ON THE FEASIBILITY OF CONCURRENT GARBAGE COLLECTION

KLAUS G. MÜLLER

ON THE FEASIBILITY OF CONCURRENT GARBAGE COLLECTION

PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR IN DE TECHNISCHE WETENSCHAPPEN AAN DE TECHNISCHE HOGESCHOOL DELFT, OP GEZAG VAN DE RECTOR MAGNIFICUS, PROF. DR. IR. H. VAN BEKKUM, VOOR EEN COMMISSIE AANGEWEZEN DOOR HET COLLEGE VAN DEKANEN, TE VERDEDIGEN OP WOENSDAG 3 MAART 1976 TE 16.00 UUR.

DOOR

KLAUS ALFRED GEORG MÜLLER

ELEKTROTECHNISCH INGENIEUR

GEBOREN TE BERLIN, DUITSLAND

DIT PROEFSCHRIFT IS GOEDGEKEURD DOOR DE PROMOTOR
PROF. DR. IR. W. L. VAN DER POEL

Meiner Mutter, Virginia,
Nicole und Sonja.
Sie machten es möglich.

ACKNOWLEDGEMENTS

SHAPE Technical Centre has given me very significant support in the preparation of this thesis. For this, I wish to thank the management of STC.

I am indebted to Dr. F.B. Kapper and Dr. W.R. O'Brien whose encouragement and friendly advice helped me to bring this work to a successful end.

My thanks are also due to Mr. H.C. Rijnders who has been of great help in finding and obtaining the publications referenced in this thesis. I am also grateful for the cooperation and kind support I had from the staff of STC's Services Group during all phases of my work.

With love and deeply felt gratitude, I must thank my wife Virginia and my daughters Nicole and Sonja. My wife typed the various drafts which finally led to this thesis. More important, however, was her moral support and understanding which kept me going, despite all difficulties. Nicole and Sonja were very patient with their father who so often during the last few years had to work and could not play with them.

SUMMARY

This thesis addresses the problem of designing and implementing concurrent garbage collection systems for dynamic storage allocation. So far, none of the important programming languages using garbage collection for the automatic run-time management of heap storage, such as ALGOL 68, LISP, and SIMULA, could be employed to program time-critical real-time programs. The reason for this has been that in all implementations of these languages to date, the garbage-collector is a sequential system procedure which is called when the free heap space is exhausted or has to be compactified. These calls occur at unpredictable points of time. While the garbage-collector is executing, the applications program must be suspended. This suspension can amount to up to several seconds during which the applications program cannot respond to external events.

The main hypothesis of this thesis is that language processors with garbage collectors which work incrementally and concurrently with applications programs can be correctly designed and implemented. Such garbage-collectors interfere with applications programs only for short, limited periods. Thus, these programs can be given a predictable real-time behavior.

The thesis describes applications program and concurrent garbage-collector as two parallel co-operating processes. For this system, two principal problem areas are identified, viz. process coordination and scheduling.

The two processes share data, and their access to this data has to be ordered in time to ensure correct operation of the complete system. The thesis developes general solutions for this problem based on invariants for the shared variables. A necessary set of 11 invariants to be maintained by the critical regions of the two processes in concurrent garbage collection systems is identified.

The scheduling problem occurs because applications program and garbage-collector are in a consumer/producer relationship concerning free space. A model of this relationship is developed, and from it the level of free space at which to trigger off the garbage collection process is derived. Based on this, the maximum possible heap utilisation and the memory space- and processor time overheads of concurrent garbage collection are established.

To show the general applicability of the approach, two concurrent garbage collection systems are presented in the form of programs. One system supports fixed-length LISP-like locations and uses a marking algorithm which works recursively in CAR and iteratively in CDR direction. The other supports variable-length locations and therefore has a compactifying garbage-collector. It uses a non-recursive copying algorithm which copies accessible locations to a new area in memory.

The two systems are programmed in PASCAL, extended by four constructs, viz. shared variables, concurrent statements, critical regions, and conditional critical regions. The correctness of both systems is proved.

Note

It appears that solutions to the problem posed by concurrent garbage collection have been "in the air". After the submission of this thesis to the supervisor on 4 August 1975, two important papers on concurrent garbage collection have been published.

The first one, by Dijkstra et.al.[1], describes a non-relocating concurrent garbage collection algorithm for LISP-like data-structures, i.e., fixed-length locations with two pointers each. The correctness of the algorithm is shown. The most significant aspect of the algorithm is the minimal mutual exclusion and synchronisation constraints on the two concurrent processes "list-processor" and "collector".

The second paper, by Steele[2], is the winning paper of ACM's 1975 George E. Forsythe Student Paper Competition. It presents in great detail a compactifying concurrent garbage-collector/list-processor system for fixed-length locations. No correctness proofs are offered.

The overlap between those two papers and this thesis is small. In this thesis, Dijkstra's and Steele's papers have neither been used nor referenced.

---

(1) E.W. Dijkstra, L.Lamport, A.J. Martin, D.S. Scholten, and E.F.M. Steffens "On-the-fly garbage collection: An exercise in cooperation", Lecture Notes of International Summer School on Language Hierarchies and Interfaces, Munich, Juli 23 to August 2, 1975.

(2) G.L. Steele Jr. "Multiprocessing compactifying garbage collection", CACM, Sep. 1975, Vol. 18, No. 9.

TABLE OF CONTENTS

		Page
INTRODUCTION		1
Chapter		
1	STORAGE MANAGEMENT	5
	1.1 THE STORAGE ALLOCATION PROBLEM	6
	1.1.1 Static allocation	9
	1.1.2 Dynamic storage allocation on a stack	9
	1.1.2.1 Fixed size data	9
	1.1.2.2 Variable-size data	10
	1.1.3 Dynamic storage allocation using a heap	11
	1.2 SCHEMES FOR HEAP MANAGEMENT	14
	1.2.1 Objective of storage reclamation	14
	1.2.2 Heap organisation	14
	1.2.3 Storage allocation under programmer control	15
	1.2.3.1 Techniques	15
	1.2.3.2 Programming languages implemented with this technique	17
	1.2.4 Storage allocation using the reference-counter approach	20
	1.2.4.1 Technique	20
	1.2.4.2 Programming languages implemented using this technique	22
	1.2.5 Storage allocation using garbage collection	22
	1.2.5.1 Technique	22
	1.2.5.2 Tracing and marking	24
	1.2.5.3 Collecting without relocation	30
	1.2.5.4 Compactification	30
	1.2.5.5 Garbage collection overheads	34
	1.2.5.6 Languages implemented with garbage collection	36
	1.2.6 Comparison between heap management systems	38
	1.3 THE REQUIREMENT FOR CONCURRENT GARBAGE COLLECTION	40
2	CONCURRENT GARBAGE COLLECTION	42
	2.1 THE PROBLEM	44
	2.2 PROCESS CONTROL	47
	2.2.1 The conceptual tools	49

Page

2.2.2 Shared variables and invariants 54
Assertions concerning termination 56
Assertions concerning the free space 59
Invariants concerning the stack 63
Invariants concerning the heap 72
Shared variables and deadlock 73

2.3 SCHEDULING OF GARBAGE COLLECTOR 76
2.3.1 Model 77
2.3.2 Design parameters for concurrent garbage collection 83

2.4 OVERHEADS 85
2.4.1 Heap space overhead 86
2.4.2 Processor overhead 86

2.5 SUITABLE ALGORITHMS 88

3 EXAMPLES 90
3.1 A NON-RELOCATING CONCURRENT GARBAGE COLLECTION SYSTEM 91
3.1.1 Data structures supported 91
3.1.2 Heap management 91
3.1.3 Program description 92
3.1.4 Correctness proofs 93
3.1.4.1 Invariants 94
3.1.4.2 Outstanding correctness proofs 98

3.2 A CONCURRENT COMPACTIFYING GARBAGE COLLECTION SYSTEM 101
3.2.1 Data structures supported 101
3.2.2 Heap management 102
3.2.3 Program description 104
3.2.4 Correctness proofs 107
3.2.4.1 Invariants 109

4 DISCUSSION 116
4.1 FINDINGS AND CONCLUSIONS 116

4.2 LIMITATIONS 120

4.3 FUTURE RESEARCH 121

FIGURES 127

Page

APPENDIX A: THE LANGUAGE USED 177

APPENDIX B: ON THE IMPLEMENTATION OF BRINCH HANSEN'S PROCESS EXTENSIONS TO PASCAL 184

REFERENCES 188

SAMENVATTING 191

CURRICULUM VITAE

LEST-LOST-LAST-LISTLESSNESS DEPRESS YOU

When too many errands accrue
it's useful to make out a list.
You're certain to lose it, it's true;
but somewhere, the thing will exist.
And then, when some accident brings
the list you have lost into view,
at least you've a list of the things
you've meanwhile forgotten to do.

Piet Hein, Grooks V

INTRODUCTION

A group of programming languages, such as ALGOL 68, LISP, and SIMULA 67, use a technique called *garbage collection* for the run-time management of storage. The name is very descriptive of what the technique is all about: the executing applications program consumes free storage without restraint. When it no longer requires a storage location, it does not attempt to reuse it, but just ignores it. The program behaves as if its free storage could not be exhausted. But, of course, there comes the point where no free storage is available any more. At this moment, the program stops, and a utility, the *garbage-collector*, takes over. It regenerates free storage by finding and collecting the locations which the applications program has thrown away. The garbage is recycled and transformed into free storage. When it has finished, the application program continues, quite unaware that it had just run out of a very valuable resource (somehow an analogy between this process and man's ecological behaviour seems to suggest itself).

Maurer (Ma 72) describes this aptly by saying ".... the (LISP) program proceeds 'like a jerky automobile' - every so often, everything stops ....". Depending on the size of storage in which the garbage has to be found, this pause can last several millions of central processor cycles, several seconds of real-time, and it can happen quite frequently if a program consumes and throws away its storage space at a high rate.

Effectively, time is traded for space, and for a large class of applications, this is acceptable. 'Real-time' programs do not belong to this class. Their computation speed must be sufficient to keep up with the process(es) they are controlling or co-operating with. Knuth (Kn 68, p.412) remarks on this "We might

also note that garbage collection is unsuitable for 'real-time' applications, because even if the garbage-collector goes into action infrequently, it requires large amounts of computer time on these occasions".

It was the increasing interest in real-time artificial intelligence applications such as robotics and pattern-recognition which in the late sixties caused computer scientists to search for means of overcoming this deficiency of list-processing languages. Bobrow (Bo 68) proposed a modification to the garbage-collector of a LISP 1.5 system on an SDS 940 computer. This modification was supposed to work incrementally as a separate process, sharing central processor time with executing LISP programs. The resulting system was expected to allow time-shared execution of several LISP programs, including real-time processes, with only a reduction in speed, but no pause during garbage collection.

Knuth attributes this idea to Minsky and poses its solution as a "moderately hard" exercise in his excellent series "The Art of Computer Programming" (Kn 68). No trace of any further work on this proposal can be found in the literature.

Barbacci et al. (Ba 71) propose a "parallel (compacting) garbage collection" system for a large microprogrammed LISP processor. This proposal goes into considerably more detail than the ones by Bobrow and Knuth. Again, there is no trace of its development or implementation anywhere in the literature.

The author can offer no explanation for this discontinuation of obviously promising approaches. Certainly, the need for a garbage collection system for real-time programs has not diminished.

The work reported in this doctoral thesis is an attempt to show that the problems related to concurrent garbage collection can be isolated by analytical means, that the conceptual tools for solving them are available, and that concurrent garbage collection systems can be designed which function safely and efficiently. In order to make the results of this research of immediate practical

use, two detailed examples of concurrent garbage collection algorithms are presented in the text. Informal proofs for the correctness of these algorithms are given.

It is worth stressing that the primary contribution of this thesis is neither the testing of the hypothesis "concurrent garbage collection is feasible" nor the development of any specific concurrent garbage collection algorithm, but the ordered analysis and structured design of concurrent garbage collection systems in general. The objective is to demonstrate that the intellectual tools for this are available and that these can be employed to design concurrent garbage collection systems whose correctness can be proved without the need for any "ad-hoccery".

The approach used by the author is entirely based on the "concurrent process" concept. The list-processing program on one side and the garbage-collector on the other are identified as concurrent processes. These two processes are shown to share resources, namely variables. They are therefore *interacting* or *communicating processes*.

Denning (De 71, p.195) discusses four control problems of importance for concurrent processes:

1) *Determinacy* - A system of processes is determinate if the outcome of the joint computation is independent of the relative speeds of the processes.

2) *Deadlock* - A deadlock of processes can occur when it is possible for processes to be blocked indefinitely because each process holds non-preemptible resources requested by other processes.

3) *Mutual exclusion* - This problem arises whenever two independent processes may require the use of a reusable resource, such as a variable. Each process may modify the state of the resource, and therefore, no two processes may be using the resource at the same instant.

4) *Synchronisation* - When the progress of one process depends on that of another, synchronisation is necessary.

These four problems are analysed for the list-processor/garbage collector system of concurrent processes. Solutions are provided which take the form of either design features or additional control mechanisms, to constrain the sequence of states through which the computation progresses, to a sequence of safe states.

To permit correctness proofs for the solutions developed, a well-structured programming language is used, namely PASCAL extended with concurrent statements and both simple and conditional critical regions as developed by Brinch Hansen (Ha 73). This notation permits assertions concerning invariant relationships among program components to be made.

For the purposes of this thesis, the author treats the synchronisation constructs used as primitives, i.e., their availability is assumed. (Appendix B discusses briefly how they could be implemented by lower-level primitives, such as semaphores and P- and V-operations on these.) However, in order to test the feasibility of implementing the designed concurrent garbage collection algorithms, both examples presented have been mapped into sequential PASCAL programs and executed on a CDC 6400 computer under a batch operating system. The concurrency of list-processor and garbage-collector was simulated by (almost) arbitrary interleaving of their execution, i.e., the *concurrent* garbage-collector was mapped into a *serial* one.

Chapter 1
STORAGE MANAGEMENT

The purpose of this chapter is to discuss the garbage collection method in the wider context of storage management on a digital stored-program computer.

The discussion is conducted in terms of individual programs and their storage requirements, i.e., in the context of storage management performed by language compilers and their run-time systems. This is not to imply, though, that the storage management techniques discussed are not applicable to the storage management which an operating system must perform, e.g. of a computer's central memory or its file store.

Starting with an abstract model of the functional interrelation between a program and its data on one side and the storage in which program and data are represented during execution on the other, three different regimes of storage allocation - static allocation, classical dynamic allocation, and dynamical allocation using a heap - are discussed. In the interest of clarity and uniformity of language, the terminology of the ALGOL 68 Report (Wi 69) will be employed wherever applicable.

It will be shown how features of programming languages lead to different requirements for storage allocation.

Garbage collection will be compared to two other techniques for heap management, namely, the reference count approach and the explicit request-and-return of storage locations in the applications program.

1.1 THE STORAGE ALLOCATION PROBLEM

A *program* is a piece of text which defines a sequence of actions to be performed by a *computer*. This sequence of actions is termed the program's *elaboration*. The actions are performed on *internal objects* in the working store of the computer. These objects are *instances* of values and are represented by bit-patterns in the computer's working store.

A program can be parsed into *external objects* (operators, declarations, identifiers, etc.). "A + 3" for example consists of three such external objects, viz. the identifier A, the constant 3, and the operator +.

External objects may *possess* internal objects (Fig. 1 - 1). E.g., the external object (denotation) 3 possesses the internal value 'three' which is some bit-pattern which may differ from computer to computer.

An internal object has three attributes:

(1) it is of some *mode*,

(2) it is an instance of a *value* of that mode,

(3) it has a *location*

The mode of objects is characterised by the operations which may be performed on them, i.e., in which operations objects of this mode may be used as operands. For example, the operations "successor of", "addition", and "multiplication" are defined for natural numbers, but not for truth values.

Several external objects may possess an instance of the same value, i.e., the mapping "possesses" does not have to be reversible: at the same time, the value "three" (a specific bit-pattern) can be possessed e.g. by a constant TRES and by another constant III.

An internal object is to be found somewhere in the computer's store, namely at its location.

In addition to primitive objects (constants) such as

characters, integers, or strings, there exists a group of internal objects which as their values possess references to instances of other values. Such objects are called *names*. Names are effectively an abstraction of the concept "address" of real computer memories.

Whereas the relationship "to possess" exists between an external and an internal object, the relationship "to refer to" exists between two internal objects, viz. the name and its value. Names themselves can be reference objects (values) of other names.

The action of making a name refer to a value is called *assignation*. In ALGOL 68, this action and the resulting reference are represented by the := sign. A pair (name, value) of objects between which a reference exists is called a *variable*. The referenced object is called the variable's *content*. The relation between the name's and the value's notation can also be represented by the := sign (Fig. 1 - 2). This relation can be changed by another assignation, and this explains the term "variable".

In a program, variables are distinguished from each other by *identifiers* which are introduced by *declarations*. In general, the elaboration of such a declaration

- generates an internal object in memory, i.e., *allocates storage*,
- makes an identifier *possess* the name of the object generated,
- gives the variable a *type*, i.e., the set of values it may refer to, and
- *initialises* the variable's content.

How then is a variable represented in a computer at program execution time? Computer memory is organised into equally sized cells, called words, bytes, or bits. The objects of different *modes* may however be of different sizes. Integer objects require more memory space than character or boolean objects. One must therefore distinguish between the addressable units (*cells*) of real memory and the space occupied by objects, their *locations*.

Each location has a unique (numerical) address identifying it, its *name*.

Fig. 1 - 3 shows a snapshot of the memory representation of a piece of an ALGOL 68 program. The snapshot is taken after the declaration of three variables of different modes, char, int, and bool which are possessed by the identifiers "FIRST INITIAL", "AGE", and "OVERWEIGHT" and which are assigned values. The memory snapshot shows a memory organised into eight-bit bytes.

On this (specific) computer, a character object requires one byte, an integer object six bytes, and a boolean object less than one byte, namely one bit.

The locations referenced by the variables with the identifiers FIRST INITIAL, AGE, and OVERWEIGHT have addresses 999, 1000, and 1006, respectively.

The user of a high-level language can effectively disregard the storage characteristics of the computer his program will finally be executed on. He programs for a *virtual store* (Ha 73, p.159).

This virtual store is the union of all variables he uses in his program. Thus, the virtual store is the mapping of identifiers into values:

VIRTUAL STORE : IDENTIFIER → VALUE

Real (i.e. physical) store, on the other hand, consists of memory cells identified by numbers called *addresses*. Each of these cells can have a content, its value. Real store can therefore be regarded as the mapping:

REAL STORE : ADDRESS → VALUE

Before a program can be executed, memory cells in real store must be assigned to its identifiers. The required mapping is called *storage allocation:*

STORAGE ALLOCATION : IDENTIFIER → ADDRESS

Storage allocation is done partly at program translation

time (*static allocation*), and partly at execution time (*dynamic allocation*).

Different programming languages use different virtual store concepts. Therefore, different systems to simulate the various virtual stores on physical storage devices are required.

In the following, three different types of storage allocation will be discussed. Terminology and approach of this discussion are based on publications by Wodon (Wo 69), Griffiths (Gr 74), and Bauer et al. (Ba 71a).

1.1.1 Static allocation

Static allocation is storage allocation at program translation time. At this time, the address that each object will occupy at run-time is fixed. For this to be possible, it is necessary that both the number and the size of objects are known at translation time, and that only one instance of each variable exists during execution. E.g. FORTRAN variables satisfy these requirements.

FORTRAN program modules - main program and subroutines - are translated so that they can be stored in a contiguous area in memory, together with all locations for local variables and intermediate results (Fig. 1 - 4).

The life-time of a variable in FORTRAN begins with the loading of the program module it is contained in and ends with the completion of program execution. Thus, all functions of storage allocation to variables at run-time and subsequent freeing of storage are left to the loader and the operating system. The program proper does not have to provide any storage-management functions.

1.1.2 Dynamic storage allocation on a stack

1.1.2.1 Fixed size data

This storage allocation method is a simple extension of the static allocation scheme. Whereas static allocation is possible for languages where the scope of variables does not overlap,

the allocation approach to be discussed here is applicable to languages which permit nested scopes. ALGOL 60 is the classical example.

Nested scopes are represented statically by a nested block structure of a program (Fig. 1 - 5). At translation time, variables are bound to the innermost declaration of matching notation.

As we are dealing with fixed size data at the moment, the amount of storage required for the locations of a given block can be determined at compile time in the same way as in the case of static allocation.

However, the number of block instances at run-time cannot be known at translation time. A procedure in ALGOL 60 may call itself recursively, giving rise to a new block instance for each nested call.

As all scopes are entered and left in a last-in/first-out fashion, the classical mechanism used for storage allocation is a stack-regime: every time a new scope is entered, a new block of storage of fixed length, the activation record, is allocated on a stack. Upon scope-exit, this block is de-allocated as a whole by just resetting the stackpointer. The stack thus allows a continual reutilisation of the available space. The possible number of variables in an ALGOL program is therefore theoretically unlimited by physical storage for the associated locations.

1.1.2.2 Variable-size data

The two schemes discussed so far fix the size of any variable's location at program translation time and are therefore only applicable to data-sets with objects of fixed (or at least bounded) storage requirements.

There are two classes of modes, however, whose location size is unbounded, namely arrays with dynamic bounds and recursively defined records.

Example (a) Consider the following ALGOL 60 program segment:

```
....
real array NUMBERS[inint:inint]
....
```

Where "inint" is an integer function which returns an input integer number. The location size required for variable NUMBERS cannot be bounded at compile-time and can vary between one and MAXINT times the size of one real location. (MAXINT = the maximum integer value on the specific computer.)

Example (b) SNOBOL permits the concatenation of text strings such as

```
C = A B
```

The location size for C cannot be bounded, as it depends on A's and B's length.

The solution to this allocation problem is to separate the problem of *access* to a location from that of providing *space* for it.

At compile-time, only space for a pointer to the variable-size location is allocated. At run-time, the location is provided out of a pool of free storage, as soon as its size is known.

For blockstructured languages which prohibit changing the size of a location inside a block, the storage required for the variable-size locations can be allocated and freed together with the activation records. Thus, the two mechanisms for allocation of space for fixed and for variable-size locations can be combined into one stack-regime.

1.1.3 Dynamic storage allocation using a heap

When the size of a location can change every time a new value is assigned to a variable, as in the case of SNOBOL variables, storage for the location has to be allocated at least every time the new object requires more space than the old one. This then happens in an order which is entirely determined by the flow

of control in the program and is no longer a last-in/first-out process. The free storage for these allocations cannot be managed as a stack of continuous memory cells, but in a random-access fashion. Such a free storage area is called a *heap*. It consists of two types of cells, free cells and cells which are in use.

Example: SIMULA 67 has a mode text whose objects can be of (almost) any size. After the execution of the following program segment

```
text THIS  ,IS,IT;
     THIS :-COPY("FIRST");
     IS   :-COPY("SECOND");
     IT   :-COPY("THIRD");
     IS   :-COPY("LAST TIME");
```

the storage allocator has allocated space for four text locations of different size, but only three locations are in use.

Another language facility which requires a heap storage allocation regime is that of *name variables (pointers)* which may refer to *anonymous* (undeclared) *variables*. As already discussed, variables are also objects which may be referred to by names (viz. names of names).

ALGOL 68 is one language which permits the declaration of such name variables which can be made to refer to anonymous variables by assignation.

Example:

```
ref ref person friend = heap ref person;
    friend := heap person := (male, "Jim");
    friend := heap person := (female, "Joan");
```

In this example, a name variable is declared first, and its storage can be allocated by the compiler. Then, first one and then another anonymous variable is generated and assigned to "friend". After the elaboration of the third line of this program segment, only the last anonymous variable is still accessible, namely via the name variable "friend" which references it. The first one has been lost.

This simple example shows the consequences of introducing name variables and anonymous variables into a language:

(1) As anonymous variables can be generated during a program's elaboration in any number, the compiler cannot perform the storage allocation for these.

(2) As anonymous variables can be generated and lost in an order which cannot be deduced from the program text, i.e., independently of the entry to, and exit from blocks, procedures, etc., the storage allocation is no longer a last-in/first-out process.

Because of (1), any computation yielding a variable must lead to an explicit or implicit call to a storage allocation (constructor) function, such as heap in ALGOL 68.

The appropriate storage allocation regime for anonymous objects which may become inaccessible is the use of a heap.

Fig. 1 - 6 shows an example of how variables are generated and lost. a) is the LISP program which has been executed. The constructor function CONS was called three times, i.e., three variables were generated. b) shows the resulting list. It is obvious that only one of the three variables is referenceable via RESULT. The others cannot be reached anymore, they have become *garbage*. c) indicates what happened to the heap: three locations (addresses 100, 101, and 102) were taken out of the free storage pool. (Before the execution of the program, the FREE pointer was at address 100.)

In order not to lose the unreferenced locations in the heap, heap management must consist not only of storage allocation, but also storage reclamation.

There are two basic approaches to heap management, viz.

- leave it to the programmer, and
- provide it automatically ("behind the scenes", like the ALGOL 60 stack mechanism).

The automatic scheme can be implemented using either the reference count or the garbage collection method.

The next section addresses itself to a description of these techniques and to their application in various implementations of list-processing and general purpose languages.

1.2 SCHEMES FOR HEAP MANAGEMENT

1.2.1 Objective of storage reclamation

Storage reclamation in the heap may be done for any one of the following reasons:

a) Release of unreferenced storage locations. This is to combat the loss of memory cells.

b) Reduction of heap fragmentation. When locations of different size are allocated from the same heap, the allocator possibly cannot allocate a location of size x, although the total number of free cells in the heap is in excess of x. This is possible because of fragmentation of the free storage area.

c) Localising and linearising references. In a computer with large virtual memory, the danger of exhausting the free locations in the heap is very small. However, as the program execution proceeds, the active locations of the heap will be spread out over more and more pages. This leads to an increase in page swapping and consequently to degradation of system performance. This can be counteracted by compacting the active locations into the smallest number of pages and by linearising the records in the same process. Linearisation puts locations and locations referenced by them into one page, if possible. If anything is known about the most frequent access patterns - in LISP, lists are scanned in CDR direction much more frequently than in CAR direction - this information can be used in optimally placing locations in the relocation process required.

1.2.2 Heap organisation

A heap can be implemented as one or more contiguous areas in storage. For the purposes of this section, the physical storage

layout is of no significance. The heap will be considered as one area.

The heap consists of a set H of storage locations. It can be divided into three mutually disjoint subsets, viz.

- The *free space* F, which is the set of all storage locations which are known to be free.
- The *active space* A, which is the set of all cells which are referenced by program variables, and
- The *garbage* G, which is the set of all allocated, but inaccessible locations.

The relationship between these sets can be expressed in set-theoretical terms as

$$H = F \cup A \cup G \qquad (1.2.1)$$

and

$$F \cap A = \{\ \} \qquad (1.2.2)$$

$$F \cap G = \{\ \}$$

$$A \cap G = \{\ \}$$

The garbage and active space are of necessity intermixed. The free space can be either a list of free locations or one single location.

Per definition, the free space is addressable from one or more pointer variables, the active space is addressable from variables in the user program, and the garbage is not addressable from any variable.

1.2.3 Storage allocation under programmer control

1.2.3.1 Techniques

Under this regime, the programmer requests locations from the free portion of the heap and returns locations to it by explicit calls to utility routines in his program. The programmer can only return locations which are referenced by variables in his program. From the definition of G as the set of all unreferenceable

locations in the heap, it follows that this system only works if locations cannot become inaccessible, i.e., G = { }. Otherwise, the set G would grow during execution, and the storage allocator could run out of free locations to allocate, i.e., the program would finally choke in its own garbage.

On the other hand, the programmer does not have to return locations as soon as he does not require them any more. If he wants to, he may do wholesale de-allocation of a whole set of locations at once. Thus, at any one time, set A is the union of two subsets, viz. A_r, the set of all required locations, and A_g, the set of all cells which are referenced by the program, but no longer required. Thus, the basic relationship (1.2.1) for the heap becomes

$$H = A_r \cup A_g \cup F \qquad (1.2.3)$$

for the programmer-controlled allocation scheme.

The utilities required for this allocation system are fairly simple to implement. Addressability of locations is maintained by giving every location an overhead location which contains a link field and a size field giving the size of the total location. The free space is referenced by a pointer FREE SPACE. Initially, the whole of the heap is one contiguous block of free storage. Every time the program requires a location of a certain size, it requests it by calling a function ALLOCATE with the size of the location as an explicit (or implicit) parameter. The location is taken off the free block, and a pointer to it is returned to the calling program. Conversely, the program returns a location by a call to a procedure FREE with a pointer to the location to be returned as a parameter. The location, with its size field unchanged, is linked to the list of free locations.

Fig. 1 - 7 shows a mini-heap with 10 cells. (a) Initially, all of it is free space, and the size field in the overhead location has the value 10. The link field points to NIL. (b) ALLOCATE is called with a request for a location of four cells. The location is split off the free block and is referenced by user

program variable X. (c)FREE (X) is called, with (hopefully) the size field of X at 4. The returned location becomes the first in the list of free locations, linking to the size-6 block via its link field. (d)ALLOCATE (5) is called. The freelist is searched for a location with at least 5 cells. The first location does not qualify, but the second one does, so the five cells are taken out of this block, and a location of size 1 is left behind.

The simple example shows clearly the necessary searching of the freelist and splitting of blocks for a system supporting variable-size locations.

If all locations are of some constant size, this searching and splitting can be avoided, and the size field is no longer required. The freelist then functions as a linked stack, with locations being allocated from, and returned to, the top of the stack.

If, however, variable-size locations must be catered for, the strategies for searching and splitting are important to control both processor overhead and fragmentation of the freelist. The set of options available includes ordering the freelist by size or address, and deciding on which block to split by either the first-fit, best-fit, or worst-fit approach.

Knuth (Kn 68 pp.435-461) gives an extensive analysis and comparison of these various options and their consequences.

The programmer-controlled storage allocation system can be very efficient in terms of run-time. However, the price for this efficiency is that the burden of keeping track of references is entirely on the programmer. Returning locations to free store which are still referenced by program variables can be disastrous. Such errors are very difficult to find, as a program can continue running after the error apparently error-free, until the returned location is allocated again.

1.2.3.2 Programming languages implemented with this technique

The first widely used languages implemented using pro-

grammer-controlled storage allocation were IPL-V (Ne 60, Ne 65) and FLPL (Ge 60). They require the heap-regime because they are list-processing languages (FLPL is not quite a language, but a set of FORTRAN subroutines) and as such support the dynamic generation of names.

IPL and FLPL attempt to assist the programmer in remembering which locations are accessible and required by the program in the following way: each item belongs to one and only one list. If an item in a list belongs to that list, an indicator bit is set. Only items belonging to a list may be erased (freed) by that list.

A more recent system for programmer-controlled storage allocation is the AED free storage package (Ro 67). It is also the most extensive scheme of this type.

The AED package has three different strategies for free-space management, namely "special strategy", "regular strategy", and "garbage-collecting strategy". Free space is broken up (according to programmer specification) into several "free zones", each with its own strategy.

"Special" caters for locations ("beads" in AED terminology) which are all of the same size.

"Regular" maintains a list of free locations in the order of increasing size. It is to be used for zones in which many requests for locations of various sizes are made and which have a high frequency of allocation and freeing.

"Garbage-collecting" is also for providing locations of various sizes, but it maintains the freelist in the order of increasing addresses. It always merges freed locations which are contiguous into larger locations. The name "garbage-collecting" is an obvious misnomer: no garbage is collected, only fragmentation is reduced.

Ross claims that the AED package is more efficient in storage and processing time than the reference count or automatic garbage collection approaches. Even if this were so (and it is

questionable for both the "garbage collecting" and the "regular" strategies), the programmer efficiency must be considered: AED has 20 different procedures with up to 5 parameters each for establishing zones, freeing, and allocating. In addition, the programmer has a HELP procedure to supplement these facilities. This interface could almost be called baroque. It burdens the programmer with too much low-level detail and is likely to cause many programming errors which are difficult to detect.

The most important language implemented with programmer-controlled storage is PL/1.

PL/1 has six storage classes, viz.

1. Internal static
2. External static
3. Automatic
4. Internal controlled
5. External controlled
6. Based

Objects of classes 4 to 6 are references: the declaration of a variable of such class declares an unbounded set of references. Storage is allocated for these locations by calls to ALLOCATE.

For CONTROLLED "variables" (PL/1 terminology), only the reference created last is denoted by the identifier declared in the program and is thus a true variable. PL/1 controlled variables can be deleted by calls to FREE. The preceding reference (if any) then becomes a variable again, even if it was otherwise inaccessible.

CONTROLLED storage thus functions as a stack of references.

BASED does not have the underlying stack. Instead, each call to ALLOCATE causes a reference to be generated and assigned to a POINTER variable (these are references ranging over references).

1.2.4 Storage allocation using the reference-counter approach

1.2.4.1 Technique

This technique, and the garbage-collection technique described in the next section, provide for automatic reclamation of unused storage. "Behind the back" of the user program, unreferenced cells are returned to free space. The prerequisite for any such automatic system is that the reclamation system must be cognizant of the modes of the objects in the heap, at least to the extent that it can distinguish pointers from non-pointers and that it can find the size of all objects. This is only possible if the user program follows a strict discipline in using locations properly. E.g., pointers may not be stored temporarily in non-pointer locations.

The reference-counter approach is simply that each list has a special header-cell, i.e., a location which has an overhead in which a counter field is stored. This counter field is used for keeping track of how many locations reference this location. This is the "reference count". Initially, a program is given a fixed number of empty lists. The headers for these lists are used by the storage-management system to identify all locations in use by that program. To these lists, new locations may be added, and the resulting lists modified, so that list structures of any size and complexity are possible.

When a list is created and attached as a sublist to another list, its reference count is initialised to 1. Subsequently, whenever it is attached to another list, its reference count is incremented, and when it is detached, it is decremented.

Every time a list is detached from another list, the system checks if its reference count is down to zero. If yes, the list is no longer referenced by the user program and may be returned to free storage. Before this is done, however, the lists referenced by this list must have their reference count decreased, as they are being detached. The process of checking and freeing thus

proceeds recursively through the referenced sublists.

Fig. 1 - 8 shows an example of a list-structure with reference counts before (a) and after (b) a DELETE LIST operation. ROOT references the master list which, in addition to its header (identified by "H"-tag), has two elements which reference two other lists initially. To detach the sublist whose header D is, one executes DELETE LIST (D). This decreases the reference count (RC) of that list to 0, and the list can be returned to free storage. But first, the list's elements are traced. E and F both reference G, so G's reference count is decremented twice.

The main advantage of the reference count technique is that it works incrementally, i.e., storage reclamation is continuous, without peaks of processor activity for reclamation. At all times, for non-recursive lists the set G is empty, and

$$H = A \cup F \tag{1.2.4}$$

holds.

For recursive lists, however, this is not true: their reference count is never decreased to zero, and they become inaccessible garbage.

As

$$F = H \setminus (A \cup G) \tag{1.2.5}$$

the free space decreases in size, even if A maintains the same number of elements. This is a severe limitation of the reference counter technique.

Another disadvantage is the relatively large field required for the reference count; any undetected overflow may cause erroneous return to free space of the associated list. If overflow is detected, however, the list can never be reclaimed safely.

The processor time required for maintaining the reference count is proportional to the number of list attachments and detachments. So, even if no storage is being allocated or de-allocated, there is a continuous processor overhead.

Also, every pointer assignment operation requires access to both the location to which the assignment is made and the location referenced. In a virtual memory system, this introduces additional page swapping.

1.2.4.2 Programming languages implemented using this technique

The reference counter technique was first described by Collins (Co 60) in his article on the REFCO I free-storage package.

The most widely known system using the reference counter method is SLIP (We 63). SLIP is a list-processing package, embedded in FORTRAN, which supports two-way lists.

To overcome the method's disadvantage of not being able to recover recursive lists, Weizenbaum designed a SLIP-implementation combining both the reference-count and the garbage-collection techniques (We 69).

1.2.5 Storage allocation using garbage collection

1.2.5.1 Technique

Garbage collection has two functions:

- to separate the unreferenced cells (the garbage) from the active cells, and
- to reconstitute the free store.

Accordingly, garbage collection has two phases, marking/tracing and collecting.

Starting with (1.2.1), we see that the garbage can be identified by completely determining the set of active cells and performing the set difference

$$G := H \setminus F \setminus A \qquad (1.2.6)$$

The set of free locations is always known, be it in the form of a free list or of a contiguous free area. The set of active cells can only be found by tracing and marking all locations which can

be reached from one or more (fixed) root pointers, such as a stack of pointers into the heap. The mark is effectively a binary reference counter which, however, is only updated at garbage collection time.

Let F_B be the set of free locations at the start of the marking phase, G_B the set of garbage at that time, and F_A the set of free locations after the collection. Then the collection phase can be described to perform the operations

$$F_A := F_B \cup G_B$$

$$G_B := \{\ \}$$

$$H := A \cup F_A$$

Here is the whole garbage collection principle in a few lines of PASCAL:

```
"The simplest collector"
type LOCATION = LOW..HIGH;
                "lower/upper address of heap"
var  HEAP : array [LOCATION] of CELL;
     ALL, MARKED, FREE : set of LOCATION;
                 ROOT : LOCATION;
procedure MARK (PAR : LOCATION);
    begin if not PAR in MARKED then
            begin MARKED := MARKED + [PAR];
                  "add location to set of marked cells"
                  MARK (HEAP[PAR].SUCCESSOR)
                  "mark locations referenced from HEAP[PAR]"
            end
    end;
begin MARKED:=[ ];
      "the empty set"
      ALL:=[LOW..HIGH];
      "the set of all heap locations"
"The marking phase:"
      MARK(ROOT);
      "MARKED = A, the set of all active cells"
"The collecting phase:"
      FREE:= ALL - MARKED
      "F = H\ A = F_B ∪ G_B"
end
```

The marking algorithm does simple recursive tracing of the list referenced by ROOT, marking locations as they are encoun-

tered and then marking their successors in the list. (All pointers in cells are assumed to be pointers into the heap.)

In the absence of hardware to perform the set difference for large sets which is required in the collecting phase, this is simulated by linearly scanning through the heap and testing set membership in MARKED for each element individually.

Thus:

```
for I:= LOW to HIGH do
if not I in MARKED then FREE := FREE +[I]
```

The algorithm just shown results in an "Emmentaler" heap - full of (free) holes. If this is not desirable, e.g., because of the heap sharing a contiguous storage area with a stack, the free locations must be compacted in the desired direction. This requires relocation of some or all active cells and is done using the set of addresses of marked cells available after the marking phase.

This discussion of the garbage collection principle makes it sound very simple. But as usual, "the devil is in the detail". In the next sections, ways of implementing the various phases of a garbage-collector will be compared, and the inherent difficulties highlighted.

1.2.5.2 Tracing and marking

As was seen above, the whole active list-structure in the heap must be traced in order to establish set A, the set of all active cells. The set is represented by a boolean array, the garbage-collector marks.

These marks can either be associated with the individual cells making up a location (cellmarking) or with the location as a whole (location marking).

