Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

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

Program derivation through transformations:
The evolution of list-copying algorithms

Department of Computer Science Report CS-R8520 September

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

Program Derivation through Transformations: The Evolution of List-Copying Algorithms

N.W.P. van Diepen
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands[1]

W.P. de Roever
Eindhoven Technical University
P.O. Box 513, 5600 MB Eindhoven, The Netherlands[1]

The introduction of Hoare 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 show the derivability and to prove the correctness of some fast list-copying algorithms developed by Robson, Fisher and Clark. This subject was motivated by an earlier paper on the same topic by Lee, De Roever, and Gerhart. Some transformation rules necessary for the correctness proofs are given. Other proof techniques used include data refinement and the use of auxiliary variables and structures.

1980 Mathematical Subject Classification: 68B10 [Analysis of Programs], 68C05 [Algorithms], 68E10 [Graph Theory].
1983-84 CR Classification: D.2 SOFTWARE ENGINEERING, D.2.2 TOOLS AND TECHNIQUES: structured programming, D.2.4 PROGRAM VERIFICATION: correctness proofs, E.1 DATA STRUCTURES: lists, F.3 LOGICS AND MEANINGS OF PROGRAMS, F.3.1 SPECIFYING AND VERIFYING AND REASONING ABOUT PROGRAMS, I.2.2 AUTOMATIC PROGRAMMING.
Key words & phrases: program verification, Hoare Logic, program transformation, list traversal, Deutsch-Schorr-Waite algorithm, list-copying, Robson's algorithm, Fisher's algorithm, Clark's algorithm, graph algorithms.
Note: this paper will be published in *Science of Computer programming.*

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.*

1. This paper is a revision of Report RUU-CS-85-3. Work on this paper was done while both authors were at the State University of Utrecht.

Report CS-R8520
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

> *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.* [11])

Though written half a decade ago the introductory statement of Lee, De Roever and Gerhart [11] is still valid today. The present paper may be considered as a follow-up on their work in that it further develops their techniques.

Verification of the correctness of computer programs is a subject becoming more important with the increasing need for reliable systems programming [15]. Presently basic tools with some claim of practical utility have been developed to prove the correctness of sequential programs. This paper aims at showing the possibilities of these tools by proving the correctness of some really intricate sequential programs.

The techniques used fall roughly into three general categories. The first category is the straightforward correctness proof. The accepted tool is a series of proof rules developed by Tony Hoare, called *Hoare Logic.* Then there is the technique of *data refinement* as described by Jones [10]. For instance a set can be represented by a search tree. Thirdly the technique of *program transformation* is used. It is based on the Hoare 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 [11].

A *list* is a directed connected graph with at most two outward going edges from each node and every node in the list can be reached from one node, the root, by following edges in the proper direction. Thus in a general directed graph with all nodes having no more than two outgoing edges a list can be found by taking a node as the root of the list and adding all nodes reachable from this node via outgoing edges. A binary tree qualifies as a list without cycles and alternate paths. The edges are further identified as the *left* and *right* pointer (or sometimes as *car* and *cdr*; this notation will not be used). The unlucky choice of the name 'list' comes from the structure used in LISP (the *List-p*rocessing language).

A list-copying algorithm is an algorithm making a duplicate of a list, i.e. to every node will be assigned a copy node, data stored in the original node are also placed in the copy node, and the pointer values of the copy node are pointing to the copy nodes of the nodes pointed to in the original node.

The structure of the correctness proofs of the three best list-copying algorithms presently known are given in the next paragraph. The technique of program transformation is explained in § 3. Full derivations of the algorithms are presented in §§ 4, 5 and 6. The latter two can be read after sections 4.1, 4.2 and the beginning of 4.3. Finally § 7 contains some notes on related work.

2. The structure of the correctness proofs of three list-copying algorithms

2.1. Introduction

A list-copying algorithm consists of two basic subtasks. Firstly, to every node in the original list precisely one new node of the same format has to be assigned. Secondly, the contents of every original node have to be transferred to the assigned copy node, allowing for the change in pointer addresses.

One of the things one has to be sure of is, that every node in the original list is visited. Algorithms which visit every node in a list structure, perform a certain action and keep the structure intact upon termination can be viewed as list-marking algorithms. In the algorithms under discussion marking will be implemented as part of the copying action.

Basically, proving the three list-copying algorithms correct consists of two stages:

1) proving the underlying list-marking algorithms correct, and
2) adding the actual copying actions and proving the correctness of these actions in light of the correctness of the marking algorithms.

Correctness of the two subtasks of list-copying mentioned above belongs entirely to the latter stage. It should be noted, that the Fisher and Clark list-marking algorithms do not perform these tasks one after another, but immediately transfer some information to the assigned new node.

2.2. The Robson algorithm

The Robson list-copying algorithm is based on two different, although closely related list-marking algorithms. The first one is simply a d.s.w.-search (for *D*eutsch-*S*chorr-*W*aite [20]) of a directed graph. This algorithm will be derived from an algorithm developed by Lee, De Roever and Gerhart [11] called *lm0* in the present paper.

The second algorithm will also be derived from *lm0.* However, early on in this derivation it will be assumed that information about a spanning tree of the searched list is already available in the nodes. Actually, this information is "left behind" by the first algorithm. At first, only existence of the latter information is assumed (separating the concerns). (Later on, it is proved that the first algorithm actually leaves this information behind.) So, if the second algorithm is executed after the first one, it performs marking correctly, in the sense indicated above. Incidentally, the second marking removes the marks of the first pass.

Once we have proved that the marking aspect of the algorithm is correct, we can add the copying actions to both traversals of the list. In the first algorithm a new node is assigned to every original node. The values of the pointers in the original node are stored in the copy, freeing these pointers to keep track of the copy node and to code the spanning tree needed by the second traversal. To prove that the original structure is not lost, but merely stored differently, an auxiliary variable will be used. With this auxiliary variable the second marking algorithm can easily be adapted to the structure modified as above, while retaining correctness. Then the marking action will be "enlarged" to storing the correct information in the pointers of every new node and restoring the information of every original node. Removing the auxiliary variables necessary for the proof results in a correct list-marking algorithm.

2.3. The Fisher algorithm

The Fisher list-copying algorithm traverses a list structure three times. The first traversal is based on a rather unusual list-marking algorithm. Every node is added to a queue if it is visited for the first time. The next node it visits is its right child, or, if the right child does not exist or has already been visited, it visits the first unvisited left child of the nodes on the queue. This algorithm will also be derived from *lm0.*

A typical list structure is shown in figure 2.1 together with the traversal path of this list-marking algorithm.

The traversal order of the nodes in this list-marking algorithm is also the order in which the nodes

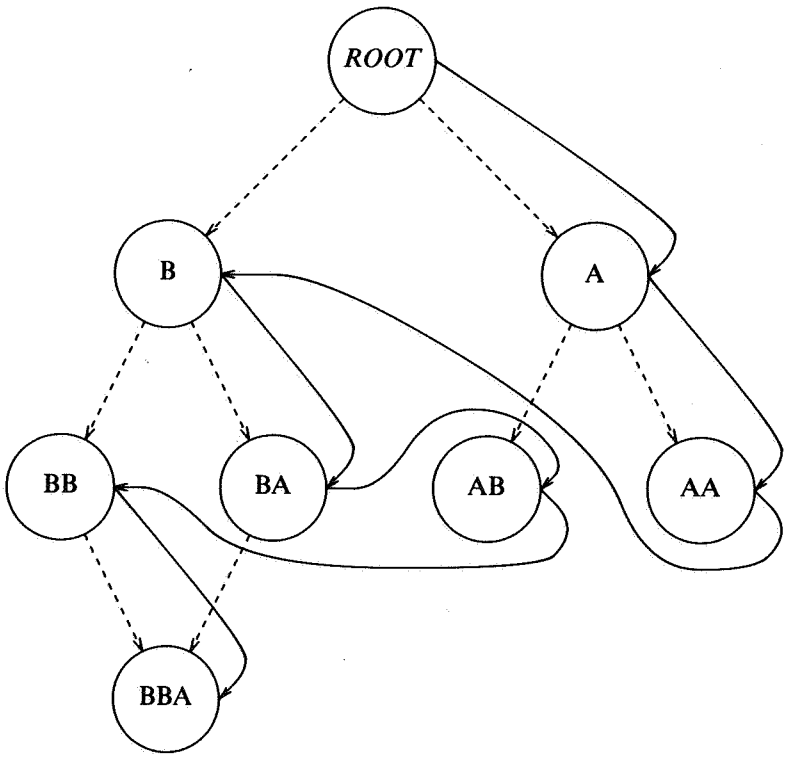

FIGURE 2.1: Search path in Fisher's algorithm.

are placed in the queue. (Fisher ([7], p. 251) claims the order of traversal of the nodes to be alphabetical if the path from the root of the list structure to a node is described using A for right and B for left. This claim seems to be incorrect, since node B is visited before node AB.)

