PROGRAM DERIVATION THROUGH TRANSFORMATIONS:
the evolution of list-copying algorithms

N.W.P. van Diepen and W.P. de Roever

RUU-CS-85-3
January 1985

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

PROGRAM DERIVATION THROUGH TRANSFORMATIONS:
the evolution of list-copying algorithms

N.W.P. van Diepen and W.P. de Roever

Technical Report RUU-CS-85-3
January 1985

Department of Computer Science
University of Utrecht
P.O. Box 80.012, 3508 TA Utrecht
the Netherlands

ABSTRACT

The introduction of Hoare's Logic made it feasible to supply correctness proofs of small sequential programs. While correctness proofs of larger programs could be given in principle, the increased size of such a proof warranted additional organization. The present paper puts emphasis on the technique of program transformation to prove the correctness of some fast list-copying algorithms developed by Robson [R], Fisher [F] and Clark [C]. This subject was motivated by an earlier paper on the same topic by Lee, De Roever and Gerhart [LRG]. Some transformation rules necessary for the correctness proofs are given. Other proof techniques used include data refinement and the use of auxiliary variables.

CONTENTS

§ 1: INTRODUCTION

> "How can one organize the understanding of complex algorithms? People have been thinking about this issue at least since Euclid first tried to explain his innovative greatest common divisor algorithm to his colleagues, but for current research into verifying state-of-the-art programs, some precise answers to the question are needed. Over the past decade the various verification methods which have been introduced (inductive assertions, structural induction, least-fixedpoint semantics, etc.) have established many basic principles of program verification (which we define as: establishing that a program text satisfies a given pair of input-output specifications). However, it is no coincidence that most published examples of the application of these methods have dealt with "toy programs" of carefully considered simplicity.
>
> Experience indicates that these "first generation" principles, with which one can easily verify a three-line greatest common divisor algorithm, do not directly enable one to verify a 10,000 line operating system (or even a 50 line list-processing algorithm) in complete detail. To verify complex programs, additional techniques of organization, analysis and manipulation are required. (That a similar situation exists in the writing of large, correct programs has long been recognized -- structured programming being one solution.)" (from Lee et al. [LRG])

Though written half a decade ago the introductory statement of Lee, De Roever and Gerhart [LRG] is still valid today. The present paper may be considered as a follow-up on their work in that it works out and extends their techniques, though it is written to be read independently.

Verification of the correctness of computer programs is a subject becoming more important with the current increasing emphasis on reliable systems programming [N]. To be able to perform correctness proofs in a complicated environment a firm grasp of the possibilities of the available basic tools is necessary. Presently basic tools with some practical claim of utility have been developed to prove the correctness of sequential programs. The present paper aims at showing the possibilities of the tools available nowadays by proving the correctness of some really intricate sequential programs.

The techniques used fall into three general categories, though overlapping occurs frequently. The first category is the straightforward correctness proof. The accepted tool is a series of proof rules developed by Tony Hoare, called Hoare's Logic. Then there is the technique of refinement as described by Jones [J]. For instance a set can be represented by a search tree. Thirdly the technique of program transformation is used. This technique will be explained in § 3. It is based on Hoare's logic system.

The algorithms chosen to demonstrate these proof techniques are three list-copying algorithms. The correctness proofs of these algorithms were already treated in varying depth by Lee, De Roever and Gerhart [LRG].

A list is a structure consisting of a number of nodes, each with two pointers. These pointers are designated as the left and right pointer respectively (or sometimes as car and cdr; this notation will not be used). One of these nodes is called the root and all other nodes can be reached by following pointers starting with the root. Loops and duplicate paths are allowed.

A list-copying algorithm is an algorithm making a duplicate of a list, i.e. to every node will be assigned a copy node, the information is copied and the pointers are replaced in the copy by pointers to the copies of the nodes pointed to.

The correctness of the three best list-copying algorithms presently known will be proved in §§ 4, 5 and 6. The outlines of the proofs are given in the next paragraph.

Finally § 7 contains some notes on related work.

structure to a node is described using A for right and B for left. This claim seems to be incorrect, since node AB is visited before node AAB.

The second traversal uses the same algorithm. The third traversal uses the queue of the first and second traversal in the opposite direction in order to visit every node. Hence this is a queue-traversal, not a list-traversal. Since every node in the list was added to the queue, trivially all nodes in the list are marked again.

In the proof of the correctness of the three traversals above, the queue will be implemented by using auxiliary variables. The Fisher list-copying algorithm will implement this queue by using the copied nodes. It demands that these nodes are of fixed size, and are placed in a contiguous part of the memory not separated by unused intervals. Then it is possible to find the next copy node by adding the size of the node to the address of the present copy node. Proving the correctness of this implementation is very much machine-dependent and should be done by the implementer of the Fisher algorithm for his particular machine. In the sequel the correctness of this implementation will be assumed.

Another advantage of the strategy to place the copy nodes in a contiguous part of the memory is the possibility to test if a pointer is pointing to this part of the memory, i.e. to a copy node. The correctness proof of this test is machine-dependent again and should be done by the implementer. In the sequel this test is treated abstractly by introducing the Boolean procedure iscopy.

Assuming correctness of the implementations of the queue, its operations, and the test iscopy, and having proved the correctness of the list-traversal used, we can add the copying actions to the various traversals in this algorithm. The first subtask of list-copying, assigning a new node to every original one, is done entirely in the first traversal. Expansion of the auxiliary variable necessary for the queue will make it possible to describe where in the original and copy nodes the necessary information is stored. The second subtask of list-copying, storing the correct value in every pointer field, is divided between the traversals. With the expanded auxiliary variables it will be proved that

after the third traversal every pointer is correctly restored in the original and copied in the copy node. The assumption of the correct implementation of the queue makes it possible to remove the auxiliary variables (otherwise only necessary for the proof), and a correct list-marking algorithm remains.

The Clark Algorithm

The Clark list-copying algorithm is based on one kind of list-marking algorithm. This algorithm is a depth-first-search of a directed graph with an auxiliary stack containing all the already marked nodes having a possibly unmarked leftchild. Again, this list-marking algorithm will be derived from lm0.

Dependent on the spanning tree defined by the marking algorithm used Clark differentiates between three types of pointers: pointers to atoms (A), forward pointers in the spanning tree (F), and back pointers in the spanning tree (B). Using these pointer types nine types of nodes can be discerned. (E.g. a node with a left forward pointer and a right back pointer is of type FB.) The used list-marking algorithm only differentiates between no forward pointers (type AA, AB, BA or BB), one forward pointer (type FA, FB, AF or BF) or two forward pointers (type FF). The Clark list-copying algorithm treats every type differently.

The list-copying algorithm traverses a list structure twice using the list-marking algorithm above. In between these two traversals a stack containing all nodes of type BB is emptied and these nodes are given a different treatment.

Like the Fisher algorithm, the Clark list-copying algorithm requires that the copy of the original list structure is placed in a contiguous part of memory. The Clark algorithm makes use of three advantages which follow from the technique of contiguous copying. The test iscopy seen in Fisher's algorithm before is used. Again, the address of the next copy node after the present copy node can be calculated by adding the size of a node to the address of the latter node. Finally a test on back pointing in the copy structure (the address of the node pointed to should be smaller than the address of the present node since the former was copied earlier) is introduced. These three machine-dependent extras will be treated abstractly. Correctness of the implemen-

tations should be proved by the implementor. In the sequel correctness of these implementations will be assumed.

The first traversal of the list-copying algorithm will be derived from the list-marking algorithm by adding part of the copying action. An auxiliary variable again makes it possible to ensure that all necessary information is stored in either the original node or its assigned copy. Every type of node is treated differently. As mentioned before, the BB-type nodes are stored on a stack to make a special treatment possible. In this traversal the entire first subtask of list-copying, assigning a copy node to every original node, is executed. The second task, assigning the correct pointer values to both the copy and the original list structure, is divided between the two traversals and the processing of the stack with BB-type nodes.

Next, the changes in the BB-type nodes will be finished by emptying the aforementioned stack. The necessary information can be traced again with the aid of the auxiliary variables.

These auxiliary variables make it possible to derive the second traversal of the list-copying algorithm from the same list-marking algorithm, since they code the information of the original list structure. Adding the copying actions and proving their correctness is the last step in which these auxiliary variables are used. Next, these variables can be removed and a correct list-copying algorithm remains.

§ 3: TRANSFORMATIONS IN HOARE'S LOGIC.

Introduction

A commonly accepted tool for the correctness proof of sequential programs is the logic developed by Tony Hoare. Its rules can by now be found in basic texts on program correctness (e.g. Livercy [L]) so they will not be repeated here again.

The basic idea of a proof in Hoare's Logic is the possibility to prove the correctness of some post-condition Q when a statement S is executed with some pre-condition P to begin with. This is written as:

$$\{P\}\ S\ \{Q\}$$

The pre-condition P can follow from the post-condition Q' of some other statement S' with pre-condition P' and the post-condition Q can induce some pre-condition P" of a statement S" with post-condition Q". Then it is possible to construct a series of statements:

$$\{P'\}\ S';\ \{Q'\}\ P\ \{S;\{\ Q\ \{P"\}\ S"\ \{Q"\}\ \text{or}\ \{P'\}\ S';\ S;\ S"\ \{Q"\}$$

So a new statement $T \triangleq S';S;S"$ is constructed with pre-condition P' and post-condition Q".

Suppose some other statement R also has post-condition Q if pre-condition P is assumed. The actual post-condition of R may be stronger, since Hoare's Logic allows weakening of post-conditions. Then it is also possible to construct:

$$\{P'\}\ S';\ \{Q'\}\{P\}\ R;\ \{Q\}\{P"\}\ S"\ \{Q"\}\ \text{or}\ \{P\}\ S';\ R;\ S"\ \{Q"\}$$

So we have a new statement $T' \triangleq S';R;S"$ with pre-condition P' and post-condition Q". As far as the specification by pre- and post-condition of T and T' is concerned both statements perform the same action. In the interior statement S is replaced by statement R in such a way that the performance isn't changed. Thus the replacement of S by R is a valid program transformation in Hoare's Logic.

The correctness dealt with so far was the so-called partial

correctness. If a statement is executed with pre-condition P then post-condition Q is valid upon termination. The key phrase is "upon termination". A non-terminating statement might validate any post-condition, since the end-situation is never reached. Then the notion of total correctness is needed. A statement is total correct with respect to pre-condition P and post-condition Q if the statement will terminate if P is valid with Q as the resulting condition.

The proof of total correctness amounts to the proof of partial correctness combined with a proof of termination. Such a termination proof consists of the definition of a well-founded set and a function from the set of possible values of all the variables in the program to this well-founded set, so that the value of the function monotonously decreases when the statement is executed. The transformations dealt with in this paper all preserve partial correctness. Total correctness generally has to be proved again, either by defining a new set and function from scratch, or from the observation that the algorithm before the transformation terminated, and the number of additional actions in the resulting algorithm is bounded.

It is impossible to give every transformation rule conceivable according to the rules indicated above, since there are infinitely many rules. For instance with array(r) and sortedarray(r) two conditions expressing that r is an array and r is a sorted array respectively, it is true that:

$$\frac{\{\text{array}(r)\}\ \text{Quicksort}(r)\ \{\text{sortedarray}(r)\}}{\{\text{array}(r)\}\ \text{Bubblesort}(r)\ \{\text{sortedarray}(r)\}}$$

is a valid transformation rule. So attempts at completeness are doomed to fail.

In the remainder of the paragraph some rules will be shown which are used in the examples in the next paragraphs. They are meant to give the reader an idea of what is possible.

Finally something is said about a technique used concerning auxiliary variables, i.e. variables only relevant to the proof and not relevant to the code of the algorithm checked. They can be introduced at will. However, they can take over control of part of the program, thus ceasing to be an auxiliary variable. This will be treated in detail in the last section of this paragraph.

Assignment transformation rules

The assignment rule in Hoare's Logic is:

$$\{P[E/x]\}\ x:=E\ \{P\}$$

where P E/x is assertion P with free occurrences of variable x replaced by expression E. This rule induces two assignment transformation rules. $\vdash$ stands for provable in Hoare's Logic.

Lemma 3.1 For assertions P and Q, expressions E and F and variables x and y the following transformations are valid:

i) (rule A)

$\vdash \{P\}\ x:=E\ \{Q\} \wedge P \rightarrow E=F$

$\Rightarrow \vdash \{P\}\ x:=F\ \{Q\}$

ii) (rule AS)

$\vdash \{P\}\ x:=E;\ y:=F\ \{Q\}$ with x not free in F and y not free in E and $x \neq y$

$\Rightarrow \vdash \{P\}\ y:=F;\ x:=E\ \{Q\}$

Proof i) Since assertion Q is valid after assignment x:=E it follows from Hoare's assignment rule that $Q[E/x]$ was valid before the assignment. Thus P induces $Q[E/x]$, and P induces E=F, so P induces $Q[F/x]$. Application of Hoare's assignment rule justifies the conclusion.

ii) Since assertion Q is valid after assignment y:=F it follows from Hoare's assignment rule that $Q[F/y]$ was valid before this assignment and $Q[F/y][E/x]$ before assignment x:=E. Thus P induces $Q[F/y][E/x]$ and $\{Q[F/y][E/x]\}\ y:=F\ \{Q[E/x]\}$ since x isn't free in F and y isn't free in $E \cup \{x\}$. Hoare's assignment rule justifies $\{Q[E/x]\}\ x:=E\ \{Q\}$, thus the conclusion is proved. □

Conclusions about termination are dependent of the precise nature of expressions E and F. If both terminate then the result of the transformation terminates.

It should be noted that inclusion of an assignment statement as part of a bigger statement is not a transformation. The correctness of the new statement has to be proved entirely in Hoare's Logic, again.

Case-clause transformation rules

The rule for the case-clause in Hoare's logic is:

$$\frac{\{P \wedge B\}\ S_1\ \{Q\} \qquad \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \text{IF B THEN } S_1 \text{ ELSE } S_2 \text{ FI}\ \{Q\}}$$

Quite a lot transformation rules can be derived from this rule. Two of the more representative cases are given below.

Lemma 3.2 For assertions P and Q, boolean expression B and statements S, S_1, S_2 the following transformations are valid:

i) (rule C1)

$\vdash \{P\}\ S_1; \{Q\}\ S_2\ \{R\} \wedge P \wedge \neg B \rightarrow Q$

$\Rightarrow \vdash \{P\}$ IF B THEN S_1; $\{Q\}\ S_2$ ELSE S_2 FI $\{R\}$

ii) (rule C2)

$\vdash \{P\}$ IF B THEN S_1; S ELSE S_2; S FI $\{Q\}$

$\Rightarrow \vdash \{P\}$ IF B THEN S_1 ELSE S_2 FI; S $\{Q\}$

Proof i) The first condition stipulates that $\{P\}\ S_1; S_2\ \{Q\}$ is valid, hence $\{P \wedge B\}\ S_1; S_2\ \{Q\}$ is valid. Since $P \wedge \neg B \rightarrow Q$ and $\{Q\}\ S_2\ \{R\}$ follow from the conditions also $\{P \wedge \neg B\}\ S_2\ \{Q\}$ is valid. Then Hoare's case-clause rule states the validity of the conclusion.

ii) Since the condition is provable there exists an assertion R such that:

$\{P \wedge B\}\ S_1\ \{R\}$ S $\{Q\}$ and

$\{P \wedge \neg B\}\ S_2\ \{R\}$ S $\{Q\}$

Then Hoare's case-clause rule states validity of $\{P\}$ IF B THEN S_1 ELSE S_2 FI $\{R\}$. Also $\{R\}$ S $\{Q\}$ is valid, so the validity of $\{P\}$ IF B THEN S_1 ELSE S_2 FI; S $\{Q\}$ can be concluded. □

Problems concerning termination of the result of the first rule can only result when statement S_2 not preceded by S_1 might not terminate if the statement S_1; S_2 terminates. The resulting statement from the second rule will always terminate if the original statement terminated, since the same statements are executed in exactly the same sequence.

Loop transformation rules

A quite hazardous type of transformation is the loop transformation. Since repetition is already built-in, the danger of introduction of an infinite repetition is very real. Therefore many transformations are eligible in some cases, but only partial correctness preserving in other cases. Three transformation rules which retain total correctness are presented.

The rule for the loop in Hoare's Logic is:

$$\frac{\{I \wedge B\}\ S\ \{I\} \qquad I \wedge \neg B \rightarrow Q}{\{I\}\ \text{WHILE B DO S OD}\ \{Q\}}$$

This rule deals only with partial correctness, of course. Termination has to be proved independently.