Cell marking is necessary when sub-locations of a location can be referenced. This is the case with ALGOL 68 where every sub-location in turn may be a location. Cell marking requires one bit of storage per active cell. This can be provided either in the

cell itself or in a separate bit-table.

In location marking, one marks a location as a whole. It is possible in languages without referenceable sub-locations (such as SIMULA 67 and LISP). Every reference to a record in such languages is to the beginning or another fixed point of the record's location and thus only complete records can become inaccessible, not individual fields in the records. Location marks can be stored either in a location's overhead cell or again in a separate bit-table.

The decision on where to store the garbage-collector marks has to take into account both storage space considerations and ease of access to the mark bits.

> Example: In computer systems with virtual memory, the inspection or setting of a mark-bit in a record causes a page-fault if the record is not in central ("real") memory. Storing the mark-bits in separate bit-tables rather than with the records themselves leads to greater address locality; therefore, less page-faults are caused by the inspection or setting of mark-bits. In the ideal case, the bit-tables can stay in real memory throughout a collection, and operations on them do not increase the page-fault rate at all.
>
> The higher the density (e.g., in mark-bits per word) of mark-bits in the tables and the lower their average density in the records in the heap, the greater is the increase gained in address locality.

Often, one can use otherwise unused bits in locations as mark bits.

> Example: Addresses on the DEC PDP-11 minicomputer are byte-addresses. Word-addresses on this computer are always even. Thus in a PDP-11 word there is space for a pointer to a word-sized location and a marking bit.

The task of tracing is equivalent to that of finding all nodes in a general directed graph. Marking and tracing are done together, i.e., as soon as a node is encountered for the first time, it is marked, and then its successor list is marked. Knuth (Kn 68) termed this "preorder" graph traversal. Marking has a double function, namely to indicate set-membership in the set of active cells and to prevent the graph-traversing tracing algorithm from entering

an infinite loop, when tracing a list-structure with cycles.

The tracing algorithm must be applied to every variable referring to a location in the heap. Also, non-declared references which are stored as intermediate results must be traced. An additional task of the tracing/marking phase is therefore to find all references to objects in the heap.

> Example: Hill (Hi 74) describes the storage administration of the Munich ALGOL 68 implementation in which all objects on the heap can be accessed by reference chains which originate in static data areas in the run-time stack. To find all static areas, all active blocks and procedure calls in the stack must be located by following the dynamic chain for procedure-calls. For each static area, the structure is known at compile time. This structure is represented in a storage allocation list which indicates whether an item is a pointer and if not, how long it is. Thus, all pointers into the heap can be found.

For tracing to be possible, the tracing algorithm must be able to determine the size and to find the pointers in an active location it has encountered. There are two basic approaches to storing this information (Wo 69). One is to store the information in the location itself. The tracing algorithm then has to interpret this information for each location as it proceeds through the list structure. This is called *interpretative tracing*.

> Example: Thorelli (Th 72) gives the following general interpretative tracing algorithm:

```
begin pointer p;
      AUX:= empty list;
      p:= root;
L:    mark node (p);
      case node type (p) of
          (Insert the elements of node (p)'s succes-
          sor list in AUX)
          ....
      end;
      while AUX ≠ empty list do
      begin p:= next element from AUX;
            if node (p) not marked then goto L
      end
end
```

This algorithm uses an auxiliary list AUX of addresses of nodes yet to be traced. Each node carries its type information, and this is interpreted in the case statement.

The other method is to generate at compile time as many recursive tracing routines as there are modes in the heap. The information on size and structure of locations is then contained in the tracing routines. For this to work, pointers must be typed, i.e., only refer to one type of object. This is the case in ALGOL 68 and SIMULA 67, but not in PL/1 where pointers can point to (almost) anything.

Example: Here is a simple-minded recursive tracing algorithm in SIMULA 67: two classes are defined, one containing two pointers, and the other one pointer. Each possesses a routine TRACE.

```
begin
   class A;
   begin ref (A)APTR;
         ref (B)BPTR;
         boolean MARKED;
         procedure TRACE;
            if not MARKED then
            begin MARKED:= true;
               if BPTR =/= none then BPTR.TRACE;
               if APTR =/= none then APTR.TRACE
            end
   end;

   class B;
   begin ref (A)APTR;
         integer I;
         boolean MARKED;
         procedure TRACE;
            if not MARKED then
            begin MARKED:= true;
               if APTR =/= none then APTR.TRACE
            end
   end;
   comment  let P be the base pointer to some list-structure containing A and B objects;
      P.TRACE
end
```

It can be seen that all information on where the pointers are and how many is written into the two (different) TRACE routines.

A major problem with any marking algorithm for garbage collection is that it has to operate under a storage constraint. When it is applied, i.e., when the garbage collector is called, storage is at a premium.

In tracing branched lists, the marking algorithm has to keep track of branch points encountered. For this, it requires an amount of storage which depends on the level of nesting in the list-structure, i.e., the maximum number of branch points in a path from the list root to any terminal node.

Recursive marking algorithms store this information in the form of procedure return points, one for each branch point whose successor list is being traced. Naturally, this is done on a stack, the size of which depends on the structure being traced.

Interpretative algorithms store the same information as sets of addresses of nodes which have been encountered, but whose sublists have not yet been traced. Unlike recursive tracing, interpretative tracing offers complete freedom concerning the order in which nodes are taken out of the set of untraced nodes. Different disciplines for this extraction lead to different marking orders.

> Example: If the set AUX in the interpretative algorithm shown above is managed as a stack (LIFO), it marks the nodes of a tree in depth-first order. Using queue discipline (FIFO), it marks in breadth-first order (Th 72).

Several methods have been developed which permit tracing of virtually any list structure which has any number of branch-points, despite the storage constraint.

Van der Mey (Me 71) has developed a LISP-variant with compact lists and variable-length atoms. Where possible, a cell's successor follows it directly in memory, thus making a pointer to it unnecessary. Only where this is not possible, a separate pointer, a link-cell, is used. As link-cells in turn may point to other link-cells, equivalent lists may contain differing numbers of link-cells. A list with the minimum possible number of link-cells is called compact.

The garbage-collector of this system relocates all referenced cells towards one end of the heap in order to regenerate non-fragmented free space. In doing so, it also compactifies non-compact lists by removing redundant link-cells. The marking algorithm works recursively, using the list-structure to store the required stack and therefore not requiring any additional memory space.

Similarly, all other recursive processing in van der Mey's system uses this list-stacking for storing branch-points. To mark modified nodes, three mark bits are used.

Schorr and Waite (Sc 67) describe an interpretative technique which moves down a list, marking each cell as it is encountered, and reverses the pointers followed. In the downward scan, all sublists are ignored. The tracing then proceeds backwards, following the reversed pointers (and restoring them to their original values) until a branch node is found. Then, for the sublist(s) referenced by this, the same process is repeated. Thus, no space overhead is incurred for storing the addresses of untraced branch-points. (This is a form of the general non-recursive algorithm given above, using stack discipline for the set AUX.)

Cheney (Ch 70) and Arnborg (Ar 72) have described non-recursive algorithms which perform tracing and at the same time copying of active cells into successive locations in a separate area. The copied sublists are then scanned in queue-discipline and any unmarked cells referenced by them are also marked and copied. Here, the copied structure itself serves to store the set of all untraced branch-points.

Van der Mey and van der Poel have developed a pragmatic method to limit the stack size required for recursive tracing. It makes use of the fact that in LISP, CDR chains are typically much longer than CAR chains, and that the maximum length of CAR chains in "normal", non-degenerate list structures is of the order 50 ... 100. Thus, their technique marks <u>iteratively</u> in CDR and <u>recursively</u> in CAR direction, requiring only a stack of 50 ... 100

elements for return addresses.

General marking algorithms have been described in great detail by Knuth (Kn 68).

Branquart (Br 71) and Wodon (Wo 71) give excellent discussions on tracing methods in ALGOL 68 garbage-collectors.

1.2.5.3 Collecting without relocation

After the marking phase, all active locations are marked. All unmarked locations are therefore not in use and must be reconstituted into free storage.

If locations of only one size are supported (as in LISP 1.5) and the heap resides in its own fixed memory area, the free-space can most economically be organised as a linked list. During the collection phase, all unmarked locations must first be found, and then added to the list.

Depending on how the mark bits are stored, the search for unmarked locations is a linear scan through either a bit-table or the heap. Whenever a free location is found, it is linked into the freelist.

1.2.5.4 Compactification

Systems which maintain free cells as a contiguous area in memory require compactifying garbage collection. In a system where a contiguous stack and a heap share a block in memory, garbage collection must be initiated when stack and heap bump into each other. Regardless of whether the system supports one or several sizes of locations, the heap must be compacted into a contiguous area away from the stack. Thus, a compaction phase instead of a collection phase follows marking.

In a system with variable-size locations, one may defer compactification until the blocks on the free list are all too small to satisfy a request for allocation of a location of given size, i.e., until fragmentation of the heap has become unacceptable.

Such systems have a collection phase and a separate store collapse phase which may be run less frequently than the collection phase. Again, when store compaction is run, it follows a marking phase. In one general type of compactifying algorithm, the compaction phase has three sub-phases which follow each other, viz.

- *planning* of new addresses for location to be moved to,
- *updating* of each pointer to the planned address of the object it references, and
- *relocating* of locations to their planned addresses.

Haddon and Waite (Ha 67) and Wegbreit (We 71) have described this type of compaction in great detail. Its greatest disadvantage is its high processing overhead. A main cause for this is the second tracing of all active cells which is required in updating all references.

A more time-economical type of compaction algorithm has been reported by Fenichel and Yochelson (Fe 69), Cheney (Ch 70), and Arnborg (Ar 72). It avoids the re-tracing sub-phase in the compaction phase by relocating locations as soon as they are encountered (during marking) for the first time.

The principle is quite simple. Instead of using one heap, this algorithm works with a circular ring-buffer of two semispaces, the current semispace, which is the currently active heap, and the future semispace, into which active cells are copied. When collection is complete, the two semispaces switch their roles.

For each base pointer into the heap, the first location referenced is copied to the address denoted by NEW (initially, this is the start of the future semispace), where NEW is the pointer to free space in the future semispace. Thus, the top level of a list has been copied. Then, a pointer SCAN scans through the pointers (if any) of the top level, copying referenced cells as encountered. When SCAN reaches NEW, the list referenced by a base pointer has been copied.

Each time a location is moved, the old location is changed

into a link cell to the new location. This permits updating moved locations by replacing references to the old location by the address of the moved record, using indirect addressing.

Arnborg has described this algorithm as follows:

```
begin SCAN:=NEW:=START[NEW SEMISPACE];
   for each base pointer do
       if MARKED[location referenced]
       then update base pointer
       else
       begin move location referenced to NEW;
          mark;
          make old location link cell to NEW;
          move NEW past copy;
          while NEW ≠ SCAN do
          begin
             for each pointer in location at SCAN do
             begin if not MARKED[location referenced]
                then begin move location referenced to NEW;
                           mark;
                           make old location link cell to NEW;
                           move NEW past copy
                     end;
                update pointer to address in link cell
             end;
             move SCAN to next location
          end
       end;
   swap semispaces
end;
```

Fig. 1 - 9 shows a snapshot of the two semispaces while a collection is going on. One can see that all locations between the start of the future semispace and SCAN have already had their pointers updated, i.e., they only reference locations in the future semispace. Between SCAN and NEW, however, all locations can only reference locations (copied or uncopied) in current semispace and therefore still require updating. This shows why "SCAN=NEW" is the terminating condition for the process.

This algorithm marks and copies in breadth-first order. It can therefore only linearise locations in the future semispace if each location contains at most one reference to another location.

Arnborg (Ar 72) gives a simple modification which permits

linearisation by priority ordering of references to be followed. Let there be n (disjoint) sets of references C_1 ... C_n. The linearisation strategy then is to first follow C_1 references, then C_2 references, etc. The SCAN pointer in the above algorithm is replaced by $SCAN_1$, $SCAN_2$, ... $SCAN_n$, i.e., one for each priority set of references. These pointers at all times are ordered:

$$SCAN_n \leq \ldots SCAN_2 \leq SCAN_1$$

At all times, references in C_i have been updated in all locations below $SCAN_i$, and the references in C_i of the location at $SCAN_i$ are updated when i is the highest number such that

$$NEW = SCAN_k, \text{ all } k < i.$$

The above algorithm has then only to be changed as follows:

```
....
while SCAN_n ≠ NEW do
begin i:= smallest i such that SCAN_i ≠ NEW;
      for all references of set C_i in location at SCAN_i do
      ....
      move SCAN_i to next location
end;
```

Applied to LISP, where one wishes to linearise in CDR direction, this modification would be

```
....
while SCANCAR ≠ NEW do
begin
     if SCANCDR ≠ NEW
     then follow CDR of location SCANCDR
     else follow CAR of location SCANCAR;
     ....
     move SCAN pointer treated to next location
end;
```

The price for this control over linearisation is the number of accesses to locations by the various $SCAN_i$, which is n times greater than the number of accesses by a single SCAN pointer.

Because of the single scan through the active locations in the heap, the linear scan through the future semispace, and the linearisation possible, this algorithm is very suitable for garbage

collectors in virtual memory systems.

1.2.5.5 Garbage collection overheads

For the best garbage collection algorithms, the time spent in each garbage collection can be approximated by

$$T_G = aA + b(H-A) \qquad (1.2.7)$$

where T_G = time for one collection
a = time taken to mark and unmark one active cell
A = number of active cells
b = time taken to unmark and collect one free cell
H = number of cells in heap

Let r be the proportion of the heap which is used, the heap utilisation factor ($A = rH$). Then a collection frees $H(1 - r)$ cells, and the collection time overhead per cell collected is

$$\frac{T_G}{(1 - r)H} = \frac{arH + b(1 - r)H}{(1 - r)H}$$

$$\frac{T_G}{(1 - r)H} = \frac{ar + b(1 - r)}{(1 - r)} \qquad (1.2.8)$$

This shows that the processing time overhead incurred by garbage collection is inversely proportional to the proportion of the heap which is recovered. The more the heap is filled up by active cells, the higher is the processing time per cell recovered, approaching infinity for larger and larger r-values.

Given that one has implemented the garbage collector in the most efficient way possible, i.e., with minimal a and b, the only control variable available to reduce the processing time overhead is the heap utilisation r.

As $r = A/H$, r can be reduced by either decreasing A or increasing H. A can be decreased by parsimony with storage used, e.g., by sharing common sub-structures. In a computer system in which memory can be obtained from a pool by calling a storage allocator at any time, or released to the pool, the size of H is a control variable with which the processing time overhead can be con-

trolled, albeit with consequences for the cost of memory allocated.

Hoare (Ho 74) proposes a simple system for optimising the total cost of a program's execution by requesting additional space for, or releasing superfluous memory from, the heap.

Measuring cost as the product of memory space and time, he shows that the minimum cost for a program with garbage collection is

$$C_{min} = (\sqrt{a} + \sqrt{(b+c)})^2 \, AN_G \qquad (1.2.9)$$

where a,b = defined by garbage-collector implementation (see 1.2.7),
A = storage used,
c = useful computing time per word collected,
N_G = total number of cells collected.

This minimum is achieved by allocating memory according to

$$H = A(1 + \sqrt{(a/(b+c))}) \qquad (1.2.10)$$

The cost-optimum heap utilisation is therefore

$$r_{opt} = \frac{1}{1 + \sqrt{(a/(b+c))}} \qquad (1.2.11)$$

Hoare proposes to estimate c by dividing the running time used by the program since the previous collection by the number of cells just collected. This permits a dynamic good estimate of the amount of heap storage to request (or release) until the next collection. Hoare's paper also shows that the cost function is quite shallow around the minimum for the practical spectrum of a, b, and c values.

Arnborg (Ar 74) reports on an actually implemented control algorithm for a virtual-memory implementation of SIMULA on the DEC-10 computer. This algorithm is different from Hoare's mainly in its assumptions about the garbage collection time function. Arnborg also discusses measurements on large sets of real-life programs in terms of means and variances of 11 controlling variables. According to the figures given, their variances are usually small (10% of the mean) for a given program. Based on the small predic-

tion errors the algorithm results in, Arnborg concludes that for practical purposes, it is close enough to the optimum possible without a priori knowledge of program behaviour.

1.2.5.6 Languages implemented with garbage collection

The list processing language LISP was the first language to be implemented with garbage collection. Bobrow (Bo 68) describes LISP storage management very clearly. The original implementations of LISP had only two heap modes, viz. list cell and atom. All storage allocation was done in fixed-length locations of one (on the IBM 7090) or more (two in the University of Delft LISP on DEC PDP-8) words. This small location size made the use of reference counters unattractive because of the large storage space overhead. Furthermore, LISP explicitly permits reentrant lists to be created by functions such as RPLACA or RPLACD.

List cells have two sub-locations in which pointers are stored. These pointers are the values of CAR and CDR, respectively, applied to the variable represented by the location. Atoms are also lists, but the locations they are composed of are of a different mode.

List cells do not have an explicit mode indicator, but atoms do; so list cells are just non-atom cells. In some LISP systems, the atom mode is denoted by a mark (the "atom mark") in the first cell of an atom list. In others, the address of the first cell indicates that it is an atom, because it lies in a range which is reserved for atom header cells.

The marking process can thus find the mode information it needs for tracing the active list structure. The other information needed for marking is the set of base registers containing pointers into the heap. In a basic LISP implementation, these are:

- pointers on the working stack, and
- reference variables in the program.

Program reference variables are fixed, and therefore no problem to find, but the working stack contains data of at least two modes, viz. references to lists and return addresses of procedure calls still to be completed. These must be denoted somehow for the marking process. One way to do this is to distinguish between the two modes by address values. This is easily possible if heap and machine code are each in contiguous areas in memory so that one comparison with e.g. the lower bound of the heap suffices.

The collection phase is very simple. It consists of a straight-forward sweep through the heap and the establishment of a new freelist. Where the heap and the stack are sharing one memory area, this may be complemented by a compaction phase.

Later variants of LISP, such as HISP (Me 70), have introduced variable-length locations for printnames, integers, program cells, etc. Therefore, these have been implemented with compactifying garbage-collectors.

Another special-purpose language implemented with garbage collection is SNOBOL, a language for string- and list-processing. Because of its requirement for variable-size locations, such as strings, it requires compaction.

Griswold (Gr 72) describes the implementation of SNOBOL 4 and also covers in some detail the storage management by garbage collection. (He calls it "storage regeneration", though; as he justifiably remarks, not the garbage is collected, but the active locations are, and the free locations are re-generated into one contiguous free area.)

Two general-purpose languages which depend on the garbage collection method for their implementation are SIMULA and ALGOL 68.

Arnborg (Ar 72) and Myhrhaug (My 70) give exhaustive accounts of the storage management requirements for SIMULA. For the purpose of this section, it suffices to remark that SIMULA supports variable-size heap locations (objects of mode text, i.e., character strings with records), access pointers, and objects of classes, and thus needs compactifying garbage collection. References to sub-

locations are not permitted, so that the use of overhead cells is very natural for implementing records in SIMULA.

The literature on ALGOL 68 and garbage collection is already very extensive: Branquart and Lewi (Br 70, Br 71), Marshall (Ma 71), Wodon (Wo 71), and Hill (Hi 74) give complete analyses of the ALGOL 68 storage problem and describe actual ALGOL 68 garbage collector implementations.

1.2.6 Comparison between heap management systems

As shown in the previous sections, there are basically three heap management schemes (programmer-controlled, reference count, and garbage collection) which can be categorised as non-automatic (first) or automatic (last two).

The main distinction between the two categories is the security provided to the programmer by the automatic schemes. Without any additional mental or programming effort on the programmer's side, he is protected from errors such as releasing locations which are still in use or not releasing storage and subsequently running out of storage. Thus, automatic storage management can give increased programmer efficiency compared to a non-automatic system.

One can compare the two automatic approaches in terms of *applicability* and *overhead*.

The reference count method cannot reclaim locations which are in a circular chain of references. Furthermore, it offers no solution to the fragmentation problem and is thus not applicable to systems with variable-size locations. The technique of garbage collection with compactification is therefore more generally applicable.

The space overhead for reference counters is one (large) counter location per location allocated. For single-word locations, e.g. of mode real or byte, the space overhead is of the order of 100% or more. Garbage collection requires only one mark-bit per location. Often, an otherwise unused bit can be used for

this.

The run-time overhead incurred by each of the automatic systems is best compared numerically. The processor time required to reclaim one cell via reference counts can be expressed as

$$T_{RC} = A_1 \times REF + A_2 \qquad (1.2.12)$$

where A_1 = time required for incrementing/decrementing the reference count,

REF= number of times the reference count has been incremented/decremented before reaching zero,

A_2 = time to return cell to free list.

It is interesting to note that Collins (Co 66), in a comparison between garbage collection and reference counts, erroneously neglected the first term and came to conclusions strongly favouring the reference count approach.

As shown in (1.2.2), the time to free one cell by garbage collection is

$$T_{GC} = ar \,/\, (1 - r) + b$$

The ratio between these two times is

$$\frac{T_{RC}}{T_{GC}} = \frac{A_1 \times REF + A_2}{ar/(1 - r) + b} \qquad (1.2.13)$$

This shows that the reference count technique gets better with increasing heap utilisation and decreasing attachment and detachment frequency, i.e., in situations where the list-structure is relatively static. The advantage of garbage collection lies in the control one has over T_{GC} through the heap utilisation r. This permits reduction of the run-time overhead by just increasing the heap-size.

Informally, one can sum up these observations in the statement that compactifying garbage collection is the storage management technique to be preferred for general purpose programming systems for larger computers.

1.3 THE REQUIREMENT FOR CONCURRENT GARBAGE COLLECTION

The comparison in 1.2.6 did neglect one important point, namely the distribution over time of the processor overhead. Whereas the reference count technique works incrementally, all processing must stop when the garbage-collector runs. The time required per collection can be of the order of seconds.

Languages implemented with garbage collection can therefore not be used for the programming of real-time systems, such as operating systems, which must respond to events such as interrupts in real time. Because of this, it is not quite appropriate to call ALGOL 68 in its present implementations a general-purpose language: if one uses these ALGOL 68 versions to program real-time systems, one must refrain from using any modes and operations which, implicitly or explicitly, lead to storage allocation on the heap. The resulting sub-set of ALGOL 68 is not even generally defined, as the use of the heap can vary from implementation to implementation.

On the other hand, ALGOL 68 contains several features, such as parallel clauses and semaphores, which are of direct applicability to programming real-time systems such as operating systems or process control applications. The present situation thus presents us with the paradox that a language's definition makes it highly relevant for the description and programming of real-time systems, but that one implementation detail, the sequential garbage-collector, precludes it from being used for this purpose. PL/1, on the other hand, has seen extensive use for real-time work, for example, as implementation language for MULTICS.

As the problem does not lie with the processor time overhead per se, but with its distribution, an obvious solution is to make the garbage-collector work incrementally, interleaved with the processing of the applications program. Effectively, the garbage-collector would operate *concurrently* with the applications program. The garbage-collector in such system would no longer be an integral part of the applications program, but be a separate task competing for the processor. It would be a background task

and could be activated e.g. when the applications program were waiting for the completion of input or output.

Taken to the extreme, the garbage collector could be implemented in hardware, e.g., microprogrammed, as a special-purpose processor which would be multiplexed amongst several applications programs. In combination with descriptors attached to memory words for mode denotation and with (to the applications programs) invisible and inaccessible mark bits, this could result in a machine architecture for efficient execution of real-time programs programmed in ALGOL 68, LISP, etc. Tanenbaum (Ta 73) proposes such a virtual machine for ALGOL 68. His proposal includes the idea of a hardware instruction "garbage collect", which would however execute sequentially with the applications program. On the other hand, he also suggests a way to implement parallel processing on the machine. Concurrent garbage collection would permit this architecture to be used for all types of ALGOL 68 programs, including real-time applications.

Chapter 2
CONCURRENT GARBAGE COLLECTION

The author has found three significant proposals for concurrent garbage collection in the literature.

The first one, by Bobrow (Bo 68), concerns a compactifying garbage collection scheme for LISP. Its operation is as follows: when only a small fraction (about 10%) of free space is left, the garbage-collector is started as a separate process running in parallel with the LISP program. The marking proceeds through the stack, from the bottom upwards. When the marking reaches the top (latest) entry on the stack, the marking phase is complete. During the marking phase, the LISP program must take special precautions whenever it applies a pointer-moving function such as RPLACA, RPLACD, and SET. If the changed cell has already been marked, the item entered must be pushed on to the stack for marking, unless it has already been marked, too.

During the list-structure relocation and list cell adjustment phases, any references must be checked to determine whether they refer to relocated cells. If this is so, the new address must be computed by indirection through the address stored at the reference.

Address adjustment and relocation of arrays must go on simultaneously. A special cell for the array currently being moved must be provided, and a special computation must be made to find an element of the array.

The incremental garbage-collector as described was planned (in 1968) for implementation in a LISP 1.5 system on an SDS 940. It was hoped that the system would support several users simultaneously, including some real-time processes.

This proposal at first glance appears sufficiently defined for implementation, as it offers ad-hoc solutions for problems resulting from simultaneous access to the heap by two processes.

One solution which could not work is that of putting references to unmarked items into the stack. If the LISP program pops items off the stack, these references would be lost. If their referenced cells still had not been marked, they would never be marked and would subsequently be collected.

It thus appears that Bobrow's idea was not sufficiently elaborated for successful implementation. In particular, not enough attention seems to have been given to synchronisation between the two co-operating processes.

The second reference to concurrent garbage collection is by Knuth (Kn 68, pp.412 and 594). He attributes the idea to Minsky and poses its solution as a research problem. The only significant suggestion for a solution is that the garbage collector must be started when there are still N cells in free space, N being sufficiently large so that the garbage collector can finish before the applications program runs out of storage. No other synchronisation is touched upon by Knuth.

The most recent and also most detailed proposal is by Barbacci et al. (Ba 71). He describes a compactifying parallel LISP garbage collector for a virtual memory computer, using Cheney's list copying algorithm. Barbacci recognises the problem of running out of storage in the future semispace. To prevent this, he proposes to adjust the timeslices for the LISP- and garbage-collection processes as follows so that the LISP process has minimal delays:

> "Let S = size of semispace
> Let UI= number of active (not garbage) cells at start of semispace flip
> Let GI= S - UI = number of recoverable (garbage) cells
> Let RG= (time quantum for collector) / (time quantum for LISP program)
> then
>
> $$RG_{OPTIMUM} = K \times GI / UI$$
>
> where K = (average time to copy an active cell) / (average time for LISP to obtain a new cell)"

Apart from the impossibility of knowing UI, this is not even the optimum relation, as will be shown in this thesis.

On the subject of synchronisation, Barbacci writes: "A conventional interrupt system is unsafe. Some LISP and GC routines cannot be suspended in the middle of their operation. When pointers are being inspected or modified, instead of interrupts, these routines will check the system timer and 'voluntarily' return control to the supervisor, perhaps getting a little more than their time-slice".

This covers the synchronisation problem only partially. In particular, it does not say which routines cannot be interrupted and why.

Barbacci also deals with the problem of applying pointer-moving functions to cells which have already been copied. The solution he offers in Figure 6.8, p.55, is however incorrect. In the case that A $\leq$ SCAN and .X not yet copied, the CAR(.A) or CDR(.A), respectively, never gets updated. After the end of the collection, it still points at X in the old semispace.

Summarising the three proposals, one can say that concurrent compactifying garbage collection for LISP appears to be feasible, given that the two co-operating processes can be properly synchronised and that LISP and garbage collector can ensure that all cells are properly collected. What is required is a design methodology which allows the development of solutions which can be proven to be correct. The goal of this chapter is to establish such design methodology.

2.1 THE PROBLEM

In systems with sequential garbage collection, the garbage collector can be regarded as a subroutine called by the storage allocator, i.e., as an integral part of the applications program:

```
....
    procedure ALLOCATOR (CELLS NEEDED : INTEGER);
              procedure COLLECTOR;
              begin
```

```
            ....
        end;
    begin
        ....
        if NRFREE < CELLS NEEDED then COLLECTOR;
        ....
    end;
....
```

The application program is a sequential program, because at any one time when it is active, only one of its statements is being executed. Thus, the program counter is either inside the procedure COLLECTOR, or it is outside it, somewhere else in the applications program (Fig. 2 - 1). The figure shows how garbage collection and problem processing (i.e., the execution of the actual application) alternate. Three complete collections are shown: t_a / t_b, t_c / t_d, and t_e / t_f. During these, problem processing ceases completely, i.e., the application program appears to be dead.

If the application program were controlling a real-time process, it could not react to any external events occurring in the collection periods t_a / t_b, t_c / t_d, or t_e / t_f before the completion of the garbage collection. In the worst case, the response time could be as long as the longest garbage collection could take. As this can be of the order of seconds, such real-time program could not be used to control processes with required response times in the millisecond range, e.g., I/O processes on a computer.

Concurrent execution of the problem program and the garbage-collector could eliminate the dead-times, if both executed each on a separate processor (multiprocessing), or could make the dead-times arbitrarily small, if both multiplexed the same processor (multiprogramming).

Fig. 2 - 2 shows the two possibilities. Several complete collections are shown.

Conceptually, there is no difference between letting two processes progress in true parallel by multiprocessing or in quasi-parallel by processor multiplexing. Of course, the rate of pro-

gress of a process getting only 1/n of a processor's time is only 1/n of what it could be if it had not to share it. Neglecting overheads for multiplexing, one can effectively map a system of processes which multiplex a processor into a system of processes which all execute on individual processors of adjusted power. Henceforth, no distinction shall therefore be made in this thesis between concurrency implemented by multi-processing or by processor multiplexing.

In the system with concurrent garbage collection, two new problems show up which did not exist in the sequential system. One is that of process co-ordination. Two processes share data, such as the heap and the stack of base-pointers. Their access to this data has to be ordered in time to ensure correct operation of the complete system. In a system with only one sequential process (e.g., a program with conventional garbage collection), such co-ordination is not required, as the effect of a sequential program is time-independent. Generally stated, in multi-process systems, there exists a control problem of permitting only those action sequences by the individual processes which result in safe, correct states.

The other new problem with concurrent garbage collection is the scheduling problem. The garbage-collector and the applications program are producer and consumer of free storage space, respectively (Fig. 2 - 3). While the garbage-collector is reclaiming storage, the applications program is consuming free storage. If garbage collection is done to prevent free space running out, the question is at which level of free space to activate the garbage collection process so that the applications program never has to wait for storage. In a conventional garbage collection system, this level can be zero, as the applications program stops consuming when the collector starts. The remainder of this chapter will analyse the problems of process control and scheduling and will develop solutions.

2.2 PROCESS CONTROL

So far in this thesis, the term "process" has been used quite loosely and without definition. For the following discussion of the process control problems, a clear definition of this term is however required.

Process is an abstraction of the activity of a processing unit running a program. Denning (De 71) defines this abstraction as follows:

> "With a given program we associate a sequence of 'actions' $a_1, a_2 \ldots a_k \ldots$ and a sequence of 'states' $s_o, s_1 \ldots s_k$. For $k \geq 1$, action a_k is a function of s_{k-1}, and s_k is the result of action a_k. The states represent conditions of a program's progress, with s_o the initial state. The sequence $a_1\ a_2 \ldots a_k \ldots$ is called an *action sequence*, and $s_o\ s_1 \ldots s_k \ldots$ a *computation*. A *process* is defined as a pair P = (s, p), where p is a program and s is a state."

Whenever the term "process"is used in the remainder of this thesis, the above definition is implied.

In a *system of concurrent processes*, an action sequence generated by the system consists of concatenations and merges of the action sequences of its component processes.

> Example: Let there be two concurrent processes P_1 and P_2 with the action sequences *abc* and *def*, respectively. Then these are a few of the system's action sequences:
>
> *abcdef*
> *abdefc*
> *adbecf*
> *defabc*
> .....

The *process control problem* is the problem of restricting the state sequences of a system of processes to those which lead to a correct computation. The sequence of states the system goes through on its way to a correct computation is a sequence of safe

states: The last state of a correct computation is a safe state. Its preceding state is therefore also always a safe state, given that all state transitions are deterministic, etc.

For disjoint processes, any interleaving of states is safe. This allows the scheduling algorithm in a multi-programming operating system complete freedom in scheduling independent jobs, as long as it can avoid resource conflicts between the jobs.

For interacting processes, this is no longer true. Each of a system of co-operating processes makes a contribution towards the goal state. The concurrent garbage collection system is such a system. It has two processes, the applications process and the garbage collection process, which share several variables, viz. the heap, the stack of base pointers referencing active lists, and the pointer to the free-list (Fig. 2 - 4). Some of the (macroscopic) actions of the applications process are requesting storage, pushing onto and popping off the stack, and pointer assignment. Some of the (also very macroscopic) actions of the garbage collection process are marking, collecting, and unmarking of cells. The action sequences of both must be so combined that they lead from safe states to other safe states. The goal state is the correct completion of the applications process. Both processes must co-operate towards that goal state. The possible number of action sequences is very large and cannot be enumerated. They must therefore be made safe by design.

The process control problem has four major aspects, viz. *determinacy, deadlock, mutual exclusion, and synchronisation* (see Denning (De 71)), all of which are to be considered for the applications process/garbage collection process system.

Let a system of processes have access to a set M of memory cells. Each process has associated with it a fixed set of input cells and a fixed set of output cells, both sets being subsets of M. Processes may read from their input cells and write into their output cells. Then a system of processes is called *determinate* if always the final contents of M depend uniquely on the initial

contents of M, regardless of the system action sequences between start state and goal state.

Deadlock is a state in a system of processes in which processes are forced to wait indefinitely for a resource to become free. Thus it can only occur in systems with limited resources in which processes can request resources while holding other resources, and in which the total demand may exceed the system's capacity, even though the demands of individual processes lie within the system's capacity. Such resources may be physical, such as tape drives, or more abstract, such as shared variables.

Mutual exclusion is the requirement that, at most, only one process have access to a shared resource at any one time. It is a necessary, but not generally sufficient condition for determinacy of a system of interacting processes.

In a system of co-operating processes, which performs a computation, some processes may have to wait until other processes have reached a certain state, e.g., delivered some information or executed some code a number of times. Thus, there may be a need for *synchronisation* between otherwise asynchronously progressing processes.

To design a concurrent garbage collection system which co-operates correctly with the applications program it supports, these four potential problem areas must be covered, and solutions be found for all of them.

The objective of the next section is to select the intellectual tools for this design process.

2.2.1 The conceptual tools

A well-structured program can be built and understood in detail by an intellectual effort proportional to its size.

This is one of the axioms of structured programming (see Dijkstra (Di 72) and Wirth (Wi 72)). The correctness of a well-structured program can be verified by postulating and proving *assertions* about its components and combining these assertions, until

assertions can be made about the complete program.

For example, the correct functioning of a sequential garbage collector can be verified as follows (Fig. 2 - 5). Let P denote an assertion before the execution of a statement (*antecedent*) and Q an assertion after the statement (*consequent*). For the garbage collector to work correctly, the consequent Q_G after the completion of the garbage collection statements must be

$Q_G \equiv$ All garbage cells and no others have been collected and the garbage collector terminates.

The function of the COLLECT statement is to collect all unmarked cells. The consequent assertion for COLLECT is therefore

$Q_C \equiv$ All unmarked cells and no others are collected and COLLECT terminates.

As $Q_C \equiv Q_G$, "garbage" $\equiv$ "unmarked", and the antecedent assertion for COLLECT is

$P_C \equiv$ All garbage cells and no others are unmarked and COLLECT is always started.

The function of MARK is to mark all live cells which it can reach, but no others. The consequent assertion for MARK is

$Q_M \equiv$ All live cells which can be reached by MARK and no other cells are marked.

As $P_C \equiv Q_M$, the assertion after MARK must be:

$Q_M \equiv$ All live cells and no other cells are marked and MARK terminates.

The necessary antecedent assertion for MARK is thus

$P_M \equiv$ All live cells can be reached by MARK.

The antecedent assertion for the garbage collector is then also:

$P_G \equiv$ All live cells can be reached by the garbage collector.

Once one has shown that the individual assertions for MARK and COLLECT always hold (this is done by making assertions

about their component statements), one can conclude that Q_G always holds, given that P_G holds upon entry to the garbage collector. Henceforth, one can then disregard the implementation details of the garbage collector, i.e., one can treat it as one component which can be trusted.

The question is whether this same approach is also feasible for concurrent processes, and if so, which tools are available.

Dijkstra (Di 68) suggested a well-structured representation of concurrent processes in a high-level programming language, namely the *concurrent statement*, in the form

cobegin S1; S2; .... Sn coend

This indicates that statements S1, S2, ... Sn can be executed concurrently. Only when all of them are completed is the following statement in the program started. Since the concurrent statement has a single starting point and a single completion point, it is well-suited to structured programming, i.e., it permits reducing assertions about simple statements to assertions about structured statements, and then reducing these in turn to assertions about a complete program (see also Brinch Hansen (Ha 73, Ha 73a)).

Assertions about concurrent statements and therefore their component statements must be time-independent. In particular, they must not make any assumptions about the relative speeds of the concurrent processes. This is generally possible for disjoint processes, i.e., systems in which no output cell of one process serves as input cell for another (requirement for determinacy).

Co-operating processes however have to communicate or interact through some common variable, i.e., they cannot be disjoint if they can access the common variable in parallel. To reestablish disjointness, one must make the access to the shared variable disjoint, i.e., serialise the accesses by permitting only one process at a time to use the variable. This is a problem of mutual exclusion.

Dijkstra (Di 68) developed the classical solution for this. He coined the term *critical region* for those statements in

a process program which operate on the shared variable. His solution to the mutual exclusion problem is based on *semaphores* on which P and V operations can be performed. Although it appears that all synchronisation problems can be solved by using P and V on general semaphores, these operations are too primitive to be of assistance in designing well-structured programs. Brinch Hansen (Ha 73, Ha 73a) argues that P and V are unsuitable language constructs for a high-level language because:

a) It is too easy for a programmer to cause time-dependent errors, even deadlock situations, with semaphores.

b) A language compiler cannot recognise errors such as a pair of P and V which are accidentally interchanged, or one or both missing, or a semaphore which is initialised incorrectly.

c) A compiler is unaware of the correspondence between a shared variable v and a semaphore *mutex* to gain access to it. Consequently, a compiler cannot protest if in a region with access control by *mutex*, reference is made to another shared variable x.

In general, he concludes, a compiler cannot give the programmer any assistance whatsoever in establishing critical regions correctly by means of semaphores, because semaphores are too general.

Brinch Hansen therefore suggests a structured notation for shared variables and critical regions as an extension to a well-structured language, PASCAL.

Let v be a variable of type (mode) T which is shared between several co-operating processes. Then this intention to share it is denoted by adding shared to its type in the variable's declaration:

var v : shared T;

Variables thus declared can only be used inside a critical region of the form

region v do S end

This clearly shows the programmer's intention and can be checked by the compiler. The region is a structured component in that it has one entry point and one exit.

For process communication, Brinch Hansen proposes the introduction of the *await* primitive to implement the *conditional critical regions* proposed by Hoare (Ho 72). Let B be a boolean expression which tests the permissibility of an operation carried out by a critical region S. Then Brinch Hansen's notation is

region v do begin await B; S end;

This specifies that S is not carried out until B is true.

