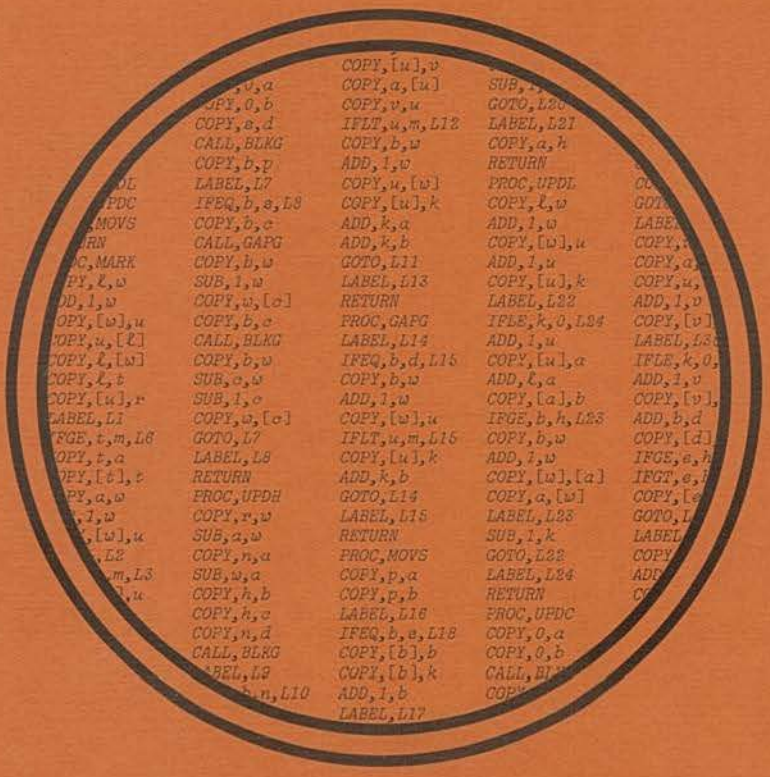


ABSTRACTION, SPECIFICATION AND IMPLEMENTATION TECHNIQUES

WITH AN APPLICATION TO GARBAGE COLLECTION

H.B.M. JONKERS

STELLINGEN BEHORENDE BIJ HET PROEFSCHRIFT
ABSTRACTION, SPECIFICATION
AND IMPLEMENTATION TECHNIQUES
WITH AN APPLICATION TO GARBAGE COLLECTION
VAN H.B.M. JONKERS

1. De invoering van de modules and separate compilation facility in ALGOL 68 maakt de reserved word style van de standard hardware representation ambigu.

   BOOM, H.J., & HANSEN, W.J., The report on the standard hardware representation for ALGOL 68, SIGPLAN Notices 12, 5 (1977), 80-87.

   LINDSEY, C.H., & BOOM, H.J., A modules and separate compilation facility for ALGOL 68, ALGOL Bulletin 43 (1978), 19-53.

2. Door gebruik te maken van een andere codering dan de gebruikelijke Manchester phase encoding is het mogelijk zonder enig verlies van betrouwbaarheid de schrijfdichtheid van digitale cassettes met gemiddeld een derde te verhogen.

   Manchester phase encoding as applied to digital cassette tape transports, Braemar application note, Spec. 0070.

3. Iedere algoritme kan op een natuurlijke wijze gezien worden als een formeel systeem, waarbij de begrippen "toestand" en "berekening" corresponderen met respectievelijk "formule" en "afleiding".

4. De formele beschrijving van een algoritmisch concept is slechts dan bevredigend als de eenvoud van de beschrijving in overeenstemming is met de intuïtieve eenvoud van het concept.

5. In de meeste informaticacurricula wordt een college "abstractiemethoden" node gemist.

   Informatica als studierichting, Nieuwsbrief van de Werkgemeenschap Theoretische Informatica 4 (1981), 28-60.

6. Het gebruik van "pointers" in de semantische beschrijving van programmeertalen waarin de pointer niet als datastructuur voorkomt, is onnodig.

   Hoofdstuk 2 van dit proefschrift.

7. Het creëren en vervolgens elimineren van overtolligheid is een algemene methode, waarmee systemen onder behoud van hun functionele eigenschappen getransformeerd kunnen worden en waardoor beschouwingen over de correctheid van de transformatie aanzienlijk vereenvoudigd worden.

   Hoofdstuk 3 van dit proefschrift.

8. De transformationele methode is bij uitstek geschikt voor het op een samenhangende wijze presenteren van grote klassen algoritmen.

   Hoofdstuk 5 van dit proefschrift.

9. Het is mogelijk een compacting garbage collection op een zowel qua tijd als ruimte efficiënte manier uit te voeren in slechts twee fasen.

   Algorithm G&C op bladzijde 203 van dit proefschrift.

10. De mogelijkheden die de microprocessortechniek biedt voor de verbetering van de situatie van zwaar lichamelijk gehandicapten worden onvoldoende benut.

ABSTRACTION, SPECIFICATION AND IMPLEMENTATION TECHNIQUES

WITH AN APPLICATION TO GARBAGE COLLECTION

ABSTRACTION, SPECIFICATION AND IMPLEMENTATION TECHNIQUES
WITH AN APPLICATION TO GARBAGE COLLECTION

PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR IN DE TECHNISCHE WETENSCHAPPEN AAN DE TECHNISCHE HOGESCHOOL EINDHOVEN, OP GEZAG VAN DE RECTOR MAGNIFICUS, PROF.IR. J. ERKELENS, VOOR EEN COMMISSIE AANGEWEZEN DOOR HET COLLEGE VAN DEKANEN IN HET OPENBAAR TE VERDEDIGEN OP VRIJDAG 12 FEBRUARI 1982 TE 16.00 UUR

DOOR

HENRICUS BERNARDUS MARIA JONKERS

GEBOREN TE AMSTERDAM

1982
MATHEMATISCH CENTRUM, AMSTERDAM

Dit proefschrift is goedgekeurd
door de promotoren

Prof.dr. F.E.J. Kruseman Aretz
en
Prof.dr. M. Rem

Aan mijn ouders

The research reported in this thesis has been conducted while the author was employed at the Mathematical Centre in Amsterdam.

CONTENTS

CHAPTER 0

INTRODUCTION

0.1. WHAT THIS MONOGRAPH IS ABOUT

This monograph is a study on what I believe to be three key-notions of computer science: abstraction, specification and implementation. Before being more specific let me explain briefly what I mean by "computer science". This may avoid unnecessary misunderstandings, since there seems to be a considerable and rather fundamental disagreement on the purport of the latter term [TRAUB 81], [DIJKSTRA et al. 81]. Computer science, as I will look upon it in this monograph, is concerned with the development and study of concepts and techniques to facilitate the use of the "computer", while abstracting from its physical realization and from the specific applications which the computer is used for. The term "computer" is used here, not in the sense of a particular device or class of devices, but rather in the (abstract) sense of any device or mechanism which is able to manipulate data in an effective way.

The above very general "definition" of computer science does not claim to define what computer science *is*. It serves only to indicate the point of view from which I wish to consider computer science in this monograph. I am well aware of the fact that by choosing a different angle of vision one can reach, with good reason, a different definition. The above definition highlights the *use* of the computer as the ultimate justification of the existence of computer science. Consequently, in this monograph we are not concerned with what is commonly referred to as "theoretical computer science". This does not imply that we shall not touch on issues of theoretical interest. The theory, however, is not a goal in itself and will be elaborated upon only as far as it is considered useful from a practical point of view. On the other hand, the definition also implies that we abstract from the specific applications for which the computer is used. This monograph, therefore, is not concerned with what is usually called "applied computer science" either.

Having sketched the context in which this monograph should be placed, we are now ready to discuss its subject-matter. As stated above, this monograph is a study on abstraction, specification and implementation. In particular, we shall be concerned with the latter three notions in connection with "algorithms" (as abstractions of the "actions" of a computer) and "data structures" (as abstractions of the "data" operated upon by a computer). The view of computer science expressed above implies that we shall focus on the practical aspects of these concepts, i.e., our primary interest is in abstraction, specification and implementation *techniques*.

Techniques for making abstractions, specifications and implementations of algorithms and data structures are the basic tools of the (practical) computer scientist. The research effort which has gone into the development of these tools has been enormous. Numerous researchers have worked on various aspects of the subject and this monograph relies heavily on their work. Instead of proceeding directly along one of the (many) lines set out by others, however, we shall attempt to make a fresh start by giving our

own "reconstruction of the facts". This reconstruction will be guided and inspired by the work and ideas of many others. In particular I wish to refer to the fundamental influence of [DAHL et al. 72]. (For more references see the respective chapters.) Yet, the reconstruction is not a mere paraphrasing of the ideas of others, since, as is the purpose of a reconstruction, it will reveal some new "facts" as well.

The above implies that this study should be regarded as a "view" of abstraction, specification and implementation, and by no means as an attempt to solve all problems. (For example, we shall not be concerned with "concurrency".) In general, we shall approach the problems from a somewhat fundamental point of view, using two guiding principles. The first is the principle of *simplicity*. Due to the progress in VLSI-technology the field of application of the computer is constantly expanding and the computer is used for ever more complex tasks. There seems to be a tendency to try and cope with the latter situation through the introduction of ever more complex tools. As argued in a most illuminating way in [HOARE 81], the only real solution, however, is the pursuit of the utmost simplicity. In this monograph we shall therefore be looking for simple concepts and tools.

Our second guiding principle is the principle of *mathematical rigour*. Unlike physical phenomena or human beings, computers are essentially discrete mathematical objects, i.e., their (abstract) behaviour can be described fully by a mathematical theory. This implies that reasoning about their behaviour is essentially mathematical reasoning. The theories which describe the behaviour of real computers are true monstrosities from a mathematical point of view. In order to get the behaviour of such machines (and other not yet existing machines) into our mental grip we use abstraction and make "models" of the various aspects of this behaviour. If these models are to be of any practical value in "controlling" the mathematical behaviour of real (or yet to become real) machines, these models should be mathematical themselves. The word "model", in the sense of a mathematically rigorous abstraction of "reality", will be recurring frequently in this monograph. (Notice that we use the word "model" in a different sense from mathematical logic [MENDELSON 64].)

As stated above, our primary interest in this monograph is in (mental) tools. There is not much sense in developing tools unless we also use them and show that they work. This monograph can therefore be divided into two parts. In the first (Chapters 1-3) we will be concerned with the tools by themselves. Successively we will consider the tools of abstraction (Chapter 1), specification (Chapter 2) and implementation (Chapter 3). In the second part (Chapters 4-7) we will present the result of applying the tools. The field to which we will apply them is "storage management" in general, and "garbage collection" in particular. Garbage collection is a most appropriate subject for the application of the tools, since it is generally considered to be of a rather tricky and elusive nature. (As such it has for example been chosen as one of the "background problems" of the ABSTRACTO project [ABSTRACTO 79].)

The applications part of this monograph can itself be divided into two parts. In the first part we will present the result of applying the tools in the *classification* of algorithms. That is, based on a model for storage management (Chapter 4), we will present a survey of garbage collection algorithms (Chapter 5). In the second part we will present the result of applying the tools in the *design* of algorithms. In particular we apply them in the design of a storage management system (Chapter 6) and in the design of a garbage collector (Chapter 7) to be used in the latter storage management system. The structure of this monograph as described above is visualized in Figure 0.1.

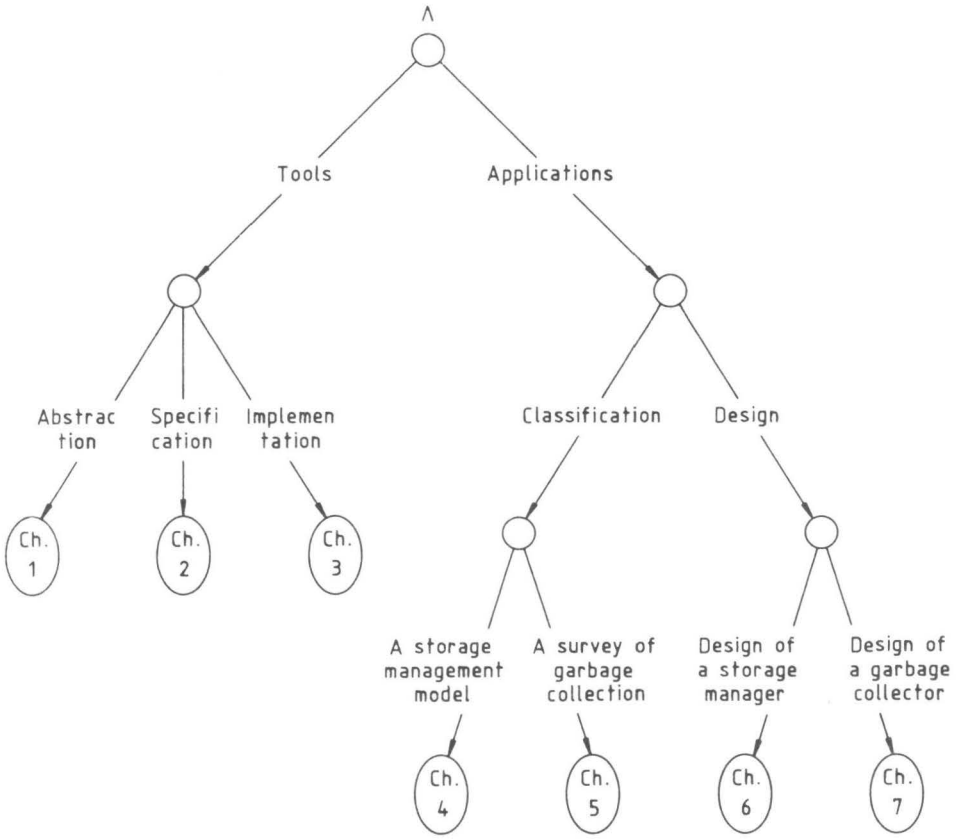


Figure 0.1

It is worth noticing that there is little or no connection between the order of the chapters and the order in which the subject-matter of this monograph has come about. (See the parts of the subject-matter which have been published previously [JONKERS 79, 80, 80a, 80b, 81].) It would even be wrong to say that the "tools" were there before the "applications". As a matter of fact, the question which came first, the "tools" or the "applications", is somewhat similar to the problem of the egg and the chicken. It was from the interaction of the two that the subject-matter of this study has emerged. Though this process of interaction need not necessarily converge, I hope that it has reached a state sufficiently stable to justify recording it in this monograph. The fact that actually triggered the entire process (or, if you like, the excuse for the whole enterprise) was the problem of designing a garbage collector for a machine independent ALGOL 68 implementation [MEERTENS 81]. Though we shall make quite a detour, this problem will eventually be solved (in an implementation independent way) in the last chapter of this monograph.

0.2. SUMMARY OF THE CHAPTERS

In this section we shall briefly describe the purpose and contents of each chapter.

0.2.1. Chapter 1

The purpose of this chapter is to discuss how abstraction can be used as a systematic tool in solving problems ("design") and in the ordering of knowledge ("classification"). We introduce a simple model for abstraction which allows us to reason about abstraction in a more concrete way than usual. For example, on the basis of this model we derive a simple lemma stating that, in a certain sense, abstraction can be viewed as "omitting details". Then we go into the use of abstraction as a tool in solving problems. We discuss how problems can be solved "in the abstract", what "good" abstractions are and how abstraction can be used in combination with a top-down problem solving technique. In the next section we indicate how abstraction can be used in classifying problems and their solutions. We discuss how, through a process of "systematic generalization", problems and their solutions can be ordered systematically at various levels of abstraction. Our particular interest in this monograph is in "algorithmic problems", and we discuss how their solutions ("algorithms") can be ordered further through "correctness-preserving transformations". (So as to avoid any confusion, we note that in this chapter we are *not* concerned with the role of abstraction in "artificial intelligence" (as in [PLAISTED 81]), but rather with abstraction as an explicit tool of the human problem solver and "taxonomist".)

0.2.2. Chapter 2

In this chapter our aim is to describe, or at least lay the foundations for, a mathematically rigorous specification method which is suitable for the specification of all (sequential) algorithms and data structures normally encountered in programming practice, including those algorithms and data structures which involve dynamic and shared data. A loose version of this language is used in the other chapters for the description of algorithms and data structures. We start off by introducing a concept which is believed to be novel: the "structure". It is essentially a simple mathematical model of the access properties of a storage structure. We show how using this model, storage structures with arbitrary sharing and circularities can be characterized without the need to introduce pointers. Using a special partial order, three primitive operations on structures are defined (among them "creation"). Subsequently, we introduce a simple but powerful typeless language for the description of arbitrary (deterministic) operations on structures. The semantics of this language is defined rigorously in terms of the primitive operations on structures. Then, led by an example, we discuss in an informal way how this language can be extended into a full-fledged specification language through the addition of a type-mechanism and the introduction of abstraction facilities and nondeterminism. The chapter is concluded with a comparison of "structures" with other characterizations of storage structures (such as "Vienna objects" and "graphs").

0.2.3. Chapter 3

The purpose of this chapter is to describe a simple implementation method for algorithms and data structures. The first part of this chapter consists of an informal presentation of the method. The method is based on a simple four-step technique to accomplish a change of data representation in a correctness-preserving way. The quintessence of this technique, which does not require the enhancement of existing verification techniques, is the temporary juxtaposition of the old and the new representation. In the second part of this chapter we demonstrate the power and flexibility of the method (particularly as a verification tool) by giving a "proof by construction" of a well-known test case for verification techniques: the Deutsch-Schorr-Waite marking algorithm [SCHORR & WAITE 67]. We start by giving a definition of the problem, from which a simple though highly nondeterministic algorithm is derived almost immediately. Choosing this obviously (partially) correct algorithm as a starting point, the Deutsch-Schorr-Waite marking algorithm is derived in five successive transformation "phases", where each phases follows exactly the four-step scheme described in the first part of the chapter.

0.2.4. Chapter 4

The purpose of this chapter is to introduce a model for storage management, which will be used as the basis for the next chapter. As we argue in the introduction of this chapter, this model, called a "storage management system", can be viewed as an instance of the more general concept of a "dynamic system", which is in fact a data structure involving highly shared data. The first part of the chapter is devoted to a very informal discussion of the subject of storage management. Its purpose is to create some familiarity with the basic concepts and to serve as a rationale for the storage management model. The intuitive concepts discussed in the first part of the chapter are laid down in unambiguous definitions in the second part. The concept of a storage management system, considered as a "dynamic system", is defined by introducing its "components" and postulating a number of "system invariants" which relate the components to each other. Associated concepts are defined in terms of the components of the system, and operations such as "garbage collection" and "compaction" are defined abstractly as internal operations of a storage management system.

0.2.5. Chapter 5

In this chapter we are concerned with giving a survey of (compacting) garbage collection algorithms. The basis of this chapter is the storage management model introduced in Chapter 4. The subject is split into three parts: (pure) garbage collection (without compaction), compaction and compacting garbage collection (the combination of garbage collection and compaction). The discussion of garbage collection and compaction algorithms is essentially analogous and follows the lines set out in Chapter 1: Starting point is an abstract algorithm which is a solution to the basic problem (defined in Chapter 4). All other algorithms are derived from this algorithm by "adding details", both to the problem and to the algorithm. Here "adding details" to an algorithm implies applying a correctness-preserving transformation to the algorithm (establishing some property or "detail"). This approach leads to a division of garbage collection algorithms into four classes and a division of compaction algorithms into

two classes. The combination of garbage collection and compaction algorithms into compacting garbage collection algorithms is discussed by means of a number of examples. Though the survey is not claimed to be complete, it contains the major (sequential) algorithms known from literature as well as a few algorithms which are believed to be novel.

0.2.6. Chapter 6

The subject of this chapter is the design of a storage management system for a machine independent ALGOL 68 implementation. The machine on which this storage management system is supposed to run is the "MIAM" ("Machine Independent Abstract Machine") [MEERTENS 81]. The treatment is entirely independent of both ALGOL 68 and the MIAM, however. This is accomplished by the introduction of a model of the MIAM, i.e., an abstraction of the MIAM capturing the information relevant to storage management, and solving the problem "in the abstract". After introducing the model of the MIAM, the storage management problem is defined by augmenting the model with a number of concepts (such as an "allocation function"). The design of a storage management system is then described. After going through four systems of increasing efficiency, we use the technique described in Chapter 3 to further reduce the overhead of the fourth system, resulting in an efficient storage management system. During the design, the garbage collection and compaction routine used are deliberately kept abstract.

0.2.7. Chapter 7

This chapter is concerned with the design of efficient algorithms for the garbage collection and compaction routine which were kept abstract in the previous chapter. In the same way as Chapter 6 is independent of the MIAM and ALGOL 68, this chapter is independent of Chapter 6. That is, a model is used to capture the information relevant to the design of the garbage collector and compacter. This model is described at the beginning of the chapter and the garbage collector and compacter are defined abstractly as operations operating on this model. The transformational design of efficient algorithms for these operations is then described, starting with two algorithms which are derived directly from the definition of the operations. In contrast to Chapter 6, the algorithms are transformed up to the machine code level (in ten "phases" each). Due to its technical nature the last part of this transformation process is not considered as "design", but rather as "implementation" and it is therefore treated separately from the actual design. The machine code used is the code of a very simple hypothetical machine (which can be viewed as an abstraction of the MIAM).

0.3. USEFUL REMARKS FOR THE READER

The material in this monograph is self-contained to a certain extent. The only preliminaries are an elementary knowledge of mathematics and computer science. For the former see, for example, Chapter 0 in [AHO & ULLMAN 72]. The latter consists mainly of a certain familiarity with regular languages (for Chapter 2 only) [HOPCROFT & ULLMAN 79], programming languages and their implementation [AHO & ULLMAN 77], and the use of assertions in the design and verification of programs [DIJKSTRA 76]. No knowledge of a particular programming language is required.

An attempt has been made to keep not only this monograph as a whole, but also each separate chapter self-contained. The only exception is Chapter 5 (which relies on Section 4.2 of Chapter 4). This implies that (with one exception) each chapter can be read without having read any of the other chapters. So the reader can pick any combination of chapters he likes from the "menu" of Figure 0.1. A minor disadvantage of this approach is that sometimes (though not too often) we have to duplicate efforts. Furthermore, the above does not imply that there is no connection between the chapters. If it is felt appropriate, this connection is indicated by remarks and cross-references, which can be skipped by the reader who did not read the chapter in question.

In the presentation of the material a two-level approach will be pursued more or less consistently. On the one hand, the material will be presented in an informal style, providing ample explanation for new concepts and a motivation for design decisions. On the other hand (as required by the principle of mathematical rigour), these intuitive descriptions will be laid down, as much as possible, in formal (i.e., unambiguous) definitions. The formal text is separated from the informal text by indenting the former and providing it with a vertical bar in the margin.

CHAPTER 1

ABSTRACTION

1.0. INTRODUCTION

The ultimate cause of many of the current difficulties in computer science lies in the very limited powers of comprehension of the human mind. The complexity of the problems encountered in computer science exceeds these powers by several orders of magnitude. Apart from having severe limitations, the human mind is fortunately also equipped with a tool to suit problems to these limitations. This tool is called "abstraction". It allows one to reduce the complexity of a problem by "omitting" irrelevant details. Dependent on the amount of details omitted, "levels" of abstraction can be distinguished. Abstraction is used unconsciously by everyone, because thinking would be impossible without it in the first place. Faced with the complexity of the problems in computer science, it is no longer sufficient to use abstraction unconsciously. The only way out of the current problems lies in a conscious and systematic use of abstraction as a tool in reducing complexity. The purpose of this chapter is to indicate how abstraction can be used as such.

A science in which abstraction traditionally plays a central role is mathematics. First of all, the objects of study in mathematics *are* abstractions and secondly, abstraction is used as an almost self-evident tool by every mathematician to apply results from one part of mathematics to another part. For example, since a "field" can be viewed as an abstraction of the system of real numbers, results from the theory of fields can be applied to the theory of real numbers. Yet the role of abstraction in computer science is even more important, if possible, than in mathematics. The reason is that the amount of detail in practical problems, which are at the root of computer science, is usually far greater than in "normal" mathematical problems. On the other hand, these practical problems are usually not very interesting from a mathematical point of view. Sophisticated mathematical reasoning is generally not required for their solution. It is the sheer amount of detail which makes these problems difficult.

So, no matter how great the mathematical genius of a computer scientist, abstraction is absolutely vital in solving problems in computer science. This becomes even more evident if one realizes that many problems in computer science are created by the practitioners of the science themselves (as a consequence of the rule that each solution of a problem generates a series of new problems). As an example, consider the problem of designing a programming language, the solution of which creates the problem of its implementation. Abstraction cannot only help in constructing a solution for the first problem, but also in keeping the complexity of the second problem under control.

Discussions about abstraction are in constant danger of becoming too abstract. In order to avoid this danger as much as possible, a model for reasoning about abstraction will be introduced in Section 1.1. This model will be used in Section 1.2 to discuss how abstraction can be used as a

systematic tool in solving problems. How abstraction can be used to classify problems and their solutions will be discussed in Section 1.3. The conclusion of this chapter is contained in Section 1.4.

1.1. A MODEL FOR ABSTRACTION

In this section we shall give a definition of the concept of abstraction. For that purpose we restrict the domain of our abstractions to the objects from mathematics. The reason for this is, that if we are to say something concrete about abstraction at all, this should be done in the context of some formal model. Furthermore, the restriction imposed is not felt as a true restriction: The objects studied in computer science are mathematical to a great extent.

1.1.1. Theories

The objects which are studied in mathematics are usually called "theories". A theory can be viewed as a decidable set of "formulas" together with a decidable set of "rules of inference". Since decidability is not our primary concern, we shall omit the decidability requirement:

> A formula is a primitive concept.
>
> A rule R is a pair (P,C), where P is a finite set of formulas and C is a formula.
>
> An element of P is called a premiss of R.
>
> C is called the conclusion of R.
>
> A theory T is a pair $\langle F,R\rangle$, where F is a set of formulas and R is a set of rules such that all premisses and conclusions of rules in R are formulas in F.
>
> An element of F is called a formula of T.
>
> An element of R is called a rule of T.

The formulas of a theory stand for assertions, which are valid iff they can be "derived" using the rules of the theory. This is more formally defined below. Notice that a rule (P,C) with $P = \emptyset$ corresponds to an "axiom". Notice also that in practice the rules of a theory are usually given by inference *schemes*, which correspond to infinitely many rules.

> Let $T = \langle F,R\rangle$ be a theory.
>
> If $P \subset F$, P finite and $C \in F$, then a derivation of C from P in T is a finite sequence $(P_1,C_1),\ldots,(P_n,C_n)$ of rules of T such that $P_i \subset P \cup \{C_1,\ldots,C_{i-1}\}$ $(i = 1,\ldots,n)$ and $C_n = C$.
>
> If $P \subset F$, P finite and $C \in F$, then C is said to be derivable from P in T if there exists a derivation of C from P in T. The fact that C is derivable from P in T will be denoted by $P \vdash_T C$.

A $C \in F$ is said to hold in T if $\emptyset \vdash_T C$.

If $C \in F$ holds in T and D is a derivation of C from $\emptyset$ in T, then D is called a proof of C in T.

1.1.2. Problems

Well now, what in this context is a "problem"? Mathematicians basically have only one problem, and that is finding proofs for their assertions. The obvious definition of a problem is therefore a pair $\langle T,A \rangle$, where T is a theory and A is a formula of T, the "statement" of the problem. In intuitive terms, the problem is to find a proof for A in T. Since A need not hold in T, this problem may be "unsolvable". If it is solvable, it will generally have more than one "solution":

A problem P is a pair $\langle T,A \rangle$, where $T = \langle F,R \rangle$ is a theory and $A \in F$.

T is called the theory of P.

A is called the statement of P.

A solution of P is a proof of A in T.

A problem is solvable if it has at least one solution; otherwise it is unsolvable.

EXAMPLE 1.1

Let T be the theory $\langle F,R \rangle$ with

$$F = \{a,b,c,d,e,f\},$$

$$R = \{(\emptyset,a),(\emptyset,b),(\{a\},c),(\{d\},e),(\{b,c\},f),(\{b,e\},d)\},$$

then the problem $P = \langle T,f \rangle$ is solvable. The following derivation is a solution of P:

$$(\emptyset,a),(\emptyset,b),(\{a\},c),(\{b,c\},f).$$

The problem $\langle T,e \rangle$, however, is unsolvable. □

The above definition of a problem may seem strange at first sight. How, for instance, can the problem of constructing an algorithm be viewed as finding a proof? Suppose the problem is to construct an algorithm S, which given the precondition P establishes the postcondition Q. The construction of S can be viewed as constructing a proof for the following formula A in a suitable theory T:

$$\exists\, S\ [\{P\}S\{Q\}].$$

Here we assume that the theory T is constructive in the sense that the formula A can only be derived by proving that the formula $\{P\}S\{Q\}$ holds in T for a certain given S. The proof that A holds in T then amounts to constructing S. This view of constructing algorithms is even very natural, because it considers the construction of the algorithm and the construction of its proof of correctness as inseparable. A deductive system based on

this view is for example described in [MANNA & WALDINGER 80].

1.1.3. Abstraction and theories

We are now in a position to define precisely what we mean by "abstraction". First, we shall define what we mean by the fact that a theory T_1 is an abstraction of a theory T_2. In intuitive terms it means that there is a correspondence between the formulas in T_1 and T_2 in such a way that if the formula F_1 is derivable from the set G_1 of formulas in T_1, then the formula F_2, corresponding to F_1, is derivable from the set G_2, corresponding to G_1, of formulas in T_2. This implies that everything we can derive in T_1 has an interpretation (or "representation") in T_2. The interpretation need not be unique. There may be many formulas in T_2 which correspond to the same formula in T_1 (but not the reverse). There may also be formulas in T_2 which are meaningless in T_1 (but, again, not the reverse). So the theory T_1 can really be considered to be simpler, or more "abstract", than T_2. The function which indicates the correspondence between the formulas of T_1 and T_2 is called the "abstraction function". As can be concluded from the above, it must be a partial and surjective function from the set of formulas of T_2 into the set of formulas of T_1.

> Let $T_1 = \langle F_1, R_1 \rangle$ and $T_2 = \langle F_2, R_2 \rangle$ be theories.
>
> An abstraction function from T_2 to T_1 is a partial function $\Phi: F_2 \to F_1$ such that:
> (1) $\Phi(dom(\Phi)) = F_1$.
> (2) For each finite $G \subset dom(\Phi)$ and each $F \in dom(\Phi)$
> $\Phi(G) \vdash_{T_1} \Phi(F) \Rightarrow G \vdash_{T_2} F$.
>
> T_1 is said to be an abstraction of T_2 if there exists an abstraction function from T_2 to T_1.

The abstraction relation between theories can easily be seen to be reflexive and transitive. It is not a partial order, because it is not antisymmetric. The abstraction relation does have a least and a greatest element: The theory $\langle \emptyset, \emptyset \rangle$ is an abstraction of each theory, and each theory is an abstraction of the theory $\langle F, R \rangle$, where F is the set of all formulas and R is the set of all rules.

The fact that a theory $T_1 = \langle F_1, R_1 \rangle$ is an abstraction of a theory $T_2 = \langle F_2, R_2 \rangle$ can be proved as follows. First define the abstraction function Φ from T_2 to T_1 and prove that Φ is a surjection. Then prove that T_2 "satisfies" the rules of T_1: For each rule (P_1, C_1) of T_1 and each finite $P_2 \subset dom(\Phi)$ and $C_2 \in dom(\Phi)$ such that $\Phi(P_2) = P_1$ and $\Phi(C_2) = C_1$ prove that $P_2 \vdash_{T_2} C_2$ holds. It is easy to see that this is indeed sufficient.

EXAMPLE 1.2

Let $T_1 = \langle F_1, R_1 \rangle$ and $T_2 = \langle F_2, R_2 \rangle$ be theories, where

$$F_1 = \{a, b, c, d\},$$

$$R_1 = \{(\emptyset, a), (\{b\}, c), (\{c\}, d)\},$$

$$F_2 = \{a,b,c,d,e,f,g\},$$

$$R_2 = \{(\emptyset,a),(\{a\},b),(\{c\},d),(\{c\},e),(\{d,e\},f),(\{b,f\},g)\},$$

then T_1 is an abstraction of T_2. The following partial function Φ: $F_2 \rightarrow F_1$ is an abstraction function from T_2 to T_1:

$\Phi(a) = a,$
$\Phi(b) = a,$
$\Phi(c) = b,$
$\Phi(d) =$ undefined,
$\Phi(e) =$ undefined,
$\Phi(f) = c,$
$\Phi(g) = d.$

The proof that Φ is indeed an abstraction function from T_2 to T_1 amounts to proving that the following assertions are valid:

$$\emptyset \vdash_{T_2} a,$$
$$\emptyset \vdash_{T_2} b,$$
$$\{c\} \vdash_{T_2} f,$$
$$\{f\} \vdash_{T_2} g.$$

□

1.1.4. Abstraction and problems

The concept of abstraction can be extended to problems in a natural way:

> Let $P_1 = \langle T_1, A_1 \rangle$ and $P_2 = \langle T_2, A_2 \rangle$ be problems.
>
> An abstraction function from P_2 to P_1 is an abstraction function from the theory T_2 to the theory T_1 such that $A_2 \in dom(\Phi)$ and $\Phi(A_2) = A_1$.
>
> P_1 is said to be an abstraction of P_2 if there exists an abstraction function from P_2 to P_1.

Notice that if the problem P_1 is an abstraction of the problem P_2, then the fact that P_1 is solvable implies that P_2 is solvable, but not the reverse. If the reverse holds also, P_1 will be called a "proper" abstraction of P_2. Moreover, if P_1 is a proper abstraction of P_2, then each solution of P_1 corresponds to at least one solution of P_2, but, again, not the reverse. Those solutions of P_2 which do not correspond to any solution of P_1 will be said not to be "expressible" in P_1, as defined below.

> An abstraction P_1 of a problem P_2 is said to be proper if the fact that P_2 is solvable implies that P_1 is solvable.

If P_1 is an abstraction of a problem P_2 and $S = (P_1,C_1),\ldots,(P_n,C_n)$ is a solution of P_2, then S is said to be expressible in P_1 if there exists an abstraction function Φ from P_2 to P_1 such that:
(1) $P_i \cup \{C_i\} \subset dom(\Phi)$ $(i = 1,\ldots,n)$.
(2) $\Phi(P_i) \vdash_{T_1} \Phi(C_i)$ $(i = 1,\ldots,n)$.
where T_1 is the theory of P_1.

1.1.5. Abstraction and omitting details

In the introduction of this chapter we informally described abstraction as "omitting (irrelevant) details". One can wonder whether this informal description fits in with the formal definition of abstraction given above. The fact that this is indeed so can be seen as follows. Consider a "detail" of a problem P as a formula or rule of the theory of P. Let P_1 and P_2 be problems such that P_1 is an abstraction of P_2. Then (according to Lemma 1.1 below) there exists a problem P_3 which is "equivalent" to P_2, i.e. P_2 and P_3 are mutual abstractions, such that all formulas and rules of the theory of P_1 are contained in the theory of P_3 and the statement of P_1 is equal to the statement of P_3. The problem P_1 can thus literally be viewed as being obtained by omitting details from P_3 (which is the "same" as P_2).

LEMMA 1.1

Let P_1 and P_2 be problems such that P_1 is an abstraction of P_2. Then there exists a problem P_3 such that:
(1) P_2 and P_3 are mutual abstractions.
(2) $F_1 \subset F_3$.
(3) $R_1 \subset R_3$.
(4) $A_1 = A_3$.
where $P_i = \langle T_i,A_i\rangle$ and $T_i = \langle F_i,R_i\rangle$ $(i = 1,2,3)$.

PROOF

Let P_1 and P_2 be as above, then there exists a partial function Φ: $F_2 \to F_1$ such that:

(1) $\Phi(dom(\Phi)) = F_1$.
(2) For each finite $G \subset dom(\Phi)$ and each $F \in dom(\Phi)$
$\Phi(G) \vdash_{T_1} \Phi(F) \Rightarrow G \vdash_{T_2} F$.
(3) $\Phi(A_2) = A_1$.

Without loss of generality we may assume that:

(4) Φ is 1-1.
(5) $F_1 \cap F_2 = \emptyset$.

(Assumption (4) is allowed because, if $F,G \in dom(\Phi)$, $F \neq G$, $G \neq A_2$ and $\Phi(F) = \Phi(G)$, then G may be omitted from $dom(\Phi)$ without affecting (1), (2) and (3).) Let $F_3 = F_1 \cup (F_2 \setminus dom(\Phi))$ and let Ψ: $F_2 \to F_3$ be defined by:

$$\Psi(F) = \begin{cases} \Phi(F) & \text{if } F \in dom(\Phi), \\ F & \text{if } F \in F_2 \setminus dom(\Phi), \end{cases}$$

then Ψ is a bijection. Let $R_3 = R_1 \cup \{(\Psi(P),\Psi(C)) \mid (P,C) \in R_2\}$ and $A_3 = A_1$. If we define $P_3 = \langle T_3,A_3\rangle$ with $T_3 = \langle F_3,R_3\rangle$ then it is not difficult to prove that P_2 and P_3 are mutual abstractions (use Ψ and Ψ^{-1} as abstraction functions). The fact that $F_1 \subset F_3$, $R_1 \subset R_3$ and $A_1 = A_3$ is obvious. □

If a problem P_1 is an abstraction of a problem P_2, the above entitles us to say that P_1 contains "less detail" than P_2.

1.2. ABSTRACTION IN SOLVING PROBLEMS

In this section we are concerned with the question how abstraction can help us in solving problems. Let us consider first how problems come into existence. Unfortunately problems are not born as pairs $\langle T,A\rangle$, where T is a theory and A is a formula of T. Problems spring up in people's minds as the result of immensely complex processes of thought. It is probably even true that most human problems (and especially "emotional" problems) do not have a theory at all. If we restrict ourselves to the more technical problems of computer science, it can be defended that each problem has a theory.

1.2.1. Specification of the problem

The first thing to do when solving a problem is to model the intuitive problem as a formal problem $\langle T,A\rangle$, where T is the theory and A is the statement of the problem. The purpose of this "specification" of the problem is to make clear in an unambiguous way *what* the problem is. This step is essential to the successful solution of a problem, yet it is often omitted. It becomes even more essential if a group of people decide that they "have a problem". The specification of the problem can then serve to make sure that they are talking about the same things.

The construction of the specification of the problem already requires a fair amount of abstraction. However, this kind of abstraction is a transformation from the intuitive world into the formal world, which can only be discussed in informal terms. The specification should be both "sound" and "complete": All formulas which are derivable in the theory T of the problem should be true intuitively, and each formula which holds intuitively should be derivable in T. Moreover, the specification should be "appropriate": The statement A of the problem should correspond to the intuitive conception of the problem. From now on we shall assume that we have a sound, complete and appropriate specification of some intuitive problem.

1.2.2. Abstraction from irrelevant details

Given the problem $P_1 = \langle T_1,A_1\rangle$, how can abstraction help us in solving this problem? If P_1 is the specification of a realistic problem, the theory T_1 of the problem will probably be extremely complicated. The details of P_1 may be so large in number or so complex, that they entirely obscure the way to a solution of P_1, i.e. a proof of A_1 in T_1. It is often easy to see that certain details of P_1 are completely irrelevant to a proof of A_1. Other details can be seen to be partly irrelevant in that they are not strictly required for a proof of A_1, but they can make the proof of A_1 simpler.

What we can do now is to try and construct an abstraction P_0 of P_1. In the rules of the theory T_0 of P_0 we then try to capture those details of T_1 which are thought to be relevant to the proof of A_1. Since the theory T_0 of P_0 will differ from the theory T_1 of P_1, the statement A_1 of P_1 must be

reformulated in P_0 as A_0. The construction of P_0 should go hand in hand with the construction of an abstraction function Φ from T_1 to T_0 with $\Phi(A_1) = A_0$. (Notice that the easiest way to dispose of irrelevant formulas in T_1 is not to contain them in the domain of Φ.) The problem P_0 may be expected to be simpler than P_1 because we "omitted" the irrelevant details from P_1. Consequently, a solution for P_0 is more easily found than for P_1.

1.2.3. Reconstructing the solution of the problem

Suppose we have found a solution S_0 for the problem P_0. How can we use S_0 to obtain a solution for P_1? S_0 is a proof of A_0 in T_0, hence S_0 is a derivation $(P_1,C_1),\ldots,(P_n,C_n)$ of A_0 from $\emptyset$ in T_0. (Notice that $C_n = A_0$.) For each $F \in P_1 \cup \{C_1\} \cup \ldots \cup P_n \cup \{C_n\}$ choose a unique $\Psi(F) \in dom(\Phi)$ such that $\Phi(\Psi(F)) = F$ and $\Psi(A_0) = A_1$ (this is possible because Φ is a surjection on the set F_0 of formulas of T_0). We know that for each finite $P \subset dom(\Phi)$, $C \in dom(\Phi)$ and $(\Phi(P),\Phi(C)) \in R_0$ (where R_0 is the set of rules of T_0) there is a derivation of C from P in T_1. (These derivations must be constructed when proving that Φ is an abstraction function from T_1 to T_0.) Hence with each rule $(P_i,C_i) = (\Phi(\Psi(P_i)),\Phi(\Psi(C_i)))$ a derivation S_i of $\Psi(C_i)$ from $\Psi(P_i)$ in T_1 can be associated $(i = 1,\ldots,n)$. Since $\Psi(C_n) = A_1$ it is easy to see that $S_1,\ldots,S_n$ is a derivation of A_1 from $\emptyset$ in T_1, i.e. $S_1,\ldots,S_n$ is a solution of P_1. This shows how a solution of P_0 can be transformed into a solution of P_1.

In the above we assumed that we made the abstraction function Φ explicit. This is not always convenient. In practice people often make abstractions of problems without making the abstraction function explicit. They keep this function somewhere in the back of their minds. Having found a solution for the "abstract" problem, they use their intuitive notion of the abstraction function to reconstruct the solution of the "concrete" problem. In contrast to the construction sketched in the previous paragraph this "solution" need not automatically be correct. No harm is done, however, if the "solution" is proved to be correct separately. (Abstraction is used then only to find the solution, and not to prove it correct.)

It is easy to see that if P_0 is an abstraction of a problem P_1 and P_1 is unsolvable, then P_0 is also unsolvable. If P_1 is solvable, P_0 may very well be unsolvable. It is therefore important not to abstract too much and keep abstractions "proper". Even if an abstraction P_0 of a solvable problem P_1 is solvable, "overabstraction" may render a solution of P_0 completely useless. The most dramatic example of this is the problem $P_0 = \langle\langle\{A\},\{(\emptyset,A)\}\rangle,A\rangle$ (where A is an arbitrary formula), which is a proper abstraction of each solvable problem P_1. A solution of P_0 (e.g., $(\emptyset,A)$) cannot tell us anything interesting about the solution of P_1 (see also below).

1.2.4. Good abstractions

In order to indicate what a "good" abstraction is let us consider a solvable problem $P_1 = \langle T_1,A_1\rangle$ and an abstraction $P_0 = \langle T_0,A_0\rangle$ of P_1 with abstraction function Φ from P_1 to P_0. Solving P_1 consists of finding a proof for the following assertion:

(1) $\emptyset \vdash_{T_1} A_1$.

If we try to solve P_1 through the abstraction P_0 with abstraction function Φ, this amounts to finding proofs for the following assertions (where R_0 is the set of rules of T_0):

$$(2)\quad \begin{cases} \emptyset \vdash_{T_0} A_0, \\ P \vdash_{T_1} C \quad (P \subset dom(\Phi),\ P \text{ finite},\ C \in dom(\Phi),\ (\Phi(P),\Phi(C)) \in R_0). \end{cases}$$

An abstraction of a problem is a "good" abstraction, if the proofs of the assertions (2) are considerably simpler than the proof of assertion (1). It becomes clear now why the problem $P_0 = \langle\langle\{A\},\{(\emptyset,A)\}\rangle,A\rangle$ is not a good abstraction of any problem $P_1 = \langle T_1,A_1\rangle$ (except the most trivial problems): In order to prove that P_0 is an abstraction of P_1 we have to prove that $\emptyset \vdash_{T_1} A_1$ (because $(\Phi(\emptyset),\Phi(A_1))$ is a rule of the theory of P_0), but the fact that we could not prove that $\emptyset \vdash_{T_1} A_1$ was the very reason to make the abstraction!

The "art" of abstraction, as it turns out above, consists of finding a good balance between the "level" of the abstraction (which should be as high as possible) and the complexity of the correctness proof of the abstraction (which should be as low as possible). Finding this balance requires a certain amount of "training", and, of course, for certain "intrinsically complex" problems good abstractions are hard or even impossible to find. Yet, even for those problems, the reduction of complexity which can be achieved through abstraction should not be underestimated. Once a problem is in its abstract form it is much easier to discover and survey possible solutions. A problem in its abstract form can also more easily be presented to other people and solutions of the abstract problem are more likely to be generally applicable. This will be demonstrated in Chapters 6 and 7, where a storage management problem will be presented and solved in its abstract form. A secondary advantage of solving problems "in the abstract" is that it may protect people against the apparently ineradicable tendency to create unnecessarily complex ("tricky") solutions to problems.

1.2.5. Abstraction and top-down problem solving

Abstraction becomes an even more powerful technique when it is used in conjunction with a top-down problem solving technique (such as "stepwise refinement", "structured design", "separation of concerns", "divide and rule", which are basically all names for the same technique). Consider a complex problem $\langle T,A\rangle$, which is already in its abstract form. Even though the problem is in its abstract form, it will generally be impossible to find a solution in a direct way. We therefore try to find a "decomposition" of the problem into a number of simpler problems. That is, we choose problems $\langle T,A_1\rangle,\ldots,\langle T,A_n\rangle$, which are believed to be simpler than $\langle T,A\rangle$, and which are such that $\{A_1,\ldots,A_n\} \vdash_T A$ holds. It is easy to see that if we have solved the problems $\langle T,A_1\rangle,\ldots,\langle T,A_n\rangle$, we also have solved $\langle T,A\rangle$. Thus we have reduced $\langle T,A\rangle$ to a number of simpler problems. (Notice, however, that the $\langle T,A_i\rangle$ need not all be solvable, even if $\langle T,A\rangle$ is solvable. The $\langle T,A_i\rangle$ must therefore be chosen with great care.)

Having reduced $\langle T,A\rangle$ to a number of simpler problems $\langle T,A_1\rangle,\ldots,\langle T,A_n\rangle$, we can try to solve each $\langle T,A_i\rangle$. The solution of $\langle T,A_i\rangle$, though simpler than the solution of $\langle T,A\rangle$, may still be difficult. Since $\langle T,A_i\rangle$ is a "subproblem" of $\langle T,A\rangle$, we will probably not need the entire theory T for the solution of $\langle T,A_i\rangle$. We can reduce the complexity of $\langle T,A_i\rangle$ then by making an abstraction $\langle T',A_i'\rangle$ of $\langle T,A_i\rangle$, which contains only the relevant information. If necessary, we can again refine $\langle T',A_i'\rangle$ into simpler problems, etc.. This is schematically pictured in Figure 1.1. Notice that, while going down in Figure 1.1, both the "level of decomposition" and the

"level of abstraction" increase.

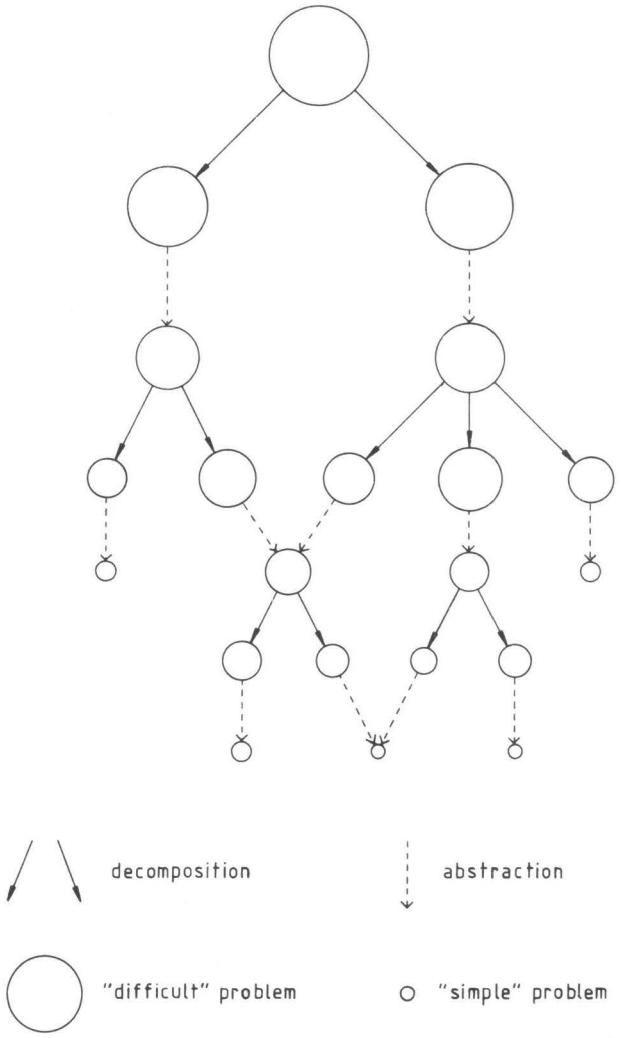


Figure 1.1

1.2.6. Standard theories and problems

The above iterative process of consecutive decompositions and abstractions, if "properly" applied, will eventually lead to a solution of the problem (i.e., if the problem is solvable). Usually it will not be necessary to go down "all the way" by reducing a problem to absolutely trivial problems. In each field of science there are certain standard

theories and standard problems with known solutions. The most illustrative example of this is mathematics, where we have a great many very abstract (and consequently generally applicable) theories like "group theory", "lattice theory", etc.. Each theory has a number of standard problems (theorems) with known solutions (proofs). Examples from computer science are theories such as "parsing theory", "automata theory" or the theory associated with a given programming language. Once a problem has been reduced to a number of standard problems, there is no use in decomposing these problems any further. This would only be a waste of time, since the well-known solutions of these problems can be used.

The above shows how important it is to be familiar with the fundamental theories of a field of science when solving problems in that field. The familiarity with such theories can serve as a guide in choosing the proper decomposition of a problem. Certain subproblems, the abstractions of which correspond to standard problems, may be "recognized" in a problem and indicate a way to an overall solution of the problem. A mathematician will for example recognize a theorem from the theory of topological spaces in proving a theorem about complex functions and make this theorem a component of his proof. A compiler writer will usually recognize a parsing problem in constructing a compiler and make this problem a "phase" of his compiler (knowing that the problem can be solved in a standard way). There is no need to say that the recognition of such standard problems is greatly facilitated if the level of abstraction is always kept as high as possible.

1.3. ABSTRACTION IN CLASSIFYING PROBLEMS AND THEIR SOLUTIONS

Each field of a science can usually be divided into a number of relatively independent subfields, which we shall call "subjects". The field of "language implementation" of computer science contains for example subjects such as "parsing", "register allocation", "garbage collection", etc.. Each subject can be viewed as a collection of "facts", where a fact is a pair $\langle P,S \rangle$, P is a problem and S is a solution of P. In the course of time, with many people working on a subject, the number of facts of a subject, i.e. the "knowledge" of the subject, may grow enormously. Time has come then to order this knowledge in a systematic way. Such a systematic treatment of the subject is not only important to the novice, whose acquaintance with the subject is greatly facilitated, but also to the expert, who can use it to widen his view and keep the growing volume of the subject under control. The purpose of this section is to indicate how abstraction can be used as a tool in giving such a systematic discussion of a subject.

1.3.1. Concrete problems

Let us consider a subject as a large class of concrete facts, where each concrete fact is a pair consisting of a concrete problem and a concrete solution, i.e. some existing problem and an actually realized solution of this problem. The subject of "garbage collection" consists e.g. of facts $\langle P(L,M),G \rangle$, where $P(L,M)$ is the problem of garbage collection in an implementation of a programming language L on a machine M and G is a garbage collector. The same concrete problem may have several concrete solutions. A first approach to discussing a subject would be to separately discuss each concrete problem together with its concrete solutions.

1.3.2. Isolated problems

The approach mentioned above suffers from two major flaws. First, the number of concrete problems is usually very large, which makes the overall discussion very long. Secondly, concrete problems generally contain a vast amount of details that are completely irrelevant to (the solutions of) the problem. These details may entirely obscure the discussion. This situation can be repaired by abstracting from the irrelevant details, a process that will be called isolation of the problem. For each concrete problem P with concrete solutions $S_1,\ldots,S_n$ isolation of P amounts to constructing an abstraction P' of P, which contains only the details relevant to P *and* to its solutions $S_1,\ldots,S_n$. The latter is very important because it is generally very simple to make an abstraction P' of P in such a way that certain solutions of P are no longer expressible in P'. By keeping the solutions $S_1,\ldots,S_n$ expressible in P', they can be "translated" into corresponding solutions $S_1',\ldots,S_n'$ of P'. The problem P' will be called an isolated problem and the solutions $S_1',\ldots,S_n'$ of P' will be called the isolated solutions.

The process of isolation of a problem is a highly complex process, which is far more than just "omitting" irrelevant details. The isolation of a concrete problem generally asks for a complete reformulation of the problem, using some complex abstraction function. In the case of garbage collection, for instance, isolating a concrete problem amounts to formulating the problem in a way as machine and language independent as possible. If the process of isolation is properly applied it will be seen that several isolated problems, corresponding to different concrete problems, coincide. So, each isolated problem represents a number of "equivalent" concrete problems.

A second approach to discussing a subject is now to discuss each isolated problem corresponding to a concrete problem instead of the concrete problem itself, together with its isolated solutions. This meets the previously raised objections to the first approach. It also introduces a new difficulty. The "concepts" that occur in isolated problems are usually rather abstract, and it may not be trivial at all to recognize what the correspondence between these concepts and the "concrete concepts" is. (This correspondence is hidden in the abstraction function.) Consequently, the discussion may easily become incomprehensible. The obvious way to prevent this is to illustrate each abstract concept by some "down to earth" equivalent.

1.3.3. Basic problem

Still the second approach is not very satisfactory. The point is that the coherence is missing. The subject is reduced to a collection of isolated problems, which are discussed independently of each other. But what is the reason to call this collection of problems a "subject"? Obviously the reason is that these problems "have something in common". Exploiting this similarity cannot only clarify the discussion a great deal, it can also save a lot of work. In order to discuss how this should be done, it is necessary to make more explicit what the problems constituting a subject "having something in common" means. Intuitively it means that these problems are instances of the same very general problem. In more formal terms the meaning is given by the following postulate: There exists a nontrivial problem P such that P is a proper abstraction of each isolated problem of the subject (and therefore also of each concrete problem). This problem will be called the basic problem and its solutions will be called

the basic solutions.

According to Lemma 1.1 the isolated problems can now be reformulated in such a way that all formulas and rules of the basic problem are also formulas and rules of all isolated problems, and moreover, that all statements of the isolated problems are equal to the statement of the basic problem. (The isolated solutions should of course be reformulated simultaneously.) Each solution of the basic problem is now also a solution of each isolated problem (not the reverse). Yet as a rule such a basic solution will be so general, that it is of little or no practical value.

A third, more coherent approach to discussing a subject is now ready at hand. At the beginning of the discussion the basic problem is introduced. Subsequently each isolated problem is treated as a specialization of the basic problem. This implies that only the additional details and solutions of the isolated problems are discussed. (Notice that specialization is literally a matter of adding details.)

1.3.4. Generalized problems

The above approach exploits the global similarity of the isolated problems. But there is also something like "local" similarity. Two isolated problems can be very much alike and differ only in a few details and solutions. (Here the word "few" should not be taken too literally, because it may mean "infinitely many" in practice.) In order to discuss how to exploit this, consider a set V of isolated problems, which apart from a few details are the same. Notice that, if $\langle\langle F_B, R_B\rangle, A_B\rangle$ is the basic problem, then each element V of V has the form $\langle\langle F_V, R_V\rangle, A_V\rangle$ with $F_B \subset F_V$, $R_B \subset R_V$ and $A_B = A_V$. As was done with the set of all isolated problems to obtain the basic problem, it is possible to abstract from the different details of the isolated problems in V. This amounts to constructing the problem $G = \langle\langle \cap\, V \in V\ [F_V], \cap\, V \in V\ [R_V]\rangle, A_B\rangle$, which contains the coinciding details of the isolated problems in V. G will be called a generalized problem and its solutions will be called generalized solutions. Each solution of G is also a solution of each isolated problem in V. Moreover, since the isolated problems in V differ only in a few details, G will also differ from the isolated problems in V in only a few details. Consequently, a "good" solution of G is likely to be a good solution of any of the isolated problems in V. Generalized problems thus provide a way to discuss solutions of a number of isolated problems together.

The process of grouping problems together and abstracting from their different details will be called generalization. This process cannot only be applied to isolated problems, it can also be applied to generalized problems themselves. By doing so "generalized generalized problems" are obtained, which will also be called "generalized problems". Then again it is possible to generalize over (groups of) these problems, etc.. Eventually this process will yield a hierarchy of generalized problems. If the basic problem has been chosen properly, it will be at the top of the hierarchy: It can be viewed as a generalization over all isolated problems. At the bottom are the isolated problems, which can be viewed as generalizations over individual isolated problems.

The above leads to a fourth approach to the discussion of a subject. In this approach the hierarchy of generalized problems is traversed in a top-down fashion, instead of the bottom-up fashion used during the generalization process. Each generalized problem, except the basic problem, can then be discussed as a slightly more detailed specialization of another generalized problem, ending up in the discussion of the isolated problems. The solutions of the isolated problems should be moved up as much as

possible in the hierarchy. This implies that a solution S of some isolated problem should be discussed with the highest generalized problem P in the hierarchy for which S is a solution. (S then is automatically a solution of each generalized problem lower than P.) The advantage of this approach is not only that it limits the efforts of discussing the problems and solutions which are part of the subject (by combining them into generalized problems), but also and above all that it clearly reveals the structure of the subject. It is only through this structure that it is possible to properly survey the subject.

1.3.5. Systematic generalization

Related to the fourth approach there is a little difficulty. It is caused by the fact that the grouping of problems together into generalized problems is far from unique. Indeed the criterion for grouping problems together was that these problems differed only in a few details. This, however, is a rather vague criterion, which allows many interpretations. As a consequence, equally many hierarchies of generalized problems can be constructed. In practice the situation will probably not be so bad, because there usually is a "natural" way to group problems together. Still it is a good thing to know that there exists a unique hierarchy of generalized problems, which corresponds to a systematic way of generalization. This unique hierarchy enables us to speak of *the* structure of the subject. In addition it has some very pleasant properties.

If J is the (finite) set of all isolated problems of the subject and $\langle\langle F_B, R_B\rangle, A_B\rangle$ is the basic problem, then the standard hierarchy of generalized problems is defined as the partially ordered set $\langle G, \sqsubset\rangle$, where

$$G = \{\langle\langle \cap\, V \in \mathcal{V}\ [F_V], \cap\, V \in \mathcal{V}\ [R_V]\rangle, A_B\rangle \mid \mathcal{V} \subset J,\ \mathcal{V} \neq \emptyset\},$$

$$P \sqsubset Q \Leftrightarrow F_P \subset F_Q \wedge R_P \subset R_Q \quad (P, Q \in G).$$

Here, for each problem P, F_P and R_P denote the set of formulas and the set of rules of the theory of P respectively. The above defines a unique way of grouping problems together into generalized problems (see Example 1.3). Notice that the relation $\sqsubset$ on G is not the same as the abstraction relation. Though $P \sqsubset Q$ implies that P is an abstraction of Q, the reverse need not hold. Notice also that $F_B \subset \cap\, V \in J\ [F_V]$ and $R_B \subset \cap\, V \in J\ [R_V]$ but not necessarily $F_B = \cap\, V \in J\ [F_V]$ and $R_B = \cap\, V \in J\ [R_V]$. It is reasonable, though, to assume that the basic problem has been chosen in such a way that the latter holds, or otherwise force the latter to hold by redefining the basic problem, thus guaranteeing that $\langle\langle F_B, R_B\rangle, A_B\rangle \in G$.

EXAMPLE 1.3

Suppose $J = \{P_1, P_2, P_3, P_4\}$ is the set of all isolated problems and all P_i $(i = 1,2,3,4)$ have the same set of formulas. We can consider the P_i as sets of rules then. Let the P_i be given by:

$$P_1 = \{a, b, c\},$$
$$P_2 = \{a, b, d\},$$
$$P_3 = \{a, d, e\},$$
$$P_4 = \{a, d, f\},$$

where a, b, c, d, e and f are rules. Then the set G of generalized problems from the standard hierarchy of generalized problems is equal to:

$$G = \{\{a\},\{a,b\},\{a,d\},\{a,b,c\},\{a,b,d\},\{a,d,e\},\{a,d,f\}\}.$$

This defines the grouping which is pictured in Figure 1.2 (where $\rightarrow$ denotes specialization).

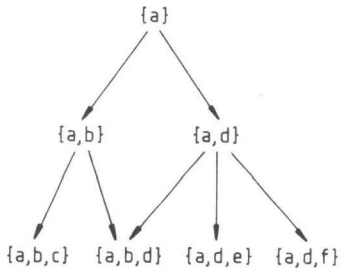


Figure 1.2

□

The standard hierarchy of generalized problems satisfies the following property: For any combination of details occurring in a generalized problem, there is a unique generalized problem Q containing those details, such that any generalized problem containing those details is a specialization of Q. This has the following two pleasant implications:

(1) Each detail needs to be introduced in only one generalized problem Q. Any other generalized problem containing this detail is a specialization of Q. This is important because the introduction of a detail may involve some overhead (such as the introduction of auxiliary notions).
(2) Each solution based on a set $\mathcal{D}$ of details needs to be discussed in only one generalized problem Q. Any other generalized problem to which the solution applies (i.e., which contains the details in $\mathcal{D}$) is a specialization of Q.

These two facts not only save work, but also contribute to a clear presentation of the subject.

1.3.6. Pruning the hierarchy

Using a hierarchy such as defined above, the application of the fourth approach will result in a systematic and exhaustive discussion of a subject. For comprehensive subjects such an exhaustive treatment may easily fill a library. Usually that is not what is intended. As the ultimate reason for someone to go into a subject may be taken to be that he has (or will have) a problem pertaining to the subject. Unless his problem is an "old" problem, the chance to come across an exact copy of his problem (in its isolated form) is quite small. Therefore he will look for a proper generalization of his problem, and solve his problem either by copying a solution of the generalized problem, by modifying it, or by inventing a

solution of his own. (This clearly demonstrates that a subject is not immutable, but, quite on the contrary, constantly growing. So any discussion of the subject is necessarily a snapshot of the "state of the art".) This suggests that, in order to keep the length of the discussion under control, the hierarchy of generalized problems be "pruned", which implies that lower parts of the hierarchy are left out of the discussion. The pruning should be done with great care, so as to avoid the occurrence of "gaps". Dependent upon the amount of pruning one can distinguish between:

(1) An introduction to the subject.
(2) A survey of the subject.
(3) A book on the subject.
(4) An encyclopedia of the subject.

As a result the fifth approach to the discussion of a subject is obtained. Starting with the discussion of the basic problem and its solutions, a pruned version of the hierarchy of generalized problems is traversed in a top-down fashion. Each generalized problem, except the basic problem, is discussed (together with its solutions) as a specialization of its direct predecessor in the hierarchy. The five approaches are visualized in Figure 1.3 (where $\rightarrow$ denotes specialization).

1.3.7. Ordering the solutions

The above shows how the problems of a subject can be ordered systematically. The ordering of the problems induces an ordering on the solutions as well. The latter ordering, like the former, is based on the different details of problems and does not work for solutions which are based on exactly the same details of a problem (these solutions are associated with the same generalized problem). Apart from details of a problem there is also something like details of a solution: Two solutions, even if they are associated with different generalized problems, may be very much alike. Just like we did for problems, we could try to exploit the similarity of solutions. This would not only enable us to make the ordering of solutions complete, but also to clearly indicate the relation between the various solutions. In order to discuss this we have to be more specific about the kind of problems and solutions we consider.

The problems discussed in this monograph are mainly "algorithmic problems", which have the following form: Construct an algorithm S, which given the precondition P establishes the postcondition Q. We already showed how an algorithm could be viewed as a "solution" of a "problem", i.e. as a proof of a formula in a suitable theory. Just like mathematical proofs, one algorithm can be "simpler" or contain "less detail" than another algorithm. Unlike mathematical proofs, we are usually not satisfied with simple algorithms as solutions to algorithmic problems (at least not in practical situations). The reason is that simple algorithms are generally "inefficient". The notion of "efficiency" (which, strictly speaking, is relative to some computation model) is meaningless for mathematical proofs. (If proofs are written in a language such as AUTOMATH [DE BRUIJN 80] it might be given a meaning, though.) For algorithms the notion of efficiency is crucial.

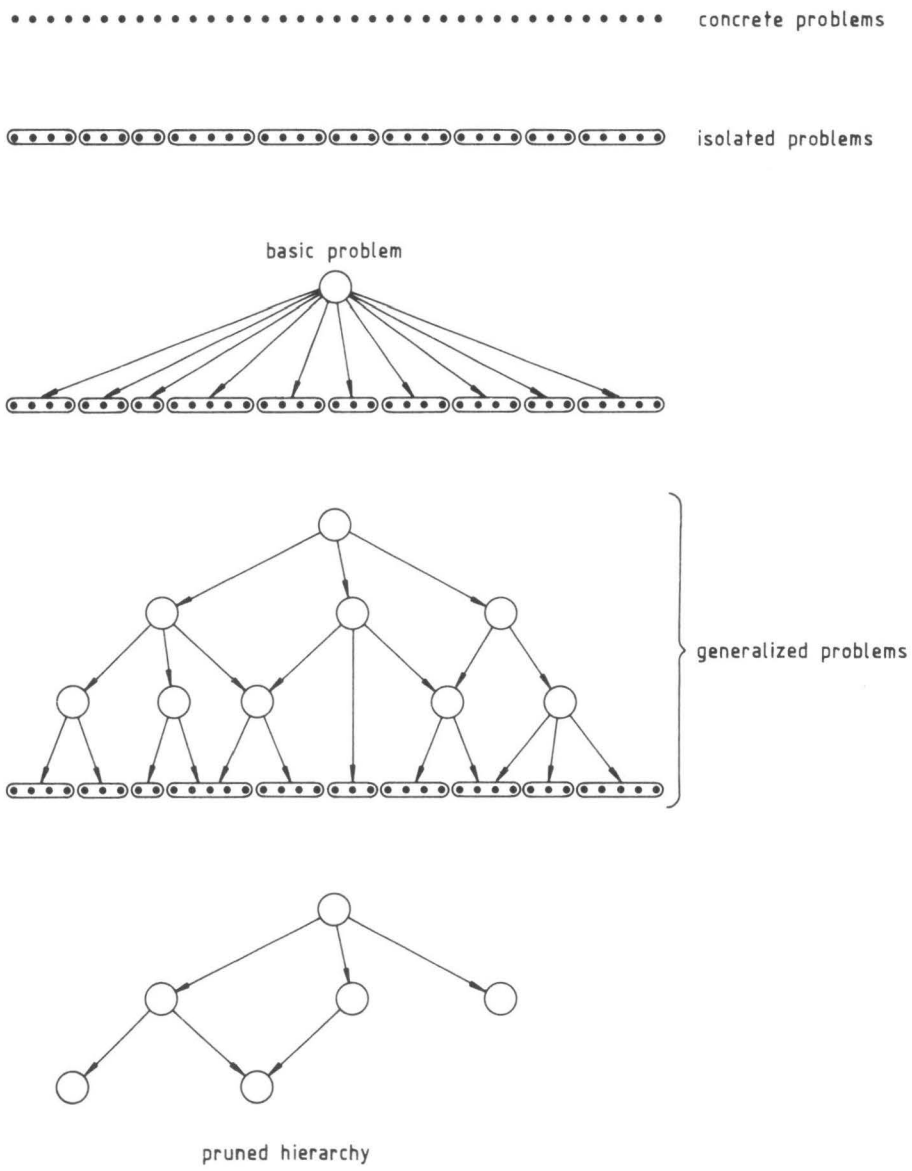

concrete problems
isolated problems
basic problem
generalized problems
pruned hierarchy

Figure 1.3

1.3.8. Correctness-preserving transformations

Since most efficient algorithms are necessarily detailed (but not the reverse!), the construction of an efficient algorithm is usually far from simple. On the other hand, the construction of a simple but inefficient algorithm is often relatively easy. An obvious approach to constructing an algorithm is to start with the simple inefficient algorithm and try to mold this algorithm into a more efficient algorithm. The tool to be used for this is the "correctness-preserving transformation" [GERHART 75], which takes a correct algorithm and transforms it into another correct (and hopefully more efficient) algorithm. The above approach to algorithm construction has become rather popular lately. It will be discussed in more detail in Chapter 3, where a simple method for performing correctness-preserving transformations will be presented and exemplified.

Correctness-preserving transformations are not only useful in the construction of algorithms, they can also successfully be used in the classification of algorithms [DARLINGTON 78]. Instead of using them primarily to improve the efficiency of algorithms, we can use correctness-preserving transformations to derive more detailed algorithms from less detailed ones. As such correctness-preserving transformations can be used as a tool in ordering solutions in addition to the ordering imposed by the hierarchy of problems and in discussing the solutions in a coherent and time-saving way. This works as follows.

Choose some very simple (very "abstract") algorithm S as the first solution of the basic problem to be discussed. Usually this algorithm can easily be seen to be correct. Then discuss all other more detailed solutions of the basic problem by showing how they can be obtained from S through correctness-preserving transformations. When going down one level of abstraction in the hierarchy of problems, discussing the generalized problem P, the same approach can be used for the discussion of the solutions associated with P. In contrast to the basic problem, there is no need to prove the correctness of the algorithm which is chosen as the starting point of the transformation process: A solution of a generalization of P (in this case the basic problem) can be chosen as such. It is even conceivable to choose more than one starting point by deriving a number of solutions of P directly from different solutions of a generalization of P (dependent upon the similarity of the former and the latter solutions). When going down additional levels of abstraction, this process can be continued until finally "all" solutions of generalized problems are obtained as transformations of the single abstract algorithm which was chosen as the starting point.

The entire approach described above will be used in Chapter 5 to discuss the subject of (compacting) garbage collection. (In fact we will split the subject into two subjects, "garbage collection" (without compaction) and "compaction", which will both be discussed as sketched above.) A final remark is that, in contrast to the ordering of problems, the ordering of solutions through correctness-preserving (and "detail-increasing") transformations is far from unique, i.e. as far as this ordering is not implied by the ordering of problems. There are usually various ways to obtain algorithms from other algorithms through correctness-preserving transformations (as we will see in Chapter 5), each of which corresponds to a different order of discussion. The choice of this order therefore is a rather subtle matter, whereby the guiding principle should be that the order corresponds to an increase in detail.

1.4. CONCLUSION

In the foregoing we discussed how abstraction can be used as a systematic tool in solving problems and in classifying problems and their solutions. None of the techniques which were discussed are novel. In a simplified way they reflect what everyone does automatically when solving a problem or when trying to grasp a subject. However, "automatically" does not yet mean "systematically". Making the mechanisms of abstraction explicit may aid in a more systematic application of these mechanisms. The result of systematically applying these mechanisms in the fields of storage management and garbage collection will be presented in Chapters 4-7 of this monograph. In contrast to the process of abstraction, which by its very nature is a bottom-up process, the presentation will be top-down. The latter is the natural way to transfer knowledge (as opposed to gathering knowledge). Everything which has been said in this chapter about the preceding process of abstraction could therefore simply be omitted. Yet, for a better comprehension of the underlying philosophy and as an incitement to a more systematic use of abstraction this chapter may be useful.

CHAPTER 2

SPECIFICATION

2.0. INTRODUCTION

In each science the need arises to describe the objects which are being studied. The main purpose of such descriptions or "specifications" is to serve as a means of communication. Specifications can make sure that people are talking about the same things. Yet, specifications are more than that. They can also serve very well as a "feedback" mechanism and handhold in thinking about problems.

The objects of study in computer science are "algorithms" and "data structures". The specification problem for algorithms and data structures differs from specification problems in other fields of science in one important respect: Apart from communication between people, specifications are also meant for communication between people and machines, and even for mutual communication of machines. (Note that the concept of a specification is used in a broad sense here. "Programs" and "bus standards", for example, are also considered as specifications, though at a low level of abstraction.) Machines, as opposed to people, are highly accurate, but also very inflexible listeners. In specifications meant for people (such as recipes in a cookbook) one can afford to take certain things for granted (like "boiling an egg"). In specifications meant for machines this is absolutely out of the question. Specifications of algorithms and data structures must therefore have *mathematical rigour*.

The purpose of this chapter is to describe a mathematically rigorous specification method for algorithms and data structures. This method, in contrast to several other specification methods, is suitable for the specification of all (sequential) algorithms and data structures normally encountered in programming practice. For reasons of time and space the method will not be elaborated in full detail. The necessary foundations will be laid and the way to erect a complete building on these foundations will be indicated. The actual construction of this building is left for later research.

2.0.1. Specifications and their meaning

In order to provide the mathematical rigour mentioned above it is essential to make a clear distinction between specifications and the objects described by them. The lack of this distinction has confused the discussion on specification methods for data structures for a long time. To begin with, let us agree on what we use the terms "algorithm" and "data structure" for. These terms will be used to denote the intuitive concepts of an algorithm and a data structure, with which each computer scientist is familiar.

A specification, in informal terms, is a description of an algorithm or data structure. The requirement of mathematical rigour implies that the description is written in a formal language, the "specification language". A specification, therefore, is basically nothing but a finite sequence of symbols. However, a specification is supposed to "specify" something or, in

other words, a specification must have a "meaning". The obvious choice is to take an algorithm or data structure as its meaning. The requirement of mathematical rigour, on the other hand, implies that the meaning of a specification is a mathematical object, which algorithms and data structures are not (as we agreed on above). Consequently, it makes sense to distinguish between the "intuitive" and the "formal" meaning of a specification: The intuitive meaning of a specification is an algorithm or data structure and the formal meaning is some mathematical object which "models" the algorithm or data structure. The former can only be described intuitively, while the latter can be defined rigorously. Of course, the mathematical models chosen for algorithms and data structures should correspond to the intuitive concepts as closely as possible.

We have three things now: specifications, their formal meaning and their intuitive meaning. The first are sequences of symbols in a formal language, the second are mathematical objects and the third are intuitive notions.

2.0.2. Implicit and explicit specification methods

There are two fundamentally different approaches to the specification of algorithms and data structures (cf. [LISKOV & ZILLES 75]). The first, which we shall call the "implicit" approach, is to describe the properties which the algorithm or data structure should satisfy in an axiomatic way (cf. [GOGUEN et al. 78], [GUTTAG & HORNING 78]). The major advantage of this method is its minimality: Only the essential properties of the algorithm or data structure are reflected in the specification. There are also two severe drawbacks. Apart from very simple algorithms and data structures it is very difficult to construct complete and consistent specifications. Specifically algorithms and data structures involving "dynamic" and "shared" data, which are frequently encountered in practice, are hard to specify. Moreover, implicit specifications are usually far from easy to comprehend.

The second way of specification, which we shall call the "explicit" approach, is to choose a "representation" and to describe the algorithm or data structure directly in terms of this representation (cf. [BERZINS 79]). This method clearly contrasts with the implicit method as to its advantages and disadvantages. First of all, specifications are more easily constructed. If the possibility of dynamic creation and sharing is already included in the representation chosen, algorithms and data structures featuring these properties are readily specified. Explicit specifications also tend to be more readable than implicit specifications. The salient disadvantage, of course, is the fact that those specifications are not representation-independent. If one is not very careful "internal" details of the representation chosen may permeate into the external world and lead to an "overspecification" of the algorithm or data structure. (Contrast this with the problem of writing complete implicit specifications.)

It is my firm belief that for realistic applications the future lies in the explicit approach. A precondition is, that the problem of representation-dependence is solved satisfactorily. The key to a solution of this problem lies in the observation that the choice of a representation need not depend on efficiency considerations. The only criteria in choosing a representation should be the clarity and naturalness of the specification. This implies first of all that the representations themselves must be free of implementation detail, or, in other words, they should be as abstract as possible. In particular, representations for algorithms ("control structures") should not include such things as labels

and gotos, and representations for data structures should not include such things as pointers, fixed size storage cells, etc.. On the other hand, the possibility of dynamic creation and sharing should be inherent (otherwise many applications are ruled out).