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 implementor of the Fisher algorithm for his particular machine. In the sequel the correctness of this implementation will be assumed.

Another advantage of the strategy of placing 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 implementor. 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-copying algorithm remains.

2.4. The Clark algorithm

The Clark list-copying algorithm is based on only 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 left-child. Again, this list-marking algorithm will be derived from *lm0*.

Depending 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 list-marking algorithm used 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 be placed in a contiguous part of memory. The Clark algorithm makes use of three advantages which follow from the technique of contiguous copying. The same test *iscopy* as in Fisher's algorithm 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 their implementations 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 allow a special treatment. 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 of BB-type nodes.

Next, the changes in the BB-type nodes will be completed 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. Some remarks about Hoare Logic

3.1. Introduction

The method for correctness proofs called Hoare Logic has found its way into standard textbooks on program verification (e.g. [13]). In this section some transformation rules based on Hoare Logic will be introduced. In addition, the concept of *auxiliary variable* is widened to facilitate reasoning in detail about data structures and the application of this concept is broadened to program transformations.

3.2. Transformation rules

To give the reader some flavour of the concept of provability in Hoare Logic of program transformations some examples are given. Only a subset of the transformations used in this paper is presented. Proofs of their correctness are omitted.

Some assignment transformations:

Equivalent assignments:

$$\frac{\{P\}\ x:=E\ \{Q\} \quad P \rightarrow E=F}{\{P\}\ x:=F\ \{Q\}} \tag{A}$$

Switching two assignments:

$$\frac{\{P\}\ x:=E;\ y:=F\ \{Q\}}{\{P\}\ y:=F;\ x:=E\ \{Q\}}\ \textit{with } x \notin FV(F),\ y \notin FV(E) \textit{ and } x \neq y \tag{AS}$$

Some transformations involving conditional statements:

Skipping a statement:

$$\frac{\{P\}\ S_1;\ \{Q\}\ S_2\ \{R\} \quad P \wedge \neg B \rightarrow Q}{\{P\}\ \textbf{if}\ B\ \textbf{then begin}\ S_1;\ \{Q\}\ S_2\ \textbf{end else}\ S_2\ \{R\}} \tag{C1}$$

Moving a statement out of a conditional statement:

$$\frac{\{P\}\ \textbf{if}\ B\ \textbf{then begin}\ S_1;\ S\ \textbf{end else begin}\ S_2;\ S\ \textbf{end}\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2;\ S\ \{Q\}} \tag{C2}$$

Some transformations involving loops:

Moving a statement through a loop:

$$\frac{\begin{array}{c}\{P\}\ S_1;\ \{P'\}\ \textbf{while}\ B\ \textbf{do begin}\ S_2;\ \{P\}\ S_1\ \textbf{end}\ \{Q\} \\ \{P \wedge \neg B'\}\ S_1\ \{P \wedge \neg B\} \quad \{P \wedge B'\}\ S_1\ \{P \wedge B\}\end{array}}{\{P\}\ \textbf{while}\ B'\ \textbf{do begin}\ S_1;\ \{P'\}\ S_2\ \textbf{end};\ S_1\ \{Q\}} \tag{L1}$$

Addition of one cycle to a loop:

$$\frac{\begin{array}{c}\{P \wedge B'\}\ \textbf{while}\ B\ \textbf{do begin}\ S_1;\ S_2\ \textbf{end};\ S_1\ \{Q\} \\ \{P \wedge B\}\ S_1;\ S_2\ \{B'\} \quad \{P \wedge \neg B \wedge B'\}\ S_1;\ S_2\ \{P \wedge \neg B'\} \\ P \wedge B \rightarrow B' \quad \{Q\}\ S_2\ \{Q\}\end{array}}{\{P \wedge B\}\ \textbf{while}\ B'\ \textbf{do begin}\ S_1;\ S_2\ \textbf{end}\ \{Q\}} \tag{L2}$$

Splitting up of a loop:

$$\frac{\{P\}\ \textbf{while}\ B\ \textbf{do if}\ B'\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{Q\}}{\begin{array}{l}\{P\}\ \textbf{while}\ B\ \textbf{do} \\ \quad \textbf{begin} \\ \quad\quad \textbf{while}\ B'\ \textbf{and}\ B\ \textbf{do}\ S_1; \\ \quad\quad \textbf{while not}\ B'\ \textbf{and}\ B\ \textbf{do}\ S_2 \\ \quad \textbf{end}\ \{Q\}\end{array}} \tag{L3}$$

An important point concerning transformations involving loops is termination. Since Hoare Logic does not express termination of a program this has to be proved separately. Under normal circumstances the motivation for a certain transformation gives a good guideline for an *ad hoc* proof.

3.3. Auxiliary variables and structures

3.3.1. Auxiliary variables. A crucial role in more involved Hoare style correctness proofs is played by auxiliary variables. An auxiliary variable has no influence on the flow of control of the program. This is captured by the following definition:

DEFINITION 3.1: A set of variables AV is an *auxiliary variable set* **iff** every variable $x \in AV$ occurs only in statements of the form:

$$x := E$$

or

$$y := E(x)$$

with E an expression and $y \in AV$.

This definition allows one to choose a set of auxiliary variables within certain limits. By default the empty set fulfills the requirements.

A set of auxiliary variables can be added to a program and deleted again whenever convenient, since the flow of control is not affected by it. This is stated in the following theorem:

THEOREM 3.2: *Let P, Q be assertions, S a program, AV a set of auxiliary variables of S with* $(FV(\mathrm{P}) \cup FV(\mathrm{Q})) \cap AV = \varnothing$ *and let S′ be program S with the assignments to elements of AV deleted, then*

$$\{\mathrm{P}\}\ S\ \{\mathrm{Q}\} \Leftrightarrow \{\mathrm{P}\}\ S'\ \{\mathrm{Q}\}$$

(Proof omitted).

Originally auxiliary variables were introduced to describe some relation between program variables which was difficult or impossible to formulate without a scratchpad. A typical example of such an auxiliary variable is a history variable that has the past of a program as its value but does not influence the program's present behaviour.

The sole purpose of this use is to make the proofs easier. A correctness proof of a program is first given with assignments to auxiliary variables added and then the input and output assertions are rewritten to eliminate the auxiliary variables. Application of Theorem 3.2 then shows the correctness of the original program.

Another use of auxiliary variables is specific to program transformations. Some auxiliary variables are added to a correct program. Next it is shown that they have the same expressive power at certain places in the program as some of the original program variables. Hence they can take over the role of these variables, thus losing their auxiliary character.

In many instances the role of some original program variables is taken over completely by the new variables. When a set of variables qualifies as an auxiliary variable set one can apply Theorem 3.2 to delete them. An application of this technique can be found in [4].

3.3.2. Auxiliary structures. The complexity of the algorithms dealt with is such that it is a difficult task to keep track of the values of the pointers in the structure. In particular it is difficult to guarantee the preservation of pointer values, since constraints on efficient use of memory imposed on these algorithms does not allow one to reserve space for every node in the original structure.

To facilitate reasoning about the current state of the data structure an expansion of the concept of auxiliary variables is introduced: the *auxiliary structure*. An auxiliary structure is a compound of new types of nodes and new fields in old nodes, each field and each node of which behaves as an auxiliary variable. Hence each auxiliary field can be inserted and deleted according to theorem 3.2.

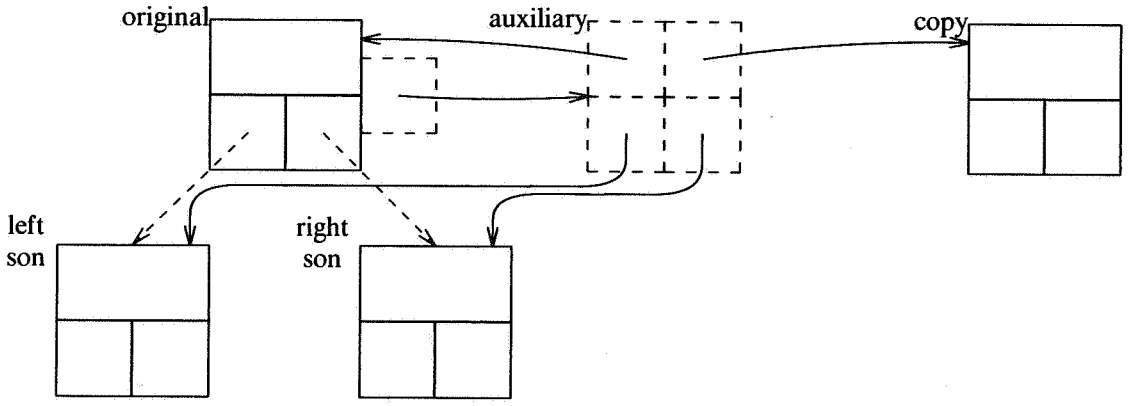

FIGURE 3.1: Example of an auxiliary node.