To verify the correctness of a system of co-operating processes, one must define the permissible operations on the shared variable v. This can be done by some proposition I which specifies a property of v which must remain true at all times outside critical regions. Such a proposition is called an *invariant* for the shared variable. The proposition I must not refer to any variable which can be changed outside the critical regions for the shared variable. The condition for correct co-operation of the co-operative processes is that each process must leave the shared variable after accessing it in a state which satisfies I. The process may assume, i.e., may be programmed using the assumption, that I is true before each entry to one of its regions. If all processes co-operate correctly, and I is initially true, then on completion of the computation I will still be true.

Let P be a predicate which holds for the variables accessible to a process outside the critical region, and I an invariant on the shared variable v. After the completion of S, a result R holds for the former variables and I has been maintained. Thus, the axiomatic properties of the two constructs just discussed are

"P"
region v do "P&I" S "R&I" end;
"R"

and

```
"P"
region v do
begin "P&I"
      ....
      await B;
      "P&I&B"
      S
      "R&I&B"
end
"R"
```

The three structured notations presented in this section, the concurrent statement "cobegin S1; S2; ...; Sn coend", the critical region "region v do S end" associated with a shared variable, and the conditional critical region "region v do begin ... await B; S end" permit the design and correctness proof of a system with co-operating processes. The author of this thesis will therefore use PASCAL, extended with the constructs above, as the description and design language for the applications process/garbage collection process system. Appendix A summarises this extended PASCAL.

It must be stressed that no compiler for such extended PASCAL exists anywhere. The language is only used here for its descriptive power and the abstraction it permits. The main benefits accruing from this are that the intention of the algorithm designer is clearly visible in the structured notation possible, and that it permits assertions about individual program components to be combined into assertions on the complete program. Correctness proofs for the individual constructs are required only once. Afterwards, one can regard the correctness as an axiomatic property of the construct.

In the following section, the shared variables for the application/garbage collector system of processes will be defined, and the permissible operations on them developed.

2.2.2 Shared variables and invariants

For this section, a representative model of an applications process/garbage collector process system will be used. The applications process will be called "list-processor", which how-

ever is not meant to limit the applicability of the model to list-processing.

The model then is the following: list-processor und garbage-collector share the heap, a stack of basepointers referencing the active lists in the heap, and a pointer to the free-list. After initialisation, both list-processor and garbage-collector are activated. In the notation discussed in the preceding section, this can be described as follows:

```
type HEAPSPACE = .... "the range of all heap-addresses"
     STACKSPACE= .... "the range of values for the stack
                       pointer"
     POINTER   = .... "the range of values a cellpointer can
                       take on, including heapspace and NIL"
     CELL = record "the locations making up the heap"
            ....
            end;
var  HEAP : shared array[HEAPSPACE] of CELL;
                  "the heap"
     LSTACK:shared record
            STACK : array[STACKSPACE] of POINTER;
            TOP : STACKSPACE
            end;
     FREE : shared POINTER;
procedure INITIALISATION; ....
begin
     INITIALISATION;
     cobegin
        LISTPROCESSOR;
        GARBAGECOLLECTOR
     coend
end.
```

Fig. 2 - 6 shows this program's basic structure.

The representation for the heap as a shared array is only one possibility. Other possibilities are

```
HEAP : array[HEAPSPACE] of shared CELL,
```

or

```
HEAP : array[PAGESPACE] of shared PAGE,
```

where PAGE is defined as

```
PAGE = array[LINESPACE] of CELL.
```

The finer the partitioning of shared resources, the smaller is the probability of several processes competing for the same one. On the other hand, the space overhead increases, as with each shared variable a semaphore is associated. Also, elements of a resource array must always be requested and released in the same hierarchal order (e.g., in the order of increasing array indices) to avoid deadlock.

Assertions concerning termination

The verification rule for a concurrent statement is

```
"P1 & P2 & P3 ...& Pn"
cobegin
   S1;
   S2;
   S3;
   .
   .
   .
   Sn
coend
"R1 & R2 & R3 ...& Rn"
```

Thus, for the concurrent garbage collection system it is (Fig. 2 - 6):

```
INITIALISATION;
"P_L & P_G"
cobegin
   LISTPROCESSOR;
   GARBAGECOLLECTOR
coend
"R_L & R_G"
```

A necessary condition for this to be correct is

$R \equiv$ program terminates.

This is only possible if

$R_L \equiv$ list-processor terminates,

and

$R_G \equiv$ garbage-collector terminates.

For simplicity, it will be assumed that the computation

is correct if, at termination, either the processing task has been accomplished, or list storage has been exhausted.
Thus,

R ≡ (processing accomplished or storage exhausted) implies program terminates.

R_L ≡ (processing accomplished or storage exhausted) implies list-processor terminates.

R_G ≡ (processing accomplished or storage exhausted) implies garbage-collector terminates.

The condition "processing accomplished" is established by the list-processor, and the condition "storage exhausted" by the garbage-collector. When one process terminates because its termination condition has become true, the other process must also terminate, even if at that time it is waiting for another condition to become true.

The solution to this synchronisation problem is a shared boolean variable

```
EXIT : shared BOOLEAN
```

with the invariant

```
EXIT ≡ processing accomplished or storage
       exhausted
```

Invariant 1

Whenever one process is thus waiting for the other to make a condition B true, this must be done in the form

```
region v do
begin
     region EXIT do
     begin
        await B or EXIT;
        ....
     end;
     ....
end;
```

in order to avoid infinite waiting for a condition which may never

become true, because the other process terminated.

The conditional critical region as proposed by Brinch Hansen and Hoare, however, does not allow the terms in the condition associated with the await to refer to two shared variables, but restricts them to be components of the innermost shared variable. This appears to be a major flaw in the proposed language construct, as it does not readily permit the prevention of infinite waits (deadlocks) in conditional critical regions. One has to associate an error condition with each shared variable which is accessed in a conditional critical region:

```
var v : shared record
        A : ....
        ERROR : BOOLEAN
        end
```

This error flag can then be set by the process which is supposed to satisfy a condition on A, but cannot do it. If there are many shared variables used in conditional critical regions, this makes for tedious and errorprone programming.

For the purpose of this thesis, this flaw is of no real importance, as there are only two conditions to be considered, viz. "processing accomplished" and "storage exhausted". It will be shown that EXIT can be made a component of a shared record.

The general form of the concurrent statement can then be represented as

```
cobegin
   "LISTPROCESSOR"
   repeat
   ....
         region .... do
         begin
            EXIT:= EXIT or PROCESSING ACCOMPLISHED;
            LPEXIT:= EXIT
         end
   until LPEXIT;

   "GARBAGECOLLECTOR"
   repeat
   ....
```

```
            region .... do
            begin
               EXIT:= EXIT or STORAGE EXHAUSTED;
               GCEXIT:= EXIT
            end
         until GCEXIT
      coend
```

As long as the remainders (indicated by ellipses) of the two processes can terminate regardless of the value of EXIT, this construction terminates.

Assertions concerning the free space

List-processor and garbage-collector are in a consumer/producer relationship: all cells which the list-processor takes out of free space must have been put into free space by the garbage-collector.

Let NRCONSUMED = number of cells taken out of free space by list-processor, and

NRPRODUCED = number of cells put into free space by garbage-collector.

Then the invariant NRPRODUCED $\geq$ NRCONSUMED must hold at all times. An integer variable NRFREE with the meaning

NRFREE = NRPRODUCED - NRCONSUMED

is introduced. Then cells may only be taken out of free space if the following invariant can be maintained:

NRFREE = NRPRODUCED - NRCONSUMED $\geq$ 0

Invariant 2

This leads to the following necessary conditional critical region in every procedure which allocates storage from free space:

```
....
region FREE do
begin
     await NRFREE ≥ NRNEEDED;
     "NRFREE ≥ NRNEEDED"
```

```
            NRFREE:= NRFREE - NRNEEDED
            "invariant 2 holds"
            ....
        end;
        ....
```

NRNEEDED is the number of cells requested by the list-processor to be allocated from free space.

But here is the danger described in the previous section, viz. the garbage-collector may be unable to reclaim a sufficient ($\geq$ NRNEEDED) number of cells. To enable both processes to terminate, the following construction must be employed:

```
....
var FREESPACE : shared record
                    FREE : POINTER;
                    NRFREE, NRNEEDED : 0..MAXCELLS;
                    EXIT : BOOLEAN
                end;
....
    cobegin
      "LISTPROCESSOR"
          ....
          procedure STORAGEALLOCATOR;
          begin
          ....
              region FREESPACE do
              begin
                  await NRFREE ≥ NRNEEDED or EXIT;
                  if EXIT then TERMINATE
                  else
                  begin NRFREE:=NRFREE-NRNEEDED
                        ....
                  end
              end
          end;

      "GARBAGECOLLECTOR"
          ....
          "after collection"
          region FREESPACE do
          begin
              STORAGE EXHAUSTED:=NRFREE<NRNEEDED;
              EXIT:= STORAGE EXHAUSTED
              ....
          end
          ....
    coend
....
```

This allows the list-processor an orderly termination (TERMINATE), e.g., closing of files, saving of results, or issuing comprehensive diagnostics describing the circumstances which led to the exhaustion of storage. In systems which permit a running program to request additional storage, this facility could be used to recover and continue. The statement EXIT:= STORAGE EXHAUSTED would then not be used in the garbage collector.

Another invariant concerning NRFREE results from the requirement that the free space cannot exceed the heap capacity:

$$NRFREE = NRPRODUCED - NRCONSUMED \leq MAXCELLS$$

Invariant 3

Where: MAXCELLS = number of cells in the heap.

The phase of the garbage collector in which cells are added to the free space is the collection phase which collects all unmarked cells. For invariant 3 to hold after that phase, three requirements must be satisfied

- the collection phase must work correctly, i.e., only collect and count unmarked cells,
- before the start of the collection phase, $P \equiv UNMARKED + NRFREE \leq MAXCELLS$ must hold, and
- NRFREE = MAXCELLS after initialisation.

Thus, to verify that invariant 3 always holds, one must prove the correctness of the marking algorithm and of the collection phase.

The garbage collector shall only run, i.e., "produce", when the level of free cells has fallen below a given level which shall be called TRIGGER. This gives rise to the invariant

$$NRFREE < TRIGGER \text{ implies } GC\ REQUIRED$$

Invariant 4

This synchronising condition can be implemented by a conditional critical region:

```
"GARBAGECOLLECTOR"
repeat
   region FREESPACE do
   begin
        await NRFREE < TRIGGER or EXIT;
        if EXIT then TERMINATE
        else GCREQUIRED:= TRUE
             "invariant 4 holds"
        ....
   end;
   while GCREQUIRED do "garbage collect"
   ....
until GCEXIT;
```

There now exists a possibility of deadlock, a circular wait by list-processor and garbage-collector. The first could be waiting in "await NRFREE ≥ NRNEEDED or EXIT", while the latter was waiting in "await NRFREE < TRIGGER or EXIT".

The deadlock condition is

(NRFREE ≥ NRNEEDED or EXIT) ≡ FALSE and (NRFREE < TRIGGER or EXIT) ≡ FALSE.

For this to occur, not EXIT and NRFREE < NRNEEDED and NRFREE > TRIGGER ≡ TRUE. From this it follows that, to prevent deadlock, the following assertion must always hold:

EXIT or NRNEEDED ≤ TRIGGER

Invariant 5

This is simply assured by never permitting a request for storage of NRNEEDED > TRIGGER cells, e.g., by a test before executing the await in the list-processor

The last invariant related to the free space is that at all times outside the critical regions for FREESPACE, NRFREE is the number of free cells:

NRFREE = number of free cells

Invariant 6

This can be assured, after correct initialisation of NRFREE, by making the allocation of a cell and the decrementing of

NRFREE, and the collection of a cell and the incrementing into atomic operations, i.e., critical regions.

Invariants concerning the stack

As the stack contains the rootpointers to all live lists, it must be accessed by the garbage-collector during the marking phase. It is therefore a shared variable.

Fig. 2 - 7 gives the decomposition of the garbage-collector into its two main phases, MARK and COLLECT, together with assertions which ideally should hold. All live cells must be accessible to the garbage-collector during the marking phase. After the marking phase, all cells which are marked are live cells, and the collection phase collects all cells which are not marked, i.e., not live. These assertions always hold for correct sequential garbage-collectors.

Let TOP = stack pointer (only updated by list-processor)

ISTACK = pointer used by garbage-collector to scan through STACK.

The mark phase of a garbage-collector satisfying these assertions could then be described as

```
var LSTACK : shared record
                STACK : array[STACKSPACE]of POINTER;
                TOP : STACKSPACE
             end;
     ISTACK : STACKSPACE;
     ....
     "MARKING PHASE"
     region LSTACK do
     begin
          for ISTACK:=0 to TOP-1 do MARK (STACK[ISTACK])
          "cell marked ≡ cell live"
     end;
....
```

The correctness of the marking procedure MARK is assumed.

Several aspects of above algorithm make it unsuitable for concurrent garbage collection. First, the assertion "cell marked ≡ cell live" outside the critical region is meaningless. A live cell

is defined to be a cell which can be reached from STACK[o]..STACK [TOP-1],i.e., the assertion depends on a shared variable and should be an invariant. However, it is not invariant: an allocation to the list-processor of a cell from free space (which is unmarked) and a subsequent push of a pointer to that cell onto the stack could invalidate it.

There are basically two solutions to this problem:

a) allocate only marked cells after completion of the marking phase and during the collecting phase,

or

b) prevent allocation of cells from free space during the collecting phase.

The latter solution causes the list-processor to wait for storage during the collection phase and is therefore in principle inferior to the first.

Solution a) can be implemented by marking the free space before leaving the critical region for LSTACK:

```
....
region LSTACK do
begin
     for ISTACK:=0 to TOP-1 do MARK (STACK[ISTACK]);
     MARK FREESPACE
end;
```

Alternatively, one can increase the degree of possible concurrency by making the unmarked cells in free space inaccessible to the list-processor, and marking the free space outside the region for LSTACK:

```
var TEMP : record
              FREE : POINTER;
              NRFREE : 0..MAXCELLS
           end;
....
region LSTACK do
begin
....
     region FREESPACE do
     begin
        TEMP:=FREESPACE;
```

```
            NRFREE:=0;
            FREE:=NIL
        end
end;

"MARK FREESPACE"
while TEMP.NRFREE >0 do
region FREESPACE do
begin PTR:=TEMP.FREE;
      MARKED[PTR]:=TRUE;
      TEMP.FREE:=HEAP[TEMP.FREE].LINK;
      TEMP.NRFREE:=TEMP.NRFREE-1;
      HEAP[PTR].LINK:=FREE;
      FREE:=PTR
end;
```

Which of the two approaches is to be preferred depends on the time it takes to mark the free space. If this is short in relation to the desired response time of the system (e.g., in the case of the free space being one contiguous area with the marking bits in a separate bit-table), the first approach should be adequate.

As the list-processor may make marked cells inaccessible, the assertion P_C before the collecting phase must be weakened to:

$P_C \equiv$ "cell live implies cell marked"

As a consequence, a concurrent garbage-collector can therefore in general not collect all garbage, but only a subset of it, i.e., the unmarked cells. This will be discussed in detail later in this thesis.

The major weakness of the first algorithm of this section is that it does all marking inside the critical region for LSTACK. This implies that for the duration of the marking phase, no stack operations can be performed by the list-processor. As the marking phase is the longest of the garbage collection phases for higher levels of heap utilisation, this makes the algorithm useless for concurrent garbage collection.

A first attempt to repair this inadequacy is the following algorithm:

```
repeat
   region LSTACK do
   begin if ISTACK < TOP then
         begin NEXT:=STACK[ISTACK];
               ISTACK:=ISTACK+1
         end
         else "ISTACK = TOP" DONE:=TRUE
   end;
   if not DONE then MARK(NEXT)
until DONE;
```

NEXT is a variable which is not shared, and it can therefore be accessed outside the critical region. The marking is thus concurrent with the list-processor operations.

It is simple to demonstrate that the results of the algorithm above are time-dependent and that it can therefore not be generally correct.

As an example, assume that the garbage-collector and list-processor share three variables (X, Y, and Z) during the marking phase, that the list-processor assigns one variable to another, and the garbage-collector marks cells and their successors:

```
var X,Y,Z : shared CELL;
....
    cobegin
       "LISTPROCESSOR"
        HEAP[X].CDR:=Y;

       "GARBAGE COLLECTOR"
       "MARK (X)"
       if not MARKED[X]
       then begin MARKED[X]:=TRUE;
                  MARK(HEAP[X].CDR)
            end
   coend;
....
```

Even assuming that the two processes' actions are indivisible, the system can be shown to be non-determinate.

Fig. 2 - 8 shows the system schematically. The shared memory consists of X, Y, and Z. Each process performs one action (its *interpretation*) on these cells. Two possible computations are shown, each starting with all three cells unmarked and HEAP[X].CDR = Z. Computation I has the action sequence *ab* and

leads to the final state that X and HEAP[X].CDR are marked. This is correct. The second computation takes the sequence *ba* and yields a final state where X is marked and HEAP[X].CDR not. This is incorrect, as the subsequent collection phase will collect the unmarked cell Y, although it is referenced by the list-processor.

The simple example demonstrates that concurrent pointer assignments and list marking form a non-determinate system, as the result of a computation in the system is not independent of its action sequence.

Whether or not all cells which are garbage at the end of the marking phase are unmarked depends on the system action sequence during the marking phase.

Let *a* = a list-processor action which makes cell X garbage, and

b = the test and marking of X.

Then *ab* leaves X unmarked, and *ba* results in a garbage cell being marked.

One must therefore conclude that, in general, the system of concurrent processes list-processor/garbage-collector is non-determinate. The only possibility of making concurrent garbage collection work is by making at least the list-processor determinate, so that the results of its computation only depend on the initial content of its memory cells and not on the merged list-processor/garbage-collector action sequence.

A correct marking algorithm will trace and mark all cells in a list exactly once. When it encounters a cell for the second time, it finds that it is marked and therefore does not attempt to trace the sublists. If there are unmarked cells in these sublists, these will never be marked. Any action sequence "mark list X, insert an unmarked cell in list X" during the marking phase of the garbage-collector will thus lead to incorrect results for the list-processor's computation.

The only way of inserting an unmarked cell into an otherwise marked list is by pointer assignment, such as RPLACA and RPLACD

in LISP, and any reference assignment in general. One cannot stop pointer assignments during the marking phase without suspending execution of the list-processor and thus arriving at a sequential garbage-collector. The only other possibility is to design every pointer assignment operation in the list-processor so that it co-operates with the garbage-collector during the marking phase with the objective of completing the action sequence "allocation, marking" for each live location. This then results in a sequence of safe states, i.e., determinacy.

One way of doing this is by pushing references to such lists onto an auxiliary (shared) stack. The algorithm which ensures that each unmarked list attached to a marked list will be traced and marked is then quite simple:

```
var LSTACK : shared record
                STACK: ....
                ISTACK: ....
                TOP: ....
                AUX: array[AUXSPACE]of POINTER;
                AUXTOP:AUXSPACE;
                MARKING:BOOLEAN
             end;
    MARKED : shared array[HEAPSPACE]of BOOLEAN;
    ....
cobegin
    "LISTPROCESSOR"
    ....
        "POINTER ASSIGNMENT HEAP[REFREF].PTR:=REF"
        region LSTACK do
        begin
        ....
           if MARKING then
           region MARKED do
           begin
              if MARKED[REFREF]and not MARKED[REF]
              then PUSH (AUX,REF)
           end
        end;
```

```
"GARBAGE COLLECTOR"
....
"MARKING PHASE"
region LSTACK do MARKING:=TRUE;
DONE:=FALSE;
repeat
"A" region LSTACK do
    begin
        if AUXTOP>0 then
        begin NEXT:=AUX[AUXTOP];
              AUXTOP:=AUXTOP-1
        end
        else if ISTACK<TOP then
        begin NEXT:=STACK[ISTACK];
              ISTACK:=ISTACK+1
        end
        else
        begin DONE:=TRUE;
              MARKING:=FALSE;
              ISTACK:=0;
              MARK FREESPACE
        end
    end;
    if not DONE then MARK (NEXT)
until DONE;
....
coend
```

This works correctly, i.e., the marking phase will terminate with the assertion "all lists referenced from STACK[0]..STACK[TOP-1] are marked" holding, if all critical regions for LSTACK maintain the following invariants:

MARKING implies TOP-ISTACK = number of lists still to be traced from STACK

Invariant 7

MARKING implies AUXTOP = number of lists still to be traced from AUX

Invariant 8

Cell live ≡ cell referenced from STACK[0]...STACK[TOP-1]

Invariant 9

If the marking is done outside the critical region for LSTACK, invariants 7 and 8 are maintained by the garbage-collector during the marking phase if the list referenced by NEXT is considered as traced. This is permissible as, once NEXT is made to reference a candidate list for tracing and marking, this list is always traced, irrespective of the values of ISTACK, TOP, and AUXTOP. If the garbage-collector marks inside a critical region for LSTACK, an intermediate variable NEXT is not required, and invariants 7 and 8 must again be maintained by that region.

In either case, the essential point is that at point A in the algorithm shown above, exactly (TOP-ISTACK)+AUXTOP calls to procedure MARK are outstanding. Then, when zero calls are outstanding upon entry of region LSTACK, DONE and MARKING are correctly made true, i.e., the marking phase is terminated with all live lists marked.

The correctness of above algorithm will now be proved.

Let k = TOP-ISTACK, m = AUXTOP. The proposition to be proved is "Starting at point A of above algorithm with k lists to be traced from STACK and m lists from AUX, the algorithm will terminate with these traced and marked".

The proof of this proposition is done in three steps by proving propositions for m, k, and termination.

Proposition a: "The algorithm traces all m lists from AUX"

Proof:

The proof is by induction in m. For m = 0, it is true because no more lists are traced from AUX. For m+1, one list from AUX is marked (DONE=FALSE), and m lists from AUX remain to be marked. By induction, these lists will be marked. This completes the proof of proposition a.

Proposition b: "The algorithm traces all k lists from STACK"

Proof:

The proof is by induction in k. Lists from STACK are only traced when AUXTOP = 0. For k = 0, DONE is made true and the algorithm terminates correctly without marking another list. For k+1, the algorithm marks one

list from STACK, and k lists remain. According to the induction hypothesis, these k lists are marked correctly. This completes the proof of proposition b.

Proposition c: "The algorithm terminates"

Proof:

As the number of unmarked cells is finite and all can be reached from either STACK[ISTACK]..STACK[TOP-1]or AUX[0]..AUX[AUXTOP-1], they will be marked with a finite number of calls to MARK. Given that MARK cannot loop or wait indefinitely, the marking phase will terminate. This completes the proof.

A necessary consequence of invariant 9 is that the list-processor may have no "private" pointers into the heap outside critical regions for LSTACK. All intermediate results still required after exit from the region have to be pushed onto the shared stack.

The list-processor maintains invariants 7 and 8 by incrementing the stackpointers TOP and AUXTOP for every push onto the respective stack and by decrementing it for every pop off the stack.

However, the list-processor must do extra work during the marking phase. As a pop operation on STACK invalidates invariant 7 whenever TOP<ISTACK after decrementing TOP, the following code must be added to procedure POP:

```
"LISTPROCESSOR"
      ....
      function POP:POINTER;
      begin
         region LSTACK do
         begin TOP:=TOP-1;
               POP:=STACK[TOP];
"Added:"       if MARKING then
                  ISTACK:=MINIMUM (TOP, ISTACK)
         end
      end;
```

The only alternative solution to this is to stop TOP from taking on values smaller than ISTACK by the following construct:

```
function POP:POINTER;
begin
     region LSTACK do
     begin
          await ISTACK<TOP;
          TOP:=TOP-1
     ....
end
```

The obvious disadvantage of this alternative is that the list-processor may have to wait up to the time it takes to mark the lists referenced by AUX[0]....AUX[AUXTOP-1] and STACK[ISTACK]. This results in poorer, data-dependent real-time performance.

Invariants concerning the heap

Two invariants must be maintained by all critical regions for HEAP, viz.

> Any location belongs to one and only one of the sets ACCESSIBLE, FREESPACE, or GARBAGE.

Invariant 10

> The mode and the contents of a location do not conflict.

Invariant 11

Fig. 2 - 9 shows what might happen if invariant 10 is not maintained. In the collection phase, an unmarked location A has been found and the garbage-collector has just put it onto the front of the freelist, but not yet updated FREE (a). At this time, the list-processor requests a location and gets cell B, which is unlinked from the freelist (b). FREE is updated to C. Then, the garbage-collector moves FREE to A (c). The result is that the freelist from C onwards is lost and invariant 10 violated for cell B, as it belongs to sets ACCESSIBLE and FREESPACE. This would also violate invariant 6, as NRFREE ≠ number of cells in the freelist.

In general, all operations (always involving reference assignments) which might affect the membership of a location to one of these sets must be indivisible, i.e., they are critical regions for HEAP.

If storage allocation functions are separate from constructor functions in a list-processor, they must initialise the sub-location(s) of the location allocated to appropriate skip values in order to maintain invariant 11.

Invariant 11 is also important for languages which permit union modes, such as ALGOL 68. For these, the marking algorithm would fail if it were applied to a location at a time when its contents were of mode a, but its mode were still denoted as b. Any assignment to a variable of union mode and the updating of the actual mode indicator would have to be atomic, i.e., a critical region. This invariant is not necessitated by concurrent garbage collection. It is required for any language which supports parallel processes.

Shared variables and deadlock

The previous section has indicated that one requires several shared variables for concurrent garbage collection, and that the critical regions for these may sometimes have to be nested. This leads to the danger of deadlock between list-processor and garbage-collector, as the following example will illustrate:

```
cobegin
   "LISTPROCESSOR"
   ....
   region HEAP do
   begin
   ....
        region LSTACK do ....
   ....
   end
   ....

   "GARBAGE COLLECTOR"
   ....
   region LSTACK do
   begin
   ....
        region HEAP do ....
   ....
   end
   ....
coend
```

Assuming that the list-processor has entered region HEAP and the garbage-collector region LSTACK, neither can proceed into its next, nested critical region, as both regions are busy: the system is deadlocked by a circular wait condition.

Coffman et al. (Co 71) give the following necessary conditions for deadlock with respect to permanent resources such as shared variables:

1) Mutual exclusion: A resource can only be acquired by one process at a time.

2) Non-preemptive scheduling: A resource can only be released by the process which has acquired it.

3) Partial allocation: A process can acquire its resources piecemeal.

4) Circular waiting: The previous conditions permit concurrent processes to acquire part of their resources and enter a state where they wait indefinitely to acquire each other's resources.

To prevent deadlock, it is therefore sufficient to ensure that at least one of these conditions never holds.

As list-processor and garbage-collector are cooperating processes with shared variables, mutual exclusion (condition 1) is required for determinacy.

Preemptive scheduling is not possible efficiently once a process is inside a critical region. Furthermore, it would require cooperation by the process to be preempted, in order to bring the system into a safe state, i.e., all invariants holding. The only use of preemptive scheduling in the garbage-collector/list-processor system is thus by conditional critical regions. In general, condition 2 will therefore hold for concurrent garbage collection.

One can always prevent deadlock by complete allocation of all shared resources needed by a process in advance of its execution. For shared variables, this implies combining all variables into one shared record. In the case of the concurrent garbage collection system, this would be:

```
var ALL : shared record
          HEAP: ....
          LSTACK: ....
          MARKED: ....
          end;
```

Whenever exclusive access to one of the variables is required, this is done by

```
region ALL do
(access variable)
end;
```

The disadvantage of total allocation is however that exclusion for one variable means exclusion for all other shared variables as well. If much of the processing in both list-processor and garbage-collector is done on shared variables, this results in a low degree of parallelism between the processes. Still, this solution is efficient for list-processor/garbage-collector algorithms which frequently require access to most or all of their shared variables in a nested sequence. Here, it minimises the overhead connected with entry to and exit from regions.

Generally speaking, such approach leads to more frequent and longer blocking between cooperating processes and thus to worse real-time response. For multi-processor implementations, it also results in worsened processor utilisation, if processor time during blocking cannot be utilised elsewhere.

The negation of condition 4 can be done by a sequential ordering of requests to prevent circular waiting. It requires an algorithm which determinates whether, upon entering a critical region, it can be left again within a finite time. An example of such algorithm is the Banker's Algorithm, which is however time consuming.

Brinch Hansen gives a proof in (Ha 73) that deadlocks of nested critical regions can also be prevented by a hierarchal ordering of common variables v1, v2, ... vn and critical regions

```
region v1 do
region v2 do
....
region vn do
```

Each level in the hierarchy of variables consists of a finite number of variables. When a process has acquired variables at level j, it can only request variables at a higher level k. Variables acquired at level k are released before the variables acquired at a lower level j.

Given that in both list-processor and garbage-collector algorithms the nested accesses to shared variables follow the same sequence, this is an efficient way of deadlock prevention. Otherwise, it may be necessary to acquire access to a variable before it is needed, thus reducing the degree of parallelism possible.

2.3 SCHEDULING OF GARBAGE-COLLECTOR

In a sequential garbage collection system, the optimum scheduling strategy is to initiate garbage collection when the free space has been exhausted. This leads to minimum overhead per reclaimed cell, as can be shown as follows:

Let H = number of cells in the heap

r = heap utilisation

m = marking rate

k = collection rate

u = unmarking rate

R = number of garbage cells reclaimed=(1-r)H-NRFREE

Then, assuming equilibrium (r=const.), the relative cost in processor time per reclaimed garbage cell can be expressed as

$$\text{rel. cost} = \frac{rH/m + (1 - r)\ H/k + H/u}{R} \qquad (2.3.1)$$

This is minimized by $R_{max} = (1 - r)H$, i.e., by initiating garbage collection when the free space is exhausted.

If this same strategy were applied to a system with concurrent garbage collection, the list-processor would have to wait

for storage for up to rH/m units of time. Therefore, for concurrent operation throughout the garbage collection phase, the collector has to start while there is still a sufficient number of free cells in the free space to satisfy the list-processor's storage requests throughout the marking phase.

To determine the marking time and subsequently the level of free cells at which to start the garbage-collector (called TRIGGER in the preceding section), a model of marking, storage allocation, and garbage production will be developed in the following section.

2.3.1 Model

The following assumptions about the concurrent garbage collection system are made:

a) The system is in equilibrium, i.e., the heap utilisation r is constant.

b) The list-processor consumes free space cells at a constant rate c.

c) The list-processor makes cells inaccessible (garbage) at a constant rate g.

d) Accessible cells become inaccessible at random. The probability of an accessible cell becoming inaccessible is independent of whether it is marked or not.

e) All cells are of equal size.

The model of the heap composition during the marking phase can then be depicted as shown in Fig. 2 - 10. At any one time during marking, a cell can be in one of the following sets of cells in the heap:

1) Free space cells (occupancy F(t))

2) Unmarked live cells (occupancy U(t))

3) Marked live cells (occupancy M(t))

4) Marked garbage cells (occupancy L(t)), and

5) Unmarked garbage cells (occupancy G(t)).

The transition rates between the sets are:

1 to 2 : the storage allocation rate c

2 to 3 : the marking rate m

2 to 5 : a proportion of the garbage rate. Because of assumption d), this is

$g \times U(t) / (U(t) + M(t))$

3 to 4 : a proportion of the garbage rate. Because of d), this is

$g \times M(t) / (U(t) + M(t))$

Sets 2 and 3 together are the live cells, and sets 4 and 5 the garbage cells. Of the latter, only set 5 can be reclaimed by the collecting phase following this marking phase. The marking phase terminates when all live cells are marked:

$$U(T_m) = 0 \tag{2.3.2}$$

At this time,

$$F(0) - F(T_m) = cT_m \tag{2.3.3}$$

free cells have been consumed. Let F(0) = TRIGGER.
Then

$$TRIGGER - cT_m \geq 0 \tag{2.3.4}$$

must hold in order to avoid waiting for storage during the marking phase.

The occupancies of the five sets shall now be determined as functions of time.

F(t)

From Fig. 2 - 9:

$$\frac{dF}{dt} = -c \tag{2.3.5}$$

Integration of (2.3.5) gives the time-function for the number of free cells

$$F(t) = TRIGGER - ct \tag{2.3.6}$$

U(t)

From Fig. 2 - 9:

$$\frac{dU}{dt} = m - \frac{U}{U+M} g \qquad (2.3.7)$$

Because of assumption a) concerning equilibrium,

$$U+M = rH \qquad (2.3.8)$$

For the same reason,

$$U(O) = rH \qquad (2.3.9)$$

Solution of (2.3.7) with start condition (2.3.9) gives the number of unmarked live cells at time t:

$$U(t) = rH(1 - \frac{m}{g} (1-\exp(- \frac{g}{rH} t))) \qquad (2.3.10)$$

Equilibrium also presupposes

$$c = g \qquad (2.3.11)$$

Thus, (2.3.10) can also be expressed in the form

$$U(t) = rH(1 - \frac{m}{c} (1-\exp(- \frac{c}{rH} t))) \qquad (2.3.10a)$$

M(t)

From Fig. 2 - 9:

$$\frac{dM}{dt} = m - \frac{M}{U+M} g \qquad (2.3.12)$$

with (2.3.8),

$$\frac{dM}{dt} = m - \frac{g}{rH} M \qquad (2.3.13)$$

The initial value of M is

$$M(O) = O \qquad (2.3.14)$$

The initial time derivative of M is

$$M(t = O) = m \qquad (2.3.15)$$

The solution of differential equation (2.3.12) is therefore

$$M(t) = \frac{mrH}{g} (1-\exp(- \frac{g}{rH} t)) \qquad (2.3.16)$$

or, with (2.3.11)

$$M(t) = \frac{mrH}{c} (1-\exp(- \frac{c}{rH} t)) \qquad (2.3.16a)$$

This is the time function of the number of marked live cells.

L(t)

From Fig. 2 - 9:

$$\frac{dL}{dt} = \frac{M}{U+M} g \qquad (2.3.17)$$

$$\frac{dL}{dt} = \frac{g}{rH} M \qquad (2.3.18)$$

$$L(t) = \frac{g}{rH} \int M dt \qquad (2.3.19)$$

Inserting (2.3.16) one gets

$$L(t) = \frac{g}{rH} \int \frac{mrH}{g} (1-\exp(- \frac{g}{rH} t))dt \qquad (2.3.20)$$

The initial values of L and L' are

$$L (0) = 0 \qquad (2.3.21)$$

$$L'(0) = 0 \qquad (2.3.22)$$

The number of marked garbage cells then has the time function

$$L(t) = mt - \frac{mrH}{c} (1-\exp(- \frac{c}{rH} t)) \qquad (2.3.23)$$

G(t)

From Fig. 2 - 9:

$$\frac{dG}{dt} = \frac{U}{U+M} g \qquad (2.3.24)$$

$$\frac{dG}{dt} = \frac{g}{rH} U \qquad (2.3.25)$$

Inserting (2.3.10) and integration yield

$$G = \frac{g}{rH} \int (rH - \frac{mrH}{g} (1-\exp(- \frac{g}{rH} t)))dt \qquad (2.3.26)$$

The initial values for G and G' are:

$$G'(0) = g \qquad (2.3.27)$$

$$G (0) = (1 - r)H - F(0) - L(0) \qquad (2.3.28)$$

With (2.3.21):

$$G(0) = (1 - r)H - F(0) \tag{2.3.29}$$

The time function of unmarked garbage cells is then

$$G(t) = (g - m)t + \frac{mrH}{g}\,(1-\exp(-\frac{g}{rH}\,t)) + (1 - r)H - F(0) \tag{2.3.30}$$

All time functions have now been arrived at. It is already obvious from these that concurrent garbage collection does not work for all values of g, m, etc. These design parameters will be discussed in the next section, based on the functions just developed.

Here, the model shall only be used to determine the marking time and the level of free cells at which collection must be initiated so that the list-processor does not "starve" during the marking phase.

The marking time T_m is derived at by using (2.3.2) and (2.3.10):

$$rH(1 - \frac{m}{g}\,(1-\exp(-\frac{g}{rH}\,T_m))) = 0 \tag{2.3.31}$$

$$T_m = -\frac{rH}{g}\ln\,(1 - \frac{g}{m}) \tag{2.3.32}$$

$$T_m = -\frac{rH}{c}\ln\,(1 - \frac{c}{m}) \tag{2.3.32a}$$

This leads to the conclusion that the marking phase only terminates if $c < m$, i.e., if the marking rate is faster than the rate of taking new, unmarked cells from the free space.

Fig. 2 - 11 shows the marking time as a function of the ratio m/c and for a range of values for the heap utilisation r. The marking time is normalised by dividing by H/c, the time it takes the list-processor to consume H cells (the whole heap) from free space. As the marking rate approaches the consumption rate, the marking time goes to infinity. This is equivalent to a queuing system where the arrival rate (influx of unmarked cells from free space) approaches the service rate (marking). Naturally, the pool from which the arrivals come is limited (=TRIGGER for t=0).

This limits the marking times, but may also cause the list-processor to wait for storage, i.e., negate concurrency. All things being equal, concurrent garbage collection has longer marking times than sequential collection. The ratio of the respective marking times is

$$\frac{T_{m(CONC)}}{T_{m(SEQU)}} = \frac{-rH \ln(1 - c/m)/c}{rH/m} = -\frac{m}{c} \ln(1 - \frac{c}{m}) \qquad (2.3.33)$$

Fig. 2 - 12 shows this function plotted. With m/c approaching 1, it grows against infinity. The number of cells which the concurrent collector marks in excess of rH is marked garbage, i.e., unretrievable.

The level of free cells TRIGGER at which to initiate collection to prevent running out of storage during marking can be derived at by using (2.3.4):

$$\text{TRIGGER} \geq -rH \ln(1 - \frac{c}{m}) \qquad (2.3.34)$$

Fig. 2 - 13 shows TRIGGER (as a proportion of H) as a function of m/c for various levels of heap utilisation. As the marking rate approaches the cell consumption rate ($m/c \to 1$), the number of cells consumed during the marking phase goes asymptotically against infinity. As the speed ratio m/c goes against infinity, the situation for sequential garbage collection (TRIGGER = 0) is approached asymptotically. This is plausible, as c = 0 during marking for sequential garbage collection.

From this figure, it becomes obvious that concurrent garbage collection is only possible for a limited range of values for r and m/g without running out of storage during the marking phase: at most, (1 - r)H cells can be in free space when the heap utilisation is r. With increasing r or decreasing m/c, the number of cells required to satisfy all storage requests during the marking phase increases, whereas the number of cells available decreases. Function (2.3.34) thus only gives the cells required to get through the marking phase without waiting for storage; this storage may

however not be available.

The next section will develop the parameter space for which concurrent garbage collection without wait conditions for storage is possible.