Lemma 3.3 For assertions P, P' and Q, boolean expressions B and B' and statements S_1 and S_2 the following transformations are valid, and if the statements in the condition terminate, so will the statement in the conclusion terminate:

i) (rule L1)

$\vdash \{P\}\ S_1; \{P'\}$ WHILE B DO S_2; $\{P\}\ S_1$ OD $\{Q\} \wedge$
$\{P \wedge \neg B'\}\ S_1\ \{P \wedge \neg B\} \wedge \{P \wedge B'\}\ S_1\ \{P' \wedge B\}$

$\Rightarrow \vdash \{P\}$ WHILE B' DO S_1; $\{P'\}\ S_2$ OD; $S_1\ \{Q\}$

ii) (rule L2)

$\vdash \{P \wedge B'\}\{P\}$ WHILE B DO $S_1; S_2$ OD; $S_1\ \{Q\} \wedge$
$\{P \wedge B\}\ S_1; S_2\ \{B'\} \wedge [P \wedge B \rightarrow B'] \wedge \{Q\}\ S_2\ \{Q\} \wedge \{P \wedge \neg B \wedge B'\}\ S_1; S_2\ \{P \wedge \neg B'\}$

$\Rightarrow \vdash \{P \wedge B\}\ \{P\}$ WHILE B' DO $S_1; S_2$ OD $\{Q\}$

iii) (rule L3)

$\vdash \{P\}$ WHILE B DO IF B' THEN S_1 ELSE S_2 FI OD $\{Q\}$

$\Rightarrow \vdash \{P\}$ WHILE B DO WHILE B' $\wedge$ B DO S_1 OD; WHILE $\neg$B' $\wedge$ B DO S_2 OD $\{Q\}$

Proof i) The condition states the partial correctness of $\{P \wedge B'\}\ S_1\ \{P' \wedge B\}$ and $\{P' \wedge B\}\ S_2\ \{P\}$. Thus $\{P \wedge B'\}\ S_1; S_2\ \{P\}$ is partial correct. Then $\{P\}$ WHILE B' DO S_1; $\{P'\}\ S_2$ OD $\{P \wedge \neg B'\}$ is partial correct. The condition states $\{P \wedge \neg B'\}\ S_1\ \{P' \wedge \neg B\}$ and since

$P' \wedge \neg B$ implies Q the partial correctness of the conclusion is proved. Termination is proved by induction on the number of iterations n of the loop in the condition. If n=0 then the upper statement reduces to $\{P\}\ S_1\ \{P' \wedge \neg B\}$. The last two propositions in the condition then indicate that $\{P \wedge \neg B'\}$ was true before the execution of statement S_1. Then the loop in the conclusion is not executed, and hence S_1 is executed, so the statement in the conclusion terminates. Suppose termination is proved for $n=k \geq 0$ iterations. Then for n=k+1 iterations the upper statement reduces to $\{P \wedge B'\}\ S_1\ \{P' \wedge B\}\ S_2\ \{P \wedge B'\}\ S_1\ \{P'\}$ followed by k iterations of the upper loop. The first two statements together form one iteration of the lower loop. The remainder of the execution of the upper statement consists of execution of statement S_1 followed by k iterations of the upper loop. By induction the transformed part terminates. Hence the lower loop terminates for all $n \geq 0$.

ii) The condition states the partial correctness of $\{P \wedge B\}\ S_1;S_2\ \{P\}$. Since $P \wedge B \rightarrow B'$ and $\{P \wedge \neg B \wedge B'\}\ S_1;S_2\ \{P \wedge \neg B'\}$ the validity of $\{P \wedge B'\}\ S_1;S_2\ \{P\}$ is always assured. When the lower loop terminates ($P \wedge \neg B'$ is true) $P \wedge \neg B \wedge B'$ must be true before the last iteration, since if $\{P \wedge B \wedge B'\}$ were true then execution of $S_1;S_2$ would result in the validity of B' afterwards, hence the lower loop wouldn't terminate. The provability of the upper loop implies $\{P \wedge \neg B\}\ S_1\ \{Q\}$ and a condition is $\{Q\}\ S_2\ \{Q\}$, so $\{P \wedge \neg B\}\ S_1;S_2\ \{Q\}$ is valid. So the post-condition of the conclusion is correct. The lower loop trivially terminates, since it adds exactly one iteration to the number of iterations of the upper loop.

iii) Define $P_1 \triangleq P \wedge B \wedge B'$ and $P_2 \triangleq P \wedge B \wedge \neg B'$. Then a terminating computation sequence of the upper loop looks like:

$$*(\{P_i\}\ S_i)_{i=1,2}\{P \wedge \neg B\}$$
$$= *(*(\{P_1\}\ S_1)\ *(\{P_2\}\ S_2))\{P \wedge \neg B\}$$

which is the computation sequence of the lower loop. Hence termination is preserved, and since $P \wedge \neg B \rightarrow Q$ follows from the provability of the upper loop the post-assertion in the lower loop is also preserved. Partial correctness preservation follows directly from the provability of the upper loop, since this implies that $\{P_1\}\ S_1\ \{P\}$ and $\{P_2\}\ S_2\ \{P\}$ are both partial correct. □

A note on auxiliary variables

An integral part of most program correctness proofs in the Hoare system is the use of auxiliary variables. Intuitively an auxiliary variable is a variable appearing only in the correctness proof. An auxiliary variable has no bearing on the flow of control or the output of the program. It merely describes something which can only with difficulty or not at all be described by the original program variables.

Definition 3.4 An auxiliary variable is a variable which appears only in statements of the form:

a:=E

where a is an auxiliary variable and E an expression.

During the transformation series presented in the next paragraphs another use of auxiliary variables will be encountered. After introduction of a new auxiliary variable - which can be done at will - a transformation is applied after which the new variable is no longer satisfying the definition presented above. However, another variable might satisfy the definition of auxiliary variable. Then this variable can be deleted - which can be done at will again -.

An elementary example will illustrate this. Take a look at the following algorithm:

$\{M \geq 1\}$ n:=1; k:=1; $\{n^2=k \leq M\}$
WHILE $(n+1)^2 \leq M$ DO n:=n+1; $k:=n^2$ OD
$\{k \leq M < (n+1)^2\}$

which is clearly correct. Now a new auxiliary variable is introduced:

$\{M \geq 1\}$ n:=1; k:=1; d:=3; $\{n^2=k \leq M \wedge d=(n+1)^2-n^2\}$
WHILE $(n+1)^2 \leq M$ DO n:=n+1; $k:=n^2$; d:=d+2 OD
$\{k \leq M < (n+1)^2 \wedge d=(n+1)^2-n^2\}$

The resulting program is again correct. Then the auxiliary variable d can take over two tasks of variable n; in the loop-condition appears the value $(n+1)^2$ which can be replaced by k+d; and the assignment $k:=n^2$ can be replaced by k:=k+d. The resulting algorithm is:

$\{M \geq 1\}$ n:=1; k:=1; d:=3; $\{n^2=k \leq M \wedge d=(n+1)^2-n^2\}$
WHILE k+d$\leq$M DO n:=n+1; k:=k+d; d:=d+2 OD
$\{k \leq M < (n+1)^2 \wedge d=(n+1)^2-n^2\}$

Variable d is not eligible for inclusion in the set of auxiliary variables in this program. However, variable n is eligible. Hence it can be deleted. The assertions have to be adapted slightly to accomodate for this deletion:

$\{M \geq 1\}$ k:=1; d:=3; $\{\exists n [n^2=k \leq M \wedge d=(n+1)^2-n^2]\}$
WHILE k+d$\leq$M DO k:=k+d; d:=d+2 OD
$\{\exists n [k=n^2 \wedge d=(n+1)^2-n^2 \wedge k \leq M < k+d]\}$

More involved examples of this technique will appear in the sequel. In general some steps will be contracted into one, since the number of intermediate algorithms would greatly hamper legibility. However, our main strategy follows the lines indicated above.

§ 4: ROBSON'S LIST-COPYING ALGORITHM

Introduction

The outline of the proof of the Robson list-copying algorithm is already given in §2. The algorithm makes two traversals in order to copy a given list. Firstly it is proved that those traversals are correct, i.e. encounter every node. Secondly, it is proved that each encountered node is correctly copied while executing these traversals.

A basic list-marking algorithm

In § 2 it was pointed out that a list-marking algorithm is needed as the first step in the derivation of a list-copying algorithm. In this section a correct archetypal algorithm is introduced.

To be able to reason about list-marking some notations have to be introduced. A list can be seen as a finite directed graph with a special node, called the root, from which every other node in the graph can be reached. A relation R on nodes can be defined as follows:

m R n <u>iff</u> there is an edge pointing from m to n

This relation induces the following set definitions:

$R(n) \triangleq \{m \mid nRm\}$
$R^*(n) \triangleq \{n\} \cup \{k \mid \exists m \in R^*(n)\ mRk\}$

Intuitively, if n is the root of a list structure then R(n) is the set of its children and $R^*(n)$ is the set of all nodes in the list. List-marking can formally be described as follows: given a node n and a relation R, construct the set $m=R^*(n)$.

A series of list-marking algorithms was given using this formalism by Lee, De Roever and Gerhart [LRG]. One of these algorithms (called MA3 in [LRG]) is given below in a different notation, as algorithm lm0 in figure 4.1.

The input of algorithm lm0 is a node n, a memory location. The relation R is implicitly defined. Some pointers to nodes,

```
PROC lm0 = (REF NODE n)VOID:
BEGIN {n∈Mem∧ R∈Mem x Mem}
  LOC REF NODE f, s;
  LOC NODESET m, b;
  f:=n; m:={f}; b:=∅;
  {f,n∈m∧m⊆R*(n)∧
   R(m-(b∪{f}))⊆m∧b⊆m-{f}}
  WHILE NOT(b=∅∧R(f)⊆m)
  DO IF NOT R(f)⊆m
    THEN s:=x.x∈R(f)-m;
      m:=m∪{s}; b:=b∪{f}; f:=s
    ELSE f:=x.x∈b; b:=b-{f}
    FI
  OD
END {m=R*(n)}
```

Fig. 4.1: the list-marking archetype

with the obvious meaning of father and son, and two sets of nodes, the set m of marked nodes and the set b (the boundary with the set of unmarked nodes) of marked nodes with possibly unmarked children excluding the father-node, are used.

The algorithm is rather straightforward. If the father-node has any unmarked children, then one is selected, marked, and the search is continued at this new node. The old father-node is added to the boundary-set, since other children might still be unmarked. If all the children of the father-node are marked, then search continues at some element of the boundary-set, unless this set is empty in which case the program terminates.

A note should be made on the random assignment introduced in lm0. A random assignment looks like:

v := x.P(x)

with v a variable, x a fresh variable of the same type, and P a proposition in which v is not a free variable. The intuitive meaning is: assign to v any value x which satisfies P(x). So a rule for this construct in Hoare logic is trivial:

{∃x P(x)} v:=x.P(x) {P(v)} with v not free in P

An example of the use of this rule can be seen in {b≠∅} f:=x.x∈b {f∈b}.

To use this statement in a program demanding mere existence of a value x satisfying P(x) is not sufficient: one must be able to effectively construct a witness x satisfying P(x). At this level of abstraction bothering about this distinction is not necessary yet, since we do not want to actually execute program lm0.

Robson's first list-marking algorithm

In the outline of the proof of the Robson list-copying algorithm in §2 it has been mentioned that this algorithm is based on two list-marking algorithms. The first algorithm, essentially a d.s.w.-algorithm, will be derived from lm0 below. This derivation relies heavily on prior knowledge of the list-marking algorithm to be derived. Therefore some steps will appear useless at first. Later on, their meaning should become clear.

Repeated application of transformation rule AS allows us to change m:=m∪{s}; b:=b∪{f}; f:=s in the then-clause into b:=b∪{f}; f:=s; m:=m∪{s}. After the assignment f:=s naturally m∪{s} equals m∪{f}, so (using rule A) m:=m∪{s} can be transformed into m:=m∪{f}. Similar transformations replace the initializations m:={f}; b:=∅ by m:=∅; b:=∅; m:=m∪{f}.

A good example of an at first sight useless transformation is the next one. The loop-invariant contains the assertion b⊆m-{f}, so the following assertions enclose the statements in the else-clause:

{b⊆m-{f}} f:=x.x∈b; {f∈m} b:=b-{f} {f∈m} {m=m∪{f}}

Transformation rule AI allows addition of m:=m∪{f} at the end of this statement, since it doesn't change the state variables. Then m:=m∪{f} is the last statement in both the then- and the else-clause, so this statement can be moved out of the if-statement (rule C2). Now we have arrived at algorithm lm1 (figure 4.2).

```
PROC lm1 =(REF NODE n)VOID:
BEGIN
  LOC REF NODE f, s;
  LOC NODESET m, b;
  f:=n; m:=∅; b:=∅;
  m:=m∪{f};
  {f,n∈m∧m⊆R*(n)∧
   R(m-(b∪{f}))⊆m∧b⊆m-{f}}
  WHILE NOT(b=∅∧R(f)⊆m)
  DO IF NOT R(f)⊆m
    THEN s:=x.x∈R(f)-m;
      b:=b∪{f}; f:=s
    ELSE f:=x.x∈b; b:=b-{f}
    FI;
    m:=m∪{f}
  OD
END {m=R*(n)}
```

Fig. 4.2: algorithm lm1

```
PROC lm2 =(REF NODE n)VOID:
BEGIN
  LOC REF NODE f, s;
  LOC NODESET m, b;
  f:=n; m:=∅; b:=∅;
  {n∈m∪{f}∧m∪{f}⊆R*(n)∧
   R(m-(b∪{f}))⊆m∪{f}∧b⊆m-{f}}
  WHILE NOT(b=∅∧R(f)⊆m∪{f})
  DO m:=m∪{f};
    IF NOT R(f)⊆m
    THEN s:=x.x∈R(f)-m;
      b:=b∪{f}; f:=s
    ELSE f:=x.x∈b; b:=b-{f}
    FI
  OD;
  m:=m∪{f}
END {m=R*(n)}
```

Fig. 4.3: algorithm lm2

The stage is now set for application of transformation rule L1. Statement S_1 is $m:=m\cup\{f\}$, S_2 is the if-statement, proposition P' is the loop-invariant, P is the loop-invariant with m replaced by $m\cup\{f\}$ and the new loop-test is $NOT(b=\emptyset \wedge R(f)\subseteq m\cup\{f\})$. Clearly the old loop-invariant is *true* if S_1 is executed when the new loop-invariant is *true* and it is *false* if the new invariant was *false*. So the application of rule L1 is allowed. The resulting program lm2 is shown in figure 4.3.

In algorithm lm3 (figure 4.4) a boolean variable down is added. At this stage it is merely an auxiliary variable expressing that the current node f has been reached by going downward or upward in the list-structure.

```
PROC lm3 =(REF NODE n)VOID:
BEGIN
  LOC REF NODE f, s;
  LOC NODESET m, b;
  LOC BOOL down;
  f:=n; m:=∅; b:=∅; down:=TRUE;
  {n∈m∪{f} ∧ m∪{f}≤R*(n) ∧
   R(m-(b∪{f}))⊆m∪{f} ∧ b⊆m-{f} ∧
   f∉m ↔ down}
  WHILE NOT (b=∅ ∧ R(f)⊆m∪{f})
  DO m:=m∪{f};
    IF NOT R(f)⊆m
    THEN s:=x.x∈R(f)-m;
      b:=b∪{f}; f:=s; down:=TRUE
    ELSE f:=x.x∈b; b:=b-{f};
      down:=FALSE
    FI
  OD;
  m:=m∪{f}
END {m=R*(n)}
```

Fig. 4.4: *algorithm lm3*

```
PROC lm4 =(REF NODE n)VOID:
BEGIN
  LOC REF NODE f, s;
  LOC NODESET m, b;
  LOC BOOL down;
  f:=n; m:=∅; b:=∅; down:=TRUE;
  {n∈m∪{f} ∧ m∪{f}≤R*(n) ∧
   R(m-(b∪{f}))⊆m∪{f} ∧ b⊆m-{f} ∧
   f∉m ↔ down}
  WHILE NOT(b=∅ ∧ R(f)⊆m∪{f})
  DO IF down
    THEN m:=m∪{f};
      IF NOT R(f)⊆m
      THEN s:=x.x∈R(f)-m;
        b:=b∪{f}; f:=s
      ELSE f:=x.x∈b; b:=b-{f};
        down:=FALSE
      FI
    ELSE IF NOT R(f)⊆m
      THEN s:=x.x∈R(f)-m;
        b:=b∪{f}; f:=s; down:=TRUE
      ELSE f:=x.x∈b; b:=b-{f}
      FI
    FI
  OD;
  m:=m∪{f}
END {m=R*(n)}
```

Fig. 4.5: *algorithm lm4*

Only when a new node is encountered, either the root at the beginning or a still unmarked child in the search-process, down should be *true*. This is trivially so at the beginning. If after the loop down is *true*, then the first branch of the if-clause was chosen. Then f=s and $s\in R(f)-m$, so $f\notin m$. If after the loop down is *false*, then the second alternative was chosen. Since $b\subseteq m$ according to the loop-invariant $f\in m$ after the assignment $f:=x.x\in b$. The assignment $b:=b-\{f\}$ doesn't change this assertion, hence $f\in m$. So $f\notin m \leftrightarrow$ down holds in the loop-invariant.

Next, unnecessary marking of already marked nodes can be removed by the introduction of a test on the formerly auxiliary variable down. Algorithm lm3 is transformed by applying transformation rule C1. A few redundant assignations to the boolean variable down are also removed. The resulting algorithm lm4 is shown in figure 4.5.

Next, the abstract set b will be implemented in the style of Jones [J] by a stack t with the usual operators push and pop. This stack is just another abstraction level. The implementation of a stack directly is possible of course, however in the case of the algorithms described in this paper a claim is made of bounded workspace. Therefore every real stack will be implemented with nodes and pointers of the original and copy structures of the list to be copied. In the sequel abstract stacks will be treated like sets have been treated, as a primitive notion with known properties and operators.

A new set C(t) - the contents of stack t - has to be defined to replace the occurrences of b in the assertions. It is defined recursively as follows:

$$C(t) \triangleq \begin{cases} \emptyset & \text{if t IS nilst} \\ \{pop(t)\} \cup C(t') & \text{if t ISNT nilst and t' is stack t after the execution of pop(t)} \end{cases}$$

The algorithm resulting after the aforementioned transformation and the implementation of the set b into a stack is given as lm5 in figure 4.6.

For the next transformation the convention pop(nilst)=nil has to be introduced. Then transfomation rule L2 can be applied. The new loop-test B' is f ISNT nil; statement S_1 is $m:=m\cup\{f\}$, which has to be moved out of the if-statement again for the occasion; and statement S_2 is the if-statement itself. The proof of the conditions of rule L2 is straightforward. The only problem is the proof of the fact that the new loop-invariant ensures termination of the loop after only one extra iteration.