An example of the application of an auxiliary node can be seen in figure 3.1. Here an auxiliary field (the dashed box) is added to an original node. This field is pointing to an auxiliary node which keeps track of the original node, its corresponding copy, and its children.

The first two fields define part of the bijection which should exist between the original and the copy structure. The last two fields retain the information necessary to reconstruct the original list. Hence other information can be stored in the left and right pointer fields instead of the dashed arrows: the original information can easily be retraced. Additionally, an easy connection from original to copy node is provided for, starting in the dashed box via the auxiliary node.

A distinct advantage of such an auxiliary structure is that there is no need to bother about the amount of storage the moment the algorithm is conceived. One only has to keep track of the fact that everything can be stored somewhere when the auxiliary structure is skipped. This approach can also help to discover that a certain value must be retained in an additional data structure, if the algorithm one is designing does not leave room in the minimally necessary data structure (i.e. the result data structure).

4. Robson's List-Copying Algorithm

4.1. 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.

4.2. 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. An infix relation R on nodes can be defined as follows:

$$mRn \quad \Leftrightarrow \quad \text{an edge is pointing from } m \text{ to } n$$

This relation induces the following set definitions:

$$R(n) \;=\; \{m \mid nRm\}$$

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

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 [11]. One of these algorithms (called MA3 in [11]) is given below in a different notation as algorithm *lm0* in figure 4.1.

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

$$v := x \mid P(x),$$

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:

$$\{\exists x\; P(x)\}\;\; v := x \mid P(x)\;\; \{P(v)\}\;\; \text{with } v \text{ not free in } P$$

An example of the use of this rule can be seen in $\{b \neq \varnothing\}\; f := x \mid x \in b\; \{f \in b\}$.

The input of algorithm *lm0* is a node n. The relation R is implicitly defined as above. Some pointers to nodes, 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 unvisited. If all the children of the father-node are marked, search continues at some element of the boundary-set, unless this set is empty. In the last case the algorithm terminates.

$\{n \in Mem \ \wedge \ R \in Mem \times Mem\}$
$f := n;\ m := \{f\};\ b := \varnothing;$
$\{f,\ n \in m \ \wedge \ m \subseteq R^*(n) \ \wedge$
$R(m-(b \cup \{f\})) \subseteq m \ \wedge \ b \subseteq m - \{f\}\}$
while not $(b = \varnothing$ **and** $R(f) \subseteq m)$
do if not $R(f) \subseteq m$
then
begin $s := x \mid x \in R(f) - m;$
$m := m \cup \{s\};\ b := b \cup \{f\};\ f := s$
end
else
begin $f := x \mid x \in b;\ b := b - \{f\}$
end
$\{m = R^*(n)\}$

FIGURE 4.1: Algorithm *lm0*.

4.3. 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 Deutsch-Schorr-Waite algorithm, will be derived from *lm0* below.

Repeated application of transformation rule *AS* allows us to change $m := m \cup \{s\};\ b := b \cup \{f\};\ f := s$ in the **then**-clause into $b := b \cup \{f\};\ f := s;\ m := m \cup \{s\}$. The last assignment can obviously be replaced by $m := m \cup \{f\}$ (application of rule *A*). Similar transformations replace the initializations $m := f;\ b := \varnothing$ by $m := \varnothing;\ b := \varnothing;\ m := m \cup \{f\}$.

Since at the end of the **else**-clause m equals $m \cup \{f\}$, rule *AI* allows introduction of $m := m \cup \{f\}$. This statement can now be moved out of the **if**-statement by application of rule *C*2. The resulting algorithm *lm1* is shown in figure 4.2.

$f := n;\ m := \varnothing;\ b := \varnothing;$
$\{f,\ n \in m \cup \{f\} \ \wedge \ m \cup \{f\} \subseteq R^*(n) \ \wedge$
$R(m \cup \{f\} - (b \cup \{f\})) \subseteq m \cup \{f\} \ \wedge \ b \subseteq m - \{f\}\}$
$m := m \cup \{f\};$
while not $(b = \varnothing$ **and** $R(f) \subseteq m)$
do
begin
if not $R(f) \subseteq m$
then
begin $s := x \mid x \in R(f) - m;$
$b := b \cup \{f\};\ f := s$
end
else
begin $f := x \mid x \in b;\ b := b - \{f\}$
end;
$m := m \cup \{f\}$
end
$\{m = R^*(n)\}$

FIGURE 4.2: Algorithm *lm1*.

The statement $m := m \cup \{f\}$ is then the last statement in the **then**-clause and the last statement before the loop. Hence the stage is set for transformation rule *L1*. The resulting algorithm *lm2* is

shown in figure 4.3.

$f:=n;\ m:=\varnothing;\ b:=\varnothing;$
$\{f,\ n\in m\cup\{f\}\ \wedge\ m\cup\{f\}\subseteq R^*(n)\ \wedge$
$R(m\cup\{f\}-(b\cup\{f\}))\subseteq m\cup\{f\}\ \wedge\ b\subseteq m-\{f\}\}$
while not $(b=\varnothing$ **and** $R(f)\subseteq m\cup\{f\})$
do
  begin
    $m:=m\cup\{f\};$
    if not $R(f)\subseteq m$
    then
      begin $s:=x\,|\,x\in R(f)-m;$
        $b:=b\cup\{f\};\ f:=s;$
      end
    else
      begin $f:=x\,|\,x\in b;\ b:=b-\{f\}$
      end
  end;
$m:=m\cup\{f\}$
$\{m=R^*(n)\}$

FIGURE 4.3: Algorithm *lm2*.

A boolean variable *down* is added in algorithm *lm3* (figure 4.4). This is an auxiliary variable expressing that the current node f has been reached by going downward (i.e. to a new node) or upward (i.e. to a node marked previously) in the list structure. Its purpose is to save the information on the current node being marked from one loop to another. Additionally, it will facilitate to remove redundant markings.

The assertion $f\notin m \leftrightarrow down$ describing the behaviour of *down* has to be added to the loop-invariant. Verification of the correctness of this assertion is trivial.

While *down* is a genuine auxiliary variable in algorithm *lm3*, the next transformation step will give it influence on the flow of control. Application of transformation rule *C1* splits the execution of the loop-body in two. When *down* equals **false** the current node f is a member of m, so marking it again can be left out.

Removal of a few redundant assignations to *down* results in algorithm *lm4* (figure 4.5).

Since the order of copying and hence the order of traversal is important for the correct execution of a copying algorithm, the marking algorithm should define this order of traversal. Hence the aselect choice of an element from set b should be replaced by a select choice.

The set b will be implemented in the style of Jones [10] as a stack t with the usual operators *push* and *pop*. This data structure and its operators will be treated abstractly.

Introduction of a stack structure violates the ultimate goal of bounded workspace, since it is of undetermined size. In the final list-copying algorithm, however, this stack will be implemented using pointers of the list structure and the copy structure. The present stack then becomes an auxiliary structure.

A new notation has to be introduced to replace the occurrence of set b in the assertions. The notation $\mathcal{C}(t)$ - the contents of stack t is defined as follows:

$$\mathcal{C}(t) = \begin{cases} \varnothing & \text{if } t = nilst \\ \{pop(t)\}\cup\mathcal{C}(t') & \text{if } t\neq nilst \text{ and } t' \text{ is } t \text{ after } pop(t) \end{cases}$$

The algorithm resulting after the implementation of set b as a stack is given as algorithm *lm5* (figure 4.6).

The loop-condition in algorithm *lm5* is rather unwieldy. In combination with the final statement

```
f:=n; m:=∅; b:=∅; down:=true;
{f, n∈m∪{f} ∧ m∪{f}⊆R*(n) ∧
 R(m∪{f}−(b∪{f}))⊆m∪{f} ∧ b⊆m−{f} ∧
 f∉m↔down}
while not (b=∅ and R(f)⊆m∪{f})
do
   begin
      m:=m∪{f};
      if not R(f)⊆m
      then
         begin s:=x | x∈R(f)−m;
            b:=b∪{f}; f:=s; down:=true
         end
      else
         begin f:=x | x∈b; b:=b−{f};
            down:=false
         end
   end;
m:=m∪{f}
{m=R*(n)}
```

FIGURE 4.4: Algorithm *lm3*.

($m:=m\cup\{f\}$) this can be changed by application of transformation rule *L2*. When we define:

$$pop(nilst) = nil,$$

the new loop-condition B' becomes $f \neq nil$.

Statement S_1 is $\{m=R^*(n)\}$ (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, except for the proof of the fact that the new loop-invariant ensures termination of the loop after only one extra iteration.

If *down* is **false** the assertion

$$\neg B \;=\; empty(t) \wedge R(f)\subseteq m\cup\{f\}$$

ensures that $f:=pop(t)$ is executed. Since t is empty then f will be nil. If *down* is **true** then the assertion $\neg B$ ensures that $m:=m\cup\{f\}$; $f:=pop(t)$; $down:=$**false** is executed and f will be *nil* again. Hence the termination of the loop is assured with the new loop-invariant.

