped from the top of $\mathcal{S}$ with impunity, however. This gives us a cheap mechanism to deallocate areas over the head of the garbage collector.

The question is where in the model the operation to pop dead areas from $\mathcal{S}$ should be inserted. The most natural places to do so seem to be those places where areas are "killed". If a killed area happens to reside at the top of $\mathcal{S}$, A and all dead areas "below" it can immediately be popped from $\mathcal{S}$. An operation "RELEASE", which does just that, will therefore be introduced. It will be inserted in the operations FINISH and JUMP(L, P), which are the only machine operations that kill areas:

```
FINISH:
   Precondition:
      environ(R) $\neq$ origin(R).
   Action:
      Let E = environ(R).
      environ(R) := establisher(E).
      status(E) := dead.
      For each B $\in \mathcal{B}$ with generator(B) = E
      |  status(B) := dead.
      RELEASE.
```

```
JUMP(L, P):
   Precondition:
      $L \in \mathcal{L}$, $L \leq_{\Lambda}$ environ(R),
      $P \in \mathcal{P}$, $P \leq_{\Pi} R$,
      origin(P) $\leq_{\Lambda} L \leq_{\Lambda}$ environ(P).
   Action:
      R := P.
      mode(R) := active.
      environ(R) := L.
      For each $M \in \mathcal{L}$ with $L <_{\Lambda} M$
      | status(M) := dead.
      For each $B \in \mathcal{B}$ with $L <_{\Lambda}$ generator(B)
      | status(B) := dead.
      For each $Q \in \mathcal{P}$ with $P <_{\Pi} Q$
      | mode(Q) := completed.
      RELEASE.
```

Notice that at all places where ALLOCATE(A) and RELEASE occur all system invariants hold (the invariants need only hold between two machine operations).

Storage management system 3 now looks as follows:

<u>System 3</u>

```
ALLOCATE(A):
   Let s = size(A).
   If FREE < s
   | COLLECT GARBAGE.
   | If FREE < max(s, minfree)
   | | ABORT.
   Case
   1. scope(A) > 0
   | $PUSH_S$(A).
   2. scope(A) = 0
   | $PUSH_H$(A).

RELEASE:
   While DEADTOP
   | $POP_S$.
```

The predicate "DEADTOP" in this system is defined as follows:

DEADTOP = $S \neq \emptyset$ and status(top(S)) = dead.

Here the "and" is used as a "McCarthy operator" and "top(S)" is the area at the top of S. If all areas which appear in S have nested lifetimes, this scheme will keep S free from dead areas. It may therefore be expected to work rather efficiently for say ALGOL 60 type ALGOL 68 programs. The only operations which may (temporarily) impede the effectiveness of this scheme are GENERATE(t, L), where $L \neq L_0$ and $L \neq$ environ(R), KEEP(B, L) and SWITCH. The latter will occur in parallel programs only, while the other two may be expected not to be used too frequently (by a good code generator). Whatever operations are performed, however, the above scheme will always work correctly. Notice that Requirement 3 for COLLECT GARBAGE is essential to the effectiveness of the scheme.

Though System 3 is a major improvement over System 2, it still rather heavily depends on garbage collection as a deallocation tool (especially in parallel programs). The role of the garbage collector can further be diminished, as we will demonstrate.

4.4 Restraining the use of the garbage collector further

Suppose in ALLOCATE(A) we run out of storage (i.e. FREE < s). It may very well turn out (especially if few areas with scope = 0 are used) that the number of dead cells in S is large compared with the size of the store. The number of dead cells in S will be denoted as "DEAD":

$$\text{DEAD} = \sum_{\substack{A \in S \\ \text{status}(A) = \text{dead}}} \text{size}(A).$$

It is profitable then not to perform a full garbage collection, but simply to remove all dead areas from S (which implies compacting the region of S). An operation "COMPACT$_S$" which accomplishes this will therefore be introduced:

COMPACT$_S$:

Remove all A $\in S$ with status(A) = dead from S while preserving the order of the remaining areas in S.

Notice that the operation on the allocation function F corresponding to COMPACT$_S$ is expressible in the primitive operations defined on F. Notice also that COMPACT$_S$ does not violate the Invariants for S and H and that it is "genetic order preserving".

The operation COMPACT$_S$ is considerably cheaper than COLLECT GARBAGE. The reason is that an expensive "marking phase", such as in the garbage collector, is not necessary in COMPACT$_S$. Moreover, the compaction (as opposed to a garbage collection) is strictly local to the pile S: Due to the "scope rules" of ALGOL 68 the fact that area A contains a pointer to area B implies that scope(A) $\geq$ scope(B). Consequently areas in H do not contain pointers to areas in S, which implies that areas in S may be moved without having to update any pointers in areas in H.

If "mindead" denotes the (possibly dynamically determined) minimum number of dead cells in $\mathcal{S}$ for which a compaction is more profitable than a garbage collection, then the new storage management system looks as follows:

```
System 4
ALLOCATE(A):
   Let s = size(A).
   If FREE < s
   |  If DEAD ≥ max(s, mindead)
   |  |  COMPACT_S.
   |  else
   |  |  COLLECT GARBAGE.
   |  |  If FREE < max(s, minfree)
   |  |  |  ABORT.
   Case
   1. scope(A) > 0
   |  PUSH_S(A).
   2. scope(A) = 0
   |  PUSH_H(A).

RELEASE:
   While DEADTOP
   |  POP_S.
```

The number DEAD in this system can be determined by traversing $\mathcal{S}$ once. In traversing $\mathcal{S}$ it must be determined for each area A $\in \mathcal{S}$ whether A is dead or not. The assumption in all this is, as it is in COMPACT$_{\mathcal{S}}$ and DEADTOP, that in a real implementation it is possible to determine the status of an area in $\mathcal{S}$. What are the consequences of this assumption?