The specification method which will be described in this chapter can be classified in the category of explicit specification methods. It is based on a novel kind of representation for data structures, which is believed to satisfy the requirements mentioned above. These representations can be viewed as abstract "storage structures" and will be called "structures", for short. Structures are free of low level concepts such as pointers and garbage, while at the same time they are general in that they allow the representation of data structures with arbitrary sharing and circularities. All useful operations on structures (such as creation and replacement) can be described in terms of only three primitive operations. The use of structures is not restricted to specification languages. It is envisaged that they can successfully be used in definitions of programming languages as well, especially in definitions of those programming languages which feature sharing ("aliasing") and dynamic creation of data.

2.0.3. Mathematical models for algorithms and data structures

So far we have not discussed what the formal meaning of a specification of an algorithm or data structure should be, or, in other words, which mathematical models we choose for algorithms and data structures. The generality of the structure concept, which enables all such things as "values", "objects", "states", "environments", etc. to be represented by structures, will make it extremely simple to associate mathematical objects with specifications.

First consider algorithms. The standard mathematical model for algorithms is the "computable function" (which can be characterized in many ways, e.g. using Turing machines [TURING 36], lambda calculus [CHURCH 41] or general recursive functions [KLEENE 36]). In mathematics computable functions are usually considered as mappings from natural numbers to natural numbers, but they can easily be extended to mappings from structures to structures. The formal meaning of a specification of an algorithm will therefore be a computable function from structures to structures.

In contrast to algorithms, the question which mathematical model should be chosen for data structures has been a "hot topic" for a long time. The introduction of the concept of an "abstract data type" [LISKOV & ZILLES 74], which is essentially a heterogeneous algebra [BIRKHOFF & LIPSON 70], seems to have settled the matter more or less. Our model for data structures is basically an abstract data type, though we shall view the latter as a homogeneous rather than a heterogeneous algebra. (The reason for this has to do with the fact that we allow the arguments of an operation of a data structure to "overlap".) The formal meaning of a specification of a data structure will be a homogeneous algebra, consisting of the set of all structures and a number of (computable partial) functions from structures to structures.

2.0.4. Abstraction facilities

The concept of a structure, which is the only kind of representation for data structures used in the specification method, enables us to achieve mathematical rigour in a simple way. The structure concept in itself is a highly mathematical and abstract concept. If, however, it were necessary to

spell out specifications in terms of primitive structures and primitive operations on structures, this would make the specification method as a whole far from abstract. The specification language would be nothing but an assembly language for an abstract "structure processor" and would not deserve the title "specification language". The latter title is deserved only by the the introduction of two abstraction facilities in the language: one for data structures and one for control structures.

The abstraction facility for data structures is a mechanism to "encapsulate" [ZILLES 73] the representation chosen for a data structure in a similar way as for example in CLU [LISKOV et al. 77]. This enables data structures to be specified in terms of other (already specified or still to be specified) data structures, without relying on the representation chosen for the latter. Thus "layers" of abstraction can be created and data structures can be specified in a "modular" way. The abstraction facility for control structures consists of the possibility to use highly nondeterministic control structures, up to a level of abstraction where a complex algorithm can be specified as a single "nondeterministic assignment" [HAREL et al. 77], which need not even be executable. The introduction of the former abstraction facility in the language will be discussed informally, while the introduction of the latter is only touched on. A more formal treatment of both facilities constitutes one of the "loose ends" which are left for later research.

2.0.5. From specifications to programs

Even when provided with the above abstraction facilities, certain people will still call the language described in this chapter a programming language rather than a specification language, and, in part, they are right. The language suits itself for use at greatly different levels of abstraction. Used at a high level of abstraction it can be viewed as a specification language and used at lower levels of abstraction it can be viewed as a programming language. The advantage of the fact that the same language can be used as both a specification language and a programming language should not be underestimated. It enables a uniform approach to the implementation of algorithms and data structures. Provided one has the necessary tools (which will be discussed in the next chapter) it is possible to descend from specifications to programs in a systematic, stepwise way without having to change the language halfway through.

The virtues of the above approach will be demonstrated in the succeeding chapters of this monograph. The language used in those chapters is a loose version of the language described in this chapter. In fact, the language described in this chapter emerged from the language used in the other chapters. This chapter can therefore also be viewed as an attempt to make the semantics of the algorithmic language used throughout this monograph more precise. This semantics is not trivial due to the fact that most algorithms and data structures discussed in this monograph incorporate shared and dynamic data.

The discussion of the specification method in this chapter is basically bottom-up. First, in Section 2.1 the fundamental concept of a "structure" is introduced, as well as some related concepts. In Section 2.2 the primitive operations which can be applied to structures are defined. In Section 2.3 a simple yet powerful language for the manipulation of structures is introduced and its semantics is defined. The extension of this language into a full-fledged specification language is discussed in Section 2.4, together with an example of a specification. This section

contrasts with the other sections in that it is very informal. A comparison of structures to other abstract representations is made in Section 2.5. The conclusion of this chapter is presented in Section 2.6.

2.1. STRUCTURES

2.1.1. Definition of a structure

The purpose of this section is to define the concept of a "structure". A structure can be viewed as an abstract "storage structure", which can be "accessed" through special keys called "accessors". Accessors will be considered as primitive concepts, usually denoted by strings of letters and digits. By repeatedly applying accessors to a structure one can follow an "access path".

> An accessor is a primitive concept.
>
> A is the set of all accessors.
>
> A^* is the set of all finite sequences of accessors.
>
> A^+ is the set of all finite nonempty sequences of accessors.
>
> Λ is the empty sequence of accessors.

The sequence $A_1,\ldots,A_n$ of accessors will be denoted by $A_1\ldots A_n$.

The following definition of the concept of a structure is based on the consideration that a (storage) structure is completely characterized by two things: First, the collection of all of its access paths and secondly, a relation which indicates whether two access paths access the same "substructure". (Notice that the latter is necessarily an equivalence relation.) Taking into account the properties of access paths as well, we arrive at the following definition:

> A structure S is a pair $\langle P,\equiv\rangle$, where $P \subset A^*$ and $\equiv$ is an equivalence relation on P such that:
> (1) $\Lambda \in P$.
> (2) $PA \in P \Rightarrow P \in P \quad (P \in A^*,\ A \in A)$.
> (3) $PA \in P \wedge P \equiv Q \Rightarrow QA \in P \wedge PA \equiv QA \quad (P,Q \in P,\ A \in A)$.
>
> A $P \in P$ will be called a path of S.
>
> $\equiv$ will be called the identification relation of S.
>
> An $X \in P|\equiv$, i.e. an equivalence class of $\equiv$, will be called an object of S.
>
> S is the set of all structures.

Property (1) states that the empty sequence of accessors is a path of S (hence $P \neq \emptyset$). Property (2) implies that any prefix of a path of S is also a path of S. Property (3) states that equivalent paths have equivalent continuations. This property of an equivalence relation is known as "right-invariance". The paths of a structure can be viewed as "names" for the objects which they represent. As will be seen later, the concept of an

object as introduced above is closely related to the intuitive concept of an object.

There are three trivial examples of a structure, which will be called the "empty structure", the "convergent structure" and the "divergent structure" respectively:

> $\perp = \langle\{\Lambda\}, \{(\Lambda,\Lambda)\}\rangle$ is a structure called the empty structure.
>
> $\top_C = \langle A^*, A^* \times A^*\rangle$ is a structure called the convergent structure.
>
> $\top_D = \langle A^*, \{(P,P) \mid P \in A^*\}\rangle$ is a structure called the divergent structure.

Notice that $\perp$ and $\top_C$ contain only a single object, while $\top_D$ contains an infinite number of objects (i.e., if $A \neq \emptyset$, which we will from now on assume). Other examples of structures will be discussed below.

EXAMPLE 2.1

Let $S = \langle P, \equiv\rangle$, where

$$P = \{\Lambda, a, b, ba\},$$

$$\equiv = \{(\Lambda,\Lambda), (a,a), (a,ba), (ba,a), (ba,ba), (b,b)\},$$

then S is a structure containing the following objects:

$$P|\equiv = \{\{\Lambda\}, \{a, ba\}, \{b\}\}.$$

Notice that the paths a and ba are "aliases" for one and the same object.

□

Before continuing some notations have to be introduced. First, if $S = \langle P, \equiv\rangle$ is a structure, then P_S and $\equiv_S$ will denote P and $\equiv$ respectively. Secondly, if X is an object of a structure S and P is a path of S such that $P \in X$, then, if no confusion can arise, $\overline{P}$ will denote X. This convention fits in with the common mathematical practice of denoting equivalence classes by their representatives. Definitions and lemmas which use this notation for objects must be proved to be independent of the choice of the representatives for the objects.

2.1.2. Finite structures

The definition of a structure does not preclude that structures use an infinite number of accessors or have an infinite number of objects. Structures that use only a finite number of accessors and have a finite number of objects constitute an important subclass. The structures in this subclass will be called the "finite structures":

> Let S be a structure.
>
> $\{A \in A \mid \exists\, P \in P_S\ [PA \in P_S]\}$ is called the accessor set of S.
>
> S is called finite iff the accessor set and the set of objects of S are finite; otherwise S is called infinite.

The empty structure $\perp$ is an example of a finite structure, and the divergent structure τ_D is an example of an infinite structure. The convergent structure τ_C is infinite if and only if A is infinite.

Finite structures can be pictured in a systematic way as follows:

Drawing algorithm for finite structures

For each object $\overline{P}$
| Draw a circle $C_{\overline{P}}$.
For each pair of objects $(\overline{P},\overline{Q})$
and each accessor A with $PA \in \overline{Q}$
| Draw an arrow labelled by A from $C_{\overline{P}}$ to $C_{\overline{Q}}$.
Label $C_{\overline{\Lambda}}$ by Λ.

Notice that this drawing algorithm is independent of the choice of the paths for the objects and that it would never terminate if applied to an infinite structure. It is easy to see that the picture thus associated with a finite structure is unique.

EXAMPLE 2.2

The empty structure $\perp$ has the following picture:

Figure 2.1

If $A = \{a,b\}$, then the picture of the convergent structure τ_C is:

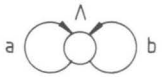


Figure 2.2

If we try the impossible and apply the drawing algorithm to the divergent structure τ_D with $A = \{a,b\}$, then we get:

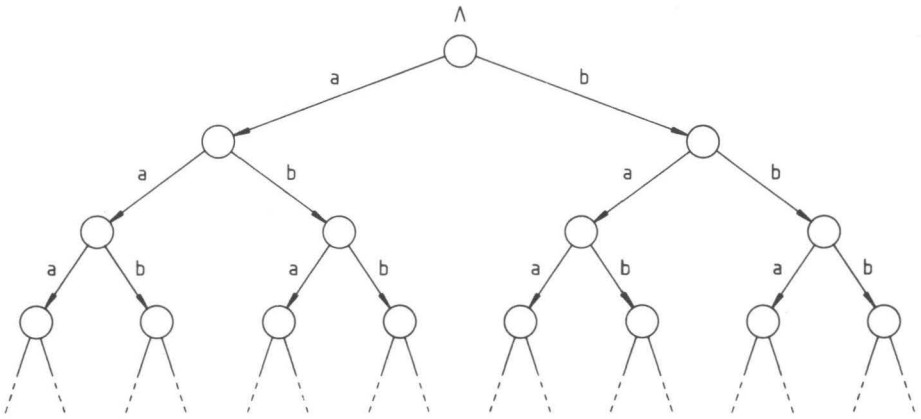


Figure 2.3

The picture of the structure S from Example 2.1 is:

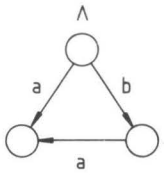


Figure 2.4

□

The above may raise the question what the difference is between a structure and a rooted graph with labelled edges. There are two crucial differences. First, the concept of "unreachability" is meaningless in a structure. Each object has at least one access path. Secondly, objects do not have a separate identity. An object simply *is* the collection of its access paths. These two facts have a number of important consequences which will be discussed in detail in Section 2.5.

2.1.3. Physical inclusion

Another important observation is that the paths of a structure should *not* be considered as "pointers": Though a path can be viewed as a name for an object, paths are not objects themselves. Instead, the arrows in the picture of a structure should be regarded as denoting physical inclusion. Since arbitrary kinds of physical inclusion (such as sharing and

circularity) can be modelled in a structure, the need to introduce pointers will not arise anywhere. The concept of physical inclusion will be made more precise by introducing three relations on the set of objects of a structure:

> Let S be a structure.
> Let $\overline{P}$ and $\overline{Q}$ be objects of S.
>
> $\overline{P}$ is a direct component of $\overline{Q}$ iff there is an $A \in \mathsf{A}$ such that $QA \in \overline{P}$.
>
> $\overline{P}$ is a component of $\overline{Q}$ iff there is an $R \in \mathsf{A}^+$ such that $QR \in \overline{P}$.
>
> $\overline{P}$ is contained in $\overline{Q}$ iff there is an $R \in \mathsf{A}^*$ such that $QR \in \overline{P}$.

Check that these definitions are independent of the choice of P and Q. If $\overline{P}$ is a direct component of $\overline{Q}$ because $QA \in \overline{P}$ for some $A \in \mathsf{A}$, we shall also call $\overline{P}$ a "direct A-component" of $\overline{Q}$. The relations "be a component of" and "be contained in" are both transitive, while the latter is also reflexive. Neither of them need be an (irreflexive or reflexive) partial order (see Example 2.3). The meaning of the fact that an object is "cyclic" can be defined as follows:

> An object of a structure is cyclic iff it is a component of itself.

It is easy to see that cyclic objects contain an infinite number of paths.

EXAMPLE 2.3

Consider the structure S of Figure 2.5.

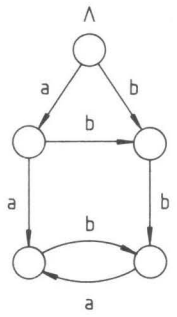


Figure 2.5

The objects of S are:

$$\overline{\Lambda} = \{\Lambda\},$$
$$\overline{a} = \{a\},$$
$$\overline{b} = \{ab, b\},$$
$$\overline{aa} = \{P(ba)^n \mid n \geq 0 \wedge P \in \{aa, abba, bba\}\},$$
$$\overline{bb} = \{P(ab)^n \mid n \geq 0 \wedge P \in \{aab, abb, bb\}\}.$$

The three inclusion relations which hold between these objects can be described schematically as follows (the plus sign indicates where the relation holds):

$\overline{P}$ is a direct component of $\overline{Q}$:

$\overline{P}$ \ $\overline{Q}$	$\overline{\Lambda}$	$\overline{a}$	$\overline{b}$	$\overline{aa}$	$\overline{bb}$
$\overline{\Lambda}$	-	-	-	-	-
$\overline{a}$	+	-	-	-	-
$\overline{b}$	+	+	-	-	-
$\overline{aa}$	-	+	-	-	+
$\overline{bb}$	-	-	+	+	-

$\overline{P}$ is a component of $\overline{Q}$:

$\overline{P}$ \ $\overline{Q}$	$\overline{\Lambda}$	$\overline{a}$	$\overline{b}$	$\overline{aa}$	$\overline{bb}$
$\overline{\Lambda}$	-	-	-	-	-
$\overline{a}$	+	-	-	-	-
$\overline{b}$	+	+	-	-	-
$\overline{aa}$	+	+	+	+	+
$\overline{bb}$	+	+	+	+	+

$\overline{P}$ is contained in $\overline{Q}$:

$\overline{P}$ \ $\overline{Q}$	$\overline{\Lambda}$	$\overline{a}$	$\overline{b}$	$\overline{aa}$	$\overline{bb}$
$\overline{\Lambda}$	+	-	-	-	-
$\overline{a}$	+	+	-	-	-
$\overline{b}$	+	+	+	-	-
$\overline{aa}$	+	+	+	+	+
$\overline{bb}$	+	+	+	+	+

The relation "be a component of" is not an irreflexive partial order here, because it is not irreflexive: $\overline{aa}$ is a component of itself. The relation "be contained in" is not a reflexive partial order because it is not antisymmetric: $\overline{aa}$ is contained in $\overline{bb}$ and $\overline{bb}$ is contained in $\overline{aa}$, but $\overline{aa} \neq \overline{bb}$. This, of course, is caused by the fact that $\overline{aa}$ and $\overline{bb}$ are cyclic objects. □

2.1.4. Structures and regular languages

The above example (and especially the expressions for the objects $\overline{aa}$ and $\overline{bb}$) suggests that there is a relation between structures and regular languages. Indeed, the objects of finite structures *are* regular languages. This can be understood intuitively by considering the picture of a finite structure as the state diagram of a finite state machine and recalling the correspondence between finite state machines and regular languages. A straightforward proof can be obtained by using the fact that each

equivalence class of a right-invariant equivalence relation with a finite index is a regular language [HOPCROFT & ULLMAN 79]. In the proof sketched below the relation between left-linear grammars and regular languages is used:

> LEMMA 2.1
>
> Let S be a finite structure, then each object of S is a regular language over A.

PROOF

Let $\overline{P}_0,\ldots,\overline{P}_n$ be the objects of S, where $\overline{P}_0 = \overline{\Lambda}$. Let $\overline{P}_k$ be one of the objects of S. Construct a left-linear grammar in the following way:

For each $i = 0,\ldots,n$
| Introduce a nonterminal symbol N_i.
For each $i,j = 0,\ldots,n$
and each accessor A with $P_jA \in \overline{P}_i$
| Introduce the production rule $N_i \rightarrow N_jA$.
Introduce the production rule $N_0 \rightarrow \Lambda$.
Choose N_k as the start symbol.

The grammar constructed this way is left-linear and independent of the choice of the P_i. Moreover, the language generated by this grammar can be proved to be equal to $\overline{P}_k$ (use induction over the path-length in one direction, and induction over the length of the derivation in the other direction). The objects $\overline{P}_i$ therefore constitute left-linear languages. Since the latter coincide with the regular languages, they are also regular languages. Notice that the grammars associated with the different objects above differ only in the choice of the start symbol. □

EXAMPLE 2.4

The left-linear grammars associated with the objects of the structures of Figures 2.1, 2.2, 2.4 and 2.5 are:

Figure 2.1:
Nonterminals: N_0 (for $\overline{\Lambda}$).
Production rules: $N_0 \rightarrow \Lambda$.
Start symbol: N_0.

Figure 2.2:
Nonterminals: N_0 (for $\overline{\Lambda}$).
Production rules: $N_0 \rightarrow N_0a$, $N_0 \rightarrow N_0b$, $N_0 \rightarrow \Lambda$.
Start symbol: N_0.

Figure 2.4:
Nonterminals: N_0 (for $\overline{\Lambda}$),
N_1 (for $\overline{a}$),
N_2 (for $\overline{b}$).
Production rules: $N_0 \rightarrow \Lambda$,
$N_1 \rightarrow N_0a$, $N_1 \rightarrow N_2a$,
$N_2 \rightarrow N_0b$.
Start symbol: N_k $(k = 0,1,2)$.

Figure 2.5:
Nonterminals: N_0 (for $\overline{\Lambda}$),
N_1 (for $\overline{a}$),
N_2 (for $\overline{b}$),
N_3 (for $\overline{aa}$),
N_4 (for $\overline{bb}$).
Production rules: $N_0 \to \Lambda$,
$N_1 \to N_0 a$,
$N_2 \to N_0 b$, $N_2 \to N_1 b$,
$N_3 \to N_1 a$, $N_3 \to N_4 a$,
$N_4 \to N_2 b$, $N_4 \to N_3 b$.
Start symbol: N_k $(k = 0,1,2,3,4)$.

□

Due to Lemma 2.1 a regular expression notation can be used for the objects of all finite structures.

EXAMPLE 2.5

The objects of the structures of Figures 2.1, 2.2, 2.4 and 2.5 can be denoted by regular expressions as follows:

Figure 2.1:
$\overline{\Lambda} = \Lambda$.

Figure 2.2:
$\overline{\Lambda} = (a+b)^*$.

Figure 2.4:
$\overline{\Lambda} = \Lambda$,
$\overline{a} = a + ba$,
$\overline{b} = b$.

Figure 2.5:
$\overline{\Lambda} = \Lambda$,
$\overline{a} = a$,
$\overline{b} = ab + b$,
$\overline{aa} = (aa + abba + bba)(ba)^*$,
$\overline{bb} = (aab + abb + bb)(ab)^*$.

□

2.1.5. Structure of objects

The concept of an object as we introduced it is closely related to the concept of a "dynamic object", as it is normally conceived in computer science. Dynamic objects are usually considered as "instances" of "values". Two dynamic objects may be instances of the same value and still be different. In mathematical models for dynamic objects this problem is usually solved by associating an "identity", which is an explicit value, with dynamic objects. As stated before, objects in structures do not have an explicit identity. It is interesting to see how the identity problem for them is solved. The objects in a structure can be viewed as instances of structures (so "structures" correspond to the "values" of dynamic objects). This is made more precise by the following definition of the "structure" of

an object:

> If S is a structure and $\overline{P}$ is an object of S, then the structure of $\overline{P}$, which will be denoted by $\sigma_S(\overline{P})$, is the structure T defined by:
>
> $$P_T = \{Q \in A^* \mid PQ \in P_S\},$$
>
> $$Q \equiv_T R \Leftrightarrow PQ \equiv_S PR \quad (Q,R \in P_T).$$

The proof that T is indeed a structure and that T is independent of the choice of P is simple. Two different objects can have the same structure (see Example 2.6). Consequently, they can be viewed as instances of that structure.

EXAMPLE 2.6

Consider the structure S of Figure 2.6.

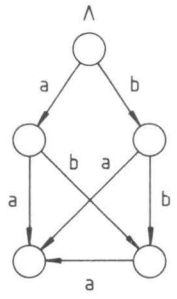


Figure 2.6

In this structure we have (using regular expression notation):

$$\overline{\Lambda} = \Lambda,$$
$$\overline{a} = a,$$
$$\overline{b} = b,$$
$$\overline{aa} = aa + aba + ba + bba,$$
$$\overline{bb} = ab + bb.$$

The structure of $\overline{a}$ is:

$$\sigma_S(\overline{a}) = \langle P_0, \equiv_0 \rangle,$$

where

$$P_0 = \{Q \in A^* \mid aQ \in P_S\} = \{\Lambda, a, ba, b\},$$

$$Q \equiv_0 R \Leftrightarrow aQ \equiv_S aR \quad (Q,R \in P_0),$$

hence $P_0 | \equiv_0 = \{\{\Lambda\}, \{a, ba\}, \{b\}\}$.

The structure of $\overline{b}$ is:

$$\sigma_S(\overline{b}) = \langle P_1, \equiv_1 \rangle,$$

where

$$P_1 = \{Q \in A^* \mid bQ \in P_S\} = \{\Lambda, a, ba, b\},$$

$$Q \equiv_1 R \Leftrightarrow bQ \equiv_1 bR \quad (Q, R \in P_1),$$

hence $P_1 | \equiv_1 = \{\{\Lambda\}, \{a, ba\}, \{b\}\}$.

So $\overline{a}$ and $\overline{b}$ have the same structure (the structure of Figure 2.4). □

EXAMPLE 2.7

Consider the structure S of Figure 2.7.

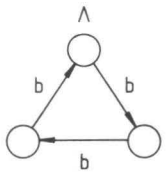


Figure 2.7

All objects have the same structure:

$$\sigma_S(\overline{\Lambda}) = \sigma_S(\overline{b}) = \sigma_S(\overline{bb}) = S.$$

□

2.2. OPERATIONS ON STRUCTURES

In Section 2.3 a simple language for the specification of operations on structures and their objects will be introduced. The meaning of these specifications will be described in terms of three primitive operations, which will be defined in this section. Apart from these three primitive operations the general concept of an operation as it will occur in the specification of an algorithm or data structure will also be discussed.

2.2.1. A partial order on structures

First, a special partial order on the set S of all structures will be introduced. This partial order will be used in the definition of the three primitive operations on structures.

The partial order $\sqsubset$ on S is defined by:

$$S \sqsubset T \Leftrightarrow P_S \subset P_T \wedge \equiv_S \subset \equiv_T \quad (S,T \in S).$$

Here "$\equiv_S \subset \equiv_T$" means that $P \equiv_S Q$ implies $P \equiv_T Q$ for each $P,Q \in P_S$. The fact that $\sqsubset$ is indeed a (reflexive!) partial order on S is trivial. In intuitive terms the fact that $S \sqsubset T$ means that all paths of S are also paths of T and that all paths which are "identified" in S are also identified in T.

EXAMPLE 2.8

The structures of Figure 2.8 form an ascending sequence:

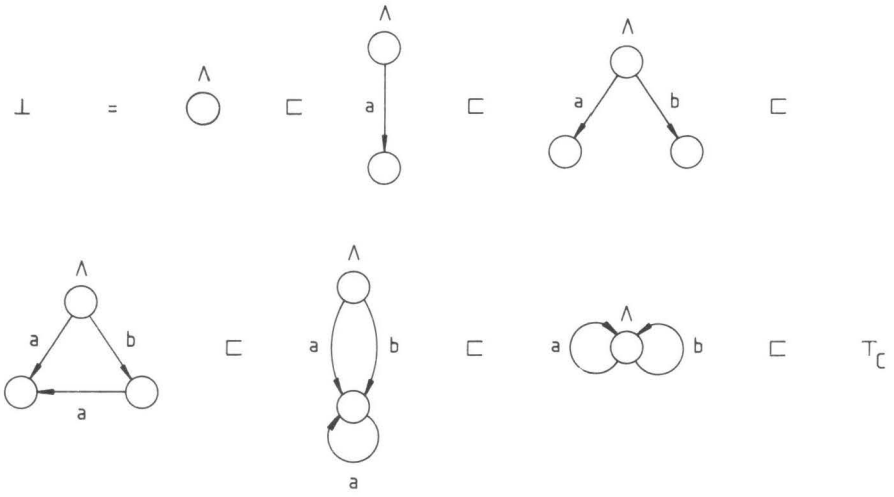


Figure 2.8

□

EXAMPLE 2.9

If we define the partial order $\sqsubset_0$ on S by:

$$S \sqsubset_0 T \Leftrightarrow S \sqsubset T \wedge P_S = P_T \quad (S,T \in S),$$

then the fact that $S \sqsubset_0 T$ means that S is a "partial expansion" of T, as illustrated by Figure 2.9.

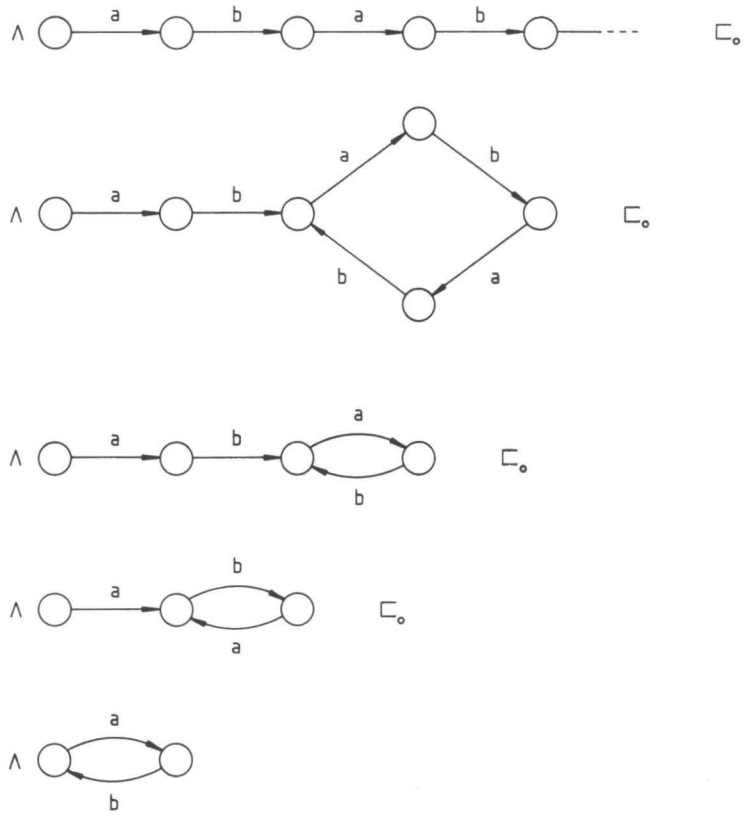


Figure 2.9

□

Notice that the partial orders $\sqsubset$ and $\sqsubset_0$ are much harder to describe in terms of graphs than in terms of structures.

2.2.2. The lattice property

The relation $\sqsubset$ is more than just a partial order: It turns S into a complete lattice. (A complete lattice is a partially ordered set where each subset has an infimum.) This is stated in:

LEMMA 2.2

$\langle S, \sqsubset \rangle$ is a complete lattice.

PROOF

Let $T \subset S$. The infimum V of T is given by:

$$V = \begin{cases} \langle \cap\, T \in T\ [P_T], \cap\, T \in T\ [\equiv_T] \rangle & \text{if } T \neq \emptyset, \\ \tau_C & \text{if } T = \emptyset. \end{cases}$$

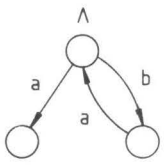
The proof that V is indeed the infimum of T is simple and is left to the reader. (First prove that V is a structure; the rest of the proof is trivial.) □

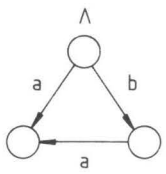
Notice that the empty structure $\perp$ and the convergent structure τ_C are the "bottom" and "top" of the complete lattice $\langle S, \sqsubset \rangle$, i.e. $\perp \sqsubset S \sqsubset \tau_C$ for each $S \in S$. A simple theorem from lattice theory states that apart from an infimum each subset also has a supremum [BIRKHOFF 67]. The following definitions are therefore in order:

> For each set T of structures the structures *inf* T and *sup* T are defined by:
>
> *inf* T = infimum of T with respect to $\sqsubset$,
>
> *sup* T = supremum of T with respect to $\sqsubset$.

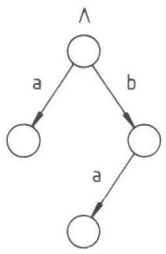
The above will enable us to define the result of operations on structures in terms of *inf*'s and *sup*'s of arbitrary sets of structures without having to worry about the existence of the *inf*'s and *sup*'s.

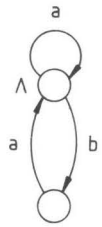
EXAMPLE 2.10

If S = 

and T = 

,

then *inf*$\{S,T\}$ = 

and *sup*$\{S,T\}$ = 

.

Figure 2.10

Check that this is indeed so! □

2.2.3. Some other partial orders

Before defining the primitive operations on structures a remark should be made about some other interesting partial orders on S. The definition of $\sqsubset$ can be written as:

$$S \sqsubset T \Leftrightarrow P_S \subset P_T \wedge \forall P,Q \in P_S\ [P \equiv_S Q \Rightarrow P \equiv_T Q] \quad (S,T \in S).$$

If we reverse the implication sign in this definition we still have a partial order, call it $\sqsubset_1$:

$$S \sqsubset_1 T \Leftrightarrow P_S \subset P_T \wedge \forall P,Q \in P_S\ [P \equiv_T Q \Rightarrow P \equiv_S Q] \quad (S,T \in S).$$

Intuitively $S \sqsubset_1 T$ means that all paths of S are also paths of T and that all paths which are "distinguished" in S are also distinguished in T. The partial order $\sqsubset_1$ has both a bottom (the empty structure $\perp$) and a top (the divergent structure τ_D). Yet, in contrast to $\sqsubset$, it does not turn S into a complete lattice (see Example 2.11).

EXAMPLE 2.11

Consider the structures in Figure 2.11.

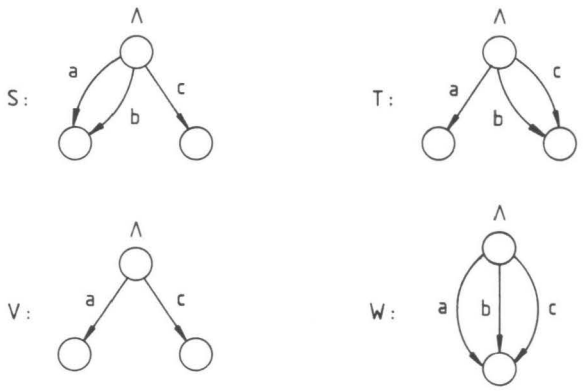


Figure 2.11

Suppose S and T have an infimum X with respect to $\sqsubset_1$. Since $V \sqsubset_1 S$ and $V \sqsubset_1 T$, we have that $V \sqsubset_1 X$. This implies that $a,c \in P_X$ and, since $a \not\equiv_V c$, also that $a \not\equiv_X c$. $W \sqsubset_1 S$ and $W \sqsubset_1 T$ imply that $W \sqsubset_1 X$, hence $b \in P_X$. $X \sqsubset_1 S$ and $a \equiv_S b$ imply that $a \equiv_X b$. Analogously, $X \sqsubset_1 T$ and $b \equiv_T c$ imply that $b \equiv_X c$. Using the transitivity of $\equiv_X$ we get $a \equiv_X c$, which is in contradiction with the fact that $a \not\equiv_X c$. Hence $\langle S,\sqsubset_1 \rangle$ is not a complete lattice. □

Another partial order of interest, call it $\sqsubset_2$, is obtained by taking the intersection of $\sqsubset$ and $\sqsubset_1$:

$$S \sqsubset_2 T \Leftrightarrow P_S \subset P_T \wedge \forall P,Q \in P_S\ [P \equiv_S Q \Leftrightarrow P \equiv_T Q] \quad (S,T \in S).$$

It is easy to see that $<S,\sqsubset_2>$ is not a complete lattice either (there is not even a greatest element).

All operations which will be introduced below are considered as partial operators on structures. They may have a number of parameters (usually objects in the structure to which they are applied, or accessors). The result of applying the operation F with parameters $X_1,\ldots,X_m$ to the structure S will be denoted by $\{S\}F(X_1,\ldots,X_m)$. The notation $F(X_1,\ldots,X_m)$ will be used to denote the (partial) operator $\lambda\ S \in S\ [\{S\}F(X_1,\ldots,X_m)]$. Concatenation will be used to denote functional composition of operators. For example, $F(X_1,\ldots,X_m)G(Y_1,\ldots,Y_n)$ denotes the operator $\lambda\ S \in S\ [\{\{S\}F(X_1,\ldots,X_m)\}G(Y_1,\ldots,Y_n)]$. This implies that composite operators can be read from left to right, which enhances readability to a great extent.

2.2.4. The operation *CRE*

The first primitive operation on structures which will be introduced amounts to the "creation" of an object in a structure. The operation, called *CRE*, has two parameters $\overline{P}$ and A. $\overline{P}$ is an object in the structure S to which *CRE* is applied and A is an accessor such that PA is not a path of S. The effect of $CRE(\overline{P},A)$ is that a new object with $\perp$ as its structure is added as a direct A-component to $\overline{P}$. This is pictured in Figure 2.12.

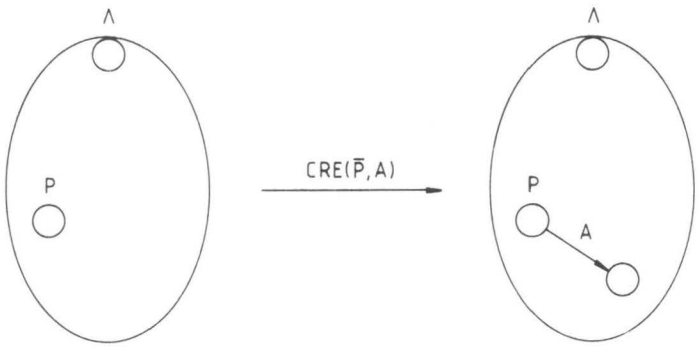


Figure 2.12

The definition of *CRE* reads:

> Let S be a structure. If $\overline{P}$ is an object of S and $A \in \mathbf{A}$ such that $PA \notin P_S$, then $\{S\}CRE(\overline{P},A)$ is the following structure:
>
> $$inf\{T \in S \mid S \sqsubset T,\ \forall R \in P_S\ [R \equiv_S P \Rightarrow RA \in P_T]\}.$$

The fact that "less" in the partial order $\sqsubset$ implies "less identification" guarantees that a new object is created and not some old object is taken as the new component of $\overline{P}$.

Though the above definition of *CRE* is intuitively clear, one may still wonder whether it really defines the operation pictured in Figure 2.12. In order to show that this is indeed so, we shall give a different characterization of the structure $\{S\}CRE(\overline{P},A)$, which more closely fits in with Figure 2.12.

LEMMA 2.3

Let S be a structure, $\overline{P}$ an object of S and $A \in \mathsf{A}$ such that $PA \notin P_S$. A characterization of $V = \{S\}CRE(\overline{P},A)$ is given by:

(V1) $P_V = P_S \cup Q$.
(V2) $\equiv_V = \equiv_S \cup \{(X,Y) \mid X,Y \in Q\}$.

where $Q = \{RA \mid R \in P_S,\ R \equiv_S P\}$.

PROOF

Let $V = \langle P_V, \equiv_V \rangle$ as defined by (V1) and (V2), and let $W = \{S\}CRE(\overline{P},A)$. We first have to prove that V is a structure, which is left to the reader. If we define:

$$T = \{T \in S \mid S \sqsubset T,\ \forall R \in P_S\ [R \equiv_S P \Rightarrow RA \in P_T]\},$$

then $W = \inf T$. We shall show that $V \in T$ and $V \sqsubset T$ $(T \in T)$, or in other words that $V = \min T$ (the minimum of T) and consequently $V = W$. The fact that $V \in T$ is obvious. Now let $T \in T$, then

(T1) $S \sqsubset T$.
(T2) $\forall R \in P_S\ [R \equiv_S P \Rightarrow RA \in P_T]$.

In order to prove that $V \sqsubset T$, we have to show that $P_V \subset P_T$ and $\equiv_V \subset \equiv_T$.

(1) Proof of $P_V \subset P_T$:
Let $X \in P_V$, then either $X \in P_S$ or $X = RA$ with $R \in P_S$ and $R \equiv_S P$. In the first case (T1) implies that $X \in P_T$ and in the second case (T2) implies that $X \in P_T$.
(2) Proof of $\equiv_V \subset \equiv_T$:
Let $X,Y \in P_V$ with $X \equiv_V Y$. There are two cases. The first is that $X,Y \in P_S$ and $X \equiv_S Y$. (T1) then implies that $X \equiv_T Y$. The second case is that $X = R_X A$ and $Y = R_Y A$ with $R_X, R_Y \in P_S$ and $R_X \equiv_S R_Y \equiv_S P$. (T1) implies that $R_X, R_Y \in P_T$ and $R_X \equiv_T R_Y$. (T2) implies that $R_X A, R_Y A \in P_T$. Consequently $R_X A \equiv_T R_Y A$, which is the same as $X \equiv_T Y$.

□

Notice that in contrast to graph models of storage structures creation is a natural operation in structures. (In graphs nodes are usually "created" by choosing them from a set of already existing "free" nodes.)

EXAMPLE 2.12

A binary tree can be generated from the empty structure by a sequence of operations such as:

$$\{\bot\}CRE(\overline{\Lambda},a)CRE(\overline{\Lambda},b)CRE(\overline{b},a)CRE(\overline{ba},a)CRE(\overline{ba},b).$$

The intermediate and final results of this sequence of operations are pictured in Figure 2.13.

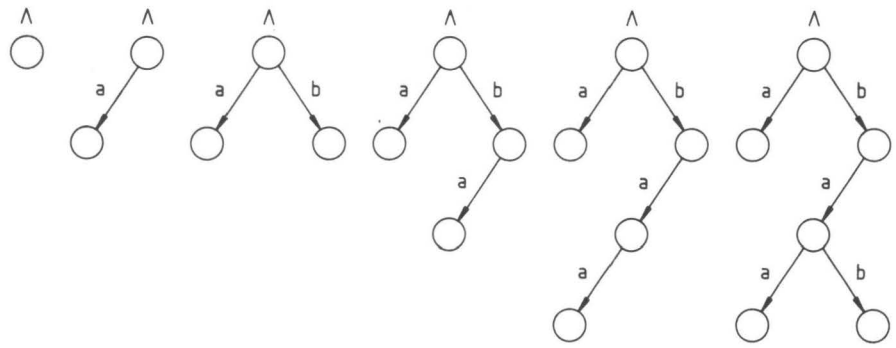


Figure 2.13

□

2.2.5. The operation *ADD*

The second primitive operation on structures is like *CRE*, except that it adds an already existing object as a direct component to an object. The operation, called *ADD*, takes three parameters $\overline{P}$, A and $\overline{Q}$. $\overline{P}$ and $\overline{Q}$ are objects in the structure S to which *ADD* is applied and A is an accessor such that PA is not a path of S. The effect of $ADD(\overline{P},A,\overline{Q})$ is that $\overline{Q}$ is added as a direct A-component to $\overline{P}$, which is pictured in Figure 2.14.

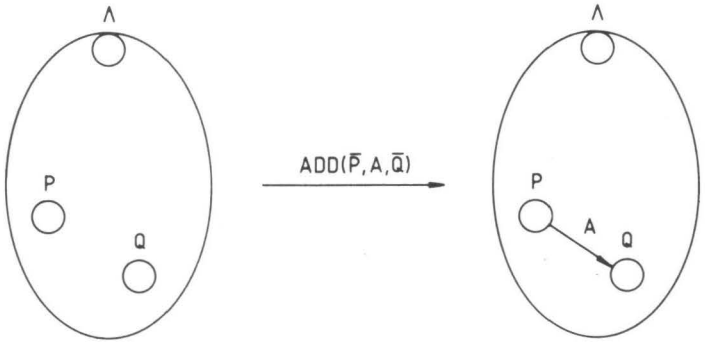


Figure 2.14

The definition of *ADD* is given below.

> Let S be a structure. If $\overline{P}$ and $\overline{Q}$ are objects of S and $A \in \mathbb{A}$ such that $PA \notin P_S$, then $\{S\}ADD(\overline{P},A,\overline{Q})$ is the following structure:
>
> $$inf\{T \in S \mid S \sqsubset T,\ \forall R \in P_S\ [R \equiv_S P \Rightarrow RA \in P_T \wedge RA \equiv_T Q]\}.$$

The infimum of the same set of structures as in the definition of *CRE* is taken here, except that the set is restricted to those structures in which the paths RA with $R \equiv_S P$ and Q are identified. This guarantees that no new object is created, but that $\overline{Q}$ is added as a new component to $\overline{P}$.

Like we did for $\{S\}CRE(\overline{P},A)$, we shall give another characterization of the structure $\{S\}ADD(\overline{P},A,\overline{Q})$, so as to strengthen our faith that *ADD* does what Figure 2.14 suggests. The characterization of $\{S\}ADD(\overline{P},A,\overline{Q})$ will not be as simple as that of $\{S\}CRE(\overline{P},A)$, which is due to the fact that *ADD* may introduce circularities in a structure. These circularities cause an "explosion" of the number of paths (see Example 2.13).

> LEMMA 2.4
>
> Let S be a structure, $\overline{P}$ and $\overline{Q}$ objects of S and $A \in \mathbb{A}$ such that $PA \notin P_S$. A characterization of $V = \{S\}ADD(\overline{P},A,\overline{Q})$ is given by:
>
> (V1) $P_V = P_S \cup Q$.
> (V2) $\equiv_V = \{(X,Y) \in P_V \times P_V \mid \Phi(X) \equiv_S \Phi(Y)\}$.
>
> where $Q = \{RAX_1A\ldots X_nAY \mid R \in P_S,\ R \equiv_S P,\ QY \in P_S,$
> $QX_i \in P_S,\ QX_i \equiv_S P\ (i = 1,\ldots,n),\ n \geq 0\}$
>
> and Φ: $P_V \to P_S$ is defined by:
>
> $$\Phi(X) = \begin{cases} X & \text{for } X \in P_S, \\ QY & \text{for } X \in Q \text{ with } X = RAX_1A\ldots X_nAY \text{ as above.} \end{cases}$$

PROOF

Let $V = \langle P_V, \equiv_V \rangle$ as defined by (V1) and (V2), and let $W = \{S\}ADD(\overline{P},A,\overline{Q})$. Check that the mapping Φ is well-defined and prove that V is a structure. Now let

$$\mathcal{T} = \{T \in S \mid S \sqsubset T,\ \forall R \in P_S\ [R \equiv_S P \Rightarrow RA \in P_T \wedge RA \equiv_T Q]\}.$$

We shall show that $V = min\ \mathcal{T}$, which implies that $V = W$. First, the fact that $V \in \mathcal{T}$ is obvious. Secondly, let $T \in \mathcal{T}$, then we have:

(T1) $S \sqsubset T$.
(T2) $\forall R \in P_S\ [R \equiv_S P \Rightarrow RA \in P_T \wedge RA \equiv_T Q]$.

Check that in order to prove that $V \sqsubset T$ it is sufficient to prove the following assertion:

$$\forall X \in P_V\ [X \in P_T \wedge X \equiv_T \Phi(X)].$$

We shall prove this assertion now. Let $X \in P_V$. Two cases can be distinguished.

(1) $X \in P_S$.
(T1) implies that $X \in P_T$, and $X \equiv_T \Phi(X)$ since $\Phi(X) = X$.
(2) $X \in Q$.
We know that $X = RAX_1A...X_nAY$ with $R \in P_S$, $R \equiv_S P$, $QY \in P_S$, $QX_i \in P_S$, $QX_i \equiv_S P$ $(i = 1,...,n)$ and $n \geq 0$. Using induction and (T2) we can prove that:

$$\left.\begin{array}{l} RAX_1A...X_iA \in P_T \\ RAX_1A...X_iA \equiv_T Q \end{array}\right\} \quad (i = 0,...,n).$$

Together with the fact that $QY \in P_T$ (since $QY \in P_S$) the above implies that:

$$RAX_1A...X_nAY \in P_T,$$
$$RAX_1A...X_nAY \equiv_T QY.$$

Or, in other words, $X \in P_T$ and $X \equiv_T \Phi(X)$.

□

EXAMPLE 2.13

Let S be the structure of Figure 2.15,

Figure 2.15

then $T = \{S\}ADD(\overline{b},a,\overline{\Lambda})$ is the structure of Figure 2.16.

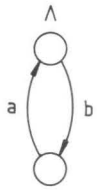


Figure 2.16

Notice that $ADD(\overline{b},a,\overline{\Lambda})$ has turned the finite number of paths in S ($P_S = \{\Lambda,b\}$) into an infinite number in T ($P_T = \{\Lambda,b,ba,bab,baba,...\}$). □

2.2.6. The operation *REM*

The third and final primitive operation can be viewed somehow as the (right) inverse of the other two primitive operations. It amounts to removing a direct component of an object. The operation, called *REM*, has two parameters $\overline{P}$ and A. $\overline{P}$ is an object in the structure S to which *REM* is applied and A is an accessor such that PA is a path of S. The effect of $REM(\overline{P},A)$ is that the direct A-component of $\overline{P}$ is removed from $\overline{P}$, as pictured in Figure 2.17.

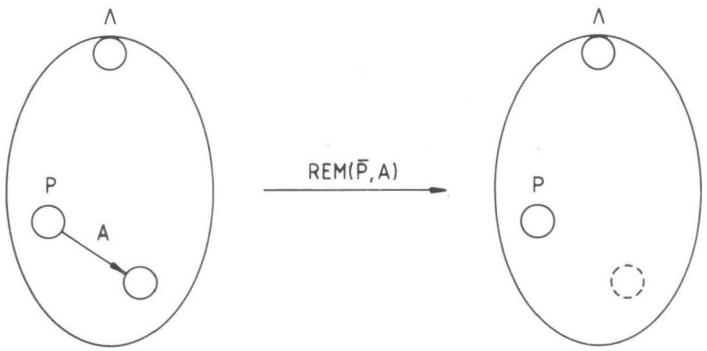


Figure 2.17

The definition of *REM* is:

> Let S be a structure. If $\overline{P}$ is an object of S and $A \in \mathsf{A}$ such that $PA \in \mathcal{P}_S$, then $\{S\}REM(\overline{P},A)$ is the following structure:
>
> $$sup\{T \in \mathcal{S} \mid T \sqsubset S,\ \forall R \in \mathcal{P}_S\ [R \equiv_S P \Rightarrow RA \notin \mathcal{P}_T]\}.$$

Notice that, due to the fact that objects may be shared, $REM(\overline{P},A)$ need not remove the object $\overline{PA}$ from S. That is why this object is represented by a dotted circle in the right part of Figure 2.17. (Strictly speaking the path name P should also be dotted, because the path P (but not the object $\overline{P}$) may be removed from S by $REM(\overline{P},A)$.) In general, $REM(\overline{P},A)$ may reduce the number of objects in a structure by a number varying from zero to all but one (see Example 2.14).

Like before we shall give another characterization of the structure $\{S\}REM(\overline{P},A)$ in order to show that the definition given is in conformity with Figure 2.17.

> LEMMA 2.5
>
> Let S be a structure, $\overline{P}$ an object of S and $A \in \mathsf{A}$ such that $PA \in \mathcal{P}_S$. A characterization of $V = \{S\}REM(\overline{P},A)$ is given by:
>
> (V1) $\mathcal{P}_V = \mathcal{P}_S \setminus \mathcal{Q}$.
> (V2) $\equiv_V = \{(X,Y) \in \mathcal{P}_V \times \mathcal{P}_V \mid X \equiv_S Y\}$.
>
> where $\mathcal{Q} = \{RAX \mid R \in \mathcal{P}_S,\ R \equiv_S P,\ RAX \in \mathcal{P}_S\}$.

PROOF

Let $V = \langle P_V, \equiv_V \rangle$ as defined by (V1) and (V2), and let $W = \{S\}REM(\overline{P},A)$. Prove that V is a structure. Define:

$$T = \{T \in S \mid T \sqsubset S,\ \forall R \in P_S\ [R \equiv_S P \Rightarrow RA \notin P_T]\},$$

then $W = sup\ T$. We shall show that $V = max\ T$ (the maximum of T) and consequently $V = W$. We have to prove that $V \in T$ and $T \sqsubset V$ $(T \in T)$. The fact that $V \in T$ is obvious. Now let $T \in T$, then

(T1) $T \sqsubset S$.
(T2) $\forall R \in P_S\ [R \equiv_S P \Rightarrow RA \notin P_T]$.

The proof of $T \sqsubset V$ falls apart in:

(1) Proof of $P_T \subset P_V$:
Let $X \in P_T$ and suppose that $X \notin P_V$. Since (T1) implies that $X \in P_S$, there must exist an $R \in P_S$, $R \equiv_S P$ and $Y \in A^*$ such that $X = RAY$. (T2) implies that $RA \notin P_T$, hence also $RAY \notin P_T$ and $X \notin P_T$. From this contradiction can be inferred that $X \in P_V$.
(2) Proof of $\equiv_T \subset \equiv_V$:
Let $X,Y \in P_T$, $X \equiv_T Y$, then (T1) implies that $X \equiv_S Y$. Since $X,Y \in P_V$ (see (1)), this implies that also $X \equiv_V Y$.

□

EXAMPLE 2.14

Consider the structure S of Figure 2.18.

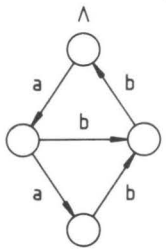


Figure 2.18

The effect of $REM(\overline{a},b)$ is:

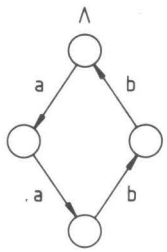


Figure 2.19

Notice: the number of objects has not changed. If $REM(\overline{\Lambda},a)$ is applied subsequently to the structure of Figure 2.19, we get:

Figure 2.20

Notice: all objects but one have "vanished". □

Check that the operations *CRE* and *ADD* could have been defined as follows:

$$\{S\}CRE(\overline{P},A) = inf\{T \in S \mid S \sqsubset T,\ PA \in P_T\},$$

$$\{S\}ADD(\overline{P},A,\overline{Q}) = inf\{T \in S \mid S \sqsubset T,\ PA \in P_T,\ PA \equiv_T Q\},$$

but that the following definition of *REM* would not have been correct:

$$\{S\}REM(\overline{P},A) = sup\{T \in S \mid T \sqsubset S,\ PA \notin P_T\}.$$

The three primitive operations introduced in the foregoing are sufficient to define the semantics of the language which will be introduced in Section 2.3.

2.2.7. The general concept of an operation

The last part of this section will be devoted to the general concept of an operation as it will be used in specifications of algorithms and data structures. An operation F will be considered to operate on a number of objects $\overline{P}_1,\ldots,\overline{P}_n$ (the "actual parameters") in an environment $\overline{E}$ (which is supposed to contain all information "global" to F). Both $\overline{E}$ and $\overline{P}_1,\ldots,\overline{P}_n$ are objects in an "embedding" structure S. (They cannot simply be considered as structures, because overlapping would then be impossible. Also, $\overline{E}$ cannot be identified with the embedding structure S, because

$\overline{P}_1,\ldots,\overline{P}_n$ need not be contained in $\overline{E}$.) The result of applying F is a value $\overline{V}$ (the "result value") and a new environment $\overline{G}$ (which may differ from the old because of "side effects" of F). $\overline{V}$ and $\overline{G}$ are objects in an embedding structure T. (The same remarks made about $\overline{E}$ and $\overline{P}_1,\ldots,\overline{P}_n$ apply to $\overline{G}$ and $\overline{V}$.) F could therefore be considered as a function which maps tuples of the type $\langle S,\overline{E},\overline{P}_1,\ldots,\overline{P}_n\rangle$ to tuples of the type $\langle T,\overline{G},\overline{V}\rangle$. This definition is not very convenient. Moreover, S and T may contain a lot of "garbage" (i.e., objects which are not contained in $\overline{E}$ or $\overline{P}_1,\ldots,\overline{P}_n$ and $\overline{G}$ or $\overline{V}$ respectively). F will therefore be considered somewhat differently as a mapping from structures of a special kind to structures of a special kind. Assume F has been specified with the formal parameters $X_1,\ldots,X_n$ (which correspond to the actual parameters $\overline{P}_1,\ldots,\overline{P}_n$ above). These formal parameters will be considered as accessors. F can now be considered as a mapping from structures to structures as indicated in Figure 2.21. The correspondence with the objects $\overline{E}$, $\overline{P}_1,\ldots,\overline{P}_n$, $\overline{G}$ and $\overline{V}$ above is indicated in the figure.

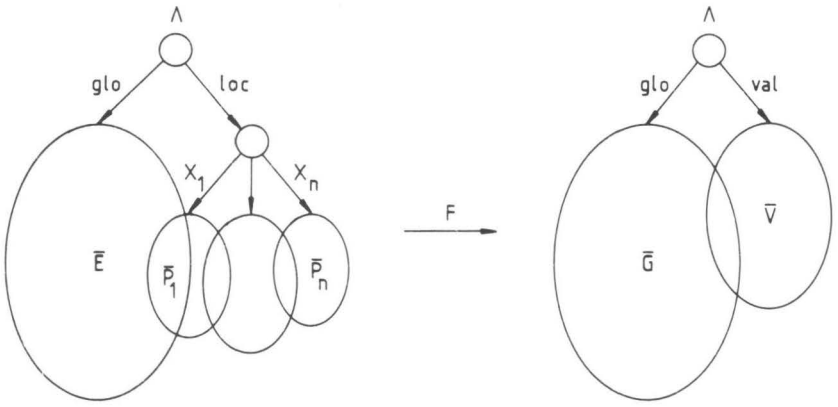


Figure 2.21

The left structure in Figure 2.21 will be called a "state". A state S is composed of two objects: the "global state" $\overline{glo}$, which corresponds to the environment prior to the application of F, and the "local state" $\overline{loc}$, which contains the actual parameters of F. Note that the formal parameters $X_1,\ldots,X_n$ of F are represented in S by the paths $loc.X_1,\ldots,loc.X_n$. (Since we have now begun to use accessors composed of more than one letter, dots will be used in path names to separate accessors if necessary.) The right structure in Figure 2.21 will be called a "result". A result R is also composed of two objects: the "global state" $\overline{glo}$, corresponding to the new environment after application of F, and the "value" $\overline{val}$, which represents the result value of F. This is all more precisely described below.

A state over a set B of accessors is a structure S such that:
(1) $A \cap P_S = \{glo,loc\}$.
(2) $\{A \in A \mid loc.A \in P_S\} = B$.

A result is a structure S such that $A \cap P_S = \{glo,val\}$.

An operation is a mapping F from a set of states over a set $\mathcal{B}_F$ of accessors into the set of results.

The elements of $\mathcal{B}_F$ are called the formal parameters of F.

The above definition of the concept of an operation has the disadvantage that it makes the names of the formal parameters part of the operation. On the other hand, if we had used the traditional concept of an operation (= n-ary function) we would have had to commit ourselves to an order for the formal parameters.

2.2.8. Side effect free, environment independent and static operations

A number of important concepts in connection with operations can now readily be defined. Let us first consider the effect which an operation F may have on the environment, i.e. on the component $\overline{glo}$ of the state S to which F is applied. F will be called "side effect free" if the structure of $\overline{glo}$ in S is the same as the structure of $\overline{glo}$ in $\{S\}F$. (Notice that this does not mean that each object $\overline{P}$ of S contained in $\overline{glo}$ is the same (set of paths) as the object $\overline{P}$ of $\{S\}F$. It does mean that the structure of $\overline{P}$ in S is the same as the structure of $\overline{P}$ in $\{S\}F$, though.) "Side effect freeness" is a useful property of operations, but imposing it as a requirement on all operations is too restrictive for our purposes. Instead, operations will be forbidden to access objects other than those which are passed as actual parameters (no "sneak access"). Side effects can only result then from the fact that the actual parameters share components with the environment.

Next consider the dependence of the result value of F on the environment and the actual parameters. F will be called "environment independent" if the value of F depends only on the structure of $\overline{loc}$ and not on the structure of $\overline{glo}$. The fact that F is environment independent does not mean that the value of F depends only on the structure of its actual parameters. The value may still depend on the way the actual parameters overlap. (This overlap is preserved in the structure of $\overline{loc}$.) If the value of F depends only on the structure of its actual parameters, F will be called "static". Being static is a stronger property than being environment independent (each static operation is environment independent). All operations which will be encountered in the sequel will be environment independent, but not necessarily static. Operations pertaining to so called "static data structures" (such as the integers) will all be static and side effect free. The various concepts are precisely defined below.

Let F be an operation with formal parameters $X_1,\ldots,X_n$.

F is side effect free iff for each $S \in dom(F)$:

$$\sigma_S(\overline{glo}) = \sigma_{\{S\}F}(\overline{glo}).$$

F is environment independent iff for each $S,T \in dom(F)$:

$$\sigma_S(\overline{loc}) = \sigma_T(\overline{loc}) \Rightarrow \sigma_{\{S\}F}(\overline{val}) = \sigma_{\{T\}F}(\overline{val}).$$

F is static iff for each $S,T \in dom(F)$:

$$\sigma_S(\overline{loc.X_i}) = \sigma_T(\overline{loc.X_i}) \ (i = 1,\ldots,n) \Rightarrow \sigma_{\{S\}F}(\overline{val}) = \sigma_{\{T\}F}(\overline{val}).$$

2.3. A LANGUAGE FOR MANIPULATING STRUCTURES

2.3.1. Syntax

The language which will be defined in this section will allow for the specification of operations in a typeless way. A program in this language is supposed to be nothing but a collection of "definitions", where each definition specifies an operation. The syntax of the language is as follows:

```
<definition>
    └→ <opname> <parameter list> = <body>
<opname>
    └→ <accessor>
<parameter list>
    ├→ ()
    └→ (<accessor> {, <accessor>})
<body>
    └→ <statement list> return <construct>
<statement list>
    ├→ <empty>
    └→ <statement> {; <statement>}
<empty>
    └→
<statement>
    ├→ <simple statement>
    ├→ <conditional statement>
    └→ <repetitive statement>
<simple statement>
    ├→ <declaration>
    └→ <assignment>
<declaration>
    └→ let <accessor> := <construct>
<assignment>
    └→ <path> := <construct>
<path>
    └→ {<accessor> .} <accessor>
<conditional statement>
    └→ if <assertion> then <range> else <range> fi
<repetitive statement>
    └→ while <assertion> do <range> od
<assertion>
    └→ <path> ≡ <path>
<range>
    └→ <statement list>
<construct>
    ├→ <selection>
    ├→ <creation>
    └→ <application>
<selection>
    └→ <path>
<creation>
    ├→ ()
    └→ (<binding> {, <binding>})
<binding>
    └→ <accessor> → <path>
```

<application>
    ⊢→ <opname> ()
    └→ <opname> (<path> {, <path>})

The curly braces "{" and "}" denote zero or more instances of the enclosed syntactical constructs. The usual (ALGOL-like) context-sensitive rules hold in this language, except that identifiers may not be redeclared, nor used before their declaration (as follows from the rules below).

2.3.2. Semantics

In order to define the semantics of the language described above, meaning functions will be associated with the respective syntactical constructs. The names of these functions are given below:

$M_F(F)$: for each *opname* F,
$M_B(D)$: for each *body* D,
$M_S(S)$: for each *statement* S,
$M_R(R)$: for each *range* R,
$M_C(C)$: for each *construct* C,
$M_A(B)$: for each *assertion* B.

$M_F(F)$ and $M_B(D)$ are operations in the sense defined in Section 2.2. $M_S(S)$, $M_R(R)$ and $M_C(C)$ are mappings from states to special structures as indicated in Figure 2.22. $M_A(B)$ is a mapping from states into the set of truth values {*true*,*false*}. (N.B. The term "mapping" is always used in the sense of a partial mapping.)

Before defining the meaning functions two definitions will be introduced.

The identity mapping J is defined by:

$$J = \lambda\ S \in S\ [S].$$

If B is a mapping from S into the set of truth values {*true*,*false*}, and F_1 and F_2 are mappings from S to S, then the mapping "if B then F_1 else F_2 fi" is defined by:

$$\text{if } B \text{ then } F_1 \text{ else } F_2 \text{ fi} = \lambda\ S \in S\ [\text{if } \{S\}B \text{ then } \{S\}F_1 \text{ else } \{S\}F_2 \text{ fi}].$$

The meaning functions are defined by the following rules:

RULE 1 (*opname*)

Let "F" be an *opname* with the following *definition*:

$$F(X_1,\ldots,X_n) = D,$$

where $X_1,\ldots,X_n$ are *accessors* ($n \geq 0$) and D is a *body*, then:

$$M_F(F) = M_B(D).$$

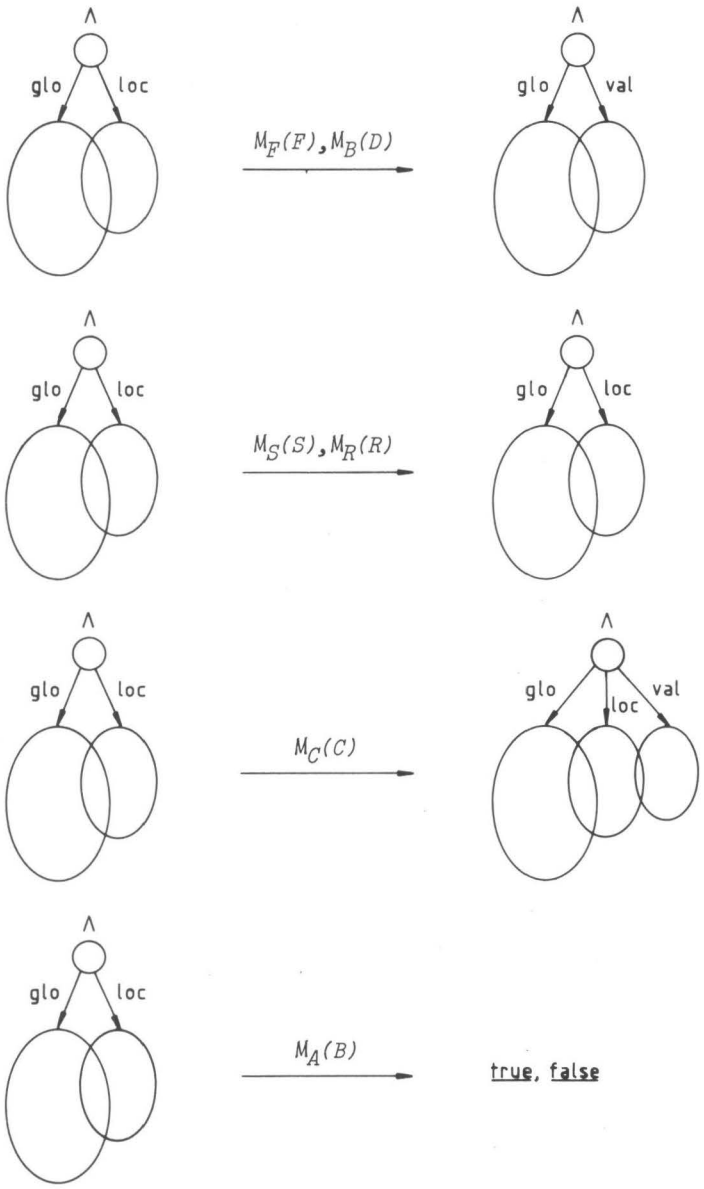

glo
loc
$M_F(F), M_B(D)$
glo
val
glo
loc
$M_S(S), M_R(R)$
glo
loc
glo
loc
val
$M_C(C)$
glo
loc
$M_A(B)$
true, false

Figure 2.22

RULE 2 (*body*)

Let "$S_1;\ldots;S_n$ return C" be a *body*, where $S_1,\ldots,S_n$ are *statements* ($n \geq 0$) and C is a *construct*, then:

$$M_B(S_1;\ldots;S_n \text{ return } C) = M_S(S_1)\ldots M_S(S_n)M_C(C)REM(\overline{\Lambda},loc).$$

RULE 3 (*declaration*)

Let "let A := C" be a *declaration*, where A is an *accessor* and C is a *construct*, then:

$$M_S(\text{let } A := C) = CRE(\overline{loc},A)M_S(A := C).$$

RULE 4 (*assignment*)

Let "PA := C" be an *assignment*, where PA is a *path*, A is an *accessor* and C is a *construct*, then:

$$M_S(PA := C) = M_C(C)ADD(\overline{\Lambda},p,\overline{loc.P})REM(\overline{p},A)ADD(\overline{p},A,\overline{val})$$
$$REM(\overline{\Lambda},p)REM(\overline{\Lambda},val).$$

(See remark below.)

RULE 5 (*conditional statement*)

Let "if B then R_1 else R_2 fi" be a *conditional statement*, where B is an *assertion* and R_1 and R_2 are *ranges*, then:

$$M_S(\text{if } B \text{ then } R_1 \text{ else } R_2 \text{ fi}) =$$
$$\text{if } M_A(B) \text{ then } M_R(R_1) \text{ else } M_R(R_2) \text{ fi}.$$

RULE 6 (*repetitive statement*)

Let "while B do R od" be a *repetitive statement*, where B is an *assertion* and R is a *range*, then:

$$M_S(\text{while } B \text{ do } R \text{ od}) =$$
$$\text{if } M_A(B) \text{ then } M_R(R)M_S(\text{while } B \text{ do } R \text{ od}) \text{ else } J \text{ fi}.$$

RULE 7 (*assertion*)

Let "$P \equiv Q$" be an *assertion*, where P and Q are *paths*, then:

$$M_A(P \equiv Q) = \lambda\ S \in S\ [\text{if } loc.P \equiv_S loc.Q \text{ then } true \text{ else } false \text{ fi}].$$

RULE 8 (*range*)

Let "$S_1;\ldots;S_n$" be a *range* ($n \geq 0$), where $S_1,\ldots,S_n$ are *statements*, and let $A_1,\ldots,A_m$ be the *accessors* following the "let" symbol of those S_i which are *declarations* ($i = 1,\ldots,n$), then:

$$M_R(S_1;\ldots;S_n) = M_S(S_1)\ldots M_S(S_n)REM(\overline{loc},A_1)\ldots REM(\overline{loc},A_m),$$

where $M_R(S_1;\ldots;S_n) = J$ if $n = 0$.

RULE 9 (*selection*)

Let "P" be a *selection*, where P is a *path*, then:

$$M_C(P) = ADD(\overline{\Lambda}, val, \overline{loc.P}).$$

RULE 10 (*creation*)

Let "$(A_1 \rightarrow P_1, \ldots, A_n \rightarrow P_n)$" be a *creation*, where $A_1, \ldots, A_n$ are *accessor*s and $P_1, \ldots, P_n$ are *path*s ($n \geq 0$), then:

$$M_C((A_1 \rightarrow P_1, \ldots, A_n \rightarrow P_n)) =$$
$$CRE(\overline{\Lambda}, val)ADD(\overline{val}, A_1, \overline{loc.P_1}) \ldots ADD(\overline{val}, A_n, \overline{loc.P_n}).$$

RULE 11 (*application*)

Let "$F(P_1, \ldots, P_n)$" be an *application*, where F is an *opname* and $P_1, \ldots, P_n$ are *path*s ($n \geq 0$). Let F be defined as in Rule 1, then:

$$M_C(F(P_1, \ldots, P_n)) = M_C((X_1 \rightarrow P_1, \ldots, X_n \rightarrow P_n))\ BEGIN\ M_F(F)\ END.$$

Here (see Figure 2.23):

$$BEGIN = CRE(\overline{\Lambda}, t)ADD(\overline{t}, glo, \overline{glo})REM(\overline{\Lambda}, glo)ADD(\overline{t}, loc, \overline{loc})REM(\overline{\Lambda}, loc)$$
$$ADD(\overline{\Lambda}, glo, \overline{t})REM(\overline{\Lambda}, t)ADD(\overline{\Lambda}, loc, \overline{val})REM(\overline{\Lambda}, val),$$

$$END = ADD(\overline{\Lambda}, t, \overline{glo})REM(\overline{\Lambda}, glo)ADD(\overline{\Lambda}, glo, \overline{t.glo})$$
$$ADD(\overline{\Lambda}, loc, \overline{t.loc})REM(\overline{\Lambda}, t).$$

Observe that the following definition of the meaning of the *assignment* "$PA := C$" in Rule 4 would not be correct:

$$M_S(PA := C) = M_C(C)REM(\overline{loc.P}, A)ADD(\overline{loc.P}, A, \overline{val})REM(\overline{\Lambda}, val).$$

The reason is that after $REM(\overline{loc.P}, A)$ the path $loc.P$ need no longer exist.

The above rules define the semantics of the language in an operational way. That is, basically they define an automaton which can compute values of the meaning functions. Due to the occurrence of recursion and loops the automaton need not terminate for certain arguments of a meaning function (see Rule 1 and Rule 6). For those arguments a meaning function is supposed to be undefined. With this convention it is easy to see that the above rules define the meaning functions completely.

Another approach would be to define the semantics of the language in a denotational way. The above rules are then used not to define an automaton, but a system of continuous operators on a function space (one operator for each recursive meaning function, i.e. each meaning function with a recursive equation). The recursive meaning functions are defined as the components of the simultaneous least fixed point of this system of continuous operators. The definition of the other (non-recursive) meaning functions follows directly from the rules. Due to the fact that this approach requires a rather elaborate mathematical apparatus and that we would still have to prove that the denotational semantics coincides with the operational semantics (since the latter is the more intuitive), we will not use the denotational approach. The construction of a denotational semantics for the language is believed not to pose any serious problems, however. For a thorough treatment of denotational semantics the reader is

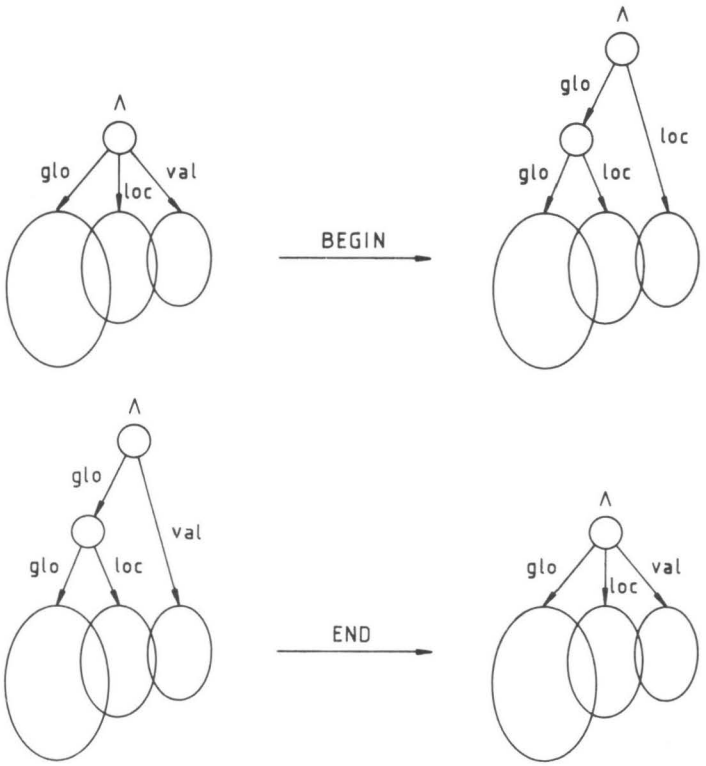


Figure 2.23

referred to [DE BAKKER 80].

2.3.3. Some remarks and examples

The language defined above is a rudimentary language in many respects. It is for example not possible to use applications as actual parameters of operations as in $F(G(x))$. Language extensions in which this is possible can readily be defined in terms of the above language, however. This also holds for the other extensions which we will discuss in the next section. The language can therefore be viewed as a kernel around which more sophisticated languages can be constructed (e.g., by "syntactic sugaring"). All necessary primitive algorithmic concepts (such as creation, selection, replacement, etc.) are included in the language.

The fact that the only test included in the language is a test for the identity of two objects may at first sight seem strange and even insufficient. It is for example not possible to test whether two arbitrary objects have the same structure. It is not even possible to test whether a given object has $\perp$ as its structure. Yet, the test for identity of objects is sufficient (at least for our purposes). Since all objects which are

manipulated by a program are constructed by that program, the programmer can take care that he always knows how to take an object apart into its direct components. The test whether two objects have the same structure can then be reduced to tests whether their direct components have the same structure. If the programmer takes care also that all objects are constructed from primitive objects, the structure of which can be compared using the identity relation for objects, then it is always possible to determine whether two objects have the same structure (see Examples 2.15 and 2.16).

EXAMPLE 2.15

We could represent the truth values *true* and *false* by the structures *true* and *false* pictured in Figure 2.24.

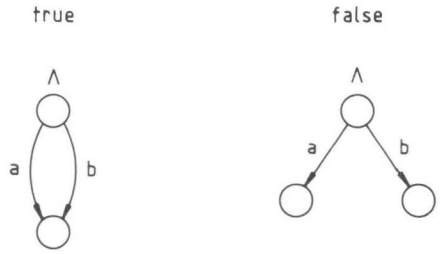


Figure 2.24

Instances of these structures can be constructed by the operations:

```
TRUE() = let e := ()
         return (a → e,b → e)

FALSE() = let e := ();
          let f := ()
          return (a → e,b → f)
```

Suppose we have an object denoted by path P in our program, which is known to have *true* or *false* as its structure. Whether the structure of P is *true* can now be determined through the test "$P.a \equiv P.b$". The operations *NOT(p)*, *AND(p,q)* and *OR(p,q)*, where p and q denote "boolean" objects, could for example be defined as follows:

```
NOT(p) = let r := ();
         if p.a ≡ p.b
            then r := FALSE()
            else r := TRUE()
         fi
         return r
```

```
AND(p,q) = let r := ();
           if p.a ≡ p.b
              then if q.a ≡ q.b
                      then r := TRUE()
                      else r := FALSE()
                   fi
              else r := FALSE()
           fi
           return r

OR(p,q) = let r := NOT(p);
          let s := NOT(q);
          let t := AND(r,s)
          return NOT(t)
```

□

EXAMPLE 2.16

Suppose we choose to represent the natural numbers as in Figure 2.25. (The structures *true* and *false* and the operations *TRUE* and *FALSE* are as in Example 2.15.)