2.3.2 Design parameters for concurrent garbage collection

For full concurrency, i.e., if the list-processor is never to wait for storage, two conditions must be satisfied (see Fig. 2 - 14):

$$F(T_m) \geq 0 \tag{2.3.35}$$

and

$$F(T_u) \geq F(0) = \text{TRIGGER} \tag{2.3.36}$$

If (2.3.36) is not satisfied, storage equilibrium is impossible, as less cells are reclaimed than are consumed during garbage collection. The system must therefore run out of storage. A sufficient condition for storage not running out is

$$D = F(T_u) - F(0) > 0 \tag{2.3.37}$$

where D = the number of cells by which the garbage collection has increased free space.

During the collection phase ($T_m \ldots T_c$), the time function of F is

$$F(t) = F(T_m) + (k - c)(t - T_m) \tag{2.3.38}$$

Taking the smallest possible value for $F(T_m)$, viz. $F(T_m) = 0$:

$$F(T_k) = (k - c)(T_k - T_m) \tag{2.3.39}$$

where k = collection rate (assumed to be constant).

Given k, $T_k - T_m$ is determined by the number of unmarked (thus retrievable) garbage cells at $t = T_m$:

$$F(T_c) = (k - c)\ G(T_m) / k = (1 - k/c)\ G(T_m) \tag{2.3.40}$$

The time function for F during the unmarking phase ($T_k \ldots T_u$) is:

$$F(t) = F(T_k) - c(t - T_k) \tag{2.3.41}$$

Assuming that always the whole heap is unmarked, the unmarking time $(T_u - T_k)$ is:

$$T_u - T_k = \frac{H}{u} \qquad (2.3.42)$$

where u = unmarking rate (assumed to be constant).

With (2.3.37), (2.3.40), and (2.3.42), the result is:

$$D = (1 - \frac{c}{k}) G(T_m) - \frac{c}{u} H - F(0) \qquad (2.3.43)$$

The function for G during marking is provided by (2.3.30), and T_m by (2.3.32).

Letting F(0) = TRIGGER and using (2.3.34), the result is:

$$D = H(1 - \frac{c}{u} - \frac{c}{k} + (1 + \frac{m}{c} - \frac{m}{k}) r \ln(1 - \frac{c}{m})) \qquad (2.3.44)$$

This is the fundamental design formula for dimensioning a concurrent garbage collection system. Under the assumptions made in section 2.3.1 for the heap model, it allows the determination of the parameter space for which fully concurrent garbage collection is possible, i.e., for which $D \geq 0$.

Fig. 2 - 15 shows the net gain of cells D, normalized by H, for a range of values for m/k, m/c, and c/u.
(Note: c/k = (c/m)(m/k).)

The faster the cell reclamation and unmarking rates, the less cells are consumed during the reclamation and unmarking phases. Fig. 2 - 15(a) shows the extreme case where unmarking and reclamation are infinitely faster than marking. For a heap utilisation of zero, the garbage collection then yields all unused cells. With increasing heap utilisation, the ratio m/c necessary to sustain concurrent collection increases rapidly.

Fig. 2 - 15(b) shows D for a system with a reclamation rate which is twice the marking rate, and an unmarking rate which is five times the cell consumption rate. Because of the cell consumption during the reclamation and unmarking phases, the free space available after a collection is significantly less than for the case shown in Fig. 2 - 15(a), i.e., zero unmarking and reclamation times. This leads to more frequent garbage collector activations and thus to more overheads.

Solving (2.3.44) for r, one can determine the maximum heap utilisation possible so that $D \geq 0$:

$$r_{max} = \frac{1 - c/k - c/u}{-(1 + m/c - m/k)\ln(1 - c/m)} \qquad (2.3.45)$$

For a heap utilisation of r_{max}, the garbage collection rate becomes infinite, i.e., the collector is always active.

Fig. 2 - 16(a) gives the heap utilisation possible for zero unmarking time. The top curve (zero reclamation time) shows the theoretical maximum for r_{max}, i.e., the best heap utilisation possible with concurrent garbage collection. For example, m/c=2 only permits a heap utilisation of up to about 50%. An increase in the reclamation time decreases r_{max} further, though not significantly.

Fig. 2 - 16(b) shows a case which is extremely (and probably unrealistically) bad, viz. an unmarking rate which is only twice the cell consumption rate. Even for m/c=10, the heap utilisation possible is well below 50%.

Formula (2.3.45) can be used to determine the size of the heap H_x required for an application program needing on average x cells and consuming cells at a rate c:

$$\frac{x}{H_x} \leq r_{max}$$

$$H_x \geq x/r_{max} \qquad (2.3.46)$$

2.4 OVERHEADS

There are two categories of overheads connected with concurrent garbage collection:

- Overheads due to synchronisation and means to make the system determinate, and
- overheads which are inherent in concurrent garbage collection.

To the first category belongs all processing time which is consumed by region entry and exit and by re-evaluating the await condition. Also, the additional work to be done by the list-processor during the marking phase and the space required to store references for the garbage-collector to unmarked sublists attached to marked lists fall under this category.

These overheads depend on the algorithms used by list-processor and garbage-collector, on the efficiency with which the critical regions are implemented, and on the data involved in the list-processor's computation. Therefore, no general functions for the processor and memory overheads of the first category will or can be offered here.

Overheads of the second category are the additional heap space needed to compensate for the unreclaimable garbage and the additional processor time caused by marking cells which then become garbage during the collection. These are intrinsic overheads which depend on rates such as cell consumption, marking, reclamation, and unmarking. In the following, the model developed in the previous sections will be used to estimate these overheads.

2.4.1 Heap space overhead

The unreclaimable (marked) garbage cells reduce the total useful heap space and thus represent a storage overhead. Their number is computed by evaluating L(t) for $t = T_m$ (Formulas 2.3.23 and 2.3.32):

$$L(T_m) = rH(- \frac{m}{c} \ln(1 - \frac{c}{m}) - 1) \qquad (2.3.47)$$

The space overhead is thus a proportion of the heap space in use and decreases asymptotically to zero with increasing speed ratio m/c.

2.4.2 Processor overhead

The greater processor overhead of concurrent compared to sequential garbage collection has two causes, viz. the higher

frequency of garbage collection and the longer duration of each individual collection.

The increase in frequency stems from the smaller number of cells reclaimed by concurrent garbage collection, and the greater processing time per collection from the marking of cells which later become garbage.

A given computation produces a fixed amount of garbage G. The number of collections needed to reclaim these cells is:

$$n = \frac{G}{R} \tag{2.3.48}$$

where R = number of cells reclaimed by one collection.

Thus, the ratio of concurrent to sequential collections for the same computation is:

$$\frac{n_c}{n_s} = \frac{R_s}{R_c} \tag{2.3.49}$$

where s = subscript for sequential collection
c = subscript for concurrent collection

The total processing overhead of concurrent relative to sequential collection is thus:

$$p = \frac{n_c T_c}{n_s T_s} \tag{2.3.50}$$

where T_c, T_s = processor time for one concurrent (resp. sequential) collection.

The processing time is the sum of the marking, reclamation, and unmarking times. From the model developed in section 2.3.1, the time for concurrent collection is derived as:

$$T_c = -\frac{rH}{c} \ln\left(1 - \frac{c}{m}\right) + \frac{H}{k}\left(1 + \frac{m}{c} r \ln\left(1 - \frac{c}{m}\right)\right) + \frac{H}{u}$$

$$= H\left(\frac{r}{c} \ln\left(1 - \frac{c}{m}\right)\left(\frac{m}{k} - 1\right) + \frac{1}{k} + \frac{1}{u}\right) \tag{2.3.51}$$

For one sequential collection, the processing time needed is:

$$T_s = \frac{rH}{m} + \frac{(1 - r)H}{k} + \frac{H}{u} \tag{2.3.52}$$

Concurrent collection retrieves unmarked garbage cells only.

With (2.3.6), (2.3.30), and (2.3.32), their number at the end of the marking phase is

$$R_c = G(t = T_m) + F(t = T_m) = H(1 + \frac{m}{c} r \ln(1 - \frac{c}{m})) \quad (2.3.53)$$

Sequential garbage collection always reclaims the total unused space:

$$R_s = H(1 - r) \quad (2.3.54)$$

By insertion in (2.3.50), the total relative processor overhead of concurrent garbage collection results:

$$p = \frac{(1 - r)(\ln(1 - c/m)(m/k - 1)\ r/c + 1/k + 1/u)}{(1 + r \ln(1 - c/m)\ m/c)(r/m + (1 - r)/k + 1/u)} \quad (2.3.55)$$

With increasing m/c, the relative overhead decreases asymptotically to 1. This shows the importance of a fast marking algorithm for the efficiency of concurrent relative to sequential garbage collection. Alternatively, this can be stated in the form that for a given concurrent garbage collection algorithm, the efficiency increases with decreasing cell consumption rate.

2.5 SUITABLE ALGORITHMS

For the design of a system with concurrent garbage collection, algorithms must be selected which

- can be implemented so that they satisfy the speed constraints discussed in the preceding section, and
- permit a high level of concurrency.

In principle, all garbage collection algorithms are suitable for concurrent operation. They do vary, however, in their real-time performance because of the length of time spent inside critical regions:

Example 1: Some garbage-collector algorithms modify

the list-structure of accessible lists by using it as a stack during the marking phase (e.g. Schorr and Waite, Sc 67). This results in invariant 9 not holding while a list is being traced, leading to long, data-dependent periods in which the garbage collector process stays inside the necessary critical region for HEAP.

Example 2: Some algorithms, such as that by Haddon and Waite (Ha 67) compactify storage as follows:

1. Mark all free storage.
2. Move all non-free elements to a compact block at one end of the heap, building up a relocation table at the same time.
3. Update all relocatable fields, using relocation table.

Between the time a location has been moved and the time all references to it have been updated, invariant 9 does not hold. This requires a very long (most of phases 2 and 3), unpredictable period inside the critical region for HEAP.

The most suitable algorithms for good real-time responsiveness are those which have to violate invariants only for short, data-independent periods. In general, these are algorithms which do not change anything in locations accessible to the list-processor.

For such algorithms, the longest period inside a critical region can be bounded by the amount of time required to process the largest single location and be independent of the length of lists or the complexity of the list-structure.

With respect to efficiency relative to sequential garbage collection, a fast marking algorithm is desirable, as it keeps the number of lost cells low. This puts e.g. the Cheney-Arnborg algorithm at a disadvantage, as it does marking and copying together.

To minimise the amount of heap-space required for full concurrency, also the collection and unmarking rates should be high. In particular on machines with large wordlength, this favours the separate array of marking bits over marking bits built into the locations themselves, as it permits much faster unmarking. For example on the CDC 6000/CYBER series with 60 bits/word, the unmarking rate using a bit array is at least 60 times that for individual mark bits.

Chapter 3
E X A M P L E S

In this chapter an attempt will be made to demonstrate the practical applicability of the approach. For this purpose, two list-processing systems have been chosen whose heap management requirements are assumed to embrace the range of requirements one encounters in practice. The first example supports a fixed number of LISP-like data-structures (list cells, atom headers, and atom cells) which can be catered for by fixed-length locations. For this, a non-relocating, concurrent garbage-collector has been developed which performs marking by recursion and iteration.

The second example supports variable-size locations with an arbitrary number of modes. The modes may contain any number of pointer and non-pointer fields. A concurrent, interpretative garbage-collector which compactifies the active part of the heap is shown in this example. It is based on the copying algorithms of Cheney (Ch 70) and Arnborg (Ar 72).

For both examples, shared variables are identified and invariants stated, and correctness proofs are offered.

The two algorithms for concurrent garbage collection have been simulated by sequential programs. For this purpose, they were programmed in PASCAL and executed on a CDC 6400 computer. The concurrency of the original algorithms was simulated by interleaving the execution of list-processor and garbage-collector. Both algorithms executed correctly for a large range of list structures. The main limitation of this test method lies in the complete mutual exclusion of both processes, i.e., the sequential simulation does not permit testing of critical regions for individual shared variables. Therefore, this thesis does not rely in any way on the simulation experiments performed or on their results. However, they have played an important role in the development of the concepts discussed.

3.1 A NON-RELOCATING CONCURRENT GARBAGE COLLECTION SYSTEM

3.1.1 Data structures supported

As this example is supposed to exemplify the heap management requirements of a LISP-like system, the following three modes are supported:

List cell: This is a pair of references (CAR, CDR) to objects either of mode "list cell" or of mode "atom".

Atom head: This contains a reference CDR to a property list and a reference PNMPTR to the atom's print-name.

Atom content: The print-name of the atom is stored in a list of locations of this mode. It has a character string STRING and a reference NEXT to the next part (if any) of the print-name.

Number atoms do not exist in this system.

The modes of cells are stored in boolean locations PNAME and ATOM in each object.

The address space of the heap is assumed to run from LOW to HIGH, with LOW > 0. Builtin atoms are assumed to have addresses < LOW. Only one builtin atom is explicitly used, namely NILP which denotes "no object". NILP is given address 0.

Fig. 3 - 1 shows objects of the three modes in which all fields are densely packed. How locations for these modes are implemented, however, is of no consequence for the algorithm. The only assumption made is that all objects are of the same size.

Fig. 3 - 2 gives an example of an atom in this system. The contents of the STRING fields are irrelevant for the garbage-collector, as they are non-pointer fields.

3.1.2 Heap management

The heap is assumed to consist of HIGH-LOW+1 fixed-size locations. No requirement for compaction is assumed. Thus, all free locations are allocated from and collected into a linked free list.

Two types of lists must be traced and marked by the marking algorithm of the garbage-collector, viz. lists made up of list cells and atom headers, and print-name lists. The latter are only referenced by atom header cells. In principle, all active lists could therefore be marked recursively by two routines, one for normal lists and one for atoms. As the maximum possible depth of recursion depends on the space available for the stack of procedure activation records, this would restrict the list size in CAR and CDR direction. A compromise solution used for LISP systems is to recurse only in CAR direction and to mark iteratively in CDR direction. This then only imposes a restriction on the maximum list depth in CAR direction. The algorithm used here adopts the same approach by marking normal lists and atoms iteratively in CDR and recursively in CAR direction.

A basic assumption made is that the list processor does not change a location's mode after allocation.

For keeping unmarked live lists attached to the marked list accessible for the marking algorithm, the same approach as described in chapter 2 is used, viz. an auxiliary stack.

3.1.3 Program description

The program structure of this example follows that shown in Fig. 3 - 3, i.e., list-processor and garbage-collector are programmed as procedures which are called inside the concurrent statement.

The declaration part for global constants, types, and variables (Fig. 3 - 4) is practically identical to what has been used and developed in chapter 2.

Fig. 3 - 5 identifies the structure of the list-processor. As the objective of this thesis is not to study list-processors, the list-processor program is not complete. Only a set of primitive operations (stack operations PUSH and POP, constructor functions CONS, pointer-moving function RPLACA, and non-pointer moving function CAR) has been selected from the ones in a LISP system which

is representative for the synchronisation requirements of a list-processing applications process in a concurrent garbage collection system.

The garbage-collector (Fig. 3 - 6) also follows the structure developed in chapter 3. The only significant addition is the marking algorithm (procedure MARK). It marks recursively in CAR direction (lines 40...43) and iteratively in CDR direction (lines 44...60). To mark the cells making up the print-name of an atom, MARK has a local procedure MARKATOM which marks iteratively. All iterative and recursive marking is done outside critical regions, so that each critical region is left within a short time which is independent of the list-structure. The only exception is the marking by iteration in MARKATOM which for reasons of simplicity only is done inside critical regions (lines 11...25). Assuming that the number of cells containing an atom print-name is small, this has no significant effect on the real-time responsiveness of the algorithm. Also, if necessary, it is easy to change MARKATOM so that all critical regions are left between marking individual cells.

Lines 96...98 of Fig. 3 - 6 show that this algorithm terminates as soon as it finds that there are no more free cells in the freelist at the end of the collection phase. Another reaction to this condition would be to request more memory from the operating system, or to attempt one more collection and stop only when this also fails to reclaim cells.

Procedure INITIALISATION (Fig. 3 - 7) establishes the free-list and unmarks the heap. It makes all invariants hold so that they hold upon entering the concurrent statement.

3.1.4 Correctness proofs

The correctness proof provided in chapter 2 for a concurrent garbage collection system such as this one assumes that

- invariants 1 to 11 hold,
- the marking algorithm is correct and terminates, and
- the system is deadlock-free.

In order to avoid repetition, the correctness of this algorithm shall be shown by demonstrating that all invariants are maintained, and that procedure MARK is correct and terminates. Also, it will be shown that the system cannot get into a deadlock condition.

3.1.4.1 Invariants

After procedure call INITIALISATION (Fig. 3 - 7)

Invariant 1

Assuming PROCESSING ACCOMPLISHED $\equiv$ FALSE, EXIT $\equiv$ NRFREE = 0 is correctly initialised after line 25.

Invariant 2

Holds if LOW $\leq$ HIGH which is ensured by the check in line 3.

Invariant 3

Holds if LOW $\leq$ HIGH, as NRFREE = HIGH-LOW+1 = MAXCELLS after line 17.

Invariant 4

Holds after line 29.

Invariant 5

In this system, all allocation is in locations with a fixed length of one cell. Thus, NRNEEDED is implied to be one, and the invariant holds after the check in line 29.

Invariant 6

Holds if LOW $\leq$ HIGH, as there are MAXCELLS cells in the heap. All cells are in the freelist and NRFREE = MAXCELLS after line 17.

Invariant 7

Holds after line 26.

Invariant 8

Holds after line 26.

Invariant 9

Holds after line 24, as there are no live cells and the stack is empty.

Invariant 10

After line 17, all cells belong to one set, viz. FREE. Holds.

Invariant 11

Holds, as there are no live cells.

Thus, it has been shown that all invariants hold upon entry to the concurrent statement.

After critical regions in list-processor (Fig. 3 - 5)

Invariant 1

Line 80: EXIT is only changed in value if "processing accomplished". (How this condition is determined shall be of no concern here.) Invariant is maintained.

Invariant 2

Lines 32...52: The conditional critical region ensures that NRFREE is only reduced by one when NRFREE > 0. Maintained.

Invariant 3

Lines 32...52: NRFREE is only decremented in this region, never incremented. Maintained.

Invariant 4

Not affected by list-processor.

Invariant 5

As all allocations are done in locations of one cell (NRNEEDED = 1), this invariant is maintained.

Invariant 6

Lines 32...52: Maintained by doing cell allocation and

decrementing of NRFREE as one atomic operation.

Invariant 7

Lines 7...8: All PUSH operations add a list to be traced to the stack. Accordingly, ST.TOP-ISTACK is increased by one by incrementing ST.TOP.

Lines 15...17: All POP operations remove a list to be traced from the stack, unless the list already had been traced (TOP < ISTACK). Correctly, the difference ST.TOP-ISTACK is decremented only if an untraced list is popped off the stack. (No error-checking code to prevent stack overflow or POP operations on an empty stack are shown, i.e., all stack operations are assumed to be correct.)

Invariant 8

Lines 70...71: Whenever a pointer to a list to be traced is pushed onto the auxiliary stack, AUX.TOP is incremented.

Invariant 9

Lines 23...28, 36...51, 57...74: Popping off of parameters from, and pushing results onto, ST.STACK is always done inside one critical region for LOC.

Invariant 10

Lines 36...51: Cells are taken out of the set of free cells and put into the set of live cells in one indivisible operation. Invariant maintained.

As all LISP selector functions (e.g. CAR) or replacement functions (e.g. RPLACA) can put cells into the set of garbage cells, no local variables may be used to reference cells outside the critical region LOC in which the function was applied. This has been adhered to.

Invariant 11

Lines 41...49: CONS does change the mode of the allocated cell. The CAR and CDR fields are assigned to in accordance with

mode "list cell". (Any other constructor function, such as RATOM in LISP, would also have to set the cell's mode indicator and assign to its fields in one indivisible operation, i.e., in a region for HEAP.)

After critical regions in garbage collector (Fig. 3 - 6)

Invariant 1

Line 96: Only at this line is EXIT assigned to. After line 98, EXIT ≡ EXIT or (NRFREE = 0), i.e., EXIT ≡ "processing accomplished" or "storage exhausted". Maintained.

Invariant 2

Lines 65...69: After this region, NRFREE = 0. Maintained.

Lines 74...83: This only increments NRFREE. Maintained.

Lines 90...94: Maintained for same reason.

Invariant 3

Lines 74...83: In this region, no new cells are added to the freelist. Maintained.

Lines 90...94: Only unmarked cells are added to the freelist. Each unmarked cell is only added once to the freelist. Given that the marking algorithm is correct, this critical region maintains the invariant.

Invariant 4

Lines 106...109: This is ensured by the critical region which is conditional on NRFREE < TRIGGER.

Invariant 5

As neither TRIGGER nor NRNEEDED are changed inside the garbage-collector, this invariant is unaffected.

Invariant 6

Lines 65...69, 74...83, 90...94: All operations involving adding cells to the freelist or taking cells from it and the associ-

ated updating of NRFREE are indivisible operations. Maintained.

Invariant 7

Lines 111...130: Whenever a list to be traced from the stack has been selected (NEXT:=ST.STACK[ISTACK]), the difference ST.TOP-ISTACK is correctly decremented by incrementing ISTACK.

Invariant 8

After selection of a list to be traced from the auxiliary stack (line 118), AUX.TOP gives the proper number of lists still to be traced from AUX.

Invariant 9

Not affected by garbage-collector.

Invariant 10

Lines 65...69, 74...83, 90...94: All deletions from and additions to the freelist are done as atomic operations. Maintained.

Invariant 11

Not affected by the garbage-collector.

3.1.4.2 Outstanding correctness proofs

Having shown that all invariants are maintained, it suffices to prove the following propositions in order to complete the correctness proof of the system:

Proposition a: "MARK terminates"

Proof a:

Only unmarked cells are marked. The number of cells is finite. Every time NOTYET = TRUE (Fig. 3 - 6, line 33), a cell is marked. Thus, line 36 can only be passed a finite number of times.
It only remains to be shown that neither the while loop in MARK nor that in MARKATOM will loop indefinitely, i.e., that lines 49 and 18 are only passed a finite number of times. This can be seen by inspection, since one previously unmarked node is marked per loop iteration.
This completes proof a.

Proposition b: "MARK only marks cells which have been live"

Proof b:

A cell is live when it can be reached from ST.STACK[0]... ST.STACK[TOP-1]. Two different calls MARK (NEXT) must be considered, viz. NEXT = ST.STACK[ISTACK], and NEXT = AUX.STACK[TOP]:

First case: Let n be a marked cell in the list referenced by ST.STACK[ISTACK]. Suppose it is marked as m^{th} cell in this list, $m \geq 1$. The proof is by induction in m. For m = 1, n = HEAP[ST.STACK[NEXT]] and thus n can be reached from ST.STACK[NEXT].
For m > 1, n must be a successor to a node n' which was marked prior to n. According to the induction hypothesis, n' can be reached from ST.STACK[NEXT] and, consequently, so can n.

Second case: Only references to cells which have been referenced from ST.STACK[0]...ST.STACK[TOP-1] are pushed onto AUX (see Fig. 3 - 5, lines 63...72). Thus, the second case reduces to the first case.

This completes proof b.

Proposition c: "MARK completely marks an unmarked list"

Proof c:

Assume that MARK has run to completion on the originally unmarked list referenced by NEXT and that n is a cell reachable from NEXT which is not marked.
Let n_0 = HEAP[NEXT], n_1, n_2 .... n_p = n be the cells along a path from cell n_0 to n. Let n_s be the first of those cells which is unmarked, $s \geq 1$.
The proof shall be done by case analysis and induction.

Assume that n_{s-1} is marked by MARKATOM. Then the while loop (Fig. 3 - 6, lines 17...23) is only left if n_s is marked, i.e., MARK completes only with n_s marked. This contradicts the original assumption.

Assume that n_{s-1} is marked by MARK at line 33. Then n_s is marked at line 42, if it follows n_{s-1} in CAR direction, or at line 49 if it follows in CDR direction. This contradicts the original assumption.

Assume that n_{s-1} is marked at line 49. Then n_s is marked at line 54 if it succeeds n_{s-1} in CAR direction, or at line 49 if in CDR. This contradicts the original assumption.

As all possible cases contradict the original assumption, the assumption must be rejected as false for s if it is false for s-1. As it is false for s = 1 (the call to MARK(NEXT) marks n_0 = HEAP[NEXT] at line 33), it is false for all s. This completes the proof.

Proposition d: "The system is deadlock-free"

Proof d:

As there can be no circular waiting in the conditional critical regions, given invariant 5 is maintained, it suffices to show that all shared variables are accessed in the same hierarchal order. This can be done by inspection:

List-processor (Fig. 3 - 5):

Lines 23...26: LOC → HEAP

Lines 32...42: FREELIST → LOC → HEAP

Lines 57...61: LOC → HEAP → MARKED

Garbage-collector (Fig. 3 - 6):

Lines 11...13: HEAP → MARKED

Lines 74...77: FREELIST → HEAP

Thus the hierarchy of shared variables which has been adhered to is FREELIST → LOC → HEAP → MARKED.

Consequently, the system is deadlock-free.

3.2 A CONCURRENT COMPACTIFYING GARBAGE COLLECTION SYSTEM

3.2.1 Data structures supported

This system supports locations with a variable number of cells. The number of cells in a location depends on its mode. Also, whether or not a cell contains a reference is determined by the location's mode.

References in this system may, but don't have to be typed. The garbage-collector algorithm only has to be able to find all references in a location, not their modes. Minimal assumptions have been made as to where the information distinguishing reference cells from non-reference cells is stored. The garbage-collector requires the mode indicator to be stored in a location and also expects the existence of a predicate (APOINTER) which, given the mode of a location and a cell's offset from the start of the location, identifies the cell as a pointer or a non-pointer sub-location.

As the algorithm is described in the following, it assumes the mode of a location to be stored in the first cell of a location. Any references to sub-locations can thus be visualised as consisting of two parts, viz. a reference to the start of a location, and the offset of the sub-location from the start.

For reasons of efficiency, only multi-cell locations have an overhead cell, with one field giving the location's mode and one giving its size, including the overhead. These configurations are representative for structured and multiple values.

Locations consisting of only one cell are also supported. Typically, they would be used for primitive modes (real, integer, etc.) and sub-ranges thereof. These locations also carry a mode field, but no length information.

One special mode always required and therefore predefined is that of "LINKCELL". The garbage-collector converts the mode of every location it has copied into LINKCELL, storing a reference to the copy in its field LINK. If an object of this mode is encoun-

tered by the list-processor when applying a function, the function is applied to the location referenced by LINK. Thus, it just causes one level of indirection.

The other use of this mode is made by the list-processor to provide access paths for the garbage-collector to otherwise (possibly) inaccessible, but live cells.

Fig. 3 - 8 shows the structure of locations supported by this system.

In principle, this system can support the heap management requirements of languages such as ALGOL 68. However, as presented in the following, it does not cater for the problem posed by the marking of sub-values as discussed e.g. by Branquart et al. (Br 71, pp.228-229). Basically, this problem arises when only a slice of an array is live, not the whole array. For reasons of access, the inaccessible gaps (garbage) in the array must be preserved in the compactification phase, but if the array is of mode "[,] ref ...", only the references from the accessible cells must be traced. Branquart offers a solution to this problem, and it appears that the garbage-collector here presented could easily be extended by using Branquart's or similar proposals.

3.2.2 Heap management

As the system must cater for variable-length locations, compactifying garbage collection is required. The algorithm selected is of the Cheney-Arnborg type already mentioned elsewhere in this thesis. The heap is assumed to consist of one contiguous address space from LOW to HIGH, i.e., with HIGH - LOW + 1 cells. This space is split into two semi-spaces (SS1 and SS2) of equal size (Fig. 3 - 9). When there is no garbage collection going on, only one of the semi-spaces (LPUSES) is active and being used by the list-processor.

All allocation of free cells takes place from a contiguous area delimited by a pointer FREE and the end of the semi-space. FREE is updated after every allocation and moves towards the end of

the semi-space.

When the free space contains less than TRIGGER cells, the garbage-collector becomes active. Starting from the stack of pointers into the heap, it copies all active locations contiguously into a second semi-space (GCUSES), converting all copied locations into LINKCELL pointers to their copies. During the garbage collection, the active heap is gradually transferred to the semi-space GCUSES, i.e., the list-processor works more and more in GCUSES. But all the time, storage is allocated in LPUSES. When all active locations have been copied, and all references have been updated so that there are no more references into semi-space LPUSES, the two semi-spaces change their roles, i.e., LPUSES becomes GCUSES, and vice versa.

In practice, there is no need for the two semi-spaces to be available all the time. The semi-space GCUSES could be requested from the operating system only when required for copying. Also, the semi-spaces do not have to be physically contiguous in memory. All that is required by the system is that from each reference, the semi-space it points into can be identified.

As in all concurrent garbage collection systems, the list-processor must maintain accessibility to the garbage-collector of unmarked (i.e., also uncopied) sub-lists attached to marked (copied) lists. This can be done quite elegantly, without an additional stack, by simply extending the basic mechanism of the Cheney-Arnborg algorithm. The algorithm functions non-recursively by first copying the top location of a list (which is referenced from the stack) and then linearly scanning through the copied locations, tracing references and copying unmarked locations. Thus, all the list-processor has to do is to add to the queue of unscanned locations a link-cell to a reference to an otherwise possibly inaccessible location. This solution has first been described by Barbacci et al. (Ba 71).

Fig. 3 - 10 shows the structure of the solution. SCAN is the pointer which scans through the queue of locations whose

references are yet to be traced and updated. The list-processor has just made an assignment to a reference which has already been passed by SCAN, i.e., which already was updated to point to an object in GCUSES. The object now referenced is in the old semi-space (LPUSES), and as it is impossible for the list-processor to determine whether there is any other reference to that object, it puts a link-cell to the reference into the queue. NEW always points one cell behind the end of the queue, and as therefore SCAN = NEW is a necessary condition for terminating the tracing and copying, SCAN will reach the link-cell before this termination. All the garbage-collector has to do then is to trace the reference referred to by the link-cell, i.e., by one level of indirection. Afterwards, the link-cell has become garbage which will be collected in the next collection.

3.2.3 Program description

The overall structure of the program for the concurrent compactifying garbage collection system is given in Fig. 3 - 11. Except for a few additional global functions, it is equivalent to that of the previous example.

Fig. 3 - 12 presents the declarations of global constants, modes, and variables. Line 11 introduces the mode of locations in the heap. Only LINKCELL is always defined. If this system were used to support a language such as ALGOL 68, more modes would be predefined, namely all modes which are part of the language (bool, real, int, etc.). All programmer-defined modes would be added to the ordered set MODE as they were encountered by the compiler.

Line 12 declares a type CELLTYPE which has the range of values SINGLE (for cells in locations of size one), OVERHEAD (for the first cell in a multi-cell location), and BODY (for all but the first cells in multi-cell locations).

In line 13, the united mode of all cells is introduced as a packed record. Packed declares the intention to have the fields of the record packed. This is supported by PASCAL. In this

case, it is assumed that each of the record variants fits into one word.

Type DATABASE (line 27) is the mode of the one shared variable ALL (line 36). This combination of all shared data into one variable is used primarily for clarity of the algorithm.

As can be seen, the array of garbage-collector marks is not shared in this system: any LINKCELL location in semi-space LPUSES indicates a copied location and the list-processor uses this. The array MARKED is nevertheless used by the garbage-collector for reasons of efficiency; it allows determination of whether or not a location has been copied without accessing the location itself. This is advantageous in virtual memory systems and also increases the marking and unmarking rates.

Fig. 3 - 13 shows procedure INITIALISATION. The first few statements partition the heap into the two semi-spaces SS1 and SS2. Arbitrarily, the list-processor starts with SS1. At first, all of LPUSES is free space (line 14). After INITIALISATION, i.e., before entering the concurrent statement, all invariants are supposed to hold.

In Fig. 3 - 14, the auxiliary functions are given. SIMPLE is a predicate which is true for all modes whose locations occupy one cell. APOINTER is a predicate which indicates for a given mode whether the cell at OFFSET from a location's beginning is a pointer. LONG gives the length of locations of a given mode. INSEMISP is a predicate of a pointer which is true if the pointer references a location in a given semi-space. (For a non-contiguous semi-space, this would have to be altered.)

Fig. 3 - 15 shows the list-processor. PUSH and POP operate on the stack of pointers into the heap or to built-in atoms (e.g., NILP). PUSH and POP are always called within a critical region for ALL. One general constructor procedure for all modes (except LINKCELL) is given (lines 16...39). In it, all storage allocation from free space is done. All initialising parameters are taken off the stack. If no specific initialisation is desired,

these parameters must be skip values for all sub-locations of "reference to" mode in order to permit tracing. The reference to the newly created location is pushed onto the stack.

The only other true list-processor function shown is procedure ASSIGNMENT for general assignment to locations of all modes, such as A:=(true, 123, "A") in ALGOL 68. It takes all parameters off the stack, but does not deliver a value. While a garbage collection is ongoing (MARKING = TRUE), it calls procedure SAVE (line 40) for every pointer being assigned to (lines 64 and 72). If the reference being assigned to is in GCUSES, i.e., already copied, SCAN is already past the reference, and the new reference value points into the old semi-space, SAVE generates a link-cell to the first reference, space in GCUSES permitting. If the system is out of copy-space, SAVE causes termination.

Obviously, LISTPROCESSOR is not intended to be in any way a complete list-processing program. It is only supposed to show how the regions must be structured, and which steps must be taken to maintain the invariants.

Procedure GC (Fig. 3 - 16) on the other hand is complete. Collection starts when the number of free cells is less than TRIGGER cells (line 31). Then, tracing and copying is performed until the last list pointer from the stack has been traced (ISTACK = TOP) and the queue of untraced locations in GCUSES is empty (SCAN = NEW). Each call to COPY copies one entire location (line 13), marks it, and converts it into a link-cell. Also, COPY updates the reference to the cell copied to the address of the copy (line 19). COPY is therefore only called for still uncopied cells in semi-space LPUSES. If the copy-space is exhausted, GCEXIT is set to TRUE (line 22).

When the collection is completed, the semi-spaces are swapped (lines 49...54), and a test is performed to check whether at least NRNEEDED cells have been reclaimed. If not, EXIT is set to TRUE for termination.

3.2.4 Correctness proofs

The structure of the marking and copying phase of the garbage-collector discussed above is different enough from that developed in section 2.2.2 to merit a new proof of its correctness.

Proposition a: "Every location that is copied has been live"

Proof a:

Let n be a copied cell. Suppose it was copied as m^{th} cell, $m \geq 1$, during a collection. The proof is by induction in m.

For m = 1, n = HEAP[STACK[0]], and thus n has been reached from the stack.

For m > 1, n must either have been copied by a call COPY (ALL, STACK[ISTACK]) at line 46, by COPY (ALL, HEAP[REF].PTR) at line 72, or by COPY (ALL, HEAP[SCAN].PTR) at line 79. In the first case, it was thus reached from the stack. In the latter cases, it must have been a successor to a copied location which, according to the induction hypothesis, had been reached from the stack. Consequently, n has also been reached from the stack.
This completes the proof.

Proposition b: "The algorithm terminates"

Proof b:

Every time line 38 is passed, either a pointer from the stack or a pointer from the queue in GCUSES is traced, or a non-pointer cell in that queue is skipped. As the stack is finite and only previously unmarked locations are marked and copied to GCUSES, line 38 can only be passed a finite number of times.

It remains to be shown that COPY does not contain any infinite loops. The only loop in COPY is the for loop on line 13 which can only be executed a finite number of times per call.
This completes the proof.

Proposition c: "At termination, every mode which can be reached from STACK[0]...STACK[TOP-1] has been copied"

Proof c:

Assume that the algorithm has reached line 49 and that n is a node which can be reached from the stack and which

has not been copied.
Let n_0 = HEAP[STACK[P]] , n_1, n_2 ... n_f = n be the locations along a path from n_0 to n. Let n_s be the first of these locations which is not copied, $s \geq 1$. n_{s-1} is copied.
There are two possible cases:

First case: n_s has become a successor to n_{s-1} before SCAN reached the pointer referencing n_s. Then SCAN < NEW at that moment, and the algorithm could not terminate without SCAN reaching the pointer to n_s. Inspection of the algorithm shows that this implies copying of n_s before SCAN = NEW, i.e., before completion. This contradicts the assumption.

Second case: n_s has become a successor to n_{s-1} after SCAN reached the pointer to n_s. For this to happen, a pointer assignment is required.

During the garbage collection (MARKING = TRUE), all pointer assignments check for this configuration (see Fig. 3 - 15, lines 64, 72, and 43). When the "dangerous" configuration is detected, the list-processor puts a link-cell to the reference to n_s into the queue of locations to be scanned and simultaneously increases NEW by one. So, as long as always SCAN $\leq$ NEW is maintained during marking, SCAN must reach the link-cell before SCAN = NEW, i.e., termination. Again, this contradicts the assumption.

Thus, given that n_{s-1} is copied, n_s is copied. To complete the induction, it remains to be shown that n_0 is always copied. Given that ISTACK $\leq$ TOP is maintained during collection, this will always be done at line 46 before completion.
This completes the proof.

Proposition d: "The system is deadlock-free"

Proof d:

As all shared variables are combined into one shared record, the only possibility for deadlock is a circular wait-condition by both processes in conditional critical regions. Given that invariant 5 holds throughout, this is impossible.
This completes the proof.

3.2.4.1 Invariants

In principle, all eleven invariants developed in chapter 2 are applicable here, too. However, in view of the fact that this garbage collection system relocates and compactifies by copying, slight amendments are required. These shall first be developed, and then it shall be shown that the invariants are maintained after initialisation.

Amendments

As the system described above cannot proceed if the copy-space is exhausted during a collection, this leads to a broader interpretation of "storage exhausted" in invariant 1:

EXIT ≡ processing accomplished or free space exhausted
or copy-space exhausted

Invariant 1'

(In an operating system environment permitting requests for storage by running programs, conditions "free space exhausted" and "copy-space exhausted" signify that all space which is requested and which the operating system can allocate is exhausted.)

In this example, the number of cells in free space NRFREE is represented by the term ENDSS[LPUSES] - FREE + 1. Similarly, the number of cells available for copying is ENDSS[GCUSES] - NEW + 1. The requirement that cells may only be allocated from either space while it is not exhausted gives rise to

ENDSS[LPUSES] $\geq$ FREE - 1 and ENDSS[GCUSES] $\geq$ NEW - 1

Invariant 2'

Accordingly, invariant 3 takes on the form:

ENDSS[LPUSES] - FREE + 1 $\leq$ ENDSS[LPUSES] - STARTSS[LPUSES] + 1

and

ENDSS[GCUSES] - NEW + 1 $\leq$ ENDSS[GCUSES] - STARTSS[GCUSES] + 1

Simplification yields

FREE $\geq$ STARTSS[LPUSES] and NEW $\geq$ STARTSS[GCUSES]

Invariant 3'

Substitution of ENDSS[LPUSES] - FREE + 1 for NRFREE in invariant 4 results in the synchronisation invariant

ENDSS[LPUSES] - FREE + 1 < TRIGGER implies GC required

Invariant 4'

Invariant 5 does not require any adaptation.

The updated form of invariant 6 is again arrived at by substitution for NRFREE:

ENDSS[LPUSES] - FREE + 1 = number of free cells

Invariant 6'

On the following two invariants, the correctness proof of proposition c in section 3.2.4 depends.