Areas as we described them have a number of entities associated to them (such as "status", "type", "scope", etc.). Except for the "size" these are abstract entities which are used in the definition of the abstract machine. Each implementer of the abstract machine will try to implement these entities as efficiently as possible, and if possible he will even avoid to implement certain entities. A number of the entities must be implemented anyway: the type and size of an area (for the garbage collector and the compaction routine), the scope of an area (for scope checks) and the establisher of a locale (in order to return to the proper locale after leaving a range). Other than for reasons of storage management the status of an area and the generator of a block need not be implemented.

In System 4, however, the status of an area is apparently supposed to be implemented. For locales this could be done by letting FINISH and JUMP(L, P), which are the only two operations that kill areas, mark dead locales as such. For blocks this is not so simple. The best way to determine whether a block B is dead seems to use Invariant B2 and check whether generator(B) is marked as dead or not. Yet this implies that the generator of a block must also be implemented. This overhead deprives System 4 of some of its attractiveness. It would be nice if the overhead could be eliminated, and indeed for sequential programs it can. We can use the redundancy caused by the introduction of the allocation function (in the shape of the piles S and H) to turn the status of an area and the generator of a block into "redundant variables" of the storage management system. This will be shown and proved in the next subsection. After that we will consider the general case of both sequential and parallel programs.

Before continuing two more requirements on the garbage collector will be imposed. From the requirements introduced so far absolutely nothing can be inferred as to which areas are or are not deallocated by the garbage collector. There are certain areas from which it is easy to see that they are (or should be) or are not (or should not be) deallocated by the garbage collector. In particular all dead areas will be deallocated by the garbage collector. (This was already stated informally.) Furthermore, the living locales will not be deallocated in a garbage collection. (They are "reachable" because control will, or should be able to ever return to them.) The following additional requirements, which allow us to use that information, will therefore be imposed on the garbage collector:

Requirements for COLLECT GARBAGE

4. All dead areas are deallocated.
5. No living locales are deallocated.

A number of additional invariants (which hold between two operations of the abstract machine) can now be proved for System 4. In order to formulate them more easily the following relation on the set of areas in $\mathcal{S}$ will be introduced:

<u>Definition "$<_{\mathcal{S}}$"</u>

"$<_{\mathcal{S}}$" is a relation on the set of areas in $\mathcal{S}$, defined as follows:

If $\mathcal{S} = \langle A_1, \ldots, A_n \rangle$

| $A_i <_{\mathcal{S}} A_j \Leftrightarrow i < j$ $\quad$ $(i, j = 1, \ldots, n)$

Due to Invariant U1 this relation is well-defined. The fact that A $<_{\mathcal{S}}$ B implies that A is "below" B in $\mathcal{S}$. The following invariants hold:

<u>Invariants for $\mathcal{S}$ and $\mathcal{H}$</u>

U5. $\mathcal{S} \cup \mathcal{H} \subset \mathcal{L} \cup \mathcal{B}$.

U6. For each L $\in \mathcal{L}$ with status(L) = alive

| L $\in \mathcal{S} \cup \mathcal{H}$.

U7. For each L $\in \mathcal{S} \cap \mathcal{L}$ with establisher(L) $\neq L_0$

| establisher(L) $\in \mathcal{S}$,

| establisher(L) $<_{\mathcal{S}}$ L.

U8. For each B $\in \mathcal{S} \cap \mathcal{B}$

| generator(B) $\in \mathcal{S}$,

| generator(B) $<_{\mathcal{S}}$ B.

U9. If $\mathcal{S} \neq \emptyset$

| status(top($\mathcal{S}$)) = alive.

Invariant U5 is based on Requirements 1 and 2 for COLLECT GARBAGE. It allows us to use all Invariants for $\mathcal{L}$ and $\mathcal{B}$ for areas in $\mathcal{S}$ and $\mathcal{H}$. Invariant U6 is based on Requirement 5 for COLLECT GARBAGE. Notice that it implies that $L_0 \in \mathcal{H}$ and L $\in \mathcal{S}$ for each L $\in \mathcal{L}$ with L $\neq L_0$ and status(L) = alive. Invariants U7 and U8 are based on Requirements 1, 3 and 4 for COLLECT GARBAGE. The informal argument for their truth is simple. The establisher of a locale L is created before the locale itself. Hence establisher(L) will occur below L in $\mathcal{S}$. The same applies to the generator of a block B and the block itself. The only operation that might violate the relation generator(B) $<_S$ B is KEEP(B, L). However, prior to KEEP(B, L) the following holds for the new generator L of B: $L_0 \neq L <_\Lambda$ environ(R) = generator(B). With Invariant U7 this implies that L $<_S$ generator(B) $<_S$ B. Finally, Invariant U9 is based on Requirements 1 and 4 for COLLECT GARBAGE and the fact that dead areas are immediately popped from $\mathcal{S}$. The (simple) formal proof of Invariants U5 through U9 is left to the reader.

4.5 Removing overhead in the sequential case

In this subsection we shall assume that only sequential programs are executed on the abstract machine. So the operations SPAWN(n), COMPLETE and SWITCH will not occur and Invariant S1 will hold, i.e. $\mathcal{P} = \{P_0\}$. The living locales in $\mathcal{L}$ then constitute a single dynamic chain, which emanates from environ(P_0) (see Fig. 4). Together with the Invariants for $\mathcal{S}$ and $\mathcal{H}$ this implies that the store looks as in Fig. 8. In this figure the circles represent the locales in the dynamic chain. Notice that if $\mathcal{S} \neq \emptyset$ there is always a living locale at the bottom of $\mathcal{S}$, which amounts to the following invariant:

Invariants for sequential programs

S2. If $\mathcal{S} = <A_1, \ldots, A_n>$ with $n > 0$
  | $A_1 \in \mathcal{L}$ and status(A_1) = alive.