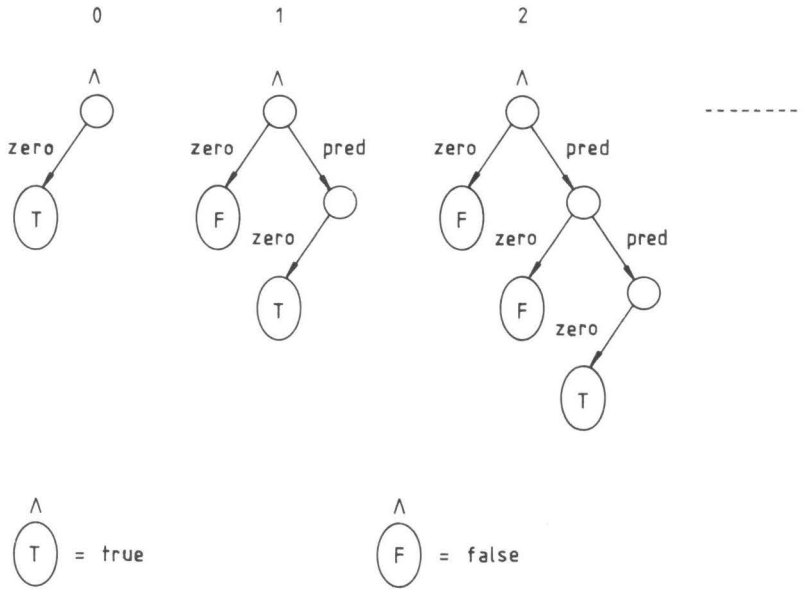


Figure 2.25

The following operations can then be defined:

```
ZERO() = let t := TRUE()
         return (zero → t)

SUCC(n) = let f := FALSE()
          return (zero → f,pred → n)

EQUAL(m,n) = let r := ();
             if m.zero.a ≡ m.zero.b
                then r := n.zero
                else if n.zero.a ≡ n.zero.b
                        then r := FALSE()
                        else r := EQUAL(m.pred,n.pred)
                     fi
             fi
             return r
```

□

In the language extension to be discussed in the next section the responsibility to "remember" how composite objects can be taken apart into their direct components will be moved entirely from the programmer to the language rules (by the introduction of a "strong typing" mechanism). More convenient kinds of assertions will also be introduced there.

2.4. TOWARDS A FULL-FLEDGED SPECIFICATION LANGUAGE

In this section we shall sketch through an example how the language described in Section 2.3 can be extended to a full-fledged specification language for algorithms and data structures. The central issue in this section will be a specification language for data structures. Since in this specification language operations of data structures will be expressed in terms of algorithms, the language could just as well be viewed as a specification language for algorithms, however.

2.4.1. The meaning of specifications of algorithms and data structures

Before presenting the example we shall discuss which formal meaning should be attached to specifications of algorithms and data structures. First, consider algorithms. There are basically two views of algorithms, which might be called the "functional" and the "procedural" view. Adopting the functional view implies that algorithms are considered as partial functions which map "states" to "results", i.e. as "operations" as defined in Section 2.2. Adopting the procedural view of algorithms implies that algorithms are considered as partial functions which map "states" to "states". For our purposes the procedural view of algorithms is the most convenient. We shall therefore adopt the latter. (Mark, however, that we maintain the functional view for operations of data structures.) The formal meaning of a specification of an algorithm will consequently be a partial function which maps "states" to "states". The procedural view of algorithms will be maintained throughout this monograph.

Next, consider data structures. It is generally agreed that data structures are fully characterized by their operations. The formal meaning of a specification of a data structure will therefore simply be a collection F of "operations", i.e. partial functions which map "states" to "results". In more algebraic terms this means that the formal meaning of a

specification of a data structure will be considered here as a homogeneous algebra <S,F>, where S is the set of all structures and F is a collection of operations. This is different from the more usual "heterogeneous approach" [BIRKHOFF & LIPSON 70], which is caused by the fact that we allow sharing. We shall show that, if all operations in F are side effect free and static (and hence environment independent), this algebra corresponds to a "normal" heterogeneous algebra with n-ary functions (which is the case covered by the algebraic specification methods).

2.4.2. An example: the data structure *Lisp*

In order to suit the previously defined language to a specification language it will be harnessed by imposing a type mechanism on it. This will be illustrated through the specification of the data structure *Lisp*, which includes the following well-known LISP operations [WEISSMAN 67]: *CONS*, *CAR*, *CDR*, *RPLACA*, *RPLACD*, *ATOM* and *EQ*. The natural numbers are used to represent "atomic values", which implies that the specification also includes the operations *ZERO*, *SUCC* and *EQUAL*, and an operation *EXP* for the conversion of a natural number into a "symbolic expression". The specification of *Lisp* is given below.

```
data structure Lisp

type Nat,Exp

representation

Nat = case zero: Bool of
      FALSE: (pred: Nat)
      esac

Exp = case atom: Bool of
      TRUE: (val: Nat)
      FALSE: (car,cdr: Exp)
      esac

operation

ZERO: Nat =
precondition TRUE
accesses Nat
return (zero → TRUE)

SUCC(n: Nat): Nat =
precondition TRUE
accesses Nat
return (zero → FALSE,pred → n)
```

```
EQUAL(m,n: Nat): Bool =
precondition TRUE
accesses Nat
return if m.zero
          then n.zero
          else if n.zero
                  then FALSE
                  else EQUAL(m.pred,n.pred)
               fi
       fi

EXP(n: Nat): Exp =
precondition TRUE
accesses Exp
return (atom → TRUE,val → n)

CONS(x,y: Exp): Exp =
precondition TRUE
accesses Exp
return (atom → FALSE,car → x,cdr → y)

CAR(x: Exp): Exp =
precondition NOT(ATOM(x))
accesses Exp
return x.car

CDR(x: Exp): Exp =
precondition NOT(ATOM(x))
accesses Exp
return x.cdr

RPLACA(x,y: Exp): Exp =
precondition NOT(ATOM(x))
accesses Exp
x.car := y
return x

RPLACD(x,y: Exp): Exp =
precondition NOT(ATOM(x))
accesses Exp
x.cdr := y
return x

ATOM(x: Exp): Bool =
precondition TRUE
accesses Exp
return x.atom

EQ(x,y: Exp): Bool =
precondition AND(ATOM(x),ATOM(y))
accesses Exp
return EQUAL(x.val,y.val)
```

2.4.3. Types

The specification above first of all contains the name of the data structure specified, i.e. *Lisp*. In the part prefixed by type the "types" which are used in the specification are listed, i.e. *Nat* and *Exp*. The type *Bool* together with its associated operations is supposed to be automatically included in each specification. A type will be associated with each syntactical construct in the specification which denotes an object. The objects denoted by syntactical constructs of type T will be called "objects of type T". Furthermore, with each type T a "representation" is associated. The representations of types are defined in the part prefixed by representation. The definitions have the shape of PASCAL [JENSEN & WIRTH 74] (variant) record type declarations. In contrast to PASCAL, the representation of a type T should not be considered to define the set of "values" of type T. (This set will be defined later.) It merely defines the way objects of type T may be created, accessed and modified. Put in more syntactical terms, the representations of types define the way syntactical constructs of the various types may be used to form new syntactical constructs. For example, if x is an accessor of type *Nat*, then the representation of type *Exp* gives us the right to write down the following construct ("creation") of type *Exp*:

$$(atom \rightarrow TRUE, val \rightarrow x).$$

(See the next paragraph on "access rights", however.) It is not allowed, at least not in the context of this specification, to write down a construct of type *Exp* such as:

$$(atom \rightarrow FALSE, car \rightarrow x, cdr \rightarrow x).$$

The representations of types can therefore best be viewed as restrictions imposed on the use of syntactical constructs. These restrictions (which can be checked "statically") guarantee that all operations performed on objects of a type T are well-defined. They also imply, of course, that the representation chosen for T is reflected in the structure of objects of type T.

2.4.4. Access rights

In the part of the specification prefixed by operation the operations which can be performed on objects of the different types are specified. An operation is specified by giving its name, its parameter list, which contains the formal parameters and their types, and the type of the result delivered by the operation. Since operations are partial functions a precondition must also be given. This precondition must hold for the actual parameters each time the operation is applied. An operation F may be defined in terms of the operations contained in the data structure specification only, unless it is explicitly specified to have an "access right" to a certain type T. In that case the "primitive" operations such as selection, creation and $\equiv$, may be applied to objects of type T in the definition of F. The types which are "accessible" to an operation are specified in the clause prefixed by accesses.

Access rights cannot freely be granted. Suppose for instance that an operation F has an access right to type T_2 but not to type T_1 and that objects of type T_1 contain components of type T_2. If there is an operation of the data structure that takes an object P_1 of type T_1 as (one of) its

parameter(s) and delivers a component $\overline{P}_2$ of $\overline{P}_1$ of type T_2 as its result, then F can sneakily access $\overline{P}_1$ through its component $\overline{P}_2$. These problems can be eliminated by introducing a special relation on the set of types of a data structure specification. A type T_2 will be called a subordinate type of a type T_1 if an object of type T_1 can have a component of type T_2. Due to the "strong typing" this relation can be determined effectively for a given specification. It constitutes a transitive relation, but not necessarily a partial order, on the set of types of a specification. For the specification of the data structure *Lisp* this relation is pictured in Figure 2.26, where "$\leftarrow$" denotes "is a subordinate type of".

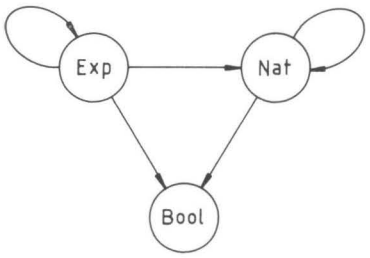


Figure 2.26

If we now require that for each operation F the following holds:

> If F uses a type T to which it has no access right,
> then F has no access right to any subordinate type of T.

then "sneak access" is impossible. Here the expression "F uses T" means that the definition of F contains a syntactical construct of type T. If F does not use T, F may have access rights to subordinate types of T. E.g., *SUCC* does not use *Exp*, but has an access right to the type *Nat*, which is a subordinate type of *Exp*. Note that the relation "be a subordinate type of" comes in rather naturally if we construct our specifications in a "modular" fashion (which of course we should).

2.4.5. The meaning of the specification of *Lisp*

The meaning of each operation is defined in terms of an algorithm followed by a return clause. As can be seen the algorithms are very simple (as they should be in a specification). The language used for the definition of the operations is a somewhat extended version of the language of Section 2.3. We shall sketch below how the operation definitions in the specification of the data structure *Lisp* can be transformed into operation definitions in the original language, and how these transformed definitions can be used to assign a meaning to the specification of *Lisp*.

First, take the "body" of the operation *EQUAL*:

```
return if m.zero
          then n.zero
          else if n.zero
                  then FALSE
                  else EQUAL(m.pred,n.pred)
               fi
       fi
```

and rewrite it as follows:

```
let e := ();
if m.zero
   then e := n.zero
   else if n.zero
           then e := FALSE
           else e := EQUAL(m.pred,n.pred)
        fi
fi
return e
```

Rewrite each operation definition as exemplified by the definition of the operation *CAR*:

```
CAR(x: Exp): Exp =
precondition NOT(ATOM(x))
accesses Exp
return x.car
```

which becomes:

```
CAR(x) = let r := ();
         if NOT(ATOM(x))
            then r := x.car
            else r := ERROR
         fi
         return r
```

Here *ERROR* is the "undefined operation", for example:

```
ERROR = while TRUE do od
        return ()
```

Replace all operation applications at places where they are not allowed in the original language by accessors, as exemplified by:

```
if NOT(ATOM(x))
   then r := x.car
   else r := ERROR
fi
```

which becomes:

```
let u := ATOM(x);
let v := NOT(u);
if v
   then r := x.car
   else r := ERROR
fi
```

Assuming that the truth values *true* and *false* are represented by the structures *true* and *false* from Figure 2.24, replace all paths at places where only assertions are allowed in the original language by assertions, as exemplified by:

```
if v
   then r := x.car
   else r := ERROR
fi
```

which becomes:

```
if v.a ≡ v.b
   then r := x.car
   else r := ERROR
fi
```

Finally, provide all applications of operations without parameters with parentheses, such as:

```
TRUE
```

which becomes:

```
TRUE()
```

Having transformed all operation definitions into the language defined in Section 2.3, we can use the function M_F to associate an "operation" with each operation definition (for the boolean operations, use the definitions given in Example 2.15). The formal meaning of the specification of the data structure *Lisp* is now defined by the following set of operations:

$$F_{Lisp} = \{M_F(F) \mid F = TRUE, FALSE, NOT, AND, OR, ZERO, SUCC, EQUAL, EXP, CONS, CAR, CDR, RPLACA, RPLACD, ATOM, EQ\}.$$

This set of operations defines a homogeneous algebra $\langle S, F_{Lisp} \rangle$ on the set S of all structures.

2.4.6. Immutable data structures

In the traditional specification methods for data structures, such as the algebraic specification methods [GOGUEN et al. 78], [GUTTAG & HORNING 78], the formal meaning of a specification is usually defined to be a heterogeneous algebra. This is possible because in these methods the arguments passed to operations are non-overlapping. Therefore, these arguments can be viewed as "values" drawn from separate "carriers" (one for each type) and the operations can be viewed as functions from Cartesian products of carriers into carriers. We shall show now that the case covered by the traditional specification methods is a special case of

the case covered by our specification method. The special case meant here is the case that all operations of a data structure are side effect free and static. A data structure with the latter properties will be called "immutable". (Notice, by the way, that operations of data structures are always environment independent: Only objects which are passed as parameters to an operation can be "seen" by that operation.) All data structures which can be specified in the traditional methods, such as for example the natural numbers, are immutable. (Check that the data structure *Lisp* is not immutable, but would have been immutable if the operations *RPLACA* and *RPLACD* had been omitted.) As we will show below, a heterogeneous algebra can be associated with each specification of an immutable data structure. This algebra corresponds to the algebras normally associated with traditional specifications.

2.4.7. The carrier of a type

In order to construct the heterogeneous algebra associated with the specification of an immutable data structure, "carriers" will be associated with the types of a specification first. These carriers are supposed to contain the "values" of a type. Since the concept of a carrier of a type is meaningful for specifications of "mutable" data structures as well, we shall define it for the types of all specifications.

First, what is a "value" of type *T*? A value of type *T* could be viewed as an object of type *T*. An object can never be viewed independently of a structure, however. A value of type *T* should therefore be regarded as a structure, i.e. as the structure (or "value") of an object of type *T*. It makes sense to consider only objects of type *T* which can actually be constructed. So the carrier of type *T* can be defined as the set of all structures of objects of type *T*, which can be constructed by an operation using exclusively the operations specified in the data structure specification to which *T* pertains. According to this definition the carrier V_{Nat} of the type *Nat* would consist of the structures pictured in Figure 2.25. The carrier V_{Exp} would among many other structures contain the cyclic structure of Figure 2.27.

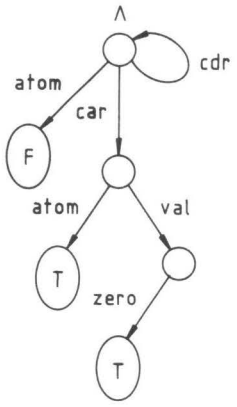


Figure 2.27

This structure can be constructed by the following operation, which uses the operations of the data structure *Lisp* only:

```
CONSTRUCT: Exp =
let x := EXP(ZERO);
x := CONS(x,x)
return RPLACD(x,x)
```

Notice that the carrier of a type contains only finite structures.

2.4.8. Indistinguishability

The above definition of the carrier of a type is not yet fully satisfactory. The reason is that the elements of a carrier as defined above cannot really be viewed as "values" in the sense of the traditional specification methods. They are in fact *representations* of values. Two different structures in a carrier may very well represent the same value. In order to make this more precise we introduce a special relation on the carrier of a type, called "indistinguishability", which corresponds to "being representations of the same value".

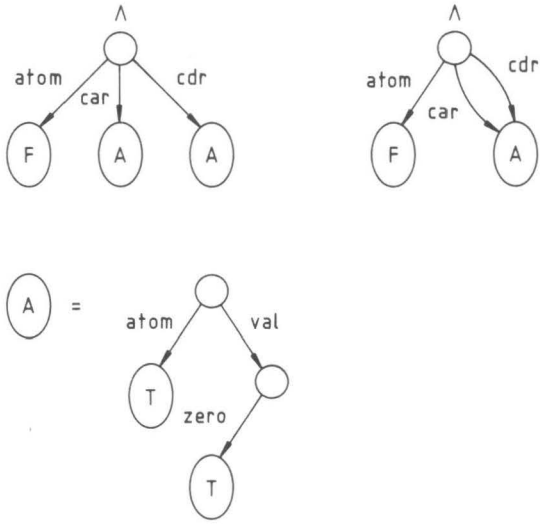
Two structures V_1 and V_2 in the carrier of a type T are "indistinguishable" if for each operation F with a single parameter of type T, which can be specified using the operations of the data structure only, the following holds: When applied to (instances of) V_1 and V_2, F is either defined in both cases or undefined in both cases (where "non-terminating" is also "undefined"). At first sight this may seem a strange definition. It becomes less strange if one realizes that the operations of the data structure are the only operations which may be used to manipulate (instances of) structures of type T outside the specification of the data structure, and that being defined or undefined is the only property of operations which is a priori observable to the outside world. So, for the outside world there is truly no way to tell two indistinguishable structures apart. The definition becomes even less strange if one realizes that the structures *true* and *false* in the carrier V_{Bool} of *Bool* are distinguishable (= not indistinguishable). The following single parameter operation "distinguishes" them:

```
DISTINGUISH(b: Bool): Bool =
while b do od
return b
```

Consequently, another way to prove that two structures V_1 and V_2 in the carrier of a type T are distinguishable is to construct an operation F with a single parameter of type T, using only the operations of the data structure to which T pertains, such that $F(V_1)$ yields *true* and $F(V_2)$ yields *false*. That is the approach we will adhere to below. Note that indistinguishability is an equivalence relation.

In Figure 2.28.a two indistinguishable structures contained in V_{Exp} are pictured. (Check that there is indeed no way to distinguish them through the operations of the data structure *Lisp*.) The two structures in Figure 2.28.b, which are also contained in V_{Exp}, are distinguishable.

2.28.a



2.28.b

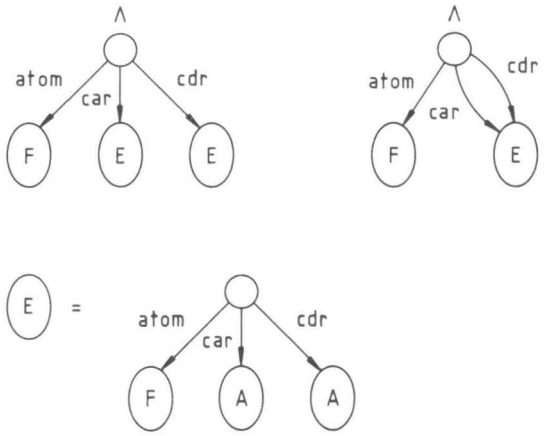


Figure 2.28

This can be seen by passing either of them (in the shape of an object) as a parameter to the following operation:

```
DISTINGUISH(x: Exp): Bool =
let y := CAR(x);
y := RPLACA(y,x);
let z := CDR(x);
z := CAR(z)
return ATOM(z)
```

This operation will deliver the value *true* when applied to the first and *false* when applied to the second structure in Figure 2.28.b. Notice that the reason why the structures of Figure 2.28.a are not distinguishable while the structures of Figure 2.28.b are, lies in the fact that there are no operations to modify atomic objects of type *Exp*, while there are such operations for non-atomic objects of type *Exp* (viz., the operations *RPLACA* and *RPLACD*). If, however, the operation *EQ* would have been defined as in certain LISP implementations by:

```
EQ(x,y: Exp): Bool =
precondition TRUE
accesses Exp
return if x ≡ y
          then TRUE
          else FALSE
       fi
```

then the structures of Figure 2.28.a would also be distinguishable.

It is obvious now to define a value of type T as an equivalence class of the indistinguishability relation. The carrier of T can then be redefined as the quotient of this equivalence relation. In this sense we will use the concepts of a value and the carrier of a type in the sequel. Notice that what we did is in fact common mathematical practice. The rational numbers, for example, are usually defined as equivalence classes of an equivalence relation defined on the set of all pairs (m,n), where m and n are integers (and $n \neq 0$). These pairs can be viewed as representations of the rational numbers, just like structures can be viewed as representations for the values of a type.

2.4.9. The meaning of the specification of an immutable data structure

Let us return to the immutable data structures. If we have a specification of an immutable data structure D, a carrier V_T can be associated with each type T of D, as described above. If F is an operation of D with formal parameters $X_1,\ldots,X_n$ of types $T_1,\ldots,T_n$ and delivering a result of type T_0, then we know, since F is static, that the structure of the result of F when applied to the objects $P_1,\ldots,P_n$ of types $T_1,\ldots,T_n$ depends only on the structures of the objects $P_1,\ldots,P_n$. Consequently, F may be considered as a (partial) function which maps structures $V_1,\ldots,V_n$ of types $T_1,\ldots,T_n$ to structures of type T_0. The result of applying F to $V_1,\ldots,V_n$ will be denoted by $F(V_1,\ldots,V_n)$.

The operation F may not only be considered as a function which maps structures to structures, it may even be considered as a function which maps values to values. In order to show this, we have to prove that, if V_i and W_i are indistinguishable structures of type T_i $(i = 1,\ldots,n)$, then $F(V_1,\ldots,V_n)$ and $F(W_1,\ldots,W_n)$ are also indistinguishable. Let V_i and W_i be indistinguishable structures of type T_i $(i = 1,\ldots,n)$. If $F(V_1,\ldots,V_n)$ and $F(W_1,V_2,\ldots,V_n)$ were distinguishable, we could construct an operation which distinguishes V_1 and W_1, but V_1 and W_1 are indistinguishable. So $F(V_1,\ldots,V_n)$ and $F(W_1,V_2,\ldots,V_n)$ are indistinguishable. Analogously, we can prove that $F(W_1,V_2,\ldots,V_n)$ and $F(W_1,W_2,V_3,\ldots,V_n)$ are indistinguishable, etc.. Using the transitivity of the indistinguishability relation, we infer that $F(V_1,\ldots,V_n)$ and $F(W_1,\ldots,W_n)$ are indistinguishable.

The above, together with the fact that F is side effect free, implies that we can view F as a (partial) function from $V_{T_1} \times \ldots \times V_{T_n}$ into V_{T_0}.

Instead of the homogeneous algebra $<S,F_D>$ associated with D, we can now associate the heterogeneous algebra $<V_T\ (T \in T),M_F\ (F \in F_D)>$ with D, where T is the set of types ("sorts") of D and M_F is the interpretation of the operation F as a mapping from values to values.

2.4.10. Behavioural equivalence

In the preceding part of this section we showed how a mathematical object can be associated with a specification of a data structure. Thus the necessary mathematical rigour of the specification method can be obtained. The question remains (even in the case of immutable data structures) whether the mathematical object associated with a specification defines the "true" meaning of the specification. Clearly, in our case it does not. We could for example have specified the data structure *Lisp* using different representations for the types. Though the formal meaning of this specification would be different from the meaning of the specification given before, we would still feel that both specifications specify the "same" data structure. Instead of assigning a more abstract formal meaning to specifications of data structures (as, in fact, we already did for specifications of immutable data structures), we shall cope with this problem by introducing a relation between specifications of data structures. This relation, called "behavioural equivalence" (cf. [BERZINS 79]), amounts to "specifying the same data structure".

First, let us define the concept of a "signature" of a specification in the usual way. The signature of a specification S will be defined to be the set of all tuples $(F,T_1,\ldots,T_n,T_0)$, where F is the name of an operation specified in S, $T_1,\ldots,T_n$ are the types of the formal parameters of F and T_0 is the type of the result delivered by F. Notice that the formal parameters, the preconditions and the access rights of operations are not included in the signature of S. Notice also, that each operation (or algorithm) which is specified in terms of the operations of S only, can also be viewed as an operation (or algorithm) specified in terms of the operations of any other specification with the same signature as S. The meaning of the operation (or algorithm) may be entirely different in either case, though.

Two specifications S_1 and S_2 of a data structure are "behaviourally equivalent" if they have the same signature and if for each parameterless operation F defined in terms of the operations of S_1 (or S_2) the following holds: The result of F according to the formal meaning of S_1 is defined iff the result of F according to the formal meaning of S_2 is defined. In view of the definition of the concept of indistinguishability, this definition will be clear. If S_1 and S_2 are behaviourally equivalent, there is really no way to tell a result produced by the operations of S_1 from a result produced in the same way by the corresponding operations of S_2. So S_1 and S_2 truly display the "same" behaviour. This implies, for example, that each "assertion" (= parameterless function defined in terms of the operations of S_1 or S_2 with a boolean result) is valid with respect to S_1 iff it is valid with respect to S_2. The above definition of behavioural equivalence is surprisingly much simpler than the one given in [BERZINS 79].

2.4.11. Nondeterministic creation

We shall conclude this informal section with the discussion of a number of features which should be added to the language in order to make it suitable as a general purpose specification language. A feature that must be added first of all is the possibility to specify nondeterministic

operations. This feature is indispensable to the construction of specifications with a high level of abstraction. The simplest way to introduce nondeterministic operations is by the addition of a "nondeterministic creation" to the language, which may be used everywhere where a "construct" is allowed. For that purpose we first have to introduce "predicates" in the language, which will allow assertions on objects, for example in first order predicate calculus, to be used in the language. If P is such a predicate on objects of type T, a nondeterministic creation might look as follows:

$$\mu\ x\ :\ T\ [P(x)]$$

The effect of the evaluation of this construct is that an arbitrary object X of type T is created which satisfies $P(X)$. (Notice that an assignment with a nondeterministic creation as its right-hand side corresponds to a "nondeterministic assignment" [HAREL et al. 77].)

The body of an operation which should deliver some arbitrary value X satisfying certain requirements $R(X)$ can now be specified at a high level of abstraction by:

$$\underline{\text{return}}\ \mu\ x\ :\ T\ [R(x)]$$

The addition of the nondeterministic creation to the language may make specifications non-executable, but for specifications this is not really an objection. Moreover, the semantics of the language must be adjusted, for example by considering operations as mappings from sets of states to sets of results instead of mappings from states to results.

Apart from the nondeterministic creation, other nondeterministic control structures should be added to the language. One could think for example of for-loops of the kind "For each x satisfying $P(x)$ do ..." or guarded commands [DIJKSTRA 75]. One should be very careful, though. The combination of even a few such control structures, though each useful in itself, may easily create a baroque language. One should therefore aim at simple but powerful nondeterministic control structures. The nondeterministic creation is believed to be such a control structure. It will be used in the ensuing chapters in the form of nondeterministic assignments such as "Let x be such that $P(x)$".

2.4.12. Miscellanea

A feature which is also useful is to distinguish "constant" and "variable" accessors in the language. Constant accessors differ from variable accessors in that they may not be used as the right-most accessor in the left-hand side of an assignment. This need not imply that the object bound to such an accessor is constant (it may contain variable components). It merely implies that the identity (whatever that may be) of the object bound to the accessor does not change. This concept of a constant accessor is easily added to the language. Things become more complicated if we want to ensure that an object bound to a constant accessor is really constant (i.e. that its structure is constant).

There are several other useful features which could be added (such as hidden operations, type parameterization, etc.), but we shall stop the discussion here. The language which will be used in the other chapters of this monograph for the expression of algorithms and data structures will include several of the features which we discussed above. The syntax of this language will furthermore be different from the language discussed

here. (It will be more "natural-language-like", but the correspondence is easy to establish.) As we tried to demonstrate in the foregoing, the semantics of this language can be defined precisely and the semantical problems caused by shared and dynamic data can be solved in a relatively simple way. Apart from defining the semantics of the language the construction of a proof system for proving assertions about specifications is also necessary. (A good indication that the construction of such a proof system is indeed feasible is [MANNA & WALDINGER 80a].) The presence of such a proof system is implicitly assumed when we reason about algorithms and data structures in the sequel. (Strictly speaking a proof system is not necessary, because we could take the semantics and ordinary mathematics to prove assertions about specifications. A proof system is almost indispensable, however, because it makes proofs of correctness considerably easier, possibly even to such an extent that these proofs can be constructed or at least verified by a machine.)

2.5. COMPARISON OF STRUCTURES WITH OTHER CHARACTERIZATIONS OF STORAGE STRUCTURES

The concept of a structure as defined in this chapter is believed to characterize storage structures in a way more abstract than other methods. It is believed to capture exactly the access properties of a storage structure and no more than that. (What else is a storage structure other than its access properties?) In order to support this assertion we shall compare structures with other methods of characterizing storage structures. This will be done by giving a short characterization of a method and by showing how the structure S of Figure 2.29 would be represented in that method.

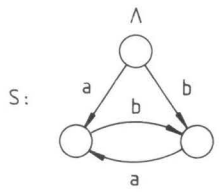


Figure 2.29

All characterizations are simplifications of the methods. Yet they are believed to capture the essential features of the methods. No attempt at completeness has been made, though the most prominent methods are all discussed.

2.5.1. Vienna objects

A "Vienna object" [WEGNER 72], [OLLONGREN 74], [STANDISH 78] can be characterized syntactically as follows:

```
<object>
    ├→ <atom>
    ├→ <compound>
    └→ Ω
<compound>
    └→ (<pair> {, <pair>})
<pair>
    └→ [<selector> : <object>]
```

Here an *atom* is a primitive object (e.g., an integer) and a *selector* corresponds to an accessor. Ω is the "null" object. The structure S of Figure 2.29 could be represented as follows:

$$S = ([a : ([b : b])],[b : ([a : a])]).$$

Vienna objects are basically trees with branches labelled by selectors and atoms as their leaves. This implies that sharing and circularity can only be described by introducing a pointer concept, which is done by allowing "composite selectors" (paths) to be used as atoms. This introduces all the disadvantages of pointers such as the danger of "dangling references" and the fact that an object may have more than one representation. E.g., we could have represented S also by:

$$S = ([a : ([b : ([a : a])])],[b : ab]).$$

Pointers are things which belong to the implementation world. They do not belong at the level of abstraction required for specifying objects. Using the structure concept, arbitrarily constructed objects can be described without the use of pointers.

2.5.2. Graphs

Another well-known characterization of a storage structure is the graph [ROSENBERG 71], [EARLEY 71], [MAJSTER 77]. Such a graph is a triple $\langle N,F,R\rangle$, where N is a set of nodes (which are the "objects"), F is a collection of partial mappings from N into N and R is a special node (the "root"). The structure S could be represented by the graph $\langle N,F,R\rangle$ with:

$$N = \{1,2,3\},$$
$$F = \{a,b\},$$
$$R = 1,$$

where:

$$a(1) = 2,\ a(2) = \uparrow,\ a(3) = 2,$$
$$b(1) = 3,\ b(2) = 3,\ b(3) = \uparrow.$$

Here "↑" means "undefined". For the description of sharing and circularity pointers are not necessary. However, the representation of a storage structure is far from unique. There are numerous graphs which correspond to the same structure. Not only may the nodes of a graph be chosen in different ways (e.g., $3,5,7$ instead of $1,2,3$), but graphs may also contain unreachable nodes (as the consequence of operations performed on them). This leads to all kinds of unnecessary complications in working with graphs (like performing a garbage collection, which, again, is something that belongs to the implementation world). Note that, in a certain sense, the

nodes of a graph are superfluous: They only act as meeting places of access paths, which is exactly what objects in a structure are about. Note also that creation is an unnatural operation in a graph: A node is "created" by selecting it from a set of already existing "free" nodes.

2.5.3. States of a state machine

A "state machine" [BERZINS 79], [PARNAS 72] can be viewed (in a very simplified way) as a triple $\langle N,\Sigma,R\rangle$, where N is a set of "objects", Σ is a set of "state functions" and R is a special object (the "root"). A state function is a total mapping from N into a set containing such things as integers, booleans, tuples and sets of objects, etc.. A state machine is always in a certain "state" $\sigma \in \Sigma$. The value of an object in a state can be determined by applying σ to that object. The structure S could for example be represented by the following state σ of a state machine $\langle N,\Sigma,R\rangle$ with $N = \{1,2,3,\ldots\}$ and $R = 1$:

$$\begin{aligned}
&\sigma(1) = \{(a,2),(b,3)\},\\
&\sigma(2) = \{(b,3)\},\\
&\sigma(3) = \{(a,2)\},\\
&\sigma(n) = \text{undefined} \quad (n \geq 4).
\end{aligned}$$

All disadvantages of graphs apply here too. There is an additional disadvantage, because objects are no longer pure storage structures. In order to access an object all kinds of operations (such as selection from a set) must be performed, which belong to the realm of abstract data types. This confusion of levels of abstraction is not desired, particularly not if storage structures are used in the specification of abstract data types as in [BERZINS 79].

2.5.4. Relational objects

A "relational object" [EARLEY 73], [KENNEDY & SCHWARTZ 75] can be characterized as follows:

> An object is either
> (1) An atom.
> (2) A set of objects.
> (3) A tuple of objects.

Storage structures are characterized in a set-theoretic way here, where an "atom" may be anything primitive. This approach is somewhat similar to the state machine approach, which implies that the same disadvantages hold (most notably, the confusion of levels of abstraction). If we choose the natural numbers as our atoms, the structure S could be represented by:

$$S = (\{(1,a,2),(1,b,3),(2,b,3),(3,a,2)\},1).$$

This could create the impression that relational objects are the same as graphs, which is not true. Relational objects are more general than graphs (each graph can be described as a relational object, but not the reverse). They have in common with graphs, that sharing can only be modelled by representing objects in some way as primitive values (the natural numbers above). The programming language SETL [KENNEDY & SCHWARTZ 75] even has a special atomic data type for this purpose.

2.6. CONCLUSION

In this chapter we addressed the specification problem for algorithms and data structures. The basis of our discussion was the novel concept of a "structure", which is essentially a simple mathematical model of the access properties of a storage structure. Using this model, storage structures with arbitrary sharing and circularities can be characterized without the need to introduce pointers. Creation and replacement become very natural operations, which cannot produce any "garbage" since the concept of unreachability is non-existent in a structure.

Due to the fact that structures are general and yet free of such low level concepts as pointers and garbage, they lend themselves very well to the basis of a specification language for realistic algorithms and data structures, including those algorithms and data structures which involve dynamic and shared data. We indicated how such a language can be constructed. First, we defined a language for manipulating structures. The semantics of this language could be defined in a very simple way. Then, led by an example of a specification, we sketched how this language can be extended into a full-fledged specification language through the addition of abstraction facilities. The latter language, though syntactically different, corresponds to a great extent to the specification language used in the following chapters.

The language which has been sketched in this section is suited for use at extremely different levels of abstraction. Instead of a specification language it could just as well be viewed as a programming language, which is comparable to other languages featuring abstract data types such as CLU [LISKOV et al. 77], ALPHARD [WULF et al. 76], EUCLID [LAMPSON et al. 77] or MODULA [WIRTH 77]. The ideas set out in this chapter may therefore also be of interest to the design of programming languages. In particular, it is contended that on the basis of these ideas it is possible to design a general purpose programming language of a simplicity comparable to PASCAL [JENSEN & WIRTH 74], but differing from it in three major respects. First, the language is free of pointers. Secondly, it has an abstract data type facility as its sole data structuring mechanism. Thirdly, it has a simple and rigorous semantics. The design of such a language lies outside the scope of this monograph, however.

CHAPTER 3

IMPLEMENTATION

3.0. INTRODUCTION

3.0.1. The transformational approach

One of the central problems in computer science is the implementation problem: How to construct an efficient algorithm or data structure from a given specification. Let us, for the time being, restrict ourselves to algorithms. A rather promising and increasingly popular approach to the implementation problem for algorithms is the transformational method. The basic idea behind the method of algorithm transformation (or "algorithmics" [MEERTENS 79]) is to start with a simple "abstract algorithm", which can easily be proved correct but which may be intolerably inefficient. Then a number of correctness-preserving transformations are applied to the algorithm, turning it into a more complex "concrete algorithm", which is still correct and (hopefully) more efficient. The virtues of this approach are widely known and will not be discussed here. For a short introduction and survey the reader is referred to [DARLINGTON 79].

The correctness of the abstract algorithm which serves as the starting point of the transformation process can be proved by conventional means, e.g. by using the inductive assertion technique [HOARE 69]. If the abstract algorithm and the problem specification coincide, this step is not even necessary. Problems arise if an attempt is made to prove that the transformations applied to the abstract algorithm do not affect the correctness of the algorithm. The conventional verification methods seem to fall short here. They are usually extended with rather heavy formal machinery (see for example [BACK 80]), which increases the complexity of the verification process considerably.

In this chapter a simple method for the derivation of algorithms through correctness-preserving transformations will be presented, which allows us to prove that a transformation is correctness-preserving using standard verification techniques. As such it could be used to prove the correctness of the transformations which are used to derive the garbage collection and compaction algorithms in Chapter 5. Instead of a verification tool, it could just as well be viewed as a design tool. (Ideally, it should be used simultaneously as both.) As such it will be used in Chapter 6.

The method is based on two considerations. First, given the surrounding intermediate assertions, the correctness of most local transformations is self-evident. Secondly, global transformations which amount to a change of representation of a variable (or variables) can be reduced to a number of local transformations. The method, therefore, is essentially a way to accomplish a change of data representation in a correctness-preserving way. It does not cover such global transformations as recursion removal or loop fusion. As we shall see in Chapters 5-7, however, the majority of all global transformations which are applied there, are changes of data representation.

3.0.2. Transforming algorithms by adding and removing variables

In a nutshell the idea is as follows. Let an algorithm S be given, which is a correct solution to a certain problem. Suppose S contains a variable V, which we would like to replace by another variable W with a more efficient representation. The addition to S of a number of well-defined assignments to W will not affect the correctness of S. (That is, if these assignments have no side effects, which we shall tacitly assume throughout this chapter.) Having added assignments to W a number of intermediate assertions, which relate W to other variables in S (particularly V), can be proved to hold inside S. These intermediate assertions can be used to replace certain expressions in S by equivalent or more restrictive ones, which clearly does not affect the correctness of S. If the proper assignments to W were added to S, it should be possible to make these replacements in such a way that V is not used anywhere else but in assignments to itself. Consequently, V has turned into a "redundant variable", the assignments to which, may be removed from S without affecting the correctness of S. (Again, assuming the absence of side effects.) Thus the global change of variable from V to W can be performed step by step by the following local transformations: adding assignments to W, making local replacements and removing assignments to V. Notice that this scheme works just as well if V and W are sets of variables instead of single variables.

The above scheme constitutes a very flexible way of changing the representation of variables. Since the derivation of many algorithms amounts to continually changing the representation of variables, it is also very general. In a derivation of an algorithm according to this scheme only small steps are taken, which can easily be seen to be correctness-preserving by proving intermediate assertions (if necessary). No enhancement of existing verification techniques is therefore required, at least not to convince oneself intuitively of the correctness-preservation of each step. From a strictly formal point of view such an enhancement is still necessary, of course. The formalization of the scheme would, among many other things, require a precise definition of such concepts as "correctness-preservation", "redundant variable", "local replacement", etc.. It is believed that this formalization will not pose any serious problems. The level of formality required for it is not sought for here. Things will be kept intuitive, yet sufficiently precise to be confident about the formal soundness.

3.0.3. Extension to data structures

The method described in this chapter can easily be extended to work for data structures as well. (In view of the fact that the method is essentially a way of accomplishing a change of data representation, this will come as no surprise.) Assume that we use the method described in Chapter 2 to specify data structures. Following this method a data structure is specified by choosing representations for its constituent types and defining the operations of the data structure in terms of algorithms operating on these representations. The implementation of a data structure can be viewed as applying correctness-preserving transformations to the specification of the data structure. These transformations can be divided into two classes: "algorithmic" and "structural" transformations.

An algorithmic transformation transforms the algorithm through which an operation is specified. The method described here can be used directly for such a transformation. A structural transformation transforms the

representation of a constituent type of a data structure. A structural transformation of a type *T* can be viewed as replacing some of the "accessors" (see Chapter 2) used in the representation of *T* by other accessors. Suppose, for example, we wish to replace the accessor *A* by another accessor *B*. This can be done as follows. Introduce the accessor *B* in the representation of *T*, which implies that each object of type *T* has an additional component accessed by *B*. Subsequently, well-defined assignments of the kind "*X.B* := *Y*", where *X* denotes an object of type *T*, can be added to the operations of the data structure without affecting the correctness of the (specification of the) data structure. A number of "representation invariants", which relate *B* to the other accessors of the representation of *T* (particularly *A*), can then be proved to hold. (Note: The representation invariants need not hold inside operations that have an access right to *T*, but they should hold "between" the operations of the data structure.) The representation invariants can be used to replace certain expressions in the operations of the data structure by equivalent or more restrictive ones. If the proper assignments were added, it should be possible to make these replacements in such a way that the accessor *A* is not really used any more in any of the operations of the data structure. This "redundant accessor" may then be removed from the representation of *T* without affecting the correctness of the data structure. Thus structural transformations can be applied in essentially the same way as algorithmic transformations.

The above justifies the fact that we shall restrict ourselves to discussing the method as an implementation technique for algorithms. A comprehensive example of the use of the method as an implementation technique for data structures can be found in Chapter 6. There the method is used to transform an abstract machine (which is essentially a data structure) into a machine with an efficient storage management system. Furthermore, the emphasis here will be on the verification aspect of the method, i.e., on the method as a tool to prove the correctness of algorithm transformations.

The method will be described in detail in the next section. In Section 3.2 the effectiveness of the method will be demonstrated in the derivation of a well-known test case for verification techniques: the Deutsch-Schorr-Waite marking algorithm [SCHORR & WAITE 67], henceforth called the DSW-algorithm. In contrast to most other proofs of correctness of the DSW-algorithm [DE ROEVER 78], [DUNCAN & YELOWITZ 79], [GERHART 79], [GRIES 79], [KOWALTOWSKI 79], [TOPOR 79], [DERSHOWITZ 80] the most general form of the algorithm will be chosen here. In Subsection 3.2.1 the problem will be defined precisely. From the specification given there, a simple algorithm can be derived almost immediately. This algorithm is presented and proved correct in Subsection 3.2.2 using the inductive assertion technique. Then, in five subsequent "phases" (Subsections 3.2.3-3.2.8), each of which follows exactly the scheme described in Section 3.1, the DSW-algorithm is derived from this algorithm by correctness-preserving transformations. The intermediate assertions which are required in this derivation process are again proved by using the inductive assertion technique. The algorithmic language used is somewhat informal. In so far as the semantics of the constructs of this language is not self-evident, this will be explained. Some concluding remarks are made in Section 3.3.

3.1. A SIMPLE IMPLEMENTATION METHOD

3.1.1. General strategy

In this section a detailed description of the implementation method will be presented as it will be applied in Section 3.2. We assume the problem is to construct an (efficient) algorithm which establishes a certain input-output relation. The first stage is to construct a simple abstract algorithm S and prove its partial correctness using the inductive assertion technique. We assume that the latter is done, as usual, by inserting intermediate assertions (henceforth called "assertions") in the algorithm. These assertions will from now on be considered to be part of the algorithm. (We need them for local replacements.) If sufficiently abstract, the algorithm S will probably be highly nondeterministic. Though it need not necessarily terminate, it must be such that a terminating (and consequently totally correct) algorithm can be derived from it by curtailing the nondeterminism.

An iterative process of global correctness-preserving transformations is now started. The objective of each iteration, or "phase" as we shall call it, is to make the algorithm S more efficient. This process is continued until a sufficiently efficient (and therefore terminating) algorithm results. The rules of the game are that essentially the objective of a phase is achieved by replacing a set X of old variables of the algorithm by a set Y of new variables, where X and Y may be arbitrarily large or small. Two major objectives which can be realized this way are: reducing nondeterminism and changing the representation of variables. In the former case we could for example have $X = \emptyset$ and $Y = \{V\}$, where V is a fresh variable which is used to "control" the nondeterminism. In the latter case we could for example have $X = \{V\}$ and $Y = \{W\}$, where W is a variable with a more efficient representation than V. The precise rules of the game are presented below.

3.1.2. The four steps

Each phase consists of the following four steps (which will be explained below):

Step 1

Choose fresh variables and insert new assertions (to be made valid) expressing a relation between the old and new variables.

Step 2

Add assignments to the new variables and using the (old and new) assertions make replacements so as to make the new assertions valid. Prove that the new assertions are valid.

Step 3

Using the assertions make replacements so as to improve the algorithm and remove all assignments to redundant variables.

Step 4

Replace the assertions containing redundant variables by equivalent or weaker assertions not containing redundant variables.

3.1.2.1. Step 1

In the first step fresh variables are introduced and the objective to be achieved by the transformation phase is laid down in a number of assertions, which relate these new variables to the old variables of the algorithm. They are inserted at the appropriate places in the algorithm. These assertions need not fully express the objective to be achieved. (This is often impossible anyway, e.g. if the objective is to impose "dynamic" restrictions on the algorithm.) They need only contain the information necessary to achieve the objective in the next steps.

3.1.2.2. Step 2

The purpose of Step 2 is to make the new assertions valid by adding assignments to the new variables. As it turns out, it is not always possible to make these new assertions hold solely by adding assignments to the new variables. It may be necessary to perform a number of replacements as well. This situation (examples of which will be encountered) occurs typically with new assertions, which are introduced in order to replace nondeterministic operations on the old variables by more deterministic operations. New assertions of this type allow the assertions on the old variables to be strengthened. Hence it is impossible to make these new assertions hold solely by adding assignments to the new variables. Replacements involving the old variables must also be performed.

It is obvious to allow replacements based on the old assertions only in Step 2. In the case of a restriction of nondeterminism, this implies that the restriction of the nondeterminism must be accomplished without the use of the new variables, which are often specifically introduced for that purpose. If possible at all, this may make the restriction of the nondeterminism an unnecessarily awkward affair. It seems reasonable, therefore, to allow replacements based on already valid new assertions as well. Yet, this is still not sufficient. Consider a replacement which is to be made inside a loop in order to make a new assertion, inside that loop, hold. Due to the cyclic nature of loops, the correctness of this replacement may depend on the new assertion, the truth of which the replacement is supposed to establish.

The way out is to allow replacements in Step 2 which are based on both the old and new assertions, even if the latter are not yet valid. At first sight this may seem to introduce a vicious circle and therefore be incorrect. The surprising thing is that it is not. (In view of the fact that the inductive assertion technique is based on induction, it may not be so surprising after all.) This will be proved at the end of this section. From a practical point of view it is a great convenience that the new assertions can be used freely before their truth has been established. Another question is whether it is really necessary from a theoretical point of view. We shall argue in Subsection 3.2.7 that in a certain sense it indeed is. In the derivation of the DSW-algorithm to be presented, there are two places where assertions are used before their truth has been established. Both could have been avoided, though in one case in a highly artificial way.

3.1.2.3. Step 3

The third step is to fully exploit the new assertions to make replacements in the algorithm so as to achieve the desired objective. Strictly speaking this could already have been done in the second step, but

from a conceptual point of view it is better to separate the replacements necessary to make the new assertions hold from the other "optimizing" replacements. The class of replacements allowed will not be defined here. The only requirement is that the replacements must be very simple and evidently correctness-preserving. The replacements can be used either to replace expressions by more efficient ones, or to turn certain variables into redundant variables. What exactly is meant by a "redundant variable" will not be defined here. Broadly speaking a variable is redundant in an algorithm if it is a local variable of the algorithm and it is used in assignments to itself only. It is obvious that the assignments to such a variable may be removed from the algorithm without affecting the correctness.

3.1.2.4. Step 4

In Step 3 all assignments to redundant variables have been removed. Consequently these variables have turned into "ghost variables" of the algorithm. Yet, they may (and probably will) still occur in the assertions. From a strictly formal point of view these assertions no longer hold now. Simply throwing them away would probably make the remaining assertions too weak for further use. In Step 4, therefore, new and sufficiently strong assertions, in which the redundant variables no longer occur, must be derived from the old assertions to take their place. This could be done in a systematic way by putting an existential quantifier before each assertion, quantifying over all redundant variables. It is easy to see that these derived assertions will hold.

3.1.3. Some remarks

Though the final algorithm obtained by the transformation process described above is partially correct "by construction", it must still be proved to terminate. This need not necessarily be done afterwards, but can be done at some intermediate stage in the derivation. Note that, if necessary, between two phases or between Steps 2 and 3 additional assertions can be proved and inserted in the algorithm. Note also that if we consider the proof of the new assertions in Step 2 as a separate step, the steps performed in a phase are almost perfectly symmetrical:

Step 1: Introducing variables &
Strengthening assertions.

Step 2: Adding assignments &
Making replacements.

Proving assertions.

Step 3: Making replacements &
Removing assignments.

Step 4: Weakening assertions &
Eliminating variables.

If desirable, the assertions of the final algorithm can be used to give an independent proof of correctness of that algorithm. This saves one the trouble of inventing the assertions required for an independent proof of correctness. It may turn out, however, that the assertions of the final

algorithm are too weak for that purpose. So, if an independent proof of correctness of the final algorithm should be possible, care must be taken to keep the assertions strong enough. The latter is entirely the responsibility of the algorithm constructor.

Since the DSW-algorithm to be derived in Section 3.2 consists of a single loop, it is more convenient to keep track of the loop invariants instead of the (intermediate) assertions. In Section 3.2 invariants will therefore be used instead of assertions. Each invariant corresponds to four assertions: one immediately before the loop, one at the beginning and one at the end of the loop body, and one immediately after the loop. If assertions at other places in the algorithm are required in order to apply a transformation, they will be derived (in a usually straightforward way) from the invariants.

3.1.4. On using assertions before they are valid

As we promised above we shall show now that in Step 2 we can indeed safely use the new assertions for replacement purposes before their truth has been established. This we shall do by applying Step 2 in a more circumstantial way. Let S be the algorithm prior to Step 2. Let X be the set of old variables, and Y the set of new variables introduced in Step 1. Let $P_i(X)$ be the old assertions, and $Q_i(X,Y)$ the new assertions introduced in Step 1 $(i = 1,2,...)$. Here the index i denotes a place in the algorithm. Consider a statement $S_i(X)$ in S, prior to which the assertion $P_i(X)$ holds. This will be denoted as follows:

$$S = \ldots \{P_i(X)\}\ S_i(X) \ldots$$

First of all strengthen $P_i(X)$ to $R_i(X)$, where $R_i(X)$ is the strongest assertion which holds prior to $S_i(X)$ (consequently $R_i(X) \Rightarrow P_i(X)$):

$$S = \ldots \{R_i(X)\}\ S_i(X) \ldots$$

Insert a nondeterministic assignment "$X,Y := [R_i(X) \wedge Q_i(X,Y)]$" prior to $S_i(X)$, which assigns values to the X- and Y-variables in such a way that $R_i(X) \wedge Q_i(X,Y)$ holds afterwards. This does not affect the correctness of the algorithm, because $R_i(X)$ still holds prior to $S_i(X)$:

$$\begin{aligned} S = \ldots\ & \{R_i(X)\}\ X,Y := [R_i(X) \wedge Q_i(X,Y)] \\ & \{R_i(X) \wedge Q_i(X,Y)\}\ S_i(X) \ldots \end{aligned}$$

Make replacements in $S_i(X)$ based on the validity of $P_i(X)$ (implied by $R_i(X)$) and $Q_i(X,Y)$ prior to $S_i(X)$. Suppose these replacements turn $S_i(X)$ into $T_i(X,Y)$:

$$\begin{aligned} S = \ldots\ & \{R_i(X)\}\ X,Y := [R_i(X) \wedge Q_i(X,Y)] \\ & \{R_i(X) \wedge Q_i(X,Y)\}\ T_i(X,Y) \ldots \end{aligned}$$

Make additional replacements in S and add assignments to Y-variables in such a way that $Q_i(X,Y)$ will hold prior to "$X,Y := [R_i(X) \wedge Q_i(X,Y)]$":

$$\begin{aligned} S = \ldots\ & \{R_i(X) \wedge Q_i(X,Y)\}\ X,Y := [R_i(X) \wedge Q_i(X,Y)] \\ & \{R_i(X) \wedge Q_i(X,Y)\}\ T_i(X,Y) \ldots \end{aligned}$$

Remove the nondeterministic assignment "$X,Y := [R_i(X) \wedge Q_i(X,Y)]$":

$$S = \ldots \{R_i(X) \wedge Q_i(X,Y)\}\ T_i(X,Y) \ldots$$

Finally weaken $R_i(X)$ to $P_i(X)$:

$$S = \ldots \{P_i(X) \wedge Q_i(X,Y)\}\ T_i(X,Y) \ldots$$

The above sequence of transformation steps is evidently correct and can be applied simultaneously to all statements of S. In its effect it is the same as Step 2. Consequently the latter is also correct.

3.2. AN EXAMPLE: THE DSW-ALGORITHM

3.2.1. Problem

Given is a finite set G of objects. Each object is composed of a finite number of components. The set of all components of an object X is denoted by *comp(X)*. Different objects have different components (so objects do not "overlap"). Associated with each object X is a unique reference, denoted by *ref(X)*, which is said to refer to X. The unique object which has reference p associated with it, will be denoted by *obj(p)*. Each component C of an object contains a value, denoted by *val(C)*. A reference is a value. Among other values (which we are not interested in here) references may therefore be contained in components of objects. A component of an object which contains a reference will be called a branch of the object. The set of all branches of an object X will be denoted by *branches(X)* and the number of branches of X by *degree(X)*. The branches of X are numbered from 1 to *degree(X)*. The i-th branch of X (where $1 \leq i \leq degree(X)$) is denoted by *branch(X,i)*. Objects will be pictured as in Figure 3.1. There is a dummy object, denoted by *null*, which is not an element of G. The reference of *null* is denoted by *nil*: $nil = ref(null)$.

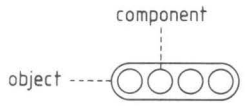


Figure 3.1

The set G of objects is closed. This implies that for each reference p contained in a branch of an object in G, the object referred to by p is also in G. There is one special object R in G, called the root. G can now be viewed as a directed graph, where the objects are the nodes and the references contained in branches are the edges of the graph. An example of how G may look like is given in Figure 3.2.

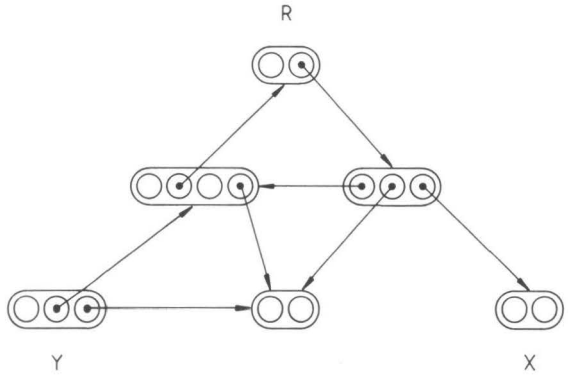


Figure 3.2

The concept of reachability for objects in G is defined by the following rules:

(1) The root R is reachable.
(2) If X is a reachable object,
$B \in branches(X)$,
$Y = obj(val(B))$,
then Y is reachable.
(3) An object is reachable on account of the above rules only.

For example in Figure 3.2, X is a reachable object and Y is an unreachable object.

The problem is to construct an algorithm which determines the set of all reachable objects. Such an algorithm is traditionally called a "marking algorithm". For the description of marking algorithms a variable set M of objects will be introduced. It is the job of a marking algorithm to establish the truth of the following assertion:

$$M = \{X \in G \mid X \text{ is reachable}\}.$$

It follows directly from the definition of reachability that this assertion is equivalent to the conjunction of the following three assertions:

(A1) $R \in M$.
(A2) $\forall X \in M \; \forall B \in branches(X) \; [obj(val(B)) \in M]$.
(A3) $\forall X \in M \; [X \text{ is reachable}]$.

The DSW-algorithm, which is a particular solution to the above problem, will now be derived in six "phases". In the initial phase (Phase 0) a simple algorithm is constructed, which serves as the starting point.

3.2.2. Phase 0: Getting started

Looking at the definition of reachability one sees that it is almost an algorithm itself. That is, if we start with $M = \{R\}$ and repeat the

following actions "long enough", M will finally become equal to the set of reachable objects:

```
Let X ∈ M.
If branches(X) ≠ ∅
| Let B ∈ branches(X).
| Let Y = obj(val(B)).
| M := M ∪ {Y}.
```

Here the operations "Let $X \in M$" and "Let $B \in branches(X)$" select an element from a set in a nondeterministic way. This nondeterminism can be thought of as being governed by a "demon". The first part of the derivation of the DSW-algorithm mainly consists of "exorcising" this demon, i.e. convert it to determinism.

The question is what "long enough" means. A marking algorithm should establish the truth of the assertions (A1), (A2) and (A3). The assertions (A1) and (A3) are initially true and are not affected by the above actions. Now one could say that "long enough" means: until assertion (A2) holds. The process need not stop exactly at the point where this assertion holds for the first time, however (most known marking algorithms do not). Any point beyond this point will do as a termination point. In order to model this the following nondeterministic construct will be introduced:

```
Beyond A
| S.
```

where S is a series of actions and A is an assertion. It prescribes that S must be repeated until some (but not necessarily the first) point where A holds. Note that prior to an execution of S, the assertion $\neg A$ need not necessarily hold. The termination point is supposed to be chosen nondeterministically by the demon.

The above construct turns out to be very useful in the derivation of algorithms. From an algorithm containing this construct a new algorithm can be derived by replacing the assertion A by another assertion B which is a sufficient condition for A, i.e. $B \Rightarrow A$. If the old algorithm was partially correct, so will the new one. Neither of the algorithms needs to terminate, however. The termination of any algorithm containing the above construct will depend on the nature of the demon. The demon could for instance be "unfair" and refuse to choose a termination point even if the termination condition holds after each iteration. This can be prevented by replacing the above construct by the deterministic construct:

```
Until A
| S.
```

which prescribes zero or more repetitions of S until A holds for the first time. Note that prior to an execution of S the assertion $\neg A$ will now hold.

As indicated above, the nondeterministic algorithms considered here need not terminate. Therefore some people may not call them algorithms at all, but here we will. Nondeterministic algorithms are viewed here as "abstractions" of (more) deterministic algorithms. The demon represents the part of these abstract algorithms which has been "abstracted away". Certain terminating and non-terminating algorithms have the same abstraction. So in the inverse process of abstraction, i.e. the derivation of algorithms, it is often possible to derive both terminating and non-terminating algorithms from nondeterministic algorithms. This also applies to the following

nondeterministic algorithm which will be chosen as a starting point for the derivation of the DSW-algorithm:

```
ALGORITHM 1
M := {R}.
Beyond ∀ X ∈ M ∀ B ∈ branches(X) [obj(val(B)) ∈ M]
| Let X ∈ M.
| If branches(X) ≠ ∅
| | Let B ∈ branches(X).
| | Let Y = obj(val(B)).
| | M := M ∪ {Y}.
```

The (partial) correctness of this algorithm should be obvious. It can be established formally by proving that (A1) and (A3) hold immediately before the loop and are kept invariant by the loop body. Assertions which satisfy the latter properties will (as usual) be referred to as "invariants". So for Algorithm 1 we have:

```
INVARIANTS
(1.1) R ∈ M.
(1.2) ∀ X ∈ M [X is reachable].
```

In the sequel the actions occurring in the body of the loop will be referred to as indicated below:

```
Let X ∈ M.                          ⟶ visiting X
If branches(X) ≠ ∅
| Let B ∈ branches(X).              ⟶ tracing B
| Let Y = obj(val(B)).
| M := M ∪ {Y}.                     ⟶ marking Y
```

All following subsections (except Subsection 3.2.5) will be divided into four parts, each of which corresponds to one of the four transformation steps described in Section 3.1. The next two derivation phases will consist of limiting the freedom of the demon in such a way, that even though the algorithm remains nondeterministic, termination is guaranteed.

3.2.3. Phase 1: Restricting the tracing of branches

Even if in Algorithm 1 the beyond-construct was replaced by an until-construct, the algorithm need not terminate. The reason is that there is too much freedom in the choice of objects to be visited and branches to be traced. The demon could for instance choose the same object and the same branch in each iteration of the loop. Consequently the termination condition would never hold (except in trivial cases). Our primary concern will therefore be to impose restrictions on the visiting of objects and tracing of branches in such a way, that the termination condition of Algorithm 1 will hold in a finite number of iterations.

What are reasonable restrictions? A general reasonable restriction which may be imposed on an algorithm is, that it should not do the same thing twice if once is enough. Let us apply this principle to the tracing of branches first. It is easy to see that it makes no sense to trace a branch more than once in Algorithm 1. The second time a branch B would be traced, the object Y referred to by the value of B would already have been marked. So the following restriction is reasonable:

RESTRICTION 1

A branch may be traced only once.

We will now transform Algorithm 1 in such a way that this restriction is met.

3.2.3.1. Step 1

The enforcement of Restriction 1 introduces a certain overhead. The demon must be prevented from selecting a branch which has already been traced. For that purpose a variable set $C(X)$ of branches of X will be associated with each object X with the following interpretation:

INTERPRETATION 1

For each object $X \in M$, $C(X)$ is equal to the set of branches of X which have not yet been traced.

This interpretation of C, which is of course strictly informal, can immediately be translated into a number of invariants for the algorithm to be derived (by adding C). First of all the obvious invariant:

INVARIANT 1.3

$\forall\ X \in M\ [C(X) \subset branches(X)]$.

Secondly, each branch of an object X which is not an element of $C(X)$ has already been traced. For each branch B which has been traced, the object referred to by the value of B has been marked. Consequently we have:

INVARIANT 1.4

$\forall\ X \in M\ \forall\ B \in branches(X) \setminus C(X)\ [obj(val(B)) \in M]$.

3.2.3.2. Step 2

Let us now insert assignments to C in Algorithm 1 according to Interpretation 1, thus making sure Invariants 1.3 and 1.4 hold. First of all $C(X)$ must be properly initialized for each object X. For the root this leads to:

ADDITION 1.1

$M := \{R\} \longrightarrow$
$\quad M, C(R) := \{R\}, branches(R)$

For all other objects Y, $C(Y)$ must be initialized to $branches(Y)$ as soon as Y is marked for the first time. Whether an object Y is marked for the first time can be determined by testing whether $Y \notin M$ prior to marking Y, resulting in:

ADDITION 1.2

$M := M \cup \{Y\} \longrightarrow$
$\quad$ If $Y \notin M$
$\quad$ | $C(Y) := branches(Y)$.
$\quad M := M \cup \{Y\}$

After tracing a branch B of an object X, B must be removed from $C(X)$. This can be accomplished by:

ADDITION 1.3

Let $B \in branches(X)$ $\longrightarrow$
    Let $B \in branches(X)$.
    $C(X) := C(X) \setminus \{B\}$

Note that $C(X)$ is well-defined here because $X \in M$. The above additions transform Algorithm 1 into Algorithm 1* for which besides Invariants 1.1 and 1.2 the additional Invariants 1.3 and 1.4 hold, as can easily be proved:

ALGORITHM 1*

$M, C(R) := \{R\}, branches(R)$.
Beyond $\forall X \in M \; \forall B \in branches(X)$ $[obj(val(B)) \in M]$
| Let $X \in M$.
| If $branches(X) \neq \emptyset$
| | Let $B \in branches(X)$.
| | $C(X) := C(X) \setminus \{B\}$.
| | Let $Y = obj(val(B))$.
| | If $Y \notin M$
| | | $C(Y) := branches(Y)$.
| | $M := M \cup \{Y\}$.

3.2.3.3. Step 3

In this step the invariants will be used to make replacements in Algorithm 1*. Among other things these replacements will be used to enforce Restriction 1. No variables will be made redundant. First, suppose an object X for which $C(X) = \emptyset$ is visited. All branches of X have then already been traced, and using Invariant 1.4 it can easily be seen that tracing a branch B of X has no effect whatsoever on M or C. Consequently tracing a branch B of an object X may be omitted if $C(X) = \emptyset$, which justifies the following replacement:

REPLACEMENT 1.1

$branches(X) \neq \emptyset$ $\longrightarrow$
    $C(X) \neq \emptyset$

Since we are now sure that $C(X) \neq \emptyset$, when selecting a branch B of X to be traced, B can just as well be selected from $C(X)$ (which is a subset of $branches(X)$ according to Invariant 1.3) instead of $branches(X)$:

REPLACEMENT 1.2

Let $B \in branches(X)$ $\longrightarrow$
    Let $B \in C(X)$

The above two replacements enforce Restriction 1. Two more replacements will be applied in order to "improve" Algorithm 1*.

Let us have a look at the termination condition of Algorithm 1* (i.e. assertion (A2)). It follows directly from Invariant 1.4 that this condition is implied by the simpler condition:

$\forall\ X \in M\ [C(X) = \emptyset]$.

The following replacement is therefore in order:

REPLACEMENT 1.3

$\forall\ X \in M\ \forall\ B \in branches(X)\ [obj(val(B)) \in M] \longrightarrow$
$\forall\ X \in M\ [C(X) = \emptyset]$

Finally, it is easy to see that marking an object Y makes sense only if $Y \notin M$. This leads to the following optimization:

REPLACEMENT 1.4

```
If Y ∉ M                          ⎤
| C(Y) := branches(Y).            ⎥ ⟼
M := M ∪ {Y}                      ⎥
     If Y ∉ M                     ⎥
     | M,C(Y) := M ∪ {Y},branches(Y) ⎦
```

This concludes the third step.

3.2.3.4. Step 4

In this step possible redundant variables are supposed to be removed. Since there are none, it suffices to give the final algorithm of this first transformation phase together with its invariants:

ALGORITHM 2

```
M,C(R) := {R},branches(R).
Beyond ∀ X ∈ M [C(X) = ∅]
| Let X ∈ M.
| If C(X) ≠ ∅
| | Let B ∈ C(X).
| | C(X) := C(X) \ {B}.
| | Let Y = obj(val(B)).
| | If Y ∉ M
| | | M,C(Y) := M ∪ {Y},branches(Y).
```

INVARIANTS

(2.1) $R \in M$.
(2.2) $\forall\ X \in M$ [X is reachable].
(2.3) $\forall\ X \in M\ [C(X) \subset branches(X)]$.
(2.4) $\forall\ X \in M\ \forall\ B \in branches(X) \setminus C(X)\ [obj(val(B)) \in M]$.

Note that only Invariant 2.4 is temporarily disturbed inside the loop.

3.2.4. Phase 2: Restricting the visiting of objects

In this phase restrictions will be imposed on the visiting of objects. Visiting an object X is useless if all branches of X have already been traced. A proper restriction would therefore be: only objects X with $C(X) \neq \emptyset$ may be visited. Since in Algorithm 2 at the beginning of a visit to an object X it is already checked whether $C(X) \neq \emptyset$, it is convenient to weaken this restriction a little and allow for one visit when $C(X) = \emptyset$. This extra visit can then be used to establish that $C(X) = \emptyset$ and take

measures to prevent X being visited again. The following restriction will therefore be imposed:

RESTRICTION 2

As soon as $C(X) = \emptyset$, X may be selected for a visit once, at most.

3.2.4.1. Step 1

Again the enforcement of the above restriction introduces a certain overhead. The demon must be prevented from selecting an object X for a visit for which $C(X) = \emptyset$ and which has already been visited (once) since $C(X) = \emptyset$. This will be accomplished through the introduction of a variable set U of marked objects. U has the following interpretation:

INTERPRETATION 2

U is equal to the set of all marked objects X for which either:
(1) $C(X) \neq \emptyset$, or
(2) $C(X) = \emptyset$ and X has not been selected for a visit since $C(X) = \emptyset$.

It follows immediately from this interpretation of U that the following invariant should hold:

INVARIANT 2.5

$U \subset M$.

Since for each marked object X, $X \notin U$ implies that $\neg(C(X) \neq \emptyset)$, we also have:

INVARIANT 2.6

$\forall X \in M \setminus U\ [C(X) = \emptyset]$.

3.2.4.2. Step 2

Assignments to U will now be added to Algorithm 2 according to Interpretation 2, so as to make Invariants 2.5 and 2.6 hold. First the initialization of U, which is obvious:

ADDITION 2.1

$M, C(R) := \{R\}, branches(R) \longrightarrow$
$\quad M, C(R), U := \{R\}, branches(R), \{R\}$

The first (and as will turn out the only) time an object is a candidate for addition to U is when the object is marked. At the moment an object X is marked (for the first and only time) in Algorithm 2, it clearly satisfies one of the two conditions specified in Interpretation 2. It should therefore be added to U:

ADDITION 2.2

$M, C(Y) := M \cup \{Y\}, branches(Y) \longrightarrow$
$\quad M, C(Y), U := M \cup \{Y\}, branches(Y), U \cup \{Y\}$

It follows from Interpretation 2 that an object X must be removed from U the first time it is selected for a visit when $C(X) = \emptyset$. This can be

accomplished by adding an else-part to the conditional clause "If $C(X) \neq \emptyset$ ..." in Algorithm 2:

```
ADDITION 2.3
If C(X) ≠ ∅ ]
| ...       ]→
      If C(X) ≠ ∅
      | ...
      else
      | U := U \ {X}
```

As soon as an object X is removed from U, $C(X) = \emptyset$ and will remain so. Hence X need never be added to U again. All provisions to keep track of U according to Interpretation 2 have thus been made. The additional Invariants 2.5 and 2.6 can easily be proved to hold for the algorithm obtained by applying the above additions to Algorithm 2:

```
ALGORITHM 2*
M,C(R),U := {R},branches(R),{R}.
Beyond ∀ X ∈ M [C(X) = ∅]
| Let X ∈ M.
| If C(X) ≠ ∅
| | Let B ∈ C(X).
| | C(X) := C(X) \ {B}.
| | Let Y = obj(val(B)).
| | If Y ∉ M
| | | M,C(Y),U := M ∪ {Y},branches(Y),U ∪ {Y}.
| else
| | U := U \ {X}.
```

3.2.4.3. Step 3

Replacements will now be made to enforce Restriction 2, using the additional information gathered in the variable U. At first sight Restriction 2 can easily be enforced by selecting an object X for a visit from U instead of M. This poses a little problem, however, because U may be empty. Therefore, first, provisions will be made to ensure that $U \neq \emptyset$ prior to an iteration of the loop.

Consider the termination condition of Algorithm 2*. It follows from Invariant 2.6 that this condition is implied by the condition:

$$U = \emptyset.$$

So the following replacement is allowed:

```
REPLACEMENT 2.1
∀ X ∈ M [C(X) = ∅] ⟶
      U = ∅
```

This replacement in itself is not enough to ensure that $U \neq \emptyset$ prior to an iteration of the loop. It is, if the beyond-construct is replaced by an until-construct:

REPLACEMENT 2.2

Beyond ⟶
    Until

Restriction 2 is now enforced by:

REPLACEMENT 2.3

Let $X \in M$ ⟶
    Let $X \in U$

3.2.4.4. Step 4

Again no redundant variables occur in the algorithm derived so far. The variables C and U have only been used to restrict nondeterminism and not to change the representation of other variables. The final algorithm of this transformation step (and consequently the entire transformation phase) is therefore equal to the final algorithm of the previous step:

ALGORITHM 3

$M, C(R), U$:= $\{R\}$, *branches(R)*, $\{R\}$.
Until $U = \emptyset$
| Let $X \in U$.
| If $C(X) \neq \emptyset$
| | Let $B \in C(X)$.
| | $C(X) := C(X) \setminus \{B\}$.
| | Let Y = *obj(val(B))*.
| | If $Y \notin M$
| | | $M, C(Y), U := M \cup \{Y\}$, *branches(Y)*, $U \cup \{Y\}$.
| else
| | $U := U \setminus \{X\}$.

INVARIANTS

(3.1) $R \in M$.
(3.2) $\forall X \in M$ [X is reachable].
(3.3) $\forall X \in M$ [$C(X) \subset$ *branches(X)*].
(3.4) $\forall X \in M \; \forall B \in$ *branches(X)* $\setminus C(X)$ [*obj(val(B))* $\in M$].
(3.5) $U \subset M$.
(3.6) $\forall X \in M \setminus U$ [$C(X) = \emptyset$].

3.2.5. Interlude: Termination

Having restrained the visiting of objects and tracing of branches drastically and having replaced the nondeterministic beyond-construct by the deterministic until-construct, Algorithm 3 may be expected to terminate irrespective of the nature of the (not yet fully exorcised) demon. This can be established more formally as follows. During each iteration of the loop in Algorithm 3 a marked object X is visited. If $C(X) \neq \emptyset$, a branch B of X is traced, which has not been traced before, according to Restriction 1. If $C(X) = \emptyset$, X is removed from U and will not be visited a next time according to Restriction 2. Hence the sum of the number of branches of marked objects, which have already been traced, and the number of marked objects which will not be visited again, will increase by one with each iteration of the loop. Translated into more formal terms this implies that the value of the following expression will increase by one

with each iteration of the loop:

$$\#(M \setminus U) + \Sigma\ X \in M\ [\#(branches(X) \setminus C(X))].$$

The fact that this is indeed so, can easily be verified. Because of the finiteness of the number of objects and branches, the value of this expression has a finite upper bound. Termination of Algorithm 3 is thereby guaranteed.

The fact that the value of the above expression increases by one with each iteration of the loop allows an even stronger statement on the termination of Algorithm 3. The initial value of the above expression is 1. At termination of Algorithm 3 $U = \emptyset$ and $C(X) = \emptyset$ for each $X \in M$. The final value of the expression is therefore:

$$\#Q + \Sigma\ X \in Q\ [\#branches(X)],$$

where Q is the set of reachable objects. Consequently, Algorithm 3 will terminate after the following number of iterations:

$$-1 + \Sigma\ X \in Q\ [1 + degree(X)].$$

Assuming that all "primitive" operations in Algorithm 3 take a constant time, the above implies that Algorithm 3 operates in a time which is linear in the number of reachable objects and the number of branches of reachable objects. This is as good as we can expect.

3.2.6. Phase 3: Changing the representation of C

In this phase and the following, the exorcising of the demon will be completed. The remaining places where the demon resides are the operations "Let $X \in U$" and "Let $B \in C(X)$". Here we shall consider the operation "Let $B \in C(X)$". The only operations which are performed on $C(X)$ are initialization, testing for equality to $\emptyset$, and selecting and immediately thereafter removing an element. The following restriction, which eliminates the demon from "Let $B \in C(X)$", is therefore enforceable:

RESTRICTION 3

Branches are selected and removed from $C(X)$ in the order of their numbering.

3.2.6.1. Step 1

Restriction 3 can be complied with by associating a variable counter $k(X)$ with each marked object X with the following interpretation:

INTERPRETATION 3

For each object $X \in M$, $k(X)$ is the number of the last branch which has been removed from $C(X)$. If no branches have been removed from $C(X)$ yet, $k(X) = 0$.

This interpretation implies that k must first of all satisfy the following invariant:

INVARIANT 3.7

$\forall\ X \in M\ [0 \leq k(X) \leq degree(X)]$.

Moreover, Restriction 3 together with Interpretations 1 and 3 imply that the following invariant should hold:

INVARIANT 3.8

$\forall\ X \in M\ [C(X) = \{branch(X,i)\ |\ k(X) < i \leq degree(X)\}]$.

3.2.6.2. Step 2

In this step assignments to k should be added in agreement with Interpretation 3 in order to make Invariants 3.7 and 3.8 hold. However, Invariant 3.8 cannot be made to hold without also making some replacements. The reason is that in contrast to the invariants derived before, Invariant 3.8 critically depends on the restriction of nondeterminism (Restriction 3) to be enforced. Invariant 3.8 can therefore only be made to hold by enforcing that restriction through a replacement first. This is an example where it is essential that a replacement is used before an addition.