If down is *false* then the assertion $\neg B \triangleq empty(t) \wedge R(f)\subseteq m\cup\{f\}$ ensures that f:=pop(t) is executed. Since t is empty then f will

```
PROC lm5 =(REF NODE n)VOID:
BEGIN
  LOC REF NODE f, s;
  LOC NODESET m; LOC STACK t;
  LOC BOOL down;
  f:=n; m:=∅; t:=nilst;
  down:=TRUE;
  {n∈m∪{f}∧m∪{f}⊆R*(n)∧
   R(m-(C(t)∪{f}))⊆m∪{f}∧
   C(t)⊆m-{f}∧f∉m ↔ down}
  WHILE NOT(empty(t)∧R(f)⊆m∪{f})
  DO IF down
     THEN m:=m∪{f};
       IF NOT R(f)⊆m
       THEN s:=x.x∈R(f)-m;
         push(t,f); f:=s
       ELSE f:=pop(t);
         down:=FALSE
       FI
     ELSE IF NOT R(f)⊆m
       THEN s:=x.x∈R(f)-m;
         push(t,f); f:=s; down:=TRUE
       ELSE f:=pop(t)
       FI
     FI
  OD;
  m:=m∪{f}
END {m=R*(n)}
```

Fig. 4.6: algorithm lm5

```
PROC lm6 =(REF NODE n)VOID:
BEGIN
  LOC REF NODE f, s;
  LOC NODESET m; LOC STACK t;
  LOC BOOL down;
  f:=n; m:=∅; t:=nilst;
  down:=TRUE;
  {n∈m∪{f}∧m∪{f}⊆R*(n)∧
   R(m-(C(t)∪{f}))⊆m∪{f}∧
   C(t)⊆m-{f}∧f∉m ↔ down}
  WHILE f ISNT nil
  DO IF down
     THEN m:=m∪{f};
       IF NOT R(f)⊆m
       THEN s:=x.x∈R(f)-m;
         push(t,f); f:=s
       ELSE f:=pop(t);
         down:=FALSE
       FI
     ELSE IF NOT R(f)⊆m
       THEN s:=x.x∈R(f)-m;
         push(t,f); f:=s;
         down:=TRUE
       ELSE f:=pop(t)
       FI
     FI
  OD
END {m=R*(n)}
```

Fig. 4.7: algorithm lm6

be nil. If down is true then the assertion ¬B ensures that m:=m∪{f}; f:=pop(t); down:=FALSE is executed and f will be nil again. So the termination of the loop with the new loop-invariant is assured.

Then the statement m:=m∪{f} can be placed inside the if-statement again. The resulting algorithm lm6 is shown in figure 4.7.

Algorithm lm6 contains a lot of program text twice, so some shortening of the program might be achieved by combining a few branches. Such a combination is possible here by postpening one branch, the branch when down is true and R(f)⊆m. At the moment the transformation used will remove only one statement, however in the Robson list-copying algorithm finally derived it will remove more statements.

Postponement of a statement in a loop-clause is a dangerous transformation. One has to be sure that the loop-invariant is preserved, a relatively easy task in this case. And one has to be sure that termination is still assured. The latter proof will depend heavily on the semantics of the program in question.

The transformation that does the job correctly is the deletion of the statement f:=pop(t) in the case when down is true

and R(f)⊆m. The correctness proof of the loop-invariant reduces to the correctness proof in the case of the single changed branch, since preservation of the loop-invariant was already proved for the other branches. Then it is sufficient to prove that:

{n∈m∪{f}∧m∪{f}∈R*(n)∧R(m-(C(t)∪{f}))⊆m∪{f}∧C(t)⊆m-{f}∧f∉m↔down∧down}
m:=m∪{f};
{n∈m∧m⊆R*(n)∧R(m-(C(t)∪{f}))⊆m∧C(t)⊆m-{f}∧f∈m}
down:=FALSE
{n∈m∪{f}∧m∪{f}⊆R*(n)∧R(m-(C(t)∪{f}))⊆m∪{f}∧C(t)⊆m-{f}∧f∉m↔down}

which is easily verified.

A conventional proof of the termination of the changed loop would consist of the definition of a well-founded set and a function from the set of states of the variables of this program to the well-founded set which is monotonously decreasing if the loop is executed with a state satisfying the loop-invariant Informally it is clear that a loop can be added if and only if down is true. Since down is true only once per node in the list the number of loops added is smaller than or equal to the number of nodes in the list. And that number is finite. Hence the loop terminates after the deletion if and only if it terminates before the deletion.

Algorithm lm6 is still acting on abstract graphs. In practice we are interested in list-structures with a left and a right pointer in every node. These pointers may be nil, in which case the convention is that they do not correspond to an edge in the graph defined by the nodes and the right and left pointers connecting nodes. The abstract relation R is defined in this concrete case for a and b nodes as:

a R b iff b ISNT nil AND (b IS l OF a OR b IS r OF a)

The next step in our transformation series will be the introduction of these concrete pointers and the corresponding tests.

Another feature of Robson's first list-marking algorithm is, that it leaves behind information about the spanning tree it defines in every node. This is done by using four markers, called 0, 1, 2 and 3. It is possible to perform some simple arithmetic with these marks. Upon initial marking - when a node is inserted in set m - the mark is set to 0. If the pointer

to the left is traversed downward, i.e. corresponds to an edge of the spanning tree, 2 is added to the mark; if the pointer to the left is a back pointer or a pointer to nil then the mark is not changed. Similarly 1 is added to the mark if the pointer to the right is added to the spanning tree. Then the mark is greater than or equal to 2 if the left pointer is a forward pointer and the mark is odd if the right pointer is a forward pointer. The algorithm accomodating these changes is given below in figure 4.8 as algorithm lm7.

```
PROC lm7 =(REF NODE n)VOID:
BEGIN
  LOC REF NODE f, s;
  LOC NODESET m; LOC STACK t;
  f:=n; m:=∅; t:=nilst;
  down:=TRUE;
{ n∈m∪{f}∧m∪{f}⊆R*(n)∧R(m-(C(t)∪{f}))⊆m∪{f}∧C(t)⊆m-{f}∧f∉m ↔ down ∧
  PointerCode1'}
  WHILE f ISNT nil
  DO IF down
    THEN m:=m∪{f}; mk OF f:=0;
      IF IF 1 OF f IS nil THEN FALSE ELSE NOT MARKED 1 OF f FI
      THEN 2 MARK f; s:=1 OF f; push(t,f); f:=s
      ELIF IF r OF f IS nil THEN FALSE ELSE NOT MARKED r OF f FI
      THEN 1 MARK F; s:=r OF f; push(t,f); f:=s
      ELSE down:=FALSE
      FI
    ELSE IF IF r OF f IS nil THEN FALSE ELSE NOT MARKED r OF f FI
      THEN 1 MARK f; s:=r OF f; push(t,f); f:=s; down:=TRUE
      ELSE f:=pop(t)
      FI
    FI
  OD
END {m=R*(n)}
```

Fig. 4.8: algorithm lm7

A few notations introduced in algorithm lm7 need some explanation. The new assertion PointerCode1' will be defined below. A field mk is added as an auxiliary variable to every node. Two primitives are defined on this field: if a is an integer value and n a node then a MARK n is shorthand notation for mk OF n := mk OF n + a; and MARKED n is shorthand for mk OF n ∈ {0,... 3}. The proper Algol68 notation for 1 OF n ∈ m is rather unwieldy, it is close to the definition of relation R however.

The spanning tree defined by program lm7 divides the nodes in the list structure into four classes depending on the inclusion of their pointers in the spanning tree. If both pointers are are forward pointers then the node is called a FF-type node, if

only the left pointer is included it is a node of type FN, with the right pointer only it is of type NF, and if both pointers are not included it is of type NN. The mark field should code this and the assertion PointerCode1' defines how this is effected:

$$\begin{aligned}\text{PointerCode1'} \triangleq\ & \forall g\in m-\{f\}\,[\,g\in C(t)-\{f\}\rightarrow(\text{type}(g)\in\{FF,FN\}\rightarrow mk\text{ OF }g=2\wedge\\ & \text{type}(g)=NF\rightarrow mk\text{ OF }g=1\wedge\\ & \text{type}(g)=NN\rightarrow mk\text{ OF }g=0)\\ & \wedge g\in m-(C(t)\cup\{f\})\rightarrow(\text{type}(g)=FF\rightarrow mk\text{ OF }g=3\wedge\\ & \text{type}(g)=FN\rightarrow mk\text{ OF }g=2\wedge\\ & \text{type}(g)=NF\rightarrow mk\text{ OF }g=1\wedge\\ & \text{type}(g)=NN\rightarrow mk\text{ OF }g=0)\\ \neg down\rightarrow\ & (\text{type}(f)\in\{FF,FN\}\rightarrow mk\text{ OF }f=2\wedge\\ & \text{type}(f)=NF\rightarrow mk\text{ OF }f=1\wedge\\ & \text{type}(f)=NN\rightarrow mk\text{ OF }f=0)\end{aligned}$$

The verification of this new assertion is a straightforward application of the definition of the pointer types.

Algorithm lm7 contains twice the same action for going downward to the right. One of them (when down is true) will be followed by nodes of type NF, the other (when down is false) will be followed by nodes of type FF. The only difference is that down is set to true in the second case, which is superfluous in the first case. The transformation to remove this redundancy is the replacement of the first statement by a statement designed to guide control to the second statement, i.e. down:=FALSE. This statement and the next down:=FALSE can conveniently be contracted. Assertion PointerCode1' isn't valid any more however, since the treatment of type-NF nodes is changed. The new assertion PointerCode1 is given below:

$$\begin{aligned}\text{PointerCode1} \triangleq\ & \forall g\in m-\{f\}\,[\,g\in C(t)-\{f\}\rightarrow(\text{type}(g)\in\{FF,FN\}\rightarrow mk\text{ OF }g=2\wedge\\ & \text{type}(g)\in\{NF,NN\}\rightarrow mk\text{ OF }g=0)\\ & \wedge g\in m-(C(t)\cup\{f\})\rightarrow(\text{type}(g)=FF\rightarrow mk\text{ OF }g=3\wedge\\ & \text{type}(g)=FN\rightarrow mk\text{ OF }g=2\wedge\\ & \text{type}(g)=NF\rightarrow mk\text{ OF }g=1\wedge\\ & \text{type}(g)=NN\rightarrow mk\text{ OF }g=0)\\ \neg down\rightarrow\ & (\text{type}(f)\in\{FF,FN\}\rightarrow mk\text{ OF }f=2\wedge\\ & \text{type}(f)\in\{NF,NN\}\rightarrow mk\text{ OF }f=0)\end{aligned}$$

Verification is not difficult again. In effect the left forward pointers are followed when down is true and the right forward pointers are followed when down is false, so the left pointer is followed before the right one.

A note on termination is necessary again. The shift of this action from the branch when down is true to the branch when down is false has the following schematic form: first the algorithm was of the form:

$\{P'\}$ WHILE B DO IF down THEN T; IF B' THEN S ELSE S' FI
ELSE IF B' THEN S; down:=TRUE ELSE S" FI
OD $\{P'\}$

P' is the old loop-invariant and both B and B' do not contain down as a free variable. After the transformation the algorithm was of the form:

$\{P\}$ WHILE B DO IF down THEN T; IF B' THEN down:=FALSE ELSE S' FI
ELSE IF B' THEN S; down:=TRUE ELSE S" FI
OD $\{P\}$

with P the modified version of P'. If the changed loop was chosen in the first algorithm then the computation sequence is:

$\{P' \wedge down\}$ T; $\{P'' \wedge down \wedge B'\}$ S $\{P' \wedge down\}$

for a certain assertion P". And in the second algorithm it is:

$\{P \wedge down\}$ T; $\{P'' \wedge down \wedge B'\}$ down:=FALSE $\{P'' \wedge \neg down \wedge B'\}$

The next iteration will be:

$\{P'' \wedge \neg down \wedge B'\}$ S; down:=TRUE $\{P \wedge down\}$

Summing up the transformation amounts to the addition of the statement down:=FALSE before S and the addidion of down:=TRUE after S. Since the first statement only happens when down is true and since down is true only once for every node in the list-structure these two statements are added a finite number of times. Hence the program after the transformation terminates.

It is clear that if an odd mark is encountered when backing up (i.e. when down is false) then both pointers to siblings are

already examined, since the right one was followed in the traversal, and left pointers are examined before right ones. Then it is possible to immediately back up again. This observation makes it possible to distinguish between backing up from the left and backing up from the right. Algorithm lm8 (figure 4.9) shows the result of these transformations.

```
PROC lm8 =(REF NODE n)VOID:
BEGIN
  LOC REF NODE f, s;
  LOC NODESET m; LOC STACK t;
  f:=n; m:=∅; t:=nilst;
  down:=TRUE;
  {n∈m∪{f}∧ m∪{f}≤ R*(n)∧R(m-(C(t)∪{f}))≤m∪{f}∧C(t)≤m-{f}∧f∉m⇒down∧
   PointerCode1}
  WHILE f ISNT nil
  DO IF down
     THEN m:=m∪{f}; mk OF f:=0;
       IF IF 1 OF f IS nil THEN FALSE ELSE NOT MARKED 1 OF f FI
       THEN 2 MARK f; s:=1 OF f; push(t,f); f:=s
       ELSE down:=FALSE
       FI
     ELSE IF ODD mk OF f
       THEN f:=pop(t) ∢ up from right ∢
       ELSE IF IF r OF f IS nil THEN FALSE ELSE NOT MARKED r OF f FI
          THEN 1 MARK f; s:=r OF f; push(t,f); f:=s; down:=TRUE
          ELSE f:=pop(t)
            FI
       FI
     FI
  OD
END {m=R*(n)}
```

Fig. 4.9: algorithm lm8

One final transformation is necessary to arrive at the first list-traversal algorithm used by Robson. To fulfill the demand for bounded use of memory in the final list-copying algorithm the stack t has to be included in the original and copy structures. Robson's choice is the use of the original structure's pointers and nodes to implement stack t. The way it is coded is given in assertion SameStack, in which t is a stack and gf a pointer to a node:

```
SameStack(t,gf) ≙ if down then SameStack'(t,gf)
                  elif gf IS nil then t IS nilst
                  elif ODD mk OF gf then Samestack'(t, r OF gf)
                                    else Samestack'(t, l OF gf)
                  fi
SameStack'(t,gf)≙ if t IS nilst then gf IS nil
                  else gf IS pop(t) ∧ if ODD mk OF gf
                    then SameStack'(t', r OF gf)
                    else SameStack'(t', l OF gf) fi
                  fi
where t' is stack t after execution of pop(t)
```

Intuitively a node which is included in the stack is pointed to by one of its siblings. The pointer field normally pointing to the sibling in the stack on top of the father node is used as stack pointer. The old value is retrievable by simply remembering which node was the last to be popped off the stack. And the value of the mark allows identification of the stack pointer as either the left or the right pointer.

A basic technique for proving the correctness of graph algorithms is introduced in the next transformation. Since the old pointer fields are bound to be changed frequently during the algorithm a scratch-pad is needed to keep track of the original values. Therefore every node is given a virtual new field, an auxiliary field called a. Useful information for the proof can be stored in this field for every node. At the moment the original pointer fields are the only data necessary. Therefore the value of both pointers is stored in l OF a and r OF a at the moment a node is marked.

Then it is possible to describe the exact nature of the pointer to the node which was popped off the stack in the last loop. The new loop-invariant will include:

$$\begin{aligned}\text{LoopInv1} \triangleq\ & n \subseteq m \cup \{f\} \wedge m \cup \{f\} \subseteq R^*(n) \wedge R(m-(C(t) \cup \{f\})) \subseteq m \cup \{f\} \wedge C(t) \subseteq m-\{f\} \wedge \\ & f \notin m \leftrightarrow down \wedge \neg down \rightarrow (\text{ODD mk OF } f \rightarrow s=r \text{ OF a OF } f \wedge \\ & \qquad \text{EVEN mk OF } f \rightarrow s=l \text{ OF a OF } f)\end{aligned}$$

The consequence of mantaining this loop-invariant is that the else-branch in the case when down is true needs also modification, since down is set to false in this branch. Then the node f is considered to be popped lastly off the stack in the list structure. Since the mark of f is 0, the left pointer should still point to the stack and this pointer should be saved in s, which pointer is used as a note-book pointer for this purpose. The addition of the statements s:=l OF f; l OF f:=gf does the trick.

The transformed version of algorithm lm8 validating all these new assertions is algorithm lm9 (fig 4.10). All auxiliary statements are underscored, with a broken line if the statement concerns stack t, with a straight line otherwise. Note that the execution of algorithm lm8 depended heavily on the stack t, while the new algorithm lm9 will work without this stack. The verification of the new assertions is straightforward.


```
PROC lm9 =(REF NODE n)VOID:
BEGIN
  LOC REF NODE gf, f, s;
  LOC NODESET m; LOC STACK t;
  LOC BOOL down;
  f:=n; m:=∅; t:=nilst; gf:=nil;
  down:=TRUE;
  {LoopInv1∧PointerCode1∧SameStack(t,gf)}
  WHILE f ISNT nil
  DO IF down
    THEN m:=m∪{f}; mk OF f:=0; a:=(l OF f, r OF f);
      IF IF l OF f IS nil THEN FALSE ELSE NOT MARKED l OF f FI
      THEN 2 MARK f; s:=l OF f; push(t,f); l OF f:=gf; gf:=f; f:=s
      ELSE s:= l OF f; l OF f:=gf; down:=FALSE
      FI
    ELSE IF ODD mk OF f
      THEN gf:=r OF f; r OF f:=s; s:=f; f:=gf; f:=pop(t)
      ELSE gf:=l OF f; l OF f:=s;
        IF IF r OF f IS nil THEN FALSE ELSE NOT MARKED r OF f FI
        THEN 1 MARK f; s:=r OF f;
             push(t,f); r OF f:=gf; gf:=f; f:=s; down:=TRUE
        ELSE s:=f; f:=gf; f:=pop(t)
          FI
      FI
    FI
  OD
END {m=R*(n)∧PointerCode }
```

Fig. 4.9: Robson's first list-traversal algorithm

The first list-traversal of Robson's algorithm has another task besides the visit of every node without the change of the structure. This second task is to leave behind the information necessary for traversal in reverse order. The way this is done is stated in assertion PointerCode1. Since this assertion precisely states how the mark is changed for every node after every loop it contains too much information. The relevant part of pointercode1 is retained in the following assertion PointerCode:

$$\begin{aligned}\text{PointerCode} \triangleq\ & \forall g \in R^*(n) \rightarrow (\text{type}(g)=FF \leftrightarrow \text{mk OF } g=3 \wedge \\ & \qquad \text{type}(g)=FN \leftrightarrow \text{mk OF } g=2 \wedge \\ & \qquad \text{type}(g)=NF \leftrightarrow \text{mk OF } g=1 \wedge \\ & \qquad \text{type}(g)=FN \leftrightarrow \text{mk OF } g=0)\end{aligned}$$

Since this part of the loop-invariant doesn't contradict the termination condition of the loop it may be included in the post-assertion of the loop. This is also the post-assertion of the entire algorithm lm9. This concludes the correctness proof of the first list-marking algorithm of Robson.