This invariant cannot be derived from the invariants formulated so far, but must be proved independently. It critically depends on the fact that dead areas are popped from $\mathcal{S}$ as soon as they occur on the top of $\mathcal{S}$.

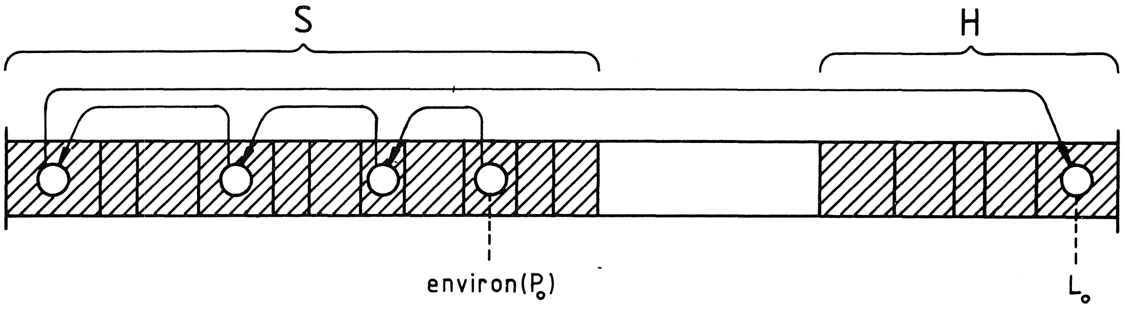

Fig. 8

The locales indicated in Fig. 8 are all alive. But what can we say about the "liveliness" of the other areas in $\mathcal{S}$? We know there is a relation between the scope of an area and its lifetime. This relation is somewhat obscured by the operations GENERATE(t, L) and KEEP(B, L). Can the scope of an area be used anyway in order to determine whether the area is dead or not? In order to answer this question the genetic order relation "$<_{\mathcal{S}}$" must be examined more closely.

Consider a living locale L in $\mathcal{S}$ and an other locale M "above" L in $\mathcal{S}$, i.e. L $<_{\mathcal{S}}$ M. At the moment M was created L was already in $\mathcal{S}$ and alive. So just after the creation of M both L and M belonged to the dynamic chain of which M was the beginning. This implies that at that moment L $<_{\Lambda}$ M. Yet, since the relation "$<_{\Lambda}$" is constant, the assertion L $<_{\Lambda}$ M will hold forever. This amounts to the following invariant:

<u>Invariants for sequential programs</u>

S3. For each $L \in \mathcal{S} \cap \mathcal{L}$ with status(L) = alive
and each $M \in \mathcal{S} \cap \mathcal{L}$
| $L <_{\mathcal{S}} M \Rightarrow L <_{\Lambda} M$.

Next consider a living locale L in $\mathcal{S}$ and a dead block B above L in $\mathcal{S}$. Let G = generator(B) and suppose that G $<_{\mathcal{S}}$ L. From Invariant B2 we know that G is dead. At the moment L was created G was already in $\mathcal{S}$ and also dead (otherwise Invariants S3 and L3 would lead to a contradiction). Since B was created after L this implies that B was created when G was already dead.

From this and Invariant B2 can be concluded that at the moment B was created apparently generator(B) $\neq$ G. Consequently the operation KEEP(B, G) must have been applied some time thereafter. The precondition of KEEP(B, G) says that G $<_\Lambda$ environ(R), which implies that status(G) = alive, however. From this contradiction can be concluded that the assertion G $<_S$ L can never hold. Since G $\neq$ L this leads to the conclusion that L $<_S$ G, which is expressed in the following invariant:

<u>Invariants for sequential programs</u>

S4. For each L $\in \mathcal{S} \cap \mathcal{L}$ with status(L) = alive
and each B $\in \mathcal{S} \cap \mathcal{B}$ with status(B) = dead
| L $<_S$ B $\Rightarrow$ L $<_S$ generator(B).

A more formal proof of the above invariants is left to the mistrustful reader.

Invariants S2, S3 and S4 give us additional information on the relation "$<_S$" which can be used profitably. Before showing this a definition is introduced. For each area A in $\mathcal{S}$ the "base" of A is defined to be the first living locale equal to or below A in $\mathcal{S}$:

<u>Definition base(A)</u>

For each A $\in \mathcal{S}$
| base(A) = $\max_{<_S}$ {L $\in \mathcal{S} \cap \mathcal{L}$ | L $\leq_S$ A, status(L) = alive}.

Notice that because of Invariant S2 the base of an area in $\mathcal{S}$ is always well-defined. The following invariant can now be derived from Invariants S3 and S4:

<u>Invariants for sequential programs</u>

S5. For each A $\in \mathcal{S}$
| status(A) = dead $\Leftrightarrow$ scope(A) > scope(base(A)).

Proof:

Let A $\in \mathcal{S}$ and let L = base(A). If A = L the proof is trivial. If A $\neq$ L, and hence L $<_S$ A, a number of cases must be distinguished. This is done schematically below.

```
A $\in \mathcal{L} \cup \mathcal{B}$.                                  (Inv. U5)
If status(A) = dead
    If A $\in \mathcal{L}$
        L $<_\Lambda$ A.                                       (Inv. S3)
        scope(A) > scope(L).                             (Inv. L5)
    If A $\in \mathcal{B}$
        L $<_S$ generator(A).                              (Inv. S4)
        L $<_\Lambda$ generator(A).                              (Inv. S3)
        scope(generator(A)) > scope(L).                  (Inv. L5)
        scope(A) > scope(L).                             (Inv. B3)
    scope(A) > scope(L).
If scope(A) > scope(L)
    If A $\in \mathcal{L}$
        If status(A) = alive
            L = A.                                     (Def. base(A))
            Contradiction.                               (L $<_S$ A)
        status(A) = dead.
    If A $\in \mathcal{B}$
        If status(A) = alive
            status(generator(A)) = alive.                (Inv. B2)
            generator(A) $<_S$ A.                          (Inv. U8)
            generator(A) $\leq_S$ L.                     (Def. base(A))
            generator(A) $\leq_\Lambda$ L.                          (Inv. S3)
            scope(generator(A)) $\leq$ scope(L).             (Inv. L5)
            scope(A) $\leq$ scope(L).                        (Inv. B3)
            Contradiction.                     (scope(A) > scope(L))
        status(A) = dead.
    status(A) = dead.
status(A) = dead $\Leftrightarrow$ scope(A) > scope(L).
```

QED.