Let us perform the additions and replacements required to make Invariants 3.7 and 3.8 hold now. The initialization of k, which should be done together with the initialization of C, is obvious and leads to the following additions:

ADDITION 3.1

$M, C(R), U := \{R\}, branches(R), \{R\} \longrightarrow$
$\quad M, C(R), U, k(R) := \{R\}, branches(R), \{R\}, 0$

ADDITION 3.2

$M, C(Y), U := M \cup \{Y\}, branches(Y), U \cup \{Y\} \longrightarrow$
$\quad M, C(Y), U, k(Y) := M \cup \{Y\}, branches(Y), U \cup \{Y\}, 0$

The only statement which disturbs Invariant 3.8 is "$C(X) := C(X) \setminus \{B\}$". An assignment to $k(X)$ should therefore be added to this statement. First we must make sure, however, that B is chosen according to Restriction 3, because otherwise it is impossible to restore Invariant 3.8. According to Interpretation 3 this can be done by choosing the $(k(X)+1)$-st branch of X instead of an arbitrary branch from $C(X)$ for B. It must be assumed, for that purpose, that Invariants 3.7 and 3.8 hold prior to "Let $B \in C(X)$". From these invariants and the fact that $C(X) \neq \emptyset$ can be derived that indeed $1 \leq k(X)+1 \leq degree(X)$ and $branch(X, k(X)+1) \in C(X)$:

REPLACEMENT 3.1

Let $B \in C(X) \longrightarrow$
$\quad$ Let $B = branch(X, k(X)+1)$

Invariant 3.8 is now restored by:

ADDITION 3.3

$C(X) := C(X) \setminus \{B\} \longrightarrow$
$\quad C(X), k(X) := C(X) \setminus \{B\}, k(X)+1$

The above is an example where it is convenient to use the new invariants for replacements before their truth has been established. Restriction 3 could have been satisfied without relying on Invariants 3.7

and 3.8 by not choosing the $(k(X)+1)$-st branch of X in Replacement 3.1, but the m-th branch of X, where $m = min\{i \mid branch(X,i) \in C(X)\}$. An extra replacement would then have been required (in the next step) to replace m by $k(X)+1$. The only thing that remains to be done is to prove that Invariants 3.7 and 3.8 indeed hold, which is left to the reader. This completes Step 2, in which Restriction 3 was enforced. The algorithm we have so far is:

```
ALGORITHM 3*
M,C(R),U,k(R) := {R},branches(R),{R},0.
Until U = ∅
| Let X ∈ U.
| If C(X) ≠ ∅
| | Let B = branch(X,k(X)+1).
| | C(X),k(X) := C(X) \ {B},k(X)+1.
| | Let Y = obj(val(B)).
| | If Y ∉ M
| | | M,C(Y),U,k(Y) := M ∪ {Y},branches(Y),U ∪ {Y},0.
| else
| | U := U \ {X}.
```

3.2.6.3. Step 3

In this step C will be turned into a redundant variable. The only place where the value of C is used in Algorithm 3* is in the test "$C(X) \neq \emptyset$". Invariants 3.7 and 3.8 imply that this test is equivalent to "$k(X) \neq degree(X)$", which results in the following replacement:

```
REPLACEMENT 3.2
C(X) ≠ ∅ ⟶
     k(X) ≠ degree(X)
```

C has now turned into a redundant variable, the assignments to which may be removed:

```
REMOVAL 3.1
C(X),k(X) := C(X) \ {B},k(X)+1 ⟶
     k(X) := k(X)+1

REMOVAL 3.2
M,C(Y),U,k(Y) := M ∪ {Y},branches(Y),U ∪ {Y},0 ⟶
     M,U,k(Y) := M ∪ {Y},U ∪ {Y},0

REMOVAL 3.3
M,C(R),U,k(R) := {R},branches(R),{R},0 ⟶
     M,U,k(R) := {R},{R},0
```

Finally the following optimizing replacement is made, the omission of which would be an eye-sore to any right-minded programmer:

REPLACEMENT 3.3

```
Let B = branch(X,k(X) + 1).          ⌋
k(X) := k(X) + 1                     ⌋⟶
     k(X) := k(X) + 1.
     Let B = branch(X,k(X))
```

3.2.6.4. Step 4

The variable C no longer occurs in the algorithm and may be disposed of. Yet C still occurs in the invariants. New (and preferably equivalent) invariants must be derived from these invariants. This is a straightforward matter. The final algorithm and the result of rewriting the invariants is:

ALGORITHM 4

```
M,U,k(R) := {R},{R},0.
Until U = ∅
| Let X ∈ U.
| If k(X) ≠ degree(X)
| | k(X) := k(X) + 1.
| | Let B = branch(X,k(X)).
| | Let Y = obj(val(B)).
| | If Y ∉ M
| | | M,U,k(Y) := M ∪ {Y},U ∪ {Y},0.
| else
| | U := U \ {X}.
```

INVARIANTS

(4.1) $R \in M$.
(4.2) $\forall X \in M$ [X is reachable].
(4.3) $\forall X \in M$ [$0 \leq k(X) \leq degree(X)$].
(4.4) $\forall X \in M \; \forall i = 1,\ldots,k(X)$ [$obj(val(branch(X,i))) \in M$].
(4.5) $U \subset M$.
(4.6) $\forall X \in M \setminus U$ [$k(X) = degree(X)$].

3.2.7. Phase 4: Changing the representation of U

Let us consider the operation "Let $X \in U$" now. Apart from this operation the only operations which are performed on U are adding an object Y (which is not yet in U) to U and removing the (arbitrarily chosen) object X from U. This makes the following a feasible restriction:

RESTRICTION 4

Objects are added to and removed from U in a last-in first-out manner.

The purpose of this restriction is, of course, to be able to implement U efficiently as a stack.

3.2.7.1. Step 1

Introduce a variable stack S of objects. This stack has the following obvious interpretation:

INTERPRETATION 4

S contains the objects in U in the order of their addition to U (the most recently added object at the top of S).

This interpretation of S implies the following invariant:

INVARIANT 4.7

If $S = \langle X_1,\ldots,X_n\rangle$ then $U = \{X_1,\ldots,X_n\}$.

Here $\langle X_1,\ldots,X_n\rangle$ is the stack containing the objects $X_1,\ldots,X_n$, where X_n is the top of the stack.

3.2.7.2. Step 2

Assignments to S should be added according to Restriction 4 and Interpretation 4, thereby establishing the truth of Invariant 4.7. As in the second step of the previous phase, this is not possible without making some replacements as well. All operations modifying U must be accompanied by operations modifying S. First of all S should be initialized together with U:

ADDITION 4.1

$M,U,k(R) := \{R\},\{R\},0 \longrightarrow$
$\quad M,U,k(R),S := \{R\},\{R\},0,\langle R\rangle$

The addition of an element to U should be accompanied by a "push" operation:

ADDITION 4.2

$M,U,k(Y) := M \cup \{Y\},U \cup \{Y\},0 \longrightarrow$
$\quad M,U,k(Y),S := M \cup \{Y\},U \cup \{Y\},0,PUSH(Y,S)$

The removal of an element from U (in "$U := U \setminus \{X\}$") poses a problem, because we can only remove an element from S if that element is at the top of S (through a "pop" operation). So we must make sure X is at the top of S. Invariant 4.7 implies that $TOP(S) \in U$, which justifies the following replacement:

REPLACEMENT 4.1

Let $X \in U \longrightarrow$
$\quad$ Let $X = TOP(S)$

X can now be popped from S:

ADDITION 4.3

$U := U \setminus \{X\} \longrightarrow$
$\quad U,S := U \setminus \{X\},POP(S)$

The above is the second example where it is convenient to use the new invariants for replacements before their truth has been established. As with the previous example, Restriction 4 could have been satisfied without using the new invariants, though in a highly artificial way. This can be seen as follows. Restrictions 3 and 4 make the algorithm entirely

deterministic. This implies that the order of visiting objects is predefined: It is the order of visiting objects in a depth-first search [TARJAN 72] of the graph of reachable objects. Consequently "*TOP(S)*" in Replacement 4.1 could have been replaced by (a more formal definition of) "the largest element of U in the depth-first numbering of reachable objects". Yet another way to avoid the use of Invariant 4.7 in Step 2 is to consider S temporarily as a sequence instead of a stack. If "*POP(S)*" in Addition 4.3 is replaced by "$S - \langle X \rangle$" (= deletion of X from S), then Replacement 4.1 can be moved to Step 3. After Replacement 4.1 has been performed it can then be observed that S is accessed stackwise only and hence S may be turned into a stack. This, however, is a trick which involves a sneaky change of data type. (The latter can be avoided by inserting an extra phase in the derivation, but that is also artificial.)

Notice that if we would perform Phase 4 before Phase 3, we would have an example where it is essential to use the new invariants before their truth has been established. Instead of being predefined the order of the objects in the stack S would then be determined by the nondeterministic order according to which branches are traced. Having lost the information on the tracing order of branches, it would not be possible in Replacement 4.1 (which would be Replacement 3.1 then) to replace "*TOP(S)*" by an expression such as "the largest element of U in the depth-first numbering of reachable objects" above. This shows that, unless we resort to the (retrospective) insertion of extra phases, it is in certain cases essential to use invariants before they are valid (as we contended in Section 3.1).

The conclusion of this step is to prove that Invariant 4.7 holds in the newly derived algorithm. Notice that this requires the proof of an additional invariant:

> INVARIANT 4.8
>
> All elements of S are different.

The combined proof of Invariants 4.7 and 4.8 is simple (use Invariant 4.5). Here is the final algorithm of this step:

```
ALGORITHM 4*
M,U,k(R),S := {R},{R},0,<R>.
Until U = ∅
| Let X = TOP(S).
| If k(X) ≠ degree(X)
| | k(X) := k(X) + 1.
| | Let B = branch(X,k(X)).
| | Let Y = obj(val(B)).
| | If Y ∉ M
| | | M,U,k(Y),S := M ∪ {Y},U ∪ {Y},0,PUSH(Y,S).
| else
| | U,S := U \ {X},POP(S).
```

3.2.7.3. Step 3

In this step the change of representation from U to S must be completed by turning U into a redundant variable and by subsequently removing all assignments to U. The value of U is used in Algorithm 4* only in the test "$U = \emptyset$". Invariant 4.7 implies that this test is equivalent to "$S = \langle\rangle$", where "$\langle\rangle$" is the empty stack:

REPLACEMENT 4.2

$U = \emptyset \longrightarrow$
$\quad S = <>$

U has become a redundant variable this way. All assignments to U may be removed:

REMOVAL 4.1

$U,S := U \setminus \{X\}, POP(S) \longrightarrow$
$\quad S := POP(S)$

REMOVAL 4.2

$M,U,k(Y),S := M \cup \{Y\}, U \cup \{Y\}, 0, PUSH(Y,S) \longrightarrow$
$\quad M,k(Y),S := M \cup \{Y\}, 0, PUSH(Y,S)$

REMOVAL 4.3

$M,U,k(R),S := \{R\},\{R\},0,<R> \longrightarrow$
$\quad M,k(R),S := \{R\},0,<R>$

3.2.7.4. Step 4

In this step the removal of U must be formally completed by eliminating U also from the invariants. As in the previous phase this is straightforward. The final algorithm of this phase together with the rewritten invariants is given below. For notational convenience the stack S is occasionally considered to denote the set of its elements in the invariants.

ALGORITHM 5

$M,k(R),S := \{R\},0,<R>$.
Until $S = <>$
| Let $X = TOP(S)$.
| If $k(X) \neq degree(X)$
| | $k(X) := k(X) + 1$.
| | Let $B = branch(X,k(X))$.
| | Let $Y = obj(val(B))$.
| | If $Y \notin M$
| | | $M,k(Y),S := M \cup \{Y\}, 0, PUSH(Y,S)$.
| else
| | $S := POP(S)$.

INVARIANTS

(5.1) $R \in M$.
(5.2) $\forall X \in M$ [X is reachable].
(5.3) $\forall X \in M$ [$0 \leq k(X) \leq degree(X)$].
(5.4) $\forall X \in M \; \forall i = 1,\ldots,k(X)$ [$obj(val(branch(X,i))) \in M$].
(5.5) $S \subset M$.
(5.6) $\forall X \in M \setminus S$ [$k(X) = degree(X)$].
(5.7) All elements of S are different.

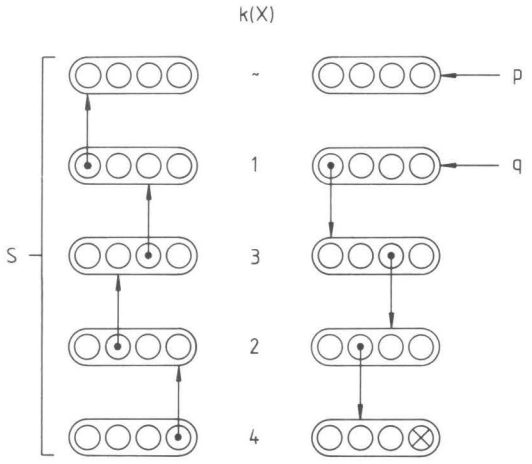
3.2.8. Phase 5: Changing the representation of S, or: the DSW-idea

In this phase the actual DSW-idea will be applied, which is in fact nothing but a change of representation. In contrast to the previous changes of representation (from C to k and U to S) this change of representation is not accompanied by a reduction of nondeterminism. This would be impossible in the first place, because, through the successive restrictions enforced in the previous phases, Algorithm 5 has turned into a completely deterministic algorithm. No "restrictions" will or can therefore be imposed in this phase.

In order to demonstrate the DSW-idea let us take a closer look at Algorithm 5. It is very easy to infer from Algorithm 5 that whenever there is an object X at the top of the stack S and an object Y is pushed on top of it, the $k(X)$-th branch of X contains a reference to Y. This makes S look as shown in Figure 3.3.a. (In this picture objects are assumed to be composed of exactly four branches.) It amounts to the following invariant which can easily be proved:

INVARIANT 5.8

If $S = \langle X_1,\ldots,X_n\rangle$, then:
(1) $\forall\ i = 1,\ldots,n-1\ [k(X_i) > 0]$.
(2) $\forall\ i = 1,\ldots,n-1\ [val(branch(X_i,k(X_i))) = ref(X_{i+1})]$.



3.3.a 3.3.b

Figure 3.3

The basic DSW-idea is that using two variable references p and q the situation of Figure 3.3.a can be transformed without loss of information into the situation of Figure 3.3.b. Here the cross in the fourth branch of the object at the bottom of the stack is the dummy reference nil (see Subsection 3.2.1). The situation of Figure 3.3.b has the advantage over the situation of Figure 3.3.a that the stack S has become redundant: All

operations on S can be expressed in terms of operations on the variables p and q and the contents of branches. Put another way: Figure 3.3.b sketches an implementation of S without any space overhead (apart from the two variable references p and q).

The application of the DSW-idea to Algorithm 5 raises a little problem. It is apparently assumed that the value of a component of an object is variable. Otherwise the transformation from Figure 3.3.a to Figure 3.3.b would never be possible. Up till now the value of a component of an object was assumed to be constant. Simply making the function *val* variable and adding modifications of *val* (according to Figure 3.3.b) to Algorithm 5 does not work, however, because these changes may affect the correctness of the algorithm. The solution, of course, is to introduce alongside the constant function *val* an extra variable function *VAL*, which is initially equal to *val*. Modifications of *VAL* may be added freely to Algorithm 5 because they in no way affect the correctness of the algorithm. Having added the variables p, q and *VAL* according to the DSW-idea to Algorithm 5, the job is then to eliminate the stack S *and* the function *val* from the algorithm (using the invariants). Finally, in order to show that *VAL* can just as well be replaced by *val* (made variable) it must be shown that the final value of *VAL* is equal to *val*.

3.2.8.1. Step 1

Let us now introduce the variables p, q and *VAL* according to the DSW-idea. Using Figure 3.3 as a guide this idea can be translated in the following invariant which the new algorithm should satisfy:

INVARIANT 5.9

Let $S = \langle X_1,\ldots,X_n\rangle$ and let $X_0 = X_{-1} = null$.
Let $V = \{branch(X_i,k(X_i)) \mid i = 1,\ldots,n-1\}$.
Then:
(1) $p = ref(X_n)$.
(2) $q = ref(X_{n-1})$.
(3) $\forall\ i = 1,\ldots,n-1\ [VAL(branch(X_i,k(X_i))) = ref(X_{i-1})]$.
(4) $\forall\ X \in G\ \forall\ C \in comp(X)\ [C \notin V \Rightarrow VAL(C) = val(C)]$.

Note that as implied by this invariant the situation where $S = \langle\rangle$ corresponds to $p = nil$, $q = nil$ and $VAL = val$.

3.2.8.2. Step 2

Assignments to the variables p, q and *VAL* must be added to Algorithm 5 in such a way that Invariant 5.9 is satisfied. First the variables should be initialized properly. *VAL* is implicitly assumed to be equal to *val* at the beginning of the algorithm. The initialization therefore amounts to:

ADDITION 5.1

$M,k(R),S := \{R\},0,\langle R\rangle \longrightarrow$
$\quad M,k(R),S,p,q := \{R\},0,\langle R\rangle,ref(R),nil$

Invariant 5.9 now holds initially. The only operations which disturb Invariant 5.9 are the operations which modify S: "S := *PUSH(Y,S)*" and "S := *POP(S)*". Consequently these operations should be accompanied by modifications of p, q and *VAL* in order to restore Invariant 5.9.

Consider the operation "S := $PUSH(Y,S)$" first. This operation makes Y the top element of S and X the subtop element. The set of branches V in Invariant 5.9 is therefore extended by this operation with $branch(X,k(X))$, which is denoted by B in Algorithm 5. This affects parts (1), (2) and (3) but not part (4) of Invariant 5.9. Part (1) can be restored by assigning to p the value $ref(Y)$, which is equal to $val(B)$. Part (4) of Invariant 5.9 implies, since $B \notin V$, that $val(B) = VAL(B)$. Part (1) can therefore be restored by assigning to p the value $VAL(B)$. Part (2) can be restored by assigning to q the value $ref(X)$, which is equal to p. Finally, part (3) can be restored by assigning to $VAL(B)$ the reference of the object "below" X in S, i.e. the value q. (Notice that this assignment to VAL does not affect part (4) of Invariant 5.9.) This leads to:

ADDITION 5.2

$M,k(Y),S := M \cup \{Y\},0,PUSH(Y,S) \longrightarrow$
$\quad M,k(Y),S,p,q,VAL(B) := M \cup \{Y\},0,PUSH(Y,S),VAL(B),p,q$

The operation "S := $POP(S)$" removes the object X at the top of S from S. In order to investigate the way this operation affects Invariant 5.9 two cases must be distinguished: the case where S contains a single object and the case where S contains two or more objects. Consider the former first. If S contains only one object the set V in Invariant 5.9 is empty and will be so after the operation "S := $POP(S)$". This implies that parts (3) and (4) of Invariant 5.9 are not affected. Part (2) is not affected either, because $ref(X_0) = ref(X_{-1}) = nil$. Only part (1) must be restored which can be done by assigning the value $ref(X_0) = nil$ to p. This covers the first case.

In the second case S contains two or more objects and consequently $V \neq \emptyset$. Let Y be the subtop element of S, i.e. the object referred to by q, and let $B = branch(Y,k(Y))$, then $B \in V$. The effect of the operation "S := $POP(S)$" on V is that B is removed from V. This does not affect part (3) of Invariant 5.9 (n decreases by one). It does affect parts (1), (2) and (4) though. Part (1) can be restored by assigning to p the value $ref(Y)$, which is equal to q. Part (2) can be restored by assigning to q the value $ref(Z)$, where Z is the (possibly imaginary) object below Y in S. Part (3) of Invariant 5.9 implies that $ref(Z) = VAL(branch(Y,k(Y))) = VAL(B)$. So part (2) can be restored by assigning the value $VAL(B)$ to q. Only part (4) remains. This part of the invariant is disturbed because B is removed from V and the assertion $VAL(B) = val(B)$ is not guaranteed to hold. As a consequence, part (4) can be restored by assigning the value $val(B)$ to $VAL(B)$. (Notice that this does not affect part (3) of Invariant 5.9.) According to part (2) of Invariant 5.8, $val(B) = val(branch(Y,k(Y))) = ref(X) = p$. So part (3) of Invariant 5.9 can be restored by assigning the value p to $VAL(B)$.

Immediately before the operation "S := $POP(S)$" in Algorithm 5 the assertion $S \neq <>$ holds. This implies that the distinction between the two cases considered above can be made by testing whether $q = nil$ or not (see Invariant 5.9). All in all this amounts to:

ADDITION 5.3

```
S := POP(S) ⟶
     If q = nil
     | S,p := POP(S),nil.
     else
     | Let Y = obj(q).
     | Let B = branch(Y,k(Y)).
     | S,p,q,VAL(B) := POP(S),q,VAL(B),p.
```

The algorithm obtained through the above additions to Algorithm 5 is given below. Though we made sure Invariant 5.9 is satisfied (not only as a loop invariant, but "everywhere"), a formal proof is still required. This proof will be obvious now and is omitted.

ALGORITHM 5*

```
M,k(R),S,p,q := {R},0,<R>,ref(R),nil.
Until S = <>
| Let X = TOP(S).
| If k(X) ≠ degree(X)
| | k(X) := k(X) + 1.
| | Let B = branch(X,k(X)).
| | Let Y = obj(val(B)).
| | If Y ∉ M
| | | M,k(Y),S,p,q,VAL(B) := M ∪ {Y},0,PUSH(Y,S),VAL(B),p,q.
| else
| | If q = nil
| | | S,p := POP(S),nil.
| | else
| | | Let Y = obj(q).
| | | Let B = branch(Y,k(Y)).
| | | S,p,q,VAL(B) := POP(S),q,VAL(B),p.
```

Before removing S it should be proved that the effect of the algorithm on VAL is nil. In other words, it must be proved that the postcondition $VAL = val$ holds. Proof: At termination of the algorithm $S = <>$, which implies that $V = \emptyset$ in Invariant 5.9. According to part (4) of Invariant 5.9 this implies that $VAL = val$.

3.2.8.3. Step 3

In this step the invariants will be applied so as to eliminate S and val from Algorithm 5* through replacements. Invariant 5.9 part (1) implies that the assertion $S = <>$ is equivalent to $p = nil$, which results in:

REPLACEMENT 5.1

```
S = <> ⟶
     p = nil
```

Invariant 5.9 part (1) also implies that, if $S \neq <>$, then $TOP(S) = obj(p)$. This gives us:

REPLACEMENT 5.2

```
Let X = TOP(S) ⟶
     Let X = obj(p)
```

Immediately after the statement "Let B = $branch(X,k(X))$" the assertion $B \notin V$ holds. From part (4) of Invariant 5.9 (which also holds there) can be inferred that this implies that $val(B) = VAL(B)$, which justifies:

REPLACEMENT 5.3

Let $Y = obj(val(B)) \longrightarrow$
    Let $Y = obj(VAL(B))$

The application of the above replacements transform Algorithm 5* into an algorithm in which val no longer occurs and in which S has become a redundant variable. The assignments to S can now be removed:

REMOVAL 5.1

$S,p,q,VAL(B) := POP(S),q,VAL(B),p \longrightarrow$
    $p,q,VAL(B) := q,VAL(B),p$

REMOVAL 5.2

$S,p := POP(S),nil \longrightarrow$
    $p := nil$

REMOVAL 5.3

$M,k(Y),S,p,q,VAL(B) := M \cup \{Y\},0,PUSH(Y,S),VAL(B),p,q \longrightarrow$
    $M,k(Y),p,q,VAL(B) := M \cup \{Y\},0,VAL(B),p,q$

REMOVAL 5.4

$M,k(R),S,p,q := \{R\},0,<R>,ref(R),nil \longrightarrow$
    $M,k(R),p,q := \{R\},0,ref(R),nil$

3.2.8.4. Step 4

In this step S will be removed from the invariants. Though in the previous steps the constant function val was removed from the algorithm together with S, this function need (and should) not be removed from the invariants (val is part of the problem specification). In contrast to the previous two phases the rewriting of the invariants containing S, so as to eliminate S, is far from obvious. Therefore the invariants will not be rewritten and an existential quantifier will be used to "eliminate" S. The final algorithm of this phase and of the entire derivation, the DSW-algorithm, is given below together with its invariants, pre- and postconditions. Strictly speaking the invariants are superfluous now, but they could be used for an independent proof of correctness, if desired.

ALGORITHM 6 (Deutsch-Schorr-Waite)

```
M,k(R),p,q := {R},0,ref(R),nil.
Until p = nil
| Let X = obj(p).
| If k(X) ≠ degree(X)
| | k(X) := k(X) + 1.
| | Let B = branch(X,k(X)).
| | Let Y = obj(VAL(B)).
| | If Y ∉ M
| | | M,k(Y),p,q,VAL(B) := M ∪ {Y},0,VAL(B),p,q.
| else
| | If q = nil
| | | p := nil.
| | else
| | | Let Y = obj(q).
| | | Let B = branch(Y,k(Y)).
| | | p,q,VAL(B) := q,VAL(B),p.
```

PRECONDITIONS

(6.1) $VAL = val$.

INVARIANTS

(6.1) $R \in M$.
(6.2) $\forall X \in M$ [X is reachable].
(6.3) $\forall X \in M$ [$0 \leq k(X) \leq degree(X)$].
(6.4) $\forall X \in M \; \forall i = 1,\ldots,k(X)$ [$obj(val(branch(X,i))) \in M$].
(6.5) There is a stack of objects $S = \langle X_1,\ldots,X_n \rangle$ such that:
(6.5.1) $S \subset M$.
(6.5.2) $\forall X \in M \setminus S$ [$k(X) = degree(X)$].
(6.5.3) All elements of S are different.
(6.5.4) $\forall i = 1,\ldots,n-1$ [$k(X_i) > 0$].
(6.5.5) $\forall i = 1,\ldots,n-1$ [$val(branch(X_i,k(X_i))) = ref(X_{i+1})$].
(6.5.6) Let $X_0 = X_{-1} = null$.
Let $V = \{branch(X_i,k(X_i)) \mid i = 1,\ldots,n-1\}$.
Then:
(6.5.6.1) $p = ref(X_n)$.
(6.5.6.2) $q = ref(X_{n-1})$.
(6.5.6.3) $\forall i = 1,\ldots,n-1$ [$VAL(branch(X_i,k(X_i))) = ref(X_{i-1})$].
(6.5.6.4) $\forall X \in G \; \forall C \in comp(X)$ [$C \notin V \Rightarrow VAL(C) = val(C)$].

POSTCONDITIONS

(6.1) $M = \{X \in G \mid X$ is reachable$\}$.
(6.2) $VAL = val$.

3.3. CONCLUSION

There are three different ways to look at the method of implementing algorithms described and demonstrated in this chapter. The first is from the point of view of algorithm verification. The method provides a simple way to prove the correctness of global transformations which amount to changes of data representation. The correctness of such a transformation is proved by decomposing the transformation into a sequence of simple and evidently correct transformations. No comprehensive "catalogue" of transformation rules as in [GERHART 75] is required, nor the use of an "abstraction function" as in [HOARE 72]. The method is also very flexible in that it allows very complex changes of representation (such as the DSW-transformation) to be proved correct without the need for enhanced verification techniques.

In connection with the above it is interesting to compare the correctness proof of the DSW-algorithm given here with other proofs of correctness of the DSW-algorithm [DE ROEVER 78], [DUNCAN & YELOWITZ 79], [GERHART 79], [GRIES 79], [KOWALTOWSKI 79], [TOPOR 79], [DERSHOWITZ 80]. The first thing to be noted is that all of the latter (except [DERSHOWITZ 80]) are proofs of more or less simplified versions of the DSW-algorithm instead of the general DSW-algorithm proved correct here. The second thing to be noted is that in [DE ROEVER 78], [GRIES 79], [KOWALTOWSKI 79], [TOPOR 79] the DSW-algorithm is considered as a given algorithm which is proved correct "independently". Here the DSW-algorithm is proved correct by proving a simple abstract algorithm correct and deriving the DSW-algorithm through a number of correctness-preserving transformations from this algorithm. In fact we proved the correctness of a number of algorithms (Algorithms 1-6). Consequently the proof given here is much longer than in the latter four references. We could have chosen Algorithm 5 (the stack algorithm) as the starting point, however. The length of the proof would then have been comparable to the length of an independent proof. The advantage of the approach pursued here is, that the correctness proof is "factorized", which makes it more suitable for human consumption. The only more or less similar approaches are [DERSHOWITZ 80], [DUNCAN & YELOWITZ 79], [GERHART 79]. In [DERSHOWITZ 80] the DSW-algorithm is derived in "Knuthian" style [KNUTH 74] from a recursive marking algorithm. The derivation has only two steps and the proofs are highly informal. In [DUNCAN & YELOWITZ 79], [GERHART 79] the DSW-algorithm is derived from an abstract algorithm. In the former "abstract/concrete mappings" are used to prove the correctness of transformations, while in the latter the successive algorithms and their assertions are supposed to be verified mechanically (see also [LEE et al. 79] for the outline of a proof using a catalogue of correctness-preserving transformations). Both proofs seem more complicated than the proof given here.

The second way to look at the implementation method described is from the point of view of algorithm construction. Can the method be of any help in the process of constructing (deriving) a new algorithm? It would not be entirely fair to judge this from the derivation of the DSW-algorithm given here. We knew beforehand what target we were aiming at and carefully directed the derivation process in order to hit that target. In constructing a new algorithm the target is unknown. Yet the derivation method described here is believed to be of help in deriving new algorithms as well. The first reason is that performing global transformations in a stepwise way aids in retaining or even gaining insight in the algorithm under development, which may lead to the discovery of new useful transformations. The second reason is that the algorithm constructor is

invited to try and perform a complex transformation, even if he has only some intuitive idea of it. He can cast his idea in a number of new variables and assertions on these variables, and start adding assignments to the variables and making replacements based on the assertions. If he does not achieve what he had in mind, too bad. If he does, he need only prove the assertions he postulated and remove whatever variables he made redundant.

The third way to consider the method described here is from the viewpoint of algorithm presentation. Presenting an algorithm by showing how it can be derived by a number of transformations from a simple algorithm can add significantly to understandability. This is an inherent advantage of the transformational method. It adds even more to understandability if not only the initial algorithm, but also all transformations applied to it are simple, as in the method described in this chapter. The transformational method in general is also very suitable for presenting classes of algorithms. Instead of walking straight ahead to the DSW-algorithm, we could have turned into several side-tracks in the derivation. If this is done in a systematic way, the entire class of marking algorithms can be discussed with a minimum of effort and a maximum of coherence. On a small scale and in a somewhat different context this was done in [DARLINGTON 78] for sorting algorithms. On a larger scale this will be done in Chapter 5 for garbage collection and compaction algorithms (though we shall take bigger steps than in the derivation of the DSW-algorithm).

Let us conclude this chapter with two remarks. First, as discussed in the introduction, the method can be used for the implementation of data structures as well. Secondly, the idea of first adding new variables and then removing redundant ones is also present in [MEERTENS 76] (but only in connection with deterministic algorithms).

CHAPTER 4

A STORAGE MANAGEMENT MODEL

4.0. INTRODUCTION

4.0.1. Mathematics and systematics

An important branch of computer science is the field of programming language implementation. The latter is usually divided into a number of "subjects" such as lexical analysis, parsing, bookkeeping, code generation, code optimization, register allocation, storage management. Some of these subjects (e.g., parsing) have evolved to systematic disciplines, which are even referred to as "theories". Other subjects (e.g., storage management) seem to escape any attempt at systematization. They constitute a more or less incoherent collection of techniques, a situation which is clearly reflected in publications on these subjects.

One may wonder what the reason for the above discrepancy is. The answer appears to be simple. A subject can only truly be systematized if it is amenable to mathematical treatment. Clearly, a subject such as parsing is. There exists a simple mathematical model (the "grammar") through which the parsing problem can be handled in a systematic way. In the terminology of Chapter 1 this can be formulated as follows. There is a simple non-trivial mathematical problem (the "basic parsing problem"), which is a proper abstraction of each problem in the following set:

$$V = \{P(L) \mid L \in \mathbb{L}\},$$

where

$P(L)$: the concrete parsing problem for the programming language L,
$\mathbb{L}$: the set of programming languages.

Now consider the following set of problems:

$$W = \{S(L,M) \mid L \in \mathbb{L},\ M \in \mathbb{M}\},$$

where

$S(L,M)$: the concrete storage management problem in an implementation of a programming language L on a machine M,
$\mathbb{L}$: the set of programming languages,
$\mathbb{M}$: the set of machines.

The general feeling seems to be that, in contrast to the set V, there does not exist a simple non-trivial mathematical problem which is a proper abstraction of each problem in W. In the absence of such a "basic storage management problem" a systematic treatment of the subject (as sketched in Chapter 1) is out of the question.

The core of the above problem lies not in computer science, but in mathematics. Traditionally, the objects studied in mathematics are static, immutable entities, the properties of which can be described by a relatively small number of axioms. The objects studied in parsing theory meet these qualifications wonderfully well. Storage management systems, which are the objects of study in storage management "theory", are dynamic entities with a relatively large number of (also dynamic) properties. These objects are rather hard to describe in the framework of traditional mathematics. They can be described very naturally, however, if we extend this framework with the proper concepts.

4.0.2. Dynamic systems

The concept we need is that of a "dynamic system". There are two aspects to this concept. First, a dynamic system is a "system", which means that it is composed of a number of "components". Each component is itself a dynamic system. Secondly, a dynamic system is "dynamic", which means that it can be altered by applying "operations" to it. The operations completely characterize the external "behaviour" of the system. A dynamic system can be described by specifying its components and expressing each operation in terms of operations on the components. This would all fit rather well into the framework of traditional mathematics, if it were not for the fact that the components of a dynamic system may (and generally will) be highly interrelated. An operation performed on a component of a dynamic system may therefore affect other components of that system. This is a rather uncommon situation in mathematics. The obvious way to model a system would be to define it as the tuple $<C_1,\ldots,C_n>$ of its components. However, when changing the i-th element of this tuple, one is not likely to expect that any of the other elements changes as well.

There are, of course, ways to overcome the above difficulties within the traditional mathematical framework. Yet, by doing so we threaten to fall into something like the "Turing Tarpit" [WULF 77]. The question is not *if*, but *how well* a dynamic system can be described in a certain framework. In the case of traditional mathematics the answer appears to be: not well enough. Since we want dynamic systems to be mathematically rigorous objects, we therefore have to develop a more powerful mathematical formalism.

4.0.3. Data structures

A first step to the development of such a formalism is to realize that a dynamic system is, in fact, nothing but a *data structure* which involves highly shared data. The components of a dynamic system correspond to a representation of the data structure, and the operations which can be applied to a dynamic system correspond to the operations of the data structure. The reason why dynamic systems have withstood a satisfactory mathematical treatment becomes apparent now. The major efforts in the field of data structure specification have been concerned with non-shared data. From a programming point of view sharing of data is often an undesirable situation indeed. In many applications (such as databases) sharing of data is a natural situation, however. Once we have the ability to specify data structures with sharing we gain tremendous descriptive power. In particular, this power can be used to describe dynamic systems in a mathematically rigorous way.

In Chapter 2 of this monograph we showed how, on the basis of the concept of a "structure", data structures involving sharing can be

specified as mathematical objects. This method can be used immediately for the description of dynamic systems. The language which will be used for the description of dynamic systems in this chapter will be based on this method, but it can be understood without having read Chapter 2. Though its semantics will be kept informal, it can be made completely rigorous, if need be, by the techniques of Chapter 2.

As an example of a dynamic system consider the concept of a "storage management system" as it will be defined in this Chapter:

> Each storage management system *H* has:
>
> - *root(H)*: constant object,
> - *graph(H)*: variable set of objects,
> - *store(H)*: constant set of cells,
> - *repr(H)*: constant mapping from values to words,
> - *alloc(H)*: variable mapping from objects to sets of cells.

Here objects and cells are dynamic systems "containing" values and words (which are also dynamic systems), respectively. The above definition implies that a storage management system *H* is a dynamic system with five components: *root(H)*, *graph(H)*, *store(H)*, *repr(H)* and *alloc(H)*. Besides these components, there are a number of operations associated with *H*, which can be expressed in terms of operations on the components of *H*. (The fact that a component such as *store(H)* is constant implies that operations may not alter the set of cells constituting *store(H)*. They may alter the contents of these cells, though.) There is for example an operation to initialize the system and an operation which amounts to the creation of a new object in the system (including the allocation of storage for the object). In fact, the storage management problem boils down to efficiently implementing the latter operation in the system.

4.0.4. System invariants

An important concept in relation to dynamic systems is the "system invariant". (The corresponding concept for data structures is usually called a "representation invariant".) A system invariant is an assertion about the components of a dynamic system, which will always hold "between" two operations performed on the system. It may, however, temporarily be disturbed "inside" (i.e., during the execution of) such an operation. Given descriptions of the operations which may be performed on a dynamic system in terms of operations on the components of the system, the system invariants are uniquely determined. When describing a dynamic system at the abstract level, it is often more convenient to give the system invariants first and then describe the operations which may be performed on the system, using statements such as "establish assertion *A* while not affecting the system invariants". This will also be the style of describing dynamic systems in the sequel.

A certain analogy between dynamic systems and ordinary mathematical systems can now be drawn. E.g., a storage management system can be viewed, in a way, as a 5-tuple $\langle root, graph, store, repr, alloc \rangle$, which satisfies a number of axioms (the system invariants). The analogy breaks down only because the elements of this tuple are variable and interdependent. E.g., if we change the contents of a cell in the *store*, the mapping *alloc* may also be affected. The analogy does not break down if we see it the other way around and regard the normal mathematical systems as (special cases of) dynamic systems. (This, however, should not be interpreted as an attempt to

turn mathematics into a branch of computer science!) The concept of a grammar from parsing theory could, for example, be described as a dynamic system in the following way:

Each grammar G has:

- $N(G)$: constant set of nonterminals,
- $\Sigma(G)$: constant set of terminals,
- $P(G)$: constant set of productions,
- $S(G)$: constant nonterminal.

The axioms which a grammar G satisfies correspond to system invariants such as $S(G) \in N(G)$, etc.. These invariants can of course never be violated since the entire system is constant. As with normal dynamic systems it is even conceivable to associate operations with a grammar. One can think for example of an operation to create (construct) a grammar and an operation which tests whether a given sequence of terminals is generated by the grammar. The parsing problem could then be defined as efficiently implementing this operation (where, for the sake of convenience, we ignore the fact that a parser should also produce a parse tree). Moreover, by making the components of a grammar variable, algorithms which transform the grammar (for example, into Greibach normal form [AHO & ULLMAN 72], if the grammar is context-free) can be described.

The above may tempt one to view computer science as the study of dynamic systems (a computer itself is a dynamic system). Elaborating on this would carry us a little too far away from the actual subject of this chapter. The essence of the above is that, just like there is a language of mathematics, there is also a language of computer science. In this language we can argue about dynamic systems just like mathematicians can argue about groups, vector spaces, etc.. This "language of computer science" is of course nothing new. It is used by almost any computer scientist when arguing about dynamic phenomena. The way it is used is usually rather informal. The important observation is that this language can be as exact as the language of mathematics.

4.0.5. The storage management system

Let us return to the subject of this chapter. In this chapter we shall describe a well-known dynamic system from the field of programming language implementation: the storage management system. The purpose is not to give a systematic treatment of storage management techniques. Such a treatment would take the size of a book. Instead, we shall focus our attention on one particular aspect of storage management, called "garbage collection". A systematic treatment of garbage collection algorithms will be given in the next chapter. Garbage collection is an internal operation of a storage management system. (I.e., the operation operates on the components of a storage management system, but is not "visible" externally.) In order to define the basic garbage collection problem we first have to define what a storage management system is. The purpose of this chapter is therefore to introduce the necessary concepts and set the stage for the discussion of the garbage collection problem and its solutions in the next chapter.

The above implies that we shall concentrate on the internal aspects of a storage management system and define it in terms of its components and system invariants. The external operations of a storage management system will only be discussed informally. We can afford to do so because we shall restrict ourselves to traditional garbage collection, which is an operation

performed in its entirety between two external operations of the storage management system. The correctness of this operation is guaranteed if and only if the operation does not violate any of the system invariants. So it suffices to consider the system invariants and ignore the external operations (which, ultimately, determine the system invariants). If we are to consider parallel ("on the fly") or "spread" garbage collection operations we would have to make the external operations explicit, because parts of the garbage collection operation must be inserted in the external operations.

4.0.6. The two layers of the model

The model of a storage management system which we shall present in this chapter consists of two layers, an abstract layer (containing the "root" and the "graph", which is composed of "objects") and a concrete layer (containing the "store", which is composed of "cells"). Furthermore, the model includes functions which map the abstract concepts onto the concrete concepts (the "representation function" and the "allocation function"). This may seem unnecessarily complex. It is rather usual to discuss storage management techniques directly in terms of operations on a machine store. Since, in the end, a storage management algorithm must operate exclusively on the machine store, it is indeed possible to describe such an algorithm in terms of operations on the machine store (take the machine code representation of the algorithm). There is no need to say that such a low level description will be far from readable. This is aggravated by the fact that these algorithms often use complicated tricks to save time and space, thus entirely obscuring the underlying abstract algorithms. Using the abstract layer of the model we are able to describe the algorithms in terms of their underlying abstract algorithms, thus abstracting from the implementation tricks and increasing readability. The implementation tricks can then be discussed separately, if necessary by going down to the store (using the representation and allocation function).

There is an even more convincing argument why a single layer approach to the description of storage management algorithms would not be appropriate. A storage management algorithm is part of the implementation of a programming language. This implies that the algorithm can be viewed as (part of) the implementation of an abstract operation from the programming language. In order to prove the correctness of this implementation it must be possible to argue about the abstract objects of the programming language and the way they are represented in the store. Therefore, without the abstract layer of the model and the "descent functions" a proof of correctness of a storage management algorithm (such as a garbage collector) would be impossible!

We shall start with an informal discussion of the subject of storage management in Section 4.1. In this somewhat philosophical section the basic storage management concepts are systematically introduced by tracing the process of implementing a programming language, starting with an abstract machine defining the language. The purpose of this section is not only to create some familiarity with the basic concepts, but also to provide a justification for the storage management model introduced in Section 4.2. This model anchors the intuitive notions discussed in Section 4.1 in unambiguous definitions. That is, the concept of a storage management system and its associated concepts are rigorously defined in Section 4.2. If desired, Section 4.1 can therefore be skipped. Some final remarks are made in Section 4.3.

4.1. INFORMAL DISCUSSION

4.1.1. Abstract machines

The starting point of each implementation of a programming language should be the definition of the semantics. There are many ways to define the semantics of a programming language. One of them is to describe an "abstract machine", which can directly execute programs in the language in question. The semantics of a program is defined by the actions of this machine when executing the program. ALGOL 68 [VAN WIJNGAARDEN et al. 76] is an example of a programming language, the semantics of which has indeed been defined this way. The actions of abstract machines can be divided into "external actions" and "internal actions". The external actions operate on an "environment" and can be observed by the user. The internal actions operate on an internal environment of the machine, the so-called "memory", and remain entirely hidden from the user (see Figure 4.1).

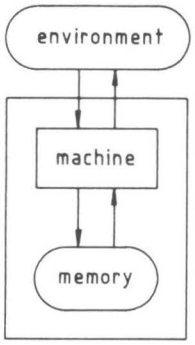


Figure 4.1

There exist many kinds of abstract machines, some less abstract than others. Broadly speaking, the implementation of a programming language will be more difficult the more abstract its defining abstract machine is. Let us therefore assume that we have a very abstract machine (what exactly we mean by this will become clear later). The memory of this abstract machine can be viewed as a collection of "objects". There are two kinds of objects, "atomic objects" and "structured objects". An atomic object has a "value", which is a primitive thing (e.g., an integer). A structured object X has a "structure", which is a set of objects called the "direct components" of X. A "component" of X is a direct component or a direct component of a component of X.

An object Y will be called a "subobject" of an object X if $X = Y$ or X is a structured object and Y is a component of X. Objects may arbitrarily share subobjects. An object may even be a component of itself. If two objects share a subobject they are said to "overlap". Objects will be pictured in the following way. An atomic object will be pictured as a circle, with the value of the object either omitted or pictured inside the circle. A structured object will be pictured as a circle with outgoing dotted arrows pointing to the pictures of the direct components of the

object.

EXAMPLE 4.1

Consider Figure 4.2.

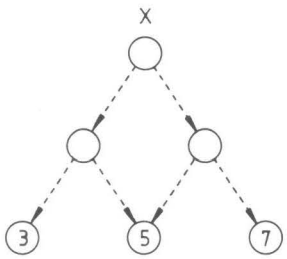


Figure 4.2

This figure shows a structured object *X* with two direct components, five components and six subobjects, three of which are atomic objects. The two direct components of *X* overlap: They share the atomic object with value *5*.

□

Note: The above concept of an object, though closely related, is not the same as the concept of an object defined in Chapter 2 (see Subsection 2.1.1).

4.1.2. Creation and modification of objects

There are basically two kinds of operations which the abstract machine can perform. First, it can "create" a new object (together with its subobjects). This operation amounts to extending the memory of the abstract machine with a new object, which may have arbitrary size. Secondly, the abstract machine can "modify" an existing object. For an atomic object this amounts to changing the value of the object. For a structured object it amounts to changing the structure of the object. The fact that the above are the only two kinds of operations performed by the abstract machine implies that objects are never removed from the memory. Consequently, the number of objects in the memory is nondecreasing.

Apart from a few "low level" programming languages, like BASIC, the number of objects which are created during the execution of a program on the corresponding abstract machine may be astronomical. In view of the hypothetical nature of abstract machines, the question where all these objects come from is merely of philosophical interest. A usual approach is to assume that the abstract machine has an inexhaustible supply of unused objects, from which one is picked each time an object is created. A more satisfactory answer to the question "where objects come from" can be found in Chapter 2. (Note: The abstract machine described in Section 2.3 has a memory from which objects can also disappear. In fact, they do so automatically once they are no longer used. This machine can easily be

redefined in such a way that all objects, once they are created, remain in memory for ever, thus conforming to the above view of abstract machines.)

4.1.3. Concrete machines

The above implies that an abstract machine has an effectively infinite memory, containing objects of the most varying sizes. How different is the "concrete machine" which the implementer of a programming language is faced with! This machine has a finite memory, containing a relatively small number of entirely identical atomic objects, called "cells". The value or "contents" of a cell is a "word", which is an integer in some predetermined range. Moreover, a unique integer is associated with each cell, called the "address" of the cell. The set of all addresses of cells in the memory usually constitutes a subrange of the integers (say from ℓ to r). The memory of the concrete machine can therefore be viewed as a row of consecutive cells (see Figure 4.3).

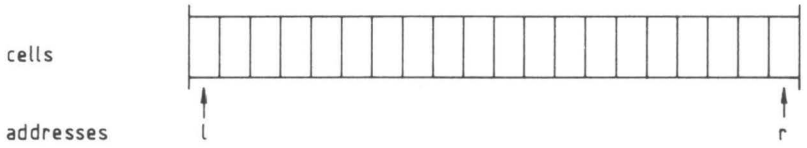


Figure 4.3

It is the duty of the implementer to make the concrete machine, as far as its external behaviour is concerned, behave like the abstract machine associated with the programming language L to be implemented. The user who offers a program in L to the concrete machine will then see this machine behave as prescribed by the semantics of L (i.e., if the implementation is correct).

Although the external behaviour of the concrete machine during the execution of a program in the implemented language is entirely prescribed, the implementer is free to choose the internal behaviour of the concrete machine: The user cannot observe anything of what is going on inside the machine. In view of the widely differing structure of the memories of the abstract and concrete machine, this freedom is indispensable. It enables the implementer to represent the objects and operations of the abstract machine by objects and operations of the concrete machine in whatever way he sees fit.

4.1.4. The allocation invariants

In implementations the most complicated representations are used. Simplified somewhat, they all amount to the following. An object from the memory of the abstract machine is represented by a set of cells, a so-called "location". The location representing the object X will be denoted by *alloc(X)*. We shall assume that an atomic object is represented by a single cell. (This is in no way essential, but it makes the discussion easier.) The value of an atomic object is represented by a word (the contents of a cell). The word representing the value V will be denoted by

repr(V). The operations of the abstract machine are represented by operations of the concrete machine in such a way that the following "allocation invariants" are not disturbed. In these invariants *val(X)* denotes the value of the atomic object *X* and *cont(C)* denotes the contents of the cell *C*:

Allocation Invariants

(A1) If *X* is an atomic object
then $cont(alloc(X)) = repr(val(X))$.
(A2) If *X* is a subobject of *Y*
then $alloc(X) \subset alloc(Y)$.
(A3) If *X* and *Y* do not overlap
then $alloc(X) \cap alloc(Y) = \emptyset$.

In (A1) an automatic conversion from a location $\{C\}$ containing a single cell *C* to the cell *C* is implicitly assumed. The above implies that running a program on the concrete machine can be viewed as running it on the abstract machine while keeping the allocation invariants valid.

The above is a simplification in many respects. For certain abstract machines (such as the one described in Section 2.3) efficient implementations would be out of the question if we really had to stick to the allocation invariants. We shall solve this, not by reformulating the allocation invariants, but by modifying the abstract machine in such a way that the allocation invariants can be satisfied efficiently. The modified abstract machine can be viewed as an "intermediate machine" placed between the abstract and the concrete machine. This intermediate machine is less abstract than, although still rather close to, the abstract machine. In particular it is very close to the traditional abstract machines used in definitions of programming languages such as ALGOL 68. Therefore we shall identify this intermediate machine with the abstract machine.

In the next subsections we shall discuss which modifications are necessary to be able to represent the operations of the abstract machine efficiently by operations of the concrete machine, according to the allocation invariants. The imaginary authority which guards over the allocation invariants during the execution of a program will be called the "storage manager". It is obvious that the storage manager will need some kind of bookkeeping. This bookkeeping contains the necessary information concerning the locations of objects. So, broadly speaking, it corresponds to the "allocation function" *alloc*. The efficiency of the implementation depends for a considerable part on the efficiency of this bookkeeping.

4.1.5. Modifying objects

As stated above, the abstract machine can perform two kinds of operations: creation and modification. Consider modification first. The modification of the value of an atomic object *X* can only disturb Invariant (A1). This invariant can be restored by modifying the contents of the cell *alloc(X)* accordingly, which does not require any updating of the bookkeeping of the storage manager. (Notice that due to Invariant (A3) changing the contents of the cell *alloc(X)* does not affect Invariant (A1) for other atomic objects.)

The situation is less simple for the modification of the structure of a structured object, which may disturb both Invariant (A2) and (A3). These invariants can only be restored by "reallocating" a number of objects in the memory of the concrete machine, i.e., by assigning new locations to

these objects in accordance with Invariants (A2) and (A3). This can, in principle, be done without the need to "move" atomic objects. Expensive "copy" operations (so as to restore Invariant (A1)) can thus be avoided, but the price to be paid for this is a more complex updating operation of the bookkeeping of the storage manager.

EXAMPLE 4.2

Consider the objects V, W, X, Y and Z in Figure 4.4.

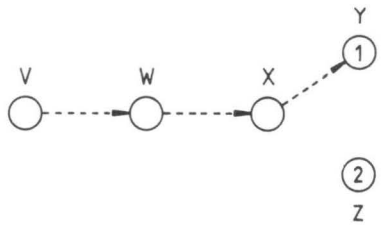


Figure 4.4

Suppose these objects have been assigned the following locations (a, b, c, d and e denote different cells):

$$alloc(V) = \{a,b,c,d\},$$
$$alloc(W) = \{b,c,d\},$$
$$alloc(X) = \{c,d\},$$
$$alloc(Y) = \{d\},$$
$$alloc(Z) = \{e\}.$$

Let $cont(d) = repr(1)$ and $cont(e) = repr(2)$. Check that the allocation invariants hold. Suppose the structure of the object X is changed, resulting in the situation of Figure 4.5.

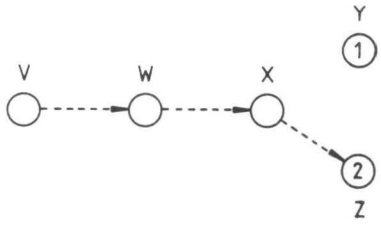


Figure 4.5

The allocation invariants can be restored in basically two ways. The first is to update *alloc* (the bookkeeping of the storage manager) as follows:

$$alloc(V) := \{a,b,c,e\}.$$
$$alloc(W) := \{b,c,e\}.$$
$$alloc(X) := \{c,e\}.$$

This requires no copying of the contents of cells, in contrast to the second way of restoring the allocation invariants:

$$alloc(Y) := \{e\}.$$
$$alloc(Z) := \{d\}.$$
$$cont(d) := repr(2).$$
$$cont(e) := repr(1).$$

□

4.1.6. The reference

The above implies that there is a trade-off between the efficiency of updating the bookkeeping of the storage manager and the amount of copying to be done. No matter which choice is made, however, frequent modifications of the structure of a complex object would make any program run like a snail. Fortunately, there is a way out: the "reference". The idea is to represent a component Y of an object X not by the object Y itself, but by an atomic object Y' with a value that represents the object Y uniquely: the reference of Y. Changing the structure of X, i.e., replacing Y as a component of X by another object Z, then amounts to replacing the value of Y' by the reference of Z. This is a matter of modifying the value of an atomic object, which, as we already saw, can be implemented efficiently. In pictures a reference to an object X will be represented by an unbroken arrow pointing to the picture of X.

EXAMPLE 4.3

Using references the objects from Figure 4.4 could be represented as follows:

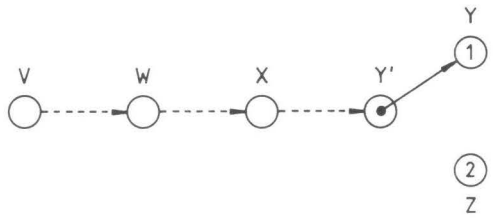


Figure 4.6

Changing the structure of the object X as in Example 4.2 amounts to changing the value of Y' as follows:

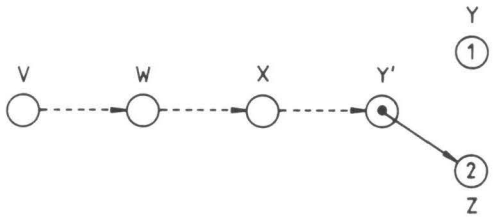


Figure 4.7

In contrast to Example 4.2 this disturbs Invariant (A1) only, which can be remedied by modifying the contents of the cell *alloc(Y')* in the appropriate way. □

It may be tempting to represent each direct component of a structured object as an atomic object with a reference as its value. This is certainly conceivable, but there is a limit, since this way of representing objects will make certain operations less efficient (in particular "random access" operations and copy operations on large objects). In practice, therefore, the choice will be somewhere in the middle (though there are extremes, compare for example arbitrary LISP and BASIC implementations). Anyway, we shall assume that the choice has already been made for us. This implies that we introduce the reference concept in the abstract machine by associating a reference as a unique value with each object. The reference of an object is said to "refer" to that object and an atomic object having a reference as its value will be called a "reference object".

Notice that the fact that an object *X* has a reference object as its component with a value referring to an object *Y* does not imply that *Y* is a component of *X*: A reference is just an ordinary value. At the more abstract level, which we have just abandoned, *Y* would be considered as a component of *X*, however. Notice also that efficiency is the only reason for introducing references. The allocation invariants can be satisfied without the introduction of references, even if circularities in objects occur. (Note that two objects which are mutual components must be assigned identical locations then.) This shows that the reference is truly an implementation concept. It is basically an "address" in abstract disguise, which need not occur in defining abstract machines of programming languages. The fact that it does occur in many such abstract machines (such as the hypothetical ALGOL 68 computer) is caused by the fact that many programming languages have the reference built in as a language concept.

4.1.7. Elimination of structural circularities

Now that we have introduced references, it is reasonable to forbid circularities in the structure of objects. These circularities can most conveniently be modelled using references.

EXAMPLE 4.4

Consider the object X from Figure 4.8, which has two circular direct components:

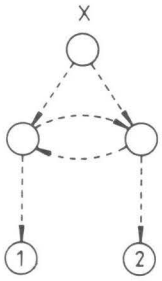


Figure 4.8

This object can be represented without structural circularities as follows:

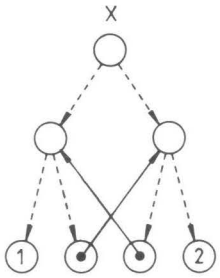


Figure 4.9

□

The above implies that the relation of "being a subobject of" will from now on be considered as a (reflexive) partial order. We could go even further by requiring that the relation of "being a direct component of" constitutes a tree structure (or better, a "forest structure") on the set of objects (as in Figure 4.9), thus ensuring that the direct components of an object do not overlap. This, however, would go too far:

EXAMPLE 4.5

Consider the object X from Figure 4.10.

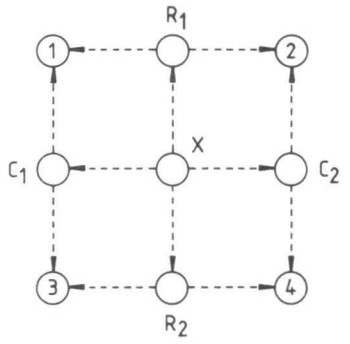


Figure 4.10

This object represents a matrix (a 2-dimensional array) with value $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, where R_1 and R_2 are the rows and C_1 and C_2 are the columns of the matrix. Clearly X has no tree structure. Representing X as a tree structure using references (so-called "edge vectors" [AHO & ULLMAN 77]) is not desirable in many implementations either, because it may slow down either row access, column access or both, in an unacceptable way. □

The fact that the relation of "being a subobject of" is a partial order makes this relation correspond to the more earthly relation of "being physically included in". This will be used to picture objects in a different way from now on. Instead of circles they will be denoted by arbitrary closed curves, and physical inclusion instead of dotted arrows will be used to denote the relation of "being a subobject of".

EXAMPLE 4.6

The objects pictured in Figures 4.2, 4.9 and 4.10 are now pictured as follows:

Figure 4.2:

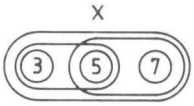


Figure 4.9:

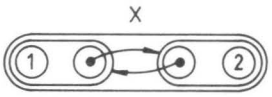


Figure 4.10:

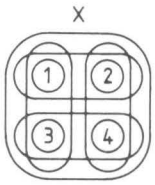


Figure 4.11

□

4.1.8. Accessing objects

Before turning to the creation of objects, it is useful to discuss in greater detail how objects are manipulated by the abstract machine. The only way for the abstract machine to manipulate, or "use", an object is by "accessing" it. The machine has two mechanisms for accessing objects. First, given a structured object *X*, the machine can access a direct component of *X*. This operation is called "selection". Secondly, given a reference object *Y*, the machine can access the object referred to by the value of *Y*. This operation is called "dereferencing". By repeatedly applying selection and dereferencing operations, the machine (controlled by the program) can follow "access paths" (see Figure 4.12).

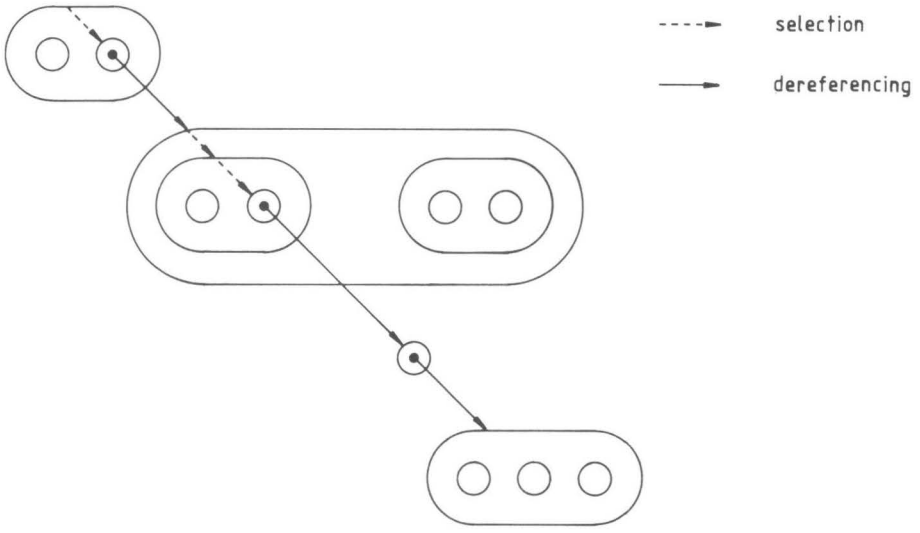


Figure 4.12

In order to be able to access even a single object this way, "starting points" are required: There must be objects which are a priori accessible to the abstract machine, i.e., without having to follow an access path first. Without loss of generality we may assume there is one such object, called the "root".

4.1.9. Creating objects

The creation of an object must be accompanied by "allocating" a location to the object, i.e., assigning a location to the object in accordance with the allocation invariants. If the new object contains old objects as its components, this location may be scattered all over the memory of the concrete machine, which can make the bookkeeping of the storage manager extremely complex. It is reasonable, therefore, to require that newly created objects do not contain any of the old objects as their components. This requirement, as a matter of fact, is only reasonable because we have references: Old components of new objects can always be represented by reference objects.

The above implies that objects are created in units which we shall call "nodes". These units of creation can be discerned in each implementation, where they have names such as "blocks", "activation records", "data areas", etc.. Creating a node amounts to creating an object (the node itself) together with all of its components. Since both the node and its components are new, none of the objects thus created will overlap with any of the old objects. However, by modifying the structure of a node after it has been created old objects can in principle still be contained as components in a node. A rigorous way to avoid this is to forbid the structure of a node (and consequently, the structure of any object) to be modified. This solution is not always preferable (cf. variant records in PASCAL). A more liberal solution is to allow only internal modifications of the structure of a node, i.e., modifications involving subobjects of the node only. The latter solution, which is adopted in the majority of implementations, will also be adopted here.

The memory of the (ever less abstract) abstract machine can now be viewed as a collection of non-overlapping nodes, where each object is a subobject of precisely one node. In order to distinguish the memory of the abstract machine from the memory of the concrete machine we shall call the former the "graph" and the latter the "store". An example of what the graph may look like is given in Figure 4.13, which pictures a graph with six nodes. The root, which we shall always assume to be a node, is indicated by *R*. This picture shows that the memory of the abstract machine can indeed be viewed as a graph, albeit a rather unusual one.

Due to the fact that nodes are non-overlapping we are able to impose the requirement that nodes be assigned "compact" locations, i.e., locations consisting of consecutive cells. This obvious way of simplifying the bookkeeping of the storage manager is used in almost all implementations. Notice that requiring all objects to be assigned compact locations would in general go too far. (Check that it is impossible to assign compact locations to all subobjects of the object *X* in Figure 4.10 without violating the allocation invariants.)

We are now in a position to make a first naive attempt at designing a storage manager. Let us assume for simplicity's sake that objects have a constant structure (i.e., objects are not "breathing"). The only interesting problem then is the allocation of storage for a new node. Let the term "free storage" denote the set of all cells in the store which are not part of a location assigned to an object. Allocating storage for a new

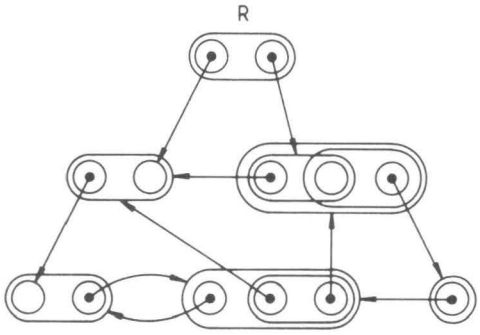


Figure 4.13

node could be done as follows. Initially the entire store consists of free storage. Each time a node is created a compact location of the proper size is "chopped" off the leftmost end of the free storage and assigned to the node.

4.1.10. Destruction of objects

The simple scheme given above requires a minimum of bookkeeping by the storage manager. However, it brings one face to face with one of the two central problems of storage management very soon: the finiteness of the store. In most programs objects are created at a rate which, using the above scheme, would cause a "storage overflow" in less than no time. This situation has been anticipated in many programming languages: Apart from operations to create and modify objects, these languages also have (more or less implicit) operations to "destroy" objects. Consequently, during the execution of programs in those languages "live" and "dead" objects can be distinguished. E.g., when leaving a block in ALGOL 60 all objects (variables) which have been created in that block are destroyed and hence they are no longer alive outside that block.

Observe that destruction, like the reference, is really an implementation concept. From a purely semantical point of view there is no need to introduce an operation to destroy objects. Since destruction is included as an operation in many programming languages and since it is important from an implementation point of view, we shall introduce destruction as a third operation in our abstract machine. The operation should not be interpreted as terminating the existence of an object, however. It should merely be viewed as appending the label "dead" to an object, which implies that the object is no longer used by the program. Since nodes are the units of creation it is obvious to choose them as the units of destruction as well. When destroying a node all of its components are destroyed with it. This implies that objects and their components are always destroyed at the same time, thus avoiding complications caused by live objects having dead components.

An obvious adjustment of the above simple storage management scheme is to "deallocate" the location of a node at the moment the node is destroyed. This means that the node is scratched from the bookkeeping of the storage

manager and that the cells of its location are returned to the free storage. Storage which would have been occupied by dead objects in the first scheme can be reused this way. Yet, uncontrolled allocation and deallocation of storage can make the free storage look like Swiss cheese, a phenomenon which is known as "fragmentation". This cannot only make the bookkeeping of the storage manager most inefficient, it can also lead to a situation that allocation of storage for an object is impossible, even though the overall size of the free storage is more than sufficient. Fragmentation is the second central problem of storage management (besides the finiteness of the store). This problem has been anticipated in many programming languages as well, particularly in those which feature a "block structure" (such as ALGOL 60). In those languages objects have nested lifetimes, which implies that they are created and destroyed in a last-in first-out ("last-created first-destroyed") manner. In implementations of these languages the free storage can be kept compact by using a "stack" for the allocation of storage [RANDELL & RUSSELL 64].

4.1.11. Dangling references

In connection with references and the destruction of objects an important problem crops up. After the destruction of an object other live objects can in principle still contain references to the object. These so-called "dangling references" are of course not supposed to be used for accessing the dead object. (The meaning of the label "dead" appended to an object is that the object is no longer used. Accessing it would be in contradiction with that.) There are several methods to prevent the use of dangling references in the abstract machine:

(1) Make it impossible for the user to lay hands on the reference of an object with a finite lifetime.
This is the simplest, but also the most restrictive method of preventing the use of dangling references. This method is for example used in PASCAL.
(2) Make sure that if an object X contains the reference of an object Y, then the lifetime of X is contained in the lifetime of Y.
This method is used in ALGOL 68, where it gives rise to a number of rules which are known as the "scope rules".
(3) Check during execution of the program for the use of dangling references.
This is a less restrictive method than (1) and (2), but it is rather expensive (especially as far as execution time is concerned). That is probably the reason why it is seldom used.
(3) Leave the responsibility to the user.
This is the most liberal, but also the most unsafe method, which is used for example in PL/I.

The first two methods do not only prevent the use of dangling references, they even prevent the occurrence of them. That is the standpoint we shall take here too: We assume that dangling references do not occur in the graph.

4.1.12. Objects with infinite lifetimes

Even if fragmentation can be avoided completely, the scheme of deallocating dead nodes will not be sufficient for many programming languages (among them LISP, PASCAL, ALGOL 68). The reason is that programs

in these languages can create objects which are never destroyed. Common names for these objects are "dynamic objects" (PASCAL) and "heap objects" (ALGOL 68). Although the lifetimes of these objects will be "infinite" (= lasting till the end of the execution of the program), the time they are used will often not. Extensive use of these objects may lead to exhaustion of the free storage very soon. The only way to avoid this is to equip the storage manager with the ability to deallocate storage, which is occupied by live, but no longer used nodes. The crucial problem in this is: How does the storage manager determine that a live node is no longer used?

Before discussing some solutions to the above problem two questions will have to be answered. The first is what we mean by the fact that a node is no longer used. Clearly, it should mean that none of its subobjects is used any more. Only then can the storage occupied by the node be deallocated safely. The second question is when the storage manager should establish that a live node is no longer used. Ideally this happens at the moment the node (i.e., one of its subobjects) is used for the last time. The storage occupied by the node could immediately be deallocated then. At the moment a node is used for the last time, however, it is generally not known that this is indeed the last time, because the use or non-use of the object may depend on things still to happen. So it is inevitable that the non-usage of a node is established some time later.

4.1.13. User controlled deallocation

The oldest solution to the problem is based on the assumption that the user knows best when a node is no longer used. The user is therefore enabled to give hints in his program as to which nodes are no longer used. Most language implementations featuring this scheme are very credulous and interpret the hint as a deallocation command. (Apart from ignoring it, there is not much else they can do.) This may give rise to a phenomenon which is very similar to the dangling reference. After deallocating a live node, other live nodes can still contain references to the deallocated node. Erroneously using these references to access the node will generally end up in disaster in the concrete machine. These references, which will be called "dangling pointers", are not the same as dangling references. Dangling references are of a semantical nature, because they arise from the destruction of an object, which is an operation prescribed by the semantics. Avoiding dangling references is a job of the language designer. Dangling pointers are of an implementation-technical nature, because they arise from "robbing" a node of its location in the store, while semantically the node lives on. Avoiding dangling pointers is a job of the language implementer.

4.1.14. Reference counting

The above solution is not only unsafe, but also delegates a part of the task of the storage manager to the user, which is undesirable. A second solution is based on the following line of argument. Apart from the root the machine can only use a node X if it has the disposal of its reference or the reference of one of its components. If none of the nodes of the graph contains a reference to a subobject of X, it is certain that X is no longer used. This leads to a method where a counter, a so-called "reference count", is associated with each node, counting the number of references to subobjects of the node [COLLINS 60]. Upon copying a reference to a subobject of the node the counter must be increased, and upon destroying ("overwriting") a reference to a subobject of the node the counter must be

decreased by one. As soon as the counter reaches zero, no more references to subobjects of the node exist. As a consequence, the node cannot be used any more and the storage occupied by the node can be deallocated. This is a safe method, but it introduces a considerable time and space overhead, even if no storage can or need be deallocated. Moreover, the method fails if circular references can occur [McBETH 63].

EXAMPLE 4.7

Consider the nodes *X* and *Y* in Figure 4.14.

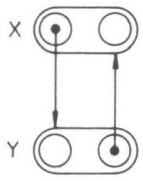


Figure 4.14

If outside the nodes *X* and *Y* no references to subobjects of *X* and *Y* occur in the graph, the counters of *X* and *Y* will remain equal to *1* for ever. Hence the storage occupied by *X* and *Y* will never be released, even though *X* and *Y* cannot be used any more. □

4.1.15. Reachability

The reason why the above method fails is that it is based on the assumption that (a subobject of) a node can be accessed as long as references to it exist. This assumption is false. Apart from the root (which is "accessible" by definition) an object *X* is only accessible if there exists another accessible object *Y* such that either *Y* is a structured object and *X* is a direct component of *Y* (in which case *X* can be accessed by accessing *Y* followed by a selection) or *Y* is a reference object and the value of *Y* refers to *X* (in which case *X* can be accessed by accessing *Y* followed by a dereferencing operation). In less recursive terms this means that an object *X* can only be accessed if there exists an access path to *X* emanating from the root. Objects for which such a path exists will be called "reachable". How objects can become "unreachable" is demonstrated in the following example.

EXAMPLE 4.8

Suppose the graph consists of three nodes *R*, *X* and *Y* as pictured in Figure 4.15, where *R* is the root.

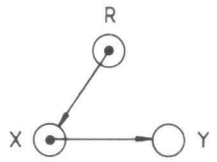


Figure 4.15

According to this figure all nodes are reachable. By modifying the value of *R* as follows:

val(R) := *ref(Y)*,

the situation of Figure 4.16 is obtained.

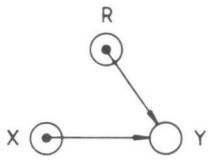


Figure 4.16

The node *X* has become unreachable now. □

Note that the fact that a node is unreachable does not imply that the storage occupied by the node may safely be deallocated: The node may have reachable components. Only if all subobjects of a node are unreachable, may the storage occupied by the node be deallocated. Nodes, all subobjects of which are unreachable, will be called "isolated". Check that there are four unreachable nodes and two isolated nodes in Figure 4.13. Another thing which should be noted is that deallocating an isolated node may introduce dangling pointers. As far as the user is concerned these dangling pointers are harmless, because they are contained in unreachable objects. The implementer, though, should beware of them.

EXAMPLE 4.9

Suppose Figure 4.17 is a picture of the graph, where *R* is the root.

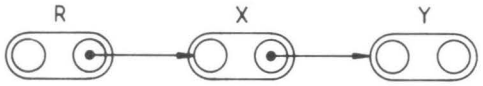


Figure 4.17

The node *Y* in this picture is isolated and may be deallocated. By doing so a dangling pointer will arise in *X*. □

Dead nodes are always isolated (otherwise, assuming that the root is always alive, we would have dangling references). Live nodes which are subject to being destroyed will normally not become unreachable (let alone isolated) during their lifetime, because if they cannot be accessed there is no way to destroy them from the program either.

4.1.16. Garbage collection

The concept of reachability is the basis of a third method to determine whether a node is still used or not. This method is called "garbage collection" and will be the subject of the next chapter. (The first description of this method can be found in [McCARTHY 60]; see also [McCARTHY et al. 65].) In this method the storage manager is supposed to have a special "employee", the so-called "garbage collector", which is charged with the tracing and deallocation of isolated nodes. The time-table of this employee was traditionally as follows. As long as the storage manager has sufficient free storage, the garbage collector is in a state of rest. When the storage manager runs out of free storage, the garbage collector is activated. It is its job to trace all isolated nodes and to deallocate the storage occupied by these nodes. After the garbage collector has completed its job, the storage manager can resume its work with a fresh (and hopefully sufficient) supply of free storage.

Garbage collection is a safe method which, in contrast to the reference counting scheme, will also work if circular references occur. Furthermore, the method (in its traditional form) will only introduce a time overhead if the free storage really gets exhausted. The main drawback of garbage collection in its traditional form is, that during a garbage collection (which may take a considerable time) the execution of the program is suspended. In time-critical applications this may be an insurmountable objection. This objection can be obviated by changing the time-table of the garbage collector. Instead of activating the garbage collector when need arises, it is made to work continuously. This implies that the program and the garbage collector work in parallel. However, in order to obtain the necessary synchronization, a considerable overhead is required. This overhead is only justified if either the continuity of the execution of the program is essential, or the overhead can be eliminated through dedicated hardware.

In contrast to traditional garbage collection the subject of parallel garbage collection can hardly be considered as a well-explored field of programming language implementation. Since we are primarily interested in giving a systematic treatment of a relatively well-established field of computer science and not in exploring a new field, we shall restrict ourselves to traditional garbage collection here (and in the next chapter).

For parallel garbage collection the interested reader is referred to [MULLER 75], [STEEL 75], [WADLER 76], [DIJKSTRA et al. 78].

Another method of avoiding a prolonged interruption of the execution of a program, consists of "spreading" the garbage collection process over the entire storage allocation process. Each time a piece of storage is allocated a few steps of a garbage collection are performed. An algorithm based on this idea is described in [BAKER 78] (main drawback: a "double store" is required). Another interesting possibility is to combine garbage collection with reference counting [DEUTSCH & BOBROW 76], [BARTH 77], [WISE & FRIEDMAN 77]. Neither of these methods will be discussed here.

4.1.17. Some complications

In practice, the job of a garbage collector is more complicated than we described. First, it is often the case that the garbage collector is not the only employee of the storage manager. Using the different lifetime properties of objects the storage manager may have several employees, each of which takes care of the storage management for a certain class of objects. In order to avoid that the garbage collector interferes with the work of the other employees, it should be unambiguously clear what belongs to the competence of the garbage collector and what does not. Since we are primarily concerned with garbage collection, we shall do away with this problem by assuming that garbage collection is the only strategy used for the deallocation of objects and that there are no objects which the garbage collector should a priori keep its hands off.

A second complication is the following. Suppose all but a few small subobjects of a huge node become unreachable. Then a very large part of the storage occupied by the node will not be used any more. Yet this storage cannot be deallocated, because the node is not isolated.

EXAMPLE 4.10

For the sake of this example, extend PASCAL with a standard function *ref*, which delivers the reference of an object. Consider the following program:

```
type row = array [1..10000] of integer;
var p: ↑integer;
begin
   .....
   var r: ↑row;
   begin
      new(r);
      .....
      p := ref(r↑[1]);
      .....
   end;
   .....
end
```

In the inner block an object X of type *row* is created and its reference is assigned to r. The reference of the direct component $X[1]$ of X is assigned to p. At exit of the inner block the object r is destroyed, and only one reference to a subobject of X remains (viz., the value *ref(X[1])* contained in p). Consequently $X[1]$ is the only one out of 10000 components of X that is reachable outside the inner block. □

In pathological cases the above may cause the garbage collector to return empty-handed, even if the major part of the store is not used. Basically this problem can be solved in two ways. (Note that in PASCAL implementations it is usually not a problem, because references to components of objects (variables) do not occur as values in PASCAL.) The first is to deallocate objects in a more subtle way. Instead of deallocating objects nodewise, they are deallocated individually. In abstract terms this implies that there is an operation which "peels" (a number of) the outer unreachable subobjects off a node, thus transforming the node into a number of smaller nodes (see Figure 4.18).

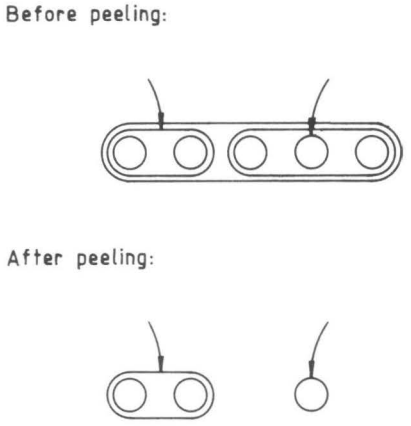


Figure 4.18

The problem with this peeling operation is that it should not violate system invariants such as: Nodes occupy compact locations and do not overlap. This makes it a rather intricate and highly implementation dependent operation, which is a potential source of garbage collector errors. Generally speaking it is therefore better to choose the second solution: decreasing the size of the nodes. This solution makes large nodes fall apart into a number of smaller nodes, which can be deallocated individually. We shall assume that the latter solution has been chosen, or in other words: We stick to the nodewise deallocation of objects.