Robson's second list-marking algorithm

The second traversal of the Robson list-copying algorithm is based on a special list-marking algorithm. The first traversal of Robson's algorithm defines a spanning tree of the list, and it stores information about the structure of this spanning tree in every node by marking every node in one of four ways. The marking code is given in assertion PointerCode in the previous section.

The second list-marking algorithm uses this information to traverse the list following the same spanning tree, albeit in the opposite direction. In a sense this algorithm is not a list-marking algorithm proper, since it doesn't work on every list structure but a prepared one.

To formalize this we define a relation S from $R^*(n)$ to $R^*(n)$ in the style of the definition of R. For nodes a, b$\in R^*(n)$:

$$a\ S\ b \ \underline{iff} \begin{array}{l} b=l \text{ OF } a \wedge type(a) \in \{FF, FN\} \\ b=r \text{ OF } a \wedge type(a) \in \{FF, NF\} \end{array} \vee$$

Following from the construction of the node-types S defines a spanning tree of the list-structure. Formally:

$$S^*(n)=R^*(n) \wedge \forall T \subseteq S\ T \neq S \rightarrow T^*(n) \neq R^*(n)$$

Fortunately a lot of work from the preceding section can be used in this section too. The starting point of the present derivation is algorithm lm6. In the assertions the relation R must be replaced by the relation S. This is valid because nothing is assumed about relation R except that it is a relation between nodes in memory. This is true for any subset of R of course.

It is possible again to postpone the statement f:=pop(t) in the case of down being <u>true</u> and S(f)$\subseteq$m. The same transformation was already proved correct in the preceding section.

Next concrete pointers have to be introduced, instead of the abstract relation S. This enables us to introduce the precondition PointerCode, which is fundamental to this algorithm. Another advantage is that it allows the same transformation already encountered between lm7 and lm8: one branch of the downward path is postponed since the same branch appears in the upward path. This time the algorithm is developed with the

postponement of the descent to the left. The correctness proof carries over again. Only small adaptions due to the changed roles of the pointers are necessary. The problem with the change of the assertion PointerCode1' to Pointercode1 doesn't appear here, since the mark fields aren't changed.

The big advantage of the mark field code is the easy determination of the necessity to descend to either left or right. The first advantage lies in the test if the mark field's value is odd. Then the algorithm should descend to the left. The test on descend to the right (the mark field should be greater than one) is not introduced yet. The reason is the certainty of the introduction of an infinite loop. A node is pushed on the stack and the search continues with its left sublist. When the node is popped off the stack again the mark is again two or three and another traversal of the left sublist would follow. The solution to this problem will be given in a moment; for the result of the other transformations mentioned see algorithm lm7' in figure 4.10.

The problem mentioned in the previous alinea will be solved next. A mark is necessary to signal if a possible descent to the left still must be made. Since it is not necessary to retain information about the node type if a descent need not be made any more the old mark field can be used. When it is checked if a descent to the left has to be made the mark is erased by substituting -1 for the old mark code. The following operator is clear then:

OP MARKED =(NODE n)BOOL: ($n \in \{0,1,2,3\}$);

The assertion PointerCode is not valid if the mark field is erased. So a new assertion PointerCode2 is necessary:

$$\begin{array}{l} PointerCode2 \triangleq \forall g \in S^*(n)\ (mk \text{ OF } g=3 \rightarrow type(g)=FF \wedge \\ \qquad mk \text{ OF } g=2 \rightarrow type(g)=FN \wedge \\ \qquad mk \text{ OF } g=1 \rightarrow type(g)=NF \wedge \\ \qquad mk \text{ OF } g=0 \rightarrow type(g)=NN \wedge \\ \qquad \neg MARKED\ g \rightarrow S(g) \subseteq m) \end{array}$$

After these preparations it is possible to replace the else-clause in the main loop (i.e. the case when down is <u>false</u>)

```
PROC lm7'=(REF NODE n)VOID:
BEGIN {PointerCode}
  LOC REF NODE f, s;
  LOC NODESET m; LOC STACK t;
  LOC BOOL down;
  f:=n; m:=∅; t:=nilst;
  down:=TRUE;
  {n∈m∪{f}∧m∪{f}⊆S*(n)∧
   S(m-(C(t)∪{f}))⊆m∪{f}∧
   C(t)⊆m-{f}∧f∈m⇒down∧
   PointerCode}
  WHILE f ISNT nil
  DO IF down
    THEN m:=m∪{f};
      IF ODD mk OF f;
      THEN s:=r OF f;
        push(t,f); f:=s
      ELSE down:=FALSE
      FI
    ELSE IF NOT S(f)⊆m
      THEN s:=1 OF f;
        push(t,f); f:=s;
        down:=TRUE
      ELSE f:=pop(t)
      FI
    FI
  OD
END {m=S*(n)}
```

Fig. 4.10: algorithm lm7'

```
PROC lm8'=(REF NODE n)VOID:
BEGIN {PointerCode}
  LOC REF NODE gf, f, s, s1;
  LOC NODESET m; LOC STACK t;
  LOC BOOL down; LOC INT mark;
  f:=n; m:=∅; t:=nilst; gf:=nil;
  down:=TRUE;
  {LoopInv2∧PointerCode2∧
   SameStack2(t,gf)}
  WHILE f ISNT nil
  DO IF down
    THEN m:=m∪{f}; a:=(1 OF f, r OF f);
      IF ODD mk OF f
      THEN s:=r OF f;
        r OF f:=gf; gf:=f; push(t,f);
        f:=s
      ELSE s1:=r OF f; r OF f:=gf;
        down:=FALSE
      FI
    ELSE IF MARKED f
      THEN mark:=mk OF f; mk OF f:=-1;
        IF mark≥2
        THEN s:=1 OF f;
          1 OF f:=r OF f; r OF f:=s1;
          gf:=f; push(t,f); f:=s;
          down:=TRUE
        ELSE gf:=r OF f; r OF f:=s1;
          s1:=f; f:=gf; f:=pop(t)
        FI
      ELSE gf:=1 OF f; 1 OF f:=s1;
        s1:=f; f:=gf; f:=pop(t)
      FI
    FI
  OD
END {m=S*(n)}
```

Fig. 4.11: Robson's second list-traversal algorithm

by the following statement (mark is a fresh integer variable):

```
ELSE IF MARKED f
  THEN mark:=mk OF f; mk OF f:=-1;
    IF mark≥2
    THEN s:=1 OF f; push(t,f); f:=s; down:=TRUE
    ELSE f:=pop(t)
    FI
  ELSE f:=pop(t)
FI
```

Verification of the old loop-invariant's preservation with assertion PointerCode replaced by assertion PointerCode2 is straightforward.

The last remaining transformation is again a mirror image of a transformation in the derivation of the first list-marking algorithm. The stack t has to be included in the list structure. To describe the implementation an assertion SameStack2 is defined recursively (t is a stack and gf a pointer to a node):

```
SameStack2(t,gf) ≙ if down then SameStack"(t,gf)
                   elif gf IS nil then t IS nilst
                   elif MARKED gf then SameStack"(t',r OF gf)
                                  else SameStack"(t',1 OF gf)
                   fi

SameStack"(t,gf) ≙ if t IS nilst then gf IS nil
                   else gf IS pop(t)∧ if MARKED gf
                     then SameStack"(t', r OF gf)
                     else SameStack"(t', 1 OF gf) fi
                   fi
```

where t' is stack t after execution of pop(t)

Again the pointer of the sibling right on top of a node in the stack is chosen for the stack pointer. This sibling is saved again in a scratch-pad variable, this time s1. To be able to reason about the contents of s1 an auxiliary field is assigned again to every node in the list structure; this field contains the original pointers of the matching node. The loop-invariant then includes:

```
LoopInv2 ≙ n∈m∪{f}∧m⊂{f}⊆S*(n)∧S(m-(C(t)∪{f}))⊆m∪{f}∧C(t)⊆m∪{f}∧
           f∉m↔down∧¬down→(MARKED f→s1=r OF a OF f ∧
                           NOT MARKED f→s1=1 OF a OF f)
```

The stage is set again for the final transformation series, the introduction of the implicit stack. The result, Robson's second list-marking algorithm, is shown in figure 4.11. The auxiliary variables are underscored again. Statements referring to stack t are underscored with a broken line, other auxiliary statements with a straight line. The verification of the loop-invariant contains no special problems.

The post-assertion m=S*(n) states that all nodes in the spanning tree defined by S are coded. According to the definition of relation S S*(n)=R*(n), so lm8' correctly marks the list defined by root n and relation R. This concludes the proof of Robson's second list-marking algorithm.

The derivation of Robson's list-copying algorithm

The Robson list-copying algorithm uses two traversals of a list structure to be copied. In the first stage a copy node is assigned to every node in the original list structure. The pointers in every original node are placed in the corresponding copy node. This results in the redundancy of the information in the pointer fields of the original node. These fields will be used cleverly to store information not otherwise available. The left pointer field will contain the reference to the corresponding copy node. The right pointer field will be used to mark the node type in the spanning tree defined by algorithm lm9.

The second stage restores the old pointer fields in every original node while determining the corresponding pointers in the copy structure. This has to be done in a very careful manner, since removal of any left pointer field in the original structure will result in the loss of the knowledge which copy node was assigned to the node containing this field, and removal of the right pointer field results in the loss of the calculated information about the spanning tree in the node considered.

To be able to reason about the nodes in both original and copy structures the auxiliary field in every node will appear again. Ultimately its role will be the basic role of an auxiliary variable: to appear in the correctness proof. Upon introduction it will be used to define the connection between the original node and its assigned copy. The information stored in this field is a reference to the matching original node, to the copy node, and to the left and right siblings of the original node. These fields will be called on, cn, l and r in the same order.

The first step towards the final copying algorithm is algorithm rlc' (figure 4.12). A pointer c is included which will point to the copy node corresponding to the node f in the original structure. The pointer copy will point to the copy of node n, the root of the copy structure. The meaning of node s2 will be explained below.

The first loop is essentially the old algorithm lm9. When a node is marked a copy node is made and the old left and right pointers are written in the pointer field of this new node. The

auxiliary field is extended as described. The changes in the values of the pointer fields in the original structure is echoed in the changes in the pointer fields of the copy structure.

The second loop is preceded by the initialization statements for the various control variables. This is also the time to set the pointer copy to its final value: the root of the copy structure, which is naturally the copy of the root of the original structure. The values of the pointers in the copy structure have to be set to their final value in this loop. One has to remember that the information in the pointer fields of the copy structure will not be available in duplo in the final algorithm. So a pointer in the copy structure may only be replaced by its final value at the moment its temporary value isn't necessary for the remainder of the traversal.

A new pointer s2 is introduced to point to the copy node corresponding to s1. Algorithm lm8 used pointer s1 to point to the subtree which was just traversed. So if the copy pointers in the copy nodes corresponding to the nodes in the subtree with s1 as root are correct, then by induction on the structure of the tree the correct copy of the pointer to s1 in the original structure is the pointer to s2. Some small problems have to be circumvented. In the case of down being true s1 might be set to nil, a "node" without fields. The solution is an assignment to s2 of the value nil if s1 was set to nil and of the copy of s1 otherwise.

An Algol68 procedure expects a statement giving the value of the result as the last statement. In this case the root of the copy structure is given by pointer copy. This algorithm copies any list structure in linear time - both loops visit every node a maximum of three times - while using extra memory space proportional to the number of nodes - each node in the original structure uses a field for the mark and a field for the pointer to its assigned copy -. The next transformations are aimed at the removal of the need for extra memory.

First the pointers from every original node to its copy node are included in the original left pointer fields. A close look reveals several steps in the derivation necessary from algorithm rlc'. First control of the traversal sequence has to be changed from the pointers in the original nodes to the pointers in the

```
PROC rlc'=(REF NODE  n)REF NODE:
BEGIN
  LOC REF NODE gf, f, s, s1, s2, c, copy;
  LOC NODESET m;
  LOC BOOL down; LOC INT mark;
  f:=n; m:=∅; gf:=nil; down:=TRUE;
  {LoopInvListMark1' ∧ LoopInvListCopy1'}
  WHILE f ISNT nil
  DO IF down
    THEN m:=m∪{f}; c:= HEAP NODE:=(1 OF f, r OF f); mk OF f:=0;
      a OF f:=(f, c, 1 OF f, r OF f);
      IF IF 1 OF f IS nil THEN FALSE ELSE NOT MARKED 1 OF f FI
      THEN 2 MARK f; s:=1 OF f; 1 OF f:=gf; 1 OF c:=gf; gf:=f; f:=s
      ELSE s:=1 OF f; 1 OF f:=gf; 1 OF c:=gf; down:=FALSE
      FI
    ELSE c:=cn OF a OF f;
      IF ODD mk OF f
      THEN gf:=r OF f; r OF f:=s; r OF c:=s; s:=f; f:=gf
      ELSE gf:=1 OF f; 1 OF f:=s; 1 OF c:=s;
        IF IF r OF f IS nil THEN FALSE ELSE NOT MARKED r OF f FI
        THEN 1 MARK f; s:=r OF f; r OF f:=gf; r OF c:=gf;
          gf:=f; f:=s; down:=TRUE
        ELSE s:=f; f:=gf
        FI
      FI
    FI
  OD; {PostListCopy1'}
  f:=n; m:=∅; gf:=nil; down:=TRUE; copy:=cn OF a OF n;
  {LoopInvListMark2'∧  LoopInvListCopy2'}
  WHILE f ISNT nil
  DO c:=cn OF a OF f;
    IF down
    THEN m:=m∪{f};
      IF ODD mk OF f
      THEN s:=r OF f; r OF f:=gf; r OF c:=gf; gf:=f; f:=s
      ELSE s1:=r OF f; s2:=IF s1 IS nil THEN nil
        ELSE cn OF a OF s1 FI; r OF f:=gf; r OF c:=gf; down:=FALSE
      FI
    ELSE IF MARKED f
      THEN mark:=mk OF f; mk OF f:=-1;
        IF mark≥2
        THEN s:=1 OF f; 1 OF f:=r OF f; 1 OF c:=r OF c;
           r OF f:=s1; r OF c:=s2; gf:=f; f:=s; down:=TRUE
        ELSE gf:=r OF f; r OF f:=s1; r OF c:=s2;
           s1:=f; s2:=c; f:=gf
        FI
      ELSE gf:=1 OF f; 1 OF f:=s1; 1 OF c:=s2;
        s1:=f; s2:=c; f:=gf
      FI
    FI
  OD;
  copy ∉ the result ∉
END {PostListCopy2'}
```

Fig. 4.12: algorithm rlc'

The various assertions are deacribed in figure 4.15.

matching copy nodes. This poses no difficulties, since in the preceding algorithm all pointers in the copy are equal to the pointers in the original. The information about pointers in the copy is destroyed in the second loop after return from the branch pointed to, so no real loss is incurred: the information isn't necessary in the remainder of the algorithm.

Then the left pointer in the original is set to the copy node. Assignations to both the left and the right pointer in the original structure are deleted, since the assignations to the pointers in the copy structure suffice. There is one exception to this rule. When in the second loop the pointer to the original subtree is restored while backing up the assignment to the pointers in the original is retained, though this is superfluous in the case of the assignment to the right pointer, which was not changed. Now it is possible to decide at what moment precisely the pointer fields in the original are no longer available to store additional information: when one returns from the descent in the subgraph originally pointed to by the node to be replaced. It should be noted that information about the corresponding copy nodes is no longer necessary in pass two in a traversed subtree of the spanning tree defined in pass one. The remainder of the original graph may contain back pointers to this subtree. A back pointer is defined as a pointer to a node already marked before in pass 1. Since pass 2 uses the opposite traversal order no nodes with back pointers to a particular node can be encountered if the node was passed for the last time; they must be encountered earlier. This observation is the salient reason the Robson list-copying algorithm can work.

To be able to retain the forward pointer until the last possible moment it is necessary to observe that the back-up phase in the case of f being marked with 0 or 1 includes a "back-up" from the left. Though a descent to the left isn't made since the left pointer was either a back pointer or a nil value, it should be restored in the left pointer field. Again the problem of the copy of a nil value has to be included.