Invariant 7 does not require adaptation.

Invariant 8 has to be replaced by an equivalent invariant which reflects the different mechanism for tracing and keeping lists accessible to the garbage-collector process. As not all cells in the queue HEAP[SCAN]...HEAP[NEW - 1] are references (some are just overhead cells, and others contain non-pointer values), the invariant must take on the form of an inequality:

NEW - SCAN $\geq$ number of lists still to be traced from semi-space GCUSES

Invariant 8'

Invariant 9, 10, and 11 are applicable unchanged.

In the following it will be shown that all these invariants hold after initialisation and that they are maintained inside the concurrent statement.

Invariants after procedure call INITIALISATION (Fig. 3 - 13)

Invariant 1'

As FREE $\leq$ ENDSS[LPUSES] and NEW $\leq$ ENDSS[GCUSES], EXIT $\equiv$ FALSE is correct after exit from the critical region, assuming that "processing accomplished" $\equiv$ FALSE.

Invariant 2'

Holds after the critical region, as FREE $\leq$ ENDSS[LPUSES] and NEW $\leq$ ENDSS[GCUSES].

Invariant 3'

Holds after the critical region, as FREE = STARTSS[LPUSES] and NEW = STARTSS[GCUSES].

Invariant 4'

Holds after the assignment to TRIGGER on line 19.

Invariant 5

Given that MAXLENGTH > 0, the invariant holds after line 20.

Invariant 6'

In the beginning, all cells in semi-space LPUSES are free. As FREE = STARTSS[LPUSES], the invariant holds after the critical region.

Invariant 7

As MARKING $\equiv$ FALSE, this invariant holds after the critical region.

Invariant 8'

Holds for same reason.

Invariant 9

There are no live cells, and TOP = 0. Holds.

Invariant 10

All cells in semi-space LPUSES belong to set FREE. All other sets are empty. The invariant therefore holds.

Invariant 11

Holds, as the mode of free cells can be regarded as bits.

Invariants after critical regions in LISTPROCESSOR (Fig. 3 - 15)

Invariant 1'

Line 44: EXIT is correctly set to TRUE when the list-processor finds the copy-space exhausted.

Line 79: It is only indicated schematically how 1' is maintained for "processing accomplished".

Invariant 2'

Lines 20...37: The conditional critical region is only completed if the allocation of NRNEEDED cells is possible.

Lines 44...51: Space for a link-cell in semi-space GCUSES is only allocated if the copy-space is not exhausted.

Invariant 3'

Not affected by the list-processor.

Invariant 4'

Not affected by the list-processor.

Invariant 5

Lines 19...20: As long as the size of the largest location is $\leq$ TRIGGER, the invariant is maintained. If this is not guaranteed by the language supported, a check must be made before entering await.

Invariant 6'

Line 35: After allocation of space, the pointer FREE is correctly updated before leaving the critical region.

Invariant 7

Lines 5...6: Whenever a pointer to a list is pushed onto the stack, ST.TOP-ISTACK is incremented by incrementing ST.TOP.

Lines 11...13: Only when the pointer popped off the stack references a list which still had to be traced is ST.TOP-ISTACK decremented. Otherwise (TOP < ISTACK), that difference is maintained at zero.

Invariant 8'

Lines 47...52: Whenever the list-processor puts a link-cell to a list to be traced from semi-space GCUSES into the queue HEAP[SCAN]...HEAP[NEW - 1], it correctly increments the number of cells in that queue (NEW - SCAN) by incrementing NEW.

Invariant 9

Lines 23...33: All parameters are popped off the stack and the pointer to the newly allocated location is pushed onto the stack in one critical region.

Invariant 10

Lines 23...33: Cells are taken out of the set of free cells and made live in one critical region.

Lines 58...75: All pop operations of parameters and assignments to a location are done in one critical region.

Invariant 11

Lines 23...33: Assuming all parameters to be of correct modes, the allocated location's mode and content do not conflict upon exit from the critical region.

Invariants after critical regions in GC (Fig. 3 - 16)

Invariant 1'

Line 22: When exhaustion of the copy-space is discovered by the garbage-collector, EXIT is set to TRUE.

Line 58: After completion of a collection, EXIT is updated correctly.

Invariant 2'

Lines 12...20: Only if sufficient cells for copying a location are available in copy-space is the copying performed and NEW updated.

Lines 49...50: The assignments to FREE and NEW maintain the invariant.

Invariant 3'

Line 20: As NEW is only increased, the invariant is maintained.

Lines 49...50: After swapping the two semi-spaces' roles, NEW and FREE point into GCUSES and LPUSES, respectively.

Invariant 4'

Line 31: Whenever the number of cells in free space has fallen below TRIGGER, a collection is either going on or is started.

Invariant 5

Not affected by the garbage-collector.

Invariant 6'

Lines 50...53: After all copying is completed, there are ENDSS[GCUSES] - NEW + 1 free cells available in free space after swapping of semi-spaces. Thus, the assignment maintains the invariant.

Invariant 7

Lines 42...46: Whenever a list referenced from the stack has been traced, the count of references still to be traced from the stack is decremented by incrementing ISTACK.

Invariant 8'

Lines 12...21: Whenever a location of size n has been

copied to semi-space GCUSES, the number of cells in the queue still to be traced (NEW - SCAN) is updated by increasing NEW by n.

Lines 67...72, 75...79: For each cell removed from the queue HEAP[SCAN]...HEAP[NEW], SCAN is incremented, and the queue length therefore decremented.

Invariant 9

Lines 12...21: When a cell directly referenced from the stack is copied, the pointer in the stack is updated to the relocated location in semi-space GCUSES before leaving the critical region. All other references to this location are via a link-cell. Similarly, a location in LPUSES referenced by a copied location is kept accessible (live) after having been copied.

Lines 45, 71, 78: As references to link-cells in LPUSES are encountered, they are replaced by references to the associated copied locations in GCUSES. Thus, finally all live cells are in the new semi-space.

Invariant 10

Lines 12...21: When a location has been copied, its original memory cells can become garbage, if there are no other references to it. The cells in copy-space into which the location has been copied may become live, if it is still reachable from the stack. Thus, all these (potential) changes of set membership are done in one critical region.

Lines 45, 71, 78: The updating of a reference to the copied location referenced by a link-cell may cause this cell to become garbage. It is therefore done as one indivisible operation.

Invariant 11

Lines 12...13: A location, i.e., mode indicator and content, is always copied as a whole.

Lines 15...16: The mode (LINKCELL) and the content of a link-cell are assigned in one critical region.

Chapter 4
D I S C U S S I O N

4.1 FINDINGS AND CONCLUSIONS

Design method

This thesis has shown that a system with concurrent garbage collection can be designed reliably and that its correctness can be proved. The design method developed can be summarised as follows:

- Given the heap management requirements (heap structure, data structures to be supported, access to the heap) for an application process, select a suitable garbage collection algorithm. This selection is governed mainly by the degree of real-time responsiveness required. In process control or operating system applications, there are maximum tolerable response times which must be guaranteed. In general, this is only possible to satisfy if the garbage collector process stays inside any critical region for a length of time which is independent of the list-structure being acted upon. The type of collector algorithm required here is one which acts upon at most one location (or a fixed number of locations) per entry into a critical region. The response time can then be bounded by the maximum amount of processing required by any location in any region. The penalty is of course that the overhead caused by region entries and exits is proportional to the number of locations to be processed (traced, marked, and collected). It appears that, in general, non-relocating algorithms are easier to adapt to the requirement of bounded time inside critical regions than relocating ones, and interpretative easier than recursive ones.
  If the real-time requirement is less severe, as for example the user requirements of acceptable response times in an interactive computing environment, it is possible to get away with a greatly reduced overhead of region entries/exits by selecting an algorithm which processes e.g. a whole list or sub-list in one critical region.

- The second step is to establish the variables shared by the two processes. By definition, the heap, the space (or list) of free cells, and all access pointers into the heap must belong to these.
  However, other variables (e.g., status variables such as the array of garbage collector marks) may also be shared. Depending on the access pattern to these variables and on the real-time responsiveness needed, the shared variables may be combined into a few shared records or even one.

- For all shared variables, the invariants to be maintained by their critical regions must be established. The eleven invariants presented in chapter 2 appear to be the minimal set of invariants for a system with concurrent garbage collection. However, they may require slight amendments and adaptations to cater for specific algorithms.

- Using the invariants, the program and its proof are developed simultaneously. The proof is made by showing that all invariants hold after initialisation and then are maintained by all critical regions. Based on this, the correctness of at least the following propositions must be shown:

  - Only inaccessible (i.e., garbage) cells are collected and returned to free space.
  - The system terminates.
  - The system is deadlock-free.

  The parallel development of proof and program is greatly aided by the use of a structured notation, using constructs whose correctness has been proved once and for all and can be taken as an axiomatic property.

In chapter 3, the approach just outlined was applied to two representative examples. The notation employed was PASCAL, enhanced by four constructs (concurrent statement, shared variable, critical region, and conditional critical region). It is felt that this notation shows the design intentions much more clearly than any unstructured notation. In particular, this is an improvement in clarity, brevity, and ease of understanding over a design expressed entirely with semaphores. This is a consequence of introducing an additional layer of abstraction (the process extensions to PASCAL) into the design. The relative brevity and simplicity of the proofs presented

could only be achieved by using the axiomatic properties of each of the constructs, rather than proving the correctness of each individual instance of the constructs.

Determinacy

The analysis of the list-processor/garbage-collector system of co-operating processes has shown that, in general, this system is only conditionally determinate: to a given heap utilisation by the list-processor, there corresponds a minimal speed ratio between garbage-collector and list-processor below which it is not determinate. If the garbage-collector process is not fast enough, the list-processor's output history is influenced not only by its input history, but also by the garbage-collector's progress in that it runs out of storage before accomplishing its processing task. Thus, for a given concurrent garbage-collector implementation, only a limited parameter space in the dimensions "relative speed" and "heap utilisation" permits the list-processor to be unconditionally determinate. For certain assumptions concerning the heap dynamics, this space is identified by formula (2.3.43). Even within the parameter space which permits full concurency between storage allocation and garbage collection, determinacy of the applications process can only be achieved by making this process support the garbage-collector during the marking phase. This support has to maintain for the marking algorithm the set of lists yet to be traced. It ensures that for all live cells, the action sequence "allocation, tracing, marking" is completed. Several solutions meeting this requirement have been developed and described.

Space requirements and overheads

In general, a system with concurrent garbage collection requires more heap space for the same computation than one with sequential collection. This has two reasons: first there must always be sufficient cells in free space to satisfy the storage

requests of the applications process. Second, the concurrent garbage-collector cannot reclaim all garbage: all cells which are marked, but become garbage before the reclamation phase can only be collected in the following garbage collection cycle. Effectively, this reduces the useful heap space.

Additional space overhead is caused by the auxiliary references which the applications process must store for the marking algorithm to ensure determinacy.

Processing time overheads result from the synchronisation and mutual exclusion mechanisms, from the marking of cells which then become garbage, and from the higher frequency of garbage collection due to the smaller yield of free cells per collection.

Applicability of concurrent garbage collection

It appears that all garbage collection algorithms are basically suitable for employment in a concurrent garbage collection system. (Although, as has been discussed, there are differences in the real-time response times which can be obtained with them.) Thus, concurrent garbage collection is a feasible approach to making languages depending on garbage collection for their heap management (e.g., ALGOL 68, LISP, SIMULA, and SNOBOL) applicable to real-time programming. Because of the overheads incurred by concurrent collection, it is probably advisable to make the selection of the garbage collection mode a compile time (for in-line code) or run time (for the run-time system) option for the user. Also, there should be software facilities to inhibit the garbage collection process for time-critical pieces of code and to enable it for less critical periods, e.g., slow I/O.

4.2 LIMITATIONS

This thesis has not addressed the question whether and how, with the present state of the art in hardware and software, the region, await, and cobegin ... coend constructs can be implemented efficiently. It has used them as abstractions for design purposes. In his book (Ha 73), Brinch Hansen has shown how they can be implemented in terms of more primitive constructs, i.e., event queues and semaphores. From that description, it appears feasible to efficiently implement the higher-level constructs, but the author is so far only aware of one language under development which will contain them (though in a modified form). This language is "Concurrent PASCAL", and it is being developed by Brinch Hansen and others at the California Institute of Technology (Ha 75).

Until the availability of such languages, the constructs can be used for design and programming of systems with concurrent garbage collection, but must either be translated (manually or by language extension) into calls to a given real-time monitor or be implemented from scratch along the lines given by Brinch Hansen (Ha 73). In either case, the correctness of the implementation of the constructs must be proved.

Another related question not answered by this thesis is whether the conceptual simplicity of the synchronisation method used (conditional critical regions) is achieved at the price of unacceptably high processor overheads caused by busy waiting. The simplicity and elegance are demonstrated by the fact that the processes being synchronised are completely unaware of each other, i.e., the scheduling mechanism is hidden, and no process has the responsibility of scheduling other processes. This necessitates the re-evaluation of the synchronisation condition associated with the await of a conditional critical region every time the associated shared variable has been accessed by another process. The result is busy waiting which may only be tolerable for more loosely coupled processes.

At which point the associated overhead becomes unacceptable is impossible to postulate for the general case and has not been addressed in this thesis. However, if it is found in a specific implementation that too much processing time is absorbed by this busy waiting in conditional regions, they can easily be replaced by explicit scheduling between the two processes, e.g., by using event variables or semaphores. This forces the programmer to be explicitly aware of scheduling details and thus removes a level of abstraction. As this is only done at the last step in the sequence "concept/design/proof of correctness/implementation", and as it is only done where required for reasons of efficiency, i.e., in a few, isolated areas of code, it is far less harmful than using these lower-level constructs throughout.

4.3 FUTURE RESEARCH

In the course of the research done for this thesis, the author has found several areas in which research promises to yield results which should be of value for the implementation of concurrent garbage collection systems. These areas are outlined below.

Adaptive scheduling

The real-time responsiveness of a given system with concurrent garbage collection depends on the availability of sufficient heap space at any time. As the analysis of the heap dynamics in chapter 2 shows, the space required for the applications process not to run out or have to wait for free cells is a function of the consumption, marking, collecting, and unmarking rates. Similarly, the level of cells in free space at which to initiate garbage collection is a function of the heap utilisation, cell consumption rate, and marking rate.

In systems which are not dedicated but shared by several application programs, memory space and processor time used are

being competed for by these programs, and normally even accountable. The cost incurred by a program is then a function of these two variables.

For a program with concurrent garbage collection, there is at any time a maximum heap utilisation for which the applications process does not run out of storage, but for which the garbage-collector process is active all the time. At this heap utilisation, the processor overhead incurred by the collector is maximal. As more heap space is made available to the application, the collection frequency and therefore the additional processor cost decreases, but the memory cost increases. This poses a cost-minimisation problem with one control parameter, viz. heap utilisation, and two constraints, the minimum memory required to keep the applications process from waiting for storage, and the maximum memory allocatable.

The simplest, yet crudest, space allocation policy is to provide the applications process with a fixed amount of heap space which suffices for the worst combination of heap utilisation and space consumption rate during its computation. For an application with varying heap utilisation and consumption rate, however, this is not a cost-optimal strategy.

Arnborg (Ar 74) and Hoare (Ho 74) have described algorithms for cost-optimal memory management for sequential garbage collection. Because of the constraint of staying below the maximum heap utilisation which maintains concurrency, these algorithms are not applicable to concurrent garbage collection. The main problem here is that this limit is not a constant, but a function of various rates which may show significant fluctuations during a computation and also differ from computation to computation.

The research proposed here is the following:

- Given an implementation of a system with concurrent garbage collection, develop a sampling scheme which permits estimation of the rates of space consumption, marking, reclamation, and unmarking. This would have to be a third process in addition to the applications and collector processes.

- Develop an algorithm which predicts the cost-minimal heap utilisation from these rates under the given constraints.

- Make the third (monitor) process request and release heap space according to that algorithm, and also update the value of TRIGGER accordingly.

It is felt that such an adaptive scheduler could be of great benefit, in particular for computer systems which process a widely varying job-mix. In highly loaded systems, it might even be the only way to permit concurrent garbage collection to be used within the resource constraints.

Parallelism in garbage-collector

The memory overhead incurred by concurrent garbage collection can be reduced by increasing the marking, reclamation, and unmarking rates. This could either be done by using a faster processor to execute the garbage collector process, or by using several processors in parallel for this. For each technology, there exists a limit to the maximum processor speed obtainable. However, providing that memory access contention remains acceptable, one can combine several processors to increase the total processor throughput.

With the simple repetitive processing task which the garbage-collector represents, one would not even have to use general-purpose processors for multi-processing the garbage collection (see also next section).

In a concurrent garbage collection system without compaction, all three phases of the collection permit parallel execution. In the marking phase, the lists of live locations can be traced and marked concurrently by several processes. Reclamation could be done by multiple processes, e.g., by letting the processes scan different areas of the heap for unmarked cells. One could proceed similarly for the unmarking phase. To facilitate such concurrency, the heap would almost certainly have to be organised as a resource array, e.g., HEAP:array[PAGESPACE]of shared PAGE.

Compactifying garbage collection offers the same scope for parallelism only during the marking and unmarking phases. As the new address (after relocation) of a location depends on the new addresses of previously relocated locations, there appears to be less potential concurrency during the relocation phase.

The research proposed is to investigate for various important garbage collection algorithms the maximum possible degree of parallelism in each phase of the collection, and to determine the cost-optimum number of processors, as a function of hardware , memory , and processor usage for given applications.

This research is of interest not only for concurrent garbage collection systems, but also for sequential ones. It is well possible that, for some systems, fast sequential garbage collection by multiple processors is a viable alternative to concurrent garbage collection, as it would not require the applications process to be burdened with any synchronisation and mutual exclusion code, or with cooperation with the marking algorithm.

Hardware environment for garbage collection

With the decreasing cost of hardware and the increase in hardware adaptability, e.g., by micro-programming, the designer of an environment for garbage collection must consider where and how he can use special hardware for the implementation.

One potential application has already been mentioned, viz. the use of multiple processors for multi-processing the garbage-collector. These processors could either be micro-processors, micro-programmed processors with an instruction set which is specialised for the garbage collection code, or of course normal central processors (although the latter would probably be an inefficient overkill). Because of the small amount of code to be executed, it is possible to store all or most of it in special memory apart from the normal central memory. This would prevent the multiple processors from degrading system performance by competing for access to central memory for fetching instructions.

Another area in which hardware can be used to facilitate garbage collection with lower overheads is the use of descriptors. If, with each memory word, a descriptor field were associated which could be set, read, and tested by special instructions, these could be used to store the mode or at least the category of the mode (pointer versus non-pointer) of the location. This would permit efficient execution not only of the garbage-collector, but also of applications programs compiled from languages such as ALGOL 68 or SIMULA. With the rigid memory organisation of the vast majority of present computers, descriptors have to be an integral part of a memory word and must therefore be designed into the basic hardware. It is hoped that future systems will provide more flexibility and permit the introduction of descriptor fields by the systems programmer.

Many parts of a garbage-collector could be made considerably faster and more efficient by associative memories, as much of the processing is of the nature "Find all (or the first) words with content X". This processing is at present done by linear scans or list searches. One use would be the array of mark bits, another the maintenance of the (scattered) set of free cells in an associative array, rather than as a linked free list.

The research proposed here is of a very wide scope. Some of the possible directions have been sketched out above, but the general objective envisaged is the development of hardware which is at least as efficient in executing programs written in languages depending on garbage collection as present stack-oriented computers are in executing programs written in ALGOL 60 or other conventional high-level languages.

Fig. 1 - 1 EXTERNAL OBJECT, INTERNAL OBJECT, AND THE RELATIONSHIP "TO POSSESS"

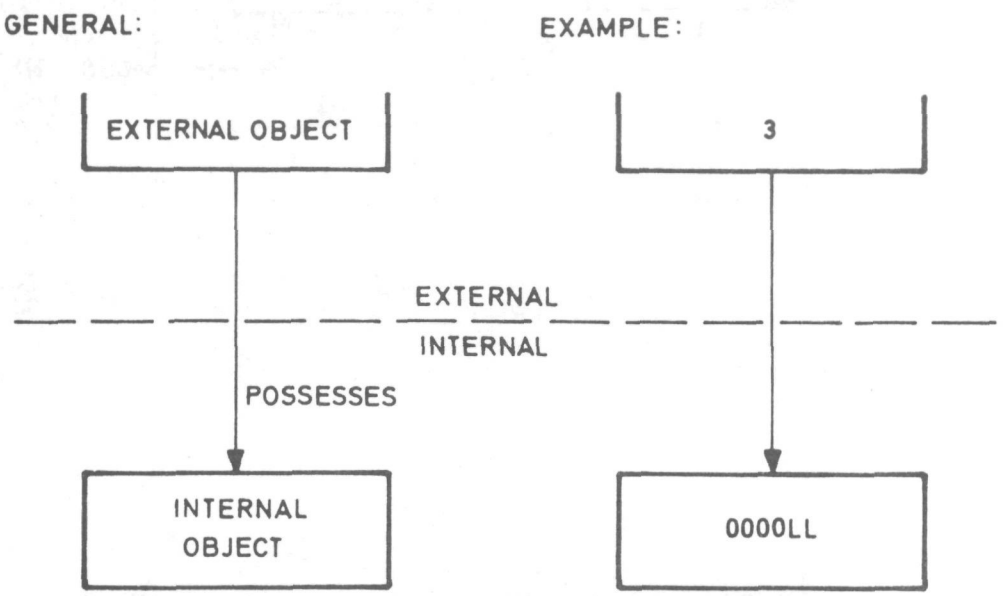


Fig. 1 - 2 THE CONCEPT "VARIABLE"

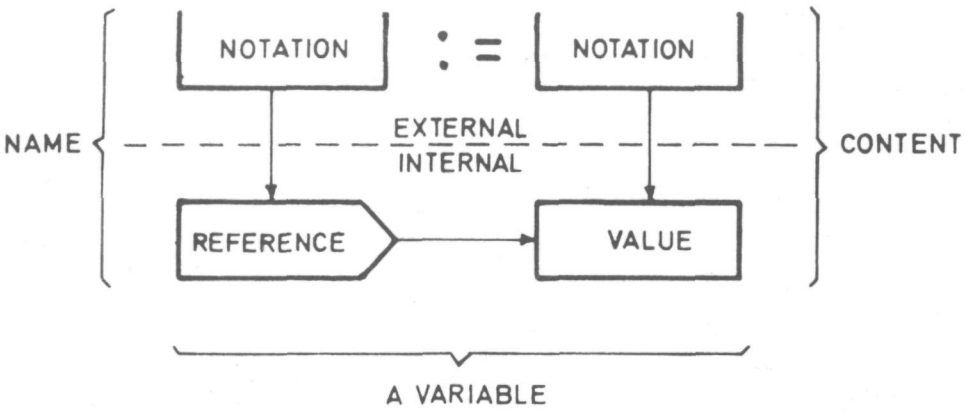


Fig. 1 - 3 HOW VARIABLES, IDENTIFIERS, ADDRESSES, LOCATIONS, AND OBJECTS INTERRELATE

```
char FIRST INITIAL := 'K';

int  AGE := 38;

bool OVERWEIGHT := true
```

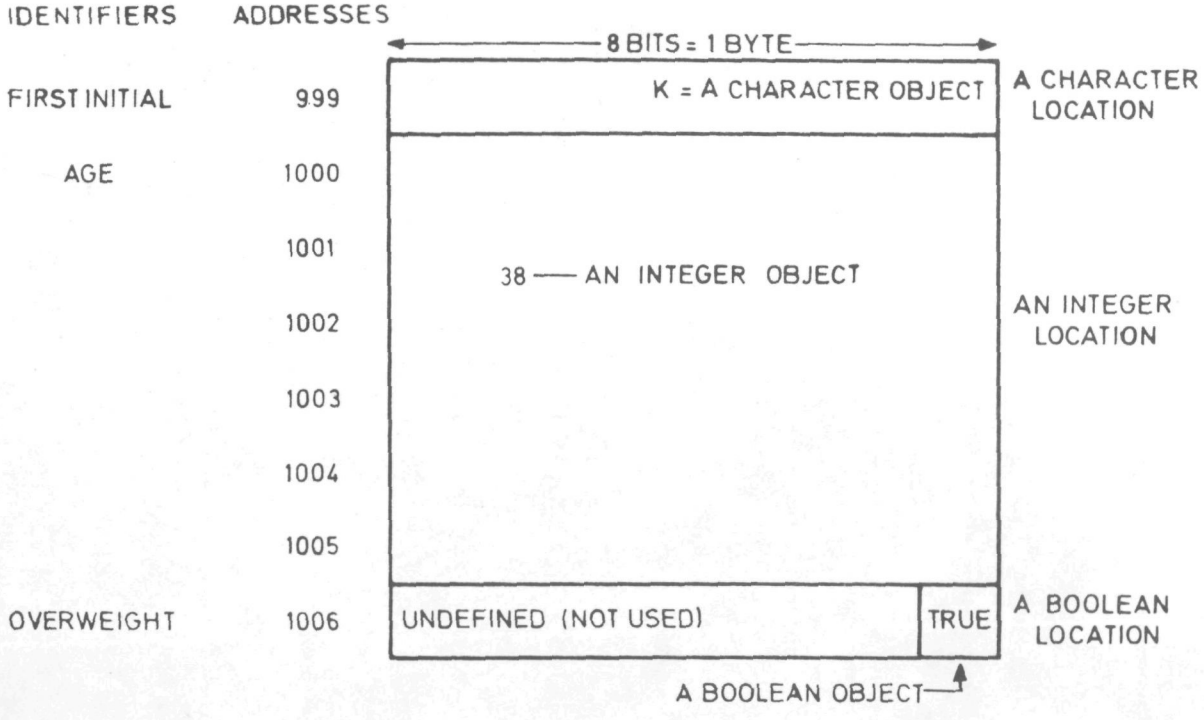


Fig. 1 - 4 STORAGE FOR A FORTRAN PROGRAM MODULE

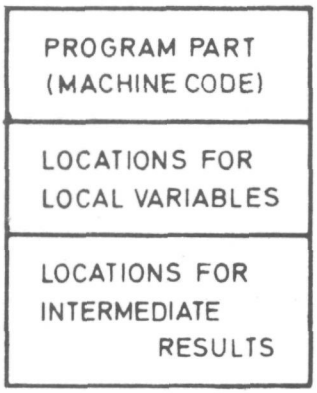


Fig. 1 - 5 STATIC BLOCK STRUCTURE A LA ALGOL 60

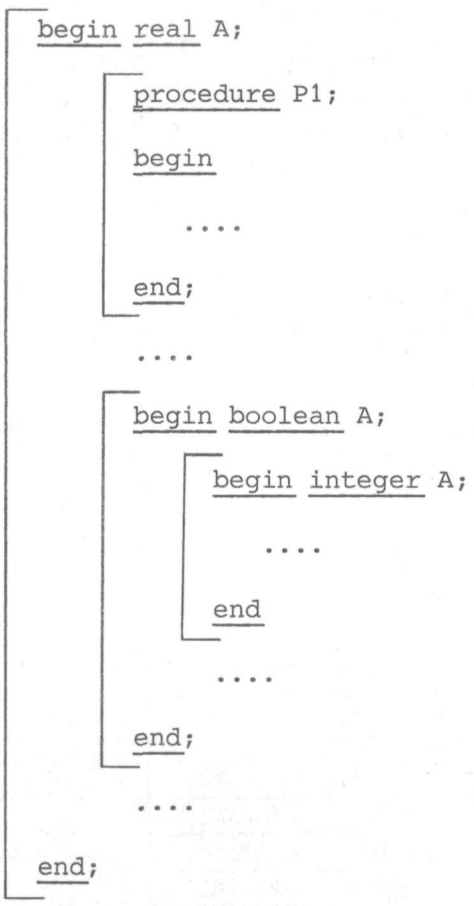
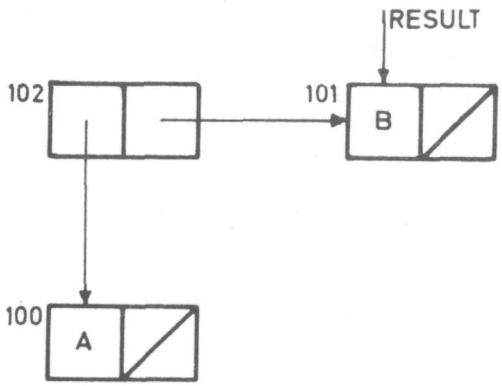


Fig. 1 - 6 EXAMPLE OF GENERATING AND LOSING REFERENCES

a) CDR (CONS (CONS A NIL) (CONS B NIL))

b)



c)

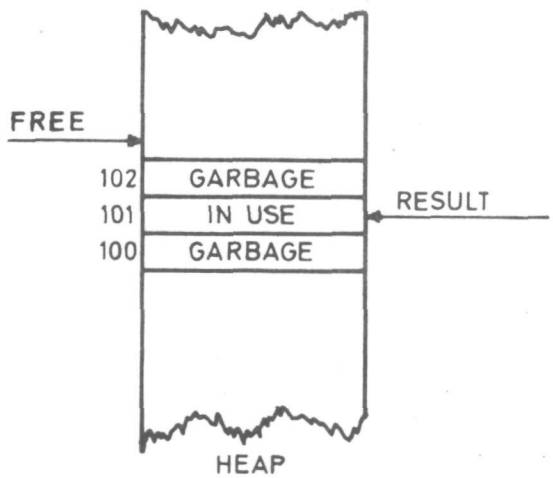
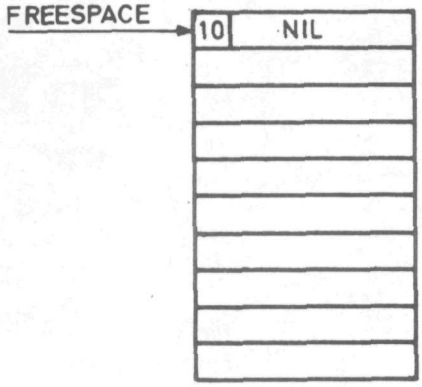
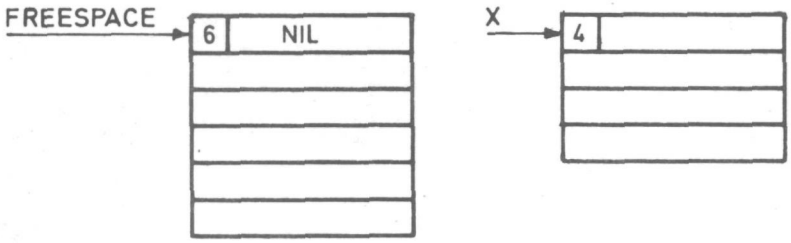


Fig. 1 - 7 PROGRAMMER-CONTROLLED STORAGE ALLOCATION

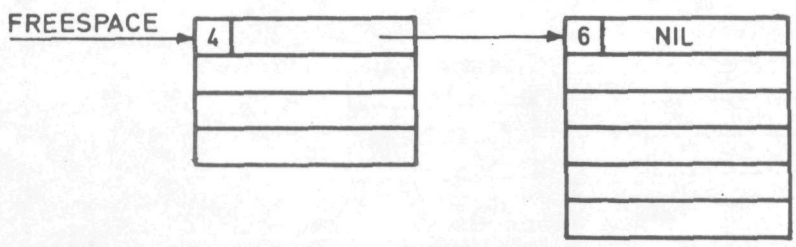
a) Initially



b) After X:=ALLOCATE(4)



c) After FREE(X)



d) After Y:=ALLOCATE(5)

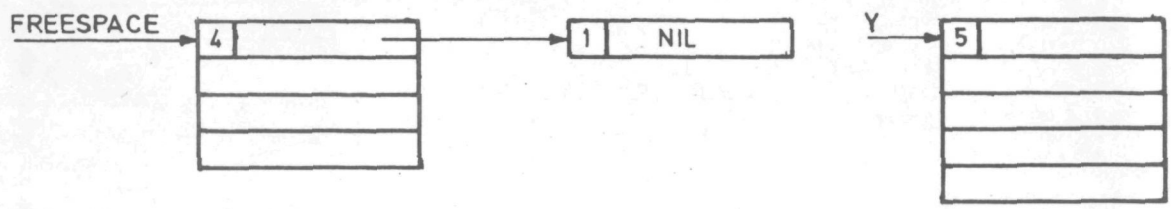
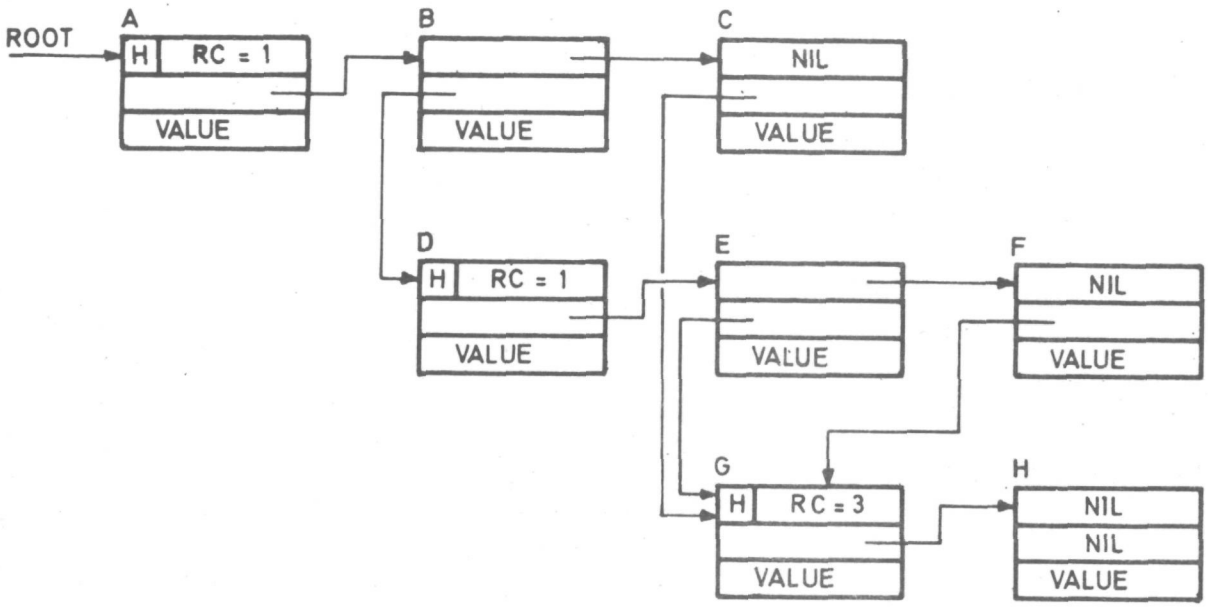


Fig. 1 - 8 LIST WITH REFERENCE COUNTS

a) Initially



b) After DELETE LIST(D)

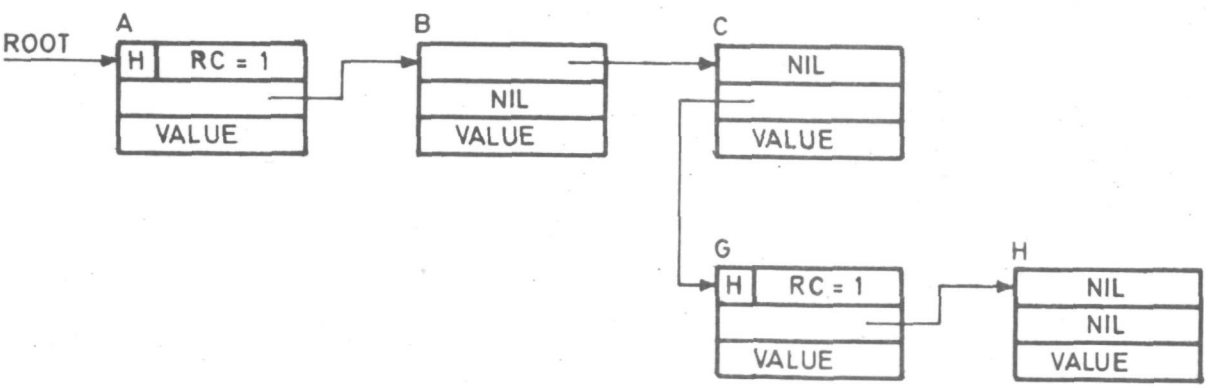


Fig. 1 - 9 STORAGE SNAPSHOT FOR COMPACTIFYING GARBAGE COLLECTOR A LA ARNBORG ET AL.