Now the statement $m:=m\cup\{f\}$ can be placed inside the **if**-statement again. The resulting algorithm *lm6* is shown in figure 4.7.

Algorithm *lm6* contains a lot of text twice, so some shortening of the algorithm might be achieved by combining a few branches. Such a combination is possible here by postponing one branch, the branch when *down* is **true** and $R(f) \subseteq m\cup\{f\}$. At the moment the transformation 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 transformation that does the job correctly is the deletion of the statement $f:=pop(t)$ in the case when *down* is **true** and next $R(f) \subseteq m$. 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. Thus it is sufficient to prove that:

```
f:=n; m:=∅; b:=∅; down:=true;
{f, n∈m∪{f} ∧ m∪{f}⊆R*(n) ∧
 R(m∪{f}−(b∪{f}))⊆m∪{f} ∧ b⊆m−{f} ∧
 f∉m↔down}
while not (b=∅ and R(f)⊆m∪{f})
do
   begin
      if down
      then
         begin
            m:=m∪{f};
            if not R(f)⊆m
            then
               begin s:=x|x∈R(f)−m;
                  b:=b∪{f}; f:=s
               end
            else
               begin f:=x|x∈b; b:=b−{f};
                  down:=false
               end
         end
      else
         begin
            if not R(f)⊆m
            then
               begin s:=x|x∈R(f)−m;
                  b:=b∪{f}; f:=s; down:=true
               end
            else
               begin f:=x|x∈b; b:=b−{f}
               end
         end
   end;
m:=m∪{f}
{m=R*(n)}
```

FIGURE 4.5: Algorithm *lm4*.

$$\{n \in m \cup \{f\} \ \wedge \ m \cup \{f\} \subseteq R^*(n) \ \wedge \ R(m \cup \{f\} - (\mathcal{C}(t) \cup \{f\})) \subseteq m \cup \{f\} \ \wedge \ \mathcal{C}(t) \subseteq m - \{f\} \ \wedge \ f \notin m \leftrightarrow down \ \wedge \ down\}$$

$$m := m \cup \{f\};$$

$$\{n \in m \ \wedge \ m \subseteq R^*(n) \ \wedge \ R(m - (\mathcal{C}(t) \cup \{f\})) \subseteq m \ \wedge \ \mathcal{C}(t) \subseteq m - \{f\} \ \wedge \ f \in m\}$$

$$down := \textbf{false}$$

$$\{n \in m \cup \{f\} \ \wedge \ m \cup \{f\} \subseteq R^*(n) \ \wedge \ R(m \cup \{f\} - (\mathcal{C}(t) \cup \{f\})) \subseteq m \cup \{f\} \ \wedge \ \mathcal{C}(t) \subseteq m - \{f\} \ \wedge \ f \notin m \leftrightarrow down\}$$

which is easily verified.

Informally it is clear that a loop can be added if and only if *down* is **true**. Since this occurs only once per node in the list the number of loops added is not greater than the number of nodes in the list. And that number is finite. Hence the loop terminates after this transformation if it terminates

```
f:=n; m:=∅; t:=nilst; down:=true;
{f, n∈m∪{f} ∧ m∪{f}⊆R*(n) ∧
 R(m∪{f}−(C(t)∪{f}))⊆m∪{f} ∧ C(t)⊆m−{f} ∧
 f∉m↔down}
while not (t=nilst and R(f)⊆m∪{f})
do
   begin
      if down
      then
         begin
            m:=m∪{f};
            if not R(f)⊆m
            then
               begin s:=x | x∈R(f)−m;
                  push(t,f); f:=s
               end
            else
               begin f:=pop(t);
                  down:=false
               end
         end
      else
         begin
            if not R(f)⊆m
            then
               begin s:=x | x∈R(f)−m;
                  push(t,f); f:=s; down:=true
               end
            else
               begin f:=pop(t)
               end
         end
   end;
m:=m∪{f}
{m=R*(n)}
```

FIGURE 4.6: Algorithm *lm5*.

before the transformation.

Algorithm *lm6* is still acting on abstract directed graphs. However we are interested in list structures with a left and a right pointer in every node. Every node in the list is a node in the corresponding graph and every non-nil pointer is an edge in the graph. Nil-pointers are not represented in the graph. Thus the relation R describing the edges is represented for a and b nodes as:

$$a\ R\ b \ \Leftrightarrow\ b \neq nil \ \wedge\ (b = a.l\uparrow \ \vee\ b = a.r\uparrow)$$

Now these concrete pointers and the corresponding tests will be introduced.

At the same time another feature of Robson's first list-marking algorithm will be added. This algorithm will leave behind information about the spanning tree it defines in every node. This is done by marking a node with a natural number between 0 and 3. 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

```
f:=n; m:=∅; t:=nilst; down:=true;
{f, n∈m∪{f} ∧ m∪{f}⊆R*(n) ∧
 R(m∪{f}−(𝒞(t)∪{f}))⊆m∪{f} ∧ 𝒞(t)⊆m−{f} ∧
 f∉m↔down}
while f≠nil
do
   begin
      if down
      then
         begin
            m:=m∪{f};
            if not R(f)⊆m
            then
               begin s:=x | x∈R(f)−m;
                  push(t,f); f:=s
               end
            else
               begin f:=pop(t);
                  down:=false
               end
         end
      else
         begin
            if not R(f)⊆m
            then
               begin s:=x | x∈R(f)−m;
                  push(t,f); f:=s; down:=true
               end
            else
               begin f:=pop(t)
               end
         end
   end
{m=R*(n)}
```

FIGURE 4.7: Algorithm *lm6*.

back pointer or a pointer to nil then the mark is not changed. Similarly 1 is added to the mark when the pointer to the right is added to the spanning tree.

Hence 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 after termination of the algorithm. This is summarized in figure 4.8.

		right pointer	
		forward	back or nil
left pointer	forward	3	2
	back or nil	1	0

FIGURE 4.8: Robson's spanning tree markers.

The algorithm accommodating these changes is given below in figure 4.9 as algorithm *lm7*.

```
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 ∧ Pointercode 1'}
while f≠nil
do
   begin
      if down
      then
         begin
            m:=m∪{f}; f.mk:=0;
            if not marked(f.l↑)
            then
               begin mark(f,2); s:=f.l;
                  push(t,f); f:=s
               end
            else
               if not marked(f.r↑)
               then
                  begin mark(f,1); s:=f.r;
                     push(t,f); f:=s
                  end
               else down:=false
         end
      else
         begin
            if not marked(f.r↑)
            then
               begin mark(f,1) s:=f.r;
                  push(t,f); f:=s; down:=true
               end
            else f:=pop(t)
   end
{m=R*(n)}
```

FIGURE 4.9: Algorithm *lm7*.

A few notations introduced in algorithm *lm7* need some explanation. The new assertion *Pointercode* 1' will be defined below. A field *mk* ;is added as an auxiliary variable to every node. Two functions are defined on this field. If a is an integer value and n a node then *mark*(n,a) is a shorthand notation for $n.mk := n.mk + a$, and *marked*(n) is shorthand for $n = nil \vee n.mk \in \{0,...3\}$.

The spanning tree defined by algorithm *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 forward pointers then the node is called an 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 *Pointercode* 1′ defines how this is effected:

$$\begin{aligned}
\textit{Pointercode}\,1' = \forall g \in m - \{f\}\ [\ g \in \mathcal{C}(t) - \{f\} \rightarrow (\textit{type}(g) \in \{\mathrm{FF},\mathrm{FN}\} \rightarrow g.mk = 2\ \wedge \\
\textit{type}(g) = \mathrm{NF} \rightarrow g.mk = 1\ \wedge \\
\textit{type}(g) = \mathrm{NN} \rightarrow g.mk = 0) \\
\wedge\ g \in m - (\mathcal{C}(t) \cup \{f\}) \rightarrow (\textit{type}(g) = \mathrm{FF} \rightarrow g.mk = 3\ \wedge \\
\textit{type}(g) = \mathrm{FN} \rightarrow g.mk = 2\ \wedge \\
\textit{type}(g) = \mathrm{NF} \rightarrow g.mk = 1\ \wedge \\
\textit{type}(g) = \mathrm{NN} \rightarrow g.mk = 0) \\
]\ \wedge \\
\neg \textit{down} \rightarrow (\textit{type}(f) \in \{\mathrm{FF},\mathrm{FN}\} \rightarrow f.mk = 2\ \wedge \\
\textit{type}(f) = \mathrm{NF} \rightarrow f.mk = 1\ \wedge \\
\textit{type}(f) = \mathrm{NN} \rightarrow f.mk = 0)
\end{aligned}$$