Lastly all references to the copy node assigned to a certain original node have to be replaced by the direct reference via the left pointer field. Then the connection between original and

```
PROC rlc"=(REF NODE n)REF NODE:
BEGIN
  LOC REF NODE gf, f, s, s1, s2, c, copy;
  LOC NODESET m;
  LOC BOOL down; LOC INT mark;
  f:=n; m:=∅; gf:=nil; down:=TRUE;
  {LoopInvListMark1" ∧ LoopInvListCopy1"}
  WHILE f ISNT nil
  DO IF down
    THEN m:=m∪{f}; c:=HEAP NODE:=(1 OF f, r OF f); mk OF f:=0;
      a OF f:=(f, c, 1 OF f, r OF f); 1 OF f:=c;
      IF IF 1 OF c IS nil THEN FALSE ELSE NOT MARKED 1 OF c FI
      THEN 2 MARK f; s:=1 OF c; 1 OF c:=gf; gf:=f; f:=s
      ELSE s:=1 OF c; 1 OF c:=gf; down:=FALSE
      FI
    ELSE c:=1 OF f;
      IF ODD mk OF f
      THEN gf:=r OF c; r OF c:=s; s:=f; f:=gf
      ELSE gf:=1 OF c; 1 OF c:=s;
        IF IF r OF c IS nil THEN FALSE ELSE NOT MARKED r OF c FI
        THEN 1 MARK f; s:=r OF c; r OF c:=gf; gf:=f; f:=s;
          down:=TRUE
        ELSE s:=f; f:=gf
        FI
      FI
    FI
  OD; {PostListCopy1"}
  f:=n; m:=∅; gf:=nil; down:=TRUE; copy:=1 OF n;
  {LoopInvListMark2" ∧ LoopInvListCopy2"}
  WHILE f ISNT nil
  DO c:=1 OF f;
    IF down
    THEN m:=m∪{f};
      IF ODD mk OF f
      THEN s:=r OF c; r OF c:=gf; gf:=f; f:=s
      ELSE s1:=r OF c; s2:=IF s1 IS nil THEN nil ELSE 1 OF s1 FI;
        r OF c:=gf; down:=FALSE
      FI
    ELSE IF MARKED f
      THEN mark:=mk OF f; mk OF f:=-1;
        IF mark>2
        THEN s:=1 OF c; 1 OF c:=r OF c;
            r OF f:=s1; r OF c:=s2; gf:=f; f:=s; down:=TRUE
        ELSE gf:=r OF c; r OF f:=s1; r OF c:=s2; s:=1 OF c;
          IF s IS nil THEN 1 OF f:=nil
          ELSE 1 OF c:= 1 OF s; 1 OF f:=s FI;
          s1:=f; s2:=c; f:=gf
        FI
      ELSE gf:=1 OF c; 1 OF f:=s1; 1 OF c:=s2;
        s1:=f; s2:=c; f:=gf
      FI
    FI
  OD;
  copy
END {PostListCopy2"}
```

Fig. 4.13: algorithm rlc"

The various assertions are described in figure 4.15.

copy node in the auxiliary field is reduced to its final role: an auxiliary variable. The resulting algorithm is listed in figure 4.13 as algorithm rlc".

The final memory saving technique is the treatment of the marks. The right pointer field will be used to store a pointer to a node from a special array mk. The following declarations will implement the handling of the marks correctly.

```
LOC [0:3] REF NODE mk;
PROC mrk =(NODE n)INT:
BEGIN
  LOC INT i; i:=0;
  TO 4 WHILE r OF n ISNT mk[i]
  DO i:=i+1 OD;
  i
END;
OP MARKED =(NODE n)BOOL:
mrk(n)<4;
OP MARK =(INT i, NODE n)VOID:
r OF n:=mk[mrk(n)+i];
```

The structure of these declarations is as follows. A row of pointers to nodes is declared. The retrieval of the value of the mark field is accomplished by simply testing all possible values. If a node is not marked the value 4 will be returned. So operation MARKED tests on the result of the mrk procedure. The final operation MARK advances the right pointer of node n i spaces along the row of marks. A correctness proof of this implementation can easily be given in the style of Jones [J].

The introduction of this implementation is given in algorithm rlc (figure 4.14). On the surface it actually amounts to only three changes, since most of the changes are implemented via the two operators MARK and MARKED. The initialization of the mark field mk OF f:=0 is replaced by an assignation to the right pointer field of f. All tests on mk OF f are replaced by tests on mrk(f). And the statement erasing the mark mk OF f:=-1 is replaced by r OF f:=s1, a statement moved out of the following case-clause. The correctness proof poses no special problems.

Deletion of all underscored statements in algorithm rlc - the statements dealing with auxiliary variables - results in Robson's list-copying algorithm proper.

```
PROC rlc =(REF NODE n)REF NODE:
BEGIN
  LOC REF NODE gf, f, s, s1, s2, c, copy;
  LOC NODESET m;
  LOC BOOL down; LOC INT mark;
  f:=n; m:=∅; gf:=nil; down:=TRUE;
  {LoopInvListMark1 ∧ LoopInvListCopy1}
  WHILE f ISNT nil
  DO IF down
    THEN m:=m_f; c:=HEAP NODE:=(1 OF f, r OF f); r OF f:=mk[0];
      a OF f:=(f, c, l OF f, r OF c); l OF f:=c;
      IF IF l OF c IS nil THEN FALSE ELSE NOT MARKED l OF c FI
      THEN 2 MARK f; s:=l OF c; l OF c:=gf; gf:=f; f:=s
      ELSE s:=l OF c; l OF c:=gf; down:=FALSE
      FI
    ELSE c:=l OF f;
      IF ODD mrk(f)
      THEN gf:=r OF c; r OF c:=s; s:=f; f:=gf
      ELSE gf:=l OF c; l OF c:=s;
        IF IF r OF c IS nil THEN FALSE ELSE NOT MARKED r OF c FI
        THEN 1 MARK f; s:=r OF c; r OF c:=gf; gf:=f; f:=s;
          down:=TRUE
        ELSE s:=f; f:=gf
        FI
      FI
    FI
  OD; {PostListCopy1}
  f:=n; m:=∅; gf:=nil; down:=TRUE; copy:=l OF n;
  {LoopInvListMark2 ∧ LoopInvListCopy2}
  WHILE f ISNT nil
  DO c:=l OF f;
    IF down
    THEN m:=m∪{f};
      IF ODD mrk(f)
      THEN s:= r OF c; r OF c:=gf; gf:=f; f:=s
      ELSE s1:=r OF c; s2:=IF s1 IS nil THEN nil ELSE l OF s1 FI;
        r OF c:=gf; down:=FALSE
      FI
    ELSE IF MARKED f
      THEN mark:=mrk(f); r OF f:=s1;
        IF mark>2
        THEN s:=l OF c; l OF c:=r OF c;
          r Of c:=s2; gf:=f; f:=s; down:=TRUE
        ELSE gf:=r OF c; r OF c:=s2; s:=l OF c;
          IF s IS nil THEN l OF f:=nil
          ELSE l OF c:=l OF s; l OF f:=s FI;
          s1:=f; s2:=c; f:=gf
        FI
      ELSE gf:=l OF c; l Of f:=s1; l Of c:=s2;
        s1:=f; s2:=c; f:=gf
      FI
    FI
  OD;
  copy
END {PostListCopy2}
```

Fig. 4.14: Robson's List-copying algorithm

The various assertions are explained in figure 4.15


```
LoopInvListMark1' ≜ ∀t (SameStack(t,gf)→(LoopInv1∧PointerCode1))
LoopInvListCopy1' ≜ ∀g∉m (l OF g=l OF cn OF a OF g∧
                          r OF g=r OF cn OF a OF g)
PostListCopy1'    ≜ ∀g∈R*(n) (l OF g=l OF cn OF a OF g∧
                              r OF g=r OF cn OF a OF g∧
                              PointerCode)
LoopInvListMark2' ≜ ∀t (SameStack2(t,gf)→(LoopInv2∧PointerCode2))
LoopInvListCopy2' ≜ ∀g∈R*(n) [g∉m→(l OF g=l OF cn OF a OF g∧
                                   r OF g=r OF cn OF a OF g)
                              ∧[g∈m∧MARKED g]→(l OF g=l OF cn OF a OF g∧
                                              r OF g=r OF cn OF a OF g∧
                                              {g=f∧¬down}→s2=copy(s1))
                              ∧[g∈m∧NOT MARKED g]→
                                   r OF cn OF a OF g=copy(r OF g)∧
                                   ∀t [SameStack2(t,gf)→
                                       {g∈C(t)}→l OF g=l OF cn OF a OF g}∧
                                       {{g=f∧¬down}→s2=copy(s1)}∧
                                       {{g∉C(t)∨{f}}→l OF cn OF a OF g=
                                                     copy(l OF g)}]]
PostListCopy2'    ≜ ∀g∈R*(n)[l OF cn OF a OF g=copy(l OF g)∧
                             r OF cn OF a OF g=copy(r OF g)]

copy(n)≜if n IS nil then nil elif atom(n) then n else cn OF a OF n fi

LoopInvListMark1" ≜ ∀t (SameStackA(t,gf)→(LoopInv1∧PointerCode1))
LoopInvListCopy1" ≜ ∀g∉m (l OF g=cn OF a OF g∧
                          r OF g=r OF cn OF a OF g)
PostListCopy1"    ≜ ∀g∈R*(n) (l OF g=cn OF a OF g∧
                              r OF g=r OF cn OF a OF g∧
                              PointerCode)
LoopInvListMark2" ≜ ∀t (SameStackB(t,gf)→(LoopInv2∧PointerCode2))
LoopInvListCopy2" ≜ ∀g∈R*(n) [g∉m→(l OF g=cn OF a OF g∧
                                   r OF g=r OF cn OF a OF g)
                              [g∈m∧MARKED g]→(l OF g=cn OF a OF g∧
                                              r OF g=r OF cn OF a OF g∧
                                              {g=f∧¬down}→s2=copy(s1))
                              [g∈m∧NOT MARKED g]→
                                   r OF cn OF a OF g=copy(r OF g)∧
                                   ∀t [SameStackB(t,gf)→
                                       {g∈C(t)}→l OF g=cn OF a OF g}∧
                                       {{g=f∧¬down}→s2=copy(s1)}∧
                                       {{g∉C(t)∨{f}}→l OF cn OF a OF g=
                                                     copy(l OF g)}]]
PostListCopy2"    ≜ PostListCopy2

SameStackA≜SameStack [r OF cn OF gf/r OF gf, l OF cn OF gf/l OF gf]
SameStackB≜SameStack2[r OF cn OF gf/r OF gf, l OF cn OF gf/l OF gf]
```

```
LoopInvListMark1 ≙ LoopInvListMark1"
LoopInvListCopy1 ≙ ∀g∈m (1 OF g=cn OF a OF g ∧ mrk(g)=mk OF g)
PostListCopy1    ≙ PostListCopy1"∧∀g∈R*(n)(mrk(g)=mk OF g)
LoopInvListMark2 = loopInvListMark2"
LoopInvListCopy2 = ∀g∈R*(n) [ g∉m→(1 OF g=cn OF a OF g
                                   mrk(g)=mk OF g)
                            ∧[g∈m∧MARKED g]→(1 OF g=cn OF a OF g∧
                                              mrk(g)=mk OF g∧
                                              {g=f∧¬down}⇒s2=copy(s1))
                            ∧[g∈m∧NOT MARKED g]→
                                  r OF cn OF a OF g=copy(r OF g)∧
                                  ∀t [SameStackB(t,gf)→
                                     {g∈C(t)⇒1 OF g=cn OF a OF g}∧
                                     {{g=f∧¬down}⇒s2=copy(s1)}∧
                                     {g∉C(t)∨{f}1 OF cn OF a OF g=
                                                 copy(1 OF g)}]]
PostListCopy2    = PostListCopy2"
```

Fig. 4.15: Assertions from figures 4.12, 4.13 and 4.14

§ 5: FISHER'S LIST-COPYING ALGORITHM

Introduction

Historically the first list-copying algorithm using linear time and bounded workspace, Fisher's algorithm was a leap forward compared with some algorithms by Lindstrom [L']. Lindstroms algorithms could reach linear time only in the case of a structure without cycles, and that at the expense of a tag-bit. Bounded workspace was attainable, however at the cost of order N^2 time, where N is the number of nodes to be copied. The only constraint was on the location of the copy structure in the memory: it should be placed in a contiguous block. A reasonable price to pay.

Since then Fisher's algorithm has been "beaten" twice. By Robson, who lifted the constraint on the location of the copy. And by Clark, who devised an even faster algorithm. Still Fisher's list-copying algorithm is one of the more complex and difficult to understand algorithms around. As illustrated in § 2 even the author himself went astray in the informal introduction. The derivation of Fisher's algorithm will be presented more brief than the derivations of the other two algorithms in this paper.

Fisher's list-copying algorithm traverses the list structure three times. The first two traversals are based on a list-marking algorithm of an unusual structure. The third traversal makes use of the array of copied nodes to retrace the reverse order of allocation of the copy nodes. The list-marking algorithm will be derived in the next section; the array structure will be treated in an abstract way in the derivation of the final list-copying algorithm.

Fisher's list-marking algorithm

For the derivation of the list-marking algorithm used by Fisher some use can be made of the work in the preceding paragraph. The present starting point is algorithm lm2 (figure 4.3). First transformation rule L2 will be used to move the statement m:=m∪{f} into the main loop. This requires some preparation, however. A boolean variable finished with obvious meaning is introduced. Its initialization value is false, of course. Then the else-clause f:=x.x∉b; b:=b-{f} is replaced by If b≠∅ THEN finished:=TRUE ELSE f:=x.x∉b; b:=b-{f} FI. Since the loop-

```
PROC lm3f=(REF NODE n)VOID:
BEGIN
  LOC REF NODE f, s;
  LOC NODESET m, b;
  LOC BOOL finished;
  f:=n; m:=∅; b:=∅;
  finished:=FALSE;
  {n∈m∪{f} ∧ m∪{f}⊆R*(n) ∧
   R(m-(b∪{f}))⊆m∪{f} ∧ b⊆m-{f} ∧
   finished → m=R*(n)}
  WHILE NOT finished
  DO m:=m∪{f};
    IF NOT R(f)⊆m
    THEN s:=x.x∈R(f)-m;
      b:=b∪{f}; f:=s
    ELSE IF b=∅
      THEN finished:=TRUE
      ELSE f:=x.x∈b; b:=b-{f}
      FI
    FI
  OD
END {m=R*(n)}
```

Fig. 5.1: algorithm lm3f

```
PROC lm4f=(REF NODE n)VOID:
BEGIN
  LOC REF NODE f, s;
  LOC NODESET m, b;
  LOC BOOL finished;
  f:=n; m:=∅; b:=∅;
  finished:=FALSE;
  {n∈m∪{f} ∧ m∪{f}⊆R*(n) ∧
   R(m-(b∪{f}))⊆m∪{f} ∧ b⊆m-{f} ∧
   finished → m=R*(n)}
  WHILE NOT finished
  DO WHILE m:=m∪{f};
       NOT R(f)⊆m
    DO s:=x.x∈R(f)-m;
      b:=b∪{f}; f:=s
    OD;
    WHILE IF b=∅
      THEN finished:=TRUE
      ELSE f:=x.x∈b; b:=b-{f}
      FI;
      IF finished THEN FALSE
      ELSE R(f)⊆m FI
    DO SKIP OD
  OD
END {m=R*(n)}
```

Fig. 5.2: algorithm lm4f

test's validity followed by the execution of $m:=m\cup\{f\}$ and the negation of the case-test ensure that $b\neq\emptyset$ this amounts to the execution of the old statement every time the new statement is executed. The execution of the new body of the loop while $b=\emptyset$ and $R(f)\subseteq m\cup\{f\}$ results in the execution of the statements $m:=m\cup\{f\}$ followed by finished:=TRUE.

Thus it is shown that finished cannot assume the value true unless $b=\emptyset$ and $R(f)\subseteq m\cup\{f\}$, and that finished will be true if the new loop-body is executed while $b=\emptyset$ and $R(f)\subseteq m\cup\{f\}$. Since the validity of the post-condition is independent of the execution of the case-clause - m, R and n are not changed in this clause - transformation rule L2 can be applied, with the test NOT finished as the new loop-test. Since it is shown to be impossible for finished to be true unless the loop terminates the post-condition will be valid when finished is true. Thus the assertion finished $\rightarrow m=R^*(n)$ can be included in the loop-invariant. The final algorithm lm3f is shown in figure 5.1.

If the statement $m:=m\cup\{f\}$ is moved into the statements of the then- and else-cases, instead of the position in front of the case-clause, the program is in the form necessary for the

application of transformation rule L3. The test $R(f)\subseteq m$ has to be adapted to $R(f)\subseteq m\cup\{f\}$, of course. After application of transformation rule L3 the interior of the while-loop splits into two loops. The first one is WHILE NOT(finished $\vee$ $R(f)\subseteq m\cup\{f\}$) DO $m:=m\cup\{f\}$; $s:=x.x\in R(f)-m$; $b:=b\cup\{f\}$; $f:=s$ OD. Since finished is false when this statement is executed for the first time, and since finished is not changed in this statement the loop-test reduces to the second component.

The second interior loop is WHILE $m:=m\cup\{f\}$; IF $b=\emptyset$ THEN finished:=TRUE ELSE $f:=x.x\in b$; $b:=b-\{f\}$ FI; NOT finished $\wedge$ $R(f)\subseteq m\cup\{f\}$ DO SKIP OD. Since $f\in m$ after execution of $f:=x.x\in b$ the last test is equivalent to $R(f)\subseteq m$. And $m:=m\cup\{f\}$ need only be executed once, so it can be moved out of the loop. After this the statement can be advanced in the first loop to the place before the loop-test (and this test has to be changed back again). The correctness proof of all these changes is lengthy but straightforward and therefore omitted. The resulting algorithm lm4f is shown in figure 5.2. Note the change of the test of the second loop to an equivalent formula.

Next the observation is made that the lists treated are a special kind of graph structure. Every node has a maximum of two edges starting in it, and these edges are designated as left- and right-pointer. Then it is possible to replace the abstract test $R(f)\subseteq m$ by the tests on the inclusion of either node pointed to in the set m. The right pointer will be tested first, so if a node is in the boundary set its right sibling is marked. Then it is not necessary to investigate any other siblings if a descent to the left sibling is made. To make use of this observation all descents to the left are made in the second loop. This can be done by postponement of the statements concerning descent to the left in the first loop. Postponement will occur a maximum of one time per node, since every node has only one left pointer. Hence termination is assured.

Then it is now known after the second while-loop that f has an unmarked left sibling, unless finished is true. Since all right siblings are investigated after left siblings f need not be included in set b to preserve the loop-invariant. Then the loop-invariant is also validated when control immediately switches to the unmarked sibling. The result of these transformations is shown in algorithm lm5f (figure 5.3).