Invariant S5 allows us to turn the status of an area and the generator of a block into redundant variables of the (augmented) model. In the entire model the generator of a block is only used to keep track of the status of areas and the status of an area is only really used in the storage management operations ALLOCATE(A) and RELEASE. It is therefore sufficient to show that the status of an area can be removed from these operations (see System 4). First consider RELEASE. In this operation the status of an area is used in the predicate DEADTOP only, which should be true iff $S \neq \emptyset$ and status(top(S)) = dead. It is easy to infer from Invariant S5 that if $S \neq \emptyset$, the assertion

status(top(S)) = dead

is equivalent to:

scope(top(S)) > scope(environ(R)).

Consequently DEADTOP can be determined as follows:

```
Determination of DEADTOP
If $S = \emptyset$
    DEADTOP := false.
else
    Let T = top($S$).
    Let E = environ(R).
    DEADTOP := scope(T) > scope(E).
```

Next consider ALLOCATE(A). In this operation the status of an area is used in the determination of the number DEAD of dead cells in S and in COMPACT$_S$ (but not in COLLECT GARBAGE). From Invariants S2 and S5 (and a few more invariants) it can be inferred that the number DEAD can be determined as follows (see also Fig. 8):

Determination of DEAD

```
Let $S$ = <$A_1$, ..., $A_n$>.
s := 0.
k := n.
L := environ(R).
While k > 0
    While $A_k \neq$ L
        If scope($A_k$) > scope(L)
            s := s + size($A_k$).
        k := k - 1.
    L := establisher(L).
    k := k - 1.
DEAD := s.
```

While traversing S this way, dead areas could at the same time be marked as such. This would make it simple for COMPACT_S to determine whether an area is dead or not without using the status of the area.

The above shows that neither the status of an area nor the generator of a block need be implemented, thus avoiding a time and space overhead. That is, if only sequential programs are executed on the abstract machine. The latter assumption will be dropped in the next subsection.

4.6 Removing overhead in the general case

In the previous section we showed that in the sequential case we could do away with the status of an area and the generator of a block entirely in System 4. But what if the actions SPAWN(n), COMPLETE and SWITCH occur? Invariant S5 will no longer hold then and the trick used to implement the status of an area free of charge cannot be applied any more. However, an invariant analogous to Invariant S5 could be formulated, which relates the status of areas created by the <u>same</u> process to their scope. In order to implement the status of an area through this invariant it must be possible to determine for each area in S by which process it has been created. In the sequential case this is obvious, because there is only on process. In the parallel case it is far from obvious, because there may be many processes and the areas created by a specific process may be scattered all over S. This is even aggravated by the fact that after a process P has been completed areas created by P may be left behind in S. The extra bookkeeping necessary to apply the generalization of the implementation trick for the status of an area may thus become rather complicated and may cause a considerable overhead (which it was supposed to avoid). It is therefore better to look for an other solution.

In section 3 it was stated that efficiency of the storage management system to be designed should primarily be achieved for sequential programs. This implies that it is reasonable if the use of parallellism costs a little extra. It would not be reasonable, however, if the overhead connected with parallellism had a negative effect on the efficiency of sequential programs. The "easy" way to avoid the latter is to have two storage management systems: one for sequential and one for parallel programs. Yet, having two different storage management systems is not a very desirable situation. Let us see how we can avoid it.

Suppose that, instead of being able to determine by which process an area has been created, it were possible to determine whether the area has been created by the initial process P_0 or not. The latter, of course, is much easier to implement than the former. Let A_0 be the class of areas created by P_0 and A_1 the class of areas created by other processes. For all areas in A_0 the implementation trick for the status can be used (through an invariant analogous to Invariant S5). This implies that in the sequential case (where $A_1 = \emptyset$) the storage management system is just as efficient as before. For the areas in A_1 something more complicated must be done. The simplest way to implement the status of the areas in A_1 seems to be as follows. Let FINISH and JUMP(L, P) (which are the only operations that kill areas) mark dead locales in A_1 as such. This makes determination of the status of a locale in A_1 trivial. The status of a block B in A_1 can be determined by using the fact that status(B) = status(generator(B)) (Invariant B2). This implies that the generator function for blocks in A_1 must be implemented.

The scheme sketched above results in a storage management system, which for sequential programs is just as efficient as before. An overhead is introduced in parallel programs exclusively, and even then only when the program is really working in "parallel mode" (inside a parallel clause). The overhead, at first sight, seems to be acceptable. The price to be paid for all this is an increase of complexity of the system. The question is whether the increase of complexity outweighs the gain in efficiency or not. An alternative would be not to use the implementation trick for the status of areas in A_0, but to implement the status of areas in A_0 just like the status of areas in A_1. This results in a uniform approach, but also introduces an overhead in sequential programs. E.g., for all blocks the generator function must now be implemented. This could be compensated by not implementing the scope function for blocks explicitly. The fact that scope(B) = scope(generator(B)) for each block B (Invariant B3) can then be used to determine the scope of a block. The latter, however, makes scope checks more complicated and less efficient. Though one can certainly argue about it, we will let efficiency considerations prevail and choose for the original approach. It will be elaborated below.