The verification of this 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, a superfluous action 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 *Pointercode* 1′ isn't valid any more however, since the treatment of type-NF nodes is changed. The new assertion *Pointercode* 1 is given below:

$$\begin{aligned}
\textit{Pointercode}\,1 = \forall g \in m - \{f\}\ [\ g \in \mathcal{C}(t) - \{f\} \rightarrow (\textit{type}(g) \in \{\mathrm{FF},\mathrm{FN}\} \rightarrow g.mk = 2\ \wedge \\
\textit{type}(g) \in \{\mathrm{NF},\mathrm{NN}\} \rightarrow g.mk = 0) \\
\wedge\ g \in m - (\mathcal{C}(t) \cup \{f\}) \rightarrow (\textit{type}(g) = \mathrm{FF} \rightarrow g.mk = 3\ \wedge \\
\textit{type}(g) = \mathrm{FN} \rightarrow g.mk = 2\ \wedge \\
\textit{type}(g) = \mathrm{NF} \rightarrow g.mk = 1\ \wedge \\
\textit{type}(g) = \mathrm{NN} \rightarrow g.mk = 0) \\
]\ \wedge \\
\neg \textit{down} \rightarrow (\textit{type}(f) \in \{\mathrm{FF},\mathrm{FN}\} \rightarrow f.mk = 2\ \wedge \\
\textit{type}(f) \in \{\mathrm{NF},\mathrm{NN}\} \rightarrow f.mk = 0)
\end{aligned}$$

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 first branch to the second has the following schematic form. First the algorithm was of the form:

```
{P'} while B do
  begin
    if down then
      begin T;
        if B' then S else S'
      end
    else
      begin
        if B'
        then begin S; down:=true end
        else S''
      end
  end
{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
  begin
    if down then
      begin T;
        if B' then down:=false else S'
      end
    else
      begin
        if B'
        then begin S; down:=true end
        else S''
      end
  end
{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''. In the second algorithm it becomes:

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

The next iteration will be:

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

Summing up the transformation amounts to the addition of the statement *down*:=**false** before S and the addition 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 algorithm 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 4.10) shows the result of these transformations.

```
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 ∧ Pointercode 1}
while f≠nil
do
   begin
      if down
      then
         begin
            m:=m∪{f}; f.mk:=0;
            if not marked(f.l↑)
            then
               begin mark(f,2); s:=f.l;
                  push(t,f); f:=s
               end
            else down:=false
         end
      else
         if odd(f.mk)
         then f:=pop(t) {up from right}
         else
            begin
               if not marked(f.r↑)
               then
                  begin mark(f,1) s:=f.r;
                     push(t,f); f:=s; down:=true
                  end
               else f:=pop(t)
            end
   end
{m=R*(n)}
```

FIGURE 4.10: 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 nodes and pointers of the original structure 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.

$$Samestack(t,gf) = \begin{cases} \textbf{if } down & Samestack'(t,gf) \\ \textbf{else if } gf=nil & t=nilst \\ \textbf{else if } odd(gf.mk) & Samestack'(t,gf.r\uparrow) \\ \textbf{otherwise} & Samestack'(t,gf.l\uparrow) \end{cases}$$

$$Samestack'(t,gf) = \begin{cases} \textbf{if } t=nilst \ \ gf=nil & \\ \textbf{otherwise} \ \ gf=pop(t) \wedge & \\ \qquad \textbf{if } odd(gf.mk) & \\ \qquad\qquad \textbf{then } Samestack'(t',gf.r\uparrow) & \\ \qquad\qquad \textbf{else } Samestack'(t',gf.l\uparrow) & \end{cases}$$

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.

Here for the first time the validation of the transformations is facilitated by the introduction of an auxiliary structure. The old pointer fields are bound to be changed frequently during the algorithm to implement the stack. Hence a scratch-pad is needed to keep track of the original values.

Every node is given an auxiliary field called a. Useful information for the proof will be stored in this node. At the moment the original pointer fields are the only information necessary. Their values will be stored in $a.l$ and $a.r$ at the moment a node is marked.

This makes it 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:

$$Loopinv\,1 = \begin{cases} n \in m \cup \{f\} \ \wedge \ m \cup \{f\} \subseteq R^*(n) \ \wedge \\ R(m - (\mathcal{C}(t) \cup \{f\})) \subseteq m \cup \{f\} \ \wedge \\ \mathcal{C}(t) \subseteq m - \{f\} \ \wedge \ f \notin m \leftrightarrow down \ \wedge \\ \neg down \rightarrow [(odd(f.mk) \rightarrow s = f.a\uparrow.r\uparrow) \ \wedge \\ \qquad (even(f.mk) \rightarrow s = f.a\uparrow.l\uparrow)] \end{cases}$$

The consequence of maintaining 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 off the stack last 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, a scratch-pad pointer for this purpose. The addition of the statements $s := f.l$; $f.l := gf$ does the trick.

The transformed version of algorithm *lm8* validating all these new assertions is algorithm *lm9* (figure 4.11). All auxiliary statements are set in a different (non-italic) font. Note that while execution of algorithm *lm8* depended heavily on the stack t, algorithm *lm9* will work without it. The verification of the new assertions is straightforward.

The first list-traversal of Robson's algorithm has another tasks besides the visit of every node without destruction of the structure. This second task is to leave behind the information necessary for a traversal in reverse order. The way this is done is stated in assertion *Pointercode* 1. This assertion describes all changes in marking. The final marking is described in assertion:

$$\begin{aligned} Pointercode\,1 \ = \ \forall g \in R^*(n) \ (&type(g) = \mathrm{FF} \leftrightarrow g.mk = 3 \ \wedge \\ &type(g) = \mathrm{FN} \leftrightarrow g.mk = 2 \ \wedge \\ &type(g) = \mathrm{NF} \leftrightarrow g.mk = 1 \ \wedge \\ &type(g) = \mathrm{NN} \leftrightarrow g.mk = 0) \end{aligned}$$

Since this part of the loop-invariant doesn't contradict the termination condition of the loop it may be included in the post-assertion.

This concludes the derivation of Robson's first list-marking algorithm.

4.4. Robson's second list-marking algorithm

The second traversal of the Robson list-copying algorithm is based on a special list-marking algorithm. As shown in the previous section Robson's first list-marking algorithm stores information about the spanning tree it defines in every node. The coding is given in assertion *Pointercode*.

The second list-marking algorithm uses this information to traverse the list following the same spanning tree in reverse order. Hence this algorithm is no real list-marking algorithm, since it needs a prepared list.

To formalize this we define a relation S from $R^*(n)$ to $R^*(n)$ describing the spanning tree edges.

```
f:=n; m:=∅; t:=nilst; gf:=nil; down:=true;
Loopinv1 ∧ Pointercode1 ∧ Samestack(t,gf)}
while f≠nil
do
  begin
    if down
    then
      begin
        m:=m∪{f}; f.mk:=0; a:=(f.l,f.r);
        if not marked(f.l↑)
        then
          begin mark(f,2); s:=f.l;
            push(t,f); f.l:=gf; gf:=f; f:=s
          end
        else
          begin
            s:=f.l; f.l:=gf; down:=false
          end
      end
    else
      if odd(f.mk)
      then
        begin
          gf:=f.r; f.r:=s;s:=f;f:=gf; f:=pop(t)
        end
      else
        begin
          gf:=f.l; f.l:=s;
          if not marked(f.r↑)
          then
            begin mark(f,1) s:=f.r;
              push(t,f); f.r:=gf;
              gf:=f; f:=s; down:=true
            end
          else
            begin
              s:=f; f:=gf; f:=pop(t)
            end
        end
  end
{m=R*(n) ∧ Pointercode}
```

FIGURE 4.11: Algorithm *lm9:* Robson's first list-traversal algorithm.

For nodes $p,\ q \in R^*(n)$:

$$p\,S\,q \;\Leftrightarrow\; q = p.l\uparrow \wedge type(p) \in \{\mathrm{FF},\mathrm{FN}\}$$
$$q = p.r\uparrow \wedge type(p) \in \{\mathrm{FF},\mathrm{NF}\}$$

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. This trivially holds for any subset of R.

Again it is possible to postpone the statement $f:=pop(t)$ in the case of $down=$**true** and $S(f)\subseteq m$. The same transformation was already proved correct in the preceding section.

Next concrete pointers have to be introduced to implement 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 descent to the left is postponed. The correctness proof of this transformation is similar, except that the problem with the replacement of assertion *Pointercode* 1′ doesn't occur since the mark fields remain unchanged.

The coding of the mark field allows easy determination of the necessity to descend to the right. This is the case when the mark is odd and one comes from above. Testing on descending to the left (the mark field should be greater than 1) gives some more problems. Coming from below in the search tree a node with mark 2 or 3 is encountered, the left subtree is marked, and coming from below a node with mark 2 or 3 is encountered again. Hence an infinite loop would result. For the result of the other transformations mentioned see algorithm *lm7'* (figure 4.12).