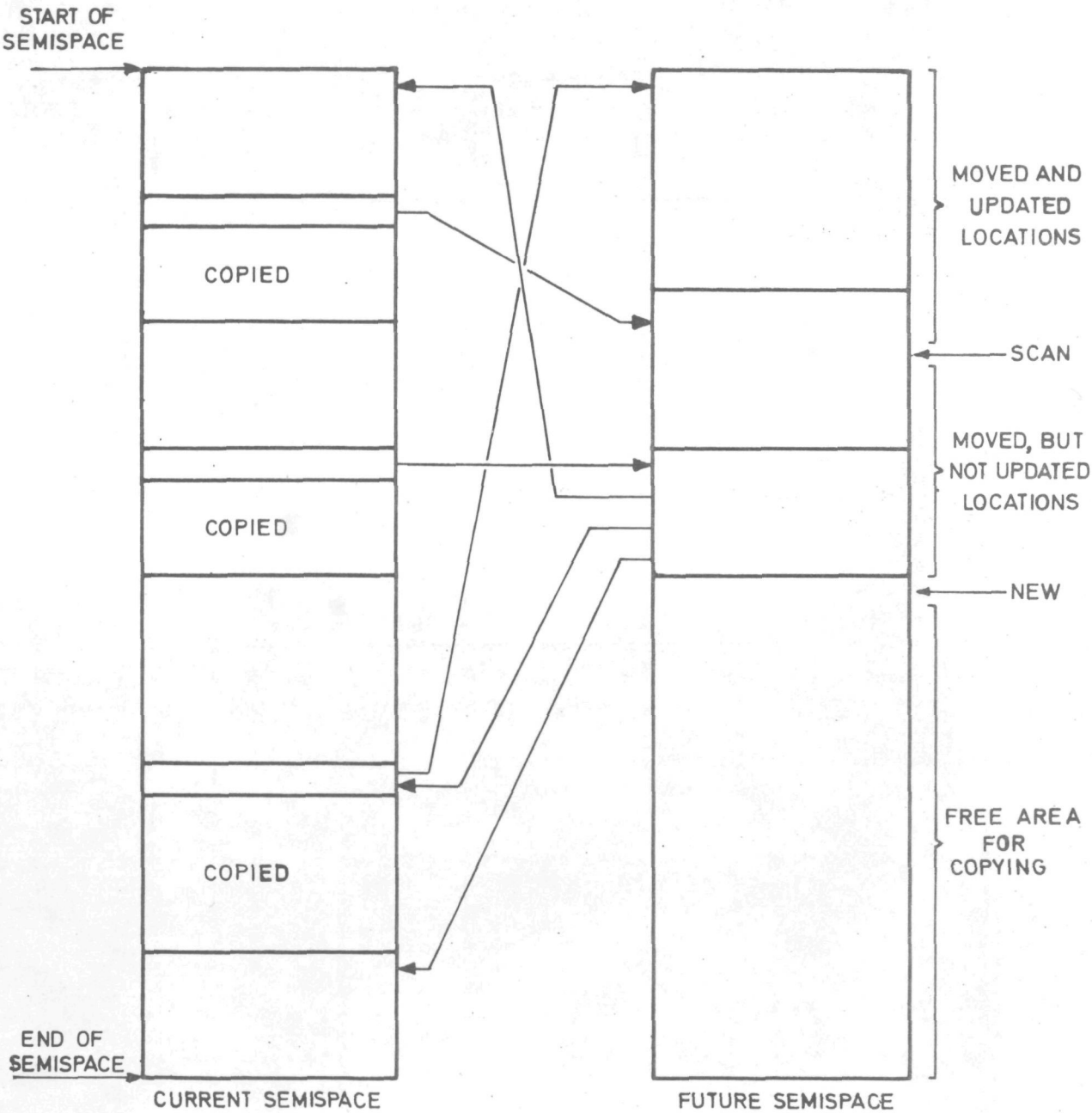


Fig. 2 - 1 ALTERNATION BETWEEN PROBLEM PROCESSING AND GARBAGE COLLECTION (SEQUENTIAL COLLECTION)

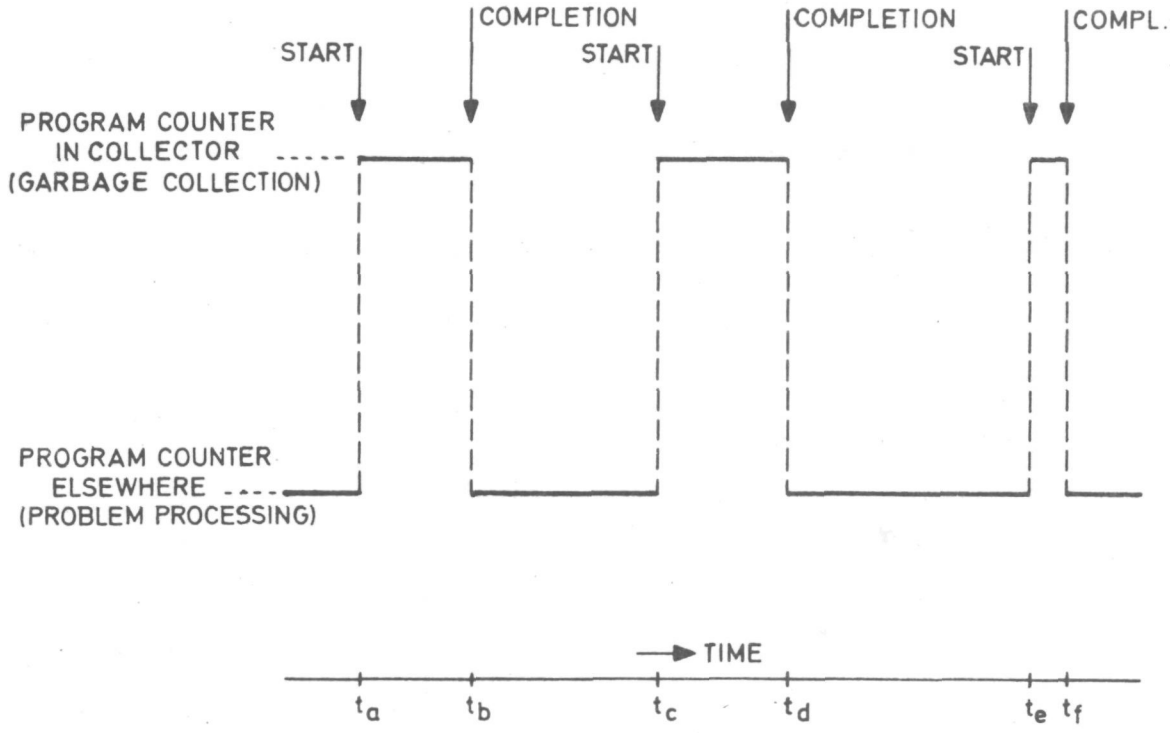
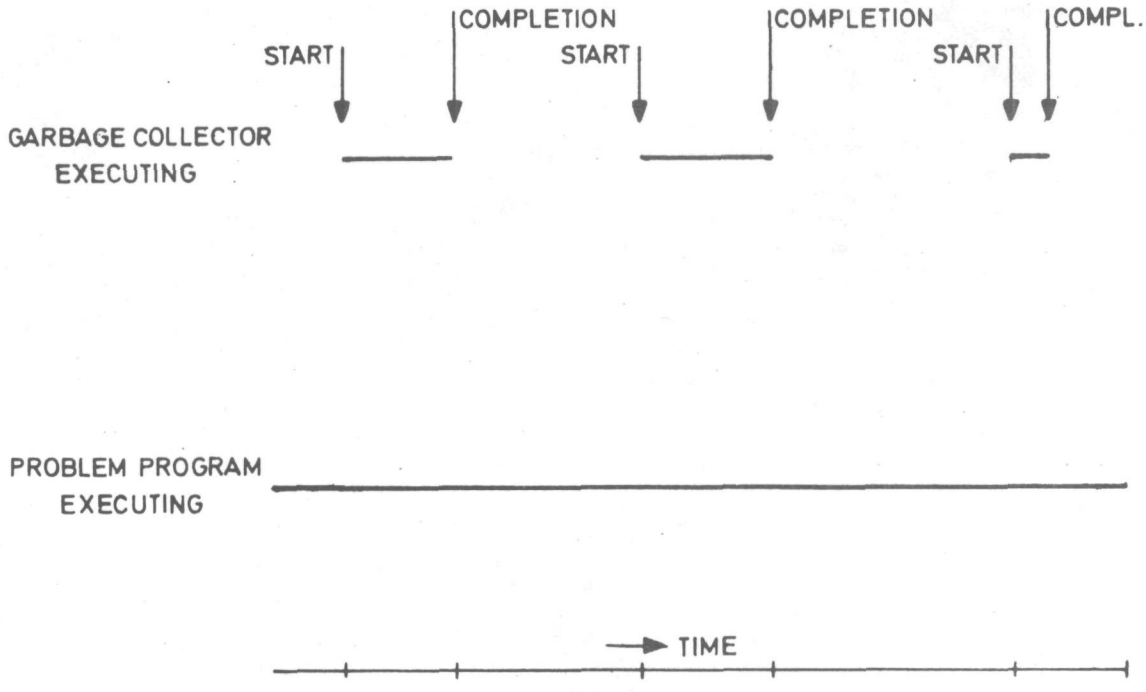


Fig. 2 - 2 CONCURRENT GARBAGE COLLECTION BY MULTI-PROCESSING AND MULTI-PROGRAMMING

a) Garbage collector and problem program on two processors



b) Garbage collector and problem program multiplexing one processor

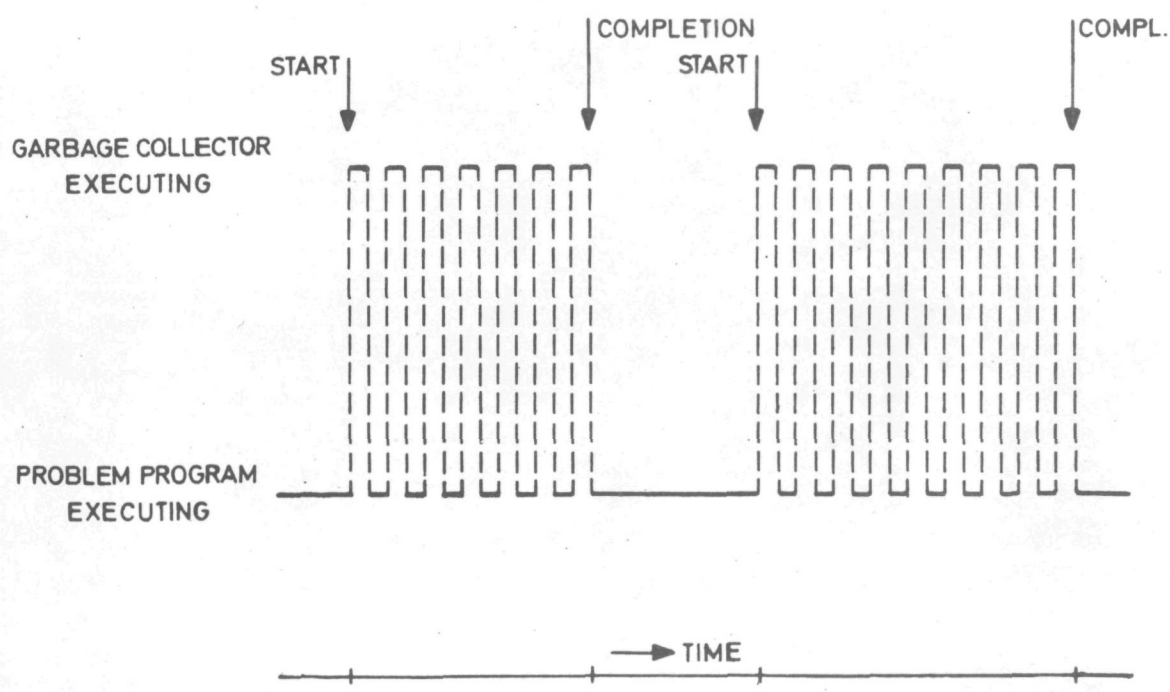


Fig. 2 - 3 CONCURRENT GARBAGE COLLECTION VISUALISED AS PRODUCER/CONSUMER SYSTEM

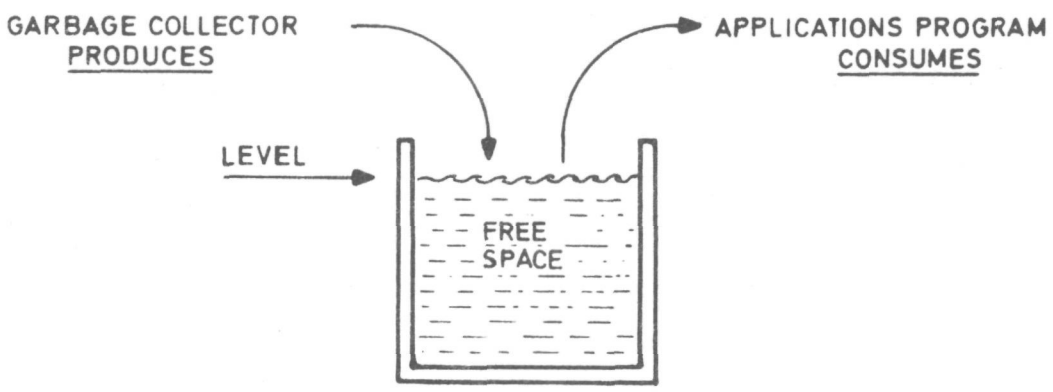
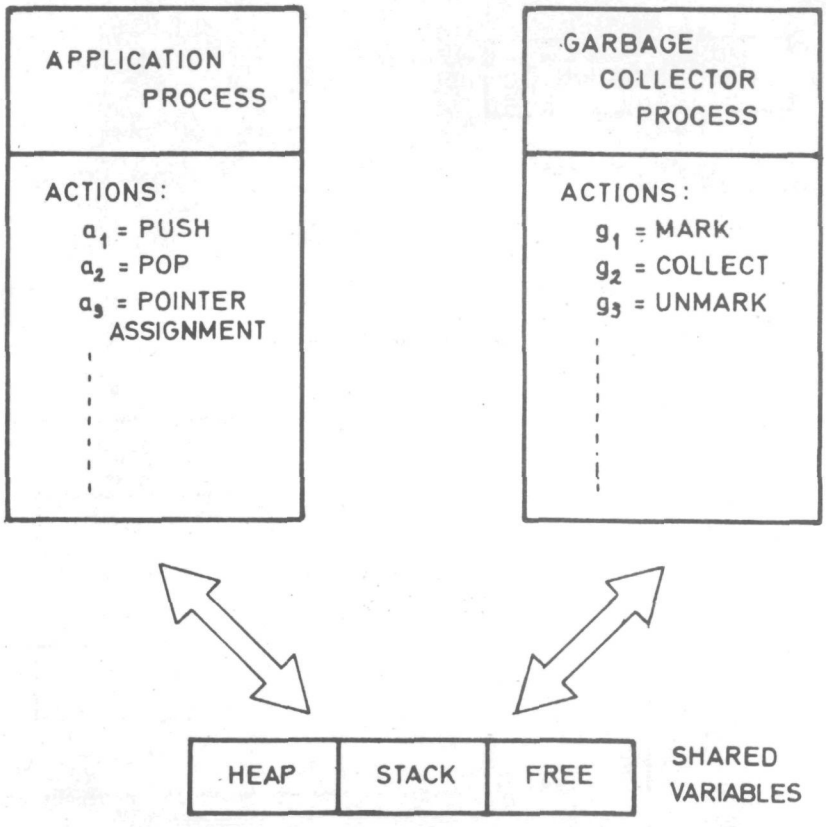


Fig. 2 - 4 APPLICATION PROCESS AND GARBAGE COLLECTION PROCESS AS SYSTEM OF CO-OPERATING PROCESSES

a) System of two co-operating processes



b) Some possible action sequences

$$\ldots a_2 g_1 g_1 g_1 a_3 a_1 g_2 \ldots\ldots\ldots$$

$$\ldots a_1 a_1 a_1 a_1 a_1 a_1 a_1 g_3 g_3 a_2 a_3 \ldots$$

$$\ldots g_2 g_2 g_2 g_2 g_3 a_1 a_1 a_2 a_2 \ldots\ldots$$

Fig. 2 - 5 ASSERTIONS ABOUT A SEQUENTIAL GARBAGE COLLECTOR

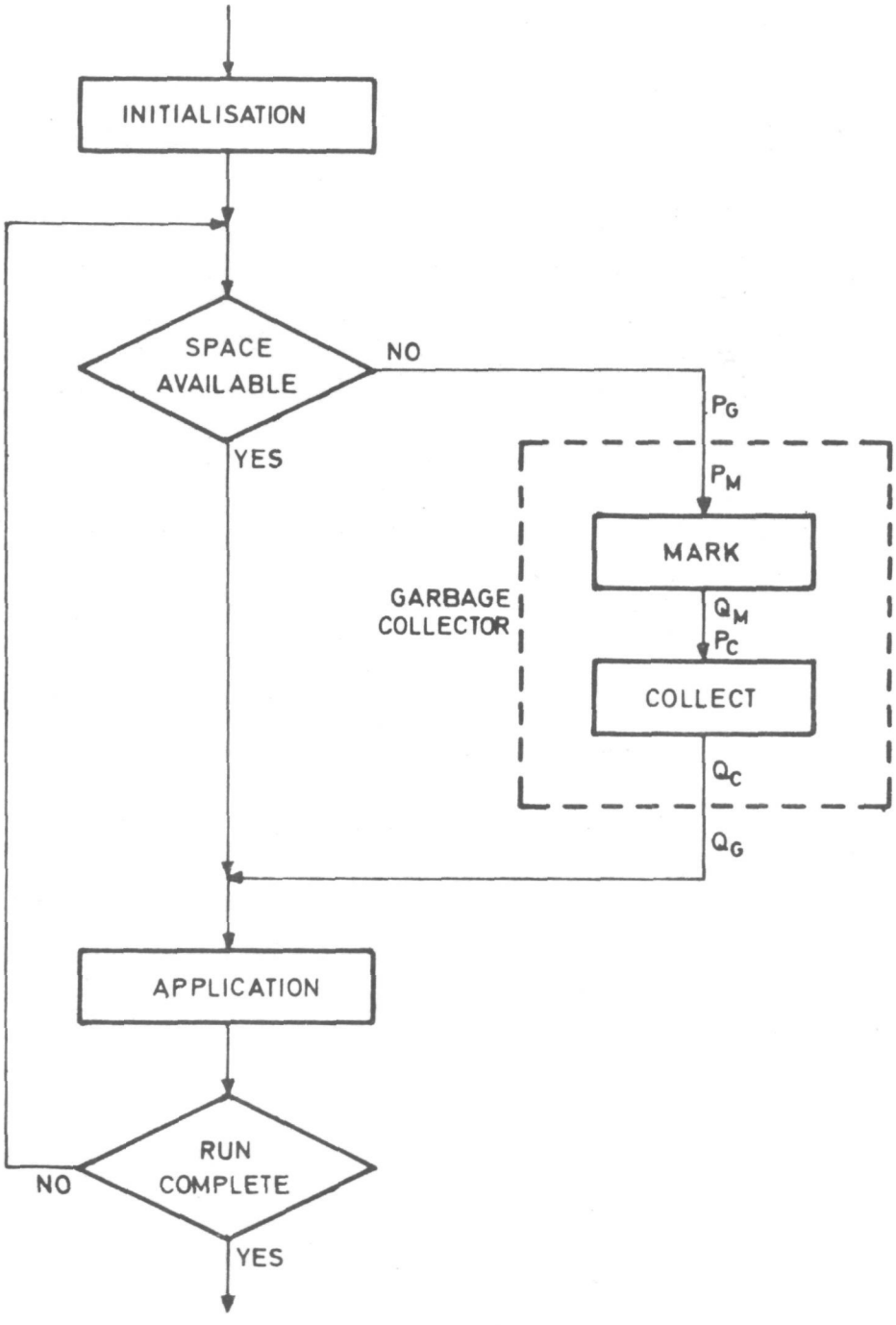


Fig. 2 - 6 BASIC PROGRAM STRUCTURE OF SYSTEM WITH CONCURRENT GARBAGE COLLECTION

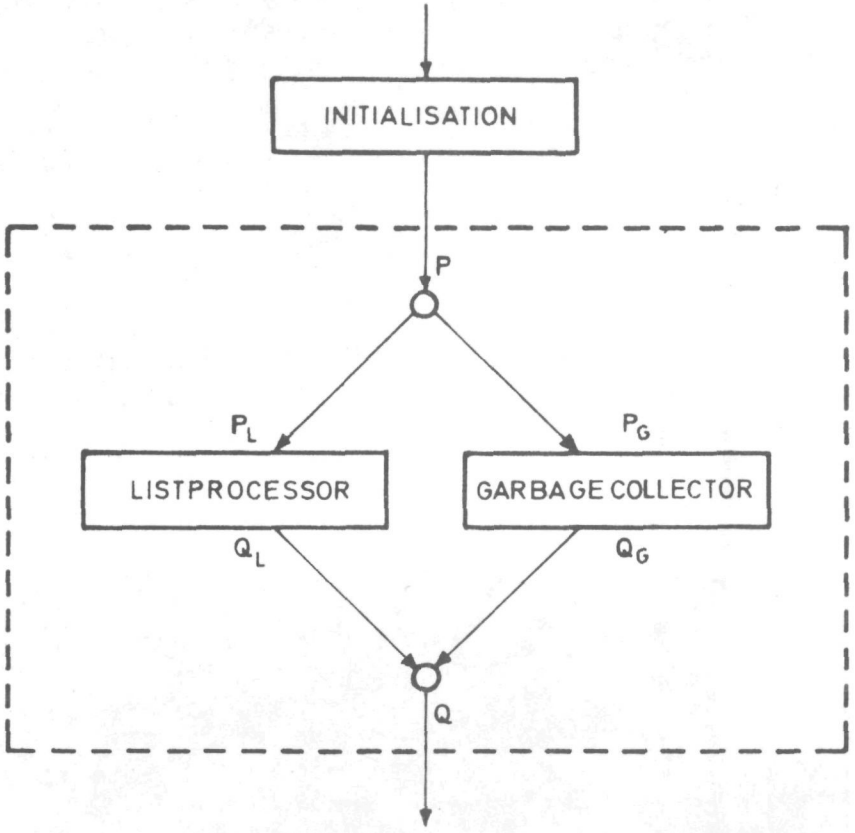


Fig. 2 - 7 MARKING AND COLLECTING PHASES OF GARBAGE COLLECTOR WITH IDEAL ASSERTIONS

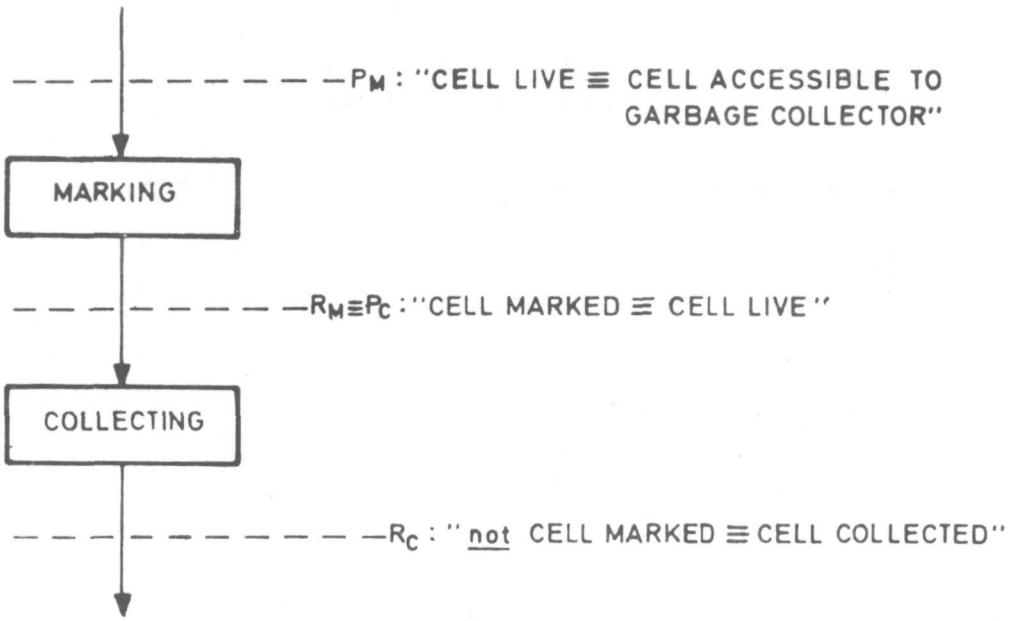
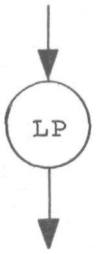


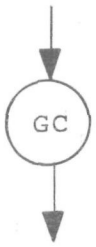
Fig. 2 - 8 LIST-PROCESSOR AND GARBAGE COLLECTOR AS NON-DETERMINATE SYSTEM

Shared memory cells

X, Y, Z

Interpretations





a: HEAP[X].CDR:=Y

b: if not MARKED[X] then
begin MARKED[X]:=true;
MARK(HEAP[X].CDR)
end

Computations

(I)

	S_O	a	b	S_E
MARKED[X]	false	false	true	true
MARKED[Y]	false	false	true	true
MARKED[Z]	false	false	false	false
HEAP[X].CDR	Z	Y	Y	Y

	S_O	b	a	S_E
MARKED[X]	false	true	true	true
MARKED[Y]	false	false	false	false
MARKED[Z]	false	true	true	true
HEAP[X].CDR	Z	Z	Y	Y

Fig. 2 - 9 VIOLATION OF INVARIANT 10

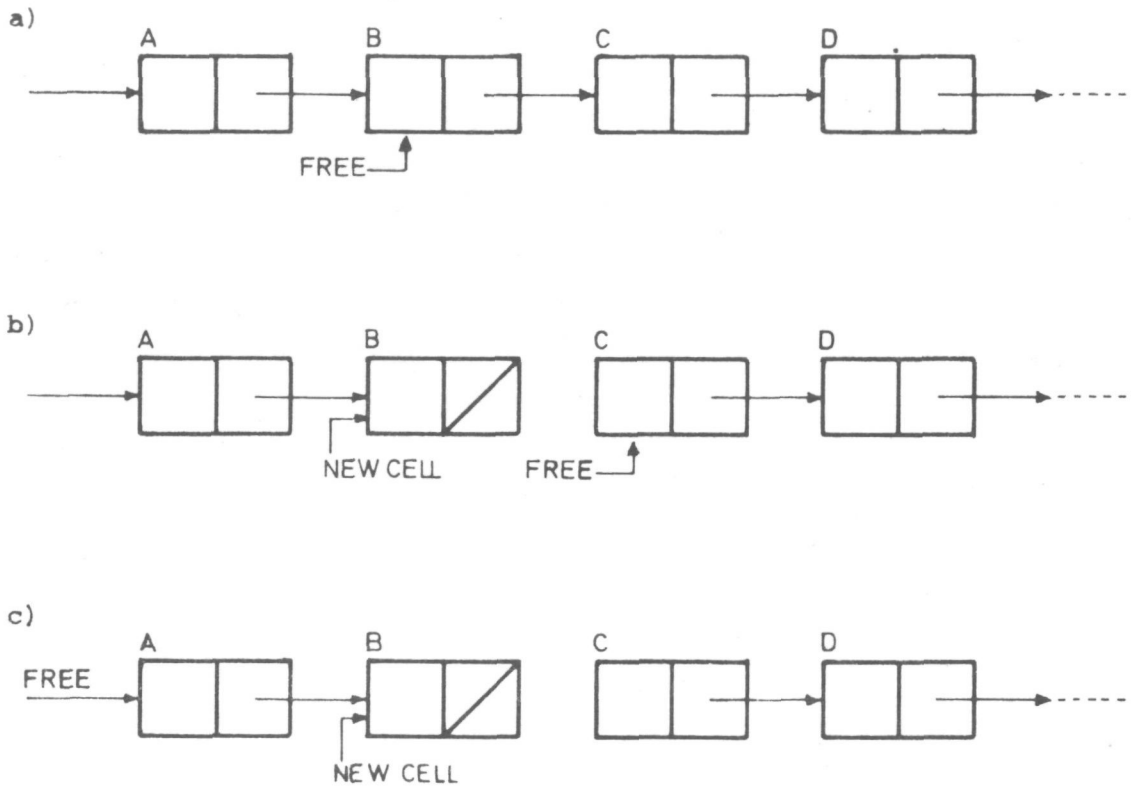


Fig. 2 - 10 MODEL OF HEAP COMPOSITION DURING MARKING PHASE OF CONCURRENT GARBAGE COLLECTOR

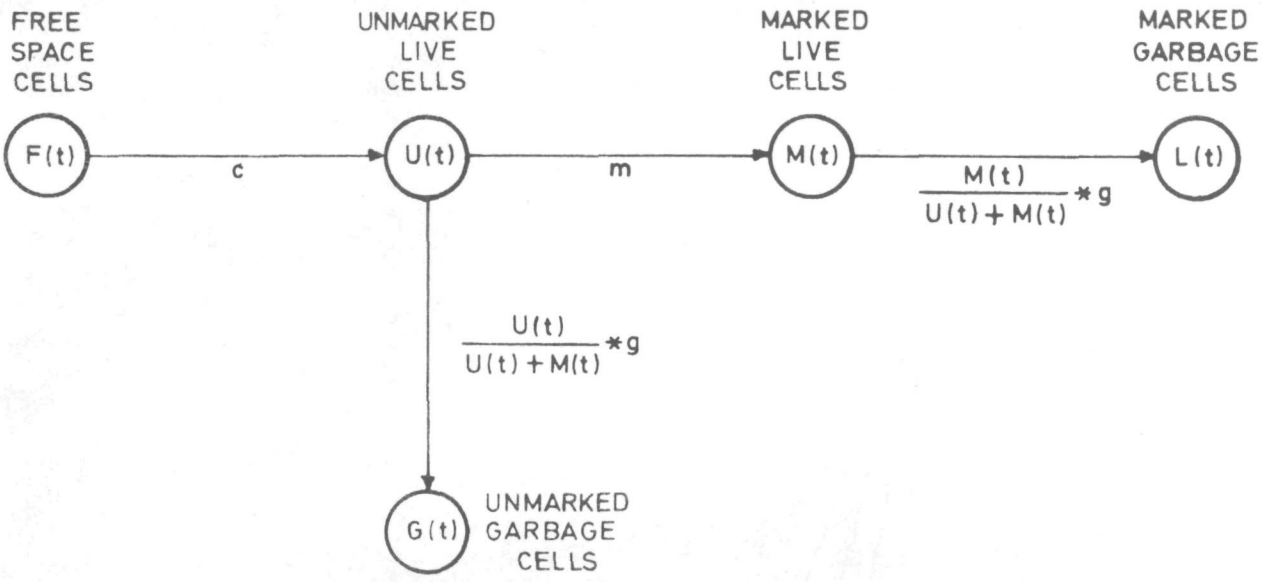


Fig. 2 - 11 MARKING TIME AS A FUNCTION OF RELATIVE MARKING SPEED AND HEAP UTILISATION

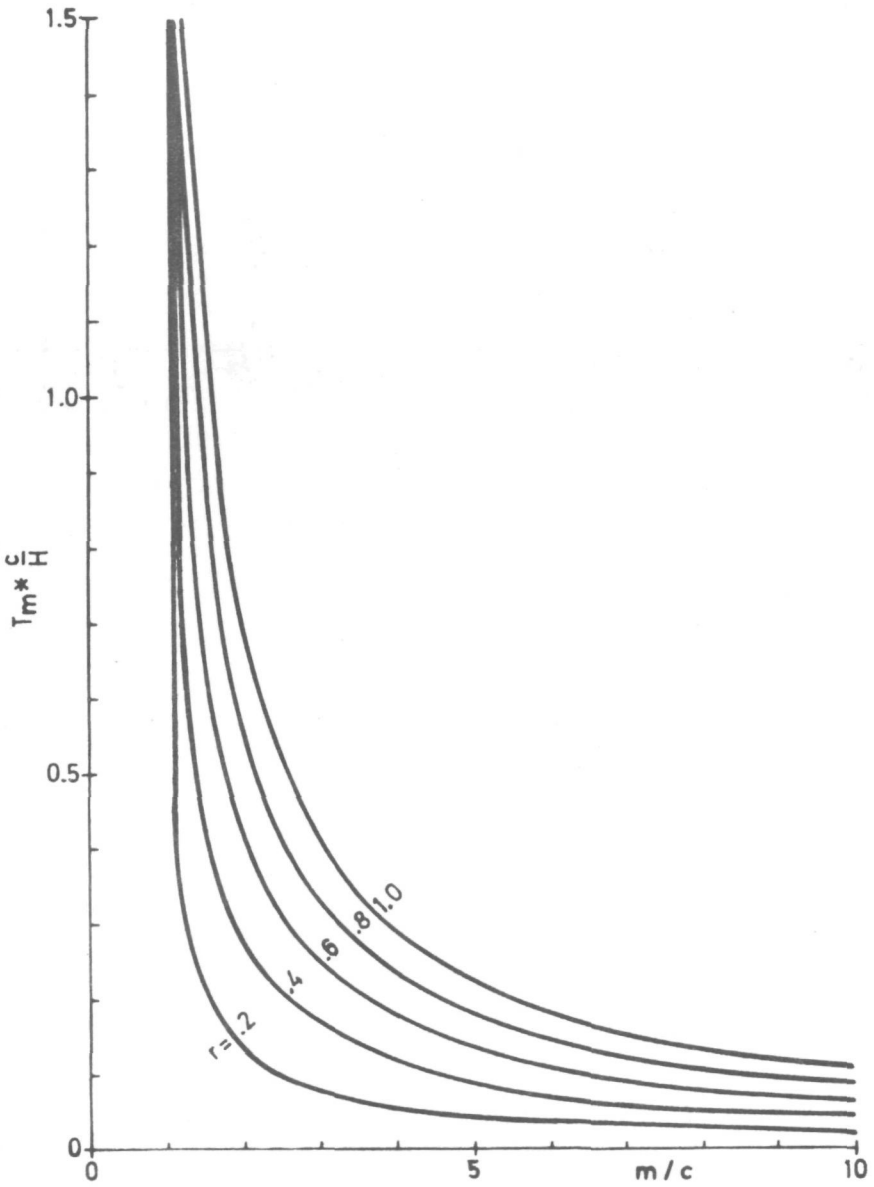


Fig. 2 - 12 THE RATIO OF MARKING TIMES FOR CONCURRENT AND SEQUENTIAL GARBAGE COLLECTION

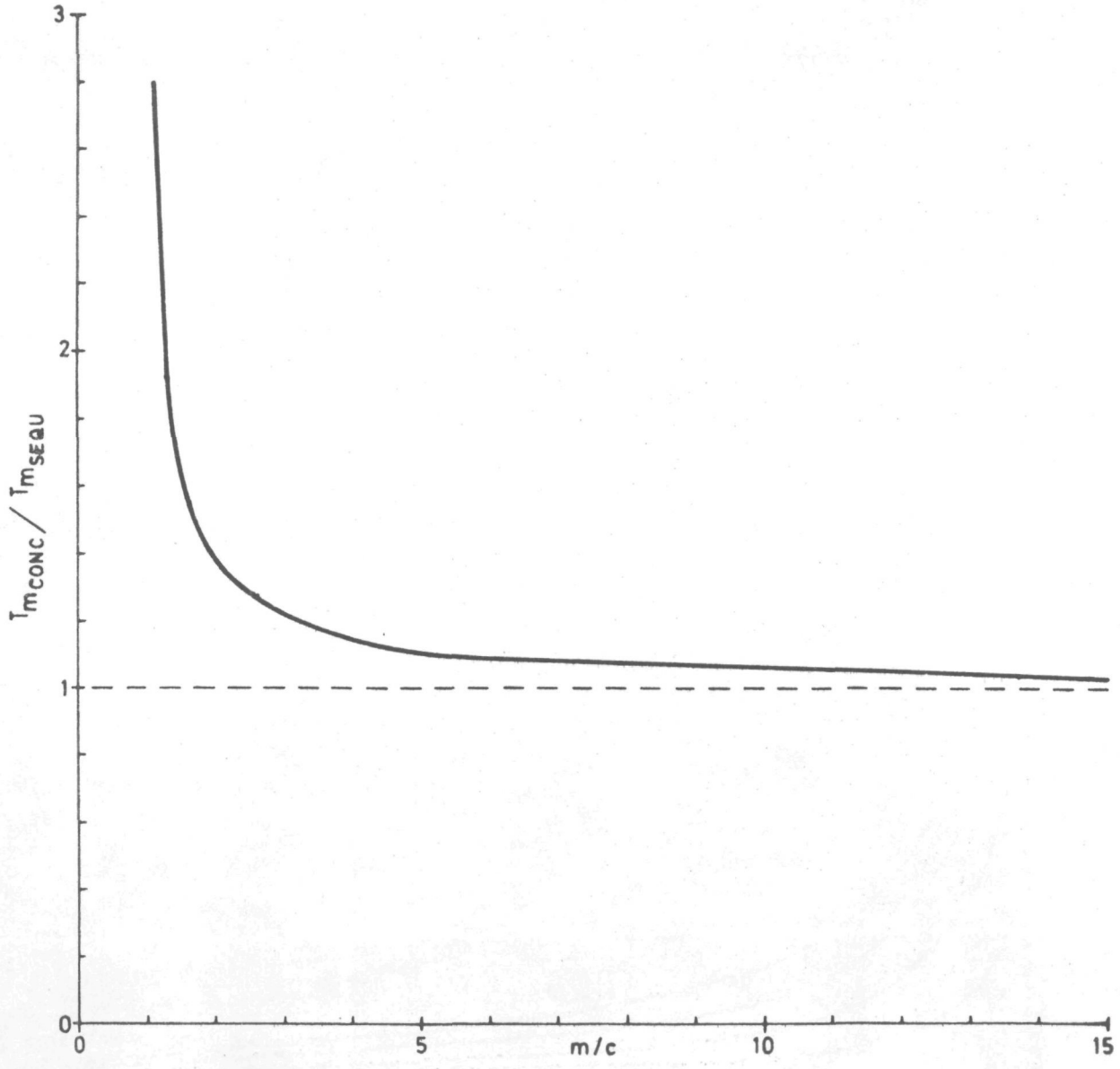


Fig. 2 - 13 LEVEL OF FREE CELLS (TRIGGER) AT WHICH TO INITIATE GARBAGE COLLECTION

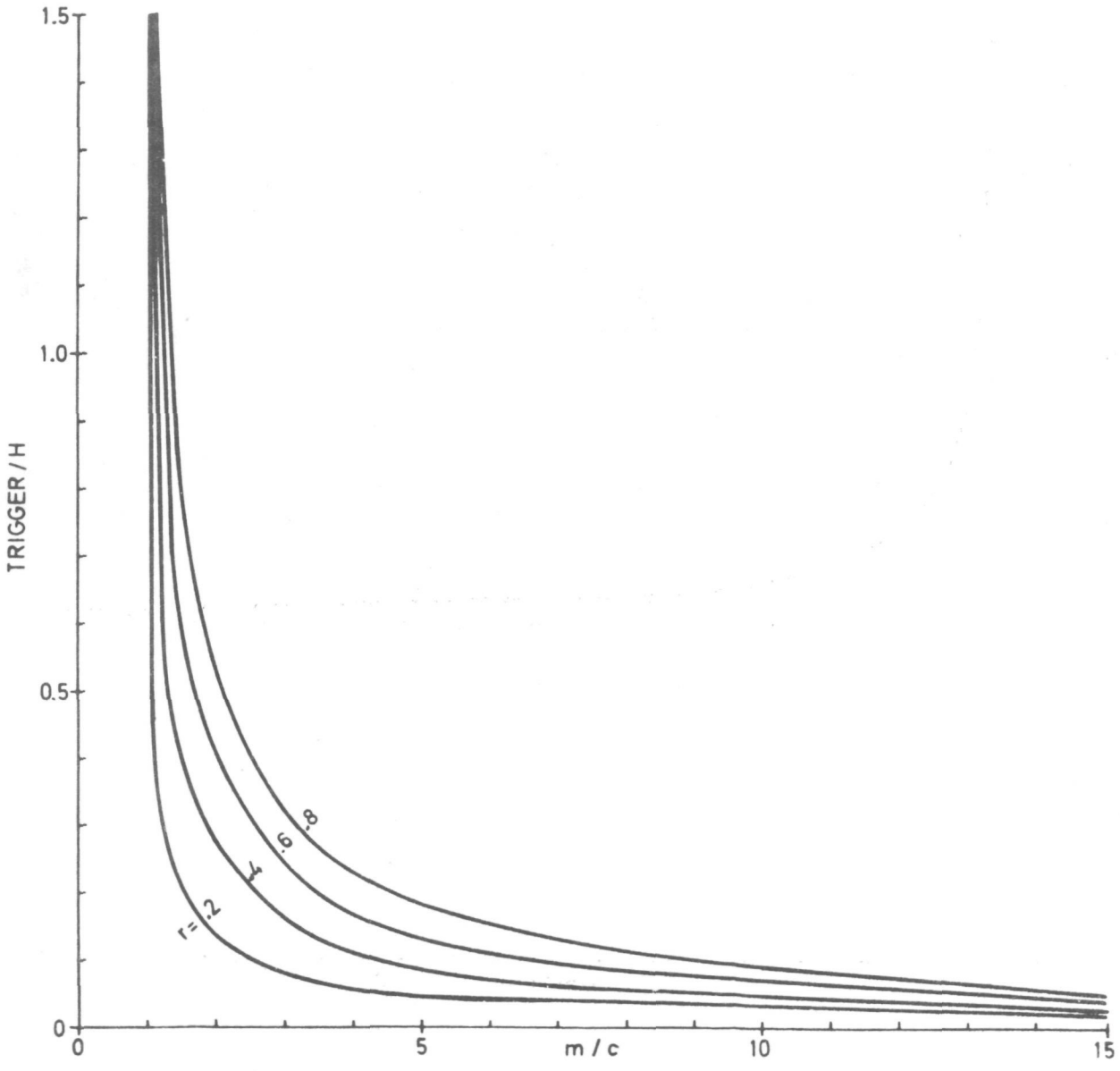


Fig. 2 - 14 THE NUMBER OF FREE CELLS DURING CONCURRENT GARBAGE COLLECTION

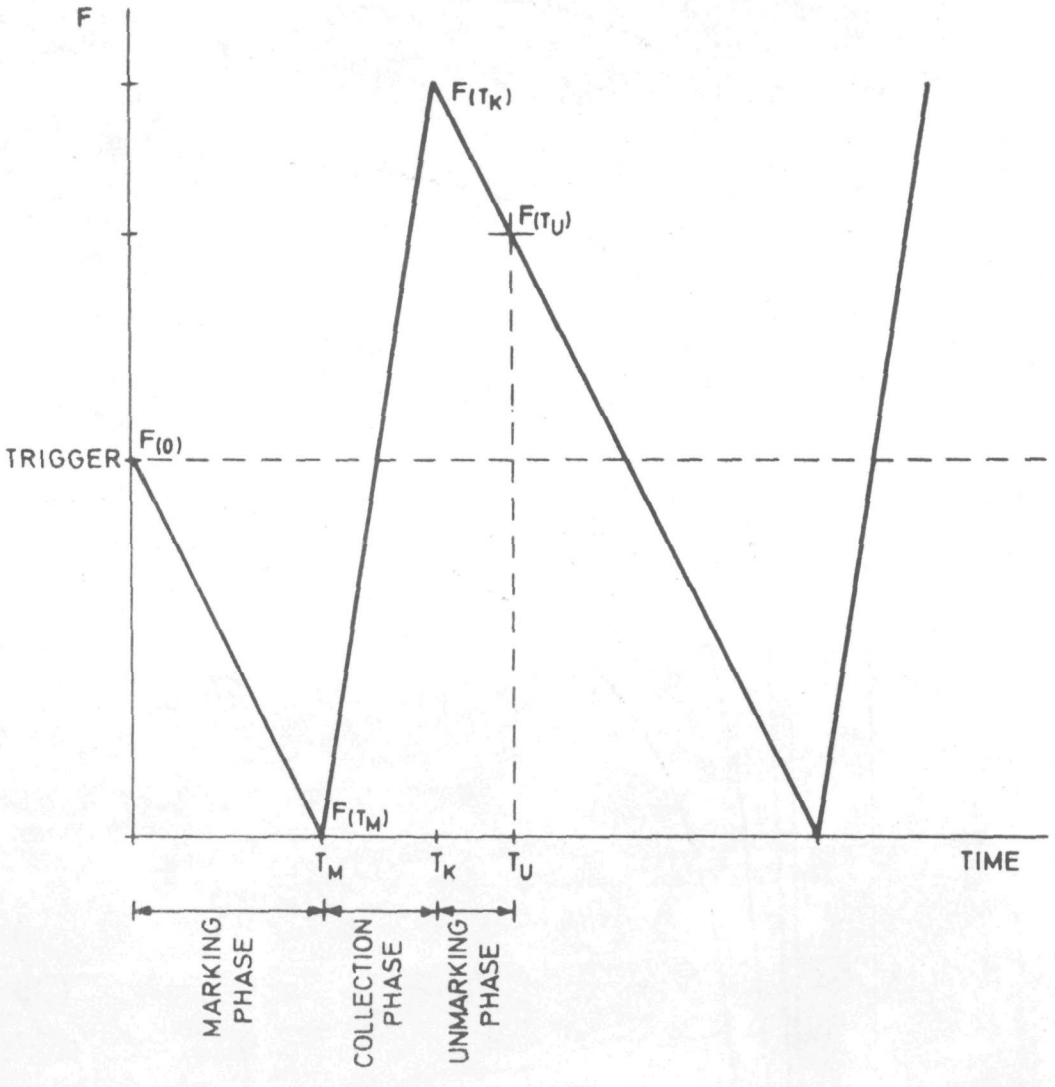


Fig. 2 - 15(a) NET GAIN OF FREE CELLS BY CONCURRENT GARBAGE COLLECTION ($u/c=\infty$; $m/k=0$)