The first thing we need is some way to distinguish areas created by P_0 from other areas. For that purpose we associate an extra entity to each locale and block:

Each locale L is augmented with
- kind(L): constant kind.

Each block B is augmented with
- kind(B): variable kind.

A <u>kind</u> is an element from the set {simple, extended}.

If an area has been created by P_0, its kind will be "simple", which implies that its status and generator (if it is a block) need not be implemented. The reverse will not hold, however. There are two reasons for that. First of all, for areas with scope = 0, which need not be created by P_0, neither the status nor the generator need be implemented: Their status is invariable "alive" and their generator (for blocks) is invariably equal to L_0. The kind of these areas will therefore also be chosen to be "simple".

Second, we wish the following invariant to hold (why this invariant is useful will turn out soon):

<u>Invariants for blocks $B \in \mathcal{B}$</u>

B4. kind(B) = kind(generator(B)).

This invariant may be disturbed by the operation KEEP(B, L). So it must be possible to change the kind of a block, which is the reason that the kind of a block is variable. The change of kind of a block, as we will see, is only from "extended" to "simple". If the reverse were also possible, this would (in view of the constant size of areas) annihilate the advantages of the distinction between simple and extended areas.

The model must be extended according to the above. First, the following should hold in the initial situation:

Initially

kind(L_0) = simple.

Note that it is now absolutely necessary that areas with scope = 0 have kind = simple (see Invariant B4). Next, when areas are created they should get the proper kind. A locale should get kind = simple iff $R = P_0$ at the moment of its creation, while a block should assume the kind of its generator. This amounts to the following additions to the operations ESTABLISH(t) and GENERATE(t, L):

```
ESTABLISH(t):
   Precondition:
      t is a type.
   Action:
      Let E = environ(R).
      Let L be a locale such that
      - $L \notin \mathcal{L}$,
      - status(L) = alive,
      - type(L) = t,
      - scope(L) = scope(E) + 1,
      - establisher(L) = E,
      - size(L) = $\sim$,
      If $R = P_0$
      | - kind(L) = simple.
      else
      | - kind(L) = extended.
      ALLOCATE(L).
      $\mathcal{L} := \mathcal{L} \cup \{L\}$.
      environ(R) := L.

GENERATE(t, L):
   Precondition:
      t is a type,
      $L \in \mathcal{L}$, $L \leq_\Lambda$ environ(R).
   Action:
      Let B be a block such that
      - $B \notin \mathcal{B}$,
      - status(B) = alive,
      - type(B) = t,
      - scope(B) = scope(L),
      - generator(B) = L,
      - size(B) = $\sim$,
      - kind(B) = kind(L).
      ALLOCATE(B).
      $\mathcal{B} := \mathcal{B} \cup \{B\}$.
```

Invariant B4 is trivially satisfied initially and is not violated by GENERATE(t, L). The only operation which might violate Invariant B4 is KEEP(B, L). This is remedied by the following addition to KEEP(B, L):

```
KEEP(B, L):
   Precondition:
      $B \in \mathcal{B}$, generator(B) = environ(R),
      $L \in \mathcal{L}$, $L \neq L_0$, $L <_\Lambda$ environ(R).
   Action:
      generator(B) := L.
      scope(B) := scope(L).
      kind(B) := kind(L).
```

Apart from Invariant B4 the following invariants can now be proved:

<u>Invariants for L_0</u>

K5. kind(L_0) = simple.

<u>Invariants for P_0</u>

O5. kind(environ(P_0)) = simple.

<u>Invariants for locales $L \in \mathcal{L}$</u>

```
L6. If kind(L) = simple
    | kind(establisher(L)) = simple.
```

<u>Invariants for processes $P \in \mathcal{P}$</u>

```
P9. If $P \neq P_0$
    | For each $L \in \mathcal{L}$ with origin(P) $<_\Lambda$ L
    | | kind(L) = extended.
```

Furthermore, if we define the set $\mathcal{T}$ as the set of simple areas in $\mathcal{S}$:

<u>Definition $\mathcal{T}$</u>

$\mathcal{T} = \{A \in \mathcal{S} \mid \text{kind}(A) = \text{simple}\}$.

then the following analogues of Invariants S2 through S4 can be proved:

<u>Invariants for $\mathcal{S}$ and $\mathcal{H}$</u>

U10. If $\mathcal{S} = \langle A_1, \ldots, A_n \rangle$ with $n > 0$ and $\mathcal{T} \neq \emptyset$
| $A_1 \in \mathcal{T} \cap \mathcal{L}$ and $\text{status}(A_1) = \text{alive}$.

U11. For each $L \in \mathcal{T} \cap \mathcal{L}$ with $\text{status}(L) = \text{alive}$
and each $M \in \mathcal{T} \cap \mathcal{L}$
| $L <_S M \Rightarrow L <_\Lambda M$.

U12. For each $L \in \mathcal{T} \cap \mathcal{L}$ with $\text{status}(L) = \text{alive}$
and each $B \in \mathcal{T} \cap \mathcal{B}$ with $\text{status}(B) = \text{dead}$
| $L <_S B \Rightarrow L <_S \text{generator}(B)$.

If we (re)define the "base" of an area in $\mathcal{T}$ as follows:

<u>Definition base(A)</u>

For each $A \in \mathcal{T}$
| $\text{base}(A) = \max\limits_{<_S} \{L \in \mathcal{T} \cap \mathcal{L} \mid L \leq_S A, \text{status}(L) = \text{alive}\}$.

then the following analogue of Invariant S5 can be derived from Invariants U11 and U12:

<u>Invariants for $\mathcal{S}$ and $\mathcal{H}$</u>

U13. For each $A \in \mathcal{T}$
| $\text{status}(A) = \text{dead} \Leftrightarrow \text{scope}(A) > \text{scope}(\text{base}(A))$.