```
{Pointercode}
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≠nil
do
   begin
      if down
      then
         begin
            m:=m∪{f};
            if odd(f.mk)
            then
               begin s:=f.r;
                  push(t,f); f:=s
               end
            else down:=false
         end
      else
         begin
            if not S(f)⊆m
            then
               begin s:=f.l;
                  push(t,f); f:=s; down:=true
               end
            else f:=pop(t)
         end
   end
{m=S*(n)}
```

FIGURE 4.12: Algorithm *lm7'*.

Next the problem mentioned in the previous paragraph will be tackled. The easy solution to the question if a descent to the right still has to be made is to mark the node. Since it is not necessary to retain information about the node type if a descent will not be made any more, the old *mk*-field can be used. When it is checked if a descent to the left has to be made the mark is erased (by putting -1 into it). So all siblings of an unmarked node are visited when this node is reached again by backing up. The following function explains itself:

$$marked(f) \Leftrightarrow f.mk \in \{0,\ldots 3\}$$

Of course the assertion *Pointercode* is invalidated when the mark field is erased. So a new assertion *Pointercode*2 is necessary:

$$\begin{aligned} Pointercode2 = \forall g \in S^*(n)\ (&g.mk=3 \rightarrow type(g)=\mathrm{FF} \wedge \\ &g.mk=2 \rightarrow type(g)=\mathrm{FN} \wedge \\ &g.mk=1 \rightarrow type(g)=\mathrm{NF} \wedge \\ &g.mk=0 \rightarrow type(g)=\mathrm{NN} \wedge \\ &\neg marked(g) \rightarrow S(g) \subseteq m) \end{aligned}$$

Now it is possible to replace the **else**-clause in the main loop by the following statement:

```
else
  begin
    if marked(f)
    then
      begin
        mark:=f.mk; f.mk:=-1;
        if mark≥2
        then
          begin s:=f.l;
            push(t,f); f:=s; down:=true
          end
        else f:=pop(t)
      end
    else f:=pop(t)
  end
```

Verification of the old loop-invariant's preservation with assertion *Pointercode*1 replaced by assertion *Pointercode*2 is straightforward.

Finally, the stack t has to be included in the list structure again. To describe the implementation an assertion *Samestack*2 is defined (with t a stack and gf a pointer):

$$Samestack2(t,gf) = \begin{cases} \textbf{if } down & Samestack''(t,gf) \\ \textbf{else if } gf=nil & t=nilst \\ \textbf{else if } marked(gf) & Samestack''(t,gf.r\uparrow) \\ \textbf{otherwise} & Samestack''(t,gf.l\uparrow) \end{cases}$$

$$Samestack''(t,gf) = \begin{cases} \textbf{if } t=nilst\ \ gf=nil & \\ \textbf{otherwise}\ \ gf=pop(t) \wedge & \\ \qquad \textbf{if } marked(gf) & \\ \qquad\qquad \textbf{then } Samestack''(t',gf.r\uparrow) & \\ \qquad\qquad \textbf{else } Samestack''(t',gf.l\uparrow) & \end{cases}$$

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 in $s1$. To be able to talk about $s1$ we need the auxiliary field of the last section again. The loop-invariant then includes:

$$Loopinv2 = \begin{cases} n \in m \cup \{f\} \ \wedge \ m \cup \{f\} \subseteq S^*(n) \ \wedge \\ S(m - (\mathcal{C}(t) \cup \{f\})) \subseteq m \cup \{f\} \ \wedge \\ \mathcal{C}(t) \subseteq m - \{f\} \ \wedge \ f \notin m \leftrightarrow down \ \wedge \\ \neg down \rightarrow [(marked(f) \rightarrow s1 = f.a\uparrow.r\uparrow) \ \wedge \\ \qquad (\neg marked(f) \rightarrow s1 = f.a\uparrow.l\uparrow)] \end{cases}$$

This clears the way 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. Again, auxiliary variables are set in a different font.

The verification of the loop-invariant contains no special problems. The post-assertion is $m = S^*(n)$. Since relation S is defined such that $S^*(n) = R^*(n)$ this algorithm correctly marks the list defined by root n and relation R.

This concludes the derivation of Robson's second list-marking algorithm.

```
{Pointercode}
f:=n; m:=∅; t:=nilst; down:=true;
{Loopinv2 ∧ Pointercode2 ∧ Samestack2(t,gf)}
while f≠nil
do
  begin
    if down
    then
      begin
        m:=m∪{f}; a:=(f.l,f.r)
        if odd(f.mk)
        then
          begin
            s:=f.r; f.r:=gf; gf:=f; push(t,f); f:=s
          end
        else
          begin
            s1:=f.r; f.r:=gf; down:=false
          end
      end
    else
      begin
        if marked(f)
        then
          begin
            mark:=f.mk; f.mk:=-1;
            if mark≥2
            then
              begin
                s:=f.l; f.l:=f.r; f.r:=s1; gf:=f;
                push(t,f); f:=s; down:=true
              end
            else
              begin
                gf:=f.r; f.r:=s1;s1:=f; f:=gf; f:=pop(t)
              end
          end
        else
          begin
            gf:=f.l; f.l:=s1;s1:=f; f:=gf; f:=pop(t)
          end
      end
  end
{m=S*(n)}
```

FIGURE 4.13: Algorithm *lm8':* Robson's second list-traversal algorithm.

4.5. 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. The left pointer field will be used to store 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 ponter field in the original structure will result in the loss of the knowledge which copy node corresponds 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 structure again an auxiliary node is assigned to every node in the original structure. The information stored in this node is a reference to the original node (*on*), to the copy node (*cn*), and to the left and right siblings of the original node (*l* and *r*).

The first step towards the final copying algorithm is algorithm *rlc'* (figure 4.14). A pointer *c* is introduced which will point to the copy node corresponding to the node *f* in the original structure. The final result will be stored in the pointer *copy*.

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 initialized as described. Changes of the pointer fields in the original structure are echoed in the copy structure.

The values of the pointers in the copy structure have to be set to their final value in the second loop, essentially *lm8'*. One has to keep in mind 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 just traversed. So if the pointers in the copy nodes corresponding to the nodes in the subtree with root $s1$ are correct copies, then (by induction on the structure of the tree) the correct copy of the pointer to $s1$ is the pointer to $s2$. Note that a copy of a nil pointer is a nil pointer.

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 - a mark field and a pointer to the copy per original node -. 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 put in the original left pointer fields. This takes several steps.

Control of the traversal sequence has to be changed from the pointers in the original nodes to the pointers in the matching copy nodes where the same information is stored. 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 since a branch need not be traversed twice.

Then the left pointer in the original is set to the copy node. Assignations to both the left and the right pointers 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 pointer in the original is retained.

In the case of the right pointer (which isn't changed) this only marks the last possibility to restore the correct value. For both pointers this is the moment of return from the descent in the subtree originally pointed to.

It should be noted that information about the corresponding copy nodes is no longer necessary in pass two in a fully traversed subtree of the spanning tree defined in pass one. The remainder of the original graph can only contain back pointers to this subtree. A back pointer is defined as a pointer to a node already marked before in pass one. Since pass two uses the opposite traversal order no nodes with back pointers to the present node can be encountered if it was passed for the last time. They must be traversed earlier. This observation is the salient reason why 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 the correct values have to be stored in the left pointer fields in original and copy.

Lastly all references to the copy node via the auxiliary node have to be replaced by the direct reference via the left pointer field. The resulting algorithm is listed in figure 4.15 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 special array *mk*. This is realized by the following declarations:

```
var mk: array[0..3] ↑node;

function mrk(n: node): integer;
  var i: integer;
  begin
    i:=0;
    while (i<4) and (n.r↑≠mk[i]) do i:=i+1;
    mrk:=i
  end;

function marked(n: node): boolean;
begin marked:=(mrk(n)<4) end;

procedure mark(i: integer; var n: node);
begin n.r:=mk[mrk(n)+i] end;
```

An array of pointers to nodes is declared. The index in this array gives the value of the mark, retrievable by function *mrk*. A correctness proof of this implementation in Jones' style [10] is straightforward.

This implementation is shown in figure 4.16 as algorithm *rlc*. The initialization of the *mk* field is replaced by an assignation to the right pointer field of *f*. All tests on *f.mk* are replaced by tests on *mrk(f)*. The statement erasing the mark field is replaced by $f.r:=s1$, already necessary to retrieve the correct value in this pointer. The correctness proof poses no special problems.

Deletion of all auxiliary variables - set in a different font - results in Robson's list-copying algorithm proper.