4.1.18. Compaction

Another complication is caused by the fact that, in order to simplify their bookkeeping, many storage managers keep the free storage compact. However, the free storage as it will be after a garbage collection will generally not be compact. The locations deallocated by the garbage collector are usually scattered all over the store. Consequently, the free storage will display a high degree of fragmentation. A garbage collector, therefore, is quite often combined with a "compacter" which "compacts" the free storage. This combination of a garbage collector and a compacter is called a "compact(ify)ing garbage collector".

It is the job of a compacter to reallocate the nodes in the store in

such a way that a compact free storage results. In doing so the compacter is usually assumed not to affect the "layout" of an object, i.e., the relative position of the location of the object inside the location of the node of which the object is a subobject. The compacter can accomplish this most easily by "shifting" objects nodewise in the store and by "copying" the contents of the cells of the old location of a node to the cells of the new location.

The above may give the impression that compaction is a trivial matter. The reason why it is not is connected with the way references of objects are represented in the store. In most implementations the representation of the reference of an object depends on the location occupied by the object. By reallocating a node the representation of the reference of each subobject of the node is changed. Therefore, all cells containing representations of these references must be "updated" by the compacter.

The fact that, in contrast to other values, the representation of the reference of an object is variable could be modelled by making the representation function *repr* variable (like the allocation function *alloc*) and introducing a system invariant which reflects the dependence of *repr* on *alloc*. It is more convenient to keep *repr* constant, however. This is possible if we make the (reasonably general) assumption that the reference *V* of an object *X* is represented by the address of the leftmost cell of the location occupied by *X*, augmented with some constant *C(V)*. The constant *C(V)* represents the part of the representation of *V* which is independent of the location of *X*. If we now reinterpret *repr(V)* as *C(V)*, the function *repr* is constant. A consequence of this reinterpretation of *repr* is that Allocation Invariant (A1) must be rewritten. It falls apart into two invariants, one for atomic objects containing references and one for atomic objects containing values other than references (see Section 4.2).

In a virtual storage environment, garbage collection makes hardly any sense without compaction. The reason for this is that in such an environment storage is not really a scarce resource. However, the larger the size of the storage in use, the more access of it will slow down (through "page faults"). A garbage collection followed by a compaction may then be used to reduce the size of the storage in use, thus speeding up storage access.

It is also possible that the question of whether or not a garbage collection must be followed by a compaction is dependent on certain conditions which cannot be checked beforehand. For instance, if a "free storage list" is used for allocating storage, a failure to find a compact location of the proper size in the free storage list may be used to trigger a garbage collection. If after this garbage collection a location of the proper size still cannot be found, a compaction may help. Otherwise a compaction is not necessary (though, in order to avoid frequent garbage collections, it may be wise to perform a compaction if the fragmentation of the free storage has become large). This kind of garbage collector will be called a "conditionally compacting garbage collector".

4.2. MODEL

The concepts which were discussed informally in the previous section will now be defined precisely. This implies that we shall introduce a *model* of a storage management system. This model will serve as the basis for the next chapter. Conceptually it consists of three parts: a framework (or "representation" in terms of Chapter 2), a number of properties ("system invariants") and a number of definitions.

4.2.1. Framework

The framework of the model is presented below:

Each storage management system *H* has:

- *root(H)*: constant object,
- *graph(H)*: variable set of objects,
- *store(H)*: constant set of cells,
- *repr(H)*: constant mapping from values to words,
- *alloc(H)*: variable mapping from objects to sets of cells.

Each object *X* has:

- *ref(X)*: constant value,
- *kind(X)*: constant element from {*atomic*,*structured*},

If *kind(X)* = *atomic*

- *sort(X)*: constant element from {*scalar*,*reference*},
- *val(X)*: variable value,

If *kind(X)* = *structured*

- *struct(X)*: constant set of objects.

Each value *V* has:

- *kind(V)*: constant element from {*scalar*,*reference*},

If *kind(V)* = *reference*

- *obj(V)*: constant object.

Each cell *C* has:

- *addr(C)*: constant word,
- *cont(C)*: variable word.

A word is an integer.

The above defines for example the concept of an object as a dynamic system *X* with two direct components *ref(X)* and *kind(X)* and a number of additional direct components, which are dependent on *kind(X)*. If *kind(X)* = *atomic* there are two additional direct components *sort(X)* and *val(X)*, otherwise there is one additional direct component *struct(X)*. The adjective "constant" of a direct component implies that the direct component itself may not be changed, though components of it (if variable) may. E.g., *obj(V)* of a value *V* with *kind(V)* = *reference* may not be changed through a statement such as "*obj(V)* := *X*", but if *obj(V)* is an object with *kind(obj(V))* = *atomic*, then a statement such as "*val(obj(V))* := *5*" is allowed.

4.2.2. Definitions and invariants

For ease of description we shall from now on assume that we have only one storage management system *H* which is a representative of all storage management systems. All properties and concepts which are associated with *H* can be associated with each storage management system. This saves us the trouble of starting each definition with "For each storage management system *H* ...". The dependency of definitions on *H* will not be reflected in the notation. The system invariants which occur below should be read as axioms. They are prefixed by the capital letter S followed by a number. All objects, values, etc. are implicitly assumed to be finite.

H is a storage management system.

root(H) is called the root and will be denoted by R.

graph(H) is called the graph and will be denoted by G.

store(H) is called the store and will be denoted by S.

repr(H) is called the representation function and will be denoted by R.

alloc(H) is called the allocation function and will be denoted by A.

The following definitions and system invariants are related to values and objects in general.

A scalar value is a value V with $kind(V) = scalar$.

A reference value is a value V with $kind(V) = reference$.

If V is a reference value, then V is said to refer to $obj(V)$.

If X is an object, then $ref(X)$ is called the reference of X.

An atomic object is an object X with $kind(X) = atomic$.

A scalar object is an atomic object X with $sort(X) = scalar$.

A reference object is an atomic object X with $sort(X) = reference$.

If X is an atomic object, then $val(X)$ is called the value of X.

A structured object is an object X with $kind(X) = structured$.

If X is a structured object, then $struct(X)$ is called the structure of X.

An object X is said to be a direct component of an object Y if Y is a structured object and $X \in struct(Y)$.

The fact that an object is a component of an object is defined by the following rules (X, Y and Z denote objects):

(1) If X is a direct component of Y,
then X is a component of Y.
(2) If X is a component of Y,
Y is a direct component of Z,
then X is a component of Z.
(3) An object is a component of an object on account of the above rules only.

An object X is said to be a subobject of an object Y if X is a component of Y or $X = Y$.

The set of all subobjects of an object X is denoted by $sub(X)$.

Two objects X and Y are said to be disjoint if $sub(X) \cap sub(Y) = \emptyset$; otherwise they are said to overlap.

An atom of an object X is a subobject of X, which is an atomic object.

The set of all atoms of an object X is denoted by $atoms(X)$.

A branch of an object X is an atom of X, which is a reference object.

The set of all branches of an object X is denoted by $branches(X)$.

The degree of an object X, denoted by $degree(X)$, is the number of branches of X.

(S1) If X is an object,
then $ref(X)$ is a reference value.

(S2) If X is an object,
then $obj(ref(X)) = X$.

(S3) If V is a reference value,
then $ref(obj(V)) = V$.

(S4) If X is a scalar object,
then $val(X)$ is a scalar value.

(S5) If X is a reference object,
then $val(X)$ is a reference value.

(S6) If X is a structured object,
then $struct(X) \neq \emptyset$.

(S7) If X and Y are objects,
X is a subobject of Y,
Y is a subobject of X,
then $X = Y$.

Invariants (S2) and (S3) state that the reference of an object X refers to X and is unique. Invariant (S7) implies that the relation of "being a subobject of" between objects is a (reflexive) partial order. The following definitions and properties are concerned with objects in the graph G.

A node is an element of G.

A subnode is a subobject of a node.

(S8) $R \in G$.

(S9) If X is a node,
then X is a structured object.

(S10) If X and Y are nodes, $X \neq Y$,
then X and Y are disjoint.

(S11) If X is a subnode,
Y is a branch of X,
then $obj(val(Y))$ is a subnode.

Invariant (S9) is not essential. It is postulated for the sake of

convenience only. (It does not limit generality in any way.)
Invariant (S11) amounts to the absence of dangling references in the graph.

If X is a subnode, then *node(X)* is the unique node Y such that X is a subobject of Y.

The fact that an object is reachable is defined by the following rules (X and Y denote objects):

(1) R is reachable.
(2) If X is reachable,
X is a structured object,
$Y \in struct(X)$,
then Y is reachable.
(3) If X is reachable,
X is a reference object,
$Y = obj(val(X))$,
then Y is reachable.
(4) An object is reachable on account of the above rules only.

A node X is called isolated if all subobjects of X are unreachable.

Check that all reachable objects are subnodes. The following definitions and invariants are concerned with the store S.

If C is a cell, then $addr(C)$ is called the address of C.

If C is a cell, then $cont(C)$ is called the contents of C.

A location is a nonempty subset of the store.

A location L is called compact if $\{addr(C) \mid C \in L\}$ is a subrange of the integers.

The size of a location L, denoted by *size(L)*, is the number of elements of L.

The left address of a location L, denoted by *left(L)*, is defined by:

$$left(L) = min\{addr(C) \mid C \in L\}.$$

The right address of a location L, denoted by *right(L)*, is defined by:

$$right(L) = max\{addr(C) \mid C \in L\}.$$

(S12) $S \neq \emptyset$.

(S13) If $C,D \in S$, $C \neq D$,
then $addr(C) \neq addr(D)$.

(S14) S is compact.

If $A \in \{addr(C) \mid C \in S\}$, then *cell(A)* is the unique cell $C \in S$ with $addr(C) = A$.

If L is a location and N is an integer such that:

$$left(S) - left(L) \leq N \leq right(S) - right(L),$$

then the location $\underline{shift(L,N)}$ is defined by:

$$shift(L,N) = \{cell(addr(C) + N) \mid C \in L\}.$$

The definitions and invariants concerned with the representation function R and the allocation function A are given below.

(S15) $dom(R) = \{V \mid V \text{ is a value}\}$.

(S16) There are nodes $X_1,\ldots,X_n$ $(n \geq 0)$ such that
$dom(A) = sub(X_1) \cup \ldots \cup sub(X_n)$.

(S17) If $X \in dom(A)$,
then $A(X) \neq \emptyset$.

(S18) If $X \in dom(A)$,
then $A(X) \subset S$.

(S19) If $X \in dom(A)$,
X is a node,
then $A(X)$ is compact.

(S20) If $X \in dom(A)$,
X is an atomic object,
then $size(A(X)) = 1$.

(S21) If X is a reachable object,
then $X \in dom(A)$.

(S22) If X is a reachable scalar object,
then $cont(A(X)) = R(val(X))$.

(S23) If X is a reachable reference object,
$Y = obj(val(X))$,
then $cont(A(X)) = R(val(X)) + left(A(Y))$.

(S24) If $X,Y \in dom(A)$,
X is a subobject of Y,
then $A(X) \subset A(Y)$.

(S25) If $X,Y \in dom(A)$,
X and Y are disjoint,
then $A(X) \cap A(Y) = \emptyset$.

If $X \in dom(A)$, then X is said to <u>occupy</u> the location $A(X)$.

The <u>free storage</u> is the subset F of the store, defined by:

$$F = \{C \in S \mid \forall X \in dom(A)\ [C \notin A(X)]\}.$$

The storage management system is said to be <u>compact</u> if the location $S \setminus F$, where F is the free storage, is compact and $left(S \setminus F) = left(S)$.

If $X \in dom(A)$, $Y = node(X)$, then the layout of X, denoted by $layout(X)$, is defined by:

$$layout(X) = \{addr(C) - left(A(Y)) \mid C \in A(X)\}.$$

Here $dom(F)$ denotes the domain of the mapping F ($F = R,A$). (Mappings are considered as partial functions.) Invariant (S16) reflects that storage is allocated and deallocated nodewise. Invariant (S20) states that an atomic object occupies a location containing a single cell. If convenient (as in Invariants (S22) and (S23)), this location is identified with the cell contained in it. Invariants (S22)-(S25) are the Allocation Invariants from Subsection 4.1.4, except that Allocation Invariant (A1) has been restricted to reachable objects and has been split into two invariants ((S22) and (S23)). Notice that in a compact storage management system the free storage is (arbitrarily) located in the right part of the store.

4.2.3. Operations

Garbage collection, compaction and compacting garbage collection will now be defined as abstract operations which can be applied to a storage management system.

COLLECT GARBAGE is an operation, called garbage collection, which may be applied to a storage management system. It is defined as follows:
- Remove all subobjects of isolated nodes from $dom(A)$.

COMPACT is an operation, called compaction, which may be applied to a storage management system. It is defined as follows:
- Change $A(X)$ for a number of $X \in dom(A)$,
- Change $cont(C)$ for a number of $C \in S$,

in such a way that:
- The system invariants are not affected,
- The layout of objects in $dom(A)$ is not affected,
- The compactness of the storage management system is established.

COLLECT GARBAGE & COMPACT is an operation, called compacting garbage collection, which may be applied to a storage management system. It is defined as follows:
- Remove all subobjects of isolated nodes from $dom(A)$,
- Change $A(X)$ for a number of $X \in dom(A)$,
- Change $cont(C)$ for a number of $C \in S$,

in such a way that:
- The system invariants are not affected,
- The layout of objects in $dom(A)$ is not affected,
- The compactness of the storage management system is established.

Check that *COLLECT GARBAGE* does not affect the system invariants or the layout of objects in $dom(A)$. Notice also that neither of the operations changes G or the value of atomic objects (though "temporary" changes are allowed).

4.2.4. On the relation with Chapter 2

We shall conclude this section with a number of remarks concerning the relation of the model presented above with Chapter 2. Dynamic systems are called "data structures" in Chapter 2. The concepts such as "storage

management system", "object", "value", etc. correspond to things called "types" in Section 2.4. The only types which are not defined are the types "integer" and the parameterized types "set of *type*" and "mapping from $type_1$ to $type_2$". These types (and some others like the parameterized types "bag of *type*" and "stack of *type*") are implicitly assumed to have been defined in the obvious way. No access rights to any of these implicit types exist. Values of these types can only be manipulated through their associated operations (e.g., sets by selection, union, intersection, equality, etc.). The notation used here differs from the notation used in Chapter 2. E.g., instead of "*H.graph*" we write "*graph(H)*".

An important point is the use of the "=" operator. This operator will be used in two different ways. First, if *X* and *Y* are dynamic systems to the components of which we have access (such as "objects"), "*X* = *Y*" will denote that *X* and *Y* are "identical". This was denoted by "$X \equiv Y$" in Chapter 2. The fact that *X* and *Y* are identical implies that all components of *X* and *Y* are also identical. The converse is not true, since dynamic systems have a "hidden identity". Secondly, if *X* and *Y* are dynamic systems which we may manipulate through their external operations only (such as "integers"), "*X* = *Y*" will denote that *X* and *Y* are "equal". For the integers this was denoted by "*EQUAL(X,Y)*" in Chapter 2. In contrast to identity, equality is an ordinary external operation provided by the "designer" of a dynamic system.

The fact that different dynamic systems may have identical components is important in relation to making abstractions of dynamic systems. A way of making an abstraction of a dynamic system is to "forget" some of its direct components. By doing so for a number of dynamic systems some of them, which differed formerly in at least one component, may become identical in all components. The latter systems are generally not supposed to be identified, however. As a simple example, consider the concept (dynamic system) of a value as defined above. This is an abstraction of the concept of a (simple) value as it occurs in programming languages. Examples of values are integers, booleans, reals and references. Here we are only interested whether a value is a reference or not, and if so, to which object it refers. Therefore, a "scalar value", i.e., a value which is not a reference, has only one component which indicates that the value is not a reference. If values with identical components would be identified, only one scalar value would be possible, which is clearly undesirable. In our approach we have a potentially infinite number of scalar values.

4.3. CONCLUSION

The main purpose of this chapter was to define the abstract concept of a storage management system and its related concepts. Since the usefulness of this model will have to be demonstrated in the next chapter, conclusions will be deferred till there. Harking back to the introduction of this chapter, one "metaconclusion" seems justified here, however: The given collection of definitions and invariants closely resembles a mathematical theory. The only thing that seems to be missing are the theorems. We could have added a number of them without great difficulty, but most of them would be self-evident and hardly worth mentioning. This is more or less typical for all dynamic systems: They constitute comprehensive, but from a mathematical point of view rather uninteresting theories. What makes them interesting is the fact that unlike "static" mathematical theories, operations can be applied to them. The control of their complexity then becomes a difficult and challenging problem.

CHAPTER 5

A SURVEY OF GARBAGE COLLECTION

5.0. INTRODUCTION

In Chapter 4 the subject of garbage collection was discussed as one of the solutions to the storage management problem. In this chapter (compacting) garbage collection will be discussed as a problem in its own right using the storage management model from Chapter 4 as a basis. This implies that a survey of garbage collection algorithms will be given. Though no claim for completeness is made, the survey will include the major algorithms known from literature.

The interesting aspect of the garbage collection problem is the fact that a garbage collector has to work under a severe storage constraint. Like every algorithm the garbage collector needs a certain (generally unpredictable) working space. Since, in the end, the store is the only data structure available to the garbage collector, this working space must be found in the store. At the moment the garbage collector is called, however, free storage is usually very scarce: The lack of free storage was the very reason for calling the garbage collector. This situation has led to the design of ingenious garbage collectors, which "encode" their working space in the store with little or no space overhead.

Both garbage collection and compacting garbage collection will be discussed here. The compacting garbage collection problem can be decomposed into a garbage collection and a compaction problem. The first two sections of this chapter will therefore deal with garbage collection and compaction algorithms respectively. The decomposition of a compacting garbage collector into a garbage collector and a compacter is only a "conceptual decomposition". It is analogous to the decomposition of a compiler into "phases": lexical analysis, parsing, semantical analysis, code generation, etc.. By "merging" these phases (such as in a one-pass compiler) a considerable increase in efficiency can be obtained. The same applies to merging garbage collection and compaction algorithms into compacting garbage collection algorithms. This merging will be discussed in another section.

5.0.1. Classification of the problems and algorithms

The discussion of the garbage collection and compaction problems and their solutions (the algorithms) will follow the lines sketched in Chapter 1. The definitions of the abstract operations *COLLECT GARBAGE* and *COMPACT* from Chapter 4 will be regarded to define the "basic garbage collection problem" and the "basic compaction problem" respectively. More concrete versions of these problems are classified by giving the additional details they satisfy. An example is the garbage collection problem where all references refer to nodes. Here the additional detail is: "all references refer to nodes". The details should not be viewed as additional system invariants, but merely as assertions which are known to hold prior to the garbage collection or compaction operation. These assertions may allow a more efficient solution of the problem.

Starting from an abstract algorithm which is a solution to the basic problem, all algorithms will be derived through correctness-preserving transformations. Like the problems, the algorithms will be classified by giving (a number of) details they satisfy. These details are usually assertions about the run-time behaviour of the algorithms, such as: "objects are marked immediately after they are traced". Each algorithm will be denoted by a label of the kind "$\pi\delta.\alpha$". Here $\pi \in \{G,C\}$ denotes the problem to which the algorithm is a solution (G for garbage collection and C for compaction), δ is a sequence of capital letters which denote the details of the problem and α is a sequence of capital letters which denote the details of the algorithm. For example, Algorithm GN.DT is a solution of the garbage collection problem if all references refer to nodes (detail N). The algorithm uses nodes as marking units (detail D) and marks them immediately after they are traced (detail T). Notice that an algorithm labelled $\pi\delta.\alpha$ is a solution of each π-problem with details $\delta' \supset \delta$. In some cases asterisks will be used to denote simple implementations of algorithms. For example, Algorithm GN.DTER* is derived from Algorithm GN.DTER using a simple implementation of the variable T from Algorithm GN.DTER.

5.0.2. On the description of the algorithms

The algorithms will be described in a more or less informal language. The semantics of this language will generally be intuitively clear. If necessary, explanation will be provided. From a more formal point of view the semantics of the algorithmic language used may not be so clear at all. This is due to the fact that the algorithms operate on dynamic systems (data structures) with highly shared components. How the semantics of this language can be formalized is demonstrated in Chapter 2, where the semantics of a language is defined which can be viewed as a "kernel" for the language used here. The extension of this kernel to a language such as the one used here is also discussed in Chapter 2.

Each algorithm will be considered to operate on the storage management system H introduced in the previous chapter. All other variables used by an algorithm, except possibly a few local variables, will be listed at the top of the algorithm. Recursion will not be used in the algorithms. The variables listed at the top of an algorithm therefore indicate the working space required for the algorithm. The required working space for a garbage collection or compaction algorithm, as we already saw, is an important datum.

No proof of correctness of the algorithms described will be given. The first reason for this is conciseness. The second reason is that the algorithms which will be described here are derived by transforming more fundamental algorithms. Starting points are a few abstract algorithms which can almost immediately be seen to be correct. Proving the correctness of an algorithm is now reduced to proving that the transformations applied in the derivation of the algorithm are correctness-preserving. It will not be difficult for the reader to convince himself intuitively of the correctness-preservation of most of these transformations. If a more rigorous proof of correctness-preservation of a transformation is required, the reader is referred to Chapter 3, where a simple method to do so is described. This method can be used for many (but not all) of the transformations applied here. The method is demonstrated in Chapter 3 in a fully elaborated derivation and proof of correctness of Algorithm GNK.DTER*** (without $RELEASE_4^*$; see Subsection 5.1.5).

In Section 5.1 a survey of garbage collection algorithms will be given as described above. Section 5.2 discusses the compaction problem in the same way. In Section 5.3 various ways of merging garbage collection and compaction algorithms into compacting garbage collectors are discussed. This section contrasts with the other two sections in that it merely presents a number of examples, instead of a hierarchy of algorithms. For reasons of conciseness only short comments on most algorithms in all three sections will be given, together with references to the literature (when appropriate). Concluding remarks are made in Section 5.4.

5.1. GARBAGE COLLECTION

5.1.1. General discussion

5.1.1.1. Marking

According to the definition of the operation *COLLECT GARBAGE* in the previous chapter, it is the job of a garbage collector to determine for each node X if X is isolated and if so, to deallocate all storage occupied by (subobjects of) X. In order to determine that a node is isolated, the garbage collector must determine that each subobject of the node is unreachable. It is not difficult to see that, in essence, the only way to determine that an object is unreachable is to generate the entire set of reachable objects and check whether the object is in it or not. Since generation of this set is expensive it is of course not very wise to perform this operation for each object over again. The best thing to do is to generate the set once and to "mark" reachable objects as such. This "marking information" can subsequently be used to determine whether a node is isolated or not. The marking information generally introduces a space overhead. If objects should be marked individually, this overhead may be overwhelming (see Example 5.1).

EXAMPLE 5.1

Suppose for a moment that it is possible to take "subarrays" of arrays in PASCAL. For example, if x has been declared as follows:

```
var x: array [0..9] of integer;
```

then $x[3..5]$ is the subarray of x, composed of the atomic objects $x[3]$, $x[4]$ and $x[5]$. Then a one-dimensional array of length n has $n(n+1)/2$ subarrays, which are all distinct objects. If objects should be marked individually, the space overhead for the marking information of the array would be proportional to $n(n+1)/2$, which is intolerable. (This becomes even worse in the case of multidimensional arrays.) □

In order to determine whether a node is isolated it is not always necessary to know whether each individual subobject of the node is reachable or not. Therefore it makes sense not to mark all objects, but only a "sufficiently large" subset of the set of objects. The elements of this subset will be called the "marking units". The choice of the marking units is highly implementation dependent. In most garbage collectors, however, only one of the following four alternatives is chosen:

(1) Objects.
(2) Atomic objects.
(3) Reference objects.
(4) Nodes.

Depending on the marking units chosen, four kinds of garbage collection algorithms will be distinguished, which will be referred to as garbage collection algorithms using "object marking", "atom marking", "reference marking" and "node marking", respectively.

Of the four ways of marking, object marking is the most natural: The concept of reachability has been defined for objects, and not exclusively for atomic objects, reference objects or nodes. As we shall see, all garbage collection algorithms using atom, reference and node marking can be derived from those using object marking. In this general discussion of garbage collection we shall therefore choose object marking as our basis.

5.1.1.2. The job of a garbage collector

The job of a garbage collector (using object marking) is to mark all reachable objects and subsequently deallocate all storage occupied by nodes which do not have a marked subobject. In order to describe this more formally an abstract variable M, representing the set of marked objects, is introduced. The job of the garbage collector can now be described as follows:

```
M := {X | X is a reachable object}.
RELEASE1.
```

where

```
RELEASE1:
    For each node X ∈ dom(A)
    | If M ∩ sub(X) = ∅
    | | dom(A) := dom(A) \ sub(X).
```

Most garbage collection algorithms which are described in literature deal only with the first part of the job of a garbage collector (marking reachable objects). These algorithms are therefore usually called "marking algorithms". Here we shall include the second part of the job of a garbage collector (releasing storage) in all algorithms, because it is an inseparable part of these algorithms: When transforming an algorithm this part may have to be transformed as well. The particular ways of releasing storage (i.e., the implementation of the above for-loop) will not be discussed. The reason is that releasing storage is the simplest and generally the least time-consuming part of a garbage collector. Moreover, it depends on the kind of bookkeeping used by the storage manager, which is highly implementation dependent. We shall therefore focus on marking and neglect the time required to release storage in complexity considerations.

Using a somewhat finer grain than above, the job of a garbage collector can be described, in a still very abstract way, as follows:

```
M := ∅.
While not sufficient
| Let X be a reachable object.
| M := M ∪ {X}.
RELEASE1.
```

Here "sufficient" is some condition implying that each reachable object is contained in M. The above "algorithm" is still far away from a practicable garbage collection algorithm. Let us first focus our attention on the statement "Let X be a reachable object".

5.1.1.3. Visiting and tracing

In order to be able to select an arbitrary reachable object, we must possess knowledge concerning the reachability of objects. In view of the generative nature of the definition of reachability, the only way to obtain this knowledge is to *generate* it. We shall model this by introducing a variable set Q, containing the objects "known" to be reachable. We have to do two things now: generate the set of reachable objects in Q, and mark the objects in Q (i.e., put them in M).

It may seem obvious to identify Q and M. However, at this very abstract level it makes sense to distinguish clearly between the act of discovering that an object is reachable, which is usually called the "tracing" of the object, and the act of marking the object. As we shall see later, there are sensible algorithms in which these operations are indeed separated.

The way to generate the set of reachable objects in Q follows almost immediately from the definition of reachability. First, we know that the root R is reachable, so initially $Q = \{R\}$. Then, repeatedly, objects in Q are "visited". During the visit to an object X the following is done. If X is a structured object, then an arbitrary direct component Y of X is chosen and put in Q. We shall call this "tracing by selection". If X is a reference object, then the object Y referred to by the value of X is determined and put in Q. This will be called "tracing by dereferencing". If X is a scalar object, nothing is done.

We can now decompose a marking algorithm into two more or less independent processes: A "tracer", which "fills" Q, and a "marker", which marks the objects in Q. These two processes are merged sequentially in the following algorithm, using the "either-or construct", which arbitrarily selects one of its two alternatives:

```
Algorithm G
Variables:
   M: set of objects,
   Q: set of objects.
Action:
   M,Q := ∅,{R}.
   While not sufficient
   | Let X ∈ Q.
   | Either
   | | M := M ∪ {X}.                          ⟵ marking X
   | or
   | | Case                                   ⟵ visiting X
   | | 1. X is a structured object
   | | | Let Y ∈ struct(X).                   ⟵ tracing Y
   | | | Q := Q ∪ {Y}.
   | | 2. X is a reference object
   | | | Let Y = obj(val(X)).                 ⟵ tracing Y
   | | | Q := Q ∪ {Y}.
   | | 3. X is a scalar object
   | | | Skip.
   RELEASE1.
```

What is meant by "tracing", "marking" and "visiting" an object is indicated in the algorithm.

Algorithm G will be chosen as the basis for the derivation of all other garbage collection algorithms, including those which do not use object marking. The algorithm is still very abstract, not in the least because the loop does not have a proper termination condition. Even if the loop had a proper termination condition, the algorithm need not terminate. The reason is that there is simply too much freedom in the marking, visiting and tracing of objects. We shall impose a number of restrictions now, such that Algorithm G can be made to terminate.

5.1.1.4. Restrictions on Algorithm G

Let us first discuss the marking of objects. In Algorithm G the marking of objects proceeds entirely independently from the visiting and tracing of objects. A reasonable restriction of the anarchy prevailing in Algorithm G is to link the former process to the latter, i.e., combine the marker with the tracer. There are two plausible ways to do so. The first is to mark objects when they are traced, and the second is to mark objects when they are visited. Each choice gives rise to a different branch in the hierarchy of algorithms which can be derived from Algorithm G. Only these two choices will be considered here. They are represented by the algorithm details T and V to be presented in Subsection 5.1.1.7. Notice that in both choices a reachable object X is processed in the following order:

trace X → mark X → visit X.

In the first choice "trace X" and "mark X" are combined, and in the second choice "mark X" and "visit X" are combined. The distinction between "marking after tracing" and "marking before visiting" is essentially the same as that noticed in [THORELLI 72].

Next, consider the visiting and tracing of objects. It is easy to see that in Algorithm G it does not make sense to visit a structured object again, once all of its direct components have been traced. Neither does it make sense to visit an atomic object again. The following are therefore reasonable restrictions:

Restrictions on the visiting and tracing of objects in Algorithm G

(1) Case 1 may not be chosen more than once for each combination of X and Y.
(2) Case 2 may not be chosen more than once for each X.
(3) Case 3 may not be chosen more than once for each X.

The above restrictions imply that atomic objects are visited at most once and structured objects are visited at most as many times as their number of direct components. We shall adhere to these restrictions as well as possible. As far as garbage collection algorithms using object marking are concerned, we can even satisfy them entirely. In the algorithms using other ways of marking we shall have to compromise, as we shall see. The danger of relaxing the restrictions is clearly demonstrated by the following example.

EXAMPLE 5.2

The following algorithm (which marks objects immediately after they are traced) is a properly terminating garbage collection algorithm using object marking, derived from Algorithm G:

```
Variables:
   M: set of objects,
   B: boolean.
Action:
   M,B := {R},true.
   While B
   | B := false.
   | For each X ∈ M
   | | Case
   | | 1. X is a structured object
   | | | For each Y ∈ struct(X)
   | | | | If Y ∉ M
   | | | | | M,B := M ∪ {Y},true.
   | | 2. X is a reference object
   | | | Let Y = obj(val(X)).
   | | | If Y ∉ M
   | | | | M,B := M ∪ {Y},true.
   | | 3. X is a scalar object
   | | | Skip.
   RELEASE1.
```

This algorithm is a generalization of Algorithm A in [KNUTH 68]. It clearly does not meet the above restrictions. In the worst case the algorithm makes $O(n^2)$ visits to objects, where n is the number of reachable objects. □

5.1.1.5. Status information

In order to let Algorithm G meet the above three restrictions some extra bookkeeping is necessary. First of all, the algorithm must be able to select an X which may still be visited. This is not enough, though. When visiting a structured object X, it must be possible for the algorithm to select a direct component Y of X, which has not been traced in any visit of X before. All this will be modelled by an abstract variable T, which is a bag (or "multiset") of pairs (X,V), where X is an object and V is a set of direct components of X. The information contained in T will be referred to as the "status information". Roughly speaking, the fact that $(X,V) \in T$ means that X may still be visited and if X is a structured object that the direct components $Y \in V$ of X have not yet been traced in any previous visit to X. If X is an atomic object, V is always empty. Supposing the restrictions are satisfied, Algorithm G can now be made to terminate by choosing the test "$T = \emptyset$" as the termination condition (instead of "sufficient").

The reason for choosing a bag instead of a set for T is mainly, that the only operations which are performed on T are adding and removing elements, where (in principle) the same element may occur more than once in T (particularly in those algorithms which do not satisfy all three restrictions given above). This kind of bag can easily be implemented, for example as a stack or queue.

The introduction of the variable T in Algorithm G will make the variable Q redundant, as we shall see. Thus two kinds of space overhead

remain: the marking and status information, represented by the abstract variables M and T. This, however, does not constitute the only garbage collector overhead.

5.1.1.6. Type information

A third kind of overhead has to do with the answer to the following questions:

How can the garbage collector determine
(1) whether an object is a structured, reference or scalar object?
(2) the direct components of an object?
(3) the object referred to by a reference?

The information required to determine (1), (2) and (3) will be referred to collectively as the "type information". The type information associated with a certain object will be called the "type" of the object. The type information, though present in the graph, need not be present in the store. (If it is, for purposes other than garbage collection, that is fortunate.) For reasons of efficiency each implementer will try to include in the store as little information from the graph as possible. Yet, since the garbage collector ultimately has to operate exclusively on the store, there must be a way for the garbage collector to get hold of the type information. There are basically three ways to solve this problem:

(1) Object typing.
In this solution a value is associated with each object, which can be used by the garbage collector to get hold of the type information. This value is typically represented by a pointer contained in the location occupied by the object, which points to a piece of encoded type information in the store. Object typing generally requires an overhead per object. This overhead can be reduced to an overhead per node in case only references to nodes occur in the graph (problem detail N, see Subsection 5.1.1.7).
(2) Reference typing.
Here a value is associated with each reference contained in the graph, which enables the garbage collector to determine the type of the object referred to by the reference. Thus the garbage collector can find the type of an object traced by dereferencing. For an object Y traced by selecting a component of an object X, the garbage collector must be able to derive the type of Y from the type of X. Reference typing requires an overhead per reference object, which is generally much better than an overhead per object. Reference typing does not work, however, if the structure of objects is variable.
(3) Type tracking.
In this approach the garbage collector is supposed to be able to derive the type of an object Y, traced either by selecting a direct component of a structured object X or dereferencing a reference object X, from the type of X. Knowing the type of the root, the garbage collector can calculate the type of each reachable object. For that purpose the garbage collector will have to keep track of the types of objects in T. If there is a "static" relation between the types of X and Y (such as in "strongly typed" languages like ALGOL 68) that might be the only space overhead this method requires. If there is a "dynamic" relation between the types of X and Y, parts of the type information must still be contained in the store, which may give

unpleasant complications. Moreover, this approach has the drawback that the garbage collector must have an intimate knowledge of the structure and representation chosen for objects. A slight change in the structure or representation of objects may require the entire garbage collector to be rewritten.

Notice that if all objects have the same structure (such as in pure LISP) the type information need not cause any overhead at all.

It appears from the above that a garbage collection algorithm requires a (space) overhead which is caused by three kinds of information:

(1) Marking information.
(2) Status information.
(3) Type information.

Limiting the overhead caused by the marking, status and type information is one of the essential parts of garbage collector design. Various ways to achieve this will be discussed in the sequel.

5.1.1.7. Garbage collection details

We shall now give the details which will be used to classify the different garbage collection problems and algorithms. First we present the problem details, which will be explained together with the algorithms that use them.

Garbage collection problem details

N: If X is a reachable reference object,
then $obj(val(X))$ is a node.

K: If X is a node,
then the branches of X are numbered from 1 to $degree(X)$.
The i-th branch of X is denoted by $branch(X,i)$ $(1 \leq i \leq degree(X))$.

This is a surprisingly small number of details. The reason for this is that the garbage collection problem as we defined it is rather abstract. It is concerned (almost) exclusively with the abstract layer of the storage management model. At this high level of abstraction few details can be distinguished. Still, all garbage collection algorithms can be expressed at this level of abstraction (with the addition of the above details, if necessary). The compaction problem is less abstract than the garbage collection problem. It is concerned with the concrete layer of the model (the store) as well. Accordingly, the number of compaction problem details will be higher (see Subsection 5.2.1.5).

The garbage collection algorithms will be classified according to the following details:

Garbage collection algorithm details

A: The marking operation of a structured object is modelled by an empty action.

B: The marking operation of a structured object is modelled by marking all of its atoms.

C: The marking operation of an object other than a reference object is modelled by an empty action.

D: The marking operation of an object other than a node is modelled by an empty action.

T: Objects are marked immediately after they are traced.

V: Objects are marked immediately before they are visited.

E: During a visit to an object all of its direct components are traced.

S: Atomic objects are visited immediately after they are traced.

Q: Objects having no branches are visited immediately after they are traced.

R: Objects other than reference objects are visited immediately after they are traced.

H: Objects traced through selection are visited immediately after they are traced.

F: Reference objects are flagged when they are visited.

Details A, B, C and D are used to derive the algorithms using atom marking (details A and B), reference marking (detail C) and node marking (detail D) respectively from Algorithm G. Details T and V were already discussed. The other details will be discussed with the algorithms featuring them. Notice that not all combinations of the details make sense.

The four classes of garbage collection algorithms (using object, atom, reference and node marking, respectively) will now be discussed in four separate subsections (5.1.2-5.1.5).

5.1.2. Garbage collection algorithms using object marking

As we discussed in Subsection 5.1.1, object marking is the most natural way of marking. It is also the least usual way of marking. The reason for this is that the overhead caused by the marking information may be tremendous, as demonstrated in Example 5.1. There are, on the other hand, situations where application of this kind of marking is very well conceivable (for example, if nodes have a tree structure). We already discussed one garbage collection algorithm using node marking: Algorithm G. This very abstract algorithm will be the starting point for the derivation of all other garbage collection algorithms.

The algorithm details relevant to garbage collection algorithms using object marking are T, V, E and S. The sensible combinations of these details lead to the hierarchy of algorithms pictured in Figure 5.1. All of these algorithms will be described below. This is to illustrate how algorithms can be derived from other algorithms by "adding" details. For garbage collection algorithms using atom, reference and node marking not all algorithms in the hierarchies in question will be described. The derivation of the algorithms which are not described is either trivial, or may be accomplished in a way similar to the derivation of the algorithms presented below.

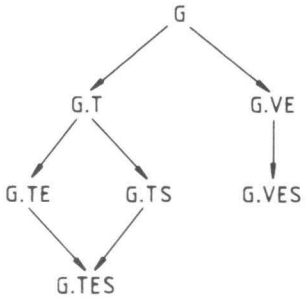


Figure 5.1

The first algorithm to be discussed is derived from Algorithm G by marking objects as soon as they are traced (according to detail T) and imposing the restrictions on the visiting and tracing of objects through the introduction of the variable T. (Notice that, in fact, the restrictions are implicit details.) It is easy to see that the variable Q then becomes redundant and can be removed, resulting in:

```
Algorithm G.T

Variables:
   M: set of objects,
   T: bag of pairs (object, set of direct components).
Action:
   M,T := {R},{(R,struct(R))}.
   While T ≠ ∅
   | Get (X,V) from T.
   | Case
   | 1. X is a structured object
   | | If V ≠ ∅
   | | | Get Y from V.
   | | | T := T ∪ {(X,V)}.
   | | | INSPECT(Y).
   | 2. X is a reference object
   | | Let Y = obj(val(X)).
   | | INSPECT(Y).
   | 3. X is a scalar object
   | | Skip.
   RELEASE1.

INSPECT(Y):
   Case
   1. Y is a structured object
   | If Y ∉ M
   | | M,T := M ∪ {Y},T ∪ {(Y,struct(Y))}.
   2. Y is an atomic object
   | If Y ∉ M
   | | M,T := M ∪ {Y},T ∪ {(Y,∅)}.
```

Here the operation "Get Y from V" is a shorthand for:

```
Let Y ∈ V.
V := V \ {Y}.
```

(Analogously for "Get (X,V) from T".)

The first question that arises in relation to the implementation of Algorithm G.T is, how to implement the set M of marked objects. Since in object marking it must be possible to mark each individual object, probably the best way to do it here is to make room for a flag in each location occupied by an object, indicating whether the object is marked or not. As already mentioned, this may cause a considerable overhead. In cases where this overhead is prohibitive it is therefore better not to use object marking.

A second question is how to implement the bag T. The obvious implementation for T is a stack (though a queue is also conceivable). This stack can either be kept in a separate location in the store or, by reserving extra room in each object location, as a linked list through these locations. The latter may involve an enormous space overhead. The objection to the former is that the maximum size of the stack is usually not known beforehand. The entries in the stack must represent pairs (X,V), where X is an object and V is a set of direct components of X. Using a separate stack, X can be represented by a pointer and, if the direct components of X are removed from V in a predefined order, V can be represented by an integer.

A first way to reduce the overhead caused by the status information is demonstrated in Algorithm G.TE. This algorithm is derived from Algorithm G.T by combining the separate visits to trace the direct components of an object into a single visit (detail E). Thus a number of operations on T are saved. Furthermore, this detail enables T to be chosen as a bag of objects, instead of a bag of pairs (object, set of direct components), because the second element of a pair has become superfluous. Combining visits to objects is a useful and generally applicable technique to reduce garbage collector overhead. Many variations of this technique will be met.

```
Algorithm G.TE

Variables:
   M: set of objects,
   T: bag of objects.
Action:
   M,T := {R},{R}.
   While T ≠ ∅
   | Get X from T.
   | Case
   | 1. X is a structured object
   | | For each Y ∈ struct(X)
   | | | If Y ∉ M
   | | | | M,T := M ∪ {Y},T ∪ {Y}.
   | 2. X is a reference object
   | | Let Y = obj(val(X)).
   | | If Y ∉ M
   | | | M,T := M ∪ {Y},T ∪ {Y}.
   | 3. X is a scalar object
   | | Skip.
   RELEASE1.
```

A second way to reduce the overhead caused by the status information is to visit atomic objects immediately after they are traced, instead of putting them in T first (detail S). This implies that after tracing an (unmarked) atomic object, the (possibly zero-length) access path emanating from the object is followed until either a marked or a structured object is encountered (see Example 5.3). This saves an add and remove operation on T per reachable atomic object and it makes only structured objects occur (as the first elements of pairs) in T.

EXAMPLE 5.3

Consider Figure 5.2.

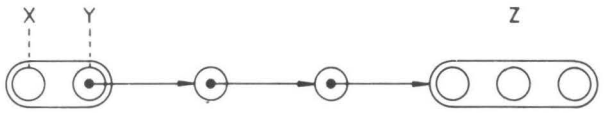


Figure 5.2

Upon tracing the unmarked scalar object X there is no need to put it in T: X need only be marked. Upon tracing the unmarked reference object Y there is also no need to put it in T: The garbage collector can traverse the chain of references emanating from Y, while marking the objects in the chain, until a marked or structured object Z is encountered. No objects, except possibly Z, need be put in T during this traversal. □

```
Algorithm G.TS
Variables:
   M: set of objects,
   T: bag of pairs (object, set of direct components).
Action:
   M,T := {R},{(R,struct(R))}.
   While T ≠ ∅
   | Get (X,V) from T.
   | Get Y from V.
   | If V ≠ ∅
   | | T := T ∪ {(X,V)}.
   | While Y is a reference object and Y ∉ M
   | | M := M ∪ {Y}.
   | | Y := obj(val(Y)).
   | If Y ∉ M
   | | M := M ∪ {Y}.
   | | If Y is a structured object
   | | | T := T ∪ {(Y,struct(Y))}.
   RELEASE1.
```

Combining details E and S leads to Algorithm G.TES, which can either be derived from Algorithm G.TE by imposing detail S, or from Algorithm G.TS by imposing detail E.

```
Algorithm G.TES

Variables:
   M: set of objects,
   T: bag of objects.
Action:
   M,T := {R},{R}.
   While T ≠ ∅
   | Get X from T.
   | For each Y ∈ struct(X)
   | | While Y is a reference object and Y ∉ M
   | | | M := M ∪ {Y}.
   | | | Y := obj(val(Y)).
   | | If Y ∉ M
   | | | M := M ∪ {Y}.
   | | | If Y is a structured object
   | | | | T := T ∪ {Y}.
   RELEASE1.
```

In all previous algorithms, objects are marked as soon as they are traced (detail T). A second approach, as we already discussed, is to mark objects immediately before visiting them (detail V). This implies that upon tracing an object it is put in T first. When the object is fetched from T for the first time, it is checked whether the object has already been marked. If not, the object is marked and subsequently visited. In order to avoid unnecessary tests to determine whether the object is actually visited for the first time, it makes sense to combine all visits to the object into a single visit. Detail V therefore occurs in combination with detail E only. By imposing details V and E and the restrictions concerning the visiting and tracing of objects on Algorithm G (using the variable T), the following algorithm can be derived, from which the variable Q has disappeared:

```
Algorithm G.VE

Variables:
   M: set of objects,
   T: bag of objects.
Action:
   M,T := ∅,{R}.
   While T ≠ ∅
   | Get X from T.
   | If X ∉ M
   | | M := M ∪ {X}.
   | | Case
   | | 1. X is a structured object
   | | | For each Y ∈ struct(X)
   | | | | T := T ∪ {Y}.
   | | 2. X is a reference object
   | | | Let Y = obj(val(X)).
   | | | T := T ∪ {Y}.
   | | 3. X is a scalar object
   | | | Skip.
   RELEASE1.
```

A comparison of Algorithm G.TE and G.VE is worthwhile. For that purpose consider a reachable object X. In both algorithms X is traced the

same number of times. Upon tracing X for the first time in Algorithm G.TE a test is performed to see whether $X \in M$ (answer: no), X is marked and X is put in T. Some time later X is fetched from T and subsequently visited. Upon tracing X for the first time in Algorithm G.VE X is put in T immediately. After a while X is fetched from T, a test is performed to see whether $X \in M$ (answer: no), X is marked and subsequently visited. So, in contrast to Algorithm G.TE marking and visiting X is done on the same occasion in Algorithm G.VE. Since both marking and visiting an object will involve accessing the object and the accesses can be combined in Algorithm G.VE, Algorithm G.VE can be regarded as being more efficient than Algorithm G.TE in this respect. This only applies to the actions related to the first time an object is traced. Upon tracing an object X for the k-th time $(k > 1)$, Algorithm G.TE tests whether $X \in M$, finds X to be marked and proceeds. Algorithm G.VE puts X in T first. After fetching X from T some time later it tests whether $X \in M$, finds X to be marked and proceeds. The conclusion is that Algorithm G.VE will only be more efficient than Algorithm G.TE if the majority of reachable objects is traced only once. (This is so if large parts of the graph have a tree structure.) In most cases Algorithm G.TE will be more efficient than Algorithm G.VE, not only with respect to time but also space: The size of T will generally be larger in Algorithm G.VE than in Algorithm G.TE. Notice also that, in contrast to Algorithm G.TE, in Algorithm G.VE objects may occur more than once in the bag T (which excludes an implementation of T as a linked list through the locations of objects in T, unless tests for double occurrences of objects in T are performed).

An obvious optimization can be obtained by visiting atomic objects immediately after they are traced (detail S). Thus Algorithm G.VES is obtained where T contains only structured objects.

```
Algorithm G.VES

Variables:
   M: set of objects,
   T: bag of objects.
Action:
   M,T := ∅,{R}.
   While T ≠ ∅
   | Get X from T.
   | If X ∉ M
   | | M := M ∪ {X}.
   | | For each Y ∈ struct(X)
   | | | While Y is a reference object and Y ∉ M
   | | | | M := M ∪ {Y}.
   | | | | Y := obj(val(Y)).
   | | | Case
   | | | 1. Y is a structured object
   | | | | T := T ∪ {Y}.
   | | | 2. Y is an atomic object
   | | | | If Y ∉ M
   | | | | | M := M ∪ {Y}.
   RELEASE1.
```

In each algorithm presented here (except Algorithm G) the number of (more or less) primitive operations performed is proportional to the number of tests of the kind "$X \notin M$". (Check this.) This implies that the number of these tests performed in an algorithm is a good indication for the time-

complexity of the algorithm. Assuming that the number of objects, of which an object is a direct component, is bounded by a constant, check that all algorithms operate in a time $O(n)$, where n is the number of reachable objects. Check also that if structured objects have a tree structure and at least two direct components, we can even take the number of reachable atomic objects for n. $O(n)$ garbage collection algorithms are as good as we can expect.

5.1.3. Garbage collection algorithms using atom marking

In implementations featuring references to components of nodes, atom marking is the most frequently used way of marking. The main reason for that is probably that atomic objects in the graph and cells in the store are somewhat analogous. This makes algorithms using atom marking easier to implement in terms of operations on the store. From a conceptual point of view, on the other hand, atom marking is more complex than object marking. In atom marking it is not possible to mark structured objects individually. This implies that a marking operation on a structured object must in some way be modelled in terms of marking operations on the atoms of the object, which leads to a number of complications.

The job of a garbage collector using atom marking is to mark all reachable atomic objects and subsequently use this information to deallocate all storage occupied by isolated nodes. Having marked all reachable atomic objects, the garbage collector can determine whether a node is isolated by inspecting all of its atoms and seeing if one of them is marked. The marking information will be represented by an abstract variable M. Instead of a set of objects, this variable is a set of atomic objects now. Using M the job of the garbage collector can be described as follows:

```
M := {X | X is a reachable atomic object}.
RELEASE2.
```

where

```
RELEASE2:
    For each node X ∈ dom(A)
    | If M ∩ atoms(X) = ∅
    | | dom(A) := dom(A) \ sub(X).
```

Notice that in contrast to object marking the overhead caused by the marking information is limited: A typical way to implement M is to reserve a bit in each cell of the store, indicating whether the atomic object located there is marked or not. Another frequently used possibility, especially if there is no room for mark bits in the cells of the store, is to reserve a compact location in the store to be used as a "bit map". Each cell of the store is mapped onto a bit in this piece of storage, which can be used as a mark bit for the cell.

The garbage collection algorithms using atom marking will be derived from those using object marking by imposing either detail A or B. Using the other details (except C and D) in the list of algorithm details given in Subsection 5.1.1.7, a rather complex hierarchy of algorithms can be constructed. Not all algorithms constituting the hierarchy will be discussed. Instead a representative subset of these algorithms will be discussed. Using these algorithms as a basis it will not be difficult to reconstruct the other algorithms in the hierarchy. The method of discussion

of the algorithms will be the same as with the garbage collection algorithms using object marking.

The first algorithm to be discussed can be derived immediately from Algorithm G.T by replacing the marking operation of a structured object by an empty action (detail A):

```
Algorithm G.AT

Variables:
   M: set of atomic objects,
   T: bag of pairs (object, set of direct components).
Action:
   M,T := ∅,{(R,struct(R))}.
   While T ≠ ∅
   | Get (X,V) from T.
   | Case
   | 1. X is a structured object
   | | If V ≠ ∅
   | | | Get Y from V.
   | | | T := T ∪ {(X,V)}.
   | | | INSPECT(Y).
   | 2. X is a reference object
   | | Let Y = obj(val(X)).
   | | INSPECT(Y).
   | 3. X is a scalar object
   | | Skip.
   RELEASE2.

INSPECT(Y):
   Case
   1. Y is a structured object
   | T := T ∪ {(Y,struct(Y))}.
   2. Y is an atomic object
   | If Y ∉ M
   | | M,T := M ∪ {Y},T ∪ {(Y,∅)}.
```

This algorithm (in a "tuned" and usually recursive form) has been the basis of several garbage collection algorithms designed for ALGOL 68 implementations [BRANQUART & LEWI 71], [MARSHALL 71], [WODON 71], [ROBSON 74]. One of the complications inherent in atom marking is already apparent from this algorithm: Upon tracing a structured object *Y* it is not possible to determine whether *Y* has been traced or even visited before. Consequently, the first of the three restrictions mentioned in Subsection 5.1.1.4 is not satisfied in the above algorithm (nor in any of the other algorithms using atom marking). Each time a structured object is traced it will also be visited (as many times as its number of direct components). Check that the other two restrictions from Subsection 5.1.1.4 are still satisfied, thus guaranteeing the termination of the marking process.

There are various ways to speed up Algorithm G.AT. A first way is to combine the tracing and visiting of the components of a (structured) object *X* with the visit to *X* (details E and H). A second way is to visit all atomic objects immediately after they are traced (detail S). A third way is to visit all objects without branches immediately after they are traced (detail Q). All these optimizations are included in the following algorithm:

```
Algorithm G.ATEHSQ

Variables:
   M: set of atomic objects,
   T: bag of objects.
Action:
   M,T := ∅,{R}.
   While T ≠ ∅
   | Get X from T.
   | For each Y ∈ atoms(X)
   | | While Y is a reference object and Y ∉ M
   | | | M := M ∪ {Y}.
   | | | Y := obj(val(Y)).
   | | Case
   | | 1. Y is a structured object
   | | | If branches(Y) = ∅
   | | | | For each Z ∈ atoms(Y)
   | | | | | If Z ∉ M
   | | | | | | M := M ∪ {Z}.
   | | | else
   | | | | T := T ∪ {Y}.
   | | 2. Y is an atomic object
   | | | If Y ∉ M
   | | | | M := M ∪ {Y}.
   RELEASE2.
```

This algorithm requires the possibility of directly determining the set of atoms (or, in lower level terms, the location) of an object, which causes an overhead when used with "type tracking" (see Subsection 5.1.1.6). The fact that it must be possible to determine that an object has no branches may also introduce some overhead.

As with the algorithms using object marking, a distinction can be made between "marking after tracing" and "marking before visiting" algorithms. An example of the latter kind is given below.

```
Algorithm G.AVE

Variables:
   M: set of atomic objects,
   T: bag of objects.
Action:
   M,T := ∅,{R}.
   While T ≠ ∅
   | Get X from T.
   | Case
   | 1. X is a structured object
   | | For each Y ∈ struct(X)
   | | | T := T ∪ {Y}.
   | 2. X is a reference object
   | | If X ∉ M
   | | | M := M ∪ {X}.
   | | | Let Y = obj(val(X)).
   | | | T := T ∪ {Y}.
   | 3. X is a scalar object
   | | If X ∉ M
   | | | M := M ∪ {X}.
   RELEASE2.
```

This algorithm can be derived from Algorithm G.VE by imposing detail A. Remarks analogous to those made in comparing Algorithms G.TE and G.VE apply to a comparison of Algorithms G.ATE (which has not been discussed, but can easily be derived from Algorithm G.AT) and G.AVE.

In all algorithms discussed so far the marking operation of a structured object was modelled by an empty action (detail A). Another approach is to model the operation by marking all atoms of the object (detail B). This approach is not without problems, because the marking information can no longer be used to determine whether a reference object has been traced before. This is compensated for somewhat by the fact that unnecessary visits to objects traced by dereferencing can be avoided by testing if all their atoms are marked. Using a rather complex transformation the following algorithm featuring detail B can be derived from Algorithm G.TS:

```
Algorithm G.BTS

Variables:
   M: set of atomic objects,
   T: bag of pairs (object, set of direct components).
Action:
   M,T := atoms(R),{(R,struct(R))}.
   While T ≠ ∅
   | Get (X,V) from T.
   | Get Y from V.
   | If V ≠ ∅
   | | T := T ∪ {(X,V)}.
   | Case
   | 1. Y is a structured object
   | | T := T ∪ {(Y,struct(Y))}.
   | 2. Y is a reference object
   | | Y := obj(val(Y)).
   | | While Y is a reference object and Y ∉ M
   | | | M := M ∪ {Y}.
   | | | Y := obj(val(Y)).
   | | Case
   | | 1. Y is a structured object
   | | | Let B = false.
   | | | For each Z ∈ atoms(Y)
   | | | | If Z ∉ M
   | | | | | M := M ∪ {Z}.
   | | | | | B := true.
   | | | If B
   | | | | T := T ∪ {(Y,struct(Y))}.
   | | | 2. Y is an atomic object
   | | | | If Y ∉ M
   | | | | | M := M ∪ {Y}.
   | | 3. Y is a scalar object
   | | | Skip.
   | RELEASE2.
```

This algorithm has been described (in a more concrete form) in [WEGBREIT 72]. Apart from the first restriction from Subsection 5.1.1.4 the algorithm does not satisfy the second restriction either (i.e., reference objects may have to be visited and dereferenced more than once).

EXAMPLE 5.4

Consider Figure 5.3, where X, X_1, X_2 and Y are as yet unmarked objects, which are reachable through the references V and W.

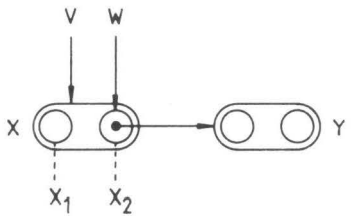


Figure 5.3

If X is traced (through V) before X_2 (through W) in Algorithm G.BTS, then X_1 and X_2 are marked and $(X,struct(X))$ is put in T, which guarantees that some time later X_2 will be visited and dereferenced. If, after tracing X, X_2 is traced through W, X_2 is found to be marked and will not be visited a second time. If, however, X_2 is traced through W before X is traced through V, then X_2 is marked, visited and dereferenced, all atoms of Y are marked and $(Y,struct(Y))$ is put in T. If thereupon X is traced through V, the atom X_1 of X is found to be unmarked. Hence X_1 is marked and $(X,struct(X))$ is put in T. The consequence of the latter is that some time later X_2 will be visited and dereferenced for the second time. In contrast to the first time all atoms of Y are now found to be marked and $(Y,struct(Y))$ is not put in T again. □

Check that even though reference objects are visited and dereferenced more than once, Algorithm G.BTS is still guaranteed to terminate.

The fact that in algorithms featuring detail B, references may have to be "followed" more than once, may have a serious impact on the efficiency of these algorithms. It is not so much the dereferencing operation itself which may take a considerable time, but the testing of all atoms of the object referred to by a reference, to see whether all these atoms are marked (which they always are after the reference has been followed for the first time). The algorithms of this kind can therefore be speeded up considerably if reference objects are "flagged" as soon as they are visited (detail F). The flag associated with a reference object can then be used to avoid visiting and dereferencing the object more than once. The price to be paid for this is a space overhead per reference object. This price may be nil if there is a spare bit in the locations (cells) occupied by reference objects. The following algorithm is an example where this principle is applied. The flags are represented by an abstract variable F, which is a set of reference objects.

```
Algorithm G.BTEHF

Variables:
   M: set of atomic objects,
   T: bag of objects,
   F: set of reference objects.
Action:
   M,T,F := atoms(R),{R},∅.
   While T ≠ ∅
   | Get X from T.
   | For each Y ∈ branches(X)
   | | If Y ∉ F
   | | | F := F ∪ {Y}.
   | | | Let Z = obj(val(Y)).
   | | | Let B = false.
   | | | For each W ∈ atoms(Z)
   | | | | If W ∉ M
   | | | | | M := M ∪ {W}.
   | | | | | B := true.
   | | | If B
   | | | | T := T ∪ {Z}.
   RELEASE2.
```

This algorithm, in fact, uses a mixture of atom and reference marking (see Subsection 5.1.4).

Though Algorithm G.BTEHF can be considered to be rather efficient, it will still perform $O(n^2)$ tests of the kind "$X \notin M$" in its worst case behaviour, where n is the number of reachable atomic objects. This assertion holds for all algorithms using atom marking, even if all objects have a tree structure. The reason for this is that each time an object is traced, all of its atoms must be inspected to see if one of them is unmarked. Though in practice things may not prove that bad, it makes sense to use one of the other kinds of marking whenever (efficiently) possible.

EXAMPLE 5.5

Consider the tree structured object X in Figure 5.4, which is reachable through the reference V only.

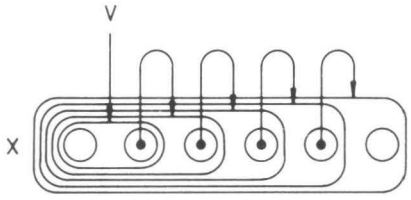


Figure 5.4

X can easily be generalized to an object with n atoms. Check that all algorithms using atom marking presented here perform $O(n^2)$ tests of the kind "$Y \notin M$" on the atoms of X. □

5.1.4. Garbage collection algorithms using reference marking

Reference marking is a rather unusual, yet interesting way of marking. In algorithms using reference marking, only the reachable reference objects are marked. This implies that after marking the fact of whether or not a node is isolated can only be determined by inspecting all marked reference objects to see whether they contain a reference to a subobject of the node. This indirect way of determining whether a node is isolated or not makes algorithms using reference marking not very suitable for use as stand-alone garbage collection algorithms. In combination with compaction algorithms, efficient compacting garbage collectors can be constructed (see Subsection 5.3).

The marking information in the algorithms using reference marking will be represented by a variable set M of reference objects. Using M the job of a garbage collector using reference marking can be described as follows:

$M := \{X \mid X$ is a reachable reference object$\}$.
$RELEASE_3$.

where

$RELEASE_3$:
  For each node $X \in dom(A)$, $X \neq R$
  | If $\{obj(val(Y)) \mid Y \in M\} \cap sub(X) = \emptyset$
  | | $dom(A) := dom(A) \setminus sub(X)$.

Notice that the overhead caused by the marking information can be considerably less than with atom marking, let alone object marking. If references are represented by addresses and an address does not occupy a full word, the space overhead may even be nil: One of the redundant bits in the cell occupied by a reference object can be used to indicate whether the object is marked or not. Notice also that the complicated test to see whether a node is isolated can be performed efficiently, if each time a reference V is followed during the marking process, the node containing $obj(V)$ is "flagged". This technique, by the way, can also be used when using object or atom marking. It requires an extra space overhead per node X (for the flag of X) and per reference object Y (in order to determine $node(obj(val(Y)))$ from $val(Y)$).

Garbage collection algorithms using reference marking will be derived from those using object marking by imposing detail C. The hierarchy of algorithms which can be obtained using the other details listed in Subsection 5.1.1.7 is again rather complex. Because of the rather limited applicability of reference marking only a few algorithms from this hierarchy will be discussed.

Modelling the marking operation of an object which is not a reference object by an empty action (detail C) in Algorithm G.T leads to:

```
Algorithm G.CT

Variables:
   M: set of reference objects,
   T: bag of pairs (object, set of direct components).
Action:
   M,T := ∅,{(R,struct(R))}.
   While T ≠ ∅
   | Get (X,V) from T.
   | Case
   | 1. X is a structured object
   | | If V ≠ ∅
   | | | Get Y from V.
   | | | T := T ∪ {(X,V)}.
   | | | INSPECT(Y).
   | 2. X is a reference object
   | | Let Y = obj(val(X)).
   | | INSPECT(Y).
   | 3. X is a scalar object
   | | Skip.
   RELEASE_3.

INSPECT(Y):
   Case
   1. Y is a structured object
   | T := T ∪ {(Y,struct(Y))}.
   2. Y is a reference object
   | If Y ∉ M
   | | M,T := M ∪ {Y},T ∪ {(Y,∅)}.
   3. Y is a scalar object
   | T := T ∪ {(Y,∅)}.
```

This algorithm satisfies only the second of the three restrictions from Subsection 5.1.1.4. Check that it terminates nevertheless.

The inefficiencies caused by repeatedly and unnecessarily visiting structured and scalar objects in Algorithm G.CT can be greatly reduced by combining the visits to the structured and scalar components of an object X with the visit to X (details E and R). This results in the following algorithm, where only reference objects occur in the bag T:

```
Algorithm G.CTER

Variables:
   M: set of reference objects,
   T: bag of reference objects.
Action:
   M,T := branches(R),branches(R).
   While T ≠ ∅
   | Get X from T.
   | Let Y = obj(val(X)).
   | For each Z ∈ branches(Y)
   | | If Z ∉ M
   | | | M,T := M ∪ {Z},T ∪ {Z}.
   RELEASE_3.
```

This simple algorithm has been described in [ZAVE 73], where the bag T is implemented as a linked list, requiring a space overhead per reference

object. Reference typing is used in order to be able to determine the branches of an object referred to by a reference, requiring an additional overhead per reference object. This overhead is only acceptable if the number of reference objects as compared to the total number of atomic objects is not too large (as in ALGOL 68, but not in LISP).

The algorithms using reference marking, like the algorithms using atom marking, all use a quadratic time in their worst case behaviour. I.e., in pathological cases (see Example 5.5) they perform $O(n^2)$ tests of the kind "$X \notin M$", where n is the number of reachable reference objects. Nevertheless, garbage collectors using reference marking can be considerably faster than those using atom marking. The $O(n^2)$ garbage collection time is the penalty to be paid for the reduction of the space overhead of the marking information and the occurrence of references to arbitrary subnodes. The garbage collection time can be reduced to $O(n)$ either by increasing the space overhead caused by the marking information (using object marking) or by putting restrictions on the occurrence of references in the graph. The latter will be done in the next subsection.

5.1.5. Garbage collection algorithms using node marking

In implementations of many programming languages (especially list processing languages) only references to nodes occur in the graph (problem detail N). This is true for example for LISP, where the concept of a garbage collector originated. If references refer only to nodes the problem of garbage collection is simplified to a great extent. The reason for this is that the graph has a much simpler structure: It can be viewed as a graph in the pure graph-theoretical sense. Moreover, garbage collection can be much more efficient: By choosing nodes as marking units the space overhead can be reduced to a nodewise overhead, which is usually better than an overhead per object, atomic object or reference object, while a garbage collection can still be performed in a time linear in the total number of branches of reachable nodes. Therefore, if all references contained in the graph refer to nodes, node marking is the marking method of choice in a garbage collector.

Strictly speaking, node marking cannot be applied if references also refer to components of nodes. The reason is, first of all, that the marking information would be insufficient to determine which nodes are isolated: Nodes which are not marked may have reachable components. This problem could be solved by "flagging" nodes as soon as one of their subobjects is traced, in the same way as discussed in Subsection 5.1.4. The second and more important reason is that the garbage collector would still need a way to indicate that an arbitrary object has already been visited (otherwise the algorithm may not terminate). The latter would amount to object marking instead of node marking, however. The situation can be "remedied" by letting the garbage collector look at references in a way different from that of the program. That is, the garbage collector considers a reference of an object as referring not to the object, but to the node of which the object is a subobject. From the garbage collector point of view all references now refer to nodes and node marking can be applied.

The problem with the above approach is that reachability from the program point of view and reachability from the garbage collector point of view are two different things now. An object which is reachable from the program point of view will always be a subobject of a node which is reachable from the garbage collector point of view. The converse is not true. A node which is reachable from the garbage collector point of view need not necessarily contain a subobject which is reachable from the

program point of view. Consequently, the garbage collector may fail to deallocate storage occupied by objects which are truly garbage from the program point of view. This may have disastrous effects (see Example 5.6).

EXAMPLE 5.6

Suppose Figure 5.5 is a picture of the graph.

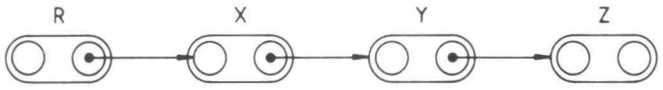


Figure 5.5

From the program point of view only the root R and one direct component of the node X are reachable. When using node marking, the garbage collector would consider the entire node X, and consequently the nodes Y and Z, to be reachable. The nodes Y and Z are thus preserved, even though they are pure garbage. □

An extra overhead is also introduced because of the different way references are interpreted by the program and the garbage collector. (The garbage collector must be able to determine the node, in which an object referred to by a reference is contained.) Yet, depending upon the specific implementation, these drawbacks may very well be outweighed by the simplicity and efficiency gained by the use of node marking. Though in all algorithms using node marking the graph will be assumed to contain only references to nodes, these algorithms may also be applied in the case of references to components of nodes, provided one uses the garbage collector's interpretation of reachability.

In the algorithms using node marking, the marking information will be represented by a variable set M of nodes. Using M, the job of a garbage collector using node marking is described by:

M := {X | X is a reachable node}.
$RELEASE_4$.

where

```
RELEASE4:
    For each node X ∈ dom(A)
    | If X ∉ M
    | | dom(A) := dom(A) \ sub(X).
```

As can be seen, the test whether a node is isolated is very simple: Being isolated is equivalent to being unreachable, which is equivalent to being unmarked. The marking information is typically implemented by a mark bit in the location of a node. If there is no room for that and all nodes have the same size, a bit map can also be used. (See also Algorithm GNK.DTEH*.)

From the hierarchy of garbage collection algorithms using node marking (which all feature detail D) only the most important algorithms will be discussed. The first algorithm to be discussed can be derived directly from Algorithm G.T by adding detail N to the problem and imposing detail D on the algorithm:

```
Algorithm GN.DT

Variables:
   M: set of nodes,
   T: bag of pairs (object, set of direct components).
Action:
   M,T := {R},{(R,struct(R))}.
   While T ≠ ∅
   | Get (X,V) from T.
   | Case
   | 1. X is a structured object
   | | If V ≠ ∅
   | | | Get Y from V.
   | | | T := T ∪ {(X,V)}.
   | | | Case
   | | | 1. Y is a structured object
   | | | | T := T ∪ {(Y,struct(Y))}.
   | | | 2. Y is an atomic object
   | | | | T := T ∪ {(Y,∅)}.
   | 2. X is a reference object
   | | Let Y = obj(val(X)).
   | | If Y ∉ M
   | | | M,T := M ∪ {Y},T ∪ {(Y,struct(Y))}.
   | 3. X is a scalar object
   | | Skip.
   | RELEASE4.
```

The above algorithm satisfies the second and third restriction from Subsection 5.1.1.4. The first restriction is only partially met (i.e., for nodes X and their direct components Y).

The only reason to trace and visit separately the direct components of an object in Algorithm GN.DT is to find the branches of the object. This way of tracing and visiting objects makes the algorithm suitable for use with type tracking. If it is possible to determine the branches of a node directly (possibly at the expense of some space overhead) Algorithm GN.DT can be speeded up substantially. Directly determining (tracing) the branches of a node amounts to tracing all components of a node during a visit to the node. In doing so it makes sense to either stop the tracing at the branches of the node and visit nodes immediately after they are traced (details E and R), or visit the branches immediately after they are traced and stop the tracing at the nodes (details E and H). Notice that these two possibilities exclude each other. To start with, consider the first possibility:

```
Algorithm GN.DTER

Variables:
   M: set of nodes,
   T: bag of reference objects.
Action:
   M,T := {R},branches(R).
   While T ≠ ∅
   | Get X from T.
   | Let Y = obj(val(X)).
   | If Y ∉ M
   | | M := M ∪ {Y}.
   | | For each Z ∈ branches(Y)
   | | | T := T ∪ {Z}.
   RELEASE4.
```

This algorithm is the basis of many practical garbage collection algorithms. We shall therefore discuss a number of important variants of this algorithm, which are of increasing concreteness.

An important question is raised by the implementation of the bag T. A naive implementation of T (such as a linked list) requires a space overhead per reference object, which may be prohibitive in case of a high "density" of reference objects. Due to problem detail N the space overhead caused by the implementation of T can be reduced to a nodewise overhead (in contrast to Algorithm G.CTER). In order to illustrate this, T will first be implemented in terms of two variables U and C, where U is a bag of nodes and C is a (partial) mapping from nodes to sets of reference objects, according to the following implementation invariants:

$$C(X) \subset branches(X) \quad (X \in U).$$

$$T = \{Y \in C(X) \mid X \in U\}.$$

Thus the following algorithm is obtained (which is essentially Algorithm 3 from Chapter 3):

```
Algorithm GN.DTER*

Variables:
   M: set of nodes,
   U: bag of nodes,
   C: mapping from nodes to sets of reference objects.
Action:
   M,U,C(R) := {R},{R},branches(R).
   While U ≠ ∅
   | Let X ∈ U.
   | If C(X) ≠ ∅
   | | Get Y from C(X).
   | | Let Z = obj(val(Y)).
   | | If Z ∉ M
   | | | M,U,C(Z) := M ∪ {Z},U ∪ {Z},branches(Z).
   | else
   | | U := U \ {X}.
   RELEASE4.
```

A more concrete version of this algorithm can be obtained by implementing U as a stack S, introducing problem detail K and implementing C as a mapping

k from nodes to integers, according to the following implementation invariants:

If $S = \langle X_1,\ldots,X_n\rangle$, then $U = \{X_1,\ldots,X_n\}$.

$C(X) = \{branch(X,i) \mid k(X) < i \leq degree(X)\}$ $\quad (X \in U)$.

This results in the following algorithm (which is essentially Algorithm 5 from Chapter 3):

```
Algorithm GNK.DTER**

Variables:
   M: set of nodes,
   S: stack of nodes,
   k: mapping from nodes to integers.
Action:
   M,S,k(R) := {R},<R>,0.
   While S ≠ <>
   | Let X = TOP(S).
   | If k(X) ≠ degree(X)
   | | k(X) := k(X) + 1.
   | | Let Y = branch(X,k(X)).
   | | Let Z = obj(val(Y)).
   | | If Z ∉ M
   | | | M,S,k(Z) := M ∪ {Z},PUSH(Z,S),0.
   | else
   | | S := POP(S).
   RELEASE4.
```

This algorithm can be viewed more or less as the "depth-first marking paradigm". It is a frequently used algorithm, which is usually described in its recursive form (concealing the stack S). The algorithm requires only a nodewise space overhead: The integer $k(X)$ associated with a node X is small ($\leq degree(X)$) and can be assumed to occupy a constant space (independent of X). The surprising thing is that the space overhead caused by the stack and the marking information can be eliminated (almost) entirely. This is sketched below.

It is easy to infer from Algorithm GNK.DTER** that if there is an object X on top of the stack and an object Y is pushed on top of it, then the $k(X)$-th branch of X contains the reference of Y as its value. This observation implies that between two strokes of the marking process the stack S looks as pictured in Figure 3.3.a (where all nodes are assumed to have four branches). Using two variable values p and q this situation can be transformed without loss of information into the situation of Figure 3.3.b. The cross in this picture is a dummy value, which will be denoted by nil. In Figure 3.3.b the stack S has become entirely superfluous. The garbage collection algorithm based on this idea has become known as the "Deutsch-Schorr-Waite algorithm" and was first described in [SCHORR & WAITE 67].

The Deutsch-Schorr-Waite algorithm can be derived very easily from Algorithm GNK.DTER** by expressing the operations on S in terms of the implementation of S as sketched in Figure 3.3.b. The operations on S performed in Algorithm GNK.DTER** and their "translations" are given below:

$S := \langle R\rangle \longrightarrow$
$\quad p,q := ref(R),nil$

```
S ≠ <> ⟶
     p ≠ nil

Let X = TOP(S) ⟶
     Let X = obj(p)

S := PUSH(Z,S) ⟶
     p,q,val(Y) := val(Y),p,q

S := POP(S) ⟶
     If q = nil
     | p := nil.
     else
     | Let Y = obj(q).
     | Let Z = branch(Y,k(Y)).
     | p,q,val(Z) := q,val(Z),p
```

The fact that $0 \leq k(X) \leq degree(X)$ for each marked node X, and the fact that $k(X)$ is not affected for any unmarked node X, can be used to encode M in the mapping k: Simply initialize all $k(X)$ to $degree(X)+1$ (or any other number < 0 or $> degree(X)$). The test "$X \notin M$" is then equivalent to "$k(X) = degree(X)+1$", making M redundant. Thus, through a simple substitution process we obtain the Deutsch-Schorr-Waite algorithm (see also Algorithm 6 in Chapter 3):

```
Algorithm GNK.DTER***

Variables:
   k: mapping from nodes to integers,
   p,q: values.
Action:
   Let nil be a scalar value.
   For each node X ∈ dom(A)
   | k(X) := degree(X) + 1.
   p,q,k(R) := ref(R),nil,0.
   While p ≠ nil
   | Let X = obj(p).
   | If k(X) ≠ degree(X)
   | | k(X) := k(X) + 1.
   | | Let Y = branch(X,k(X)).
   | | Let Z = obj(val(Y)).
   | | If k(Z) = degree(Z) + 1
   | | | p,q,val(Y),k(Z) := val(Y),p,q,0.
   | else
   | | If q = nil
   | | | p := nil.
   | | else
   | | | Let Y = obj(q).
   | | | Let Z = branch(Y,k(Y)).
   | | | p,q,val(Z) := q,val(Z),p.
   RELEASE*4.

RELEASE*4:
   For each node X ∈ dom(A)
   | If k(X) = degree(X) + 1.
   | | dom(A) := dom(A) \ sub(X).
```

Notice that, while traversing the nodes in $RELEASE_4^*$, the counter $k(X)$ of each (reachable) node X can be re-initialized to $degree(X) + 1$. If the storage manager takes care that, when creating an object X, $k(X)$ is set to $degree(X) + 1$ as well, the initialization loop for k at the beginning of the algorithm can be skipped.