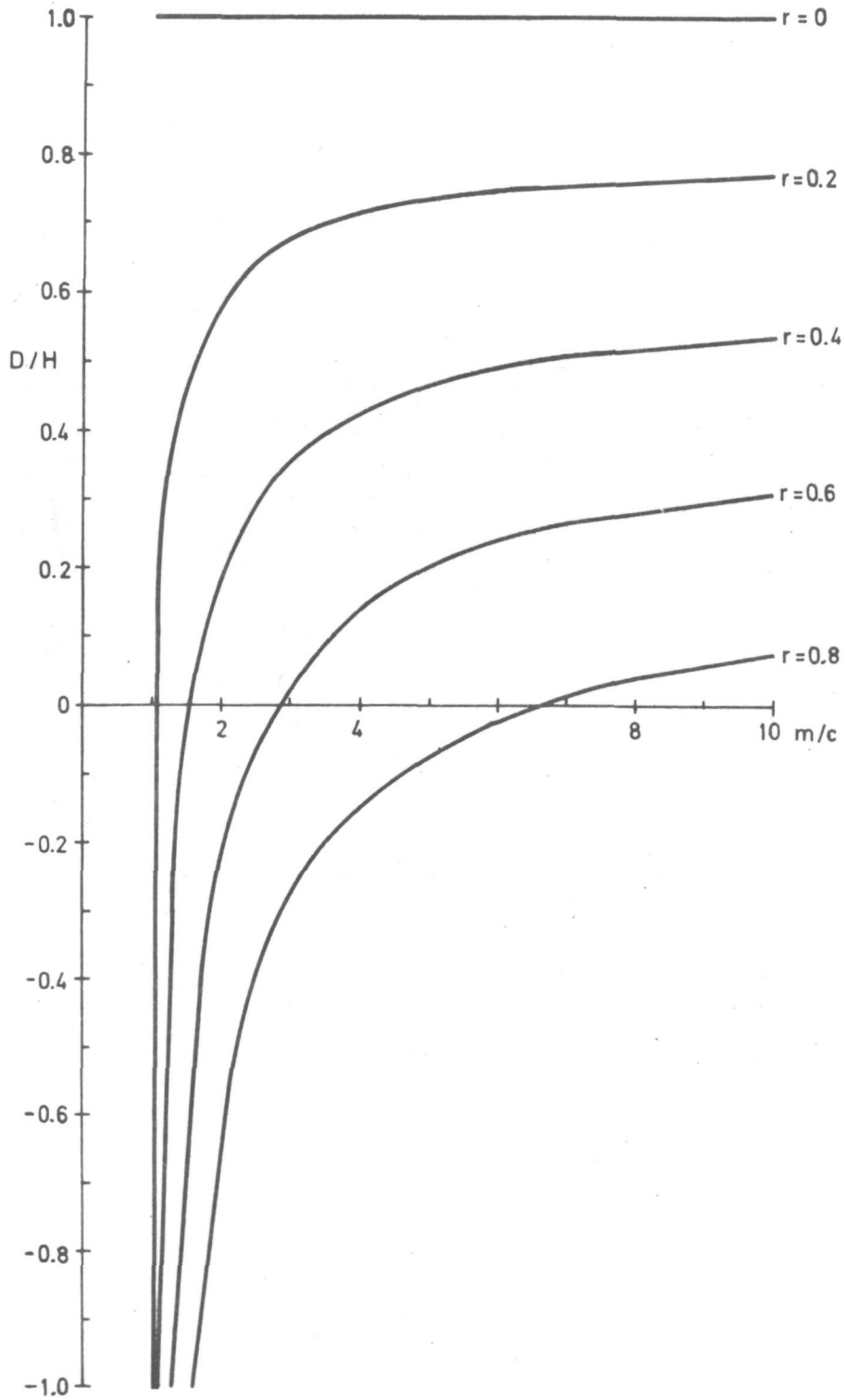


Fig. 2 - 15(b) NET GAIN OF FREE CELLS BY CONCURRENT GARBAGE COLLECTION (u/c=5; m/k=0.5)

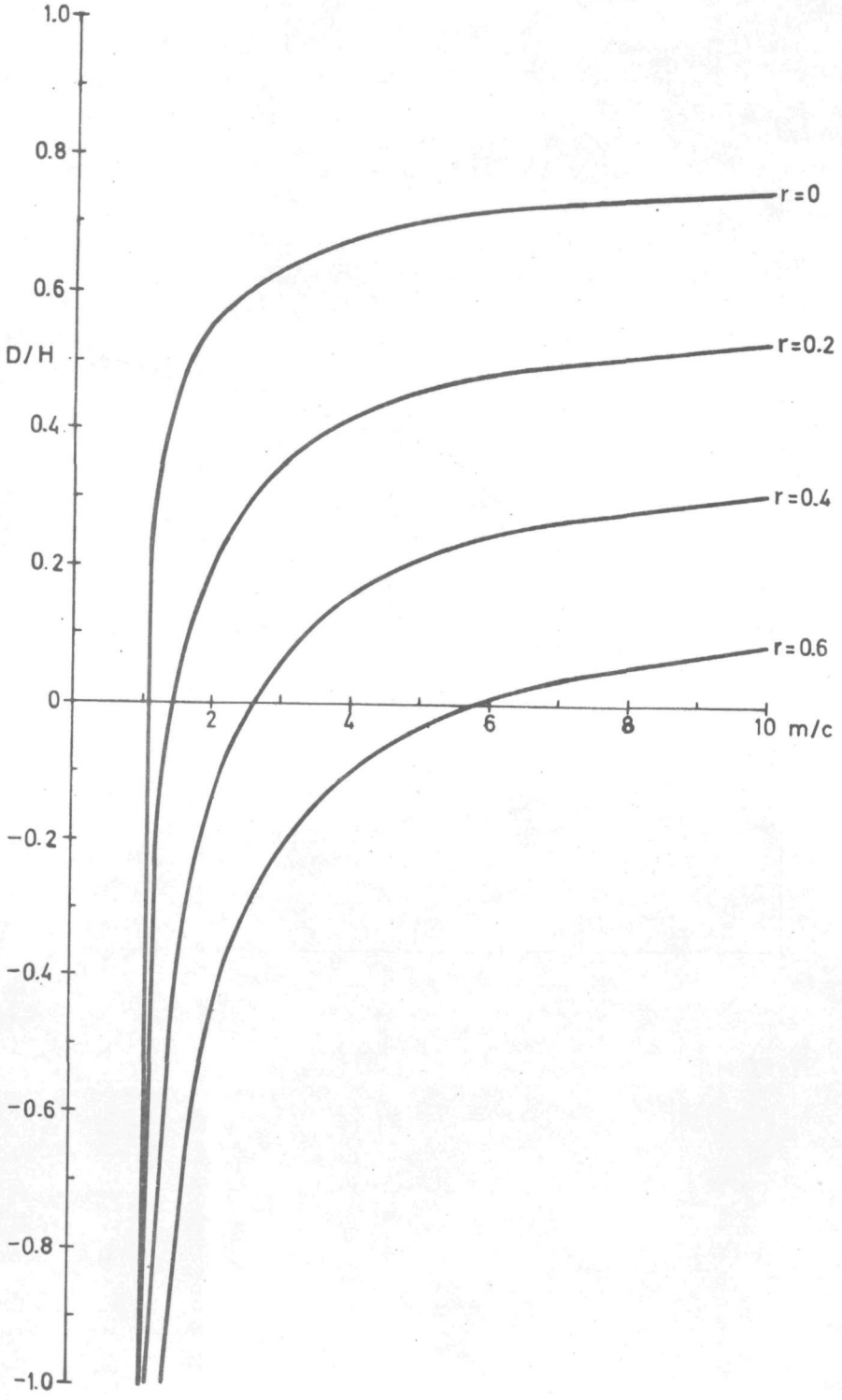


Fig. 2 - 16(a) MAXIMUM HEAP UTILISATION POSSIBLE FOR CONCURRENT GARBAGE COLLECTION ($u/c = \infty$)

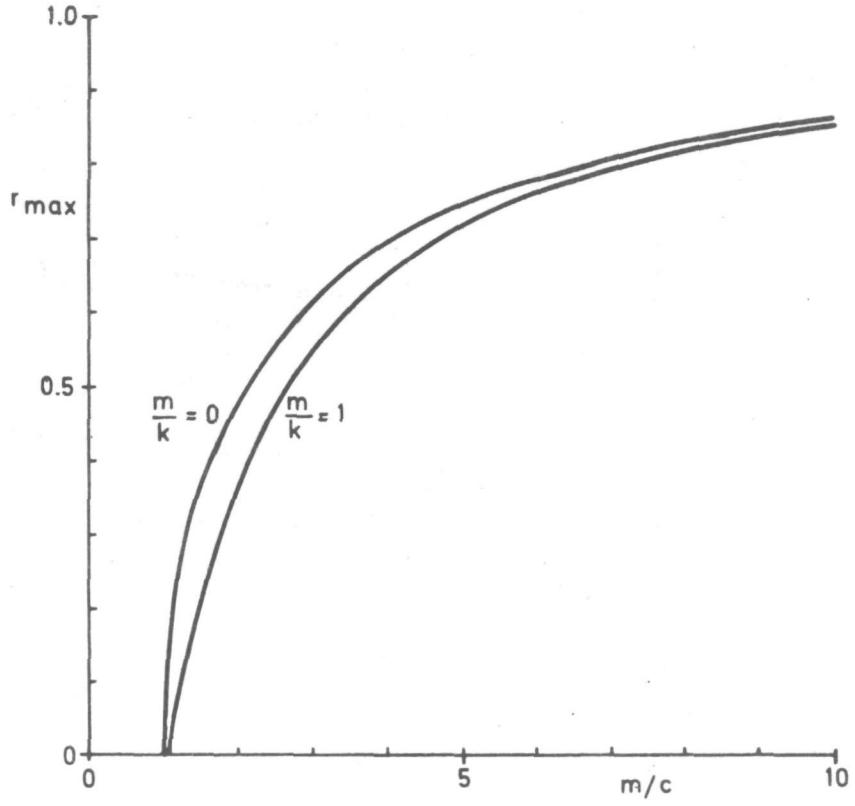


Fig. 2 - 16(b) MAXIMUM HEAP UTILISATION POSSIBLE FOR CONCURRENT GARBAGE COLLECTION (u/c=2)

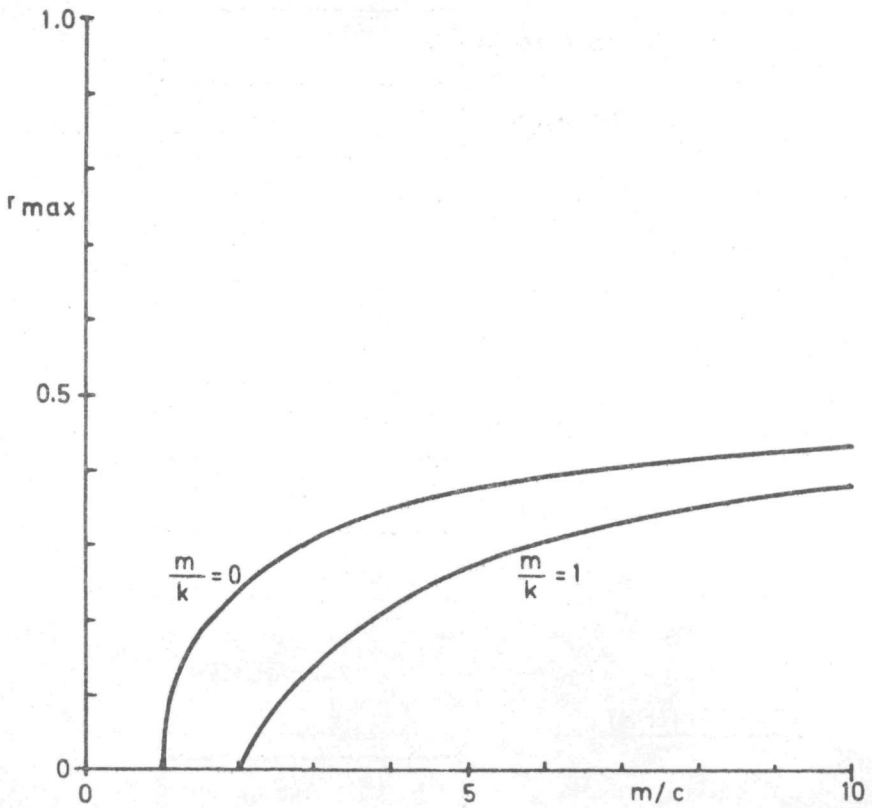


Fig. 3 - 1 MODES SUPPORTED BY NON-RELOCATING CONCURRENT GARBAGE COLLECTION SYSTEM

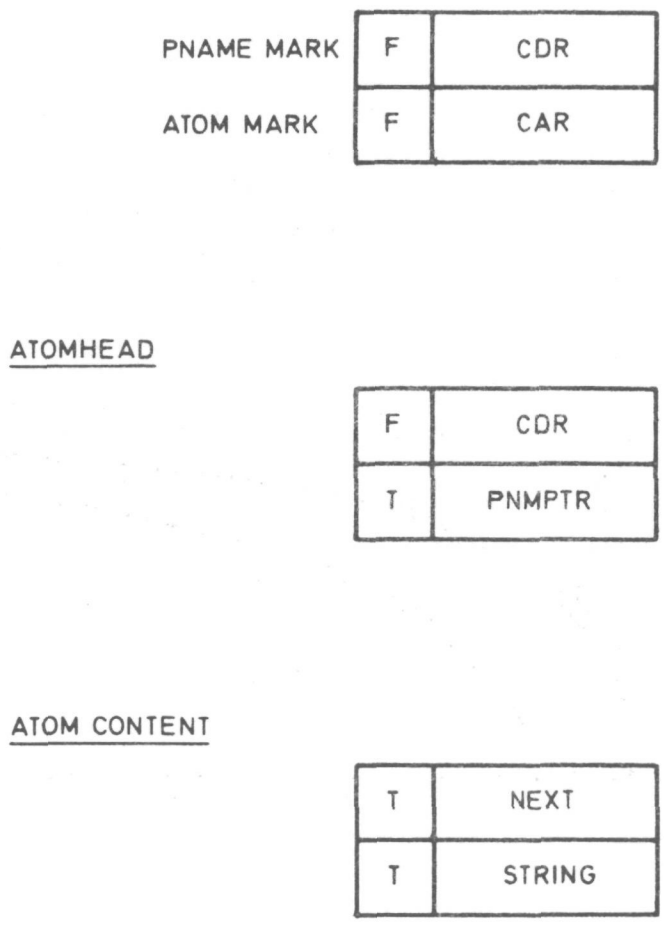


Fig. 3 - 2 EXAMPLE OF AN ATOM

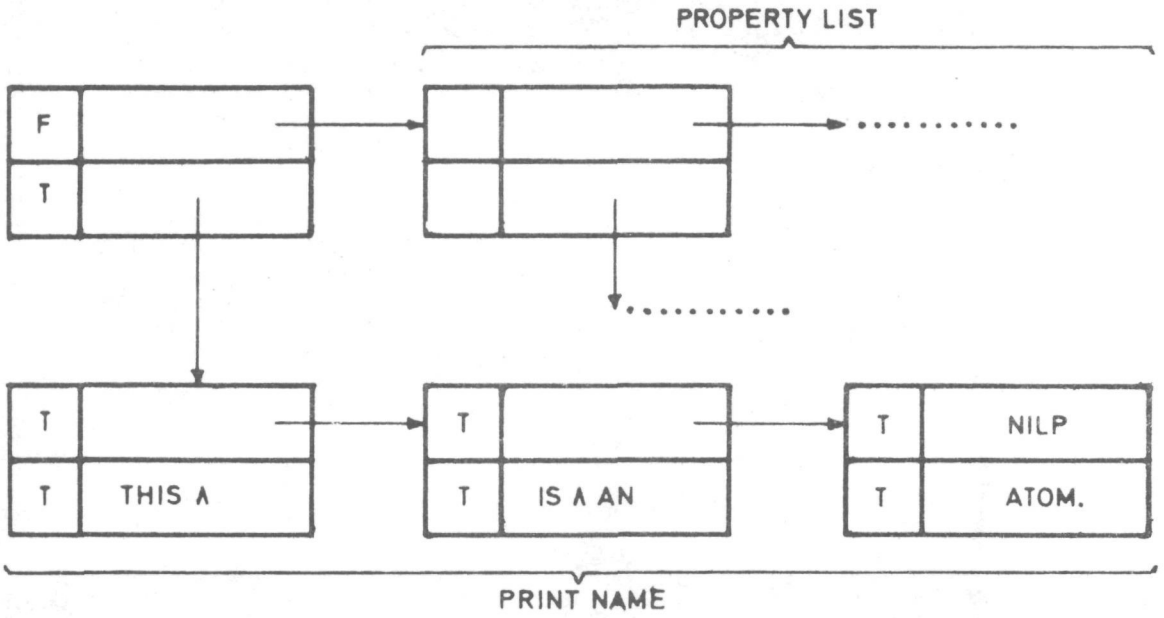


Fig. 3 - 3 STRUCTURE OF EXAMPLES OF CONCURRENT GARBAGE COLLECTION SYSTEMS

```
procedure INITIALISATION ...
procedure GC ...
procedure LISTPROCESSOR ...
begin
      INITIALISATION;
      cobegin
         LISTPROCESSOR;
         GC
      coend
end.
```

Fig. 3 - 4 CONSTANTS, TYPES, AND GLOBAL VARIABLES FOR NON-RELOCATING CONCURRENT GARBAGE COLLECTION SYSTEM

```
      const LOW = ....;    "lowest stack address"
            HIGH = ....;   "highest stack address"
            NILP = 0;    "no object"
            MAXSTACK = ....;    "maximum stacksize"
"5"         MAXCELLS = HIGH - LOW + 1;    "number of cells in heap"

      type  POINTER = 0..HIGH;    "mode of all references"
            CELL = record    "union mode of all list objects"
                   case PNAME : BOOLEAN of
                   FALSE : (CDR : POINTER;
"10"                        case ATOM : BOOLEAN of
                            FALSE : (CAR : POINTER);
                            TRUE : (PNMPTR : POINTER));
                   TRUE  : (NEXT : POINTER;
                            case AATOM : BOOLEAN of
"15"                        FALSE : (  );
                            TRUE : (STRING : .....))
                   end;
            HEAPSPACE = LOW..HIGH;    "mode of heap addresses"
            STACKSPACE = 0..MAXSTACK;    "mode of stack indices"
"20"        PDS = record    "mode of stack objects"
                  STACK : array[STACKSPACE]of POINTER;
                  TOP : STACKSPACE
                  end;
            LOCTYPE = record    "mode of stack and AUX combined"
"25"                  ST, AUX : PDS;
                      ISTACK : STACKSPACE;
                      MARKING : BOOLEAN
                      end;
            CELLINT = 0..MAXCELLS;    "mode of any cell-counter"

"30"  var   HEAP : shared array[HEAPSPACE]of CELL;    "the heap"
            LOC : shared LOCTYPE;    "list- and auxiliary stack"
            MARKED : shared array[HEAPSPACE]of BOOLEAN; "GC marks"
            FREELIST : shared record    " the free-list"
                       FREE : POINTER;
"35"                   NRFREE : CELLINT;
                       EXIT : BOOLEAN
                       end;
            TRIGGER : CELLINT;    "when to start collection"
```

Fig. 3 - 5 STRUCTURE OF LISTPROCESSOR FOR NON-RELOCATING CONCURRENT GARBAGE COLLECTION SYSTEM

```
      procedure LISTPROCESSOR;
         var LPEXIT : BOOLEAN;
         procedure PUSH (var S : LOCTYPE; ARG : POINTER);
         begin
"5"         with S, ST do
            begin
               STACK[TOP]:= ARG;
               TOP := TOP + 1
            end
"10"     end;
         procedure POP (var S : LOCTYPE; var RESULT : POINTER);
         begin
            with S, ST do
            begin
"15"           TOP := TOP - 1;
               RESULT := STACK[TOP];
               if MARKING then ISTACK := MIN(TOP, ISTACK)
            end
         end;
"20"     procedure CAR;
            var TEMP : HEAPSPACE;
         begin
            region LOC do
            begin
"25"           POP(ST, TEMP);
               region HEAP do
                  PUSH(ST,HEAP[TEMP].CAR)
            end
         end;
"30"     procedure CONS;
            var TEMPA, TEMPD : POINTER; TEMPC : HEAPSPACE;
         begin region FREELIST do
               begin await (NRFREE>0) or EXIT;
                     LPEXIT := EXIT;
"35"                 if not LPEXIT then
                     region LOC do
                     begin
                        TEMPC := FREE;
                        POP(ST, TEMPA);
"40"                    POP(ST, TEMPD);
```

Fig. 3 - 5 (continued)

```
                        region HEAP do
                        with HEAP[TEMPC] do
                        begin
                           FREE := CDR;
"45"                       NRFREE := NRFREE - 1;
                           ATOM := FALSE;
                           CAR := TEMPA;
                           CDR := TEMPD
                        end;
"50"                    PUSH(ST, TEMPC)
                     end
               end
         end;
         procedure RPLACA;
"55"        var REFREF : HEAPSPACE; REF : POINTER;
         begin
            region LOC do
            begin
               POP(ST, REFREF);
"60"           POP(ST, REF);
               region HEAP do
               begin
                  HEAP[REFREF].CAR := REF; PUSH(ST, REFREF);
                  if MARKING
"65"              then region MARKED do
                       if MARKED[REFREF] and REF ≥ LOW
                       then if not MARKED[REF]
                            then with AUX do
                                 begin
"70"                                STACK[TOP]:=REF;
                                    TOP := TOP + 1
                                 end
               end
            end
"75"     end;
      begin   "main body of list processor"
         repeat
         ....
            region FREELIST do
"80"        begin EXIT := EXIT or   "processing accomplished";
                  LPEXIT := EXIT
            end
         until LPEXIT
      end ;
```

Fig. 3 - 6 STRUCTURE OF NON-RELOCATING CONCURRENT GARBAGE COLLECTOR

```
        procedure GC;
           var GCEXIT, DONE : BOOLEAN;
               NEXTPTR, TEMP : POINTER;
           procedure MARK(PAR : POINTER);
"5"           var NOTYET, MORE, ATOMIC : BOOLEAN;
                  CAND : POINTER;
              procedure MARKATOM(AT : POINTER);
                 var ITER : POINTER; MORE : BOOLEAN;
              begin
"10"             ITER := AT;
                 region HEAP do
                 begin
                    region MARKED do
                    begin
"15"                   if ITER≥LOW then MORE:=not MARKED[ITER]
                       else MORE := FALSE;
                       while MORE do
                       begin
                          MARKED[ITER]:=TRUE;
"20"                      ITER:=HEAP[ITER].NEXT;
                          if ITER≥LOW then MORE:=not MARKED[ITER]
                          else MORE:=FALSE
                       end
                    end
"25"             end
              end;
           begin
              if PAR≥LOW then
              begin
"30"             region MARKED do
                 begin
                    NOTYET:=MARKED[PAR];
                    if NOTYET then MARKED[PAR]:=TRUE
                 end;
"35"             if NOTYET then
                 begin
                    region HEAP do
                    begin
                       ATOMIC:=HEAP[PAR].ATOM;
"40"                   CAND:=HEAP[PAR].CAR;
                    end;
```

Fig. 3 - 6 (continued)

```
                    if ATOMIC then MARKATOM(CAND)
                    else MARK(CAND);
                    region HEAP do CAND:=HEAP[PAR].CDR;
"45"                region MARKED do
                       if CAND ≥ LOW then MORE:=not MARKED[CAND]
                       else MORE:=FALSE;
                    while MORE do
                    begin region MARKED do MARKED[CAND]:=TRUE;
"50"                      region HEAP do
                          begin ATOMIC:=HEAP[CAND].ATOM
                                NOW:=HEAP[CAND].CAR
                          end;
                          if ATOMIC then MARKATOM(NOW)
"55"                      else MARK(NOW);
                          region HEAP do CAND:=HEAP[CAND].CDR;
                          region MARKED do
                             if CAND≥LOW then MORE:=not MARKED[CAND]
                             else MORE:=FALSE
"60"                end
                 end
              end
           end;

           procedure SAVEFLIST;
"65"       begin region FREELIST do
                 begin TEMP:=FREE;
                       FREE:=NILP;
                       NRFREE:=0
                 end
"70"       end;

           procedure MARKFLIST;
              var P : POINTER;
           begin while TEMP ≠ NILP do
                 region FREELIST do
"75"             begin P:=FREE;
                       FREE:=TEMP;
                       region HEAP do
                       begin TEMP:=HEAP[TEMP].CDR;
                          HEAP[FREE].CDR:=P;
"80"                      NRFREE:=NRFREE+1;
                          region MARKED do MARKED[FREE]:=TRUE
                       end
                 end
           end;
```

Fig. 3 - 6 (continued)

```
"85"      procedure COLLECT;
             var RUNNER : HEAPSPACE; GARB : BOOLEAN;
          begin for RUNNER:=LOW to HIGH do
                begin region MARKED do GARB:=not MARKED[RUNNER];
                      if GARB then
"90"                    region FREE LIST do
                           region HEAP do
                           begin HEAP[RUNNER].CDR:=FREE; FREE:=RUNNER;
                                 NRFREE:=NRFREE+1
                           end
"95"                end;
                    region FREELIST do begin EXIT:=EXIT or (NRFREE=0);
                                             GCEXIT:=EXIT
                                       end
          end;
"100"     procedure UNMARK;
             var RUNNER : HEAPSPACE
          begin region MARKED do
                   for RUNNER:=LOW to HIGH do MARKED[RUNNER]:=FALSE
          end;
"105"  begin   "main body of GC"
          repeat region FREELIST do
                 begin await NRFREE<TRIGGER or EXIT;
                       GCEXIT:=EXIT
                 end;
"110"            if not GCEXIT then
                 begin region LOC do MARKING:=TRUE;
                       DONE:=FALSE;
                       repeat
                          region LOC do
"115"                     if AUX.TOP>0 then
                              with AUX do
                              begin TOP:=TOP-1;
                                    NEXT:=STACK[TOP]
                              end
"120"                     else if ISTACK<ST.TOP then
                                 begin NEXT:=ST.STACK[ISTACK];
                                       ISTACK:=ISTACK+1
                                 end
                          else begin MARKING:=FALSE;
"125"                                ISTACK:=0;
                                     SAVEFLIST;
                                     DONE:=TRUE
                               end;
```

Fig. 3 - 6 (continued)

```
                        if not DONE then MARK(NEXT)
"130"                  until DONE;
                       MARKFLIST;
                       COLLECT;
                       UNMARK
        until GCEXIT
"135"  end;
```

Fig. 3 - 7 INITIALISATION PROCEDURE FOR NON-RELOCATING CONCURRENT GARBAGE COLLECTOR

```
      procedure INITIALISATION;
         var RUNNER : HEAPSPACE; P : POINTER;
      begin if LOW>HIGH then ERRORSTOP;
         region FREELIST do
"5"      begin
            region MARKED do
            begin
               region HEAP do
               begin NRFREE:=0;
"10"                 P:=NILP;
                     for RUNNER:=LOW to HIGH do
                     begin MARKED [RUNNER]:=FALSE;
                           FREE:=RUNNER;
                           HEAP [FREE].CDR:=P;
"15"                       P:=FREE;
                           NRFREE:=NRFREE+1
                     end
               end
            end
"20"     end;
         region LOC do
         begin ISTACK:=0;
               AUX.TOP:=0;
               ST.TOP:=0;
"25"           region FREELIST do EXIT:=NRFREE=0;
               MARKING:=FALSE
         end;
         TRIGGER:= ...
         if TRIGGER≤0 then ERRORSTOP
"30"  end;
```

Fig. 3 - 8 MODES SUPPORTED BY CONCURRENT COMPACTIFYING GARBAGE COLLECTION SYSTEM

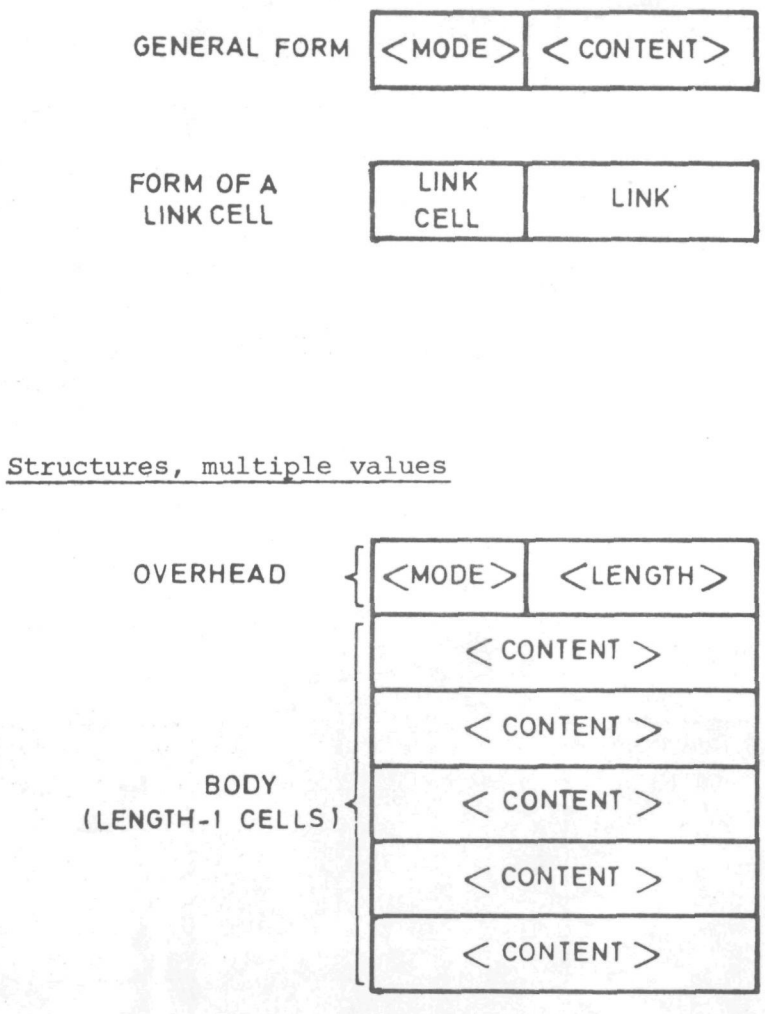
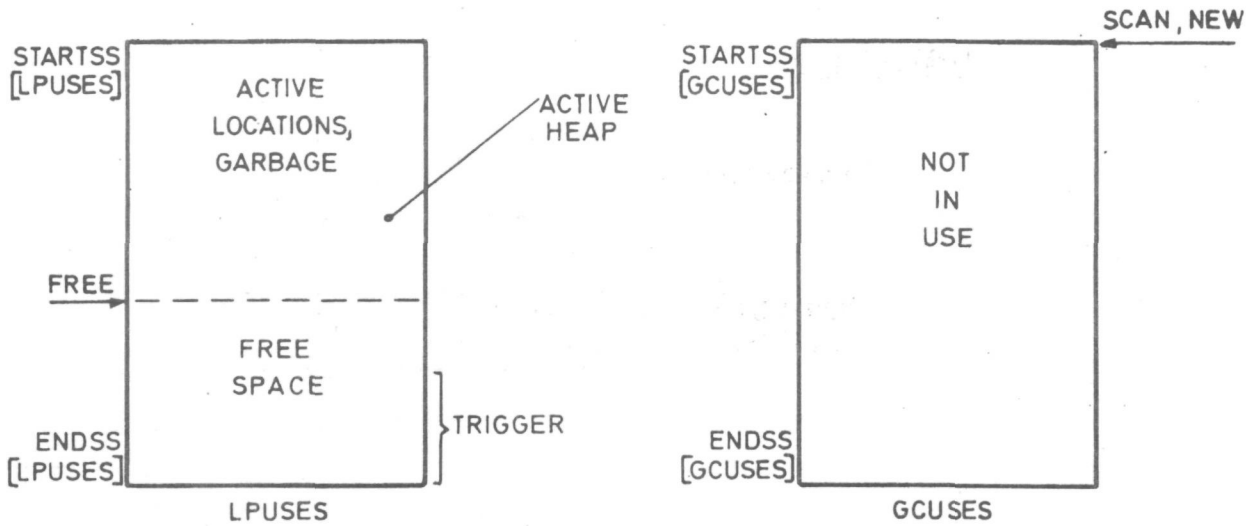


Fig. 3 - 9 STRUCTURE OF HEAP FOR CONCURRENT COMPACTIFYING GARBAGE COLLECTION SYSTEM

a) No collection going on



b) During collection

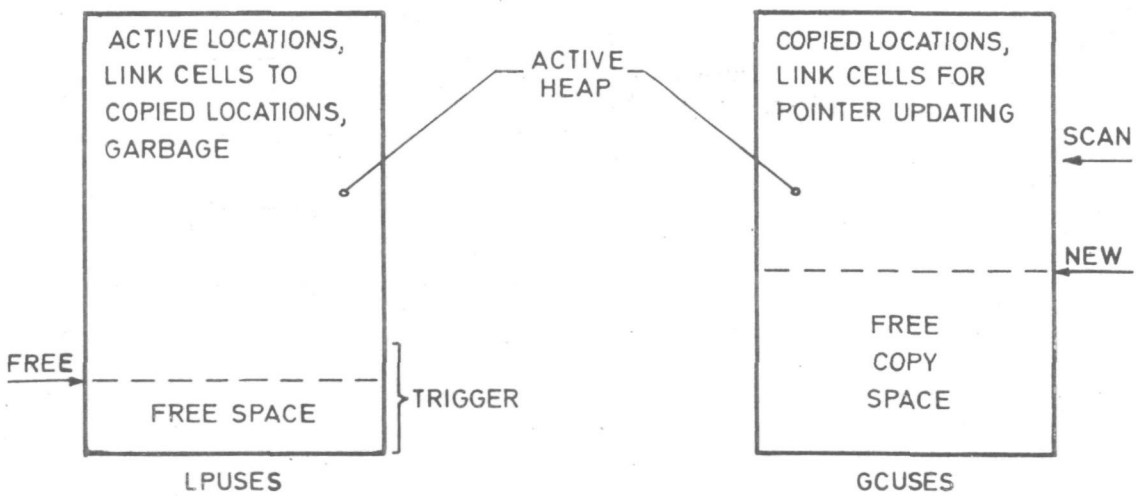


Fig. 3 - 10 THE FUNCTION OF LINKCELLS IN SEMISPACE "GCUSES"