```
f:=n; m:=∅; gf:=nil; down:=true;
{LoopinvListmark 1′ ∧ LoopinvListcopy 1′}
while f≠nil
do
   begin
      if down
      then
         begin
            m:=m∪{f}; new(c); c.l:=f.l c.r:=f.r; f.mk:=0; f.a:=(f,c,f.l,f.r);
            if not marked(f.l↑)
            then
               begin mark(f,2); s:=f.l;
                  f.l:=gf; c.l:=gf; gf:=f; f:=s
               end
            else
               begin
                  s:=f.l; f.l:=gf; c.l:=gf; down:=false
               end
         end
      else
         begin
            c:=f.a.cn;
            if odd(f.mk)
            then
               begin
                  gf:=f.r; f.r:=s; c.r:=s; s:=f; f:=gf
               end
            else
               begin
                  gf:=f.l; f.l:=s; c.l:=s;
                  if not marked(f.r↑)
                  then
                     begin mark(f,1) s:=f.r;
                        f.r:=gf; c.r:=gf;
                        gf:=f; f:=s; down:=true
                     end
                  else
                     begin
                        s:=f; f:=gf
                     end
               end
         end
   end;
{PostListcopy 1′}
f:=n; m:=∅; gf:=nil; down:=true; copy:=n.a.cn;
```

```
{LoopinvListmark2' ∧ LoopinvListcopy2'}
while f≠nil
do
   begin
      c:=f.a.cn;
      if down
      then
         begin
            m:=m∪{f};
            if odd(f.mk)
            then
               begin
                  s:=f.r; f.r:=gf; c.r:=gf; gf:=f; f:=s
               end
            else
               begin
                  s1:=f.r;
                  if s1=nil then s2:=nil else s2:=s1.a.cn;
                  f.r:=gf; c.r:=gf; down:=false
               end
         end
      else
         begin
            if marked(f)
            then
               begin
                  mark:=f.mk; f.mk:=-1;
                  if mark≥2
                  then
                     begin
                        s:=f.l; f.l:=f.r; c.l:=c.r;
                        f.r:=s1; c.r:=s2; gf:=f;
                        f:=s; down:=true
                     end
                  else
                     begin
                        gf:=f.r; f.r:=s1; c.r:=s2; s1:=f;
                        s2:=c; f:=gf
                     end
               end
            else
               begin
                  gf:=f.l; f.l:=s1; c.l:=s2; s1:=f;
                  s2:=c; f:=gf
               end
         end
   end
{PostListcopy2'}
```

FIGURE 4.14: Algorithm *rlc'*:
The assertions are described in figure 4.17.

```
f:=n; m:=∅; gf:=nil; down:=true;
LoopinvListmark 1'' ∧ LoopinvListcopy 1''}
while f≠nil
do
   begin
      if down
      then
         begin
            m:=m∪{f}; new(c); c.l:=f.l; c.r:=f.r; f.mk:=0; f.a:=(f,c,f.l,f.r);
            f.l:=c;
            if not marked(c.l↑)
            then
               begin mark(f,2); s:=c.l;
                  c.l:=gf; gf:=f; f:=s
               end
            else
               begin
                  s:=c.l; c.l:=gf; down:=false
               end
         end
      else
         begin
            c:=f.l;
            if odd(f.mk)
            then
               begin
                  gf:=c.r; c.r:=s; s:=f; f:=gf
               end
            else
               begin
                  gf:=c.l; c.l:=s;
                  if not marked(c.r↑)
                  then
                     begin mark(f,1) s:=c.r; c.r:=gf;
                        gf:=f; f:=s; down:=true
                     end
                  else
                     begin
                        s:=f; f:=gf
                     end
               end
         end
   end;
{PostListcopy 1''}
f:=n; m:=∅; gf:=nil; down:=true; copy:=n.l;
```

```
{LoopinvListmark 2'' ∧ LoopinvListcopy 2''}
while f≠nil
do
  begin
    c:=f.l;
    if down
    then
      begin
        m:=m∪{f};
        if odd(f.mk)
        then
          begin
            s:=c.r; c.r:=gf; gf:=f; f:=s
          end
        else
          begin
            s1:=c.r; if s1=nil then s2:=nil else s2:=s1.l;
            c.r:=gf; down:=false
          end
      end
    else
      begin
        if marked(f)
        then
          begin
            mark:=f.mk; f.mk:=−1;
            if mark≥2
            then
              begin
                s:=c.l; c.l:=c.r; f.r:=s1; c.r:=s2;
                gf:=f; f:=s; down:=true
              end
            else
              begin
                gf:=c.r; f.r:=s1; c.r:=s2; s:=c.l;
                if s=nil then f.l:=nil
                else begin c.l:=s.l; f.l:=s end;
                s1:=f; s2:=c; f:=gf
              end
          end
        else
          begin
            gf:=c.l; f.l:=s1; c.l:=s2; s1:=f;
            s2:=c; f:=gf
          end
      end
  end
{PostListcopy 2''}
```

FIGURE 4.15: Algorithm *rlc'':*
The assertions are described in figure 4.17.

```
f:=n; m:=∅; gf:=nil; down:=true;
LoopinvListmark 1 ∧ LoopinvListcopy 1}
while f≠nil
do
   begin
      if down
      then
         begin
            m:=m∪{f}; new(c); c.l:=f.l; c.r:=f.r; f.r:=mk[0]; f.a:=(f,c,f.l,f.r);
            f.l:=c;
            if not marked(c.l↑)
            then
               begin mark(f,2); s:=c.l;
                  c.l:=gf; gf:=f; f:=s
               end
            else
               begin
                  s:=c.l; c.l:=gf; down:=false
               end
         end
      else
         begin
            c:=f.l;
            if odd(f.mk)
            then
               begin
                  gf:=c.r; c.r:=s; s:=f; f:=gf
               end
            else
               begin
                  gf:=c.l; c.l:=s;
                  if not marked(c.r↑)
                  then
                     begin mark(f,1) s:=c.r; c.r:=gf;
                        gf:=f; f:=s; down:=true
                     end
                  else
                     begin
                        s:=f; f:=gf
                     end
               end
         end
   end;
{PostListcopy 1}
f:=n; m:=∅; gf:=nil; down:=true; copy:=n.l;
```