The proofs of Invariants U10 through U13 are (almost) entirely analogous to the proofs of Invariants S2 through S5, hence they are omitted.

The status of a simple locale, the status of a block and the generator of a simple block can now be turned into redundant variables. As in the sequential case it suffices to show this for the determination of DEADTOP and DEAD. Consider DEADTOP first. Invariant U13 implies that if $\mathcal{S} \neq \emptyset$ and $\text{kind}(\text{top}(\mathcal{S})) = \text{simple}$, the assertion

$\text{status}(\text{top}(\mathcal{S})) = \text{dead}$

is equivalent to

$$\text{scope}(\text{top}(\mathcal{S})) > \text{scope}(\text{environ}(P_0)).$$

If kind(top($\mathcal{S}$)) = extended, two cases must be distinguished. First, if top($\mathcal{S}$) is a locale its status can be determined directly. Second, if top($\mathcal{S}$) is a block, its status is equal to the status of its generator G (Invariant B2). The status of G, again, can be determined directly, because kind(G) = extended (Invariant B4). Notice that if Invariant B4 did not hold, it were much more difficult to determine the status of G. The above implies that DEADTOP can be implemented as follows:

```
Determination of DEADTOP
If $\mathcal{S} = \emptyset$
    DEADTOP := false.
else
    Let T = top($\mathcal{S}$).
    If kind(T) = simple
        Let E = environ($P_0$).
        DEADTOP := scope(T) > scope(E).
    else
        If T $\in \mathcal{L}$
            DEADTOP := status(T) = dead.
        else
            Let G = generator(T).
            DEADTOP := status(G) = dead.
```

Invariants U10 and U13 imply that the number of dead cells in $\mathcal{S}$ can be determined as follows:

Determination of DEAD

```
Let $S = \langle A_1, \ldots, A_n \rangle$.
s := 0.
k := n.
L := environ($P_0$).
While k > 0
    While k > 0 and $A_k \neq L$
        If kind($A_k$) = simple
            If scope($A_k$) > scope(L)
                s := s + size($A_k$).
        else
            If $A_k \in L$
                If status($A_k$) = dead
                    s := s + size($A_k$).
            else
                Let G = generator($A_k$).
                If status(G) = dead
                    s := s + size($A_k$).
        k := k - 1.
    If k > 0
        L := establisher(L).
        k := k - 1.
DEAD := s.
```

We have shown above, that in order to implement System 4 in the general case of both sequential and parallel programs neither the status of a simple area nor the status of a block nor the generator of a simple block need be implemented. We will formally complete the design by removing these variables from the model.

4.7 Stripping the model

The removal of redundant variables from the model starts with a reduction of the entities associated to locales and blocks.

```
Each locale L has
- type(L): constant type,
- scope(L): constant integer,
- establisher(L): constant locale,
- size(L): constant integer,
- kind(L): constant kind,
If kind(L) = extended
|  - status(L): variable status.

Each block B has
- type(B): constant type,
- scope(B): variable integer,
- size(B): constant integer,
- kind(B): variable kind,
If kind(B) = extended
|  - generator(B): variable locale.
```

The entities associated to processes remain the same. The initial state of the model is reduced only as far as the initial conditions for L_0 are concerned:

Initially
- type(L_0) = $\sim$,
- scope(L_0) = 0,
- establisher(L_0) = L_0,
- size(L_0) = $\sim$,
- kind(L_0) = simple.

Next, the redundant variables must be removed from the operations of the (no longer entirely) abstract machine. For the operations ESTABLISH(t), FINISH and GENERATE(t, L) this is straightforward. These operations are given below.

```
ESTABLISH(t):
   Precondition:
      t is a type.
   Action:
      Let E = environ(R).
      Let L be a locale such that
      - L $\notin \mathcal{L}$,
      - type(L) = t,
      - scope(L) = scope(E) + 1,
      - establisher(L) = E,
      - size(L) = $\sim$,
      If R = $P_0$
      | - kind(L) = simple.
      else
      | - kind(L) = extended,
      | - status(L) = alive.
      ALLOCATE(L).
      $\mathcal{L}$ := $\mathcal{L} \cup$ {L}.
      environ(R) := L.

FINISH:
   Precondition:
      environ(R) $\neq$ origin(R).
   Action:
      Let E = environ(R).
      environ(R) := establisher(E).
      If kind(E) = extended
      | status(E) := dead.
      RELEASE.
```

```
GENERATE(t, L):
   Precondition:
      t is a type,
      $L \in \mathcal{L}$, $L \leq_{\Lambda}$ environ(R).
   Action:
      Let B be a block such that
      - $B \notin \mathcal{B}$,
      - type(B) = t,
      - scope(B) = scope(L),
      - size(B) = $\sim$,
      - kind(B) = kind(L),
      If kind(L) = extended
      | - generator(B) = L.
      ALLOCATE(B).
      $\mathcal{B} := \mathcal{B} \cup \{B\}$.
```

In removing the redundant variables from the operation KEEP(B, L) an interesting problem arises. The precondition of KEEP(B, L) contains a condition on the generator of B. However, in the new model the generator of B need not exist (to wit, if kind(B) = simple). We can do two things now. First, we can simply do away with the preconditions of operations. They are not supposed to be implemented (as run-time checks) anyway. They are only meant for the code generator, who must make sure they are satisfied whenever an operation is used. Second, we can replace the condition on the generator of B by an equivalent one which does not use the generator of B if kind(B) = simple. The latter seems the more elegant solution, which we will choose for here. It requires the proof of an additional invariant (in the "old" model):

<u>Invariants for $\mathcal{S}$ and $\mathcal{H}$</u>

U14. For each $L \in \mathcal{T} \cap \mathcal{L}$ with status(L) = alive
and each $B \in \mathcal{T} \cap \mathcal{B}$
| generator(B) = L $\Leftrightarrow$ $L <_{\mathcal{S}} B$ and scope(B) = scope(L).