The final touch is the implementation of set b as a still

```
PROC lm5f=(REF NODE n)VOID:
BEGIN
  LOC REF NODE f, s;
  LOC NODESET m, b;
  LOC BOOL finished;
  f:=n; m:=∅; b:=∅;
  finished:=FALSE;
  {n∈m∪{f}∧m∪{f}⊆R*(n)∧
   R(m-(b∪{f}))⊆m∪{f}∧b∪m-{f}∧
   finished → m=R*(n)}
  WHILE NOT finished
  DO WHILE m:=m∪{f};
       r OF f∉m
     DO b:=b∪{f}; f:=r OF f OD;
     WHILE IF b=∅
       THEN finished:=TRUE
       ELSE f:=x.x∈b; b:=b-{f}
       FI;
       IF finished THEN FALSE
       ELSE 1 OF f∈m FI
     DO SKIP OD;
     IF NOT finished
     THEN f:=1 OF f FI
  OD
END {m=R*(n)}
```

Fig. 5.3: algorithm lm5f

```
PROC lm6f=(REF NODE n)VOID:
BEGIN
  LOC REF NODE f;
  LOC NODESET m; LOC QUEUE q;
  LOC BOOL finished;
  f:=n; m:=∅; q:=nilq;
  finished:=FALSE
  {n∈m∪{f}∧m∪{f}⊆R*(n)∧
   R(m-(C(q)∪{f}))⊆m∪{f}∧C(q)⊆m-{f}∧
   finished → m=R*(n)}
  WHILE NOT finished
  DO WHILE m:=m∪{f};
       r OF f∉m
     DO enq(q,f); f:=r OF f OD;
     WHILE IF q=nilq
       THEN finished:=TRUE
       ELSE f:=deq(q)
       FI;
       IF finished THEN FALSE
       ELSE 1 OF f∈m FI
     DO SKIP OD;
     IF NOT finished
     THEN f:=1 OF f FI
  OD
END {m=R*(n)}
```

Fig. 5.4: algorithm lm6f

abstract queue q. The correctness of the queue-operations enq and deq will be taken for granted at the moment: a verification is necessary at the implementation level in the correctness proof of the marking algorithm proper. It cannot be done at an earlier stage, since the implementation will depend on the structure of the nodes of the final copy list.

No problems are encountered here either if the correctness proof is done in the style of Jones [J]. For use in the assertions an equivalent to the set b has to be defined along the lines of the definition of the contents of a stack encountered in the previous paragraph. Thus the contents of queue q, C(q) is defined as:

$$C(q) \triangleq \begin{cases} \emptyset & \text{if } q=nilq \\ \{deq(q)\} \cup C(q') & \text{if } q \neq nilq \text{ and } q' \text{ is queue } q \text{ after the execution of } deq(q) \end{cases}$$

The resulting algorithm lm6f (figure 5.4) is the starting point for the derivation of the first two list-traversals in Fisher's copying algorithm. The third traversal via a stack will depend on the abstract stack-operators introduced.

Derivation of Fisher's List-copying algorithm

Fisher's list-copying algorithm demands the placement of the copy structure sequentially in a contiguous block of the available memory locations. Since organization of the structure of various memory allocators is beyond the scope of this text the various properties following from this demand used will be treated abstractly. Also no allowance will be made for the possibility of running out of memory. The memory space allocated is supposed to be sufficient.

The primitives used are the availability of a zero location Mem, a function next which gives the node directly following the argument node in the copy part of memory, a function prev, which gives the previous node in the copy part of memory, and a function iscopy, which states wether a node is in the copy part of memory (and hence a copy node). So next(Mem) is the first available location in the copy part of memory.

Then procedure flc' (figure 5.5), the starting point of the derivation, can be introduced. Pointers c and c1 will point to various nodes, mainly in the copy structure. Pointer copy is used to point to the root of the resulting structure. It is set to the value of next(Mem) at the initialization part of the algorithm, and it isn't changed. Queue invq is a queue in which nodes fetched from the queue q are stored in inverse order.

In the first loop node c is set to the next available memory space. An auxiliary field is added to every node copied with pointers to its original siblings, its copy and itself. Upon traversal of the right sublist the right pointer is copied in the copy structure. Thus the right pointer field will be available to be overwritten, since its content is saved. If the right sublist need not be traversed then the copy of the original pointer is written in the right pointer field of the copy. If a left pointer field is traversed the copy pointer field is immediately filled with the copy of the original pointer. Note that in the case of a pointer to be traversed the copy pointer will point to the next field in the copy space.

The second traversal at the moment merely allocates a pointer field - the right one - in the copy structure to the task of maintaining the link between original and copy node, and

```
PROC flc'=(REF NODE n)REF NODE:
BEGIN
  LOC REF NODE f, c, c1, copy;
  LOC NODESET m; LOC QUEUE q, invq;
  LOC BOOL finished;
  f:=n; m:=∅; q:=nilq; finished:=FALSE; c:=Mem; copy:=next(c);
  {LoopInvListMark1' ∧ LoopInvListCopy1'}
  WHILE NOT finished
  DO WHILE m:=m∪{f} ; c:=next(c); a OF f:=(f, c, 1 OF f, r OF f);
      IF r OF f∈m THEN r OF c:=cn OF a OF r OF f
      ELSE r OF c:=r OF f FI;
      r OF f∉m
    DO enq(q,f); f:=r OF f OD;
    WHILE IF q=nilq
      THEN finished:=TRUE
      ELSE f:=deq(q); c1:=cn OF a OF f;
        IF 1 OF f∈m
        THEN 1 OF c1:=cn OF a OF 1 OF f
        ELSE IF atom(1 OF f) THEN 1 OF c1:=1 OF f
          ELSE 1 OF c1:=next(c ) FI
        FI
      FI;
      IF finished THEN FALSE ELSE 1 OF f∈m FI
    DO SKIP OD;
    IF NOT finished THEN f:=1 OF f FI
  OD; {PostListCopy1'}
  f:=n; m:=∅; q:=nilq; invq:=nilq; finished:=FALSE;
  {LoopInvListMark2' ∧ LoopInvListCopy2'}
  WHILE NOT finished
  DO WHILE m:=m∪{f}; c1:=cn OF a OF f; r OF c1:=f;
      r OF f∉m
    DO enq(q,f); f:=r OF f OD;
    WHILE IF q=nilq
      THEN finished:=TRUE
      ELSE f:=deq(q); addq(invq, f)
      FI;
      IF finished THEN FALSE ELSE 1 OF f m FI
    DO SKIP OD;
    IF NOT finished THEN f:=1OF f FI
  OD; {PostListCopy2'}
  WHILE f:=deq(invq); c1:=cn OF a OF f;
    IF atom(r OF f)
    THEN r OF c1:=r OF f
    ELSE r OF c1:=cn OF r OF f
    FI;
    invq≠nilq
  DO SKIP OD;
  copy ∉ the result ∉
END {PostListCopy3'}
```

Fig. 5.5: algorithm flc'

The various assertions are described in figure 5.7.

builds the queue invq in the correct order. The proper queue-handling primitive is not enq, which adds a node at the back, but addq, which adds a node at the front. The correctness of the new primitive will be assumed again.

The last loop empties the queue invq while setting the right pointer fields in the copy structure to their final value. The reason for the treatment in the opposite direction will be clear in the next algorithm. If a right pointer is a forward pointer then the copy pointer will be to the next copy node. If it is a back pointer, then the copy of the node pointed to will not be treated yet, since it was put on queue q earlier.

The implementation of both queues is dependent on the function next and its inverse prev. Thus by definition next(prev(c))= prev(next(c))=c. Then it is possible to define two queues in memory space, starting with a certain node qp.

$$Qu(qp) \triangleq \begin{cases} nilq & \text{if } qp=next(c) \\ addq(Qu(next(qp)),\ qp) & \text{otherwise} \end{cases}$$

$$Iq(qp) \triangleq \begin{cases} nilq & \text{if } qp=prev(copy) \\ addq(Iq(prev(qp)),\ qp) & \text{otherwise} \end{cases}$$

The last space used in the part of memory reserved for the copy is pointed to by node c; and the first space is used for the root of the copy structure. So next(c) and prev(copy) fall exactly outside the list of nodes in which the copy structure is located. Note that the information in the copy nodes is sufficient. In the first and second loops the only pointers asked for are the pointers to the left child of the original nodes in the order the corresponding copy nodes are placed in the allocated memory space. In the last loop both the original and the copy node are necessary in matching pairs. This is assured by the second loop's action of setting the right pointer of the copy to the original.

The advantage of this approach is first of all the economy in memory necessary for the queues. Secondly the queues are built implicitly by the assignments to c, so speed is also increased.

The other transformations will be treated in the order in which they appear in the program. In the first loop the pointer from each original node to its assigned copy node is implemented by overwriting the value in the right pointer field of the original node by a pointer to the copy node; the value was saved in

the right pointer field of the copy, so no information is lost. As an important side-effect every right pointer field of a node in the set m is pointing to a copy node, and thus the test f∈m is equivalent to the test iscopy(r OF f).

After the first loop the left pointers in both original and copy nodes are correctly filled in. The right pointers of every original node is pointing to its assigned copy. If the original right pointer was pointing forward - is followed in the spanning tree defined by lm6f - then it is stored in the right pointer field of the copy. Otherwise the copy of this node is stored in the right pointer field of the copy. The same search structure is used to switch roles between the right pointer fields in original and copy structures.

Thus after the second loop the left pointer fields are still correctly filled in. The right pointer field in every copy node is pointing to the corresponding original node. The right pointer field in the original node is pointing to the right sibling of the original if it was a forward pointer, and to the right sibling of the copy if it was a back pointer. In the first case the right pointer field of the copy node should point to the copy of the next original node traversed in the first loop. This is the next field in the copy list, since the spanning tree defined by the Fisher list-marking algorithm is going downward to the right in this case. The second case necessitates retrieval of the original corresponding to the right child of the copy node. This is still possible, since the nodes are treated in reverse order of the original search.

The final list-copying algorithm devised by Fisher can be found in figure 5.6. Auxiliary variables are the set m and the values in field a corresponding to every node. The statements dealing with these values are underscored. They can be deleted to produce the original Fisher list-copying algorithm.


```
PROC flc =(REF NODE n)REF NODE:
BEGIN
  LOC REF NODE f, c, c1, copy, qp;
  LOC NODESET m; LOC BOOL finished;
  f:=n; m:=∅; finished:=FALSE; c:=Mem; copy:=next(c); qp:=copy;
  {LoopInvListMark1 ∧ LoopInvListCopy1}
  WHILE NOT finished
  DO WHILE m:=m∪{f}; c:=next(c); a OF f:=(f, c, l OF f, r OF f);
       c1:=r OF f; r OF f:=c; l OF c:=l OF f;
       IF iscopy(r OF c1) THEN r OF c:=r OF c1 ELSE r OF c:=c1 FI;
       iscopy(r OF c1)
     DO f:=c1 OD;
     WHILE IF qp=next(c)
       THEN finished:=TRUE; FALSE
       ELSE c1:=qp; qp:=next(qp);
         IF NOT atom(l OF c1)
         THEN IF iscopy(r OF l OF c1)
           THEN l OF c1:=r OF l OF c1; TRUE
           ELSE l OF c1:=next(c ); FALSE
           FI
         ELSE TRUE
         FI
       FI
     DO SKIP OD;
     IF NOT finished THEN f:=l OF f FI
  OD; {PostListCopy1}
  f:=n; m:=∅; finished:=FALSE; qp:=copy;
  {LoopInvListMark2 ∧ LoopInvListCopy2}
  WHILE NOT finished
  DO WHILE m:=m∪{f}; c1:=r OF f; r OF f:=r OF c1; r OF c1:=f;
       IF atom(r OF f) THEN FALSE ELSE iscopy(r OF r OF f) FI
     DO f:=r OF f OD;
     WHILE IF qp=next(c)
       THEN finished:=TRUE; FALSE
       ELSE c1:=qp; qp:=next(qp);
         IF atom(l OF c1) THEN TRUE ELSE NOT iscopy(r OF l OF c1) FI
       FI
     DO SKIP OD;
     IF NOT finished THEN f:=l OF c1 FI
  OD; {PostListCopy2}
  WHILE qp:=prev(qp); c1:=qp; f:=r OF c1;
     IF atom(r OF f)
     THEN r OF c1:=r OF f;
     ELSE IF iscopy(r OF f)
       THEN c:=r OF f; r OF f:=r OF c1; r OF c1:=c
       ELSE r OF c1:=next(c1)
       FI
     FI;
     qp≠copy
  DO SKIP OD;
  copy
END {PostListCopy3}
```

Fig. 5.6: Fisher's list-copying algorithm

The various assertions are described in figure 5.7.

$$
\begin{aligned}
\text{LoopInvListMark1}' \triangleq\ & n \notin m \cup \{f\} \wedge m \cup \{f\} \subseteq R^*(n) \wedge R(m - C(q) \cup \{f\}) \in m \cup \{f\} \wedge \\
& C(q) \subseteq m \cup \{f\} \wedge \text{finished} \rightarrow m = R^*(n)
\end{aligned}
$$

$$
\begin{aligned}
\text{LoopInvListCopy1}' = \ & \forall g \in m\ (cn\ OF\ a\ OF\ r\ OF\ g = next(cn\ OF\ a\ OF\ g) \rightarrow \\
& r\ OF\ cn\ OF\ a\ OF\ g = r\ OF\ g \ \wedge \\
& cn\ OF\ a\ OF\ r\ OF\ g \neq next(cn\ OF\ a\ OF\ g) \rightarrow \\
& r\ OF\ cn\ OF\ a\ OF\ g = cn\ OF\ a\ OF\ r\ OF\ g \wedge \\
& g \notin C(q) \rightarrow 1\ OF\ cn\ OF\ a\ OF\ g = cn\ OF\ a\ OF\ 1\ OF\ g)
\end{aligned}
$$

$$
\begin{aligned}
\text{PostListCopy1}' = \ & \forall g \in R^*(n)\ (cn\ OF\ a\ OF\ r\ OF\ g = next(cn\ OF\ a\ OF\ g) \rightarrow \\
& r\ OF\ cn\ OF\ a\ OF\ g = r\ OF\ g \wedge \\
& cn\ OF\ a\ OF\ r\ OF\ g \neq next(cn\ OF\ a\ OF\ g) \rightarrow \\
& r\ OF\ cn\ OF\ a\ OF\ g = cn\ OF\ a\ OF\ r\ OF\ g \wedge \\
& 1\ OF\ cn\ OF\ a\ OF\ g = cn\ OF\ a\ OF\ 1\ OF\ g)
\end{aligned}
$$

$$
\begin{aligned}
\text{LoopInvListMark2}' = \ & \text{LoopInvListMark1}' \wedge \forall g \in C(invq)\ (m > 1 \\
& (g = deq^m(invq) \rightarrow next(cn\ OF\ a\ OF\ g) = \\
& \qquad cn\ OF\ (deq^{m-1}(invq)))
\end{aligned}
$$

$$
\begin{aligned}
\text{LoopInvListCopy2}' = \ & \forall g \in R^*(n)\ (1\ OF\ cn\ OF\ a\ OF\ g = cn\ OF\ a\ OF\ 1\ OF\ g \cdot \\
& \wedge g \in m \rightarrow [cn\ OF\ a\ OF\ r\ OF\ g = next(cn\ OF\ a\ OF\ g) \rightarrow \\
& \qquad r\ OF\ cn\ OF\ a\ OF\ g = r\ OF\ g \ \wedge \\
& \qquad cn\ OF\ a\ OF\ r\ OF\ g \neq next(cn\ OF\ a\ OF\ g) \rightarrow \\
& \qquad r\ OF\ cn\ OF\ a\ OF\ g = cn\ OF\ a\ OF\ r\ OF\ g] \\
& \wedge g \notin m \rightarrow [cn\ OF\ a\ OF\ r\ OF\ g = g \wedge g \in C(invq)])
\end{aligned}
$$

$$
\begin{aligned}
\text{PostListCopy2}' = \ & \forall g \in R^*(n)\ (1\ OF\ cn\ OF\ a\ OF\ g = cn\ OF\ a\ OF\ 1\ OF\ g \cdot \\
& \wedge\ g \in C(invq)\ \ r\ OF\ cn\ OF\ a\ OF\ g = g \wedge \\
& \quad (\forall m > 1 (g = deq^m(invq) \rightarrow next(cn\ OF\ a\ OF\ g) \\
& \qquad = cn\ OF(deq^{m-1}(invq)))) \\
& \wedge g \notin C(invq) \rightarrow r\ OF\ cn\ OF\ a\ OF\ g = \\
& \qquad cn\ OF\ a\ OF\ r\ OF\ g)
\end{aligned}
$$

$$
\begin{aligned}
\text{PostListCopy3}' = \ & \forall g \in R^*(n)\ 1\ OF\ cn\ OF\ a\ OF\ g = cn\ OF\ a\ OF\ 1\ OF\ g \wedge \\
& r\ OF\ cn\ OF\ a\ OF\ g = cn\ OF\ a\ OF\ r\ OF\ g
\end{aligned}
$$

$$
C'(cq) = \{g \mid \exists c \in C(cq)\ \ cn\ OF\ a\ OF\ g = c\}
$$

$$
\begin{aligned}
\text{LoopInvListMark1} &= \text{LoopInvListMark1}' [C'(Qu(qn))/C(q)] \\
\text{LoopInvListCopy1} &= \text{LoopInvListCopy1}' [C'(Qu(qn))/C(q)] \wedge \\
& \quad [\forall g \in m\ r\ OF\ g = cn\ OF\ a\ OF\ g] \\
\text{PostListCopy1} &= \text{PostListCopy1}' [\forall g \in R^*(n)\ r\ OF\ g = cn\ OF\ a\ OF\ g] \\
\text{LoopInvListMark2} &= \text{LoopInvListMark2}' [C'(Qu(qn))/C(q)] \\
\text{LoopInvListCopy2} &= \text{LoopInvListCopy2}' [C'(Iq(qn))/C(invq)] \wedge \\
& \quad \forall g \in R^*(n)\ (g \in m \rightarrow r\ OF\ g = cn\ OF\ a\ OF\ g \\
& \quad \wedge g \in m \rightarrow [cn\ OF\ a\ OF\ r\ OF\ a\ OF\ g = \\
& \qquad next(cn\ OF\ a\ OF\ g) \rightarrow \\
& \qquad r\ OF\ g = r\ OF\ a\ OF\ g \\
& \qquad \wedge cn\ OF\ a\ OF\ r\ OF\ a\ OF\ g \neq \\
& \qquad next(cn\ OF\ a\ OF\ g) \rightarrow \\
& \qquad r\ OF\ g = cn\ OF\ a\ OF\ r\ OF\ a\ OF\ g]) \\
\text{PostListCopy2} &= \text{PostListCopy2}' [C'(Iq(qn))/C(invq)] \\
& \quad \forall g \in C'(Iq(qn))\ [cn\ OF\ a\ OF\ r\ OF\ a\ OF\ g = \\
& \qquad next(cn\ OF\ a\ OF\ g) \rightarrow \\
& \qquad r\ OF\ g = r\ OF\ a\ OF\ g \\
& \qquad \wedge cn\ OF\ a\ OF\ r\ OF\ a\ OF\ g \neq \\
& \qquad next(cn\ OF\ a\ OF\ g) \rightarrow \\
& \qquad r\ OF\ g = cn\ OF\ a\ OF\ r\ OF\ a\ OF\ g] \\
& \quad \forall g \notin C'(Iq(qn))\ \ r\ OF\ g = r\ OF\ a\ OF\ g \\
\text{PostListCopy3} &= \text{PostListCopy3}'
\end{aligned}
$$

Fig. 5.7: Assertions from algorithms flc' and flc

§ 6: CLARK'S LIST-COPYING ALGORITHM.