A new phenomenon can be observed in this algorithm. In all algorithms discussed before none of the components of the storage management system H was affected (not even temporarily), except A (as prescribed by the definition of *COLLECT GARBAGE*). As a consequence of this all system invariants are satisfied everywhere in these algorithms. In the above algorithm, apart from A, the graph G is also affected: In the algorithm the value of reference objects is changed, thus disturbing a number of system invariants (for example, (S5)). We had better be sure that the changes made in G are only temporary and that at the end of the algorithm, G is the same as before (thus at the same time restoring the system invariants). The fact that proving this, and the correctness of the above algorithm in general, is not a sinecure is demonstrated by the comprehensive literature on the subject [DE ROEVER 78], [DUNCAN & YELOWITZ 79], [GERHART 79], [GRIES 79], [KOWALTOWSKI 79], [TOPOR 79], [DERSHOWITZ 80]. A detailed proof of correctness of the above algorithm (without $RELEASE_4^*$ and the implementation trick for M) can also be found in Chapter 3.

Let us now consider the second way of speeding up Algorithm GN.DT, i.e., the introduction of details E and H. Instead of nodes (as in Algorithm GN.DTER) reference objects are then visited immediately after they are traced:

```
Algorithm GN.DTEH

Variables:
   M: set of nodes,
   T: bag of nodes.
Action:
   M,T := {R},{R}.
   While T ≠ ∅
   | Get X from T.
   | For each Y ∈ branches(X)
   | | Let Z = obj(val(Y)).
   | | If Z ∉ M
   | | | M,T := M ∪ {Z},T ∪ {Z}.
   RELEASE4.
```

This is essentially the algorithm described in [THORELLI 72], p. 560. Like Algorithm GN.DTER this algorithm is the basis of many practical garbage collection algorithms. It has the advantage over Algorithm GN.DTER that the bag T contains nodes instead of reference objects, which makes it easy to see that the algorithm requires only a nodewise space overhead. A particularly efficient implementation of Algorithm GN.DTEH is possible if T is implemented as a stack in linked list representation. (Check that nodes do not occur twice in T.) The marking information can then be encoded in the "link field" of a node, as described in:

```
Algorithm GNK.DTEH*

Variables:
   ℓ: mapping from nodes to values,
   p: value.
Action:
   Let nil and unmarked be different scalar values.
   For each node X ∈ dom(A)
   | ℓ(X) := unmarked.
   p,ℓ(R) := ref(R),nil.
   While p ≠ nil
   | Let X = obj(p).
   | p := ℓ(X).
   | For k = 1 to degree(X)
   | | Let Y = branch(X,k).
   | | Let Z = obj(val(Y)).
   | | If ℓ(Z) = unmarked
   | | | p,ℓ(Z) := ref(Z),p.
   RELEASE4**.

RELEASE4**:
   For each node X ∈ dom(A)
   | If ℓ(X) = unmarked
   | | dom(A) := dom(A) \ sub(X).
```

This algorithm is the first part of the so-called "LISP 2 garbage collector" (see the solution of Exercise 2.5.33 in [KNUTH 68]). Notice that the initialization loop for ℓ can again be eliminated (see the remark below Algorithm GNK.DTER***). Notice also that if we had used a queue discipline instead of a stack discipline for the implementation of T, a list of all reachable nodes emanating from the root R would have remained after the marking phase (see [THORELLI 72], p. 563). This list could be used to speed up $RELEASE_4^{**}$.

It is interesting to compare Algorithm GNK.DTEH* (LISP 2) with Algorithm GNK.DTER*** (Deutsch-Schorr-Waite). The first thing to be noted is that Algorithm GNK.DTEH* will generally be considerably faster than Algorithm GNK.DTER***. There are basically two reasons for this. The first is that in Algorithm GNK.DTEH* all branches of a node are visited at the same time while in Algorithm GNK.DTER*** they are visited on separate occasions. The latter requires the recording (in the variable k) of information concerning the branches of a node still to be visited. The second reason is the rather complicated implementation of the stack S (from Algorithm GNK.DTER**) in Algorithm GNK.DTER***. Though this implementation causes no space overhead, the coding and decoding operations used are rather expensive. (This can be compensated for somewhat by using Algorithm GNK.DTER** with a small finite stack S and changing to Algorithm GNK.DTER*** if this stack is full [SCHORR & WAITE 67].) In Algorithm GNK.DTEH* the bag T (from Algorithm GN.DTEH) is implemented in a simple and efficient way, albeit that this implementation requires a space overhead.

This brings us to the space requirements of both algorithms. At first sight there does not seem to be much difference: Both algorithms require space for a mapping from nodes to values. These mappings are typically implemented by reserving space in the location of each node X, in which (the representation of) $k(X)$ or $\ell(X)$ is stored. The difference is that $k(X)$ is an integer in the range $(0,degree(X)+1)$, while $\ell(X)$ is an arbitrary scalar or reference value. If the number of branches of nodes is small (for

example 2, as in pure LISP), $k(X)$ can often be encoded in a few unused bits in the location of X. If the number of branches of nodes is large, or even potentially infinite, a full extra cell in the location of X will be necessary to store $k(X)$. An extra cell will usually also be necessary to store $\ell(X)$. Summarizing, we can say that Algorithm GNK.DTER*** should only be preferred to Algorithm GNK.DTEH* if speed is not all important and if the number of branches per node X is so small, that the counter $k(X)$ causes little or no space overhead.

The last garbage collection algorithm to be discussed is an example of an algorithm which combines marking with visiting instead of tracing (detail V):

```
Algorithm GN.DVEH

Variables:
   M: set of nodes,
   T: bag of nodes.
Action:
   M,T := ∅,{R}.
   While T ≠ ∅
   | Get X from T.
   | If X ∉ M
   | | M := M ∪ {X}.
   | | For each Y ∈ branches(X)
   | | | Let Z = obj(val(Y)).
   | | | T := T ∪ {Z}.
   RELEASE4.
```

This algorithm corresponds to the algorithm described in [THORELLI 72], p. 556. Remarks analogous to those made in comparing Algorithms G.TE and G.VE (see Section 5.1.2) apply to a comparison of Algorithms GN.DTEH and GN.DVEH.

All algorithms which have been discussed in this subsection perform $O(n)$ tests of the kind "$X \notin M$", where n is the number of reachable reference objects (= total number of branches of reachable nodes). Check that, except for Algorithm GN.DT, the number of these tests performed in an algorithm is proportional to the number of "primitive" operations performed (i.e., operations which can be implemented in such a way that they take $O(1)$ time). The reason why this is not so for Algorithm GN.DT is that the branches of a node are determined indirectly there. If the branches of a node can be determined directly, garbage collection using node marking takes $O(n)$ time, which makes it the fastest of the four methods (when applicable). Still, there are considerable differences in speed between the various algorithms using node marking, as we have seen.

5.2. COMPACTION

5.2.1. General discussion

5.2.1.1. Moving

The definition of the operation *COMPACT* as given in Chapter 4 implies that it is the job of a compacter to establish the compactness of a storage management system by reallocating objects and changing the contents of cells, while not affecting the system invariants and the layout of objects. The fact that a compacter has to change the locations allocated to objects

and the contents of cells so as to establish the compactness of the storage management system implies that, in principle, many system invariants and the layout of many objects may be disturbed during a compaction. The problems caused by this are greatly reduced if a compaction is performed in terms of simple operations which do not affect any system invariants or the layout of any objects. Unfortunately, such simple (in the sense of easily implementable) operations do not exist. An operation which comes very close is the following. It "moves" a node (and all of its components) to a new location not occupied by another node, thereby copying the contents of the old location to the new location. This basic operation will be used in all compaction algorithms to be discussed:

```
MOVE(X,a):
   Precondition:
      X is a node, X ∈ dom(A),
      a is an integer, left(S) ≤ a ≤ left(A(X)),
      For each node Y ∈ dom(A), Y ≠ X
      | A(Y) ∩ shift(A(X),a - left(A(X))) = ∅.
   Action:
      Let b = left(A(X)).
      Let s = b - a.
      For each Y ∈ sub(X)
      | A(Y) := shift(A(Y),-s).
      For i = 0 to size(A(X)) - 1
      | Let C = cell(a + i).
      | Let D = cell(b + i).
      | cont(C) := cont(D).
```

The effect of *MOVE(X,a)* is that the node *X* is moved to a location with left address *a*. The operation is defined for moves to the left only. It could have been defined for moves to the right as well, but we shall not need the latter in any of the algorithms to be discussed. The reason, of course, is that a compacter must move all nodes to a compact location in the left part of the store (see the definition of "compact" storage management system). Notice that *MOVE* is indeed a "simple" operation: The abstract operations on A reduce to empty actions in most (but not necessarily all) implementations. What remains is the copying of a block of storage, for which many concrete machines even have special instructions.

5.2.1.2. Updating

The operation *MOVE*, as can be checked without great difficulty, does not affect the layout of any objects, while it violates only one system invariant: (S23). (We assume, of course, that *MOVE* is only used when its precondition is satisfied.) This invariant is concerned with the representation of references in the store. It states that the reference of an object *Y* is represented in the store by a "pointer" to the location of *Y*, where a pointer is an address (= *left(A(Y))*) augmented with (constant) additional information (= *R(ref(Y))*). This is pictured schematically in Figure 5.6.a, where *R(ref(Y)) = 0* for each node *Y*.

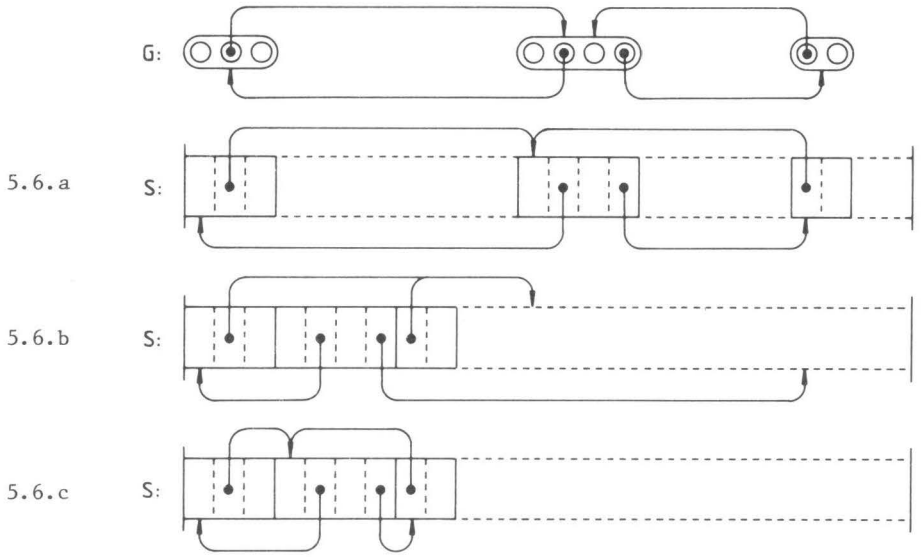


Figure 5.6

If we choose the obvious compaction approach and simply move all nodes as far to the left as possible, the violations of system invariant (S23) are characterized by Figure 5.6.b: Instead of pointing to the new locations of objects (as prescribed by system invariant (S23)), pointers still point to the old locations. The compacter will therefore have to "update" all pointers in the store so as to restore system invariant (S23), resulting in the situation of Figure 5.6.c.

The above description of the work of a compacter gives rise to a number of questions. The first is in which order the nodes should be moved by the compacter, so as not to violate the precondition of *MOVE*. This precondition implies that a node may only be moved to a location which does not overlap with the location of any of the other nodes. The simplest way to achieve this is to move nodes in the order from left to right. So, first the leftmost node is moved as far to the left as possible, then the leftmost but one, etc.. This method has the pleasant property that the order of nodes in the store is preserved. The method is so-called "genetic order preserving" [TERASHIMA & GOTO 78]. This does not only have advantages in a virtual storage environment (where it may prevent "thrashing"), but it also enables the implementer to make use of the fact that nodes have a fixed order in the store. If we assume certain things concerning the size of (the locations of) nodes, or if we assume that nodes are moved to a separate free part of the store (see details E and V in Subsection 5.2.1.5) other ways to move nodes safely are conceivable. In algorithms where we wish to keep the order of moving nodes abstract, we shall use the term "moving order" to denote an arbitrary safe order of moving nodes.

The work of a compacter can now be described more precisely, though very abstractly, by the following algorithm:

```
Algorithm C

Variables:
   a: integer.
Action:
   a := left(S).
   For each node X ∈ dom(A) in moving order
   | MOVE(X,a).
   | a := a + size(A(X)).
   For each reachable reference object X
   | Let Y = obj(val(X)).
   | cont(A(X)) := R(val(X)) + left(A(Y)).
```

In the first for-loop nodes are moved and in the second, pointers are updated, i.e., system invariant (S23) is restored.

5.2.1.3. Bookkeeping

Algorithm C may give the impression that compaction is a trivial affair. The reason why it is not is that in a concrete implementation the compacter must operate exclusively on the store S. In order to apply the above algorithm the abstract components G, R and A of the storage management system H must be "eliminated" from the algorithm. This elimination, though highly implementation dependent, is rather obvious (as it was for the garbage collection algorithms discussed) except for the part of the algorithm where pointers are updated. In that part the location $A(Y)$ of an object Y must be determined, which is obtained by dereferencing the reference object X. In terms of operations on the store this amounts to reading the contents P of $A(X)$ and using P to determine the location of Y. Since Y has been moved, by "location of Y" we mean the new location of Y. P, however, is a pointer to the old location of Y. This raises the problem of how to determine the new location of an object from a pointer to its old location. The information necessary to solve this problem must be built up by the compacter itself and will be called the "updating information". The process of building-up this information will be referred to as "bookkeeping".

As can be inferred from the above, the work to be done by a compacter splits up into three "phases":

(1) Bookkeeping.
    In this phase the updating information is built up.
(2) Moving.
    All nodes are moved to their new locations.
(3) Updating.
    Using the updating information all pointers are updated.

This is only a "conceptual decomposition". In practice these phases can be merged in many ways, as we shall see.

5.2.1.4. Some remarks

There is a minor problem, which has to do with the location of the root. All objects are accessed through access paths emanating from the root. This implies that after a compaction the concrete machine must know where to find the location of the root. The simplest way to achieve this is to give the root a fixed location in the leftmost part of the store, as in

Figure 5.6. None of the compaction algorithms which will be discussed in the sequel will make any assumptions as to the location of the root, however, though all of the algorithms keep the root in the leftmost part of the store if it is already there (which is a consequence of the precondition of *MOVE*). If the root is not in the leftmost part of the store, the algorithms will generally move the root. The compacter should then let the concrete machine know where the location of the root is. How, is left open here.

The main overhead of a compacter is caused by the updating information. Apart from that, there is also information involved in a compacter which is comparable to the type and status information in a garbage collector. For the required type information, the compacter can usually rely entirely on the type information for the garbage collector, which is already there. This also applies to the status information, except that the space requirements for this kind of information are often much less than in the garbage collector. Space reserved for the status information of the garbage collector can then be used for other purposes (such as storing updating information). The compaction algorithms will therefore be classified according to the way the updating information is represented. This leads to a classification of compaction algorithms in two classes which will be discussed in Subsections 5.2.2 and 5.2.3.

The number of compaction algorithms which will be discussed is considerably less than the number of garbage collection algorithms discussed in Section 5.1. The reason for this is not that there is less literature on compaction algorithms than on garbage collection algorithms. Quite the contrary, there is more literature on the former than on the latter. The reason is that in compaction algorithms, even more than in garbage collection algorithms, there is the opportunity to use low level implementation tricks ("pointer juggling"). If however, we remove the low level implementation "sauce" from the algorithms, only a few really different algorithms remain. These are the algorithms which will be discussed. The removal of the implementation sauce has the pleasant side effect of making the algorithms more digestible. The implementation tricks will be discussed separately with the algorithms in which they can be used. Each implementer will be able to make the translation from abstract to concrete algorithm using the trick described. In some cases, the concrete algorithm will also be described.

5.2.1.5. Compaction details

The details which will be used to classify the different compaction problems and algorithms are given below. They will be explained the first time they are used.

Compaction problem details

V: For each node $X \in dom(A)$
$\quad | \; left(S) + \Sigma\, Y \in G \cap dom(A)\ [size(A(Y))] \leq left(A(X))$.

E: There is an integer N such that
for each node $X \in dom(A)$
$\quad | \; size(A(X)) = N,$
$\quad | \; size(S) \equiv 0 \ (\underline{\text{mod}}\ N),$
$\quad | \; left(A(X)) - left(S) \equiv 0 \ (\underline{\text{mod}}\ N).$

L: If V is a reference value,
then $R(V) = 0$.

D: If X is a reference object, $X \in dom(A)$,
then X is reachable.

N: If X is a reachable reference object,
then $obj(val(X))$ is a node.

S: If X is a reachable reference object,
$Y = obj(val(X))$,
then $left(A(Y)) \leq left(A(X))$.

H: Each node $X \in dom(A)$ has a component, denoted by $head(X)$, which is a scalar object.

R: If X is a node, $X \in dom(A)$,
then $cell(left(A(X)))$ is occupied by a branch of X.

Compaction algorithm details

F: The updating information is represented by a relocation map.

B: The updating information is represented by branch sets.

G: The order of nodes in the store is preserved.

M: The bookkeeping phase is combined with the moving phase.

U: The bookkeeping phase is combined with the updating phase.

P: The moving phase is combined with the updating phase.

5.2.2. Compaction algorithms using a relocation map

In the first class of compaction algorithms to be discussed the updating information is represented by a mapping F, which maps the address of a cell in the old location of a node to the address of the corresponding cell in the new location of the node, as indicated in Figure 5.7.

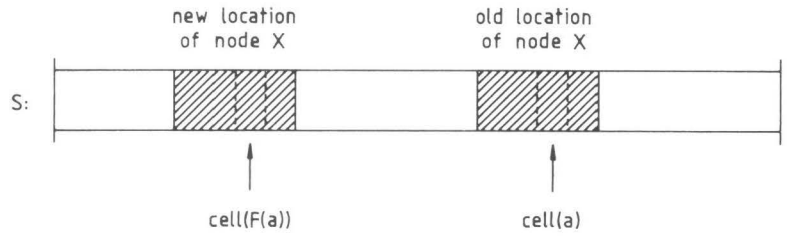


Figure 5.7

The three phases of a compacter which uses such a "relocation map" are described by the following algorithm:

```
Algorithm C.F

Variables:
   F: mapping from integers to integers,
   a: integer.
Action:
   Bookkeeping:
      F,a := ∅,left(S).
      For each node X ∈ dom(A) in moving order
      | Let b = left(A(X)).
      | Let s = b - a.
      | For each cell C ∈ A(X)
      | | F(addr(C)) := addr(C) - s.
      | a := a + size(A(X)).
   Moving:
      a := left(S).
      For each node X ∈ dom(A) in moving order
      | MOVE(X,a).
      | a := a + size(A(X)).
   Updating:
      For each reachable reference object X
      | Let b = cont(A(X)) - R(val(X)).
      | cont(A(X)) := R(val(X)) + F(b).
```

This algorithm will be the basis for all other algorithms to be discussed in this subsection. Its implementation raises a series of questions. In the bookkeeping and moving phase all nodes in the store are "visited" in some unspecified safe order (which, of course, is assumed to be the same in both phases). A first question is how the compacter is able to "find" the nodes in the store. There are many (obvious) solutions to this problem, of which we mention only two. First, the garbage collector can build a linked list of the nodes in the store in moving order (usually from left to right). This need not cost extra space if the free space between the nodes in the store or the vacant space for the status information of the garbage collector is used to build this list. Secondly, the marking information left behind by the garbage collector can be used to find the nodes in the store.

In the updating phase all reachable reference objects are visited. If nodes do not contain any type information and also the type information is not known statically, the reachable reference objects must in principle be visited in the same way as in the garbage collector (i.e., by keeping track of status and type information). In all other cases it is possible to visit nodes one by one (as in the other phases) and upon visiting a node, visit all of its reachable branches. This implies that it must be possible to tell whether an arbitrary branch of a node is reachable or not, which can (possibly) be done by using the marking information of the garbage collector. Check that this is not necessary if there are no dangling pointers in the store (as with problem detail D). The updating phase can then be implemented as follows:

```
   Updating:
      For each node X ∈ dom(A)
      | For each Y ∈ branches(X)
      | | Let b = cont(A(Y)) - R(val(Y)).
      | | cont(A(Y)) := R(val(Y)) + F(b).
```

It will not be difficult to check now that, in principle, it is possible to implement Algorithm C.F in such a way that it takes $O(n)$ time, where n is the total number of cells occupied by nodes in the store. The space requirements for such an implementation would also be $O(n)$: At least n cells are required to store F, which is absolutely prohibitive in most implementations. We shall now discuss a number of ways to reduce the space overhead caused by F. As we have already seen with garbage collection algorithms, the reduction of the space overhead will go at the expense of either the speed or the generality of the algorithm.

If the old locations of nodes do not overlap with their new locations (problem detail V) it is possible to store F in the old locations of nodes, thus reducing the space overhead to zero. Typically, this situation occurs when compacting in a virtual storage environment from one "semispace" [FENICHEL & YOCHELSON 69] to another. In order to implement F this way, the bookkeeping and moving phase must be combined, as described in:

```
Algorithm CV.FM

Variables:
   a: integer.
Action:
   a := left(S).
   For each node X ∈ dom(A)
   | Let A = A(X).
   | Let b = left(A).
   | Let s = b - a.
   | MOVE(X,a).
   | For each cell C ∈ A
   | | cont(C) := addr(C) - s.
   | a := a + size(A).
   For each reachable reference object X
   | Let b = cont(A(X)) - R(val(X)).
   | cont(A(X)) := R(val(X)) + cont(cell(b)).
```

Notice that the requirement that nodes must be visited "in moving order" has been omitted: Any order will do. The implementation trick for F used in this algorithm is essentially the same as used in all so-called "list-moving algorithms" [FENICHEL & YOCHELSON 69], [HANSEN 69], [CHENEY 70], [REINGOLD 73], [CLARK 76] (see also Section 5.3), the only difference being that the latter algorithms are concerned with the "LISP case" instead of the general case of references to arbitrary subobjects of nodes. Note that in the former case (where problem detail N applies) the statement:

```
For each cell C ∈ A
| cont(C) := addr(C) - s.
```

can be replaced by:

```
cont(cell(b)) := a.
```

Another oppurtunity to implement F without any space or time overhead arises when all nodes occupy locations of the same size N at boundaries of the kind $(left(S) + k * N)$, where $k \geq 0$ (problem detail E). The idea is that, if n is the number of nodes in the store, only those nodes need be moved which occupy a location with left address $\geq b$, where $b = left(S) + n * N$: They fit exactly in the "holes" (of size N) with left address $< b$. The

locations abandoned by the moved nodes are never overwritten and can be used to implement F according to the following implementation invariant:

$$\forall\, c \in dom(F)\ [F(c) = \underline{\text{if}}\ c < b\ \underline{\text{then}}\ c\ \underline{\text{else}}\ cont(cell(c))\ \underline{\text{fi}}].$$

If, furthermore, all references refer to nodes (detail N), if there are no unreachable reference objects (detail D) and if $R(V) = 0$ for all references V (detail L) then the following algorithm can be derived:

```
Algorithm CENDL.FM

Variables:
   a,b: integer.
Action:
   a,b := left(S),right(S) + 1.
   While a < b
   | While OCCUPIED(a)
   | | a := a + N.
   | While FREE(b)
   | | b := b - N.
   | If a < b
   | | Let X = NODE(b).
   | | MOVE(X,a).
   | | cont(cell(b)) := a.
   a,b := left(S),a.
   While a < b
   | Let X = NODE(a).
   | For each Y ∈ branches(X)
   | | Let c = cont(A(Y)).
   | | If c ≥ b
   | | | cont(A(Y)) := cont(cell(c)).
   | a := a + N.

OCCUPIED(a):
   a ∈ {addr(C) | C ∈ ∪ X ∈ dom(A) [A(X)]}.

FREE(a):
   a ∉ {addr(C) | C ∈ ∪ X ∈ dom(A) [A(X)]}.

NODE(a):
   The node X ∈ dom(A) such that left(A(X)) = a.
```

This algorithm has been described in [HART & EVANS 64], [BOBROW 68]. When used in a virtual storage environment the algorithm has the disadvantage that it affects the order of nodes in the store in a rather wild way.

If neither problem detail V or E applies, implementation of F without space or time overhead becomes more difficult. In the case of references referring only to nodes, a reasonably space efficient implementation of F is possible, provided that nodes are not moved before all pointers to them have been updated, such as in the following algorithm:

```
Algorithm CNDL.FG

Variables:
   F: mapping from integers to integers,
   a: integer.
Action:
   F,a := ∅,left(S).
   For each node X ∈ dom(A) from left to right
   | Let b = left(A(X)).
   | F(b) := a.
   | a := a + size(A(X)).
   For each node X ∈ dom(A) from left to right
   | For each Y ∈ branches(X)
   | | Let b = cont(A(Y)).
   | | cont(A(Y)) := F(b).
   a := left(S).
   For each node X ∈ dom(A) from left to right
   | MOVE(X,a).
   | a := a + size(A(X)).
```

Due to the fact that nodes are not moved before all pointers to them have been updated, F can be implemented by reserving a cell in the location of each node X, in which the new value of $left(A(X))$ is stored. If, for instance, the leftmost cell of the location of a node is chosen for this purpose, "$F(b)$" can systematically be replaced by "$cont(cell(b))$" in the above algorithm. The algorithm obtained this way is the compaction algorithm used in the LISP 2 garbage collector (see [KNUTH 68], p. 602), where it is combined with garbage collection algorithm GNK.DTEH*. The space reserved in each node for the implementation of the mapping ℓ in Algorithm GNK.DTEH* can then be re-used for the implementation of F. Anyway, the space overhead caused by F in Algorithm C.F can be reduced to a nodewise overhead in Algorithm CNDL.FG without deteriorating the speed of the algorithm. Check that it is possible to divide the updating phase over both the bookkeeping and moving phase in Algorithm CNDL.FG, thus reducing the number of "scans" of the store to two instead of three. (This need not necessarily improve the speed of the algorithm, because pointers must now be inspected twice instead of once, in order to see whether they point to the right or to the left.)

The implementation methods for F discussed so far all amount to implementing F as an array, viz., the store itself. The algorithms based on these implementations operate in $O(n)$ time, where n is the total number of cells occupied by nodes, but they either require extra space or are applicable in special cases only. We shall show now that, even in the most general case, it is possible to implement F without any (significant) space overhead. The price to be paid for this is an $O(n \log n)$ compaction time.

Prior to a compaction the part of the store occupied by nodes can be viewed as a collection of compact locations with one or more free cells between them. These locations will be called "blocks" (see Figure 5.8). It is not difficult to see that if nodes are moved in the order from left to right in Algorithm C.F, then (after the bookkeeping phase) the value $(addr(C) - F(addr(C)))$ for each cell C in a given block B will be the same: It is the number of cells that B will be moved to the left. This number will be called the "shift" of B (see Figure 5.8). By recording the shift of B in $F(left(B))$ instead of recording the new address of each cell $C \in B$ in $F(addr(C))$ the space requirements for F can be reduced considerably as described in Algorithm C.FG below.

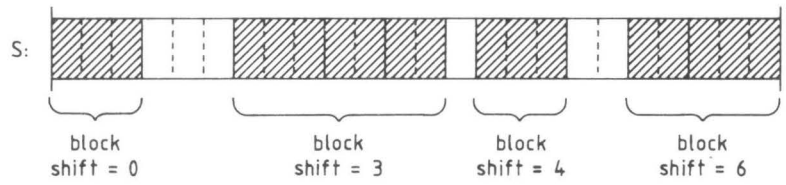


Figure 5.8

```
Algorithm C.FG

Variables:
   F: mapping from integers to integers,
   a,t: integer.
Action:
   Bookkeeping:
      F,a,t := ∅,left(S),-1.
      For each node X ∈ dom(A) from left to right
      | Let b = left(A(X)).
      | Let s = b - a.
      | If s > t
      | | F(b),t := s,s.
      | a := a + size(A(X)).
   Moving:
      a := left(S).
      For each node X ∈ dom(A) from left to right
      | MOVE(X,a).
      | a := a + size(A(X)).
   Updating:
      For each reachable reference object X
      | Let b = cont(A(X)) - R(val(X)).
      | Let c = max{d ∈ dom(F) | d ≤ b}.
      | Let s = F(c).
      | cont(A(X)) := cont(A(X)) - s.
```

The number of entries in F in this algorithm is equal to the number m of blocks in the store. Since each block, except possibly the leftmost, has at least one free cell immediately at its left, the free storage contains at least $m-1$ cells. This implies that if each entry of F can be encoded in a single cell, the implementation of F need not cause any space overhead (apart from a small constant space): F can be encoded (almost) entirely in the free storage.

Though the implementation of F need not cause any space overhead, the process of updating a pointer has become more complex in Algorithm C.FG. In order to find the shift of a block into which a (naked) pointer b points, the domain of F must be searched for the maximal element d with $d \leq b$. The efficiency of this searching operation depends entirely on the implementation of F. A number of these implementations will now be discussed.

A first way to implement F [WEGBREIT 72] is to perform the updating phase before the moving phase and record $F(c)$ in $cell(c-1)$ for each

$c \in dom(F)$. This is possible because each $c \in dom(F)$ is the left address of a block, so $cell(c-1)$ belongs to the free storage (where we assume for convenience's sake that $cell(left(S))$ is also free). Determining the maximal value $d \in dom(F)$ such that $d \leq b$ for a given pointer b then amounts to searching the first free cell at the left of $cell(b)$. This searching process makes compaction essentially an $O(n^2)$ process though a considerable speed-up can be obtained by using a small amount of extra storage [WEGBREIT 72]. If there is sufficient room in a cell, or if the holes between blocks are always sufficiently large, the cells containing the values of F can be arranged in a binary tree, reducing compaction time to $O(n \log n)$. An efficient method to do so has been described in [TERASHIMA & GOTO 78]. When using this kind of implementation of F, the combination of phases will generally be awkward, though not impossible.

A second way to implement F [HADDON & WAITE 67] is to represent F as a table, which contains pairs of the kind $(c,F(c))$ $(c \in dom(F))$. If these pairs fit in a single cell and if the bookkeeping and moving phase are combined, the table can be constructed in the hole between the blocks which have been moved and those which have not: If m blocks have been moved, this hole will contain at least $m-1$ cells, while F contains m entries. Since this hole moves to the right during the combined bookkeeping and moving phase, the table "rolls" through the store. Though, in principle, this rolling is a linear process [HADDON & WAITE 67] it shuffles the entries in the table. Sorting is therefore necessary to restore the order of the entries, so as to enable binary searching. Both the sorting and the binary searching imply that compaction using this kind of implementation of F is an $O(n \log n)$ process. Methods to reduce the degree of shuffling of the table and to speed up searching, using the extra free storage outside the table, are described in [WAITE 73], [FITCH & NORMAN 78].

5.2.3. Compaction algorithms using branch sets

The second class of compaction algorithms to be discussed differs entirely from the first. Instead of using a relocation map, the updating information is represented by associating a set $B(X)$ of reference objects with each node X. This set contains all reference objects with a value referring to a subobject of X, as indicated in Figure 5.9.

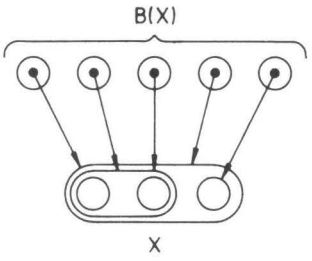


Figure 5.9

The three phases of a compacter which uses these "branch sets" are described by the following algorithm, in which the moving phase is preceded by the updating phase for reasons to become clear later:

```
Algorithm C.B

Variables:
   B: mapping from nodes to sets of reference objects,
   a: integer.
Action:
   Bookkeeping:
      B := {(X,∅) | X ∈ G ∩ dom(A)}.
      For each reachable reference object X
      | Let Y = node(obj(val(X))).
      | B(Y) := B(Y) ∪ {X}.
   Updating:
      a := left(S).
      For each node X ∈ dom(A) in moving order
      | Let b = left(A(X)).
      | Let s = b - a.
      | While B(X) ≠ ∅
      | | Get Y from B(X).
      | | cont(A(Y)) := cont(A(Y)) - s.
      | a := a + size(A(X)).
   Moving:
      a := left(S).
      For each node X ∈ dom(A) in moving order
      | MOVE(X,a).
      | a := a + size(A(X)).
```

In contrast to Algorithm C.F the reachable reference objects are now visited in the bookkeeping phase. The same remarks made about the way of visiting these reference objects in Subsection 5.2.2 apply here. Notice that in the updating phase nodes could just as well be visited in reverse moving order, provided we use the variable a in the reverse way as well. (This remark, by the way, also applies to the bookkeeping phase of Algorithm C.F.)

At first sight the implementation of Algorithm C.B may seem to require a considerable space overhead. First, from a reference value V it must be possible to determine the node containing $obj(V)$. This will generally require a space overhead per reference object. Secondly, a straightforward implementation of the branch sets (for example, as linked lists) will also require a considerable space overhead per reference object. All this overhead will usually be unacceptable, unless the density of reference objects is small. We shall show now, however, that it is often possible to eliminate the overhead almost entirely. For that purpose we shall assume, for the time being, that references refer to nodes only (thus eliminating the first kind of space overhead). Algorithm C.B can then be written as follows:

```
Algorithm CN.B

Variables:
   B: mapping from nodes to sets of reference objects,
   a: integer.
Action:
   Bookkeeping:
      B := {(X,∅) | X ∈ G ∩ dom(A)}.
      For each reachable reference object X
      | Let Y = obj(val(X)).
      | B(Y) := B(Y) ∪ {X}.
   Updating:
      a := left(S).
      For each node X ∈ dom(A) in moving order
      | Let c = R(ref(X)) + a.
      | While B(X) ≠ ∅
      | | Get Y from B(X).
      | | cont(A(Y)) := c.
      | a := a + size(A(X)).
   Moving:
      a := left(S).
      For each node X ∈ dom(A) in moving order
      | MOVE(X,a).
      | a := a + size(A(X)).
```

In order to show how the branch sets can be implemented in a space-efficient way, let us for each node $X \in dom(A)$ choose an arbitrary atom of X, which will be denoted by $head(X)$. Immediately after the bookkeeping phase in Algorithm CN.B the situation for a node X and its associated branch set $B(X)$ can be pictured as in Figure 5.10.a.

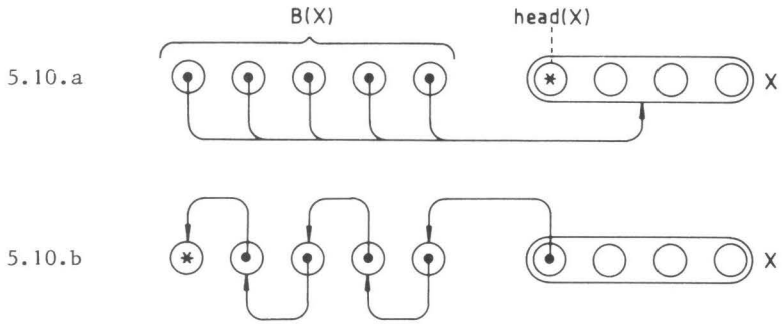


Figure 5.10

The situation can be transformed in such a way that the reference objects in $B(X)$ constitute a linked list, where the new value of $head(X)$ refers to the head of the list and the old value of $head(X)$ acts as a list terminator (see Figure 5.10.b). This transformation can be applied to all nodes X simultaneously and without loss of information if and only if $head(X)$ is not itself a reference object, i.e., if $head(X)$ is a scalar

object. Let us for the time being assume that this condition is satisfied or, in other words, that problem detail H applies. This assumption, by the way, is not unrealistic. It is often necessary to endow a node with information concerning its type or size. An extra atom introduced in a node X for this purpose is a good candidate for $head(X)$.

Figure 5.10 now sketches an implementation of the branch sets without any space overhead. This trick, which is somewhat similar to the Deutsch-Schorr-Waite trick discussed in Subsection 5.1.5, is in fact age-old: It is essentially the same trick as used in one-pass assemblers to handle forward references [WILKES et al. 57]. The literature on the use of this trick in compaction algorithms is of a rather recent date [FISHER 74], [THORELLI 76], [DEWAR & McCANN 77], [HANSON 77], [MORRIS 78], [JONKERS 79]. One thing which one should bear in mind when using this trick is the fact that in a real implementation the linked lists do not consist of reference objects (which are abstract things), but of the cells occupied by these reference objects. This implies that a node X may never be moved if one of its branches is still contained in the $B(Z)$ of some node Z (with possibly $X = Z$):

Additional precondition for $MOVE(X,a)$

$$branches(X) \cap \bigcup Z \in G \cap dom(A)\ [B(Z)] = \emptyset.$$

It is easy to see that Algorithm CN.B satisfies this requirement (but only because we made the updating phase precede the moving phase). Thus, by rewriting Algorithm CN.B using the implementation trick for the branch sets a three-phase compacter can be obtained. For reasons of efficiency it is useful to combine phases, however. The following algorithm combines the updating phase with both the bookkeeping phase and the moving phase and does not violate the additional precondition for $MOVE$:

```
Algorithm CND.BGUP

Variables:
   B: mapping from nodes to sets of reference objects,
   a: integer.
Action:
   B,a := {(X,∅) | X ∈ G ∩ dom(A)},left(S).
   For each node X ∈ dom(A) from left to right
   | UPDATE(X,a).
   | For each Y ∈ branches(X)
   | | Let Z = obj(val(Y)).
   | | B(Z) := B(Z) ∪ {Y}.
   | a := a + size(A(X)).
   a := left(S).
   For each node X ∈ dom(A) from left to right
   | UPDATE(X,a).
   | MOVE(X,a).
   | a := a + size(A(X)).

UPDATE(X,a):
   Let c = R(ref(X)) + a.
   While B(X) ≠ ∅
   | Get Y from B(X).
   | cont(A(Y)) := c.
```

This is the abstract version of the algorithm described in [JONKERS 79].

Just like we showed for the Deutsch-Schorr-Waite algorithm, a more concrete version of this algorithm can be obtained from the abstract version through a simple substitution process. For that purpose let us assume that problem detail H applies. Using Figure 5.10 for guidance the operations on the abstract variable B can be translated as follows:

$B := \{(X,\emptyset) \mid X \in G \cap dom(A)\} \longrightarrow$
    Skip

$B(Z) := B(Z) \cup \{Y\} \longrightarrow$
    Let $H = head(Z)$.
    $val(Y),val(H) := val(H),ref(Y)$

While $B(X) \neq \emptyset \longrightarrow$
    Let $H = head(X)$.
    While $val(H)$ is a reference value

Get Y from $B(X) \longrightarrow$
    Let $Y = obj(val(H))$.
    $val(Y),val(H) := ref(X),val(Y)$

Thus the following algorithm can be derived from Algorithm CND.BGUP:

Algorithm CNDH.BGUP*

Variables:
  a: integer.
Action:
  $a := left(S)$.
  For each node $X \in dom(A)$ from left to right
  | $UPDATE(X,a)$.
  | For each $Y \in branches(X)$
  | | Let $Z = obj(val(Y))$.
  | | Let $H = head(Z)$.
  | | $val(Y),val(H) := val(H),ref(Y)$.
  | $a := a + size(A(X))$.
  $a := left(S)$.
  For each node $X \in dom(A)$ from left to right
  | $UPDATE(X,a)$.
  | $MOVE(X,a)$.
  | $a := a + size(A(X))$.

$UPDATE(X,a)$:
  Let $c = R(ref(X)) + a$.
  Let $H = head(X)$.
  While $val(H)$ is a reference value
  | Let $Y = obj(val(H))$.
  | $val(Y),val(H) := ref(X),val(Y)$.
  | $cont(A(Y)) := c$.

Like the Deutsch-Schorr-Waite algorithm, this algorithm makes temporary changes in the graph G, thus affecting a number of system invariants (besides system invariant (S23)).

An even more concrete version of the above algorithm can be obtained by translating the operations on the graph G in terms of operations on the store S. This is a highly implementation dependent affair which will be omitted. One thing should be noted, though. In Algorithm CNDH.BGUP* the

fact that it can be established whether the value of *head(X)* is a scalar or a reference value, is used. In low level terms this implies that it must be possible to distinguish the representation of the original value of *head(X)* (i.e., *cont(A(head(X)))*) from the representation of a reference value (i.e., a pointer). If this should not be possible, it can be achieved artificially by using an extra mark bit in each pointer, if available.

Algorithm CNDH.BGUP* is based on the assumption that each node *X* has an atom *head(X)* which is a scalar object. Even if we drop this assumption and choose an arbitrary atom of *X* for *head(X)*, the described implementation trick for the branch sets can be applied. An additional problem is then to take care that not at the same time $B(X) \neq \emptyset$ and $head(X) \in B(Z)$ for some *Z*. In order to achieve this it is sufficient to keep the following assertion invariant:

Additional invariant for nodes $X \in dom(A)$

$$B(X) \neq \emptyset \Rightarrow branches(X) \cap \bigcup Z \in G \cap dom(A)\ [B(Z)] = \emptyset.$$

Notice that this invariant can only be satisfied if phases are combined, as in the following algorithm, which also satisfies the additional precondition for *MOVE*:

```
Algorithm CND.BGMUP

Variables:
   B: mapping from nodes to sets of reference objects,
   a: integer.
Action:
   Let s = Σ X ∈ G ∩ dom(A) [size(A(X))].
   B,a := {(X,∅) | X ∈ G ∩ dom(A)},left(S) + s.
   For each node X ∈ dom(A) from right to left
   | a := a - size(A(X)).
   | UPDATE(X,a).
   | For each Y ∈ branches(X)
   | | Let Z = obj(val(Y)).
   | | If left(A(Z)) < left(A(X))
   | | | B(Z) := B(Z) ∪ {Y}.
   | | else
   | | | If Z = X
   | | | | cont(A(Y)) := R(ref(X)) + a.
   a := left(S).
   For each node X ∈ dom(A) from left to right
   | UPDATE(X,a).
   | MOVE(X,a).
   | For each Y ∈ branches(X)
   | | Let Z = obj(val(Y)).
   | | If left(A(Z)) > left(A(X))
   | | | B(Z) := B(Z) ∪ {Y}.
   | a := a + size(A(X)).

UPDATE(X,a):
   Let c = R(ref(X)) + a.
   While B(X) ≠ ∅
   | Get Y from B(X).
   | cont(A(Y)) := c.
```

This ingenious algorithm is an abstract version of the algorithm described

in [MORRIS 78]. Notice that in the first phase of the algorithm the nodes in the store are visited in the order from right to left instead of left to right. Notice also that if all pointers point to the left (problem detail S), large parts of the algorithm can be skipped. The algorithm then corresponds to the algorithm described in [FISHER 74].

The operations on the abstract variable B can again be translated in terms of the implementation sketched in Figure 5.10. The only difference now is the translation of the test "$B(X) \neq \emptyset$", which amounts to:

> *val(head(X))* is a reference value and
> *obj(val(head(X)))* is not a node.

(Note that an object can never be both a reference object and a node, since nodes are structured objects.) In practice (i.e., in the store) this test usually cannot be implemented without some overhead. It can always be implemented at the expense of an extra bit per pointer. The actual translation of Algorithm CND.BGMUP is left to the reader.

It is not difficult to see that, using the implementation trick for the branch sets, Algorithms CN.B, CND.BGUP and CND.BGMUP can be implemented in such a way that they operate in a time $O(n)$, where n is the total number of cells occupied by nodes, and require a space overhead of at most one bit per pointer. Furthermore, Algorithms CND.BGUP and CND.BGMUP demonstrate that the entire compaction process can be performed in two phases. This raises the question which of these two algorithms should be preferred.

Though both algorithms operate in $O(n)$ time, Algorithm CND.BGMUP will generally be slower than Algorithm CND.BGUP for the following reasons:

(1) Reference objects are visited twice instead of once.
(2) Extra tests are performed to determine the direction a pointer is pointing in.
(3) Nodes are visited in different orders in the two phases.
(4) The total number of cells occupied by nodes must be determined beforehand.

The space overhead of both algorithms is the same. A point in favour of Algorithm CND.BGMUP is that it is more generally applicable, because for its implementation we do not have to rely on problem detail H (see also below). The conclusion therefore is that, unless problem detail H does not apply, Algorithm CND.BGUP should be preferred to Algorithm CND.BGMUP.

There is a realistic situation where we can even do better than Algorithm CND.BGUP. Suppose detail H applies and we also have an unused cell in the location of each node, for example because this cell has been used by the garbage collector to store status information. We can use this cell to store a relocation map, thus enabling a combination of the branch set and relocation map compaction techniques. This idea is exploited in the following algorithm, where the relocation map is used to update pointers to the left and branch sets are used to update pointers to the right:

```
Algorithm CND.BFGU

Variables:
   B: mapping from nodes to sets of reference objects,
   F: mapping from nodes to integers,
   a: integer.
Action:
   B,F,a := {(X,∅) | X ∈ G ∩ dom(A)},∅,left(S).
   For each node X ∈ dom(A) from left to right
   | Let c = R(ref(X)) + a.
   | While B(X) ≠ ∅
   | | Get Y from B(X).
   | | cont(A(Y)) := c.
   | F(X) := c.
   | For each Y ∈ branches(X)
   | | Let Z = obj(val(Y)).
   | | If left(A(Z)) ≤ left(A(X))
   | | | cont(A(Y)) := F(Z).
   | | else
   | | | B(Z) := B(Z) ∪ {Y}.
   | a := a + size(A(X)).
   a := left(S).
   For each node X ∈ dom(A) from left to right
   | MOVE(X,a).
   | a := a + size(A(X)).
```

Notice that the relocation map F has been defined in a slightly more abstract way than in Subsection 5.2.2. This algorithm (which was only discovered when writing Chapter 7) has the advantage over Algorithm CND.BGUP that it has a separate moving phase which can be optimized extremely (using "block moves"; see also Chapter 7). The extra tests to determine the direction of pointers in Algorithm CND.BFGU pay off in the immediate updating of pointers to the left. The only disadvantage of Algorithm CND.BFGU as compared with Algorithm CND.BGUP is that the former requires a space overhead and therefore is less generally applicable than the latter.

So far we have restricted ourselves to the case of references referring to nodes only (problem detail N). Let us now consider the general case of references to arbitrary subobjects of nodes (Algorithm C.B). The implementation of the branch sets and the implementation of the expression "*node(obj(val(X)))*" will then, in general, require an additional space overhead. Yet, there are situations where this space overhead may be eliminated to a great extent. Suppose, for reasons other than compaction, it must be possible, given a reference value *V*, to determine *node(obj(V))*. A frequently used way to achieve this is to represent references in an "enriched" way. For example, a reference to a subobject *Y* of a node *X* may be represented by an address-offset pair:

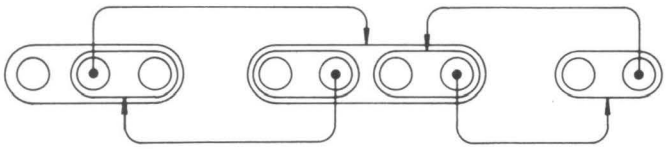
$$(left(A(X)), left(A(Y)) - left(A(X))).$$

The first element of this pair can be employed to implement the branch sets through the trick described above. The compacter, in fact, can treat all references as if they refer to nodes by considering only the address part of pointers. Provided that pointers are enriched it is easy to see, therefore, that Algorithms CND.BGUP, CND.BGMUP and CND.BFGU can be generalized to the case of references to arbitrary subobjects of nodes

without causing any significant time or space overhead.

Algorithm CND.BGMUP can be adjusted in such a way that it is applicable in the general case without any additional time or space overhead, even if pointers are not enriched (which is a second point in favour of Algorithm CND.BGMUP as compared with Algorithm CND.BGUP). This can be explained as follows. Consider the state of the graph as pictured in Figure 5.11.a, where the order from left to right corresponds to the order according to which atomic objects are located in the store.

5.11.a



5.11.b

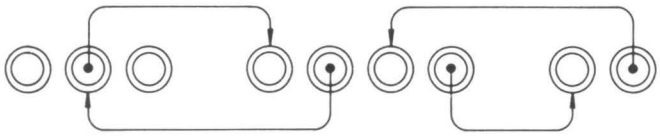


Figure 5.11

If we replace each reference value *V* in Figure 5.11.a by a reference value referring to the leftmost atom of *obj(V)*, if we remove all objects except atomic objects and if we replace each atomic object *X* by a node containing *X* as its single component, then the situation of Figure 5.11.b is obtained. We can perform a compaction now as if the graph had the shape of Figure 5.11.b. It is not difficult to see that this compaction will (under certain conditions) have the same effect on the store as when the graph had the shape of Figure 5.11.a. Since references refer to nodes only in Figure 5.11.b, Algorithm CND.BGMUP can be used (but not Algorithm CND.BGUP, let alone Algorithm CND.BFGU). This unstructured application of Algorithm CND.BGMUP cannot be expressed well in terms of the graph of Figure 5.11.a. It can in fact only be expressed in terms of operations on the store, which is done in [MORRIS 78]. In using this low level version of the algorithm one should be very careful to make sure that it has the desired high level effect.

5.3. COMPACTING GARBAGE COLLECTION

As can be inferred from the definition of the operation *COLLECT GARBAGE & COMPACT*, the effect of a compacting garbage collection should be equal to the effect of a garbage collection followed by a compaction. The simplest way to construct a compacting garbage collector is therefore to independently construct a garbage collector and a compacter, and join the two together. The overhead, both in time and space, required for a compacting garbage collector can often be reduced considerably,

however, if the garbage collector and the compacter are tuned to each other and their actions are combined. A precondition for such a combination is that we know beforehand that a compaction must be performed, and do not let this depend on the outcome of the garbage collection (such as in a "conditionally compacting garbage collector").

In this section we shall discuss through a number of examples how garbage collectors and compacters can be efficiently combined. Since the garbage collection and compaction algorithms used in these examples have already been discussed (at least in their abstract form), only short comments on the examples will be given, including references to literature. The algorithms will be denoted by the label "G&C" followed by a number of problem details (so no algorithm details are specified). The problem details are chosen from the garbage collection and compaction problem details (see Subsections 5.1.1.7 and 5.2.1.5).

A general remark which can be made is that compaction algorithms using a relocation map are not very suitable to be merged with garbage collection algorithms. The reason is (see Algorithm C.F) that in the bookkeeping phase of these compaction algorithms the nodes in the store must be visited in moving order. This moving order can usually only be determined after the garbage collection has been completed. There is one exception. If the old locations of nodes cannot be overwritten by the new locations (problem detail V) any order will do as a moving order. In particular, if we assume that references refer to nodes only (problem detail N), the order according to which nodes are visited by the garbage collector can be chosen. Upon tracing a node for the first time, the garbage collector can immediately move the node to its new location and update the pointer through which the node was traced. Upon tracing the node after the first time, only the pointer in question need be updated. Thus garbage collection and compaction can be performed in a single phase.

The compacting garbage collection problem with additional details V and N is also known as the "list moving problem" [REINGOLD 73], [CLARK 76], though the latter problem is usually restricted to the case of LISP-like nodes. The "list copying problem" [LINDSTROM 74], [FISHER 75], [ROBSON 77], [CLARK 78] is the same as the list moving problem except that the contents of the old locations of nodes may not be destroyed. Each list copying algorithm will do as a list moving algorithm, but is unnecessarily complicated for the purpose of moving a list. List copying algorithms will therefore not be discussed. List moving algorithms typically require no additional storage (though old versions [FENICHEL & YOCHELSON 69], [HANSEN 69] did, because they used recursion). The algorithm described below is a generalization of [CHENEY 70] (without the "cdr encoding"). It is essentially a combination of garbage collection Algorithm GN.DTEH and compaction Algorithm CV.FM. The bag T from Algorithm GN.DTEH is implemented as a queue and the set M is implemented by using the fact that the first cell of the location of a node contains the representation of a reference (problem detail R) and that old and new representations of references can easily be distinguished (assuming problem detail L):

```
Algorithm G&CNVRL

Variables:
   a,b: integer.
Action:
   a,b := left(S),left(S).
   Let C = cell(left(A(R))).
   MOVE(R,b).
   cont(C) := b.
   b := b + size(A(R)).
   While a ≠ b
   | Let X = NODE(a).
   | a := a + size(A(X)).
   | For each Y ∈ branches(X)
   | | Let c = cont(A(Y)).
   | | Let D = cell(c).
   | | If cont(D) ≥ b
   | | | Let Z = NODE(c).
   | | | MOVE(Z,b).
   | | | cont(D) := b.
   | | | b := b + size(A(Z)).
   | | cont(A(Y)) := cont(D).
   dom(A) := dom(A) \ {X ∈ dom(A) | left(A(X)) ≥ b}.

NODE(a):
   The node X ∈ dom(A) such that left(A(X)) = a.
```

The list moving algorithms described in [REINGOLD 73], [CLARK 76] are LISP-tuned variations of this algorithm. They use garbage collection Algorithm GN.DTER rather than Algorithm GN.DTEH, where the bag T is implemented in the old locations of nodes as a stack in linked list representation. [CLARK 76] is essentially the same as [REINGOLD 73], except that the use of the stack and the updating of pointers is optimized in the former.

Compaction algorithms using branch sets, in contrast to those using a relocation map, can be combined very well with garbage collection algorithms. In the bookkeeping phase of these compaction algorithms all reachable reference objects are visited, which is also done in a garbage collection algorithm. Therefore these visits can be combined. An example of this is the following algorithm, which is a combination of Algorithm GNK.DTER** and Algorithm CN.B:

```
Algorithm G&CNK

Variables:
   M: set of nodes,
   S: stack of nodes,
   k: mapping from nodes to integers,
   B: mapping from nodes to sets of reference objects,
   a: integer.
Action:
   M,S,k(R) := {R},<R>,0.
   B := {(X,∅) | X ∈ G ∩ dom(A)}.
   While S ≠ <>
   | Let X = TOP(S).
   | If k(X) ≠ degree(X)
   | | k(X) := k(X) + 1.
   | | Let Y = branch(X,k(X)).
   | | Let Z = obj(val(Y)).
   | | If Z ∉ M
   | | | M,S,k(Z) := M ∪ {Z},PUSH(Z,S),0.
   | | B(Z) := B(Z) ∪ {Y}.
   | else
   | | S := POP(S).
   a := left(S).
   For each node X ∈ dom(A) from left to right
   | If X ∉ M
   | | dom(A) := dom(A) \ sub(X).
   | else
   | | Let c = R(ref(X)) + a.
   | | While B(X) ≠ ∅
   | | | Get Y from B(X).
   | | | cont(A(Y)) := c.
   | | a := a + size(A(X)).
   a := left(S).
   For each node X ∈ dom(A) from left to right
   | MOVE(X,a).
   | a := a + size(A(X)).
```

This is the abstract version of the algorithms described in [THORELLI 76], [DEWAR & McCANN 77], [HANSON 77]. The latter algorithms differ only in the way the variables S and k are implemented. The variable M is implemented in all three algorithms using the following invariant of the first while loop:

```
For each node X ∈ dom(A)
| X ∈ M ⇔ B(X) ≠ ∅ ∨ X = R.
```

The variable B is implemented using problem detail H and the trick described in Subsection 5.2.3. Notice that if upon visiting a node X the original value of *head(X)* is required, the entire list emanating from *head(X)* must be traversed.

In [DEWAR & McCANN 77] an explicit (bounded) stack is used for the implementation of both S and k. [THORELLI 76], [HANSON 77] use an extra atom per node for the implementation of S and k. [HANSON 77] uses the extra atom for a linked list representation of S, while [THORELLI 76] uses the extra atom to store k and as a list head for the implementation of B. Both exploit the fact that if $X = TOP(S)$ and $Y = TOP(POP(S))$, then $branch(Y,k(Y))$ is terminating the list emanating from $head(X)$. In [HANSON 77] this is used to recover $k(Y)$ and in [THORELLI 76] this is used to recover Y when popping S. The list traversals necessary for this make the algorithms operate in $O(n^2)$ time in the worst case. The list traversals can be avoided at the expense of a second extra atom per node. This makes it rather surprising that Algorithm GN.DTEH has not been used instead of Algorithm GNK.DTER**. The former is not only faster than the latter, it also requires only a single extra atom per node in combination with Algorithm CN.B to obtain an $O(n)$ three-phase compacting garbage collector (see also Chapter 7).

The two compacting garbage collection algorithms discussed above are based on problem detail N (though Algorithm G&CNVRL can easily be generalized). We shall now present an algorithm which is applicable in the general situation of references to arbitrary subobjects of nodes. The idea behind this algorithm is as follows. At the end of a garbage collection algorithm using reference marking, the marking information M consists of the set of all reachable reference objects, which is exactly the union of all branch sets after the bookkeeping phase of Algorithm C.B. Assuming problem detail L, the branch sets can be recovered from M by using the following property:

$$\text{For each node } X \in dom(A) \mid B(X) = \{Y \in M \mid left(A(X)) \leq cont(A(Y)) \leq right(A(X))\}.$$

Efficient use of this property can only be made if the elements of M are sorted according to increasing contents. This is done in the following algorithm, which can be viewed as a combination of Algorithm G.CTER and Algorithm C.B. The variable M from Algorithm G.CTER is represented in this algorithm by a variable set L of reference objects using the following invariants:

$$M = L \cup T \text{ and } L \cap T = \emptyset.$$

```
Algorithm G&CL

Variables:
   L,T: set of reference objects,
   a,k: integer.
Action:
   L,T := ∅,branches(R).
   While T ≠ ∅
   | Let X ∈ T.
   | L,T := L ∪ {X},T \ {X}.
   | Let Y = obj(val(X)).
   | For each Z ∈ branches(Y)
   | | If Z ∉ L ∪ T
   | | | T := T ∪ {Z}.
   Let Y1,...,Yn be the elements of L, ordered in such a way that
   cont(A(Yi)) ≤ cont(A(Yi+1))   (i = 1,...,n - 1).
   a,k := left(S),1.
   For each node X ∈ dom(A) from left to right
   | Let b = left(A(X)).
   | Let s = b - a.
   | Let ℓ = k.
   | While k ≤ n and cont(A(Yk)) ≤ right(A(X))
   | | cont(A(Yk)) := cont(A(Yk)) - s.
   | | k := k + 1.
   | If ℓ = k and X ≠ R
   | | dom(A) := dom(A) \ sub(X).
   | else
   | | a := a + size(A(X)).
   a := left(S).
   For each node X ∈ dom(A) from left to right
   | MOVE(X,a).
   | a := a + size(A(X)).
```

This is the algorithm described in [ZAVE 73], where the sets L and T are implemented as linked lists. The space overhead per reference object required for the implementation of L and T can be reduced by using the fact that $L \cap T = \emptyset$, while the test "$Z \notin L \cup T$" can be performed cheaply using the same trick as in Algorithm GNK.DTEH* (though in [ZAVE 73] an explicit mark bit is used). The sorting of the elements of L can be done efficiently using the radix list sort [KNUTH 73]. In [ZAVE 73] the first pass of the sort is combined with the operations on L (using a multi-linked-list representation for L).

The use of garbage collection Algorithm G.CTER makes Algorithm G&CL operate in an $O(n^2)$ worst case time, where n is the number of reachable reference objects. An $O(n \log n)$ time overhead is also introduced because of the necessary sorting. Furthermore, a considerable space overhead per reference object is required. The algorithm, on the other hand, is applicable in the case of references to arbitrary subobjects of nodes. Yet, the use of a combination of Algorithms G.CTER and C.B in a way similar to Algorithm G&CNK may deserve consideration instead, requiring no sorting and about the same space overhead. We shall conclude this section with the presentation of another combination of Algorithms G.CTER and C.B. The algorithm to be presented shows that, even in the most general case, it is possible to perform an entire compacting garbage collection in only two phases. Since the algorithm visits reference objects twice instead of once and since it does not have a separate moving phase (which is "impossible"

in a two-phase compacting garbage collector) it need not necessarily be more efficient than a three-phase combination of Algorithms G.CTER and C.B analogous to Algorithm G&CNK, however. The algorithm is presented without comment below.

```
Algorithm G&C
Variables:
   M,T: set of reference objects,
   B: mapping from nodes to sets of reference objects,
   a: integer.
Action:
   M,T,B := branches(R),branches(R),{(X,∅) | X ∈ G ∩ dom(A)}.
   While T ≠ ∅
   | Get X from T.
   | Let Y = obj(val(X)).
   | For each Z ∈ branches(Y)
   | | If Z ∉ M
   | | | M,T := M ∪ {Z},T ∪ {Z}.
   | Let Z = node(Y).
   | If left(A(Z)) ≤ left(A(X))
   | | B(Z) := B(Z) ∪ {X}.
   a := left(S).
   For each node X ∈ dom(A) from left to right
   | If B(X) = ∅ and X ≠ R
   | | dom(A) := dom(A) \ sub(X).
   | else
   | | Let b = left(A(X)).
   | | Let s = b - a.
   | | While B(X) ≠ ∅
   | | | Get Y from B(X).
   | | | cont(A(Y)) := cont(A(Y)) - s.
   | | MOVE(X,a).
   | | a := a + size(A(X)).
   | | For each Y ∈ branches(X)
   | | | If Y ∈ M
   | | | | Let Z = node(obj(val(Y))).
   | | | | If left(A(Z)) > left(A(Y))
   | | | | | B(Z) := B(Z) ∪ {Y}.
```

5.4. CONCLUSION

In this chapter we have made an attempt to systematically order (a substantial part of) the existing knowledge in the field of garbage collection, according to the lines sketched in Chapter 1. The key to the success of this enterprise was the storage management model introduced in the previous chapter. This model has turned out to be sufficiently concrete to enable the definition of the garbage collection and compaction problem, as well as sufficiently abstract to allow for the description of widely differing garbage collection and compaction algorithms in an implementation independent way.

In certain respects the model can be criticized, though. The model includes a number of simplifications, such as: there is only one root, there are no objects with fixed locations, atomic objects occupy exactly one cell, cells contain (unbounded) integers, there are no alignment

restrictions, etc.. The purpose of these simplifications is, of course, to keep the model, and consequently the algorithms, as simple as possible. Though the model could certainly be extended to eliminate a number of the simplifications, it is felt that each implementer is perfectly capable of undoing these simplifications when adjusting an algorithm to his particular implementation.

A criticism which is related to the above is that the algorithms which we have described are too abstract, too far away from "real" garbage collectors. Indeed, almost all algorithms which we have presented are more abstract than their counterparts in literature. The reason is that (helped by the model) we have systematically tried to catch the essence of an algorithm and dispose of irrelevant detail. This increases both the applicability and the legibility of the algorithms. It also increases the distance to an implementation. Still, as with the simplifications in the model, it is felt that each implementer can easily bridge the gap from abstract algorithm to concrete implementation. The abstract algorithm is even believed to make the process of implementing a (compacting) garbage collector easier and more reliable. The abstract algorithm is easy to understand and can act as a handhold in keeping the implementation process in one's mental grip. At the same time it provides the implementer with the necessary freedom to choose an efficient implementation.

The fact that we could describe the algorithms as abstractly as we did, can be attributed for the major part to the language which we have used. This was not a fixed, and consequently inflexible, programming or specification language, but rather a loose algorithmic language which we were free to extend whenever we felt like it. The latter is similar to the way mathematicians introduce new notations whenever they find it useful to do so. As we have tried to argue in the introduction of Chapter 4, the resemblance to mathematics goes further than this. The language used by mathematicians and the language used here for the description of the algorithms and data structures (in particular, the model) are both loose, but precise. The fact that the language is indeed precise we have tried to demonstrate in Chapter 2, where the first steps to a formalization of this language are taken.

CHAPTER 6

DESIGN OF A STORAGE MANAGER

6.0. INTRODUCTION

The usual way to obtain a portable implementation of a programming language L is to construct a compiler C, which translates programs in L into code for an "abstract machine" M [ELSWORTH 79]. The latter is a hypothetical machine, which is designed in such a way that it is easily implementable on a large class E of existing machines. Given the compiler C, the job of an implementer is then, apart from installing C, to implement M on his particular machine M'. If $M' \in E$, this involves only a minor overhead. The price to be paid for portability is thus kept to a minimum.

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

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

This chapter deals with the problem of designing a (standard) storage management system as described above. It will be demonstrated by means of an example, how this problem can be tackled in a systematic way. The example is not artificial, it is taken from the construction of an ALGOL 68 [VAN WIJNGAARDEN et al. 76] compiler which has been under development at

the Mathematical Centre. In this compiler the above approach is pursued. The abstract machine used in this compiler is called the "MIAM" ("Machine Independent Abstract Machine") [MEERTENS 81]. The treatment will be such that no knowledge of either ALGOL 68 or the MIAM is required.

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

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

As mentioned before, the method will be demonstrated by the design of a machine independent storage management system for the abstract machine MIAM, which is used in a machine independent ALGOL 68 implementation. In the next section the model of the MIAM will be presented (after which one can forget about the MIAM completely). Then the storage management problem will be formulated in Section 6.2. The design of an efficient machine independent storage management system will be described in Section 6.3. The conclusion of this chapter is contained in Section 6.4.

6.1. MODEL

6.1.1. Data structures

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

An area is either a locale or a block.

Speaking in technical terms a locale corresponds to an "activation record" and a block corresponds to a "data area". That is, if during the execution of an ALGOL 68 program the range *S* showed in Figure 6.1 is entered, a locale *L* will be created in the MIAM, after which we say that "control resides in *L*". On arrival at the declaration of the array *A* a block *B* (for the elements of *A*) will be created. We shall say that *B* is "created in *L*".

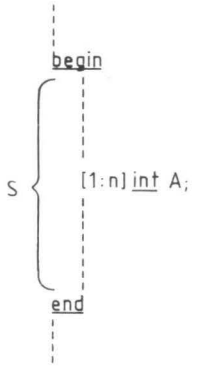


Figure 6.1

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

> Each locale *L* has:
>
> - *status(L)*: variable status,
> - *type(L)*: constant type,
> - *scope(L)*: constant integer,
> - *establisher(L)*: constant locale.

status(L), *type(L)*, *scope(L)* and *establisher(L)* will be called the "status", "type", "scope" and "establisher" of *L*, respectively. The status of *L* indicates whether *L* is "alive" or "dead":

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

Intuitively speaking a locale is alive if control resides in it or if control will ever return to it. Otherwise the locale is dead. The type of *L* is a value, the exact nature of which is completely irrelevant here. The only thing we need to know is that the (machine independent) type of *L* determines the (machine dependent) size of the piece of storage corresponding to *L*. The scope of *L* is the ALGOL 68 scope of the range corresponding to *L*. The latter is an integer which indicates the lifetime of the locale (the larger the scope, the shorter the locale will live). The establisher of *L* corresponds to what is usually called a "dynamic link". It is the locale where control resided immediately before control was transferred to *L*. For instance, if in Figure 6.1 prior to entering the range *S* control resides in the locale *M* and entry of the range *S* results in

the creation of a locale L, then *establisher(L)* = M.

Next consider blocks:

> Each block B has:
>
> - *status(B)*: variable status,
> - *type(B)*: constant type,
> - *scope(B)*: variable integer,
> - *generator(B)*: variable locale.

status(B), *type(B)*, *scope(B)* and *generator(B)* will be called the "status", "type", "scope" and "generator" of B, respectively. The status, type and scope of B are analogous to the corresponding entities associated with locales. The generator of B is the locale in which B was created. For instance, if in Figure 6.1 entry of the range S and execution of the declaration of A resulted in the creation of a locale L and a block B respectively, then immediately after that *generator(B)* = L. The reason why the scope and generator of a block are variable and not constant entities will be discussed later.

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

Let us first discuss processes informally. In general a number of processes will be active simultaneously during the execution of a program on the MIAM, where each process has its own control. Only when executing a sequential program is there only one active process. Suppose control of the active process P resides in the locale L corresponding to the range S in Figure 6.2. When control arrives at the parallel clause "par *(S1,S2,S3)*", which specifies that *S1*, *S2* and *S3* should be executed in parallel, three

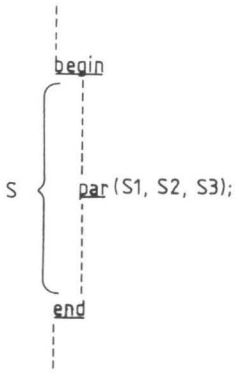


Figure 6.2

new active processes *P1*, *P2* and *P3* (corresponding to *S1*, *S2* and *S3*) will be created, while *P* becomes inactive until *P1*, *P2* and *P3* are completed. That is, *P* "ramifies" over *P1*, *P2* and *P3*. We shall say that *P1*, *P2* and *P3* are "created by *P* in *L*".

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

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

mode(P), *origin(P)*, *environ(P)* and *spawner(P)* will be called the "mode", "origin", "environ" and "spawner" of *P*, respectively. The mode of *P* indicates whether *P* is "active", "spawned" (= ramified over a number of processes) or "completed":

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

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

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

> The model consists of:
>
> - *L*: variable set of locales,
> - *B*: variable set of blocks,
> - *P*: variable set of processes,
> - *R*: variable process.

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

Prior to the execution of a program the following holds:

Initially:

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

where L_0 is a locale such that:

- $status(L_0) = alive$,
- $type(L_0) = \sim$,
- $scope(L_0) = 0$,
- $establisher(L_0) = L_0$,

and P_0 is a process such that:

- $mode(P_0) = active$,
- $origin(P_0) = L_0$,
- $environ(P_0) = L_0$,
- $spawner(P_0) = P_0$.

Here "~" denotes some unspecified type. The locale L_0, which will stay alive during the entire execution of a program, will be called the "initial locale". The process P_0 will be called the "initial process".

This completes the data structure part of the MIAM model. A thing one can argue about is whether the data structures described capture all information relevant to the storage management problem. An important concept that seems to be missing is that of a "reference" between areas, or put more abstractly, the concept of "reachability". This is an important concept because of the occurrence of "heap objects" in ALGOL 68, which correspond to areas with "infinite" lifetimes (their scope is zero). The only effective way to cope with the storage management problems caused by these objects is the use of a "garbage collector" (see Chapter 4, Section 4.1). The design of a garbage collector is a problem in its own right, which will not be discussed in this chapter. The concept of reachability will therefore not be introduced in the model. Instead a garbage collection operation will be introduced as a primitive operation in the problem definition. The design of a garbage collector, i.e., the implementation of the latter primitive operation, will be discussed in the next chapter.

6.1.2. Operations

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

Definition $\leq_\Lambda$ and $<_\Lambda$

$\leq_\Lambda$ and $<_\Lambda$ are relations on the set of all locales, defined as follows (L and M are locales):

$$L \leq_\Lambda M \Leftrightarrow \exists\, n \geq 0\ [L = establisher^n(M)],$$

$$L <_\Lambda M \Leftrightarrow L \leq_\Lambda M \wedge L \neq M.$$

Definition $\leq_\Pi$ and $<_\Pi$

$\leq_\Pi$ and $<_\Pi$ are relations on the set of all processes, defined as follows (P and Q are processes):

$$P \leq_\Pi Q \Leftrightarrow \exists\, n \geq 0\ [P = spawner^n(Q)],$$

$$P <_\Pi Q \Leftrightarrow P \leq_\Pi Q \wedge P \neq Q.$$

Here "$establisher^n(M)$" and "$spawner^n(Q)$" denote the result of applying *establisher* and *spawner* n times to M and Q respectively. So in other words, $\leq_\Lambda$ and $\leq_\Pi$ are the reflexive and transitive closures of the relations "$L = establisher(M)$" and "$P = spawner(Q)$" respectively, while $<_\Lambda$ and $<_\Pi$ are the antireflexive contractions of $\leq_\Lambda$ and $\leq_\Pi$ respectively. Note that all four relations are constant. Restricted to the sets $\mathcal{L}$ and $\mathcal{P}$ they are variable, however (because $\mathcal{L}$ and $\mathcal{P}$ are variable). To illustrate these relations, consider Figure 6.3 which shows a possible state of the machine. That is, it shows the locales, blocks and processes in $\mathcal{L}$, $\mathcal{B}$ and $\mathcal{P}$ respectively at a certain point of the execution of a program. In this figure among other things the following holds:

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

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

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

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

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

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

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

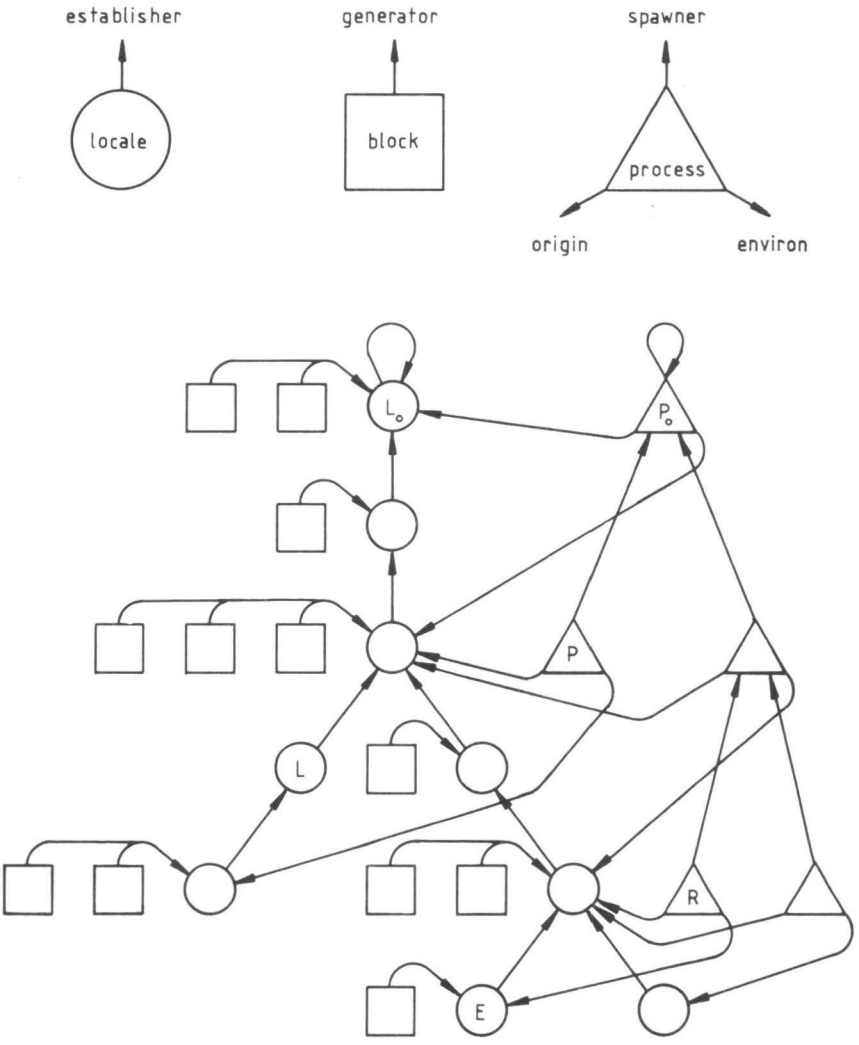

establisher
locale
generator
block
spawner
process
origin
environ
L_o
P_o
P
L
R
E

Figure 6.3

FINISH:
  Precondition:
    environ(R) ≠ *origin(R)*.
  Action:
    Let *E* = *environ(R)*.
    environ(R) := *establisher(E)*.
    status(E) := *dead*.
    For each *B* ∈ 𝔅 with *generator(B)* = *E*
    | *status(B)* := *dead*.

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

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

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

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

The fourth operation is somewhat trickier than the ones met with before in the sense that it does not correspond directly to any ALGOL 68 operation. It is an operation which is concerned with efficiency. The point is that it is sometimes useful to be able to extend the lifetime of a (large) block, for example to prevent an expensive copy operation. A typical example is found in passing result values of procedures. This lifetime extension is exactly what the following operation accomplishes:

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

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

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

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

$\mathcal{P} := \mathcal{P} \cup \mathcal{Q}$.
mode(R) := *spawned*.
Let $P \in \mathcal{P}$ with *mode(P)* = *active*.
R := *P*.

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

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

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

This operation "completes" the current running process *R*. The precondition is that control of *R* has returned to the locale in which *R* was created and

that R is not the initial process. Having changed the mode of R to *completed*, the process S which created R is determined. This is a spawned process, which should be made active if all processes created by it (all processes in the set $\mathcal{Q}$) are completed. Then, an arbitrary active process P must be selected and made to be the new running process.

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

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

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

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

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

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

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

6.1.3. Invariants

Before going deeper into the problem of storage management it is worthwhile to take a closer look at the model. The model satisfies a number of invariants ("system invariants" in the terminology of Chapter 4), which are listed below. They can be proved by showing that they hold initially and by checking that each operation, assuming its precondition holds, does not affect them. This is a simple job, which is left to the reader.

Invariants for L_0

(K1) $L_0 \in \mathcal{L}$.
(K2) $status(L_0) = alive$.
(K3) $scope(L_0) = 0$.
(K4) $establisher(L_0) = L_0$.

Invariants for P_0

(O1) $P_0 \in \mathcal{P}$.
(O2) $mode(P_0) \neq completed$.
(O3) $origin(P_0) = L_0$.
(O4) $spawner(P_0) = P_0$.