This invariant can be derived from the invariants already formulated (in particular from Invariant U11). The generator of a simple block can now also be removed from the precondition of KEEP(B, L):

```
KEEP(B, L):
   Precondition:
      $B \in \mathcal{B}$, $L \in \mathcal{L}$, $L \neq L_0$, $L <_\Lambda$ environron(R),
      If kind(B) = simple
      | environ(R) $<_S$ B and
      | scope(B) = scope(environ(R)).
      else
      | generator(B) = environ(R).
   Action:
      scope(B) := scope(L).
      kind(B) := kind(L).
      If kind(B) = extended
      | generator(B) := L.
```

The operations SPAWN(n), COMPLETE and SWITCH remain entirely the same. This leaves only the operation JUMP(L, P). Suppose that prior to JUMP(L, P) the assertion $R = P_0$ holds (i.e. the program is in "sequential mode"). From the fact that $P \leq_\Pi P_0$ (see the precondition of JUMP(L, P)) and $P_0 \leq_\Pi P$ (Invariant P4) we know that $P = P_0$. Hence the actions:

```
R := P.
mode(R) := active.
```

in the definition of JUMP(L, P) reduce to dummy actions. Though this is not so for the action:

```
environ(R) := L.
```

it also applies to the rest of the actions in the definition of JUMP(L, P) except to RELEASE of course. First consider

```
For each $M \in \mathcal{L}$ with $L <_\Lambda M$
| status(M) := dead.
```

This action can be removed because the following assertion (which is not disturbed by "environ(R) := L") holds prior to JUMP(L, P):

```
For each $M \in \mathcal{L}$ with $L <_\Lambda M$
| status(M) = alive $\Rightarrow$ kind(M) = simple.
```

Next, the action

```
For each $B \in \mathcal{B}$ with $L <_\Lambda$ generator(B)
| status(B) := dead.
```

can be removed because the status of a block is a redundant variable. Finally, the action

```
For each $Q \in \mathcal{P}$ with $P <_\Pi Q$
| mode(Q) := completed.
```

can be removed because P (= P_0) is the only process in $\mathcal{P}$ which is not yet completed. If $R \neq P_0$ prior to JUMP(L, P) the required changes in JUMP(L, P) are obvious. All in all we get:

```
JUMP(L, P):
   Precondition:
      $L \in \mathcal{L}$, $L \leq_\Lambda$ environ(R),
      $P \in \mathcal{P}$, $P \leq_\Pi R$,
      origin(P) $\leq_\Lambda L \leq_\Lambda$ environ(P).
   Action:
      If $R = P_0$
      |  environ(R) := L.
      else
      |  R := P.
      |  mode(R) := active.
      |  environ(R) := L.
      |  For each $M \in \mathcal{L}$ with $L <_\Lambda M$ and kind(M) = extended
      |  |  status(M) := dead.
      |  For each $Q \in \mathcal{P}$ with $P <_\Pi Q$
      |  |  mode(Q) := completed.
      RELEASE.
```

The only thing that remains to be done is the rewriting of the invariants. That is, the redundant variables must also be removed from the invariants. This is a straightforward matter which will be omitted. The "stripping" of the model is herewith completed. Yet, the system is still in a rather abstract form. The final implementation of the system in "hard code", which is a purely technical matter, will be discussed in the next subsection.

4.8 Final implementation

In the introduction we stated that the storage management system should be written in a subset of the code of the abstract machine. Up till now we only have a description in terms of algorithms which operate on the abstract variables of the (enhanced) machine model. In order to obtain a storage management system written in abstract machine code the entire model must be mapped back to the abstract machine. There are two aspects to this mapping.

First of all there is the data structure aspect. A layout for the data structures of the model must designed. This layout specifies how the entities associated to locales, blocks and processes are implemented as subfields of the fields occupied by these data structures in the store of the abstract machine. Note that in reality there are more entities associated to locales, blocks and processes than we discussed here. We discussed only those entities which were of interest to the storage management problem. The design of such a layout is not difficult. Optimizations are often possible by combining different entities in the same subfield. Having defined a layout, the referencing or changing of an entity associated to a locale, block or process can directly be translated into an instruction of the abstract machine which accesses the corresponding subfield (using a "pointer" and an "offset").

Second, there is the control structure aspect. Most of the control structures used in the algorithms (such as while-loops) can directly be translated into abstract machine code. The only two control structures the translation of which is not entirely trivial are the two for-loops in JUMP(L, P). Using the (rewritten) invariants one can transform the definition of JUMP(L, P) into the following more readily translatable form, however:

```
JUMP(L, P):
   Precondition:
      $L \in \mathcal{L}$, $L \leq_\Lambda$ environ(R),
      $P \in \mathcal{P}$, $P \leq_\Pi R$,
      origin(P) $\leq_\Lambda L \leq_\Lambda$ environ(P).
   Action:
      If $R = P_0$
        environ(R) := L.
      else
        Let E = environ(P).
        R := P.
        mode(R) := active.
        environ(R) := L.
        If $P \neq P_0$
          M := E.
          While $M \neq L$
            status(M) := dead.
            M := establisher(M).
        $\mathcal{Q} := \{S \in \mathcal{P} \mid$ spawner(S) = P, mode(S) $\neq$ completed$\}$.
        While $\mathcal{Q} \neq \emptyset$
          Let $Q \in \mathcal{Q}$.
          $\mathcal{Q} := \mathcal{Q} \setminus \{Q\}$.
          mode(Q) := completed.
          M := environ(Q).
          While $M \neq$ origin(Q)
            status(M) := dead.
            M := establisher(M).
          $\mathcal{Q} := \mathcal{Q} \cup \{S \in \mathcal{P} \mid$ spawner(S) = Q, mode(S) $\neq$ completed$\}$.
      RELEASE.
```

In the other algorithms several optimizations are possible too.