Introduction

The fastest list-copying algorithm known to date has been published by Douglas W. Clark [C] in 1978. This algorithm requires that the copy of the original list structure is placed in a contiguous part of memory, a prerequisite also encountered with the Fisher algorithm. However, while Fisher's algorithm needs three passes through the entire list-structure, Clark's algorithm needs only two plus an additional traversal of an (usually small) stack.

Clark's list-marking algorithm.

In § 2 it was mentioned that Clark's algorithm uses a depth-first-search of a directed graph as the underlying marking algorithm. Below this algorithm will be derived from lm2, which in turn was derived from lm0 in § 4.

In lm2 every node searched was put in a boundary set b if it had any unmarked siblings. The set b thus consisted of all nodes encountered with possibly unmarked siblings. Since the next node marked is one of the unmarked siblings, a node need not be placed in the set b if it has only one unmarked son. Additionally, if in selecting a node out of the set b a node is found with only marked siblings, then deletion of this node from set b and selection of another node would not violate the loop invariant. Thirdly, if the node selected has exactly one unmarked sibling left, it is not necessary to visit it again, so the search may continue immediately with the next node visited, the as yet unmarked son. Again the loop invariant is not violated. Inclusion of these three refinements in lm2 will diminish the number of loops, thus speeding up the final program. Note that introduction of the first and third refinements doesn't make the second one superfluous. In a directed graph is might be possible to reach one child via another. If on first inspection of the father node two children are unmarked, on the next visit both can be marked.

The first refinement is realized through replacement of the statement b:=b∪{f} in lm2 by IF R(f)-(m∪{s})≠∅ THEN b:=b∪{f} FI. Node f will only be placed in b if there is another son besides s

```
PROC lm3c =(REF NODE n)VOID:
BEGIN
  LOC REF NODE f, s:
  LOC NODESET m, b;
  (f, m, b) := (n, ∅, ∅);
  {n∈m∪{f}∧m∪{f}⊆R*(n)∧
   R(m-b∪{f})⊆m∪{f}∧
   b⊆m-{f}}
  WHILE NOT(b=∅ AND
            R(f)⊆m∪{f})
  DO m:=m∪{f};
    IF NOT R(f)⊆m
    THEN s:=x.x∈R(f)-m;
      IF R(f)-(m∪{s})≠∅
      THEN b:=b∪{f} FI;
      f:=s
    ELSE
      WHILE f:=x.x∈b; b:=b-{f};
        b≠∅ AND
          R(f)⊆m
      DO SKIP OD;
      IF R(f)⊄m
      THEN s:=x.x∈R(f)-m;
        IF R(f)-m={s} THEN f:=s FI
      FI
    FI
  OD;
  m:= m∪{f}
END {m=R*(n)}
```

Fig. 6.1: algorithm lm3c

```
PROC lm4c =(REF NODE n)VOID:
BEGIN
  LOC REF NODE f, s;
  LOC NODESET m; LOC STACK kst;
  (f, m, kst) := (n, ∅, nilst);
  {n∈m∪{f}∧m∪{f}⊆R*(n)∧
   R(m-C(kst)∪{f})⊆m∪{f}∧
   C(kst)⊆m-{f}}
  WHILE NOT(kst IS nilst AND
            R(f)⊆m∪{f})
  DO m:=m∪{f};
    IF type(f)∈{AF,BF,FA,FB,FF}
    THEN s:=x.x∈R(f)-m;
      IF type(f)=FF
      THEN push(kst,f) FI;
      f:=s
    ELSE ¢ type(f)∈{AA,AB,BA,BB} ¢
      WHILE f:=pop(kst);
        kst ISNT nilst AND
          type(f)≠FF
      DO SKIP OD;
      IF type(f)=FF
      THEN f:=x.x∈R(f)-m
      FI
    FI
  OD;
  m:= m∪{f}
END {m=R*(n)}
```

Fig. 6.2: algorithm lm4c

which is not yet marked. The second refinement consists of replacing (f:=x.x∈b; b:=b-{f}), which simply picks an element of b, by a loop that repeats this action until a node with an unmarked sibling is found unless the set b is depleted. Thirdly, a test is included on the number of unmarked siblings. The resulting algorithm lm3c is shown in figure 6.1.

Since the number of iterations of the main loop has decreased in algorithm lm3c as compared to algorithm lm2 termination of the former is clearly assured. The only possible snag is the introduction of a new inner loop. However, since this inner loop empties a finite set and terminates ultimately if the set is empty, there is no problem here either.

List structures are a special subclass of directed graphs. Every node has two pointer fields, each containing either an atom or a pointer to another node in the structure. The search-pattern used defines a spanning tree of the list structure. Every pointer in the list structure is either a part of the spanning tree, pointing forward, or it is pointing backward in the tree. So a pointer

field in the list can be of three types, called A (atom), B (back) and F (forward). Every node will be of one of nine types, depending on its pointers. E.g. a node of type AF has a left atom pointer and a right forward pointer. Note again that a node of type FF at first sight might turn out to be of type BF or FB.

The next transformation depends on the availability of a procedure type, which gives the type of a node as given above. The exact implementation of this procedure will depend on the amount of information given in the original structure about the spanning tree. Clark's list-copying algorithm passes the list structure twice. In the first pass all true FF-nodes will be marked to distinguish them from so-called inscrutable nodes, so the type-check of the second pass will differ from the first. At the moment correctness of type-checking - with the exception of inscrutable nodes upon first encounter - will be assumed for all nodes. Later on correctness of the various implementations has to be proved.

A second transformation is the replacement of the set b by a stack called kst. In this case a queue would have worked just as well, the only restriction will be that both traversals of the copying algorithm must use the same traversal order. The procedures push and pop are standard and the use of the notation C(st) indicating the contents of stack st in the assertions was encountered before. Lastly, a simplification is possible when a node of type FF was fetched from kst. Since it has two siblings and one is marked already it need not be put on the stack again. The application of these three transformations will result in algorithm lm4c (fig. 6.2).

Next introduction of the convention pop(nilst)=nil makes it possible to use transformation rule L2. Execution of the body of the loop while kst IS nilst AND R(f)⊆m∪{f} results in f being nil, which doesn't happen earlier during execution, so the test f ISNT nil is a candidate for the new loop-test. Since the set m isn't changed in the IF-statement the post-condition isn't invalidated by an extra loop. It remains to be proved that the loop-invariant and the old loop-test imply in conjunction the new loop-test. To this end the assertion f IS nil → kst IS nilst has to be added to the invariant, which is clearly valid. The old loop-test NOT(kst IS nilst AND R(f)⊆m∪{f}) and this new assertion imply f ISNT nil, since R(nil)=∅.

```
PROC lm5c =(REF NODE n)VOID:
BEGIN
  LOC REF NODE f;
  LOC NODESET m; LOC STACK kst;
  (f, m, kst) := (n, ∅, nilst);
  {n∈m∪{f}∧m∪{f}⊂ R*(n)∧
   R(m-C(kst)∪{f})⊆m∪{f}∧
   C(kst)⊆ m→{f}∧∀x∈C(kst)[r OF x∈m∪{f}]}
  WHILE f ISNT nil
  DO m:=m∪{f};
    CASE type(f)
    IN AA: f:=popS(kst),
       AB: f:=popS(kst),
       AF: f:=r OF f,
       BA: f:=popS(kst),
       BB: f:=popS(kst),
       BF: f:=r OF f,
       FA: f:=l OF f,
       FB: f:=l OF f,
       FF: push(kst, f);
           f:=r OF f
    ESAC
  OD
END {m=R*(n)}
```

Fig. 6.3: algorithm lm5c

```
PROC popS =(REF STACK s)REF NODE:
BEGIN
  LOC REF NODE f;
  WHILE f:=pop(kst);
    IF f ISNT nil THEN type(f)≠FF
    ELSE FALSE FI
  DO SKIP OD;
  IF type(f)=FF
  THEN f:=l OF f
  FI;
  f ¢ the result ¢
END
```

Fig. 6.4: algorithm popS

This new assertion can be used again to replace WHILE f:= pop(kst); kst ISNT nilst AND type(f)≠FF DO SKIP OD by WHILE f:= pop(kst); IF f ISNT nil THEN type(f)≠FF ELSE FALSE FI DO SKIP OD. The latter loop pops elements off kst until an FF-type node is encountered or the stack is emptied. The whole resulting ELSE-clause will be replaced by a procedure named popS for shortage of notation.

Furthermore it is desired to be more explicit about which son is selected while descending into the list structure. To achieve this every type of node has to be divided into four classes: no forward pointer, one left forward pointer, one right forward pointer and two forward pointers. Since we want to treat every type separately lateron, it is convenient to split the first three classes into their separate types, using a CASE-statement instead of the cumbersome repeated IF-statements. In the case of two forward pointers (the FF-type) the right son will be selected. Then for all nodes on the stack kst the right son will be marked, thus we can be sure that the left son should be selected when a node is popped off kst. The resulting algorithm lm5c and the procedure popS are given in figures 6.3 and 6.4.

Some auxiliary procedures

Clark's list-marking algorithm uses quite a few procedures. To keep the line of thought in the main algorithm more transparent some of them will be treated in advance.

The Clark algorithm demands the placement of the copy structure into a contiguous part of memory. This requirement enables Clark to introduce the test iscopy seen with Fisher again. Also, if a node is in this copy space, it is possible to ask for the address of the copy node directly following the present copy node, even before it is filled in. This again is a machine-level action which will be represented by an abstract procedure called next. Thus, if c is a node in the copy space, next(c) is the node directly following c. Actually the size of a node will be added to the address of c to give the address of next(c).

Two stacks are used by Clark's algorithm. Since the algorithm uses bounded workspace these stacks must be implemented using the actual nodes in either the original or the copy structure. Therefore nodes will be linked by their right pointer fields. Two rather trivial changes are necessary to the usual stack operators push and pop. These modified algorithms, called pushC and popC, are given below in figure 6.5.

```
MODE STACK = REF NODE;
PROC pushC =(REF STACK st, REF NODE n)VOID:
(r OF n :=st; st :=n);
PROC popC =(REF STACK st)REF NODE:
IF st IS nilst THEN nil
ELSE LOC REF NODE res; res:=st; st:=r OF st;
     res
FI
```

Fig. 6.5: algorithms pushC and popC

Incidentally, the MODE-declaration for stack shows an instant where the language Algol68 actually supports program refinements at compilation level. Unfortunately only one refinement per concept is allowed at a time.

Clark's list-copying algorithm

Clark's list-copying algorithm passes the list structure to be copied twice entirely. In the first pass the left pointer in every original node is used to implement the mapping between the original node and its copy. The other three pointer fields in the original and copy nodes are used to store sufficient information to be able to reconstruct every pointer field in both the original and the copy node. This information consists of the old pointers and in the case of a back pointer also the address of the copy of the node pointed to. Pointers to atoms can simply be copied and pointers forward in the original list structure will generally point to the next node in the copy. It will be possible to reconstruct the copies of the latter pointers in the second pass. To be able to perform the type-check in the second pass correctly every type of node will be stored in a distinct manner.

Problems arise with two types of nodes. Firstly the FF-type nodes have to be stored in a stack structure, the familiar kst. Since only three pointer fields are necessary to store all information relevant for the copying process (both original pointers and a pointer from the original node to the copy node) the fourth pointer can be used for the stack. Secondly inscrutable nodes by definition appear to be of type FF when they are encountered for the first time, so they will be treated as such. When an inscrutable node is popped off kst it can be recognised as being of type BF. At this moment it will be treated as a regular node of type BF. For efficiency reasons every true FF-node will be marked by means of the pointer field left over by the stack.

The other type giving problems is the BB-type. Normally five fields are necessary to store all essential pointers (two original and two new back pointers and a pointer from original node to copy). However, only four fields are available. This problem is solved by storing only the original pointers and the pointer for the mapping between copy and original. The fourth pointer field is used to store all BB-type pointers in a special stack called bst. This stack is emptied between the two passes of the entire structure and all pointer fields in both original and copy nodes are given their final value.

The second pass again traverses the entire list structure

while restoring the original values and storing the correct values in the pointer fields of the original and copy nodes respectively. Since all inscrutable FF-type nodes are distinguishable as BF-type nodes only true FF-type nodes will be stacked on the stack kst. At the moment an FF-type node is encountered going downward a pointer field is marked. This mark will not be needed any more, so the field can be used for the stack kst. When the node is popped off kst the last pointer fields can be filled in.

The basic list-copying algorithm used by Clark is shown in algorithm clc' (Clark's List-Copying algorithm) in figure 6.6. A new pointer c is introduced to point to the copy node attached to the node f currently examined. Both siblings of f are stored in convenient temporary variables oldl and oldr during the first pass. In this pass pointers to the original node, it's siblings and the matching copy node are stored in the auxiliary field of every node of the original structure. While used mainly in the correctness proofs, the pointer to the copy node is necessary at present in the algorithm.

The case-clause in the first pass is essentially the case-clause from lm5c. Preceding the statements of the latter algorithm the fields of the newly assigned copy node are filled in with useful information and in the case of the BB-type nodes a stack is filled. A few preparations for the necessary transformations are made. In every case the pointer to the lefthand sibling of the original node (oldl) is saved in another pointer field. The right pointer field of the FF-type nodes isn't used, so the copy nodes of all FF-type nodes can be linked into a stack as indicated in the previous section. To be able to link all BB-type nodes in a similar stack the pointer to the righthand sibling of this type of node is saved too. The pointer to the copy node matching a node pointed to by a back pointer is saved, unless the original node was of type BB. Copies of forward pointers can be calculated in pass two. For ease of identification the copy of the right pointer is calculated in the case of type AF. Lastly algorithm popS needed a slight modification, since all inscrutable FF-type nodes are really of type BF and will be treated as such. The new algorithm pop' is shown in figure 6.7.

The pointer c is advanced one node in the copy space at the end of every loop. This is accomplished by the abstract procedure call next treated in the last section.

```
PROC clc' =(REF NODE n)REF NODE:
BEGIN
  LOC REF NODE f, c, copy, oldl, oldr;
  LOC NODESET m, m'; LOC STACK kst, bst;
  (f, m, kst) := (n, ∅, nilst);
  c:=next(c); copy:=c; bst:=nilst;
  {LoopInvListMark' ∧ LoopInvListCopy1'}
  WHILE f ISNT nil ∉ PASS 1 ∉
  DO m:=m∪{f}; (oldl, oldr):=(l OF f, r OF f);
    a OF f := (f, c, oldl, oldr);
    CASE type(f)
    IN AA: l OF c:=oldl; r OF c:=oldr;                          f:=pop'(kst),
       AB: l OF c:=oldl; r OF c:=cn OF a OF oldr;               f:=pop'(kst),
       AF: l OF c:=oldl; r OF c:=next(c);                       f:=r OF f,
       BA: l OF c:=cn OF a OF oldl; r OF c:=oldl;               f:=pop'(kst),
       BB: l OF c:=oldl; r OF c:=oldr; push(bst,f);             f:=pop'(kst),
       BF: l OF c:=cn OF a OF oldl; r OF c:=oldl;               f:=r OF f,
       FA: l OF c:=oldl; r OF c:=oldr;                          f:=l OF f,
       FB: l OF c:=oldl; r OF c:=cn OF a OF oldr;               f:=l OF f,
       FF: l OF c:=oldl;                          push(kst,f);  f:=r OF f
    ESAC;
    c:=next(c)
  OD;
  {PostListCopy1'}
  WHILE bst ISNT nilst ∉ BSTACK-PROCESSING ∉
  DO f:=pop(bst); c:=cn OF a OF f;
    oldl:=l OF c; oldr:=r OF c;
    l OF c:=cn OF a OF oldl; r OF c:=cn OF a OF oldr;
    l OF f:=oldl; r OF f :=oldr
  OD;
  (f, m', kst) := (n, ∅, nilst);
  c:=copy;
  {LoopInvListMark'[m'/m] ∧ LoopInvListCopy2'}
  WHILE f ISNT nil ∉ PASS 2 ∉
  DO m':=m'∪{f};
    CASE type(f)
    IN AA:                                                      f:=pop"(kst),
       AB:                                                      f:=pop"(kst),
       AF:                                                      f:=r OF f,
       BA: r OF c:=r OF f;                                      f:=pop"(kst),
       BB:                                                      f:=pop"(kst),
       BF: r OF c:=next(c);                                     f:=r OF f,
       FA: l OF c:=next(c);                                     f:=l OF f,
       FB: l OF c:=next(c);                                     f:=l OF f,
       FF:                                        push(kst,f);  f:=r OF f
    ESAC;
    c:=next(c)
  OD;
  copy
END {PostListCopy2'}
```

Fig. 6.6: algorithm clc'

The various assertions are described in figure 6.14.


```
PROC pop' =(REF STACK s)REF NODE:
BEGIN
  LOC REF NODE f, c;
  WHILE f:=pop(kst);
    IF f ISNT nil THEN type(f)≠FF
    ELSE FALSE FI
  DO c:=cn OF a OF f;
    r OF c:=l OF c; l OF c:=cn OF a OF l OF c
  OD;
  IF type(f)=FF THEN f:=l OF f FI;
  f
END
```

Fig. 6.7: algorithm pop'

Between the two passes through the entire list structure all nodes and copies of nodes of type BB are processed. Every copy pointer is calculated and put into place. The original pointers are restored, though their value wasn't changed as yet. However, it is shown to be possible.