Invariants for R

(R1) $R \in \mathcal{P}$.
(R2) $mode(R) = active$.

Invariants for locales $L \in \mathcal{L}$

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

Invariants for blocks $B \in \mathcal{B}$

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

Invariants for processes $P \in \mathcal{P}$

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

From these invariants can be inferred that the relations $\leq_{\Lambda}$ and $\leq_{\Pi}$ indeed impose a tree structure on $\mathcal{L}$ and $\mathcal{P}$ with treetops L_0 and P_0 respectively, as was indicated in Figure 6.3. The set of all living locales in $\mathcal{L}$ constitutes a subtree with treetop L_0 of the tree imposed by $\leq_{\Lambda}$ on $\mathcal{L}$. The leaves of this subtree are formed by the environs of the active processes. Analogously, the set of all not yet completed processes in $\mathcal{P}$ constitutes a subtree with treetop P_0 of the tree imposed by $\leq_{\Pi}$ on $\mathcal{P}$. The leaves of this subtree are the active processes. Notice that the invariants imply that the operation "Let $P \in \mathcal{P}$ with $mode(P) = active$" in *COMPLETE* is well-defined. Notice also that dead areas and completed processes are really "garbage" in the model: They are not referenced or used in any other way any more.

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

Invariants for sequential programs

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

Together with Invariant (L4) this invariant implies that the living locales in $\mathcal{L}$ constitute a single linear list (the "dynamic chain") as indicated in Figure 6.4. The locales in $\mathcal{L}$ as a whole, however, need not constitute a linear list (but a tree).

6.2. PROBLEM

In this section the storage management problem is supposed to be defined. However, in the model as we described it there is no storage management problem. Areas which are created in the model simply appear out of the blue. The question of where they come from is completely irrelevant. The storage management problem is a problem which arises only in the *implementation* of the abstract machine. When implementing the abstract machine on a real machine the creation of an area must be modelled by "allocating" a piece of storage to it. In contrast to the number of areas the amount of storage is limited. It is here that the storage management problem arises. In order to arrive at the point where the storage management problem can be formulated, we will start implementing (the model

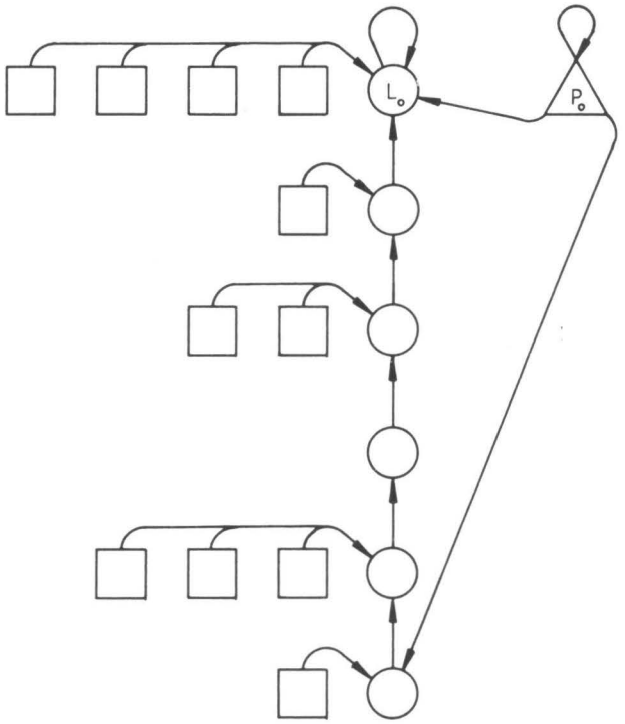


Figure 6.4

of) the abstract machine described before. The method of "adding and removing variables", which has been described in detail in Chapter 3, will be used for that purpose. In a nutshell this method amounts to the following. An algorithm (or a machine) is implemented by adding extra variables, and assignments to these variables, to the algorithm. This creates redundancy in the algorithm which enables certain expressions containing the "old" variables of the algorithm to be replaced by equivalent expressions containing the "new" variables. When applied in a systematic way, the old variables of the algorithm can be turned into "ghost variables", which may be removed from the algorithm. Thus an implementation of the algorithm in terms of the new variables is obtained. The method will be applied here by augmenting the model with an extra variable (the "allocation function"). Moreover, an abstract operation on this variable will be introduced. This operation is supposed to model or "implement" the creation of an area, which is expressed in its specification. The operation is inserted in the model at those points where areas are created. The storage management problem can then be defined as implementing this operation as efficiently as possible. In so doing, a number of primitive operations on the allocation function are allowed, which may be inserted throughout the model. In particular, an attempt should be made to "remove" as many abstract variables (such as for example

the "status" of areas) from the implementation. Thus the overhead caused by the storage management system is kept to a minimum.

The first thing that needs to be done is to introduce some model of a "store". This model should conform as closely to the store of the MIAM and the stores of existing machines as possible. We will assume the store to be a row of "cells" labelled by "addresses", which are integers $0,\ldots,N-1$. Here N is some (large) machine dependent integer. A set of consecutive cells in the store will be called a "field" and the number of cells in a field F will be denoted by $size(F)$. See Figure 6.5. Though this model of a store does not cover segmented memories, it is sufficiently general to be called machine independent.

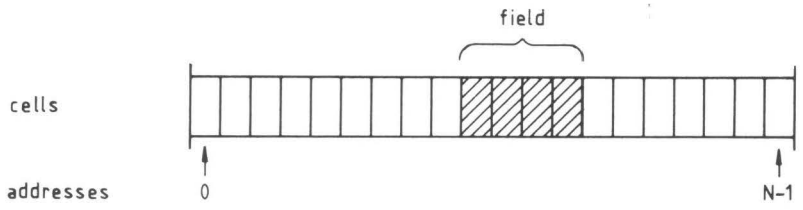


Figure 6.5

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

> The model is augmented with:
> - F: variable mapping from areas to fields.

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

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

Each locale L is augmented with:

- $size(L)$: constant integer.

Each block B is augmented with:

- $size(B)$: constant integer.

The two invariants which F must satisfy can now be formulated as follows:

Invariants for F

(F1) For each area $A,B \in dom(F)$
$A \neq B \Rightarrow F(A) \cap F(B) = \emptyset$.

(F2) For each area $A \in dom(F)$
$size(F(A)) = size(A)$.

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

Initially:

- $dom(F) = \{L_0\}$,
- $size(F(L_0)) = size(L_0)$.

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

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

ALLOCATE(A):
Precondition:
A is an area, $A \notin L \cup B$.
Action:
Establish the truth of the assertion $A \in dom(F)$ by means of the primitive operations defined on F.

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

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

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

Here "~" is some implementation dependent integer, which depends on the type t.

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

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

What are "no longer used" areas? One thing we know for sure is that dead areas are not used any more. So dead areas can be deallocated with impunity. Yet even the deallocation of all dead areas may not help. The only escape then is to deallocate no longer used living areas too. The

latter areas are considerably harder to detect than dead areas. The use of a "garbage collector" is required for that. The design of a garbage collector will not be discussed in this chapter. Consequently, an unspecified primitive operation *COLLECT GARBAGE* on F is introduced. This operation is supposed to deallocate all no longer used areas (including all dead areas), while it may also move areas. It is a very expensive operation which should only be used as a last resort. As far as certain properties of *COLLECT GARBAGE* are important or even essential to the storage management system to be designed, these properties will be postulated in the form of "Requirements for *COLLECT GARBAGE*". If even a garbage collection does not help, the only way out is to abort the program.

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

Primitive operations on F

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

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

The storage management problem now boils down to:

Problem

Implement *ALLOCATE(A)* efficiently.

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

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

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

6.3. DESIGN

A general approach to the design of a storage management system is to divide the areas into a number of classes, dependent on certain properties. For each class a special storage management strategy is used, which exploits the properties of the areas in that class. Let us assume n classes $C_1,\ldots,C_n$ of areas are distinguished. Then the allocation function F can correspondingly be written as $F = F_1 \cup \ldots \cup F_n$, where $dom(F_i) \subset C_i$ $(i = 1,\ldots,n)$. Let us call the set of all cells occupied by the areas in $dom(F_i)$ the "region" of F_i. The job is to implement the operation *ALLOCATE(A)* efficiently in terms of operations on the F_i. These operations may be chosen freely from the set of primitive operations defined on F. If

the operations are applied arbitrarily, a comprehensive bookkeeping is necessary in order to ensure the invariants for the allocation function are satisfied. This bookkeeping can be simplified greatly if the regions of the F_i are kept "compact" (= constituting a field). In that case only operations may be performed on the F_i, which do not disturb the compactness of the regions.

We shall comply with the above by abstractly modelling each F_i as a "pile" U_i. A pile is a stack of areas which (apart from *PUSH* and *POP*) has a number of additional operations defined on it (to be discussed later). If a pile U contains the areas $A_1,\ldots,A_m$ in the order from bottom to top, this will be denoted by $U = \langle A_1,\ldots,A_m\rangle$. A pile $U = \langle A_1,\ldots,A_m\rangle$ can be "located" in the store in two different ways as indicated in Figure 6.6. Here the areas A_i occupy contiguous fields of $size(A_i)$ cells.

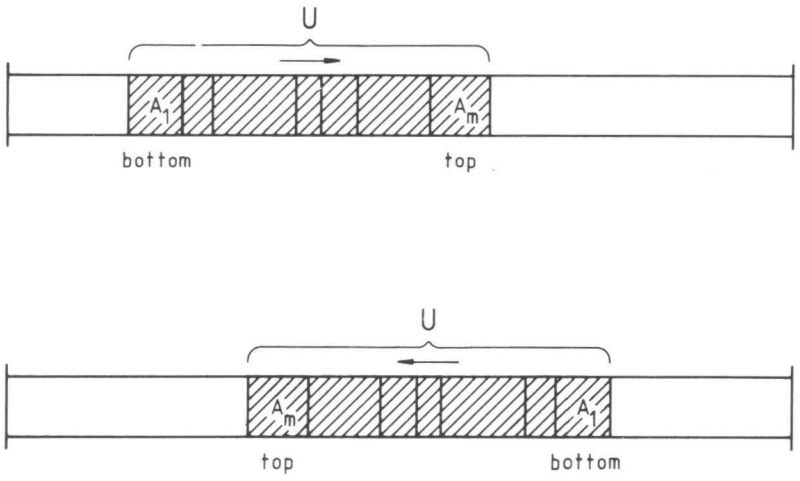


Figure 6.6

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

Up to two piles can be accommodated efficiently in the store (in the

case of two piles: one at each end of the store). Though storage management systems with a larger number of piles are certainly conceivable, we shall restrict ourselves to a maximum of two piles. The following are plausible choices:

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

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

The above implies that we have two piles S and H in our storage management system. S contains the areas with scope > 0 and H those with scope = 0. We assume they are located in the store as indicated in Figure 6.7.

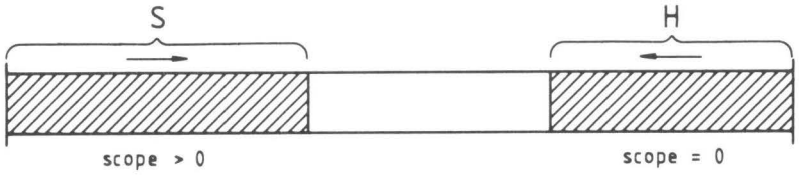


Figure 6.7

As with the allocation function F the piles S and H must satisfy a number of invariants. First, the fact that S and H correspond to (the domain of) a mapping (viz., F) implies that no area may occur twice in S and H. Secondly, the Invariants for F must be translated into invariants for S and H. Invariant (F1) boils down to the fact that the sum of the sizes of the areas in S and H must be less or equal to the size N of the store. Invariant (F2) need not nor can be expressed any more. (This invariant is incorporated in the correspondence between F and the piles S and H.) Thirdly, S and H should contain only areas with scope > 0 and scope = 0 respectively. So we have:

Invariants for S and H

(U1) If $S = \langle A_1,\ldots,A_m\rangle$, $H = \langle A_{m+1},\ldots,A_n\rangle$
| $i \neq j \Rightarrow A_i \neq A_j \quad (i,j = 1,\ldots,n)$.
(U2) $\Sigma\, A \in S \cup H\ [size(A)] \leq N$.
(U3) For each $A \in S$
| $scope(A) > 0$.
(U4) For each $A \in H$
| $scope(A) = 0$.

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

> Initially:
>
> - $S = <>$,
> - $H = <L_0>$.

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

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

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

> Requirements for *COLLECT GARBAGE*
>
> (1) No areas are added to S.
> (2) No areas are added to H.

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

> Requirements for *COLLECT GARBAGE*
>
> (3) The order of the remaining areas in S and H is not affected.

It states that the garbage collector must be "genetic order preserving", which is a desirable property of garbage collectors [TERASHIMA & GOTO 78]. Why this is so will come up soon. Notice that the removal of a number of areas from S and H may affect the compactness of the regions of S and H. Consequently, the garbage collector must perform a "compaction" in order to restore the situation of Figure 6.7. This need not be expressed in the Requirements for *COLLECT GARBAGE* because *COLLECT GARBAGE* is considered as an operation on the "abstract" piles S and H here. It follows directly from the correspondence between S and H and the allocation function F.

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

6.3.1. The initial system

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

```
FREE:
    N - Σ A ∈ S ∪ H [size(A)].
```

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

```
System 1

ALLOCATE(A):
    Let s = size(A).
    If FREE < s
    | COLLECT GARBAGE.
    | If FREE < s
    | | ABORT.
    Case
    1. scope(A) > 0
    | PUSH(A,S).
    2. scope(A) = 0
    | PUSH(A,H).
```

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

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

6.3.2. Avoiding successive garbage collections

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

System 2

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

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

6.3.3. Restraining the use of the garbage collector

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

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

```
FINISH:
   Precondition:
      environ(R) ≠ origin(R).
   Action:
      Let E = environ(R).
      environ(R) := establisher(E).
      status(E) := dead.
      For each B ∈ B with generator(B) = E
      | status(B) := dead.
      RELEASE.
```

```
JUMP(L,P):
   Precondition:
      L ∈ L, L ≤_Λ environ(R),
      P ∈ P, P ≤_Π R,
      origin(P) ≤_Λ L ≤_Λ environ(P).
   Action:
      R := P.
      mode(R) := active.
      environ(R) := L.
      For each M ∈ L with L <_Λ M
      | status(M) := dead.
      For each B ∈ B with L <_Λ generator(B)
      | status(B) := dead.
      For each Q ∈ P with P <_Π Q
      | mode(Q) := completed.
      RELEASE.
```

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

Storage management system 3 now looks as follows:

```
System 3

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

RELEASE:
   While DEADTOP
   | POP(S).
```

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

```
DEADTOP:
   S ≠ <> and status(TOP(S)) = dead.
```

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

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

6.3.4. Further restraining the use of the garbage collector

Suppose in *ALLOCATE(A)* we run out of storage (i.e., $FREE < s$). It may very well appear (especially if few areas with scope = *0* are used) that a relatively large part of the store is occupied by dead areas in S. The number of "dead cells" in S will be denoted by *DEAD*:

> *DEAD*:
> $\Sigma\ A \in S,\ status(A) = dead\ [size(A)]$.

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

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

The implementation of *COMPACT* will be discussed in Chapter 7, together with the implementation of *COLLECT GARBAGE*. Notice that the operation on the allocation function F corresponding to *COMPACT* is expressible in the primitive operations defined on F. Notice also that *COMPACT* does not violate the Invariants for S and H and that the operation is "genetic order preserving".

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

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

```
System 4

ALLOCATE(A):
    Let s = size(A).
    If FREE < s
    | If DEAD ≥ max(s,mindead)
    | | COMPACT.
    | else
    | | COLLECT GARBAGE.
    | | If FREE < max(s,minfree)
    | | | ABORT.
    Case
    1. scope(A) > 0
    | PUSH(A,S).
    2. scope(A) = 0
    | PUSH(A,H).

RELEASE:
    While DEADTOP
    | POP(S).
```

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

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

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

Before continuing, two more requirements on the garbage collector will be imposed. From the requirements introduced so far absolutely nothing can be inferred as to which areas are or are not deallocated by the garbage collector. There are certain areas for which it is easy to see that they are (or should be) or are not (or should not be) deallocated by the garbage

collector. In particular all dead areas will be deallocated by the garbage collector. (This was already stated informally.) Furthermore, the living locales will not be deallocated in a garbage collection. (They are "reachable" because control will, or should at least be able to return to them.) The following additional requirements, which allow us to use that information, will therefore be imposed on the garbage collector:

Requirements for *COLLECT GARBAGE*

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

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

Definition $<_S$

$<_S$ is a relation on the set of areas in S, defined as follows:

If $S = \langle A_1,\ldots,A_n\rangle$
$A_i <_S A_j \Leftrightarrow i < j \quad (i,j = 1,\ldots,n)$.

Due to Invariant (U1) this relation is well-defined. The fact that $A <_S B$ implies that A is "below" B in S. The following invariants hold:

Invariants for S and H

(U5) $S \cup H \subset L \cup \mathcal{B}$.
(U6) For each $L \in L$ with $status(L) = alive$
$L \in S \cup H$.
(U7) For each $L \in S \cap L$ with $establisher(L) \neq L_0$
$establisher(L) \in S$.
$establisher(L) <_S L$.
(U8) For each $B \in S \cap \mathcal{B}$
$generator(B) \in S$.
$generator(B) <_S B$.
(U9) If $S \neq \langle\rangle$
$status(TOP(S)) = alive$.

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

6.3.5. Removing overhead in the sequential case

In this subsection we shall assume that only sequential programs are executed on the abstract machine. So the operations *SPAWN(n)*, *COMPLETE* and *SWITCH* will not occur and Invariant (S1) will hold, i.e., $P = \{P_0\}$. The living locales in L then constitute a single dynamic chain, which emanates from $environ(P_0)$ (see Figure 6.4). Together with the Invariants for S and H this implies that the store looks like Figure 6.8. In this figure the circles represent the locales in the dynamic chain. Notice that if $S \neq <>$ there is always a living locale at the bottom of S, which amounts to the following invariant:

Invariants for sequential programs

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

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

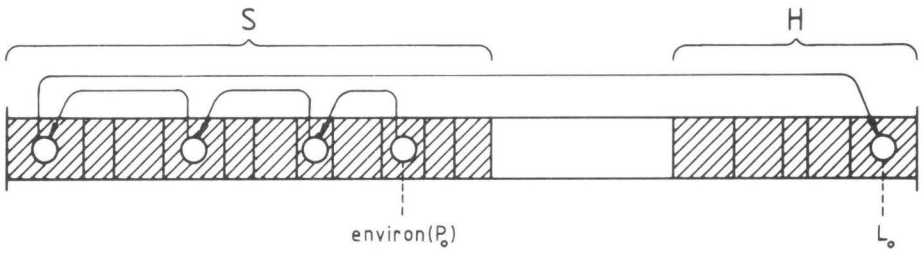


Figure 6.8

The locales indicated in Figure 6.8 are all alive. But what can we say about the "liveliness" of the other areas in S? We know there is a relation between the scope of an area and its lifetime. This relation is somewhat obscured by the operations *GENERATE(t,L)* and *KEEP(B,L)*. Can the scope of an area be used, anyway, in order to determine whether the area is dead or not? In order to answer this question the genetic order relation $<_S$ must be examined more closely.

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

Invariants for sequential programs

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

Next consider a living locale L in S and a dead block B above L in S. Let $G = generator(B)$ and suppose that $G <_S L$. From Invariant (B2) we know that G is dead. At the moment L was created G was already in S and also dead (otherwise Invariants (S3) and (L3) would lead to a contradiction). Since B was created after L this implies that B was created when G was already dead. From this and Invariant (B2) can be concluded that at the moment B was created, apparently $generator(B) \neq G$. Consequently, the operation $KEEP(B,G)$ must have been applied some time thereafter. The precondition of $KEEP(B,G)$ says that $G <_\Lambda environ(R)$, which implies that $status(G) = alive$, however. From this contradiction it can be concluded that the assertion $G <_S L$ can never hold. Since $G \neq L$ this leads to the conclusion that $L <_S G$, which is expressed in the following invariant:

Invariants for sequential programs

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

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

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

Definition $base(A)$ for sequential programs

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

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

Invariants for sequential programs

(S5) For each $A \in S$
$\mid status(A) = dead \Leftrightarrow scope(A) > scope(base(A)).$

PROOF

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

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

□

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

$$status(TOP(S)) = dead$$

is equivalent to:

$$scope(TOP(S)) > scope(environ(R)).$$

Consequently *DEADTOP* can be determined as follows:

Determination of *DEADTOP* for sequential programs

```
If S = <>
| DEADTOP := false.
else
| Let T = TOP(S).
| Let E = environ(R).
| DEADTOP := scope(T) > scope(E).
```

Next consider *ALLOCATE(A)*. In this operation the status of an area is used in the determination of the number *DEAD* of dead cells in *S* and in *COMPACT* (but not in *COLLECT GARBAGE*). From Invariants (S2) and (S5) (and a few more invariants) it can be inferred that the number *DEAD* can be determined as follows (see also Figure 6.8):

Determination of *DEAD* for sequential programs

```
Let A1,...,An be such that S = <A1,...,An>.
s := 0.
k := n.
L := environ(R).
While k > 0
| While Ak ≠ L
| | If scope(Ak) > scope(L)
| | | s := s + size(Ak).
| | k := k - 1.
| L := establisher(L).
| k := k - 1.
DEAD := s.
```

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

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

6.3.6. Removing overhead in the general case

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

avoid). It is therefore better to look for another solution.

In Section 6.2 it was stated that efficiency of the storage management system to be designed, should primarily be achieved for sequential programs. This implies that it is reasonable if the use of parallellism costs a little extra. It would not be reasonable if the overhead connected with parallellism had a negative effect on the efficiency of sequential programs. The "easy" way to avoid the latter is to have two storage management systems: one for sequential and one for parallel programs. Yet, having two different storage management systems is not a very desirable situation. Let us see how we can avoid it.

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

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

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

> Each locale L is augmented with:
>
> - *kind(L)*: constant kind.
>
> Each block B is augmented with:
>
> - *kind(B)*: variable kind.
>
> A kind is an element from the set {*simple*,*extended*}.

If A is an area, then $kind(A)$ will be called the "kind" of A. If an area has been created by P_0, its kind will be *simple*, which implies that its status and generator (if it is a block) need not be implemented. The reverse will not hold. There are two reasons for that. First of all, for areas with scope = 0, which need not be created by P_0, neither the status nor the generator need be implemented: Their status is invariably *alive* and their generator (for blocks) is invariably equal to L_0. The kind of these areas will therefore also be chosen to be *simple*. Secondly, we wish the following invariant to hold (why this invariant is useful will be seen soon):

> Invariants for blocks $B \in \mathcal{B}$
>
> (B4) $kind(B) = kind(generator(B))$.

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

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

> Initially:
>
> - $kind(L_0) = simple$.

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

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

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

$ALLOCATE(B)$.
$\mathsf{B} := \mathsf{B} \cup \{B\}$.

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

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

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

Invariants for L_0

(K5) $kind(L_0) = simple$.

Invariants for P_0

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

Invariants for locales $L \in \mathsf{L}$

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

Invariants for processes $P \in \mathsf{P}$

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

Furthermore, if we define the set T to be the set of simple areas in S:

Definition T

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

then the following analogues of Invariants (S2)-(S4) can be proved:

Invariants for S and H

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

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

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

If we (re)define the "base" of an area in T as follows:

Definition $base(A)$

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

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

Invariants for S and H

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

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

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

$$status(TOP(S)) = dead$$

is equivalent to:

$$scope(TOP(S)) > scope(environ(P_0)).$$

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

```
Determination of DEADTOP
If S = <>
| DEADTOP := false.
else
| Let T = TOP(S).
| If kind(T) = simple
| | Let E = environ(P0).
| | DEADTOP := scope(T) > scope(E).
| else
| | If T ∈ L
| | | DEADTOP := status(T) = dead.
| | else
| | | Let G = generator(T).
| | | DEADTOP := status(G) = dead.
```

Invariants (U10) and (U13) imply that the number of dead cells in S can be determined as follows:

```
Determination of DEAD
Let A1,...,An be such that S = <A1,...,An>.
s := 0.
k := n.
L := environ(P0).
While k > 0
| While k > 0 and Ak ≠ L
| | If kind(Ak) = simple
| | | If scope(Ak) > scope(L)
| | | | s := s + size(Ak).
| | else
| | | If Ak ∈ L
| | | | If status(Ak) = dead
| | | | | s := s + size(Ak).
| | | else
| | | | Let G = generator(Ak).
| | | | If status(G) = dead
| | | | | s := s + size(Ak).
| | k := k - 1.
| If k > 0
| | L := establisher(L).
| | k := k - 1.
DEAD := s.
```

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

6.3.7. Stripping the model

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

Each locale L has:

- *type(L)*: constant type,
- *scope(L)*: constant integer,
- *establisher(L)*: constant locale,
- *size(L)*: constant integer,
- *kind(L)*: constant kind,

If *kind(L)* = *extended*
- *status(L)*: variable status.

Each block B has:

- *type(B)*: constant type,
- *scope(B)*: variable integer,
- *size(B)*: constant integer,
- *kind(B)*: variable kind,

If *kind(B)* = *extended*
- *generator(B)*: variable locale.

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

Initially:

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

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

ESTABLISH(t):
Precondition:
t is a type.
Action:
Let E = *environ(R)*.
Let L be a locale such that:
- *type(L)* = t,
- *scope(L)* = *scope(E)* + 1,
- *establisher(L)* = E,
- *size(L)* = $\sim$,

If $R = P_0$
- *kind(L)* = *simple*.

else
- *kind(L)* = *extended*,
- *status(L)* = *alive*.

ALLOCATE(L).
$L := L \cup \{L\}$.
environ(R) := L.

FINISH:
Precondition:
environ(R) ≠ *origin(R)*.
Action:
Let *E* = *environ(R)*.
environ(R) := *establisher(E)*.
If *kind(E)* = *extended*
| *status(E)* := *dead*.
RELEASE.

GENERATE(t,L):
Precondition:
t is a type,
L ∈ 𝕃, *L* $\leq_{\Lambda}$ *environ(R)*.
Action:
Let *B* be a block such that:
- *type(B)* = *t*,
- *scope(B)* = *scope(L)*,
- *size(B)* = ~,
- *kind(B)* = *kind(L)*,
If *kind(L)* = *extended*
| - *generator(B)* = *L*.
ALLOCATE(B).
𝔹 := 𝔹 ∪ {*B*}.

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

Invariants for *S* and *H*

(U14) For each *L* ∈ *T* ∩ 𝕃 with *status(L)* = *alive*
and each *B* ∈ *T* ∩ 𝔹
| *generator(B)* = *L* ⇔ *L* $<_S$ *B* and *scope(B)* = *scope(L)*.

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

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

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

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

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

environ(R) := *L*.

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

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

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

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

Next, the action:

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

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

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

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

```
JUMP(L,P):
   Precondition:
      L ∈ L, L ≤Λ environ(R),
      P ∈ P, P ≤Π R,
      origin(P) ≤Λ L ≤Λ environ(P).
   Action:
      If R = P0
      | environ(R) := L.
      else
      | R := P.
      | mode(R) := active.
      | environ(R) := L.
      | For each M ∈ L with L <Λ M and kind(M) = extended
      | | status(M) := dead.
      | For each Q ∈ P with P <Π Q
      | | mode(Q) := completed.
      RELEASE.
```

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

6.3.8. Final implementation

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

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

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

```
JUMP(L,P):
   Precondition:
      L ∈ L, L ≤_Λ environ(R),
      P ∈ P, P ≤_Π R,
      origin(P) ≤_Λ L ≤_Λ environ(P).
   Action:
      If R = P_0
      | environ(R) := L.
      else
      | Let E = environ(P).
      | R := P.
      | mode(R) := active.
      | environ(R) := L.
      | If P ≠ P_0
      | | M := E.
      | | While M ≠ L
      | | | status(M) := dead.
      | | | M := establisher(M).
      | Q := {S ∈ P | spawner(S) = P, mode(S) ≠ completed}.
      | While Q ≠ ∅
      | | Let Q ∈ Q.
      | | Q := Q \ {Q}.
      | | mode(Q) := completed.
      | | M := environ(Q).
      | | While M ≠ origin(Q)
      | | | status(M) := dead.
      | | | M := establisher(M).
      | | Q := Q ∪ {S ∈ P | spawner(S) = Q, mode(S) ≠ completed}.
      | RELEASE.
```

In the other algorithms several optimizations are possible too.

The translation of the algorithms into abstract machine code will thus not pose any serious problems. The only operations which cannot be translated directly are *COLLECT GARBAGE* and *COMPACT*. The design of efficient algorithms for these operations (based on the specifications given previously) will be treated in the next chapter.

6.4. CONCLUSION

The implementation of a programming language is a highly complex process. In order to keep this process under control a "divide and rule" approach is mandatory. The job should be divided into clearly interfaced sub-tasks, which should be as independent of each other as possible. The division should be made with great care in order to keep the implementation as efficient as possible. One of the sub-tasks is the construction of a storage management system. In this chapter it has been demonstrated through an example that indeed the design of a storage management system can be viewed as a relatively independent part of the language implementation process. The interface with the other parts of the implementation consisted of an abstract model, which contained exactly the information relevant to the storage management problem and no more than that. It allowed us to approach the problem in a systematic and rigorous way, up to a level of formality which allowed proofs of correctness. Since all irrelevant details were discarded, the design process remained transparent and things could be kept relatively simple. The final result of the design process was an

efficient storage management system. The majority of the techniques used in this system are certainly not novel. The primary goal of this chapter was not to demonstrate some fancy storage management technique. Its main purpose was to demonstrate a technique to *design* a storage management system in a systematic way.

Apart from a number of advantages already mentioned in the introduction, the major advantage of the method demonstrated in this chapter over the more usual ("classical") approach to the design of storage management systems (such as described in [KNUTH 68], [BRANQUART & LEWI 71], [GRIES 71], [HILL 74]) is considered to be the fact that it forces one to a separation of concerns. In the process of designing a storage management system for an implementation of a programming language *L* on a machine *M* a number of concerns can be distinguished, which were clearly separated before:

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

Concerns (1) and (2) are often not well separated. In any but a purely interpretive implementation a storage management system should not be designed for (the machine corresponding to) the programming language *L*, but for the machine *M* into which code programs in *L* are translated. The operation *KEEP(B,L)*, which does not correspond to any ALGOL 68 construct, demonstrates this clearly. Of course, if the abstract machine approach is pursued, there will usually be a certain correspondence between *L* and *M* (the more abstract the machine is, the closer this correspondence will generally be). Good abstract machines (or better, their definitions) should be such that they can be implemented without using information on the programming language *L* or the way programs in *L* are translated into code for *M*. Admittedly, with the MIAM [MEERTENS 81] we were in a rather fortunate position in this respect. During the design process we only seldom needed a reference to ALGOL 68.

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

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

A sixth concern could be added to the above list of separated

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

CHAPTER 7

DESIGN OF A GARBAGE COLLECTOR

7.0. INTRODUCTION

In this chapter we shall design efficient algorithms for the operations *COLLECT GARBAGE* and *COMPACT* described (very) abstractly in Chapter 6. The treatment will be entirely independent of Chapter 6, using the same approach as in Chapter 6 to obtain independence of the abstract machine MIAM and ALGOL 68. That is, the interface with Chapter 6 will be laid down in a *model*. The model constitutes the basic data structure upon which the operations *COLLECT GARBAGE* and *COMPACT* will be defined. It constitutes what might be called a "concretized abstraction" of the model described in Chapter 6. This implies that, on the one hand, the former can be viewed as an abstraction of the latter in that only the relevant information from the latter is contained in the former. For example, we shall abstract from the distinction between "locales" and "blocks", and even forget about "processes" entirely. On the other hand, the model can be viewed as a "concretion" of the model from Chapter 6 in that it is augmented with a number of concepts which were regarded as "irrelevant" to the problem under consideration in Chapter 6. For example, we shall introduce the concept of a "branch" of an area and, associated with it, the concept of "reachability".

The model to be introduced bears a close resemblance to the general storage management model described in Chapter 4 (which was used for the description of the class of "all" garbage collection and compaction algorithms), though we have tailored it to our specific application. It differs from the model described in Chapter 6 in the following way. We shall not introduce "external" operations (such as *ESTABLISH(t)*, *FINISH*, etc.) and subsequently prove the invariants of the model. Instead, the invariants will be introduced as postulates (as in Chapter 4).

The model will be introduced in Section 7.1. For the reader of Chapter 6, the correspondence of this model with that of Chapter 6 will be indicated. After introducing the model, the problem will be defined in the same section by giving abstract definitions of the operations *COLLECT GARBAGE* and *COMPACT*. These definitions will comply with the definitions of *COLLECT GARBAGE* and *COMPACT* given in the previous chapter, including the "Requirements for *COLLECT GARBAGE*". In Section 7.2 the design of efficient algorithms for the latter operations by means of a process of transformation will be described. The transformation process will result in two efficient algorithms (for *COLLECT GARBAGE* and *COMPACT*, respectively).

The final algorithms from Section 7.2, though efficient(ly implementable), are still abstract in a number of respects. First of all, they use certain abstract parts of the model. Secondly, they are formulated in terms of abstract variables and "high level" control structures. In the end the algorithms will have to operate exclusively on the store (which is part of the model) and will have to be formulated in machine code (in order to execute them). The elimination of the abstract parts of the model from the algorithms and the translation of the algorithms into "hard code"

(which involves additional design decisions) is a rather straightforward affair, which could be omitted (as we did in Chapter 6). Still, having made a long journey to abstract worlds it is felt appropriate to conclude this monograph with a safe return to earth. In Section 7.3 we shall therefore, by way of illustration, perform the "final implementation" of the algorithms and translate them into machine code.

The obvious choice for the machine code would be (a subset of) the code of the abstract machine MIAM [MEERTENS 81], on the store of which the operations *COLLECT GARBAGE* and *COMPACT* are supposed to operate. The model introduced in Section 7.1, however, enables us to view the algorithms for *COLLECT GARBAGE* and *COMPACT* as general purpose runtime routines ("utilities"), which are independent of the MIAM. Instead of the code of the MIAM, which most readers will be unfamiliar with, a very simple code for a hypothetical machine will be used, in which to formulate the final versions of the algorithms. The latter machine code, the meaning of which can be described in a few lines, can be viewed as a simplified version of (a subset of) the code of the MIAM. The conclusion of this chapter can be found in Section 7.4.

7.1. MODEL

7.1.1. Framework

As in Chapter 6, the abstract objects which programs operate upon will be called "areas". Areas are the units of allocation and deallocation (so they correspond to the "nodes" of Chapter 4).

> Each area *A* has:
>
> - *status(A)*: constant element from {*alive*,*dead*},
> - *size(A)*: constant integer,
> - *atoms(A)*: constant set of atoms.

status(A) and *size(A)* will be called the "status" and "size" of *A*, respectively. The elements of *atoms(A)* will be called the "atoms" of *A*. The status of *A* indicates whether or not *A* has been "destroyed" by the program. The size of *A* indicates the size of the "field" (see Figure 6.5 in Chapter 6) which *A* occupies in the store (to be introduced below). The atoms of *A* are the elementary components of *A*:

> Each atom *X* has:
>
> - *offset(X)*: constant integer,
> - *kind(X)*: constant element from {*leaf*,*branch*},
>
> If *kind(X)* = *leaf*
> | - *value(X)*: constant integer,
>
> If *kind(X)* = *branch*
> | - *target(X)*: constant area.

offset(X), *kind(X)*, *value(X)* and *target(X)* will be called the "offset", "kind", "value" and "target" of *X*, respectively. The offset of *X* is the displacement of the location of *X* within the field of the (unique) area of which *X* is an atom. There are two kinds of atoms, which are called "leaves" and "branches", respectively. A leaf has a value, which is an integer. A branch "refers" to an area, to wit its target. Though programs may change the values of leaves and the targets of branches, the operations

COLLECT GARBAGE and *COMPACT* are supposed to leave these entities unaffected, which is the reason why they are defined to be constant. Atoms are supposed to occupy a single "cell" in the store. A cell C is merely a container of a value (an integer), which is called the "contents" of C:

> Each cell C has:
> - $cont(C)$: variable integer.

The framework of the model (i.e., the model without the definitions and invariants) is now described by a collection of four variables:

> The model consists of:
> - L: constant area,
> - S: variable sequence of areas,
> - H: variable sequence of areas,
> - C: constant sequence of cells.

Here L, which will be called the "root", corresponds to the initial locale L_0 from Chapter 6. S and H correspond to the "piles" S and H from Chapter 6. C represents the "store" (which was kept implicit in the model of Chapter 6).

7.1.2. Definitions and invariants

The definitions and invariants which hold in the model will now be presented. The invariants are prefixed by the capital letter I followed by a number. The following definitions and invariants are concerned with areas in general and the root L in particular.

> If A is an area, then the set $leaves(A)$ of atoms is defined by:
> $leaves(A) = \{X \in atoms(A) \mid kind(X) = leaf\}$.
>
> If A is an area, then the set $branches(A)$ of atoms is defined by:
> $branches(A) = \{X \in atoms(A) \mid kind(X) = branch\}$.
>
> (I1) For each area A,B
> $\mid A \neq B \Rightarrow atoms(A) \cap atoms(B) = \emptyset$.
>
> (I2) $status(L) = alive$.
>
> (I3) For each area A with $status(A) = alive$
> and each $X \in branches(A)$
> $\mid status(target(X)) = alive$.
>
> The fact that an area is reachable is defined by the following rules:
> (1) L is reachable.
> (2) If A is a reachable area,
> $X \in branches(A)$,
> $B = target(X)$,
> then B is reachable.
> (3) An area is reachable on account of the above rules only.
>
> An area is unreachable if it is not reachable.

Invariant (I1) states that different areas do not "overlap" (i.e., do not share atoms). Invariant (I2) corresponds to Invariant (K2) from Chapter 6. The absence of "dangling references" (i.e., references from live areas to dead areas) is expressed in Invariant (I3). The root L is the starting point for all access paths to other areas, as reflected in the definition of the concept of "reachability". Notice that Invariants (I2) and (I3) imply that reachable areas are always alive.

We now turn to the sequences S and H of areas:

(I4) If $S = \langle A_1,\ldots,A_m\rangle$ and $H = \langle A_{m+1},\ldots,A_n\rangle$
| $i \neq j \Rightarrow A_i \neq A_j \quad (i,j = 1,\ldots,n)$.

(I5) If $H = \langle A_1,\ldots,A_n\rangle$
| $n > 0$ and $A_1 = L$.

(I6) For each $A \in S \cup H$ with $status(A) = alive$
and each $X \in branches(A)$
| $target(X) \in S \cup H$.

(I7) For each $A \in H$
| $status(A) = alive$.

(I8) For each $A \in H$ with $A \neq L$
and each $X \in branches(A)$
| $target(X) \in H$.

Invariant (I4) corresponds to Invariant (U1) from Chapter 6. It must be formulated, not only because S and H contain different areas, but also because sequences may, in principle, contain multiple occurrences of their elements. Invariant (I5) expresses the fact that in the model of Chapter 6 the initial locale is invariably at the bottom of the pile H.

Invariant (I6) states that live areas in $S \cup H$ do not contain "dangling pointers" (i.e., references to already deallocated areas). In this invariant (and elsewhere, if convenient) the sequences S and H are considered as the sets of their elements. Invariant (I6) can be explained as follows. Dangling pointers can arise only because of the deallocation of areas. Apart from the areas deallocated by *COLLECT GARBAGE*, all areas deallocated in the model of Chapter 6 are dead areas. Due to the absence of "dangling references" (Invariant (I3)) this can never give rise to the occurrence of dangling pointers in live areas, though dangling pointers may occur in other dead areas. Notice that Invariant (I6), together with Invariants (I2) and (I5), implies that all reachable areas are contained in $S \cup H$.

Invariant (I7) is a direct translation of a "fact" from the model of Chapter 6, in contrast to Invariant (I8). The latter invariant states that there are no references from areas in $H \setminus \{L\}$ to areas in S. This invariant is due to the "scope rules" of ALGOL 68, which (almost) entirely carry over to the MIAM. (See also the remarks in Subsection 7.1.4.)

Definitions and invariants concerned with the store C are given below.

The integer N is defined to be the number of cells in C.

If $C = \langle C_0,\ldots,C_{N-1}\rangle$ and $a \in \{0,\ldots,N-1\}$, then *cell(a)* is the cell C_a.

(I9) For each $a,b \in \{0,\ldots,N-1\}$
| $a \neq b \Rightarrow cell(a) \neq cell(b)$.

If $C \in \mathbb{C}$, then $\underline{addr(C)}$ is the unique $a \in \{0,\ldots,N-1\}$ such that $cell(a) = C$.

Invariant (I9) states that the cells in the store $\mathbb{C}$ are all different, and therefore have a unique "address".

Finally, we introduce the concepts and invariants related to the representation of areas in the store.

(I10) For each area A
| $size(A) > 0$.

(I11) For each area A
and each $X \in atoms(A)$
| $0 \leq offset(X) < size(A)$.

(I12) For each area A
and each $X, Y \in atoms(A)$
| $X \neq Y \Rightarrow offset(X) \neq offset(Y)$.

(I13) $\Sigma\, A \in S \cup H\ [size(A)] \leq N$.

The integer $\underline{PTR(A)}$ for areas $A \in S \cup H$ is defined by the following rules:

(1) If $S = \langle A_1,\ldots,A_m\rangle$ and $A = A_k$
| $PTR(A) = \Sigma\, i = 1,\ldots,k-1\ [size(A_i)]$.
(2) If $H = \langle B_1,\ldots,B_n\rangle$ and $A = B_k$
| $PTR(A) = N - \Sigma\, i = 1,\ldots,k\ [size(B_i)]$.

The cell $\underline{LOC(X)}$ for atoms $X \in \bigcup A \in S \cup H\ [atoms(A)]$ is defined by:

$$LOC(X) = cell(PTR(A) + offset(X)),$$

where A is the unique area $A \in S \cup H$ such that $X \in atoms(A)$.

(I14) For each $A \in S \cup H$ with $status(A) = alive$
and each $X \in leaves(A)$
| $cont(LOC(X)) = value(X)$.

(I15) For each $A \in S \cup H$ with $status(A) = alive$
and each $X \in branches(A)$
| $cont(LOC(X)) = PTR(target(X))$.

An area A is represented by a field F of $size(A)$ cells in the store. Invariant (I10) guarantees that F is not empty. An atom X of A is represented by a cell C with displacement $offset(X)$ from the leftmost cell of F. Invariant (I11) makes sure that C is contained in F. The requirement that different atoms of A occupy different cells in F is expressed in Invariant (I12). In accordance with Chapter 6, the areas in S and H are supposed to be located in the store as indicated in Figure 7.1. The requirement that all areas must fit in the store is formulated in Invariant (I13) (cf. Invariant (U2) from Chapter 6). A reference to an area A is represented by the address of the leftmost cell of the field of A and is denoted by $PTR(A)$. The cell representing an atom X of an area is denoted by $LOC(X)$. (Capital letters are used for PTR and LOC to give warning that they are variable functions: They are affected by operations on S and H.) Invariants (I14) and (I15) define how the value of a leaf and the target of

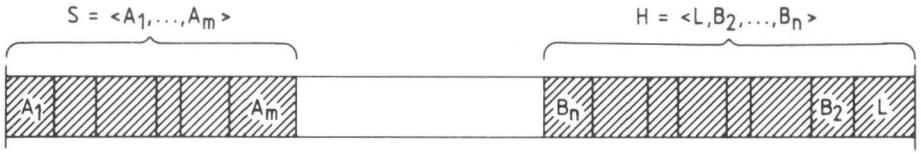


Figure 7.1

a branch of an area are represented in the store. Notice that these invariants are defined for all live areas in $S \cup H$ and not exclusively for the reachable areas. (The importance of this will become clear in Section 7.2.) Notice also that none of the invariants implies that all cells in the field of an area A are occupied by atoms of A. There may be "free" cells in the field of A, which can, for example, be used as workspace for *COLLECT GARBAGE* and *COMPACT*.

7.1.3. Operations

In the model which has been presented above, the operations *COLLECT GARBAGE* and *COMPACT* can be specified as follows:

> *COLLECT GARBAGE*:
> Let $Q = \{A \in S \cup H \mid A$ is unreachable$\}$.
> $S,H := S \setminus Q, H \setminus Q$.
> Restore the invariants of the model by changing the contents of cells in C.
>
> *COMPACT*:
> Let $Q = \{A \in S \mid status(A) = dead\}$.
> $S := S \setminus Q$.
> Restore the invariants of the model by changing the contents of cells in C.

Here the operation "$S := S \setminus Q$" implies that all areas from the set Q which occur in the sequence S are removed from S, while the order of the remaining areas in S is not affected (analogously for "$H := H \setminus Q$"). Notice that the operations *COLLECT GARBAGE* and *COMPACT* are quite different from the operations with the same name defined in Chapter 4.

It is easy to see that the above definitions comply with 4 of the 5 "Requirements for *COLLECT GARBAGE*" and the definition of *COMPACT* from Chapter 6. Since we have abstracted from the difference between "locales" and "blocks" Requirement (5) for *COLLECT GARBAGE* has become meaningless. Requirement (5) should be satisfied when incorporating the garbage collector to be developed here, in the storage management system of Chapter 6. It can be satisfied by keeping all live locales reachable (which they should be anyway; see the remark preceding Requirements (4) and (5) for *COLLECT GARBAGE*).

The design of efficient algorithms for *COLLECT GARBAGE* and *COMPACT* will be discussed in Section 7.2. We shall conclude this section with a number of remarks which provide additional information concerning the relation of the above model to ALGOL 68, the MIAM and the model from

Chapter 6. These remarks do not add anything to the model and can be skipped by the reader who prefers the "sec" model.

7.1.4. Some remarks

The "status" of an area, which was removed as a component from certain areas in Chapter 6, has returned in our model. This does not imply that the algorithms for *COLLECT GARBAGE* and *COMPACT* to be designed can only be incorporated in the storage management system of Chapter 6 if the status is re-introduced as a component of all areas. In the system of Chapter 6 a call of *COLLECT GARBAGE* or *COMPACT* is always preceded by a traversal of S to determine the number of dead cells in S (see System 4). During this traversal dead areas can be marked as such, thus providing the necessary information on the status of areas. (Areas in H are always alive; see Invariant (I7).)

In ALGOL 68 references to arbitrary subobjects of objects are allowed. Consequently, the MIAM features references to arbitrary subareas of areas. We have ignored this fact in the model: The branches of an area refer to areas and not to subareas of areas. This implies that the design decision to use "node marking" (see Subsection 5.1.5) has already been incorporated in the model. The advantage of node marking is its simplicity and time efficiency. Furthermore, the requirement that it must be possible to determine the area which a reference "points in" does not cost us anything extra: This information is already contained in references in the MIAM (where it is used for "scope checking"). The only remaining disadvantage of node marking (in this case) is that the garbage collector is more "conservative" than we might wish: It need not necessarily remove all areas from the store which are unreachable from the program point of view (see Example 5.6 in Subsection 5.1.5). A great deal of this objection can be removed by the code generator, who has control over the representation of ALGOL 68 objects in the store of the MIAM.

The "scope rules" of ALGOL 68 imply that there are no references from an object X to an object Y if $scope(X) < scope(Y)$. Analogously, in the MIAM there are no references from an area A to an area B if $scope(A) < scope(B)$. This assertion, however, applies only to "ALGOL 68 references" in the MIAM. Apart from the latter there are also other kinds of references in the MIAM, such as the references corresponding to the "establisher" of a locale or the "environ" of a process. Most of these references also satisfy the scope rules, but some of them do not. (For example, the "environ" reference of a process does not satisfy the scope rules because a process is embedded in its "origin".) In so far as references from blocks to other areas are concerned the scope rules are fully satisfied, which justifies Invariant (I8). (All areas in H, except L, are blocks.)

Invariant (I8) permits L to contain references to areas in S. If L contained no references to areas in S, all areas in S would be deallocated, which is usually not intended. Prior to initiating a garbage collection or compaction the storage manager is therefore supposed to store the "base pointers" through which all areas in S and H are accessed in L (if not already in L). In particular, L should contain a reference to the "running process".

As a final remark we note that unreachable live areas can occur not only in H, but also in S. This is due to the way "local generators" may be used in ALGOL 68.

7.2. DESIGN

In this section we are concerned with the design of efficient algorithms for the operations *COLLECT GARBAGE* and *COMPACT*. The algorithms will be designed through a process of transformation, starting with very abstract and obviously correct algorithms. For reasons of space, correctness proofs of the transformations are omitted. The transformation steps taken are generally so small, however, that it is simple to check their correctness (if not self-evident). In the rare case of a "large" transformation step, the necessary assertions (invariants) for a proof of correctness will be provided. In taking design decisions we shall to some extent rely on basic knowledge concerning garbage collection and compaction techniques (to be found in Chapter 5).

This section will be divided into two subsections, which are concerned with the design of algorithms for *COLLECT GARBAGE* and *COMPACT* respectively. The design of an algorithm for *COMPACT* is partly analogous to the design of an algorithm for *COLLECT GARBAGE* and will therefore be discussed in a less detailed way than the latter.

7.2.1. *COLLECT GARBAGE*

7.2.1.1. A straightforward algorithm

It is the job of *COLLECT GARBAGE* to delete all unreachable areas from the sequences S and H and restore the invariants of the model by changing the contents of cells in C. Inspection of the invariants tells us that only Invariants (I14) and (I15) are violated by the deletion of all unreachable areas from S and H (because the remaining areas in S and H occupy new fields in the store). The obvious way to restore these invariants leads to the following algorithm (in which we use the fact that all reachable areas are alive):

```
COLLECT GARBAGE_1:
   Variables:
      None.
   Action:
      Let Q = {A ∈ S ∪ H | A is unreachable}.
      S,H := S \ Q,H \ Q.
      For each A ∈ S ∪ H
      and each X ∈ leaves(A)
      | cont(LOC(X)) := value(X).
      For each A ∈ S ∪ H
      and each X ∈ branches(A)
      | cont(LOC(X)) := PTR(target(X)).
```

Notice that the two for-loops may be interchanged. More than that, they may even be "fused" arbitrarily.

7.2.1.2. Preserving the allocation information

The above algorithm, though evidently correct, has no practical value whatsoever. The reason is that it has been formulated almost exclusively in abstract terms. In order to turn the algorithm into a practical algorithm the abstract parts must be transformed in such a way that only operations on (the cells in) the store C remain. This implies, for example, that in the final algorithm the abstract "value" of a leaf X, which is used in

COLLECT GARBAGE$_1$, will have to be extracted from the store. Indeed, we know from Invariant (I14) that this is possible: Prior to the first for-loop the cell formerly occupied by X contains the value of X. Still, we run into a problem since after the statement:

$$S,H := S \setminus Q, H \setminus Q$$

all information concerning the old fields of areas has been lost. First of all, we shall therefore reformulate *COLLECT GARBAGE*$_1$ in such a way that this "allocation information" is preserved. We shall do this by moving the above statement to the end of the algorithm. This has the conceptual advantage that S and H, and consequently *PTR(A)* for areas A, and *LOC(X)* for atoms X, are constant in the entire algorithm except in the last statement. The values which *PTR(A)* and *LOC(X)* will have at the end of the algorithm will be denoted by *PTR*(A)* and *LOC*(X)* respectively. The reformulation of *COLLECT GARBAGE*$_1$ (which is almost straightforward) leads to:

COLLECT GARBAGE$_2$:
  Variables:
    None.
  Action:
    Let $Q = \{A \in S \cup H \mid A$ is unreachable$\}$.
    For each $A \in (S \cup H) \setminus Q$
    and each $X \in$ *leaves(A)*
    | *cont(LOC*(X))* := *value(X)*.
    For each $A \in (S \cup H) \setminus Q$
    and each $X \in$ *branches(A)*
    | *cont(LOC*(X))* := *PTR*(target(X))*.
    $S,H := S \setminus Q, H \setminus Q$.

  PTR(A)*:
    Case
    1. $S = \langle A_1,\ldots,A_m\rangle$ and $A = A_k$
    | $\Sigma\ i = 1,\ldots,k-1,\ A_i \notin Q$ [*size*(A_i)].
    2. $H = \langle B_1,\ldots,B_n\rangle$ and $A = B_k$
    | $N - \Sigma\ i = 1,\ldots,k,\ B_i \notin Q$ [*size*(B_i)].

  LOC(X)*:
    cell(PTR(A)* + *offset(X))*,
    where A is the unique area $A \in S \cup H$ such that $X \in$ *atoms(A)*.

7.2.1.3. Preserving the information contained in the fields of areas

It is tempting to replace the abstract "*value(X)*" in the first for-loop of *COLLECT GARBAGE*$_2$ by the more concrete "*cont(LOC(X))*". We have to be careful, though, since the original contents of *LOC(X)* may already have been "overwritten". In general, a statement such as:

cont(LOC(X))* := *cont(LOC(X))*

will destroy the information originally contained in *LOC*(X)*. This may not only be information concerning the values of leaves, but also information concerning the targets of branches or special information for the garbage collector (contained in the "free" cells of fields). All information of interest is contained in the fields of reachable areas (the rest is "garbage"). It makes sense, therefore, in the first for-loop of

COLLECT GARBAGE$_2$ to "move" entire areas, i.e., copy the contents of their old fields to their new fields, instead of copying the contents of the locations of leaves only. It is easy to see that this can be accomplished without loss of information if the (reachable) areas in S are moved in order from left to right (in the store) and those in H in order from right to left (in the store) as described in the following transformed version of *COLLECT GARBAGE*$_2$:

```
COLLECT GARBAGE3:
   Variables:
      None.
   Action:
      Let Q = {A ∈ S ∪ H | A is unreachable}.
      Let A1,...,Am be such that S = <A1,...,Am>.
      Let B1,...,Bn be such that H = <B1,...,Bn>.
      For k = 1 to m
      | If Ak ∉ Q
      | | MOVEℓ(Ak,PTR*(Ak)).
      For k = 1 to n
      | If Bk ∉ Q
      | | MOVEr(Bk,PTR*(Bk)).
      For each A ∈ (S ∪ H) \ Q
      and each X ∈ branches(A)
      | cont(LOC*(X)) := PTR*(target(X)).
      S,H := S \ Q,H \ Q.

   MOVEℓ(A,a):
      Let b = PTR(A).
      For i = 0 to size(A) - 1
      | cont(cell(a + i)) := cont(cell(b + i)).

   MOVEr(A,a):
      Let b = PTR(A).
      For i = size(A) - 1 down to 0
      | cont(cell(a + i)) := cont(cell(b + i)).
```

Notice that *COLLECT GARBAGE*$_3$ is a correct implementation of *COLLECT GARBAGE* (the definition) but not of *COLLECT GARBAGE*$_2$: Because the contents of all cells in the old field of an area (including the contents of "free" cells) are copied to the new field of the area, *COLLECT GARBAGE*$_3$ may change the contents of cells which were not affected by *COLLECT GARBAGE*$_2$. Yet we shall call the transformation from *COLLECT GARBAGE*$_2$ to *COLLECT GARBAGE*$_3$ "correctness-preserving", since it is the definition of *COLLECT GARBAGE* which determines the correctness of the algorithm.

7.2.1.4. Preserving the updating information

Now let us consider the last for-loop in *COLLECT GARBAGE*$_3$, where "pointers" (i.e., the contents of cells occupied by branches) are "updated". In this loop the abstract "target" of a branch X is used. In the final algorithm the information concerning the target of X must again be extracted from the store. We know from Invariant (I15) and the fact that areas have been moved that this information is contained in *LOC*(X)* and consists of the integer *PTR(target(X))*, i.e., a pointer to the old field of *target(X)*. Since areas have been moved, however, the necessary information to derive *PTR*(target(X))* from *PTR(target(X))* will generally have been

lost. We have the choice now, either to collect and store that information prior to moving the areas, or to perform the updating of pointers before the moving of areas instead of afterwards. The first choice amounts to the use of a "relocation map" (see Subsection 5.2.2), which gives rise to complex algorithms. We "know" we can do better than that and therefore choose the second option. This implies that the last for-loop is moved to the point immediately before the "moving phase". Since at that point areas have not yet been moved, "$LOC^*(X)$" must be replaced by "$LOC(X)$":

```
COLLECT GARBAGE4:
   Variables:
      None.
   Action:
      Let Q = {A ∈ S∪H | A is unreachable}.
      Let A1,...,Am be such that S = <A1,...,Am>.
      Let B1,...,Bn be such that H = <B1,...,Bn>.
      For each A ∈ (S∪H)\Q
      and each X ∈ branches(A)
      | cont(LOC(X)) := PTR*(target(X)).
      For k = 1 to m
      | If Ak ∉ Q
      | | MOVEℓ(Ak,PTR*(Ak)).
      For k = 1 to n
      | If Bk ∉ Q
      | | MOVEr(Bk,PTR*(Bk)).
      S,H := S\Q,H\Q.
```

7.2.1.5. Marking reachable areas

Having rearranged the operations in the proper way, we are now in a position to try and "improve" the algorithm by adding (abstract) variables (which will again be removed in Subsection 7.3.1). The first problem is how to determine efficiently the set Q of unreachable areas. This will, as usual, be accomplished indirectly by "marking" all reachable areas. For that purpose a variable set M of areas is introduced. The marking of reachable areas is now abstractly described by the statement:

$$M := \{A \in S \cup H \mid A \text{ is reachable}\}.$$

By putting this statement at the beginning of *COLLECT GARBAGE*$_4$, expressions such as "$A \in (S \cup H) \setminus Q$", "$A_k \notin Q$" and "$B_k \notin Q$" can be replaced by "$A \in M$", "$A_k \in M$" and "$B_k \in M$", respectively. This would not take us any further, unless we "refine" the above assignment to M and replace it by a "marking algorithm". Here we appeal to knowledge of garbage collection algorithms (see Chapter 5) and choose (in this case) the most efficient "node marking" algorithm we know of (see Algorithm GN.DTEH in Subsection 5.1.5). This algorithm, which requires an auxiliary variable set T of areas, is incorporated in:

```
COLLECT GARBAGE_5:
   Variables:
      M: set of areas,
      T: set of areas.
   Action:
      Let Q = {A ∈ S ∪ H | A is unreachable}.
      Let A_1,...,A_m be such that S = <A_1,...,A_m>.
      Let B_1,...,B_n be such that H = <B_1,...,B_n>.
      M,T := {L},{L}.
      While T ≠ ∅
      | Get A from T.
      | For each X ∈ branches(A)
      | | Let B = target(X).
      | | If B ∉ M
      | | | M,T := M ∪ {B},T ∪ {B}.
      For each A ∈ M
      and each X ∈ branches(A)
      | cont(LOC(X)) := PTR*(target(X)).
      For k = 1 to m
      | If A_k ∈ M
      | | MOVE_k(A_k,PTR*(A_k)).
      For k = 1 to n
      | If B_k ∈ M
      | | MOVE_n(B_k,PTR*(B_k)).
      S,H := S \ Q,H \ Q.
```

Here "Get A from T" is equivalent to:

```
Let A ∈ T.
T := T \ {A}.
```

Notice that we did not remove Q from the algorithm. The reason for this is not only that Q is used in the definition of PTR^*, but also that it relieves us from the duty of transforming the statement "$S,H := S \setminus Q, H \setminus Q$" each time we transform M. (In the final algorithm the latter entirely abstract statement reduces to an empty action anyway.)

7.2.1.6. Calculating $PTR^*(A)$

A second problem is caused by the calculation of the $PTR^*(A)$ $(A \in M)$. It is easy to see that the calculation of these values in the moving phase can be accomplished in an overall $O(n)$ time (where $n = \#M$). The calculation of the $PTR^*(A)$ in the updating phase poses a problem, however. A straightforward application of the definition of $PTR^*(A)$ to calculate $PTR^*(target(X))$ from $cont(LOC(X))$ (= $PTR(target(X))$) would result in an $O(n^2)$ time for a garbage collection. A way to avoid this is to calculate all $PTR^*(A)$ for $A \in M$ once and for all and store these values "somewhere", say in $F(A)$, where F is a variable mapping from areas to integers. (Typically, F is implemented by storing $F(A)$ in a free cell of the field of A for $A \in M$.) Using F, the updating of pointers can be described by:

```
F := ∅.
For each A ∈ M
| F(A) := PTR*(A).
For each A ∈ M
and each X ∈ branches(A)
| cont(LOC(X)) := F(target(X)).
```

As can be seen, this solution will cost us an extra "scan". Another solution is, after calculating $PTR^*(A)$, not to store $PTR^*(A)$ but to use it immediately to update all pointers to A. If we define the set of branches $B(A)$ for each area $A \in M$ by:

$$B(A) = \{X \in \bigcup B \in M\ [branches(B)] \mid target(X) = A\},$$

this can be described by:

```
For each A ∈ M
| Let a = PTR*(A).
| For each X ∈ B(A)
| | cont(LOC(X)) := a.
```

At first sight it seems that we do not need an extra scan now. However, we are faced with the problem of determining the sets $B(A)$ $(A \in M)$. Calculation of the B-sets "on the spot" would be unacceptably inefficient, so we shall have to determine and store them beforehand. This raises the question of whether it is possible to store the B-sets efficiently at all. Again, we appeal to knowledge concerning garbage collection techniques to contend that these "branch sets" can be stored efficiently, even without any space overhead at all (see Subsection 5.2.3). Turning B into a variable function which maps areas to sets of branches, the construction of the B-sets can be described by:

```
B := {(A,∅) | A ∈ S ∪ H}.
For each A ∈ M
and each X ∈ branches(A)
| Let B = target(X).
| B(B) := B(B) ∪ {X}.
```

This amounts to an extra scan, which would invalidate the advantage of the B-solution over the F-solution. Let us consider more closely what we do in the above loop, however: For each reachable area A and each branch X of A the target B of X is determined and then X is put in $B(B)$. Apart from the latter, that is exactly what is done during marking! The construction of the B-sets can therefore be combined most efficiently with the marking phase, as described in the following version of *COLLECT GARBAGE* (cf. Algorithm G&CNK in Section 5.3):

```
COLLECT GARBAGE_6:
   Variables:
      M: set of areas,
      T: set of areas,
      B: mapping from areas to sets of branches.
   Action:
      Let Q = {A ∈ S ∪ H | A is unreachable}.
      Let A_1,...,A_m be such that S = <A_1,...,A_m>.
      Let B_1,...,B_n be such that H = <B_1,...,B_n>.
      M,T,B := {L},{L},{(A,∅) | A ∈ S ∪ H}.
      While T ≠ ∅
      | Get A from T.
      | For each X ∈ branches(A)
      | | Let B = target(X).
      | | If B ∉ M
      | | | M,T := M ∪ {B},T ∪ {B}.
      | | B(B) := B(B) ∪ {X}.
      For each A ∈ M
      | Let a = PTR*(A).
      | While B(A) ≠ ∅
      | | Get X from B(A).
      | | cont(LOC(X)) := a.
      For k = 1 to m
      | If A_k ∈ M
      | | MOVE_ℓ(A_k,PTR*(A_k)).
      For k = 1 to n
      | If B_k ∈ M
      | | MOVE_r(B_k,PTR*(B_k)).
      S,H := S \ Q,H \ Q.
```

Notice that we have formulated the updating part of the algorithm in such a way that at the end of the algorithm $B(A) = \emptyset$ for each $A \in S \cup H$.

7.2.1.7. Removing PTR^* from the algorithm

In the two for-loops of the moving phase of the above algorithm the calculation of the values $PTR^*(A)$ for $A \in M$ can be accomplished in an overall $O(n)$ time by keeping track of a counter during the traversal of S and H. This assertion also applies to the updating phase, if we traverse S and H in a way similar to the moving phase. In contrast to the moving phase we can choose the direction of traversal of the sequences S and H: either from first to last element, or from last to first element. The obvious choice may seem to be to use the order from first to last, as we (perforce) did in the moving phase. For S this traversal order is indeed obvious, but for H it is not. The reason for this is that a reference to an area A is represented by the address of the leftmost cell of the field of A. This makes it extremely simple, given a pointer to an area A, to find the pointer of the area immediately to the right of A in the store. Unless we introduce an overhead, finding the pointer of the area immediately to the left of A in the store is much more difficult. The natural order of traversing S and H is therefore from left to right in the store, which for H means from last to first element (see Figure 7.1).

The above raises a question concerning the traversal of H in the moving phase. There the traversal order is the "difficult" order from first to last. How are we going to implement that? Fortunately, the "easy" traversal of H in the updating phase can be used to make the reverse

traversal in the moving phase, and even the moving of areas itself, extremely simple and efficient (see Subsection 7.3.1). There is another problem, though. When starting to traverse H in the updating phase, we have to know the value $PTR^*(B)$ for the first marked area B to be encountered in H. This value is equal to:

$$\begin{aligned} PTR^*(B) &= N - \Sigma\, A \in H \cap M\ [size(A)] \\ &= N - (r - a), \end{aligned}$$

where $r = \Sigma\, A \in M\ [size(A)]$
and $a = \Sigma\, A \in S \cap M\ [size(A)]$.

This implies that $PTR^*(B)$ can be determined by keeping track of r during the marking phase and traversing S before H in the updating phase to determine a. All this leads to the following algorithm in which PTR^* no longer occurs:

```
COLLECT GARBAGE7:
   Variables:
      M: set of areas,
      T: set of areas,
      B: mapping from areas to sets of branches,
      a,r: integer.
   Action:
      Let Q = {A ∈ S ∪ H | A is unreachable}.
      Let A1,...,Am be such that S = <A1,...,Am>.
      Let B1,...,Bn be such that H = <B1,...,Bn>.
      M,T,B,r := {L},{L},{(A,∅) | A ∈ S ∪ H},size(L).
      While T ≠ ∅
      | Get A from T.
      | For each X ∈ branches(A)
      | | Let B = target(X).
      | | If B ∉ M
      | | | M,T,r := M ∪ {B},T ∪ {B},r + size(B).
      | | B(B) := B(B) ∪ {X}.
      a := 0.
      For k = 1 to m
      | If Ak ∈ M
      | | UPDATE(Ak,a).
      | | a := a + size(Ak).
      a := N - (r - a).
      For k = n down to 1
      | If Bk ∈ M
      | | UPDATE(Bk,a).
      | | a := a + size(Bk).
      a := 0.
      For k = 1 to m
      | If Ak ∈ M
      | | MOVEℓ(Ak,a).
      | | a := a + size(Ak).
      a := N.
      For k = 1 to n
      | If Bk ∈ M
      | | a := a - size(Bk).
      | | MOVEr(Bk,a).
      S,H := S \ Q,H \ Q.
```

```
UPDATE(A,a):
   While B(A) ≠ ∅
   | Get X from B(A).
   | cont(LOC(X)) := a.
```

Check that, if each "primitive" operation (such as "Get A from T") takes $O(1)$ time, the algorithm operates in $O(m+n)$ time, where:

$$m = \#(S \cup H),$$
$$n = \Sigma\, A \in S \cup H,\ A \text{ is reachable } [size(A)].$$

Here "S" and "H" denote the initial values of S and H.

We shall stop the design here. Though the algorithm still contains many abstract expressions, the transformation of these expressions into concrete expressions is almost entirely a purely technical matter, which will no longer be called "design", but rather "implementation". The transformation process of the algorithm will therefore be continued in Subsection 7.3.1. The above description of the garbage collector is believed to be suitable for inclusion in the accompanying documentation of the compiler in which the garbage collector is used. The algorithm is still reasonably readable, in contrast to the final product of Subsection 7.3.1.

7.2.2. *COMPACT*

7.2.2.1. A straightforward algorithm

The operation *COMPACT* should delete all dead areas from S and restore the invariants of the model by changing the contents of cells in C. Inspection of the invariants tells us that, again, only Invariants (I14) and (I15) are violated by the deletion of all dead areas from S. A straightforward restoration of these invariants leads to:

```
COMPACT1:
   Variables:
      None.
   Action:
      Let Q = {A ∈ S | status(A) = dead}.
      S := S \ Q.
      For each A ∈ S ∪ H
      and each X ∈ leaves(A)
      | cont(LOC(X)) := value(X).
      For each A ∈ S ∪ H
      and each X ∈ branches(A)
      | cont(LOC(X)) := PTR(target(X)).
```

7.2.2.2. Limiting the restoration overhead of the invariants

A first observation is that the statement:

$$S := S \setminus Q$$

only "partly" violates Invariants (I14) and (I15) in that it keeps the following assertions invariant:

For each $A \in H$
and each $X \in leaves(A)$
| $cont(LOC(X)) = value(X)$.

For each $A \in H \setminus \{L\}$
and each $X \in branches(A)$
| $cont(LOC(X)) = PTR(target(X))$.

The invariance of the second assertion is based critically on the fact that there are no references from areas in $H \setminus \{L\}$ to areas in S (Invariant (I8)). The above implies that $COMPACT_1$ can be rewritten as follows:

$COMPACT_2$:
Variables:
None.
Action:
Let $Q = \{A \in S \mid status(A) = dead\}$.
$S := S \setminus Q$.
For each $A \in S$
and each $X \in leaves(A)$
| $cont(LOC(X)) := value(X)$.
For each $A \in S \cup \{L\}$
and each $X \in branches(A)$
| $cont(LOC(X)) := PTR(target(X))$.

7.2.2.3. Preserving the allocation information

In a first attempt to replace the abstract entities in the above algorithm by values extracted from the store, we run into the same problems as in $COLLECT\ GARBAGE_1$: First of all, after the statement:

$S := S \setminus Q$

all information on the old fields of areas has been lost. This problem will be solved in the same way as in $COLLECT\ GARBAGE_2$, i.e., by moving the above statement to the end of the algorithm. Expressions of the kind "$PTR(A)$" for areas A and "$LOC(X)$" for atoms X must then be replaced by expressions denoting the values which $PTR(A)$ and $LOC(X)$ will have after the removal of dead areas from S. These values will again be denoted by $PTR^*(A)$ and $LOC^*(X)$ respectively. The other necessary modifications of the algorithm are obvious and lead to:

$COMPACT_3$:
Variables:
None.
Action:
Let $Q = \{A \in S \mid status(A) = dead\}$.
For each $A \in S \setminus Q$
and each $X \in leaves(A)$
| $cont(LOC^*(X)) := value(X)$.
For each $A \in (S \setminus Q) \cup \{L\}$
and each $X \in branches(A)$
| $cont(LOC^*(X)) := PTR^*(target(X))$.
$S := S \setminus Q$.

```
PTR*(A):
   Case
   1. S = <A_1,...,A_m> and A = A_k
   | Σ i = 1,...,k-1, A_i ∉ Q [size(A_i)].
   2. H = <B_1,...,B_n> and A = B_k
   | N - Σ i = 1,...,k [size(B_i)].

LOC*(X):
   cell(PTR*(A) + offset(X)),
   where A is the unique area A ∈ S ∪ H such that X ∈ atoms(A).
```

7.2.2.4. Preserving the information contained in the fields of areas

A second problem is the possible loss of information because of the overwriting of the original contents of cells in the fields of live areas. As in *COLLECT GARBAGE*$_3$ we shall solve this problem by rewriting the first for-loop in such a way that the live areas in S are "moved" in order from left to right to their new fields, as described in:

```
COMPACT_4:
   Variables:
      None.
   Action:
      Let Q = {A ∈ S | status(A) = dead}.
      Let A_1,...,A_m be such that S = <A_1,...,A_m>.
      For k = 1 to m
      | If A_k ∉ Q
      | | MOVE_ℓ(A_k, PTR*(A_k)).
      For each A ∈ (S \ Q) ∪ {L}
      and each X ∈ branches(A)
      | cont(LOC*(X)) := PTR*(target(X)).
      S := S \ Q.

   MOVE_ℓ(A,a):
      Let b = PTR(A).
      For i = 0 to size(A) - 1
      | cont(cell(a + i)) := cont(cell(b + i)).
```

7.2.2.5. Preserving the updating information

A third problem is the fact that in the above algorithm the information necessary to derive $PTR^*(target(X))$ from $PTR(target(X))$ (= $cont(LOC(X))$ according to Invariant (I15)) has been lost. Again we can either store that information before moving the areas, or perform the updating of pointers before moving the areas. For the same reasons as before we shall choose the latter:

```
COMPACT_5:
   Variables:
      None.
   Action:
      Let Q = {A ∈ S | status(A) = dead}.
      Let A_1,...,A_m be such that S = <A_1,...,A_m>.
      For each A ∈ (S \ Q) ∪ {L}
      and each X ∈ branches(A)
      | cont(LOC(X)) := PTR*(target(X)).
      For k = 1 to m
      | If A_k ∉ Q
      | | MOVE_ℓ(A_k,PTR*(A_k)).
      S := S \ Q.
```

Notice, by the way, that if Invariant (I15) had been defined for reachable areas only (cf. Invariant (S23) in Section 4.2) it would be impossible to extract $PTR(target(X))$ (and hence $PTR^*(target(X))$) from $LOC(X)$ for each branch X of an unreachable live area. Consequently, the first for-loop would have to be restricted to the reachable areas $A \in (S \setminus Q) \cup \{L\}$, requiring the tracing of all reachable areas in S, as in the marking phase of *COLLECT GARBAGE*. The tracing of the dead areas in S is considerably simpler and does not require an expensive marking phase.

7.2.2.6. Calculating $PTR^*(A)$

Now that we have arranged the operations in the proper order, we can start improving the algorithm by adding abstract variables to the algorithm. Not having a marking problem, we can immediately turn to the problem of efficiently calculating the $PTR^*(A)$ $(A \in (S \cup H) \setminus Q)$. Again it is easy to see that the calculation of these values in the moving phase can be accomplished in an overall $O(n)$ time (where $n = \#(S \setminus Q)$). As to the calculation of the $PTR^*(A)$ in the updating phase, we can first of all observe that $PTR^*(A) = PTR(A)$ for each $A \in H$. Using Invariant (I15) we can therefore rewrite the updating phase of $COMPACT_5$ as follows:

```
For each A ∈ (S \ Q) ∪ {L}
and each X ∈ branches(A)
| Let B = target(X).
| If B ∉ H
| | cont(LOC(X)) := PTR*(B).
```

Thus the problem of calculating the $PTR^*(A)$ is reduced to the calculation of the $PTR^*(B)$ for $B \in S \setminus Q$ in the updating phase.

As discussed in Subsection 7.2.1.6 there are basically two ways to calculate efficiently the $PTR^*(B)$ in the updating phase. The first is to construct a mapping F which maps each $A \in S \setminus Q$ to $PTR^*(A)$. This makes the updating phase fall apart into the following two loops:

```
F := ∅.
For each A ∈ S \ Q
| F(A) := PTR*(A).
For each A ∈ (S \ Q) ∪ {L}
and each X ∈ branches(A)
| Let B = target(X).
| If B ∉ H
| | cont(LOC(X)) := F(B).
```

The second way is to construct a mapping B which maps each $A \in S \setminus Q$ to the set of all branches X with $target(X) = A$. This also makes the updating phase fall apart into two loops:

```
B := {(A,∅) | A ∈ S \ Q}.
For each A ∈ (S \ Q) ∪ {L}
and each X ∈ branches(A)
| Let B = target(X).
| If B ∉ H
| | B(B) := B(B) ∪ {X}.
For each A ∈ S \ Q
| Let a = PTR*(A).
| For each X ∈ B(A)
| | cont(LOC(X)) := a.
```

In *COLLECT GARBAGE*$_6$ the construction loop of B was disposed of by combining it with the marking phase. The absence of a marking phase in *COMPACT*$_5$ makes this impossible now. Consequently, the reason why the B-solution was preferred over the F-solution in Subsection 7.2.1.6 is not valid here. Are there any other reasons to prefer either of the two solutions?

A point against the F-solution seems to be that it requires a space overhead, while we know that the B-solution does not require any space overhead at all. Upon closer scrutiny, however, the F-solution does not introduce a space overhead either: The space required for the implementation of the set T in *COLLECT GARBAGE*$_7$ is vacant in *COMPACT*$_5$ and can be used for the implementation of F. Moreover, we know (see Chapter 5) that in both the F- and B-solution we can reduce the number of scans of the store from three to two by dividing the actual updating scan over the "construction scan" and the moving scan. The latter has the disadvantage that the moving scan becomes more complicated and cannot be optimized very well, as would be possible with a separate moving scan (using a "block traversal and moving" technique, see Section 7.3). If possible, the number of scans should therefore be reduced to two by combining the construction and the updating scan, leaving the moving scan unaffected.