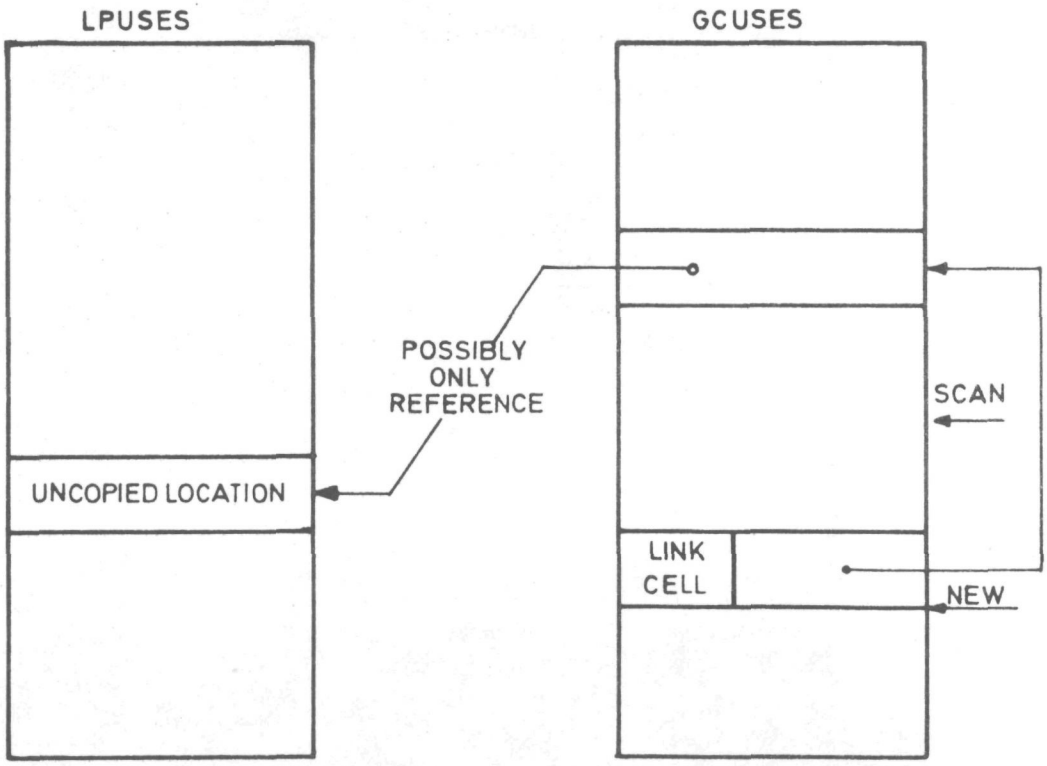


Fig. 3 - 11 OVERALL STRUCTURE OF CONCURRENT COMPACTIFYING GARBAGE COLLECTION SYSTEM

```
"Declarations of constants, modes, and global
 variables"
procedure INITIALISATION ...
function SIMPLE (...) : BOOLEAN ...
function APOINTER (...) : BOOLEAN ...
function LONG (...) ...
function INSEMISP (...) : BOOLEAN ...
procedure LISTPROCESSOR ...
procedure GC ...
begin
     INITIALISATION;
     cobegin
        LISTPROCESSOR;
        GC
     coend
end.
```

Fig. 3 - 12 CONSTANTS, MODES, AND GLOBAL VARIABLES FOR CONCURRENT COMPACTIFYING GARBAGE COLLECTION SYSTEM

```
      const LOW = ...;  "lowest heap address"
            HIGH = ...;  "highest heap address"
            NILP = 0;  "no object"
            MAXLENGTH = ...;  "maximum size of a location"
"5"         MAXSTACK = ...;  "maximum value of a stackpointer"
            NRCELLS=(HIGH-LOW+1)/2;  "number of cells in semi-space"

      type  POINTER = NILP..HIGH;  "mode of pointers"
            HEAPSPACE = LOW..HIGH;  "mode of heap address"
            STACKSPACE = 0..MAXSTACK;  "mode of stackpointers"
"10"        SEMISPACE = (SS1, SS2);  "mode of semi-spaces"
            MODE = (LINKCELL,TYPE1,TYPE2, ...TYPEn);  "modes of locations"
            CELLTYPE = (SINGLE,OVERHEAD,BODY);  "mode of celltypes"
            CELL = packed record  "united mode of cells"
                   case CELLTYPE of
"15"               SINGLE : (KIND : MODE;
                             case MODE of
                             LINKCELL : (LINK : POINTER);
                             TYPE1,TYPE2, ...:(CONTENT: ...));
                   OVERHEAD : (LOCKIND : MODE;
"20"                           LENGTH : 2..MAXLENGTH);
                   BODY : (CONTEN : ...)
                   end;
            PDS = record  "mode of stacks"
                  STACK : array [STACKSPACE] of POINTER;
"25"              TOP : STACKSPACE
                  end;
            DATABASE = record  "mode of all shared data"
                       ST : PDS;
                       EXIT, MARKING : BOOLEAN;
"30"                   ISTACK : STACKSPACE;
                       NRNEEDED : 1..MAXLENGTH;
                       HEAP : array [HEAPSPACE] of CELL;
                       LPUSES, GCUSES : SEMISPACE;
                       FREE, SCAN, NEW : LOW..HIGH + 1
"35"                   end;
      var   ALL : shared DATABASE;  "all shared data"
            STARTSS, ENDSS : array [SEMISPACE] of HEAPSPACE;
                             "first and last address of semi-spaces"
            TRIGGER : 0..NRCELLS;
```

Fig. 3 - 13 INITIALISATION PROCEDURE FOR CONCURRENT COMPACTIFYING GARBAGE COLLECTION SYSTEM

```
      procedure INITIALISATION;
      begin if HIGH + 1 ≤ LOW then ERRORSTOP
            else begin    STARTSS[SS1]:=LOW;
                          STARTSS[SS2]:=LOW+(HIGH-LOW+1) div 2;
"5"                       ENDSS[SS1]:=STARTSS[SS2]-1;
                          ENDSS[SS2]:=HIGH;
                          region ALL do
                          begin LPUSES := SS1;
                                GCUSES := SS2;
"10"                            EXIT   := FALSE;
                                MARKING:= FALSE;
                                ISTACK := 0;
                                ST.TOP := 0;
                                FREE   := START[LPUSES];
"15"                            SCAN   := START[GCUSES];
                                NEW    := START[GCUSES];
                                NRNEEDED := 0
                          end;
                          TRIGGER := ...;
"20"                      if TRIGGER < MAXLENGTH then ERRORSTOP
                 end
      end;
```

Fig. 3 - 14 OUTLINE OF AUXILIARY FUNCTIONS FOR CONCURRENT COMPACTIFYING GARBAGE COLLECTION SYSTEM

```
      function SIMPLE(M : MODE) : BOOLEAN;
      begin
         SIMPLE := LONG(M) = 1
      end;
"5"   function APOINTER(M:MODE; OFFSET:0..MAXLENGTH) : BOOLEAN;
      begin
         APOINTER:=   "cell at position OFFSET from start of
                    location of mode M contains a pointer"
      end;
      function LONG(M : MODE) : 0..MAXLENGTH;
"10"  begin
         LONG := "number of cells (inc. overhead) in location of
                  mode M"
      end;
      function INSEMISP(PTR : POINTER; SEM : SEMISPACE) : BOOLEAN;
      begin
"15"     INSEMISP := PTR ≥ STARTSS[SEM] and PTR ≤ ENDSS[SEM]
      end;
```

Fig. 3 - 15 OUTLINE OF LIST PROCESSOR IN CONCURRENT COMPACTIFYING GARBAGE COLLECTION SYSTEM

```
      procedure LISTPROCESSOR;
         var LPEXIT : BOOLEAN;

         procedure PUSH(var A : DATABASE; ARG : POINTER);
         begin with A,ST do
"5"         begin STACK[TOP]:=ARG;
               TOP:=TOP+1
            end
         end;

         procedure POP(var A : DATABASE; var RES:POINTER);
"10"     begin with A,ST do
            begin TOP:=TOP-1;
               RES:=STACK[TOP];
               if MARKING then ISTACK:=MIN(ISTACK,TOP)
            end
"15"     end;

         procedure CONS(MDE : MODE);
            var I, NRPARS : 1..MAXLENGTH;
         begin region ALL do
            begin NRNEEDED:=LONG(MDE);
"20"           await FREE+NRNEEDED≤ENDSS[LPUSES]+1 or EXIT;
               if EXIT then LPEXIT:=TRUE
               else
               begin with HEAP[FREE]do
                  begin
"25"                 KIND:=MDE;
                     if not SIMPLE(MDE) then
                     begin LENGTH:=LONG(MDE);
                        NRPARS:=LENGTH-1;
                        for I:=1 to NRPARS do
"30"                       POP(ALL,HEAP[FREE+I].CONTEN)
                     end
                     else POP(ALL,CONTENT)
                  end;
                  PUSH(ALL,FREE);
"35"              FREE:=FREE+NRNEEDED;
                  NRNEEDED:=0
               end
            end
         end;
```

Fig. 3 - 15 (continued)

```
"40"     procedure SAVE(var A:DATABASE;LHS:HEAPSPACE;RHS:POINTER);
         begin
            with A do
            if INSEMISP(LHS,GCUSES)and LHS<SCAN and INSEMISP(RHS,LPUSES)
            then
"45"        begin if NEW>ENDSS[GCUSES]then begin EXIT:=TRUE;
                                                 LPEXIT:=TRUE
                                           end
                  else
                  begin with HEAP[NEW]do
"50"                    begin KIND:=LINKCELL; LINK:=LHS end;
                        NEW:=NEW+1
                  end
            end
         end;
"55"     procedure ASSIGNMENT;
            var LHS:HEAPSPACE; I,NRVALUES:1..MAXLENGTH;
                TEMP:POINTER;
         begin region ALL do
            begin POP(ALL,LHS);
"60"           if HEAP[LHS].KIND=LINKCELL then LHS:=HEAP[LHS].LINK;
               if SIMPLE(HEAP[LHS].KIND) then
               begin POP(ALL,HEAP[LHS].CONTENT);
                  if APOINTER(HEAP[LHS].KIND,0) then
                     if MARKING then SAVE(ALL,LHS,HEAP[LHS].CONTENT)
"65"           end
               else
               begin
                  NRVALUES:=LONG(HEAP[LHS].KIND)-1;
                  for I:=1 to NRVALUES do
"70"              begin POP(ALL,HEAP[LHS+I].CONTEN);
                     if APOINTER(HEAP[LHS].KIND,I) then
                        if MARKING then SAVE(ALL,LHS+I,HEAP[LHS+I].CONTEN)
                  end
               end
"75"        end
         end;
      begin
         repeat
            ....
"80"        region ALL do
            begin EXIT:=EXIT or  "processing accomplished";
               LPEXIT:=EXIT
            end
         until LPEXIT
"85"  end;
```

Fig. 3 - 16 CONCURRENT COMPACTIFYING GARBAGE COLLECTOR

```
      procedure GC;
         var GCEXIT:BOOLEAN;
             OFFSET,CURLENGTH:0..MAXLENGTH;
             TEMP:SEMISPACE;
 "5"         CURMODE:MODE;
             REF:POINTER;
             MARKED:array[HEAPSPACE]of BOOLEAN;

         procedure COPY(var D:DATABASE; var P:POINTER);
            var I:0..MAXLENGTH-1;SIZE:1..MAXLENGTH;
"10"     begin
            with D do
            begin if SIMPLE(HEAP[P].KIND) then SIZE:=1
                  else SIZE:=HEAP[P].LENGTH;
                  if NEW+SIZE≤ENDSS[GCUSES]+1 then
"15"              begin for I:=0 to SIZE-1 do HEAP[NEW+I]:=HEAP[P+I];
                     with HEAP[P]do
                     begin KIND:=LINKCELL; LINK:=NEW end;
                     MARKED[P]:=TRUE;
                     P:=NEW;
"20"                 NEW:=NEW+SIZE
                  end
                  else begin GCEXIT:=TRUE; EXIT:=TRUE end
            end
         end;
"25"     procedure UNMARK;
            var RUNNER:HEAPSPACE;
         begin for RUNNER:=LOW to HIGH do MARKED[RUNNER]:=TRUE end;

      begin UNMARK;
         repeat
"30"        region ALL do
            begin await ENDSS[LPUSES]-FREE+1<TRIGGER or EXIT;
               GCEXIT:=EXIT;
               MARKING:=TRUE;
               OFFSET:=0;
"35"           DONE:=FALSE
            end;
```

Fig. 3 - 16 (continued)

```
          repeat
          region ALL do
             if SCAN=NEW then
"40"         begin if ISTACK<TOP then
                begin
                   if BUILTIN(STACK[ISTACK])then "skip"
                   else if INSEMISP(STACK[ISTACK],GCUSES) then "skip"
                   else if MARKED[STACK[ISTACK]] then
"45"                              STACK[ISTACK]:=HEAP[STACK[ISTACK]].LINK
                   else COPY(ALL,STACK[ISTACK]);
                   ISTACK:=ISTACK+1
                end
                else begin FREE:=NEW;
"50"                    NEW:=START[LPUSES];
                        SCAN:=START[LPUSES];
                        TEMP:=LPUSES;
                        LPUSES:=GCUSES;
                        GCUSES:=TEMP;
"55"                    ISTACK:=TEMP;
                        MARKING:=FALSE;
                        DONE:=TRUE;
                        EXIT:=ENDSS[LPUSES]-FREE+1<NRNEEDED or EXIT;
                        GCEXIT:=EXIT
"60"                 end
             end
             else
             begin if OFFSET=0 then begin CURMODE:=HEAP[SCAN].KIND;
                                          if SIMPLE(CURMODE)then CURLENGTH:=1
"65"                                      else CURLENGTH:=HEAP[SCAN].LENGTH
                                     end;
                if CURMODE=LINKCELL then
                begin REF:=HEAP[SCAN].LINK;
                   if INSEMISP(HEAP[REF].PTR,GCUSES) then "skip"
"70"               else if MARKED[HEAP[REF].PTR] then
                                HEAP[REF].PTR:= HEAP[HEAP[REF].PTR].LINK
                   else COPY(ALL,HEAP[REF].PTR)
                end
                else if APOINTER(CURMODE,OFFSET) then
"75"                    if BUILTIN(HEAP[SCAN].PTR) then "skip"
                        else if INSEMISP(HEAP[SCAN].PTR,GCUSES)then"skip"
                        else if MARKED[HEAP[SCAN].PTR] then
                                HEAP[SCAN].PTR:=HEAP[HEAP[SCAN].PTR].LINK
                        else COPY(ALL,HEAP[SCAN].PTR);
"80"            SCAN:=SCAN+1;
                if OFFSET=CURLENGTH-1 then OFFSET:=0
             end
          until DONE or GCEXIT;
```

Fig. 3 - 16 (continued)

```
            if not GCEXIT then UNMARK;
"85"        region ALL do
            begin EXIT:=EXIT or GCEXIT;
                  GCEXIT:=EXIT
            end
          until GCEXIT
"90" end;
```

Appendix A
THE LANGUAGE USED

The language which has been used in this thesis is PASCAL as defined in the Revised Report (Je 74), extended by the language constructs proposed by Brinch Hansen (Ha 73, Ha 73a) for concurrent processes.

This appendix has been incorporated in the thesis for the benefit of readers without access to the defining documents referenced above. It provides a brief description of syntax and semantics of Revised Report PASCAL and of Brinch Hansen's extensions.

Structure of a PASCAL program

```
const <constant definitions>
type <type definitions>
var <variable definitions>
procedure <procedure definition> "may be repeated"
function <function definition> "may be repeated"
begin
      <statement part>
end
```

Procedures and functions have the same structure as given above.

Primitive data types

Constants are denoted by numbers or identifiers.

```
const A = 123
```

introduces A as a synonym for the constant 123.

Variables are declared in the form

```
var V1,V2, ... Vn : <type>
```

This associates the notations (identifiers) Vi with a data type. Types can be defined by enumeration, e.g.

```
type WEEK = (MON, TUE, WED, THU, FRI, SAT, SUN)
```

or as ranges:

```
type OCTAL = 0..7
```

PASCAL implementations are expected to have the primitive types

```
INTEGER
REAL
BOOLEAN
CHAR
```

predefined.

Structured data types

PASCAL has two data structures, the *array* and the *record*.

Array types are defined in the form

```
array [ <type>] of <type>
```

For example:

```
type MATRIX = array[1..10,1..10]of REAL
```

PASCAL arrays are always of fixed size.

Record types define data structures consisting of a fixed number of components which may be of different types:

```
record
   F1 : <type>;
   F2 : <type>;

   ...
   Fn : <type>
end
```

Example:

```
type ANIMAL = record
                 SORT : (DOG, COW, OTHER);
                 WEIGHT : 1..1000;
                 DOMESTIC : BOOLEAN
              end
```

Records may have *variants*. These are introduced as follows:

```
record
....
   case FV : <type> of
       CONSTANT 1 : (V1 : <type>; V2 : <type>...);
       CONSTANT 2 : ( ... );
       CONSTANT n : ( ... );
....
end;
```

Example:

```
type MAN = record
           FIRSTNAME, LASTNAME : ALFA;
           case STATUS : (SINGLE, MARRIED, DIVORCED) of
           SINGLE : (GIRLFRIEND : PERSON);
           MARRIED : (WIFE : PERSON; WEDDINGDATE : DATE);
           DIVORCED : (FROM WHOM : PERSON; GUILTY : BOOLEAN)
           end
```

The variant-field may be omitted, in which case no storage is reserved for it:

```
record
....
case <type> of
....
end
```

A component F of a record variable V is denoted by

```
V.F
```

For example:

```
var FIDO : ANIMAL;
....
    FIDO.SORT:=DOG;
    FIDO.WEIGHT:=40;
....
```

Statement types

Primitive

Jump statements in PASCAL have the form

```
goto <label>
```

Labels are positive integers.

Assignment statements have the syntax

v := <expression>

Unlike ALGOL 60, the expression may only consist of operators and functions applied to constants, variables, and other expressions. Procedure- and function statements are like the ones in ALGOL 60:

PF (a_1, a_2, a_3, ..., a_n)

Structured

Statements can be concatenated into a *compound statement:*

begin S1; S2; ... Sn end

Statement selection is performed for boolean expressions by

if <boolean expression> then S1 else S2

For expressions of a primitive type, the *case selection* is provided:

```
type T = (C1, C2, ..., Cn);
....
case <expression of type T> of
     C1 : S1;
     C2 : S2;

     ....
     Cn : Sn
end
```

A statement is repeatedly executed while a condition is true by

while <boolean expression> do S

Alternatively, it can be repeated until a condition holds:

repeat S until <boolean expression>

A statement may also be repeated, with a sequence of primitive values being assigned to a loop variable v:

for v := min to max

For records, the structured statement

with v do S

is provided. Inside S, the fields of record variable v can be referred to by their identifiers alone, instead of using the dot-notation "v.f".

Example:

```
with FIDO do
begin
   SORT := DOG;
   WEIGHT := 40
end
```

Procedure and function declarations

Procedures and functions are declared by

```
procedure P(p1; p2; ...; pn);
<local declarations>
begin
   s1; s2; ...; sn
end

function F(p1; p2; ...; pn) : <result type>;
<local declarations>
begin
   s1; s2; ...; sn
end
```

The declarations of formal parameters p_i have the form

$$v_i : T_i$$

for constant parameters of type T_i, and

$$\underline{\text{var}}\ v_j : T_j$$

for variable parameters

Functions may only have results of primitive types or subranges of these.

Brinch Hansen's extensions for concurrent processes

The concurrent statement

To indicate that the statements S1, S2, ..., Sn can be executed concurrently, Brinch Hansen uses the *concurrent statement*

cobegin S1; S2; ...; Sn coend

It was first proposed by Dijkstra.
The semantics of the concurrent statement in the sequence

SO; cobegin S1; S2; ...; Sn coend; Sn+1

is as follows: First, SO is executed, and then S1, S2, ..., Sn are executed concurrently. When all statements in the concurrent statement have terminated, Sn+1 is executed.

To enable time-dependent errors to be caught by a compiler, the designer or the reader of an algorithm, the structured statement's semantics are restricted to mean that its statements S1; S2; ...; Sn define *disjoint processes*. This implies that a variable subject to change in one of these processes may not be referred to by another. To permit checking of this disjointness, the language used must have the property that a statement's constant and variable parameters can be determined by inspecting the statement.

Hoare (Ho 72) developed the following set of rules for a language with such property:

- Arrays must be private to a single process, because their components are selected at run-time.
- Procedure calls may not have side-effects.
- All variables used as variable parameters in a procedure statement must be distinct and cannot occur as constant parameters in the same statement.
- Jumps out of concurrent statements are forbidden.

These rules have been obeyed in the algorithms presented in this thesis.

The critical region

For the declaration of *shared variables*, Brinch Hansen uses the notation

var v : shared <type>

Concurrent processes can only refer to and change common variables within structured statements called *critical regions:*

```
region v do s
```

This enables compilers and humans to verify that shared variables are only accessed inside critical regions.

Critical regions referring to the same variable exclude one another in time.

The conditional critical region

To enable a process to wait until an arbitrary condition B between the components of a common variable holds, Brinch Hansen proposes the await primitive:

```
var v : shared T;
....
region v do
begin .... await B; .... end
```

The await must be textually enclosed by a critical region associated with variable v. For nested regions, the synchronising condition B is associated with the innermost enclosing region.

This notation is a generalisation of the *conditional critical region* proposed by Hoare (Ho 72) which is of the form (in Hoare's notation):

```
with v when B do S
```

Hoare's construct thus only permits the testing of condition B at the beginning of the critical region.

Appendix B
ON THE IMPLEMENTATION OF BRINCH HANSEN'S PROCESS EXTENSIONS TO PASCAL

Introduction

The purpose of this appendix is to describe how the constructs concurrent statement, shared variable, critical region, and conditional critical region could be implemented in terms of more primitive constructs.

In this discussion, the availability of a real-time monitor is assumed, and for each construct, the services required from this monitor are described.

The discussion is based entirely on Brinch Hansen's implementation proposals in (Ha 73).

The concurrent statement

The concurrent statement

```
S0;
cobegin S1; S2; ....; Sn coend;
Sn+1
```

can be implemented as follows: The process that executes statement S0 initiates the concurrent processes S1, S2, ..., Sn by calling a monitor procedure *initiate process* once for each of these. Then, by a call to a monitor procedure *delay process*, it is delayed until all the processes in the concurrent statement are terminated:

```
S0;
for every Si do initiate process (initial state);
delay process;
Sn+1;
```

Initial state defines the initial register values for the new processes.

Initiate process initialises a process description for the new process and enters a pointer to it in the queue of ready

processes. Finally, it increases the number of children of the calling process and continues this process.

Monitor procedure *delay process* functions as follows: If the number of children of the calling process is greater than zero, it is delayed, and another process from the ready queue is given its processor. Otherwise, the calling process continues.

The children processes S1, S2, ..., Sn in a concurrent statement each execute a statement Si and call a monitor procedure *terminate process:*

```
"Process i"   Si;
              terminate process;
```

Terminate process frees the description of the calling process and decrements the number of children of its parent process. If the number of children becomes zero and the parent is delayed, the parent is continued. Otherwise, the processor is allocated to another ready process.

Shared variable and critical region

Shared variables can be implemented by associating a semaphore with the variable:

```
"var R : shared T"      var R : record
                                value : T;
                                mutex : semaphore
                                end
```

Thus, each shared variable has its own private semaphore which should be inaccessible (anonymous) inside a concurrent statement. Critical regions for R can then be implemented in the form

```
R.mutex := 1;                "initialisation"
  cobegin
  ....
     with R do                    "region R do"
     begin P(mutex);              "begin"
     S;                           "S"
     V(mutex)
     end;                         "end";
  ....
  coend
```

P and *V* are Dijkstra's classical semaphore operations. These are the only monitor functions required to implement the region construct.

Conditional critical regions

When a process encounters a critical region for a shared variable V, it joins a queue which is associated with V's semaphore. From this queue, the processes enter the region sequentially one after the other (mutual exclusion).

At the *await B* of a conditional critical region, it inspects the shared variable V to test whether the condition B holds. If it does, it completes the remainder of the critical region. Otherwise, the process leaves the critical region temporarily and joins an *event queue* QE(V) associated with the shared variable V. Other processes can then enter critical regions for V. If they are also made to wait by a conditional critical region, they join the same event queue QE(V).

Whenever a critical region for V has been completed, V may have been changed and consequently the waiting conditions for some of the processes in the event queue QE(V) may be satisfied. Therefore, at the exit of each critical region for V, all processes in the event queue are transferred to the main queue connected to the shared variable's semaphore. This allows those processes to re-enter their critical regions and inspect the shared variable V again.

The conditional critical region can thus be implemented by using two monitor procedures, *await* and *cause*, in the following way:

```
region V do                          "region V do"
begin                                "begin"
   while not B do await(e);          "await B";
   S;                                "S"
   cause(e)
end                                  "end"
```

An event variable (e) consists of two components: A queue of processes waiting for the event, and a pointer to the semaphore

on which these processes must wait to reenter their critical region, after the occurrence of the event.

The *await* procedure enters the calling process in an event queue and performs a V operation on the associated semaphore, so that another process can enter its critical region.

The *cause* procedure transfers all processes from an event queue to an associated semaphore queue. The calling process continues. For each shared variable accessed in a conditional critical region, all critical regions (including the simple ones) must conclude with a call to *cause*.

REFERENCES

Ar 72 Arnborg, S., Storage administration in a virtual memory SIMULA system. *BIT*, Vol. 12 (1972), pp. 125-141.

Ar 74 Arnborg, S., Optimal memory management in a system with garbage collection. *BIT*, Vol. 14 (1974), pp. 375-381.

Ba 71 Barbacci, M., A LISP processor for C.ai. Carnegie Mellon Univ., CMU-CS-71-103.

Ba 71a Bauer, F.L., and Goos, G., *Informatik*. Berlin, Springer Verlag, 1971.

Bo 68 Bobrow, D.G., Storage management in LISP. In: *Symbol Manipulation Languages*. Amsterdam, North Holland Publ. Comp., 1968.

Br 70 Branquart, P., and Lewi, J., General principles of ALGOL 68 garbage collector. Techn.Note, N60. Brussels, MBLE Manufacture Belge de Lampes et de Matériel Electronique, Jan 1970.

Br 71 Branquart, P., A scheme of storage allocation and garbage collection for ALGOL 68. In: Pe 71, p. 199.

Ch 70 Cheney, C.J., A non-recursive list compacting algorithm. *Communications of the ACM*, Vol. 13 (1970), p. 677.

Co 60 Collins, G.E., A method for overlapping and erasure of lists. *Communications of the ACM*, Vol. 3 (1960), pp. 655-657.

Co 66 Collins, G.E., PM, a system for polynomial manipulation. *Communications of the ACM*, Vol. 9 (1966), pp. 578-589.

Co 71 Coffman, E.G., System deadlocks. *Computing Surveys*, Vol.3 (1971), pp. 67-78.

De 71 Denning, P.J., Third generation computer systems. *Computing Surveys*, Vol. 3 (1971), pp. 175-211.

Di 68 Dijkstra, E.W., Co-operating sequential processes. In: *Programming Languages*, by F.Genuys (ed.). New York, Academic Press, 1968.

Di 72 Dijkstra, E.W., Notes on structured programming. In: *Structured Programming*, by O.J.Dahl, E.W. Dijkstra, and C.A.R. Hoare. New York, Academic Press, 1972.

Fe 69 Fenichel, R.R., and Yochelson, J.C., A LISP garbage collector for virtual memory systems. *Communications of the ACM*, Vol.12 (1969), pp. 611-612.

Ge 60 Gelernter, H.,Hansen, J.R., and Gerberich, C.L., A FORTRAN - compiled List Processing Language. *J.Assn.Comp.Mchy*. Vol. 7 (1960), pp. 87-101.

Gr 72 Griswold, R.E., *The Macro Implementation of SNOBOL 4.* San Francisco, W.H.Freeman & Co., 1972.

Gr 74 Griffiths, M., Runtime storage management. In: *Lecture notes "Advanced Course on Compiler Construction"*. TU Munich, March 4 to 15, 1974.

Ha 67 Haddon, B.K., and Waite, W.M., A compaction procedure for variable length storage elements. *Computer Journal*, Vol.10 (1967-68), pp. 162-165.

Ha 73 Hansen, P.Brinch, *Operating System Principles*. Englewood Cliffs, N.J., Prentice-Hall, 1973.

Ha 73a Hansen, P.Brinch, Concurrent programming concepts.*Computing Surveys*, Vol. 5 (1973.12), no. 4.

Ha 75 Hansen, P.Brinch, The purpose of Concurrent PASCAL. *SIGPLAN Notices*, Vol. 10 (1975), no. 6, pp. 305-309.

Hi 74 Hill, U., Special run-time organisation techniques for ALGOL 68. In: *Lecture Notes "Advanced Course on Compiler Construction"*, Chapter 3 C, pp. 26-28. TU Munich, March 4 to 15, 1974.

Ho 72 Hoare, C.A.R., Towards a theory of parallel programming. In: *Operating System Techniques*, New York, Academic Press, 1972.

Ho 74 Hoare, C.A.R., Optimisation of store size for garbage collection. *Information Processing Letters*, Vol. 2 (1974), pp. 165-166. (North Holland Publ.Comp.)

Je 74 Jensen, K., and Wirth, N., *PASCAL User Manual and Report.* Berlin, Springer Verlag, 1974 (Lecture Notes in Computer Science, Vol. 18).

Kn 68 Knuth, D., *Fundamental Algorithms*; 2nd pr. Reading, Mass., Addison-Wesley Publ.Comp., 1969 (The Art of Computer Programming, Vol. 1)

Ma 71 Marshall, S., An ALGOL 68 garbage collector. In: Pe 71, p.239.

Ma 72 Maurer, W.S., *The Programmer's Introduction to LISP.* London, McDonald/American Elsevier, 1972.

Me 70 van der Mey, G., and van der Poel, W.L., A manual of HISP for the PDP 9. TH Delft, unpublished note, ca. 1970.

Me 71 van der Mey, G., General list processing. TH Delft, unpublished note, ca. 1971.

My 70 Myhrhaug, B., Storage assignment systems. Norwegian Computing Centre, Publ. S-15.

Ne 60 Newell, A., and Tonge, F.M., An introduction to IPL-V. *Communications of the ACM*, Vol. 3 (1960), pp. 205-211.

Ne 65 Newell, A. et.al., *Information Processing Language-Manual;* 2nd ed., Englewood Cliffs, N.J., Prentice-Hall, 1965.

Pe 71 Peck, J.E.L. (ed.), *ALGOL 68 Implementation*. (IFIP). Amsterdam, North-Holland, Publ.Comp., 1971.

Ro 67 Ross, D.T., The AED free storage package. *Communications of the ACM*, Vol. 10 (1957), pp. 481-492.

Sc 67 Schorr, H., and Waite, W.M., An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, Vol. 10 (1967), pp. 501-506.

Ta 73 Tanenbaum, A.S., *Design and Implementation of an ALGOL 68 Virtual Machine*. I.W. 4/73, June 1973. Amsterdam Math.Centrum.

Th 72 Thorelli, L.E., Marking algorithms. *BIT*, Vol. 12 (1972) pp. 555-568.

We 63 Weizenbaum, J., Symmetric list processor. *Communications of the ACM*, Vol. 6 (1963), pp. 524-544.

We 68 Wegner, P., *Programming Languages, Information Structures, and Machine Organisation*. New York, Mc Graw-Hill, 1968.

We 69 Weizenbaum, J., Recovery of re-entrant list structures in SLIP. *Communications of the ACM*, Vol. 12 (1969),pp. 370-372.

We 71 Wegbreit, B., A generalised compactifying garbage collector. *Computer Journal*, Vol. 15 (1972) pp. 204-208.

Wi 69 van Wijngaarden, A. (ed.), Report on the Algorithm Language ALGOL 68. *Numerische Mathematik* 14 (1969), pp. 79-218.

Wi 72 Wirth, N., *Systematisches Programmieren*. Stuttgart, Teubner Verlag, 1972. (Teubner Studienbücher)

Wo 69 Wodon, P.L., Data structure and storage allocation. *BIT*, Vol. 9 (1969).

Wo 71 Wodon, P.L., Methods of garbage collection. In: Pe 71, p. 245.

SAMENVATTING

Dit proefschrift behandelt het vraagstuk met betrekking tot het onwerpen en implementeren van concurrent garbage collector systemen voor dynamische geheugenallocatie. Tot dusver kon geen van de belangrijke programmeertalen, zoals ALGOL 68, LISP en SIMULA, die garbage collection gebruiken voor het automatische beheer van het heap geheugen gedurende run tijd, gebruikt worden voor het programmeren van tijd-afhankelijke real-time programma's. De reden hiervoor is, dat tot op heden de garbage collector in alle implementaties van deze talen een sekwentiële systeem procedure is, die wordt aangeroepen als er geen vrije heap ruimte meer beschikbaar is of als deze moet worden samengepakt. Dit aanroepen gebeurt op niet voorspelbare tijdstippen. Gedurende het uitvoeren van de garbage collector moet de uitvoering van het applikatie programma worden opgeschort. Dit opschorten kan oplopen tot enkele seconden gedurende welke het applikatie programma niet kan reageren op externe gebeurtenissen.

De voornaamste stelling van dit proefschrift is, dat er taal processors met garbage collectors, die incrementeel en concurrent werken met applikatie programma's, correct kunnen worden ontworpen en geimplementeerd. Zulke garbage collectors verstoren de uitvoering van applikatie programma's slechts gedurende korte en beperkte perioden. Daarom kan aan deze programma's een voorspelbaar real-time gedrag worden toegekend.

Het proefschrift beschrijft applikatie programma en concurrent garbage collector als twee parallelle, samenwerkende processen. Voor dit systeem worden twee hoofd probleem gebieden vastgesteld, nml. coördinatie en scheduling van de processen.

De twee processen delen gegevens en hun toegang tot deze gegevens moet in de tijd geordend worden om de juiste werking van het gehele systeem te verzekeren. Het proefschrift ontwikkelt

algemene oplossingen voor dit probleem, gebaseerd op invarianten voor de gezamelijke variabelen. Een noodzakelijke verzameling van 11 invarianten, die bijgehouden moeten worden door de kritieke gebieden van de twee processen met concurrent garbage collection systemen wordt hiertoe vastgesteld.

Het scheduling probleem treedt op doordat applikatie programma en garbage collector wat betreft de vrije ruimte een producent/consument relatie tot elkaar hebben. Een model van deze relatie wordt ontwikkeld en het niveau van de vrije ruimte, waarbij het garbage collection proces wordt ingeschakeld, wordt hieruit afgeleid. Gebaseerd hierop wordt het maximaal mogelijke nuttige heap gebruik en de overhead van geheugenruimte en processor tijd vastgesteld.

Om de algemene toepasbaarheid van de benadering te laten zien worden twee concurrent garbage collection systemen in de vorm van programma's aangeboden. Eén systeem onderhoudt locaties met een vaste lengte, zoals in LISP, en gebruikt een merkende algorithme, die recursief in CAR en iteratief in CDR richting werkt. Het andere systeem onderhoudt locaties met een variabele lengte en heeft daarom een samenpakkende garbage collector. Het gebruikt een niet-recursieve kopiëer algorithme, die de toegankelijke locaties naar een nieuw gebied in het geheugen kopiëert.

De twee systemen zijn geprogrammeerd in PASCAL, dat is uitgebreid met vier constructies, nml. shared variables, concurrent statements, critical regions, en conditional critical regions. De correctheid van beide systemen wordt bewezen.

CURRICULUM VITAE

Born 12 October 1937 in Berlin, Germany.
Attended the Ratsgymnasium in Goslar, Germany, from 1948 to 1957.
Studied electrical engineering at the Technological University in Braunschweig, Germany, from 1957 to 1964.
Was granted the degree of Diplom-Ingenieur in 1964.
Worked in an industrial laboratory on the design and implementation of fail-safe logic circuits.
Joined a consultancy firm and was project leader for major operations research studies.
Joined SHAPE Technical Centre, The Hague, Netherlands, as member of the Operations Research Division. Engaged in military operations research and specialised in system description, -design, and -simulation.
Has been a member of SHAPE Technical Centre's Mathematics and Computer Division since 1971. Had project responsibility for the design and implementation of a very large interactive simulation system and several real-time systems.
Special research interests lie in the areas of programming methodology, data structures, system implementation languages, and computer graphics.
At the time of publication of this thesis, holds a Branch Head position in the Mathematics and Computer Division.

STELLINGEN

1

Although not adding anything to the semantics of ALGOL 68 or SIMULA, concurrent garbage collection permits the use of these languages for the programming of real-time software. Thus, this implementation approach considerably widens the scope of application of these languages.

2

Garbage collection, as a readily identifiable overhead of programs in languages such as ALGOL 68 and SIMULA, has put these languages at a psychological disadvantage relative to PL/1. Concurrent garbage collection, particularly if it is supported by hardware, results in incremental, more continuous heap management and could therefore significantly reduce the psychological disadvantage.

3

Well-structured high-level language constructs will permit programming to make the necessary transition from an art to an engineering discipline. They will have an impact on software design equivalent to that which standardised reliable components such as ball-bearings or screws made on the design of machinery.

4

The mushrooming extensions to FORTRAN in the direction of "Structured FORTRAN" are mere cosmetics. They are harmful to progress in the programming field, as they will extend the lifetime of FORTRAN, and will proliferate into a large number of non-standard FORTRAN dialects.

5

Unlike ALGOL 60, which led to the development of stack-oriented central processors, ALGOL 68 has so far not influenced the architecture of commercially available computer hardware.

6

In variation of Dijkstra's statement that *"Program testing can be used to show the presence of bugs, but never to show their absence"*, it might be said that correctness proofs of a program can be used to demonstrate that the program correctly implements a design, but never to show the design's correctness.

7

The teaching of BASIC and similar languages in secondary schools is dangerous in that it hides the true potential of computers from the pupils. Also, the ubiquitous teletypewriter is probably the least satisfactory pupil-machine interface in use, because of its slow speed and basically one-dimensional output. The development of software, hardware, and a curriculum for teaching informatics in secondary and possibly also primary schools is an urgent task which is worthy of the attention of the best computer scientists, psychologists, and pedagogues.

8

In his article "The architecture of complexity", H.A. Simons has written that the search for state and process descriptions of the same phenomenon is characteristic of human problem solving, and that both modes are equally important. The German schooling system and probably most other European systems have traditionally overemphasized the state-description mode. Simulation, role playing, and teaching games are a means which could serve to support the teaching of the process-description mode.

Cf. Simon, H.A., "The architecture of complexity", Proc. American Philosophical Society 106,6, pp.468-482, 1962.

9

One of man's greatest gifts is his ability to construct mental models; one of his greatest handicaps is his tendency to mistake these models for reality.

10

Television has traditionally been a centralized medium which projected the centres' urban value systems. Cable television could provide the technical means to re-establish a balance by enabling the peripheral regions and small towns to give expression to their own value systems.

11

The large number of civil service staff supporting the governments of European democracies has put the legislative bodies and the electorate in these countries at a disadvantage in the political process.

12

The secret, unauthorised manipulation of the future ought to be made a punishable offence.

Klaus G. Müller

Delft, 3 Maart 1976