In pass two of the entire structure all remaining pointers in the copy structure are filled in with their final values. The final values of copy pointers in type FF copy nodes can be calculated in the action of emptying the stack kst; a modified procedure pop" is presented in figure 6.8. Note that the nodes remaining on stack kst are only scrutable FF-type nodes, so a loop to skip all inscrutable nodes is superfluous.

```
PROC pop" =(REF STACK s)REF NODE:
BEGIN
  LOC REF NODE f, a;
  f:=pop(kst);
  IF f ISNT nil
  THEN a:=cn OF a OF f; f:=l OF f;
    l OF a:=next(c); ∉ the copy of f ∉
    r OF a:=next(a)
  FI;
  f
END
```

Fig. 6.8: algorithm pop"

The treatment of the FF-type nodes is as usual: since there are no inscrutable nodes left the list traversal continues with the left child unless the stack was empty. The left child of the copy node should be the copy matching the left child of the node popped off kst, and this node is next(c). The copy of the right pointer is pointing to the node traversed directly following node f, so the right copy pointer should point to the next node in the copy space after the present copy node a.

To accomplish the claim for bounded workspace two stacks and per original node a pointer have to be included in the available structures. The pointer from original node to matching copy node will be stored in the space for the lefthand child. The stack bst will be implemented by linking the right pointer fields. The stack kst will be realized by linking the copy nodes by their right pointer fields. Procedures pop' and pop" have to be modified to accomodate these changes. Figures 6.9 and 6.10 contain the transformed procedures pop1 and pop2.

```
PROC pop1=(REF STACK s)REF NODE:
BEGIN
  LOC REF NODE f,c;
  WHILE c:=popC(kst);
    IF c ISNT nil
    THEN iscopy(1 OF 1 OF c)
    ELSE f:=nil; FALSE
    FI
  DO r OF c:=1 OF c;
    1 OF c:=1 OF 1 OF c
  OD;
  IF c ISNT nil
  THEN f:=1 OF c; r OF f:=markFF
  FI; f
END
```

Fig. 6.9: algorithm pop1

```
PROC pop2=(REF STACK s)REF NODE:
BEGIN
  LOC REF NODE f, a;
  a:=popC(kst);
  IF a ISNT nil
  THEN f:=1 OF a;
    1 OF a:=next(c);
    r OF a:=next(a)
  ELSE f:=nil
  FI;
  f
END
```

Fig 6.10: algorithm pop2

During pop1 every node popped off kst was either a scrutable or an inscrutable FF-type copy node. The left pointer contains the old left pointer of the original node matching this copy node. If the node pointed to was already marked, then its left pointer points to its copy node, otherwise its left pointer points to the original left sibling. So the test type(f)≠FF can be replaced by the test iscopy(1 OF 1 OF c). The other transformations are quite straightforward. A redundant assignation to c is deleted and an initial assignation to f is introduced in the case of an empty stack. The transformations from pop" to pop2 are similar to these last two transformations.

The final list-copying algorithm according to Clark is shown in figure 6.11. The algorithm proper excludes the underscored statements dealing with auxiliary variables. Transformations from clc' to clc are of three categories. The introduction of the specific stack-operators pop1, pop2, popC (included in the former two) and pushC was already explained, including the change of variables in pushing elements on kst. In the first loop the variables old1 and oldr are used in the statements defining the list-


```
PROC clc =(REF NODE n)REF NODE:
BEGIN
  LOC REF NODE f, c, copy, old1, oldr;
  LOC NODESET m, m'; LOC STACK kst, bst;
  (f, m, kst) := (n, ∅, nilst);
  c:=next(c); copy:=c; bst:=nilst;
  {LoopInvListMark ∧ LoopInviListCopy1}
  WHILE f ISNT nil ≠ PASS 1 ≠
  DO m:=m∪{f}; (old1, oldr):=(1 OF f, r OF f);
    a OF f := (f, c, old1, oldr); 1 OF f := c;
    CASE type1(f)
    IN AA: 1 OF c:=old1; r OF c:=oldr;                  f:=pop1(kst),
       AB: 1 OF c:=old1; r OF c:=1 OF oldr;             f:=pop1(kst),
       AF: 1 OF c:=old1; r OF c:=next(c);               f:=oldr,
       BA: 1 OF c:=1 OF old1; r OF c:=old1;             f:=pop1(kst),
       BB: 1 OF c:=old1; r OF c:=oldr;pushC(bst,f);     f:=pop1(kst),
       BF: 1 OF c:=1 OF old1; r OF c:=old1;             f:=oldr,
       FA: 1 OF c:=old1; r OF c:=oldr;                  f:=old1,
       FB: 1 OF c:=old1; r OF c:=1 OF oldr;             f:=old1,
       FF: 1 OF c:=old1;              pushC(kst,c );    f:=oldr
    ESAC;
    c:=next(c)
  OD;
  {PostListCopy1}
  WHILE bst ISNT nilst ≠ BSTACK-PROCESSING ≠
  DO f:=popC(bst); c:=1 OF f;
     old1:=1 OF c; oldr:=r OF c;
     1 OF c:=1 OF old1; r OF c:=1 OF oldr;
     1 OF f:=old1; r OF f:=oldr
  OD;
  (f, m', kst) := (n, ∅, nilst);
  c:=copy;
  {LoopInvListMark[m'/m] ∧ LoopinvListcopy2}
  WHILE f ISNT nil ≠ PASS 2 ≠
  DO m':=m'∪{f};
    CASE type2(f)
    IN AA: 1 OF f:=1 OF c;                              f:=pop2(kst),
       AB: 1 OF f:=1 OF c;                              f:=pop2(kst),
       AF: 1 OF f:=1 OF c;                              f:=r OF f,
       BA: 1 OF f:=r OF c; r OF c:=r OF f;              f:=pop2(kst),
       BB:                                              f:=pop2(kst),
       BF: 1 OF f:=r OF c; r OF c:=next(c);             f:=r OF f,
       FA: 1 OF f:=1 OF c; 1 OF c:=next(c);             f:=1 OF f,
       FB: 1 OF f:=1 OF c; 1 OF c:=next(c);             f:=1 OF f,
       FF: 1 OF f:=1 OF c;              pushC(kst,c);   f:=r OF f
    ESAC;
    c:=next(c)
  OD;
  copy
END {PostListCopy2}
```

Fig. 6.11: Clark's list-copying algorithm

The various assertions are described in figure 6.14.

marking part of the algorithm, since the old values are sometimes overwritten. Lastly, the pointer to the copy node via the old auxiliary value is replaced by the direct pointer stored in the left pointer field of the original node.

This last transformation series reveals the reasen why the BB-type nodes must be stored in a stack. The first BB-type node encountered cannot point to any other BB-type node, since none of them was encountered as yet. Similarly the second BB-type node encountered can only point to the first one, not to others. By removing the pointer to the copy of the last BB-type node encountered no information is lost about pointers in the copy nodes of the others. Then the last but one can safely be removed, etcetera. The stack structure is the easiest way to accomplish such an algorithm elegantly, and in this case efficiently.

Nodetypes

One part of the transformation series from list-marking algorithm lm4c to the final algorithm clc was treated rather abstractly: the definition of the procedure type, which returns the correct type of a node, with the exception of an inscrutable FF-type node (sometimes given as FF instead of BF). In some instances this constructional void was filled in. The main use of the type-check is at the instant of decision which branch to take in the two case-clauses. The two procedures necessary differ of course since the original structure is altered considerably at the instant of the procedure call in the second clause. The first one, type1, is given in figure 6.12, the second one in 6.13.

```
PROC type1 =(NODE n)TYPE:
BEGIN
  REF NODE l =l OF n, r =r OF n;
  IF atom(l)
  THEN IF atom(r) THEN AA
       ELIF iscopy(l OF r)
       THEN AB ELSE AF FI
  ELIF iscopy(l OF l)
  THEN IF atom(r) THEN BA
       ELIF iscopy(l OF r)
       THEN BB ELSE BF FI
  ELSE IF atom(r) THEN FA
       ELIF iscopy(l OF r)
       THEN FB ELSE FF FI
  FI
END
```

Fig. 6.12: procedure type1

```
PROC type2 =(NODE n)TYPE:
BEGIN
  REF NODE c =l OF n;
  IF NOT iscopy(c) THEN BB
  ELIF atom(l OF c)
  THEN IF atom(r OF n) THEN AA
       ELIF r OF c=next(c)
       THEN AF ELSE AB FI
  ELIF atom(r OF n)
  THEN IF atom(r OF c)
       THEN FA ELSE BA FI
  ELIF r OF c=markFF THEN FF
  ELIF iscopy(l OF f) THEN BF
  ELSE FB
  FI
END
```

Fig. 6.13: procedure type2

A correctness proof for procedure type1 is superfluous. It is almost the definition, with the exception of the test iscopy (l OF f) where the definition should say f&m. The correctness of the second typecheck follows straightforward from the results of the first pass and the BB-stack processing.

```
LoopInvListMark' ≜ n∈mu{f}∧mu{f}⊆R*(n)∧R(m-(C(kst)u{f}))⊆mu{f}∧
                   C(kst)⊆ m-{f}∧∀g∈C(kst) r OF g∈mu{f}
LoopInvListCopy1'≜∀g∈m ∃c=cn OF a OF g∧
                   (type(g)=AA→1 OF c=1 OF a OF g ∧
                               r OF c=r OF a OF g
                   ∧type(g)=AB→1 OF c=1 OF a OF g∧
                               r OF c=cn OF a OF r OF a OF g
                   ∧type(g)=AF→1 OF c=1 OF a OF g∧
                               r OF c=next(c)
                   ∧type(g)=BA→1 OF c=cn OF a OF 1 OF a OF g∧
                               r OF c=1 OF a OF g
                   ∧type(g)=BB→1 OF c=1 OF a OF g∧
                               r OF c=r OF a OF g g C(bst)
                   ∧type(g)=BF→g∉C(kst)→(1 OF c=cn OF a OF 1 OF a OF g)∧
                                          r OF c=1 OF a OF g∧
                               g∈C(kst)→1 OF c=1 OF a OF g
                   ∧type(g)=FA→1 OF c=1 OF a OF g∧
                               r OF c=r OF a OF g
                   ∧type(g)=FB→1 OF c=1 OF a OF g∧
                               r OF c=cn OF a OF r OF a OF g
                   ∧type(g)=FF→1 OF c=1 OF a OF g)
PostListCopy1'   ≜∀g∈R*(n)∃c=cn OF a OF g∧
                   (type(g)=AA→....
                    ....       r OF c=r OF a OF g
                   ∧type(g)=BB→g∈C(bst)→1 OF c=1 OF a OF g∧
                                        r OF c=r OF a OF g
                               g∉C(bst)→1 OF c=cn OF a OF 1 OF a OF g∧
                                        r OF c=cn OF a OF r OF a OF g
                                                  )
                   ∧.....
LoopInvListCopy2'≜∀g∈R*(n)∃c=cn OF a OF g∧
                   (type(g)=AA→....
                    ....       r OF c=next(c)
                   ∧type(g)=BA→1 OF c=cn OF a OF 1 OF a OF g∧
                               g∉m'→r OF c=1 OF a OF g∧
                               g∈m'→r OF c=r OF a OF g
                  ∧ ....
                   ∧type(g)=BF→1 OF c=cn OF a OF 1 OF a OF g∧
                               g∉m'→r OF c=1 OF a OF g∧
                               g∈m'→r OF c=next(c)
                   ∧type(g)=FA→r OF c=r OF a OF g ∧
                               g∉m'→1 OF c=1 OF a OF g∧
                               g∈m'→1 OF c=next(c)
                   ∧type(g)=FB→r OF c=cn OF a OF r OF a OF g∧
                               g∉m' 1 OF c=1 OF a OF g∧
                               g∈m' 1 OF c=next(c)
                   ∧type(g)=FF→g∉m'-C(kst)→1 OF c=1 OF a OF g∧
                               g∈m'-C(kst)→
                                 (1 OF c=cn OF a OF 1 OF a OF c∧
                                  r OF c=cn OF a OF r OF a OF c)
PostListCopy2'   ≜∀g∈R*(n)∃c=cn OF a OF g(1 OF c=copy(1 OF g)∧
                                          r OF c=copy(r OF g))

copy(p)=if atom(p) then p else cn OF a OF p fi

LoopInvListMark  ≜ LoopInvListMark'
LoopInvListCopy1 ≜∀g∈m ∃c =cn OF a OF g∧1 OF f=c∧
                   (type(g)=AA→....
                   ∧type(g)=FF→1 OF c=1 OF a OF g∧
                               c∉C(kst)→r OF g=markFF)
```



```
PostListCopy1    ≜∀g∈R*(n)∧(type(g)≠BB∨g∈C(bst)→1 OF g=cn OF a OF g)
                   ∧PostListCopy1'
LoopInvListCopy2 ≜∀g∈R*(n)-m' 1 OF g=cn OF a OF g
                   ∧LoopInvListCopy2'
PostListCopy2    ≜ PostListCopy2'
```

Fig 6.14: Assertions from algorithms clc' and clc

§ 7: RELATED WORK

The introduction of Hoare's logic made it relatively easy to prove the correctness of small conventional programs. Correctness proofs of larger programs can be given in Hoare's logic, however, they tend to expand to impractical sizes. Thus some more structure is needed in the proof of a large program in Hoare's style.

Well developed is the technique of data refinement. An abstract data type is given together with the necessary axioms and operators to work with it. Then the abstract type and its abstract operators are replaced by a more concrete implementation. Validity of all these concretizations is proved and hence it is concluded that the new implementation is correct. A practical treatment of this technique can be found in the book of Jones [J].

Experience is also available on the technique of control transfer between program variables and auxiliary variables. Blikle [Bl] gives a very thorough example of this technique on a rather simple program: calculation of the integer square root.

Scherlis [Sc] gives some very nice program transformation rules. However, these rules do not alter the control structure of the original program. Thus many transformations in the present paper are not included, though termination can be proved easily.

To prove termination of a program after transformation is treated ad hoc in the present paper. Usually a transformation altering the control structure is intended to improve the speed of a program. Thus the termination of the resulting program is the first aim of the transformation. Then the rationale for the transformation is the first step towards a formal ad hoc proof. A more formal treatment of the termination criteria is possible. Apt and Delporte [AD] worked on this topic using a version of temporal logic as proof language.

An interesting development is also the introduction of a new primitive programming tool for graph algorithms by Suzuki [Su]. While his pointer rotation technique has some interesting advantages it doesn't work as smooth as he would like it to. Yet we obviously agree that good and reliable program tools have the clear advantage of making programs more transparent, and that transparent programs are more easily proved correct.

The important differences between the father article by Lee, De Roever and Gerhart [LRG] and the present paper are the more extensive use of transformations involving auxiliary variables and the greater concern with termination proofs. The basic idea, proving the correctness of the three list-copying algorithms through transformations starting with a list-marking algorithm, is retained.

BIBLIOGRAPHY

[AD] Apt, K.R. and Delporte, C. An axiomatization of the intermittent assertion method (extended abstract). Report 82-70 of the Laboratoire Informatique Thèorique et Programmation, Paris (1983).

[Ba] Back, R.J.R. Correctness preserving program refinements: proof theory and applications. MC Tracts 131, Amsterdam (1980).

[Bl] Blikle, A. Specified programming. ICS PAS Reports 333, Warszawa (1978).

[C] Clark, D.W. A fast algorithm for copying list structures. Comm. ACM Vol. 21 No. 5, p. 351-357 (1978).

[D] Darlington, J. A synthesis of several sorting algorithms. Acta Informatica 11, p. 1-30 (1980).

[F] Fisher, D.A. Copying cyclic list structures in linear time using bounded workspace. Comm. ACM Vol. 18 No. 5, p. 251-252 (1975).

[G1] Gerhart, S.L. Correctness preserving program transformations. Proc. second POPL Symp., p. 54-66, Palo Alto (1975).

[G2] Gerhart, S.L. Proof theory of partial correctness verification systems. SIAM J. Comput. Vol. 5 No. 3, p. 355-377 (1976).

[J] Jones, C.B. Software development: a rigorous approach. Prentice/Hall Series in Computer Science, London (1980).

[L] Livercy, C. Théorie des Programmes. Schémas, preuves, sémantique. Paris (1978).

[L'] Lindstrom, G. Copying list structures using bounded workspace. Comm. ACM Vol. 17 No. 4, p. 198-202 (1974).

[LRG] Lee, S. De Roever, W.P. and Gerhart, S.L. The evolution of list copying algorithms and the need for structured program verification. Conf. Rec. sixt ann. ACM Symp. on principles of progr. languages, p. 53-67, San Antonio (1979).

[R] Robson, J.M. A bounded storage algorithm for copying cyclic list structures. Comm. ACM Vol. 20 No. 6, p. 431-433 (1977).

[RS] Reif, J.H. and Scherlis, W.L. Deriving efficient graph algorithms. Report CMU-CS-82-155, (1982).

[Sc] Scherlis, W.L. Program improvement by internal specialization. Conf. Rec. eight ann. ACM Symp. on principles of progr. languages, p. 41-49 (1981).

[Su] Suzuki, N. Analysis of pointer "rotation". Comm. ACM Vol. 25 No. 5, p. 330-335 (1982).

[SW] Schorr, H. and Waite, W.M. An efficient machine-independent procedure for garbage collection in various list structures. Comm. ACM Vol. 10, p. 501-506 (1967).

[N] Neumann, P.G. Computer System Security Evaluation. 1978 National computer Conference, Anaheim CA (1978).