It is not difficult to see that an efficient combination of the scans as suggested above is impossible if we stick to either the F- or B-solution. But what if we use *both* solutions? Is it possible to have the best of both worlds? Indeed it is surprising that it is. In order to show this let $A \in S \setminus Q$ and let X be a branch with $target(X) = A$, which is involved in the updating process. The best of the F-world is that if X is visited *after* A then $cont(LOC(X))$ can be updated immediately since $F(A)$ has been assigned the value $PTR^*(A)$. The best of the B-world is that if X is visited *before* A we can make sure that $cont(LOC(X))$ is updated (when A is visited) by putting X in $B(A)$. Consequently, by using F to update $cont(LOC(X))$ for branches X which are visited after their targets, and by using B to update $cont(LOC(X))$ for branches X which are visited before their targets, we can update all pointers in a single scan. Only we must be careful with the root L, for which we defined neither $F(L)$ nor $B(L)$ (since $L \in H$). The latter case is therefore treated separately in the following version of *COMPACT*:

```
COMPACT_6:
   Variables:
      B: mapping from areas to sets of branches,
      F: mapping from areas to integers.
   Action:
      Let Q = {A ∈ S | status(A) = dead}.
      Let A_1,...,A_m be such that S = <A_1,...,A_m>.
      B,F := {(A,∅) | A ∈ S \ Q},∅.
      For each X ∈ branches(L)
      | Let B = target(X).
      | If B ∉ H
      | | B(B) := B(B) ∪ {X}.
      For each A ∈ S \ Q
      | Let a = PTR*(A).
      | While B(A) ≠ ∅
      | | Get X from B(A).
      | | cont(LOC(X)) := a.
      | F(A) := a.
      | For each X ∈ branches(A)
      | | Let B = target(X).
      | | If B ∉ H
      | | | If B ∈ dom(F)
      | | | | cont(LOC(X)) := F(B).
      | | | else
      | | | | B(B) := B(B) ∪ {X}.
      For k = 1 to m
      | If A_k ∉ Q
      | | MOVE_L(A_k,PTR*(A_k)).
      S := S \ Q.
```

Notice that we use the test "$B \in dom(F)$" to determine whether B has been visited or not and that we have taken care that $B(A) = \emptyset$ for each $A \in S \setminus Q$ at the end of the algorithm. The correctness of the rewritten updating phase (which is far from obvious) can be proved by using the following invariants of the second for-loop and the fact that at the end of the loop $dom(F) = S \setminus Q$:

```
dom(F) ⊂ S \ Q.

For each A ∈ dom(F)
| For each X ∈ P(A)
| | cont(LOC(X)) = PTR*(A).
| B(A) = ∅.
| F(A) = PTR*(A).

For each A ∈ (S \ Q) \ dom(F)
| B(A) = P(A).
```

where, for each area A:

$$P(A) = \{X \in branches(B) \mid B \in dom(F) \cup \{L\},\ target(X) = A\}.$$

7.2.2.7. Removing PTR^* from the algorithm

The calculation of the $PTR^*(A)$ in the updating phase can be accomplished in an overall $O(n)$ time by visiting the areas in $S \setminus Q$ in the (obvious) order from left to right and keeping track of a counter. The same holds for the moving phase, which makes the removal of PTR^* from the algorithm extremely simple. The fact that the areas in $S \setminus Q$ are visited in order from left to right enables us to replace the test "$B \in dom(F)$" by "$PTR(B) \leq PTR(A)$". If we also "remove" Q from any but the first and last statement of the algorithm, this leads to:

```
COMPACT7:
   Variables:
      B: mapping from areas to sets of branches,
      F: mapping from areas to integers,
      a: integer.
   Action:
      Let Q = {A ∈ S | status(A) = dead}.
      Let A1,...,Am be such that S = <A1,...,Am>.
      B,F := {(A,∅) | A ∈ S, status(A) = alive},∅.
      For each X ∈ branches(L)
      | Let B = target(X).
      | If B ∉ H
      | | B(B) := B(B) ∪ {X}.
      a := 0.
      For k = 1 to m
      | If status(Ak) = alive
      | | While B(Ak) ≠ ∅
      | | | Get X from B(Ak).
      | | | cont(LOC(X)) := a.
      | | F(Ak) := a.
      | | For each X ∈ branches(Ak)
      | | | Let B = target(X).
      | | | If B ∉ H
      | | | | If PTR(B) ≤ PTR(Ak)
      | | | | | cont(LOC(X)) := F(B).
      | | | | else
      | | | | | B(B) := B(B) ∪ {X}.
      | | a := a + size(Ak).
      a := 0.
      For k = 1 to m
      | If status(Ak) = alive
      | | MOVEℓ(Ak,a).
      | | a := a + size(Ak).
      S := S \ Q.
```

Check that, assuming that each "primitive" operation takes $O(1)$ time, the algorithm operates in $O(m+n)$ time, where:

$$m = \#S,$$
$$n = \Sigma\, A \in S \cup \{L\},\ status(A) = alive\ [size(A)].$$

Here "S" and "H" denote the initial values of S and H.

The design of the algorithm will be concluded here. The further transformation ("implementation") of the algorithm will be described in Subsection 7.3.2.

7.3. IMPLEMENTATION

In this section we shall "implement" the two algorithms derived in the previous section, i.e., we shall translate them into code for a very simple von Neumann machine, which will be described using conventional terminology below. The memory of the machine is the store C of our model. Apart from that, the machine has a number of registers, denoted by $a,b,c,\ldots$, which can contain integers and may be used as index registers. The instructions of the machine have three kinds of operands: sources, destinations and labels. A "source" is an integer (denoted by $0,1,2,\ldots$), the contents of a register (denoted by $a,b,c,\ldots$) or the contents of a register-addressed memory cell (denoted by $[a],[b],[c],\ldots$). A "destination" is a register (denoted by $a,b,c,\ldots$) or a register-addressed memory cell (denoted by $[a],[b],[c],\ldots$). A "label" is either a subroutine label (denoted by a name in capital letters), which uniquely identifies a subroutine address in the code, or a branch label (denoted by $L1,L2,L3,\ldots$), which uniquely identifies a branch address in the code. The instructions and their meaning are described below, where s, d, P and L denote a source, destination, subroutine label and branch label, respectively.

COPY,s,d:	$d := s$.
ADD,s,d:	$d := d + s$.
SUB,s,d:	$d := d - s$.
PROC,P:	Defines subroutine label P.
CALL,P:	Subroutine call of P.
RETURN:	Return from subroutine.
LABEL,L:	Defines branch label L.
GOTO,L:	Branch to L.
IFα,s_1,s_2,L:	If the relation α holds between s_1 and s_2, then branch to L, where α = *GT*,*GE*,*EQ*,*NE*,*LE*,*LT*, meaning $>, \geq, =, \neq, \leq, <$, respectively.

The indexed addressing mode $([a],[b],[c],\ldots)$ is allowed with the *COPY* instruction only.

The remarks made in the beginning of Section 7.2 apply more or less to this section as well. The section will be divided into two subsections dealing with the implementation of *COLLECT GARBAGE*$_7$ and *COMPACT*$_7$ on the above machine respectively.

7.3.1. *COLLECT GARBAGE*

7.3.1.1. Optimizing the moving of areas

In Subsection 7.2.1.7 we discussed the fact that in the moving phase of *COLLECT GARBAGE*$_7$ the areas in H are visited in the "difficult" order (from right to left). Preparations should therefore be made in the updating phase to make the traversal of H in the moving phase easier. These preparations, which can be extended to S, can be exploited to make the entire moving phase much more efficient than it would be in a straightforward implementation of *COLLECT GARBAGE*$_7$. First of all, it is reasonable to assume that after some time during the execution of a program a certain "residuum" of reachable areas will originate at the bottom of S and H. Areas in this residuum need not be moved. The size of the residuum in S and H can be determined cheaply in the updating phase. Secondly, contiguous marked areas and contiguous unmarked areas constitute chunks in

S and *H*, which will be called "blocks" and "gaps" respectively. Instead of moving one marked area at a time and skipping one unmarked area at a time it would greatly increase efficiency if areas are moved "blockwise" and skipped "gapwise" in the moving phase. Arrangements to this end can also be made cheaply in the updating phase.

The technique which will be used to implement the above optimizations is more or less standard: In the updating phase information is stored in the cells of a gap (which are all garbage) concerning the size of the gap and the size of the (possibly empty) block following the gap. In particular, we shall use the "first" cell of a gap to store the address of the "last" cell of the gap and we shall use the last cell of the gap to store the size of the block immediately following the gap, as indicated in Figure 7.2 (in which the fields of marked areas are shaded).

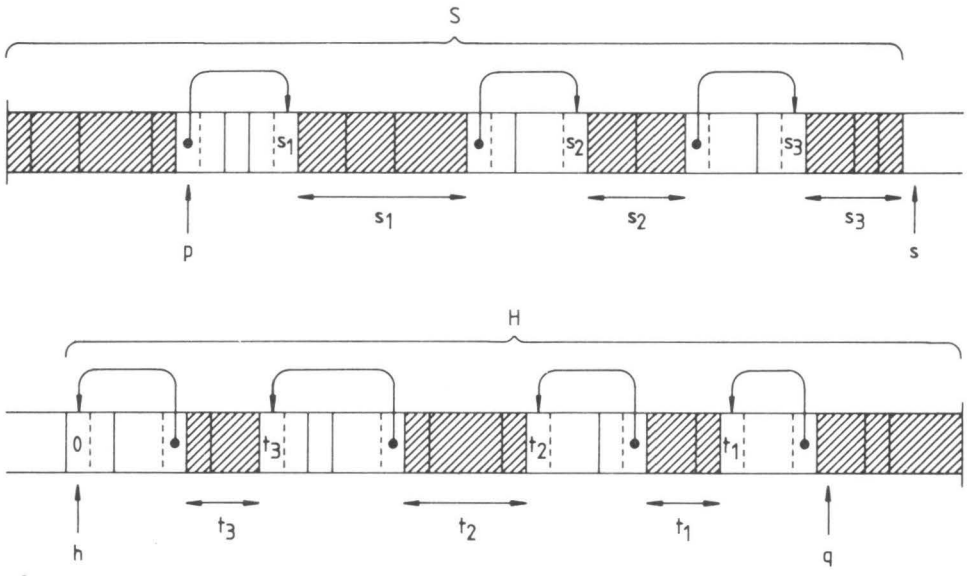


Figure 7.2

The above technique works well only if gaps of size *1* cannot occur. A sufficient condition for the absence of gaps of size *1* is the absence of areas of size *1*. In view of the overhead to be contained in the fields of areas (such as type information) the latter is a reasonable assumption. Although we could make things work for gaps of size *1* as well, we shall turn the latter from an assumption into a fact by adding the following invariant to the model:

(I16) For each area *A*
| *size(A)* ≠ *1*.

The rewriting of the updating and moving phase to employ the technique sketched above is simple. The rewritten updating and moving phase are

incorporated in the eighth version of *COLLECT GARBAGE* presented below. In this algorithm the variables p and q are used to record the address of the "top" of the residuum in S and H respectively (see Figure 7.2). The text between the double braces "{{" and "}}" is comment.

```
COLLECT GARBAGE8:
   Variables:
      M: set of areas,
      T: set of areas,
      B: mapping from areas to sets of branches,
      a,b,c,d,k,p,q,r: integer.
   Action:
      Let Q = {A ∈ S ∪ H | A is unreachable}.
      Let s = Σ A ∈ S [size(A)].
      Let h = N - Σ A ∈ H [size(A)].
   {{Mark all reachable areas and construct the branch sets.}}
      M,T,B,r := {L},{L},{(A,∅) | A ∈ S ∪ H},size(L).
      While T ≠ ∅
      | Get A from T.
      | For each X ∈ branches(A)
      | | Let B = target(X).
      | | If B ∉ M
      | | | M,T,r := M ∪ {B},T ∪ {B},r + size(B).
      | | B(B) := B(B) ∪ {X}.
   {{Update all pointers to areas in S and prepare to move.}}
      a,b,d := 0,0,s.
      PROCESS BLOCK.
      p := b.
      While b ≠ s
      | c := b.
      | PROCESS GAP.
      | cont(cell(c)) := b - 1.
      | c := b.
      | PROCESS BLOCK.
      | cont(cell(c - 1)) := b - c.
   {{Update all pointers to areas in H and prepare to move.}}
      a,b,c,d := N - (r - a),h,h,N.
      PROCESS BLOCK.
      While b ≠ N
      | cont(cell(b)) := b - c.
      | c := b.
      | PROCESS GAP.
      | cont(cell(b - 1)) := c.
      | c := b.
      | PROCESS BLOCK.
      q := c.
   {{Move the areas in S.}}
      a,b := p,p.
      While b ≠ s
      | b := cont(cell(b)).
      | k := cont(cell(b)).
      | b := b + 1.
      | While k > 0
      | | cont(cell(a)) := cont(cell(b)).
      | | a,b := a + 1,b + 1.
      | | k := k - 1.
```

```
{{Move the areas in H.}}
   a,b := q,q.
   While b ≠ h
   | b := b - 1.
   | b := cont(cell(b)).
   | k := cont(cell(b)).
   | While k > 0
   | | a,b := a - 1,b - 1.
   | | cont(cell(a)) := cont(cell(b)).
   | | k := k - 1.
{{Remove the unreachable areas.}}
   S,H := S \ Q,H \ Q.

PROCESS BLOCK:
   While b ≠ d and AREA(b) ∈ M
   | Let A = AREA(b).
   | While B(A) ≠ ∅
   | | Get X from B(A).
   | | cont(LOC(X)) := a.
   | k := size(A).
   | a,b := a + k,b + k.

PROCESS GAP:
   While b ≠ d and AREA(b) ∉ M
   | Let A = AREA(b).
   | k := size(A).
   | b := b + k.

AREA(b):
   The area A ∈ S ∪ H with PTR(A) = b.
```

7.3.1.2. Removing the abstract concepts from the algorithm

We shall now remove the abstract variables and other abstract concepts from *COLLECT GARBAGE*$_8$, thus making the algorithm suitable for a direct translation into machine code. As we shall see, the model will have to be extended by a number of invariants concerning the representation of the abstract concepts in the store. A first observation is concerned with what we called the "type information" and the "status information" in Chapter 5 (represented by the *branches* of an area and the variable T respectively). For the representation of these two kinds of information space must be reserved in the store. We shall accomplish this by reserving a free cell for either kind of information in the field of an area. We shall assume that the offsets of these cells are the same for all areas. The offsets will be denoted by *type* (for the cell containing the type information) and *link* (for the cell containing the status information). The offsets are "defined" by means of the following additional invariant:

```
(I17) There are integers, denoted by type and link, such that
      | type ≠ link.
      | For each area A
      | | 0 ≤ type,link < size(A).
      | | type,link ∉ {offset(X) | X ∈ branches(A)}.
```

The cell $\underline{TYP(A)}$ for areas $A \in S \cup H$ is defined by:

$$TYP(A) = cell(PTR(A) + type).$$

The cell $\underline{LNK(A)}$ for areas $A \in S \cup H$ is defined by:

$$LNK(A) = cell(PTR(A) + link).$$

Notice that Invariant (I17) makes Invariant (I16) redundant.

First consider the type information. For a given area A, this information consists of the set of all offsets of branches of A. It will be represented by a pointer, stored in the cell $TYP(A)$, which points to a "template". This template is a piece of storage, containing the information concerning the offsets of branches of A. There are many ways to represent these templates. Which way of representation is appropriate depends to a great extent on implementation details. Since this section is only meant as an illustration of how a garbage collector can be implemented from an abstract description such as given by *COLLECT GARBAGE*$_7$, we shall choose a simple, yet reasonably general approach. The implementation of the garbage collector on the MIAM, as a matter of fact, would require a more complicated representation of the templates (due to the occurrence of areas with a "dynamic" type).

The representation of the templates is pictured in Figure 7.3.

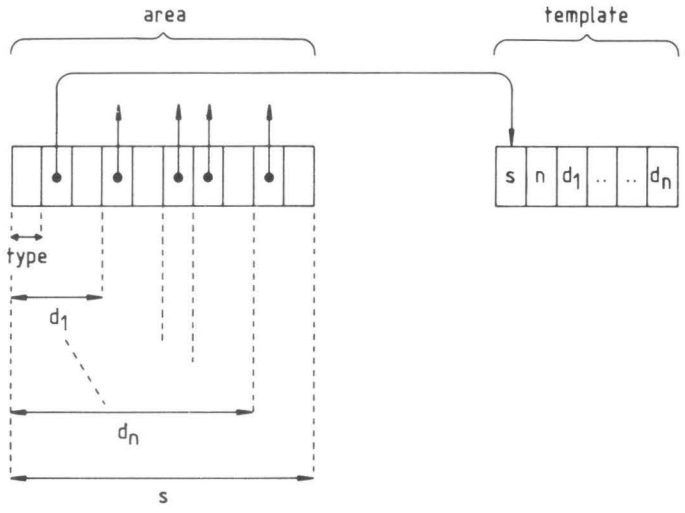


Figure 7.3

As can be seen, the first cell of a template associated with an area A contains the size of A. Strictly speaking the size information does not belong to the type information, but containing the size information in the templates saves us the trouble of reserving yet another free cell in the field of an area. The latter, of course, pays off only if there is a many-to-one correspondence between areas and templates. The second cell of the

template contains the number of branches of A. It is immediately followed by cells containing the offsets of these branches (in arbitrary order). This is all more formally described in the following invariant:

(I18) For each $A \in S \cup H$
  Let $n = \#branches(A)$.
  Let $t = cont(TYP(A))$.
  $0 \leq t \leq N - n - 2$.
  $cont(cell(t)) = size(A)$.
  $cont(cell(t+1)) = n$.
  $\{cont(cell(t+2+k)) \mid 0 \leq k < n\} =$
    $\{offset(X) \mid X \in branches(A)\}$.

The obvious place to store the templates is in a free part of the field of the root L, thus making sure that the templates are not moved and, consequently, that the contents of $TYP(A)$ need not be updated in a garbage collection. Apart from not overlapping the cells occupied by atoms of L, this free part of the field of L should not overlap $TYP(L)$ and $LNK(L)$ either, as expressed in the invariant below:

(I19) For each $A \in S \cup H$
  Let $n = \#branches(A)$.
  Let $t = cont(TYP(A))$.
  $PTR(L) \leq t$.
  Let $T = \{cell(a) \mid t \leq a < t+n+2\}$.
  $T \cap \{LOC(X) \mid X \in atoms(A)\} = \emptyset$.
  $TYP(L), LNK(L) \notin T$.

Notice that Invariants (I18) and (I19) are not violated by $COLLECT\ GARBAGE_8$ (though, without Invariant (I19), Invariant (I18) would have been).

The above invariants allow us to remove the abstract *branches* and *size* of an area from the algorithm (using a function *BRANCH*, which, analogous to the function *AREA* in $COLLECT\ GARBAGE_8$, maps an address b to the branch X with $addr(LOC(X)) = b$). The abstract *target* of a branch can also be removed (through Invariant (I15)). What remains are the abstract variables M, T and B. Let us first of all consider T, for the implementation of which we have already reserved a free cell $LNK(A)$ in the field of each area A. This free cell will be used to implement T as a linked list, where the (new) variable t acts as a pointer to the head of the list. The only "problem" is how to indicate the end of the list. The solution to this problem will be deferred temporarily.

Next, consider the variable B. For each area $A \in S \cup H$ the set $B(A)$ will be implemented in the standard way as a linked list, where the cell $TYP(A)$ acts as a list head (see Figure 5.10). A requirement for the use of this implementation trick is that we can efficiently distinguish the original contents of $TYP(A)$ (= a pointer to a template) from the address of a cell occupied by a branch. The simplest way to satisfy this requirement is to demand that all templates are stored in cells to the right of cells occupied by branches of L, say in the cells $cell(M)$ through $cell(N-1)$ (see Figure 7.4).

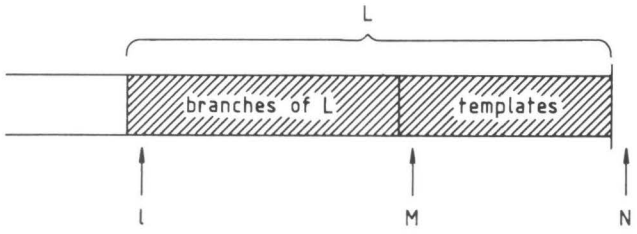


Figure 7.4

The test of whether a given address a is a pointer to a template or the address of a "branch cell" then reduces to "$a \geq M$". The following invariant and definition authenticate this trick:

(I20) There is an integer, denoted by M, such that
  For each $X \in branches(L)$
  | $addr(LOC(X)) < M$.
  For each $A \in S \cup H$
  | $cont(TYP(A)) \geq M$.

The implementation of $\mathcal{B}$ does not introduce any space overhead at all. There is a small time penalty, however: When visiting an area A with $\mathcal{B}(A) \neq \emptyset$ in the marking phase, we have to traverse the linked list emanating from $TYP(A)$ in order to find the pointer to the template of A (which we need to find the branches of A). Check that this time overhead is $O(n)$, where n is the total number of branches of reachable areas.

Finally, consider the variable M, representing the "marking information". There are (at least) two ways to implement M without any space overhead at all. The first (cf. Algorithm GNK.DTEH* in Chapter 5) is to initialize each $LNK(A)$ for $A \in S \cup H$ before the marking phase to some value $unmarked$ outside the address range $0,\ldots,N-1$. Since marking an area A is always accompanied by adding A to T and since the latter implies that the contents of $LNK(A)$ are changed into a value inside the address range, the test "$A \in M$" can be replaced by "$cont(LNK(A)) \neq unmarked$" (assuming that $cont(LNK(A))$ is not changed when A is removed from T). The second way is to use the following invariant which holds during the marking phase:

For each $A \in S \cup H$
| $A \in M \Leftrightarrow \mathcal{B}(A) \neq \emptyset \vee A = L$.

This invariant, together with the chosen implementation for $\mathcal{B}$, implies that the test "$A \in M$" can be replaced by "$cont(TYP(A)) < M$ or $PTR(A) = PTR(L)$". Since the second method does not require any initialization (which cannot be avoided in the first implementation of M since $LNK(A)$ is also used for the implementation of the variable F in $COMPACT_7$, see Subsection 7.3.2) we shall choose this implementation of M.

The fact that we have to perform the test "$PTR(A) = PTR(L)$" each time we test that $A \in M$, is a nuisance. We can dispose of this test by initializing $\mathcal{B}(L)$ to $\{B\}$ at the beginning of the marking phase, where B is a "dummy branch" with $target(B) = L$. A proper candidate for B (or better,

LOC(B)) is the cell *LNK(L)*. The only function of this cell in the implementation of T is, that it should contain the end of list indicator. By pretending that *LNK(L)* contains a pointer to L and initializing $B(L)$ as indicated, *LNK(L)* will contain (and continue to contain during the entire marking phase) a pointer u to the template of L. This pointer makes a fine end of list indicator, since it can be distinguished from a pointer to an area $A \in S \cup H$ by the test "$u \geq M$". That is, provided we add the following invariant:

(I21) $PTR(L) < M$.

Thus the list termination "problem" for the implementation of T is also solved (i.e., the test "$T \neq \emptyset$" can be implemented as "$t < M$").

The above describes informally how *COLLECT GARBAGE*$_8$ can be stripped of abstract concepts. A systematic, stepwise translation of *COLLECT GARBAGE*$_8$ according to the implementations sketched above, including a proof of correctness of the implementations, is a tedious, though not really difficult process (to which the method described in Chapter 3 is ideally suited). Only the final result of this process is presented below. The reader is invited to check the correctness of the translation given. In particular, it should be checked that none of the new invariants is violated.

```
COLLECT GARBAGE9:
   Variables:
      a,b,c,d,k,p,q,r,t,u,v: integer.
   Action:
      Let Q = {A ∈ S ∪ H | A is unreachable}.
      Let s = Σ A ∈ S [size(A)].
      Let h = N - Σ A ∈ H [size(A)].
      Let ℓ = N - size(L).
   {{Mark all reachable areas and construct the branch sets.}}
      u := cont(cell(ℓ + type)).
      cont(cell(ℓ + link)) := u.
      cont(cell(ℓ + type)) := ℓ + link.
      t := ℓ.
      r := cont(cell(u)).
      While t < M
      | a := t.
      | t := cont(cell(t + link)).
      | u := cont(cell(a + type)).
      | While u < M
      | | u := cont(cell(u)).
      | k := DEGREE(u).
      | While k > 0
      | | b := a + OFFSET(u,k).
      | | c := cont(cell(b)).
      | | v := cont(cell(c + type)).
      | | If v ≥ M
      | | | cont(cell(c + link)) := t.
      | | | t := c.
      | | | r := r + SIZE(c).
      | | cont(cell(b)) := v.
      | | cont(cell(c + type)) := b.
      | | k := k - 1.
```

```
{{Update all pointers to areas in S and prepare to move.}}
   a := 0.
   b := 0.
   d := s.
   PROCESS BLOCK.
   p := b.
   While b ≠ s
   | c := b.
   | PROCESS GAP.
   | cont(cell(c)) := b - 1.
   | c := b.
   | PROCESS BLOCK.
   | cont(cell(c - 1)) := b - c.
{{Update all pointers to areas in H and prepare to move.}}
   a := N - (r - a).
   b := h.
   c := h.
   d := N.
   PROCESS BLOCK.
   While b ≠ N
   | cont(cell(b)) := b - c.
   | c := b.
   | PROCESS GAP.
   | cont(cell(b - 1)) := c.
   | c := b.
   | PROCESS BLOCK.
   q := c.
{{Move the areas in S.}}
   a := p.
   b := p.
   While b ≠ s
   | b := cont(cell(b)).
   | k := cont(cell(b)).
   | b := b + 1.
   | While k > 0
   | | cont(cell(a)) := cont(cell(b)).
   | | a := a + 1.
   | | b := b + 1.
   | | k := k - 1.
{{Move the areas in H.}}
   a := q.
   b := q.
   While b ≠ h
   | b := b - 1.
   | b := cont(cell(b)).
   | k := cont(cell(b)).
   | While k > 0
   | | a := a - 1.
   | | b := b - 1.
   | | cont(cell(a)) := cont(cell(b)).
   | | k := k - 1.
{{Remove the unreachable areas.}}
   S,H := S \ Q,H \ Q.
```

```
PROCESS BLOCK:
   While b ≠ d and cont(cell(b + type)) < M
   | u := cont(cell(b + type)).
   | While u < M
   | | v := cont(cell(u)).
   | | cont(cell(u)) := a.
   | | u := v.
   | cont(cell(b + type)) := u.
   | k := SIZE(b).
   | a := a + k.
   | b := b + k.

PROCESS GAP:
   While b ≠ d and cont(cell(b + type)) ≥ M
   | k := SIZE(b).
   | b := b + k.

SIZE(b):
   cont(cell(cont(cell(b + type)))).

DEGREE(u):
   cont(cell(u + 1)).

OFFSET(u,k):
   cont(cell(u + k + 1)).
```

7.3.1.3. The final translation

The translation of *COLLECT GARBAGE*$_9$ into code for the machine described in the beginning of Section 7.3 is straightforward. In the translation given below we have chosen specific values for *type* and *link*, as described by:

```
(I22) type = 1, link = 0.
```

The registers of the machine which are used are a, b, c, d, h, k, ℓ, m, n, p, q, r, s, t, u, v and w. These registers correspond to the variables of *COLLECT GARBAGE*$_9$, except for w, which is used as a general purpose working register, and m, n, s, h and ℓ, which should initially contain the values M, N, s, h and ℓ from *COLLECT GARBAGE*$_9$, respectively. This is indicated in the algorithm below by initialization statements for the registers m, n, s, h and ℓ. At the end of the algorithm, s and h will contain a pointer to the new top of S and H respectively, while m, n and ℓ are not affected. For reasons of clarity the statements have been rearranged somewhat. (The statement "$S,H := S \setminus Q, H \setminus Q$" has returned to its old place.)

```
COLLECT GARBAGE10:
   Registers:
      a,b,c,d,h,k,ℓ,m,n,p,q,r,s,t,u,v,w.
   Action:
      m := M.
      n := N.
      s := Σ A ∈ S [size(A)].
      h := N - Σ A ∈ H [size(A)].
      ℓ := N - size(L).
      Let Q = {A ∈ S∪H | A is unreachable}.
```

```
    S,H := S \ Q,H \ Q.
    CALL,COLG.

PROC,COLG
    CALL,MARK
    CALL,UPDS
    CALL,UPDH
    CALL,MOVS
    CALL,MOVH
    RETURN

PROC,MARK
    COPY,ℓ,w
    ADD,1,w
    COPY,[w],u
    COPY,u,[ℓ]
    COPY,ℓ,[w]
    COPY,ℓ,t
    COPY,[u],r
LABEL,L1
    IFGE,t,m,L6
    COPY,t,a
    COPY,[t],t
    COPY,a,w
    ADD,1,w
    COPY,[w],u
LABEL,L2
    IFGE,u,m,L3
    COPY,[u],u
    GOTO,L2
LABEL,L3
    ADD,1,u
    COPY,[u],k
LABEL,L4
    IFLE,k,0,L1
    ADD,1,u
    COPY,[u],b
    ADD,a,b
    COPY,[b],c
    COPY,c,w
    ADD,1,w
    COPY,[w],v
    IFLT,v,m,L5
    COPY,t,[c]
    COPY,c,t
    COPY,[v],w
    ADD,w,r
LABEL,L5
    COPY,v,[b]
    COPY,c,w
    ADD,1,w
    COPY,b,[w]
    SUB,1,k
    GOTO,L4
LABEL,L6
    RETURN
```

```
PROC,UPDS
   COPY,0,a
   COPY,0,b
   COPY,s,d
   CALL,BLKG
   COPY,b,p
LABEL,L7
   IFEQ,b,s,L8
   COPY,b,c
   CALL,GAPG
   COPY,b,w
   SUB,1,w
   COPY,w,[c]
   COPY,b,c
   CALL,BLKG
   COPY,b,w
   SUB,c,w
   SUB,1,c
   COPY,w,[c]
   GOTO,L7
LABEL,L8
   RETURN

PROC,UPDH
   COPY,r,w
   SUB,a,w
   COPY,n,a
   SUB,w,a
   COPY,h,b
   COPY,h,c
   COPY,n,d
   CALL,BLKG
LABEL,L9
   IFEQ,b,n,L10
   COPY,b,w
   SUB,c,w
   COPY,w,[b]
   COPY,b,c
   CALL,GAPG
   COPY,b,w
   SUB,1,w
   COPY,c,[w]
   COPY,b,c
   CALL,BLKG
   GOTO,L9
LABEL,L10
   COPY,c,q
   RETURN
```

```
PROC,BLKG
LABEL,L11
   IFEQ,b,d,L13
   COPY,b,w
   ADD,1,w
   COPY,[w],u
   IFGE,u,m,L13
LABEL,L12
   COPY,[u],v
   COPY,a,[u]
   COPY,v,u
   IFLT,u,m,L12
   COPY,b,w
   ADD,1,w
   COPY,u,[w]
   COPY,[u],k
   ADD,k,a
   ADD,k,b
   GOTO,L11
LABEL,L13
   RETURN

PROC,GAPG
LABEL,L14
   IFEQ,b,d,L15
   COPY,b,w
   ADD,1,w
   COPY,[w],u
   IFLT,u,m,L15
   COPY,[u],k
   ADD,k,b
   GOTO,L14
LABEL,L15
   RETURN

PROC,MOVS
   COPY,p,a
   COPY,p,b
LABEL,L16
   IFEQ,b,s,L18
   COPY,[b],b
   COPY,[b],k
   ADD,1,b
LABEL,L17
   IFLE,k,0,L16
   COPY,[b],[a]
   ADD,1,a
   ADD,1,b
   SUB,1,k
   GOTO,L17
LABEL,L18
   COPY,a,s
   RETURN
```

```
PROC,MOVH
   COPY,q,a
   COPY,q,b
LABEL,L19
   IFEQ,b,h,L21
   SUB,1,b
   COPY,[b],b
   COPY,[b],k
LABEL,L20
   IFLE,k,0,L19
   SUB,1,a
   SUB,1,b
   COPY,[b],[a]
   SUB,1,k
   GOTO,L20
LABEL,L21
   COPY,a,h
   RETURN
```

7.3.2. *COMPACT*

7.3.2.1. Optimizing the moving of areas

As we did for *COLLECT GARBAGE*$_7$, we shall first of all optimize the moving phase of *COMPACT*$_7$. In contrast to the moving phase of *COLLECT GARBAGE*$_7$, only the (live) areas in S are moved in *COMPACT*$_7$. The technique which will be used to optimize the moving of the areas in S is the same as described in Subsection 7.3.1.1. This implies that we use the updating phase to determine the "residuum" in S (recorded in the variable p) and to organize S in "gaps" and "blocks", as indicated in Figure 7.2. The blockwise moving of the areas in S is then entirely the same as in *COLLECT GARBAGE*$_8$. This is all described in *COMPACT*$_8$, where, of course, we assume that Invariant (I16) is valid.

```
COMPACT8:
   Variables:
      B: mapping from areas to sets of branches,
      F: mapping from areas to integers,
      a: integer.
   Action:
      Let Q = {A ∈ S | status(A) = dead}.
      Let s = Σ A ∈ S [size(A)].
   {{Process all pointers from L to areas in S.}}
      B,F := {(A,∅) | A ∈ S, status(A) = alive},∅.
      For each X ∈ branches(L)
      | Let B = target(X).
      | If B ∉ H
      | | B(B) := B(B) ∪ {X}.
```

```
{{Update all pointers to areas in S and prepare to move.}}
   a,b := 0,0.
   PROCESS BLOCK.
   p := b.
   While b ≠ s
   | c := b.
   | PROCESS GAP.
   | cont(cell(c)) := b - 1.
   | c := b.
   | PROCESS BLOCK.
   | cont(cell(c - 1)) := b - c.
{{Move the areas in S.}}
   a,b := p,p.
   While b ≠ s
   | b := cont(cell(b)).
   | k := cont(cell(b)).
   | b := b + 1.
   | While k > 0
   | | cont(cell(a)) := cont(cell(b)).
   | | a,b := a + 1,b + 1.
   | | k := k - 1.
{{Remove the dead areas.}}
   S := S \ Q.

PROCESS BLOCK:
   While b ≠ s and status(AREA(b)) = alive
   | Let A = AREA(b).
   | While B(A) ≠ ∅
   | | Get X from B(A).
   | | cont(LOC(X)) := a.
   | F(A) := a.
   | For each X ∈ branches(A)
   | | Let B = target(X).
   | | If B ∉ H
   | | | If PTR(B) ≤ PTR(A)
   | | | | cont(LOC(X)) := F(B).
   | | | else
   | | | | B(B) := B(B) ∪ {X}.
   | k := size(A).
   | a,b := a + k,b + k.

PROCESS GAP:
   While b ≠ s and status(AREA(b)) = dead
   | Let A = AREA(b).
   | k := size(A).
   | b := b + k.

AREA(b):
   The area A ∈ S ∪ H with PTR(A) = b.
```

7.3.2.2. Removing the abstract concepts from the algorithm

The removal of the abstract concepts from $COMPACT_8$ is in many respects the same as it was for $COLLECT\ GARBAGE_8$ (see Subsection 7.3.1.2). We shall copy Invariants (I17)-(I21), thus providing simple implementations for the *branches* and *size* of an area. For the variable B the same implementation will be chosen as in $COLLECT\ GARBAGE_9$, though there is no need to introduce a "dummy branch" now. The only new problems are the removal of the *status* of an area (in S) and the variable F. The obvious way to implement F is to use the cell $LNK(A)$ for the recording of $F(A)$ for each $A \in S$. The *status* of an area in S will be implemented by the following additional invariant:

```
(I23) For each A ∈ S
      | status(A) = dead ⇔ cont(LNK(A)) = N.
```

Check that this invariant is not violated by $COLLECT\ GARBAGE_9$ (and consequently not by $COLLECT\ GARBAGE_{10}$). Check also that $COMPACT_9$, which is presented below, does not violate any of the additional invariants.

```
COMPACT9:
   Variables:
      a,b,c,d,e,k,p,u,v: integer.
   Action:
      Let Q = {A ∈ S | status(A) = dead}.
      Let s = Σ A ∈ S [size(A)].
      Let h = N - Σ A ∈ H [size(A)].
      Let ℓ = N - size(L).
   {{Process all pointers from L to areas in S.}}
      u := cont(cell(ℓ + type)).
      k := DEGREE(u).
      While k > 0
      | a := ℓ + OFFSET(u,k).
      | b := cont(cell(a)).
      | If b < h
      | | cont(cell(a)) := cont(cell(b + type)).
      | | cont(cell(b + type)) := a.
      | k := k - 1.
   {{Update all pointers to areas in S and prepare to move.}}
      a := 0.
      b := 0.
      PROCESS BLOCK.
      p := b.
      While b ≠ s
      | c := b.
      | PROCESS GAP.
      | cont(cell(c)) := b - 1.
      | c := b.
      | PROCESS BLOCK.
      | cont(cell(c - 1)) := b - c.
```

```
{{Move the areas in S.}}
   a := p.
   b := p.
   While b ≠ s
   | b := cont(cell(b)).
   | k := cont(cell(b)).
   | b := b + 1.
   | While k > 0
   | | cont(cell(a)) := cont(cell(b)).
   | | a := a + 1.
   | | b := b + 1.
   | | k := k - 1.
{{Remove the dead areas.}}
   S := S \ Q.

PROCESS BLOCK:
   While b ≠ s and cont(cell(b + link)) ≠ N
   | u := cont(cell(b + type)).
   | While u < M
   | | v := cont(cell(u)).
   | | cont(cell(u)) := a.
   | | u := v.
   | cont(cell(b + type)) := u.
   | cont(cell(b + link)) := a.
   | k := DEGREE(u).
   | While k > 0
   | | d := b + OFFSET(u,k).
   | | e := cont(cell(d)).
   | | If e < h
   | | | If e ≤ b
   | | | | cont(cell(d)) := cont(cell(e + link)).
   | | | else
   | | | | cont(cell(d)) := cont(cell(e + type)).
   | | | | cont(cell(e + type)) := d.
   | | k := k - 1.
   | k := SIZE(b).
   | a := a + k.
   | b := b + k.

PROCESS GAP:
   While b ≠ s and cont(cell(b + link)) = N
   | k := SIZE(b).
   | b := b + k.

SIZE(b):
   cont(cell(cont(cell(b + type)))).

DEGREE(u):
   cont(cell(u + 1)).

OFFSET(u,k):
   cont(cell(u + k + 1)).
```

7.3.2.3. The final translation

The translation of $COMPACT_9$ into machine code, assuming the validity of Invariant (I22) (besides (I1)-(I21) and (I23)), is again straightforward. The registers of the machine which are used are a, b, c, d, e, h, k, ℓ, m, n, p, s, u, v and w. The registers a, b, c, d, e, k, p, u and v correspond to the variables of $COMPACT_9$. As to the registers w, m, n, s, h and ℓ the same remarks apply as made in Subsection 7.3.1.3. The labels used in the code are chosen in such a way that the code is "compatible" with that of $COLLECT\ GARBAGE_{10}$, thus enabling the integration of $COLLECT\ GARBAGE_{10}$ and $COMPACT_{10}$ (see the next section). Among other things, this implies that the subroutine *MOVS* is identical to the subroutine of the same name used in $COLLECT\ GARBAGE_{10}$. The statement "$S := S \setminus Q$" has been moved to its old place.

```
COMPACT10:
   Registers:
      a,b,c,d,e,h,k,ℓ,m,n,p,s,u,v,w.
   Action:
      m := M.
      n := N.
      s := Σ A ∈ S [size(A)].
      h := N - Σ A ∈ H [size(A)].
      ℓ := N - size(L).
      Let Q = {A ∈ S | status(A) = dead}.
      S := S \ Q.
      CALL,COMP.

   PROC,COMP
      CALL,UPDL
      CALL,UPDC
      CALL,MOVS
      RETURN
```

```
PROC,UPDL
   COPY,ℓ,w
   ADD,1,w
   COPY,[w],u
   ADD,1,u
   COPY,[u],k
LABEL,L22
   IFLE,k,0,L24
   ADD,1,u
   COPY,[u],a
   ADD,ℓ,a
   COPY,[a],b
   IFGE,b,h,L23
   COPY,b,w
   ADD,1,w
   COPY,[w],[a]
   COPY,a,[w]
LABEL,L23
   SUB,1,k
   GOTO,L22
LABEL,L24
   RETURN

PROC,UPDC
   COPY,0,a
   COPY,0,b
   CALL,BLKC
   COPY,b,p
LABEL,L25
   IFEQ,b,s,L26
   COPY,b,c
   CALL,GAPC
   COPY,b,w
   SUB,1,w
   COPY,w,[c]
   COPY,b,c
   CALL,BLKC
   COPY,b,w
   SUB,c,w
   SUB,1,c
   COPY,w,[c]
   GOTO,L25
LABEL,L26
   RETURN
```

```
PROC,BLKC
LABEL,L27
   IFEQ,b,s,L34
   COPY,[b],t
   IFEQ,t,n,L34
   COPY,b,w
   ADD,1,w
   COPY,[w],u
LABEL,L28
   IFGE,u,m,L29
   COPY,[u],v
   COPY,a,[u]
   COPY,v,u
   GOTO,L28
LABEL,L29
   COPY,u,[w]
   COPY,a,[b]
   COPY,u,v
   ADD,1,v
   COPY,[v],k
LABEL,L30
   IFLE,k,0,L33
   ADD,1,v
   COPY,[v],d
   ADD,b,d
   COPY,[d],e
   IFGE,e,h,L32
   IFGT,e,b,L31
   COPY,[e],[d]
   GOTO,L32
LABEL,L31
   COPY,e,w
   ADD,1,w
   COPY,[w],[d]
   COPY,d,[w]
LABEL,L32
   SUB,1,k
   GOTO,L30
LABEL,L33
   COPY,[u],k
   ADD,k,a
   ADD,k,b
   GOTO,L27
LABEL,L34
   RETURN
```

```
PROC,GAPC
LABEL,L35
   IFEQ,b,s,L36
   COPY,[b],t
   IFNE,t,n,L36
   COPY,b,w
   ADD,1,w
   COPY,[w],u
   COPY,[u],k
   ADD,k,b
   GOTO,L35
LABEL,L36
   RETURN

PROC,MOVS
   COPY,p,a
   COPY,p,b
LABEL,L16
   IFEQ,b,s,L18
   COPY,[b],b
   COPY,[b],k
   ADD,1,b
LABEL,L17
   IFLE,k,0,L16
   COPY,[b],[a]
   ADD,1,a
   ADD,1,b
   SUB,1,k
   GOTO,L17
LABEL,L18
   COPY,a,s
   RETURN
```

7.4. CONCLUSION

In this chapter we have described the transformation of the operations *COLLECT GARBAGE* and *COMPACT* from their specification to their final implementation. This transformation process was preceded and accompanied by the construction of a "theory" (the model). The importance of the latter cannot be emphasized enough. First of all, the model allowed us to unambiguously specify the problem in a way entirely independent of Chapter 6. Secondly, due to the fact that the model contained only relevant information, the complexity of the design process could be kept under control relatively easily. Thirdly, all design decisions (such as the representation of the "templates") upon which algorithms were based, had to be recorded in the model. The (final) model therefore provides all information necessary to incorporate the machine code routines for *COLLECT GARBAGE* and *COMPACT* correctly in a given storage management system.

The fourth and most important virtue of the model was, that it ensured that we were always on firm ground: At each stage of the transformation process correctness proofs were possible. The main reason why we omitted them was lack of space. Moreover, the transformations used were often (but not always) so simple that, when proving the correctness of these transformations, we would have felt like mathematicians might feel, having to prove that $1 + 1 = 2$ (or worse, that $111 + 111 = 222$). In relation to this it is worth noting that the first versions of $COLLECT\ GARBAGE_{10}$ and $COMPACT_{10}$ were, in fact, incorrect, which was discovered in a vain attempt to execute them. Upon inspection the errors appeared to have been made in the final transformation phase: one was a clerical error and the other was the result of an unjustified attempt to make a local optimization. This shows that certain transformations (in particular those from $COLLECT\ GARBAGE_9$ to $COLLECT\ GARBAGE_{10}$ and from $COMPACT_9$ to $COMPACT_{10}$) can be performed better by a machine, not because they are difficult, but because the amount of detail involved in them can easily confuse a human being.

It is not difficult to check that, assuming the execution of an instruction takes $O(1)$ time, the machine code routines for *COLLECT GARBAGE* and *COMPACT* operate in $O(m + n)$ time, where m and n are the integers defined at the end of Subsections 7.2.1.7 and 7.2.2.7 (for *COLLECT GARBAGE* and *COMPACT* respectively). This makes the asymptotic behaviour of these routines as good as we can expect. The routines are also believed to be faster than most other linear time garbage collection and compaction routines. The compaction routine used is believed to be novel (see Algorithm CND.BFGU in Chapter 5) and was discovered only in "redoing" the original design (which was based on Algorithm CND.BGUP from Chapter 5) and carefully judging each design decision. The discovery of this compaction algorithm can be credited for the major part to the transformational method we used in the design.

We shall conclude this chapter by giving a full listing of the machine code for *COLLECT GARBAGE* and *COMPACT*. This listing not only marks the end of this chapter and the end of this monograph, it is also a symbol for that which triggered the research reported in this monograph: the complexity of garbage collector design. The listing in itself is meaningless, of course, unless embedded in the appropriate theory.

```
PROC,COLG
CALL,MARK
CALL,UPDS
CALL,UPDH
CALL,MOVS
CALL,MOVH
RETURN
PROC,COMP
CALL,UPDL
CALL,UPDC
CALL,MOVS
RETURN
PROC,MARK
COPY,ℓ,w
ADD,1,w
COPY,[w],u
COPY,u,[ℓ]
COPY,ℓ,[w]
COPY,ℓ,t
COPY,[u],r
LABEL,L1
IFGE,t,m,L6
COPY,t,a
COPY,[t],t
COPY,a,w
ADD,1,w
COPY,[w],u
LABEL,L2
IFGE,u,m,L3
COPY,[u],u
GOTO,L2
LABEL,L3
ADD,1,u
COPY,[u],k
LABEL,L4
IFLE,k,0,L1
ADD,1,u
COPY,[u],b
ADD,a,b
COPY,[b],c
COPY,c,w
ADD,1,w
COPY,[w],v
IFLT,v,m,L5
COPY,t,[c]
COPY,c,t
COPY,[v],w
ADD,w,r
LABEL,L5
COPY,v,[b]
COPY,c,w
ADD,1,w
COPY,b,[w]
SUB,1,k
GOTO,L4
LABEL,L6
RETURN
PROC,UPDS
COPY,0,a
COPY,0,b
COPY,s,d
CALL,BLKG
COPY,b,p
LABEL,L7
IFEQ,b,s,L8
COPY,b,c
CALL,GAPG
COPY,b,w
SUB,1,w
COPY,w,[c]
COPY,b,c
CALL,BLKG
COPY,b,w
SUB,c,w
SUB,1,c
COPY,w,[c]
GOTO,L7
LABEL,L8
RETURN
PROC,UPDH
COPY,r,w
SUB,a,w
COPY,n,a
SUB,w,a
COPY,h,b
COPY,h,c
COPY,n,d
CALL,BLKG
LABEL,L9
IFEQ,b,n,L10
COPY,b,w
SUB,c,w
COPY,w,[b]
COPY,b,c
CALL,GAPG
COPY,b,w
SUB,1,w
COPY,c,[w]
COPY,b,c
CALL,BLKG
GOTO,L9
LABEL,L10
COPY,c,q
RETURN
PROC,BLKG
LABEL,L11
IFEQ,b,d,L13
COPY,b,w
ADD,1,w
COPY,[w],u
IFGE,u,m,L13
LABEL,L12
COPY,[u],v
COPY,a,[u]
COPY,v,u
IFLT,u,m,L12
COPY,b,w
ADD,1,w
COPY,u,[w]
COPY,[u],k
ADD,k,a
ADD,k,b
GOTO,L11
LABEL,L13
RETURN
PROC,GAPG
LABEL,L14
IFEQ,b,d,L15
COPY,b,w
ADD,1,w
COPY,[w],u
IFLT,u,m,L15
COPY,[u],k
ADD,k,b
GOTO,L14
LABEL,L15
RETURN
PROC,MOVS
COPY,p,a
COPY,p,b
LABEL,L16
IFEQ,b,s,L18
COPY,[b],b
COPY,[b],k
ADD,1,b
LABEL,L17
IFLE,k,0,L16
COPY,[b],[a]
ADD,1,a
ADD,1,b
SUB,1,k
GOTO,L17
LABEL,L18
COPY,a,s
RETURN
PROC,MOVH
COPY,q,a
COPY,q,b
LABEL,L19
IFEQ,b,h,L21
SUB,1,b
COPY,[b],b
COPY,[b],k
LABEL,L20
IFLE,k,0,L19
SUB,1,a
SUB,1,b
COPY,[b],[a]
SUB,1,k
GOTO,L20
LABEL,L21
COPY,a,h
RETURN
PROC,UPDL
COPY,ℓ,w
ADD,1,w
COPY,[w],u
ADD,1,u
COPY,[u],k
LABEL,L22
IFLE,k,0,L24
ADD,1,u
COPY,[u],a
ADD,ℓ,a
COPY,[a],b
IFGE,b,h,L23
COPY,b,w
ADD,1,w
COPY,[w],[a]
COPY,a,[w]
LABEL,L23
SUB,1,k
GOTO,L22
LABEL,L24
RETURN
PROC,UPDC
COPY,0,a
COPY,0,b
CALL,BLKC
COPY,b,p
LABEL,L25
IFEQ,b,s,L26
COPY,b,c
CALL,GAPC
COPY,b,w
SUB,1,w
COPY,w,[c]
COPY,b,c
CALL,BLKC
COPY,b,w
SUB,c,w
SUB,1,c
COPY,w,[c]
GOTO,L25
LABEL,L26
RETURN
PROC,BLKC
LABEL,L27
IFEQ,b,s,L34
COPY,[b],t
IFEQ,t,n,L34
COPY,b,w
ADD,1,w
COPY,[w],u
LABEL,L28
IFGE,u,m,L29
COPY,[u],v
COPY,a,[u]
COPY,v,u
GOTO,L28
LABEL,L29
COPY,u,[w]
COPY,a,[b]
COPY,u,v
ADD,1,v
COPY,[v],k
LABEL,L30
IFLE,k,0,L33
ADD,1,v
COPY,[v],d
ADD,b,d
COPY,[d],e
IFGE,e,h,L32
IFGT,e,b,L31
COPY,[e],[d]
GOTO,L32
LABEL,L31
COPY,e,w
ADD,1,w
COPY,[w],[d]
COPY,d,[w]
LABEL,L32
SUB,1,k
GOTO,L30
LABEL,L33
COPY,[u],k
ADD,k,a
ADD,k,b
GOTO,L27
LABEL,L34
RETURN
PROC,GAPC
LABEL,L35
IFEQ,b,s,L36
COPY,[b],t
IFNE,t,n,L36
COPY,b,w
ADD,1,w
COPY,[w],u
COPY,[u],k
ADD,k,b
GOTO,L35
LABEL,L36
RETURN
```

REFERENCES

[ABSTRACTO 79]
Examples for IFIP WG2.1, Meeting, Brussels (1979).

[AHO & ULLMAN 72]
AHO, A.V., & ULLMAN, J.D., The Theory of Parsing, Translation, and Compiling, Volume 1: Parsing, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1972).

[AHO & ULLMAN 77]
AHO, A.V., & ULLMAN, J.D., Principles of Compiler Design, Addison-Wesley Publishing Company, Reading, Massachusetts (1977).

[BACK 80]
BACK, R., Correctness Preserving Program Refinements: Proof Theory and Applications, MC Tract 131, Mathematical Centre, Amsterdam (1980).

[BAKER 78]
BAKER, H.G., Jr., List processing in real time on a serial computer, Communications of the ACM 21 (1978), 280-294.

[BARTH 77]
BARTH, J.M., Shifting garbage collection overhead to compile time, Communications of the ACM 20 (1977), 513-518.

[BAUER & BROY 79]
BAUER, F.L., & BROY, M., (Eds.), Program Construction, International Summer School, Springer-Verlag, Berlin (1979).

[BAUER & EICKEL 74]
BAUER, F.L., & EICKEL, J., (Eds.), Compiler Construction, Springer-Verlag, New York (1974).

[BERKELEY & BOBROW 64]
BERKELEY, E.C., & BOBROW, D.G., (Eds.), The Programming Language LISP: Its Operation and Applications, Information International, Inc., Cambridge, Massachusetts (1964).

[BERZINS 79]
BERZINS, V.A., Abstract Model Specifications for Data Abstractions, Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts (1979).

[BIRKHOFF 67]
BIRKHOFF, G., Lattice Theory, American Mathematical Society Colloquium Publications, Volume XXV, American Mathematical Society, Providence, Rhode Island (1967).

[BIRKHOFF & LIPSON 70]
BIRKHOFF, G., & LIPSON, J.D., Heterogeneous algebras, Journal of Combinatorial Theory 8 (1970), 115-133.

[BOBROW 68]
BOBROW, D.G., Storage management in LISP, In [BOBROW 68a].

[BOBROW 68a]
BOBROW, D.G., (Ed.), Symbol Manipulation Languages and Techniques, North-Holland Publishing Company, Amsterdam (1968).

[BRANQUART & LEWI 71]
BRANQUART, P., & LEWI, J., A scheme of storage allocation and garbage collection for ALGOL 68, In [PECK 71].

[CHENEY 70]
CHENEY, C.J., A nonrecursive list compacting algorithm, Communications of the ACM 13 (1970), 677-678.

[CHURCH 41]
CHURCH, A., The calculi of lambda-conversion, Annals of Mathematical Studies, 6, Princeton University Press, Princeton (1941).

[CLARK 76]
CLARK, D.W., An efficient list-moving algorithm using constant workspace, Communications of the ACM 19 (1976), 352-354.

[CLARK 78]
CLARK, D.W., A fast algorithm for copying list structures, Communications of the ACM 21 (1978), 351-357.

[COLLINS 60]
COLLINS, G.E., A method for overlapping and erasure of lists, Communications of the ACM 3 (1960), 655-657.

[DAHL et al. 72]
DAHL, O.J., DIJKSTRA, E.W., & HOARE, C.A.R., Structured Programming, Academic Press, London (1972).

[DARLINGTON 78]
DARLINGTON, J., A synthesis of several sorting algorithms, Acta Informatica 11 (1978), 1-30.

[DARLINGTON 79]
DARLINGTON, J., Program transformation: an introduction and survey, Computer Bulletin 22, 2 (1979), 22-24.

[DE BAKKER 80]
BAKKER, J.W. de, Mathematical Theory of Program Correctness, Prentice-Hall International, Inc., London (1980).

[DE BAKKER & VAN VLIET 81]
BAKKER, J.W. de, & VLIET, J.C. van, (Eds.), Algorithmic Languages, North-Holland Publishing Company, Amsterdam (1981).

[DE BRUIJN 80]
BRUIJN, N.G. de, A survey of the project AUTOMATH, In [SELDIN & HINDLEY 80].

[DE ROEVER 78]
ROEVER, W.P. de, On backtracking and greatest fixpoints, In [NEUHOLD 78].

[DERSHOWITZ 80]
DERSHOWITZ, N., The Schorr-Waite marking algorithm revisited, Information Processing Letters 11 (1980), 141-143.

[DEUTSCH & BOBROW 76]
DEUTSCH, L.P., & BOBROW, D.G., An efficient, incremental, automatic garbage collector, Communications of the ACM 19 (1976), 522-526.

[DEWAR & McCANN 77]
DEWAR, R.B.K., & McCANN, A.P., MACRO SPITBOL - a SNOBOL4 compiler, Software-Practice and Experience 7 (1977), 95-113.

[DIJKSTRA 68]
DIJKSTRA, E.W., Cooperating sequential processes, In [GENUYS 68].

[DIJKSTRA 75]
DIJKSTRA, E.W., Guarded commands, nondeterminacy and formal derivation of programs, Communications of the ACM 18 (1975), 453-457.

[DIJKSTRA 76]
DIJKSTRA, E.W., A Discipline of Programming, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1976).

[DIJKSTRA et al. 78]
DIJKSTRA, E.W., LAMPORT, L., MARTIN, A.J., SCHOLTEN, C.S., & STEFFENS, E.M.F., On-the-fly garbage collection: an exercise in cooperation, Communications of the ACM 21 (1978), 966-975.

[DIJKSTRA et al. 81]
DIJKSTRA, E.W., KRUSEMAN ARETZ, F.E.J., LUNBECK, R.J., & REM, M., Aan wie het regardeert!, Letter to the Dutch computing science community (1981).

[DUNCAN & YELOWITZ 79]
DUNCAN, A.G., & YELOWITZ, L., Studies in abstract/concrete mappings in proving algorithm correctness, Proceedings of the Sixth Colloquium on Automata, Languages and Programming, Graz, Austria (1979), 218-229.

[EARLEY 71]
EARLEY, J., Toward an understanding of data structures, Communications of the ACM 14 (1971), 617-627.

[EARLEY 73]
EARLEY, J., Relational level data structures for programming languages, Acta Informatica 2 (1973), 293-309.

[ELSWORTH 79]
ELSWORTH, E.F., Compilation via an intermediate language, Computer Journal 22 (1979), 226-233.

[FENICHEL & YOCHELSON 69]
FENICHEL, R.R., & YOCHELSON, J.C., A LISP garbage-collector for virtual-memory computer systems, Communications of the ACM 12 (1969), 611-612.

[FISHER 74]
FISHER, D.A., Bounded workspace garbage collection in an address-order preserving list processing environment, Information Processing Letters 3 (1974), 29-32.

[FISHER 75]
FISHER, D.A., Copying cyclic list structures in linear time using bounded workspace, Communications of the ACM 18 (1975), 251-252.

[FITCH & NORMAN 78]
FITCH, J.P., & NORMAN, A.C., A note on compacting garbage collection, Computer Journal 21 (1978), 31-34.

[GENUYS 68]
GENUYS, F., (Ed.), Programming Languages, Academic Press, London (1968).

[GERHART 75]
GERHART, S.L., Correctness-preserving program transformations, Conference Record of the Second ACM Symposium on Principles of Programming Languages, Palo Alto (1975), 54-66.

[GERHART 79]
GERHART, S.L., A derivation-oriented proof of the Schorr-Waite marking algorithm, In [BAUER & BROY 79].

[GOGUEN et al. 78]
GOGUEN, J.A., THATCHER, J.W., & WAGNER, E.G., An initial algebra approach to the specification, correctness and implementation of abstract data types, In [YEH 78].

[GRIES 71]
GRIES, D., Compiler Construction for Digital Computers, John Wiley & Sons, New York (1971).

[GRIES 79]
GRIES, D., The Schorr-Waite graph marking algorithm, Acta Informatica 11 (1979), 223-232.

[GUTTAG & HORNING 78]
GUTTAG, J.V., & HORNING, J.J., The algebraic specification of abstract data types, Acta Informatica 10 (1978), 27-52.

[HADDON & WAITE 67]
HADDON, B.K., & WAITE, W.M., A compaction procedure for variable-length storage elements, Computer Journal 10 (1967), 162-165.

[HANSEN 69]
HANSEN, W.J., Compact list representation: definition, garbage collection, and system implementation, Communications of the ACM 12 (1969), 499-507.

[HANSON 77]
HANSON, D.R., Storage management for an implementation of SNOBOL4, Software-Practice and Experience 7 (1977), 179-192.

[HAREL et al. 77]
HAREL, D., PNUELI, A., & STAVI, J., A complete axiomatic system for proving deductions about recursive programs, Proceedings of the Ninth Annual ACM Symposium on Theory of Computing, Boulder, Colorado (1977).

[HART & EVANS 64]
HART, T.P., & EVANS, T.G., Notes on implementing LISP for the M-460 computer, In [BERKELEY & BOBROW 64].

[HILL 74]
HILL, U., Special run-time organization techniques for ALGOL 68, In [BAUER & EICKEL 74].

[HOARE 69]
HOARE, C.A.R., An axiomatic basis for computer programming, Communications of the ACM 12 (1969), 576-580.

[HOARE 72]
HOARE, C.A.R., Proof of correctness of data representations, Acta Informatica 1 (1972), 271-281.

[HOARE 81]
HOARE, C.A.R., The emperor's old clothes, ACM Turing award lecture, Communications of the ACM 24 (1981), 75-83.

[HOPCROFT & ULLMAN 79]
HOPCROFT, J.E., & ULLMAN, J.D., Introduction to Automata Theory, Languages, and Computation, Addison-Wesley Publishing Company, Reading, Massachusetts (1979).

[JENSEN & WIRTH 74]
JENSEN, K., & WIRTH, N., PASCAL User Manual and Report, Springer-Verlag, Berlin (1974).

[JONKERS 79]
JONKERS, H.B.M., A fast garbage compaction algorithm, Information Processing Letters 9 (1979), 26-30.

[JONKERS 80]
JONKERS, H.B.M., Garbage collection, In [VAN VLIET 80].

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

[JONKERS 80b]
JONKERS, H.B.M., Designing a machine independent storage management system, Report IW 148, Mathematical Centre, Amsterdam (1980).

[JONKERS 81]
JONKERS, H.B.M., Abstract storage structures, In [DE BAKKER & VAN VLIET 81].

[KENNEDY & SCHWARTZ 75]
KENNEDY, K., & SCHWARTZ, J., An introduction to the set theoretical language SETL, Computers & Mathematics with Applications 1 (1975), 97-119.

[KING 74]
KING, P.R., (Ed.), Proceedings of an International Conference on ALGOL 68 Implementation, Winnipeg (1974).

[KLEENE 36]
KLEENE, S.C., General recursive functions of natural numbers, Mathematische Annalen 112 (1936), 340-353.

[KNUTH 68]
KNUTH, D.E., The Art of Computer Programming, Volume 1: Fundamental Algorithms, Addison-Wesley Publishing Company, Reading, Massachusetts (1968).

[KNUTH 73]
KNUTH, D.E., The Art of Computer Programming, Volume 3: Sorting and Searching, Addison-Wesley Publishing Company, Reading, Massachusetts (1973).

[KNUTH 74]
KNUTH, D.E., Structured programming with goto statements, Computing Surveys 6 (1974), 261-301.

[KOWALTOWSKI 79]
KOWALTOWSKI, T., Data structures and correctness of programs, Journal of the ACM 26 (1979), 283-301.

[LAMPSON et al. 77]
LAMPSON, B.W., HORNING, J.J., LONDON, R.L., MITCHELL, J.G., & POPEK, G.L., Report on the Programming Language EUCLID, SIGPLAN Notices 12, 2 (1977).

[LEE et al. 79]
LEE, S., ROEVER, W.P. de, & GERHART, S.L., The evolution of list-copying algorithms and the need for structured program verification, Conference Record of the Sixth ACM Symposium on Principles of Programming Languages, San Antonio (1979), 53-67.

[LINDSTROM 74]
LINDSTROM, G., Copying list structures using bounded workspace, Communications of the ACM 17 (1974), 198-202.

[LISKOV et al. 77]
LISKOV, B., SNYDER, A., ATKINSON, R., & SCHAFFERT, C., Abstraction mechanisms in CLU, Communications of the ACM 20 (1977), 564-576.

[LISKOV & ZILLES 74]
LISKOV, B., & ZILLES, S., Programming with abstract data types, Proceedings of a Symposium on Very High Level Languages, SIGPLAN Notices 9, 4 (1974), 50-59.

[LISKOV & ZILLES 75]
LISKOV, B.H., & ZILLES, S.N., Specification techniques for data abstractions, IEEE Transactions on Software Engineering SE-1 (1975), 7-19.

[MAJSTER 77]
MAJSTER, M.E., Extended directed graphs, a formalism for structured data and data structures, Acta Informatica 8 (1977), 37-59.

[MANNA & WALDINGER 80]
MANNA, Z., & WALDINGER, R., A deductive approach to program synthesis, Transactions on Programming Languages and Systems 2 (1980), 90-121.

[MANNA & WALDINGER 80a]
MANNA, Z., & WALDINGER, R., Problematic features of programming languages: a situational-calculus approach, Part 1: Assignment statements, Technical Note 226, SRI International, Menlo Park, California (1980).

[MARSHALL 71]
MARSHALL, S., An ALGOL 68 garbage collector, In [PECK 71].

[McBETH 63]
McBETH, J.H., On the reference counter method, Communications of the ACM 6 (1963), 575.

[McCARTHY 60]
McCARTHY, J., Recursive functions of symbolic expressions and their computation by machine, Part I, Communications of the ACM 3 (1960), 184-195.

[McCARTHY et al. 65]
McCARTHY, J., et al., LISP 1.5 Programmer's Manual, MIT Press, Cambridge, Massachusetts (1965).

[MEERTENS 76]
MEERTENS, L.G.L.T., From abstract variable to concrete representation, In [SCHUMAN 76].

[MEERTENS 79]
MEERTENS, L.G.L.T., Abstracto 84: the next generation, Proceedings of the 1979 Annual Conference of the ACM, Detroit (1979), 33-39.

[MEERTENS 81]
MEERTENS, L.G.L.T., Definition of an abstract ALGOL 68 machine, To appear, Mathematical Centre, Amsterdam.

[MENDELSON 64]
MENDELSON, E., Introduction to Mathematical Logic, Van Nostrand Company, Inc., Princeton, New Jersey (1964).

[MORRIS 78]
MORRIS, F.L., A time- and space-efficient garbage compaction algorithm, Communications of the ACM 21 (1978), 662-665.

[MULLER 75]
MULLER, K.G., On the Feasibility of Concurrent Garbage Collection, Ph.D. Thesis, Technical University, Delft, The Netherlands (1975).

[NEUHOLD 78]
NEUHOLD, E.J., (Ed.), Formal Descriptions of Programming Concepts, North-Holland Publishing Company, Amsterdam (1978).

[OLLONGREN 74]
OLLONGREN, A., Definition of Programming Languages by Interpreting Automata, Academic Press, London (1974).

[PARNAS 72]
PARNAS, D.L., A technique for software module specification with examples, Communications of the ACM 15 (1972), 330-336.

[PECK 71]
PECK, J.E.L., (Ed.), ALGOL 68 Implementation, North-Holland Publishing Company, Amsterdam (1971).

[PLAISTED 81]
PLAISTED, D.A., Theorem proving with abstraction, Artificial Intelligence 16 (1981), 47-108.

[RANDELL & RUSSELL 64]
RANDELL, B., & RUSSELL, L.J., ALGOL 60 Implementation, Academic Press, London (1964).

[REINGOLD 73]
REINGOLD, E.M., A nonrecursive list moving algorithm, Communications of the ACM 16 (1973), 305-307.

[ROBSON 74]
ROBSON, J.M., Garbage collection with limited stack size, In [KING 74].

[ROBSON 77]
ROBSON, J.M., A bounded storage algorithm for copying cyclic list structures, Communications of the ACM 20 (1977), 431-433.

[ROSENBERG 71]
ROSENBERG, A.L., Data graphs and addressing schemes, Journal of Computer and System Sciences 5 (1971), 193-238.

[SCHORR & WAITE 67]
SCHORR, H., & WAITE, W.M., An efficient machine-independent procedure for garbage collection in various list structures, Communications of the ACM 10 (1967), 501-506.

[SCHUMAN 76]
SCHUMAN, S.A., (Ed.), New Directions in Algorithmic Languages, IRIA, Rocquencourt (1976).

[SELDIN & HINDLEY 80]
SELDIN, J.P., & HINDLEY, J.R., (Eds.), To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Academic Press, London (1980).

[STANDISH 78]
STANDISH, T.A., Data structures - An axiomatic approach, In [YEH 78].

[STEEL 75]
STEEL, G.L., Jr., Multiprocessing compactifying garbage collection, Communications of the ACM 18 (1975), 495-508.

[TARJAN 72]
TARJAN, R., Depth-first search and linear graph algorithms, SIAM Journal on Computing, Volume 1 (1972), 146-160.

[TERASHIMA & GOTO 78]
TERASHIMA, M., & GOTO, E., Genetic order and compactifying garbage collectors, Information Processing Letters 7 (1978), 27-32.

[THORELLI 72]
THORELLI, L., Marking algorithms, BIT 12 (1972), 555-568.

[THORELLI 76]
THORELLI, L., A fast compactifying garbage collector, BIT 16 (1976), 426-441.

[TOPOR 79]
TOPOR, R.W., The correctness of the Schorr-Waite list marking algorithm, Acta Informatica 11 (1979), 211-221.

[TRAUB 81]
TRAUB, J.F., (Ed.), Quo vadimus: Computer science in a decade, Communications of the ACM 24 (1981), 351-369.

[TURING 36]
TURING, A.M., On computable numbers, with an application to the Entscheidungsproblem, Proceedings of the London Mathematical Society, Series 2, 42 (1936), 230-265.

[VAN VLIET 80]
VLIET, J.C. van, (Ed.), Colloquium Capita Implementatie van Programmeertalen, MC Syllabus 42, Mathematical Centre, Amsterdam (1980).

[VAN WIJNGAARDEN et al. 76]
WIJNGAARDEN, A. van, MAILLOUX, B.J., PECK, J.E.L., KOSTER, C.H.A., SINTZOFF, M., LINDSEY, C.H., MEERTENS, L.G.L.T., & FISKER, R.G., Revised Report on the Algorithmic Language ALGOL 68, Springer-Verlag, New York (1976).

[WADLER 76]
WADLER, P.L., Analysis of an algorithm for real time garbage collection, Communications of the ACM 19 (1976), 491-500.

[WAITE 73]
WAITE, W.M., Implementing Software for Non-numeric Applications, Prentice-Hall, Englewood Cliffs, New Jersey (1973).

[WEGBREIT 72]
WEGBREIT, B., A generalised compactifying garbage collector, Computer Journal 15 (1972), 204-208.

[WEGNER 72]
WEGNER, P., The Vienna Definition Language, Computing Surveys 4 (1972), 5-63.

[WEISSMAN 67]
WEISSMAN, C., LISP 1.5 Primer, Dickenson Publishing Company, Inc., Belmont, California (1967).

[WILKES et al. 57]
WILKES, M.V., WHEELER, D.J., & GILL, S., The Preparation of Programs for an Electronic Digital Computer, Second Edition, Addison-Wesley, New York (1957).

[WIRTH 77]
WIRTH, N., MODULA: a language for modular multiprogramming, Software-Practice and Experience 7 (1977), 3-35.

[WISE & FRIEDMAN 77]
WISE, D.S., & FRIEDMAN, D.P., The one-bit reference count, BIT 17 (1977), 351-359.

[WODON 71]
WODON, P.L., Methods of garbage collection, In [PECK 71].

[WULF 77]
WULF, W.A., Languages and structured programs, In [YEH 77].

[WULF et al. 76]
WULF, W.A., LONDON, R., & SHAW, M., An introduction to the construction and verification of ALPHARD programs, IEEE Transactions on Software Engineering SE-2 (1976), 253-264.

[YEH 77]
YEH, R.T., (Ed.), Current Trends in Programming Methodology, Volume I, Software Specification and Design, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1977).

[YEH 78]
YEH, R.T., (Ed.), Current Trends in Programming Methodology, Volume IV, Data Structuring, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1978).

[ZAVE 73]
ZAVE, D.A., A fast compacting garbage collector, Technical Report, Sperry Univac (1973).

[ZILLES 73]
ZILLES, S.N., Procedural encapsulation: a linguistic protection technique, Proceedings of an ACM SIGPLAN-SIGOPS Interface Meeting, SIGPLAN Notices 8, 9 (1973), 140-146.

SAMENVATTING

Dit proefschrift vindt zijn oorsprong in het ontwerp van een garbage collector voor een machine-onafhankelijke ALGOL-68-implementatie. Bij een poging het ontwerp van een garbage collector op een systematische manier aan te pakken wordt men geconfronteerd met een aantal problemen. Allereerst is er het probleem van de complexiteit. Een garbage collector, in zijn uiteindelijke vorm, is een "laag-niveau" routine, die moet opereren in een uiterst complex systeem. Ten einde deze complexiteit tijdens iedere fase van het ontwerpproces onder de duim te houden is het gebruik van *abstractie* onontbeerlijk. Op de tweede plaats is er het probleem hoe de garbage collector, en het systeem waarin deze opereert, op verschillende niveaus van abstractie in een begrijpelijke en precieze vorm te beschrijven. Dit is het probleem van de *specificatie*. Tenslotte is er de vraag hoe men, uitgaande van een abstracte specificatie van het garbage-collection-probleem, op een bewijsbaar correcte manier kan komen tot een werkende garbage collector, oftewel het probleem van de *implementatie*.

Deze drie begrippen - abstractie, specificatie en implementatie - vormen het centrale thema van dit proefschrift. Dit houdt in dat dit proefschrift gezien kan worden als een studie over abstractie, specificatie en implementatie. Doel van de bestudering van deze begrippen is het ontwikkelen van eenvoudige wiskundige concepten en methoden, waarbij de practische toepasbaarheid voorop staat. Gezien het algemene en fundamentele karakter van deze begrippen is hun bestudering uit het beperkte kader van garbage collection getild. Garbage collection, dat de voedingsbodem was voor de ideeën neergelegd in dit proefschrift, wordt uitsluitend gebruikt om de practische toepasbaarheid van deze ideeën aan te tonen.

Het bovenstaande houdt in dat dit proefschrift in wezen in twee delen uiteenvalt. In het eerste deel (hoofdstukken 1 t/m 3) wordt een aantal ideeën over abstractie, specificatie en implementatie ontwikkeld, welke in het tweede deel (hoofdstukken 4 t/m 7) worden toegepast op het onderwerp geheugenbeheer in het algemeen, en garbage collection in het bijzonder.

In hoofdstuk 1 wordt op basis van een eenvoudig model voor abstractie besproken hoe abstractie op een systematische manier gebruikt kan worden bij de oplossing van problemen en bij het ordenen van grote klassen problemen en hun oplossingen (met name "algoritmen"). In hoofdstuk 2 wordt

de aanzet gegeven tot de ontwikkeling en formalisering van een specificatietaal voor algoritmen en datastructuren, die in grove lijnen overeenkomt met de (informelere) taal die in het proefschrift gebruikt wordt voor de beschrijving van algoritmen en datastructuren. Uitgangspunt daarbij is het begrip "structure", dat het mogelijk maakt willekeurige datastructuren te modelleren zonder gebruikmaking van "pointers". Hoofdstuk 3 beschrijft een eenvoudige implementatiemethode voor zowel algoritmen als datastructuren, gebaseerd op een vierstapstechniek voor het bewerkstelligen van een verandering van datarepresentatie. De effectiviteit van deze methode (met name als een stuk verificatiegereedschap) wordt gedemonstreerd aan de hand van een afleiding van de Deutsch-Schorr-Waite-markeringsalgoritme.

Het doel van hoofdstuk 4 is de introductie van een model voor geheugenbeheer, dat gebruikt wordt als basis voor hoofdstuk 5. De formele presentatie van dit model wordt voorafgegaan door een uitvoerige informele discussie. Op basis van de ideeën uit hoofdstuk 1 en het model voor geheugenbeheer uit hoofdstuk 4 wordt in hoofdstuk 5 een overzicht gegeven van het onderwerp garbage collection. Uitgaande van twee abstracte algoritmen worden de voornaamste garbage-collection- en compactie-algoritmen met behulp van "correctheid-behoudende transformaties" afgeleid.

Onderwerp van hoofdstuk 6 is het ontwerp van een systeem voor geheugenbeheer, te gebruiken in de eerder genoemde machine-onafhankelijke ALGOL-68-implementatie. De gebruikte methode is die uit hoofdstuk 3, waarbij een abstract model gebruikt wordt om de complexiteit in de hand te houden. Het ontwerp van de garbage collector, die in het in hoofdstuk 6 ontwikkelde systeem abstract wordt gehouden, wordt beschreven in hoofdstuk 7. De aanpak is analoog aan die van hoofdstuk 6, zij het dat de transformatie van de garbage collector wordt doorgezet tot op het niveau van machinecode.

CURRICULUM VITAE

De schrijver van dit proefschrift werd op 6 juli 1949 geboren in de Watergraafsmeer te Amsterdam. In 1968 behaalde hij aan het St.-Vituscollege te Bussum het diploma HBS-B. In 1978 legde hij aan de Technische Hogeschool Eindhoven het ingenieursexamen wiskunde af. Sinds 1 april 1978 is hij als wetenschappelijk medewerker verbonden aan het Mathematisch Centrum te Amsterdam, waar hij werkzaam is in de Afdeling Informatica onder leiding van Prof.dr. J.W. de Bakker.

Current address of the author:

H.B.M. Jonkers
Mathematisch Centrum
Kruislaan 413
1098 SJ Amsterdam