The translation of the algorithms into abstract machine code will thus not pose any serious problems. The only operations which cannot directly be translated are COLLECT GARBAGE and $\text{COMPACT}_{\mathcal{S}}$. The design of efficient algorithms for these operations (based on the specifications given in the foregoing) will be treated in a separate paper [7].

5. CONCLUSION

The implementation of a programming language is a highly complex process. In order to keep this process under control a "divide and rule" approach is absolutely mandatory. The job should be divided in clearly interfaced sub-tasks, which should be as independent from each other as possible. The division should be done with great care in order to keep the implementation as efficient as possible. One of the sub-tasks is the construction of a storage management system. In this paper it was demonstrated through an example that indeed the design of a storage management system can be viewed as a relatively independent part of the language implementation effort. The interface with the other parts of the implementation consisted of an abstract model, which contained exactly the information relevant to the storage management problem and no more than that. It allowed us to approach the problem in a systematic and rigorous way, up to a level of formality which allowed proofs of correctness. Since all irrelevant details were discarded, the design process remained transparent and things could be kept relatively simple. The final result of the design process was an efficient storage management system. The majority of the techniques used in this system are certainly not novel. The primary goal of this paper was not to demonstrate some fancy storage management technique. Its main purpose was to demonstrate a technique to design a storage management system in a systematic way.

Apart from a number of advantages already mentioned in the introduction, the major advantage of the method demonstrated in this paper over the more usual ("classical") approach to the design of storage management systems (such as described in e.g. [1, 4, 5, 8]) is considered to be the fact that it forces one to a separation of concerns. In the process of designing a storage management system for an implementation of a programming language L on a machine M a number of concerns can be distinguished, which were clearly separated in the foregoing:

1. The programming language L.
2. The machine M.
3. The definition of the problem.
4. The design of the algorithms.
5. The implementation of the system.

Concerns 1 and 2 are often not well separated. In any but a purely interpretive implementation a storage management system should not be designed for (the machine corresponding to) the programming language L, but for the machine M into code of which programs in L are translated. The operation KEEP(B, L), which does not correspond to any ALGOL 68 construct, demonstrates this clearly. Of course, if the abstract machine approach is pursued, there will usually be a certain correspondence between L and M (the more abstract the machine is, the closer this correspondence will generally be). Good abstract machines (or better, their definitions) should be such that they can be implemented without using information on the programming language L or the way programs in L are translated into code for M, however. Admittedly with the MIAM [9] we were in a rather fortunate position in this respect. During the design process we only seldom needed a reference to ALGOL 68.

The third concern, the definition of the problem, is usually either omitted or taken for granted. Lacking a simple model free of irrelevant detail it is indeed not easy to define precisely what the storage management problem amounts to. Yet an unambiguous statement of the problem is essential to the reliability of the system to be developed. E.g., should we not have clearly defined what operations on the allocation function were allowed, we might have erroneously deallocated living areas.

Concerns 4 and 5 are most often confused. It is generally accepted that in designing an algorithm (or a program) one should keep the algorithm (or the program) free from implementation detail as long as possible. It enables one to keep a clear view at the algorithm under development. Thus possible improvements of the algorithm are more easily discovered. An example of this was the discovery of Invariant S5, which enabled a substantial improvement of the efficiency of the storage management system. It is questionable whether this invariant would ever have been discovered (let alone that it could have been proved) if we had not kept the system as abstract as we did. The separation of concerns 4 and 5 also helps in keeping the presentation of the algorithms digestible. Interspersing an algorithm with implementation details can make the algorithm utterly unreadable.

A sixth concern could be added to the above list of separated concerns: the design of the garbage collector. This concern was separated because it constitutes a problem so different from the rest of the design that it justifies a separate treatment. The fact that this concern could be separated shows the power of the technique.

REFERENCES

[1] BRANQUART, P. & J. LEWI, A scheme of storage allocation and garbage collection for ALGOL 68, In: ALGOL 68 Implementation, Ed. J.E.L. Peck, North Holland Publishing Company, Amsterdam (1971).

[2] DIJKSTRA, E.W., Cooperating sequential processes, In: Programming Languages, Ed. F. Genuys, Academic Press, London (1968).

[3] ELSWORTH, E.F., Compilation via an intermediate language, Computer Journal 22 (1979), 226-233.

[4] GRIES, D., Compiler Construction for Digital Computers, John Wiley & Sons, New York (1971).

[5] HILL, U., Special run-time organization techniques for ALGOL 68, In: Compiler Construction, Ed. F.L. Bauer & J. Eickel, Springer-Verlag, New York (1974).

[6] JONKERS, H.B.M., Deriving algorithms by adding and removing variables, Mathematical Centre Report IW 134/80, Amsterdam (1980).

[7] JONKERS, H.B.M., Designing a garbage collector, To appear, Mathematical Centre, Amsterdam.

[8] KNUTH, D.E., The Art of Computer Programming, Vol. 1: Fundamental Algorithms, Addison-Wesley, Reading, Mass. (1968).

[9] MEERTENS, L.G.L.T., Definition of an abstract ALGOL 68 machine, Working document, Mathematical Centre, Amsterdam.

[10] TERASHIMA, M. & E. GOTO, Genetic order and compactifying garbage collectors, Information Processing Letters 7 (1978), 27-32.

[11] WIJNGAARDEN, A. VAN, B.J. MAILLOUX, J.E.L. PECK, C.H.A. KOSTER, M. SINTZOFF, C.H. LINDSEY, L.G.L.T. MEERTENS & R.G. FISKER (Eds.), Revised Report on the Algorithmic Language ALGOL 68, Springer-Verlag, New York (1976).