```
{LoopinvListmark2 ∧ LoopinvListcopy2}
while f≠nil
do
  begin
    c:=f.l;
    if down
    then
      begin
        m:=m∪{f};
        if odd(f.mk)
        then
          begin
            s:=c.r; c.r:=gf; gf:=f; f:=s
          end
        else
          begin
            s1:=c.r; if s1=nil then s2:=nil else s2:=s1.l;
            c.r:=gf; down:=false
          end
      end
    else
      begin
        if marked(f)
        then
          begin
            mark:=f.mk; f.r:=s1;
            if mark≥2
            then
              begin
                s:=c.l; c.l:=c.r; c.r:=s2;
                gf:=f; f:=s; down:=true
              end
            else
              begin
                gf:=c.r; c.r:=s2; s:=c.l;
                if s=nil then f.l:=nil
                else begin c.l:=s.l; f.l:=s end;
                s1:=f; s2:=c; f:=gf
              end
          end
        else
          begin
            gf:=c.l; f.l:=s1; c.l:=s2; s1:=f;
            s2:=c; f:=gf
          end
      end
  end
{PostListcopy2}
```

FIGURE 4.16: Algorithm *rlc:* Robson's list-copying algorithm.
The assertions are described in figure 4.17.

$$
\begin{aligned}
\mathit{LoopinvListmark}\,1' &= \forall t\ (\mathit{Samestack}\,(t,gf) \rightarrow (\mathit{Loopinv}\,1 \wedge \mathit{Pointercode}\,1)) \\
\mathit{LoopinvListcopy}\,1' &= \forall g \in m\ (g.l = g.a.cn.l\ \wedge \\
&\qquad\qquad g.r = g.a.cn.r) \\
\mathit{PostListcopy}\,1' &= \forall g \in R^*(n)\ (g.l = g.a.cn.l\ \wedge \\
&\qquad\qquad\qquad g.r = g.a.cn.r\ \wedge \\
&\qquad\qquad\qquad \mathit{Pointercode}) \\
\mathit{LoopinvListmark}\,2' &= \forall t\ (\mathit{Samestack}\,2(t,gf) \rightarrow (\mathit{Loopinv}\,2 \wedge \mathit{Pointercode}\,2)) \\
\mathit{LoopinvListcopy}\,2' &= \forall g \in R^*(n)[g \notin m \rightarrow \\
&\qquad\qquad\qquad (g.l = g.a.cn.l\ \wedge \\
&\qquad\qquad\qquad\ g.r = g.a.cn.r)\ \wedge \\
&\qquad\qquad (g \in m \wedge \mathit{marked}(g)) \rightarrow \\
&\qquad\qquad\qquad (g.l = g.a.cn.l\ \wedge \\
&\qquad\qquad\qquad\ g.r = g.a.cn.r\ \wedge \\
&\qquad\qquad\qquad\ [g = f \wedge \neg \mathit{down}] \rightarrow s\,2 = \mathit{copy}\,(s\,1))\ \wedge \\
&\qquad\qquad (g \in m \wedge \neg \mathit{marked}(g)) \rightarrow \\
&\qquad\qquad\qquad (g.a.cn.r = \mathit{copy}\,(g.r)\ \wedge \\
&\qquad\qquad\qquad\ \forall t\ \mathit{Samestack}\,2(t,gf) \rightarrow \\
&\qquad\qquad\qquad\ [(g \in \mathcal{C}(t) \rightarrow g.l = g.a.cn.l)\ \wedge \\
&\qquad\qquad\qquad\ (g = f \wedge \neg \mathit{down} \rightarrow s\,2 = \mathit{copy}\,(s\,1))\ \wedge \\
&\qquad\qquad\qquad\ (g \notin \mathcal{C}(t) \cup \{f\} \rightarrow g.a.cn.l = \mathit{copy}\,(g.l))]\ ) \\
&\qquad\qquad] \\
\mathit{PostListcopy}\,2' &= \forall g \in R^*(n)\ (g.a.cn.l = \mathit{copy}\,(g.l)\ \wedge \\
&\qquad\qquad\qquad g.a.cn.r = \mathit{copy}\,(g.r)\ ) \\
\mathit{copy}\,(n) &= \begin{cases} \mathit{nil} & \textbf{if } n = \mathit{nil} \\ n & \textbf{if } \mathit{atom}\,(n) \\ n.a.cn & \textbf{otherwise} \end{cases} \\
\mathit{LoopinvListmark}\,1'' &= \forall t\ (\mathit{SamestackA}\,(t,gf) \rightarrow (\mathit{Loopinv}\,1 \wedge \mathit{Pointercode}\,1)) \\
\mathit{LoopinvListcopy}\,1'' &= \forall g \in m\ (g.l = g.a.cn\ \ \wedge \\
&\qquad\qquad g.r = g.a.cn.r) \\
\mathit{PostListcopy}\,1'' &= \forall g \in R^*(n)\ (g.l = g.a.cn\ \ \wedge \\
&\qquad\qquad\qquad g.r = g.a.cn.r\ \wedge \\
&\qquad\qquad\qquad \mathit{Pointercode})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{LoopinvListmark}\,2'' &= \forall t\ (\mathit{SamestackB}\,(t,gf) \rightarrow (\mathit{Loopinv}\,2 \wedge \mathit{Pointercode}\,2)) \\
\mathit{LoopinvListcopy}\,2'' &= \forall g \in R^*(n)[g \notin m \rightarrow \\
&\qquad (g.l = g.a.cn \ \wedge \\
&\qquad\ g.r = g.a.cn.r) \ \wedge \\
&\qquad (g \in m \wedge \mathit{marked}(g)) \rightarrow \\
&\qquad\ (g.l = g.a.cn \ \wedge \\
&\qquad\ g.r = g.a.cn.r \ \wedge \\
&\qquad\ [g = f \wedge \neg \mathit{down}] \rightarrow s\,2 = \mathit{copy}\,(s\,1)\,) \ \wedge \\
&\qquad (g \in m \wedge \neg \mathit{marked}(g)) \rightarrow \\
&\qquad\ (g.a.cn.r = \mathit{copy}\,(g.r) \ \wedge \\
&\qquad\ \forall t\ \mathit{SamestackB}(t,gf) \rightarrow \\
&\qquad\ [(g \in \mathcal{C}(t) \rightarrow g.l = g.a.cn) \ \wedge \\
&\qquad\ (g = f \wedge \neg \mathit{down} \rightarrow s\,2 = \mathit{copy}\,(s\,1)) \ \wedge \\
&\qquad\ (g \notin \mathcal{C}(t) \cup \{f\} \rightarrow g.a.cn.l = \mathit{copy}\,(g.l))]\,) \\
&\qquad] \\
\mathit{PostListcopy}\,2'' &= \mathit{PostListcopy}\,2' \\
\mathit{SamestackA} &= \mathit{Samestack}\,[gf.cn.r\ /\ gf.r,\ gf.cn.l\ /\ gf.l] \\
\mathit{SamestackB} &= \mathit{Samestack}\,2[gf.cn.r\ /\ gf.r,\ gf.cn.l\ /\ gf.l] \\
\mathit{LoopinvListmark}\,1 &= \mathit{LoopinvListmark}\,1'' \\
\mathit{LoopinvListcopy}\,1 &= \forall g \in m\ (g.l = g.a.cn \ \wedge \\
&\qquad\ \mathit{mrk}(g) = g.mk) \\
\mathit{LoopinvListmark}\,2 &= \mathit{LoopinvListmark}\,2'' \\
\mathit{LoopinvListcopy}\,2 &= \forall g \in R^*(n)[g \notin m \rightarrow \\
&\qquad (g.l = g.a.cn \ \wedge \\
&\qquad\ \mathit{mrk}(g) = g.mk) \ \wedge \\
&\qquad (g \in m \wedge \mathit{marked}(g)) \rightarrow \\
&\qquad\ (g.l = g.a.cn \ \wedge \\
&\qquad\ \mathit{mrk}(g) = g.mk \ \wedge \\
&\qquad\ [g = f \wedge \neg \mathit{down}] \rightarrow s\,2 = \mathit{copy}\,(s\,1)\,) \ \wedge \\
&\qquad (g \in m \wedge \neg \mathit{marked}(g)) \rightarrow \\
&\qquad\ (g.a.cn.r = \mathit{copy}\,(g.r) \ \wedge \\
&\qquad\ \forall t\ \mathit{SamestackB}(t,gf) \rightarrow \\
&\qquad\ [(g \in \mathcal{C}(t) \rightarrow g.l = g.a.cn) \ \wedge \\
&\qquad\ (g = f \wedge \neg \mathit{down} \rightarrow s\,2 = \mathit{copy}\,(s\,1)) \ \wedge \\
&\qquad\ (g \notin \mathcal{C}(t) \cup \{f\} \rightarrow g.a.cn.l = \mathit{copy}\,(g.l))]\,) \\
&\qquad] \\
\mathit{PostListcopy}\,2 &= \mathit{PostListcopy}\,2''
\end{aligned}
$$

FIGURE 4.17: Assertions from figures 4.14, 4.15 and 4.16.

5. FISHER'S LIST-COPYING ALGORITHM

5.1. Introduction

The first list-copying algorithm using linear time and bounded workspace, Fisher's algorithm was a leap forward compared with some algorithms by Lindstrom [12]. Lindstroms algorithms could reach linear time only in the case of a structure without cycles, and that at the expense of a tag field. Bounded workspace was attainable, however at the cost of order N^2 time, where N is the number of nodes to be copied. Fisher's only constraint was on the location of the copy structure in memory: it should be placed in a contiguous block. A reasonable price to pay.

Since then Fisher's algorithm has been bested 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 went astray in the informal introduction.

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 algorithm.

5.2. 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 section. The present starting point is algorithm *lm2* (figure 4.3).

First transformation rule $L2$ will be used to move the statement $m:=m\cup\{f\}$ into the main loop. This requires some preparation. A boolean variable *finished* with obvious meaning is introduced. Then the **else**-clause $f:=x\,|\,x\in b;\ b:=b-\{f\}$ is replaced by **if** $b=\varnothing$ **then** *finished*$:=$**true else begin** $f:=x\,|\,x\in b;\ b:=b-\{f\}$ **end**. Since the validity of the looptest followed by the execution of $m:=m\cup\{f\}$ and the negation of the case-test ensure that $b\neq\varnothing$ 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 when $b=\varnothing$ 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 $b=\varnothing$ and $R(f)\subseteq m\cup\{f\}$, and that *finished* will be **true** if the new loop-body is executed while $b=\varnothing$ 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 $\neg$ *finished* as the new loop-test.

Since it is shown to be impossible for *finished* to be **true** unless the loop terminates the postcondition 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 changed to $R(f)\subseteq m\cup\{f\}$, of course.

Application of transformation rule *L3* splits the interior of the loop into two loops. The first one is:

```
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
   begin
      m:=m∪{f};
      if not R(f)⊆m
      then
         begin s:=x|x∈R(f)−m;
            b:=b∪{f}; f:=s;
         end
      else
         if b=∅
         then finished:=true;
         else
            begin f:=x|x∈b; b:=b−{f}
            end
   end
{m=R*(n)}
```

FIGURE 5.1: Algorithm *lm3f*.

```
while not (finished or R(f)⊆m∪{f})
do
   begin
      m:=m∪{f};
      s:=x|x∈R(f)−m;
      b:=b∪{f}; f:=s
   end
```

Since *finished* is **false** when this statement is executed for the first time, and since this value is not changed during execution of this statement, the loop-test reduces to the second component.

The second interior loop is:

```
loop
   m:=m∪{f};
   if b=∅
   then finished:=true;
   else
      begin f:=x|x∈b; b:=b−{f}
      end
until finished or not R(f)⊆m∪{f}
```

Since $f \in m$ after execution of $f:=x|x\in b$; the last test is equivalent to **not** $R(f)\subseteq m$. 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, if this test is 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.

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-

```
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
   begin
      while
         m:=m∪{f};
         not R(f)⊆m
      do
         begin s:=x | x∈R(f)−m;
            b:=b∪{f}; f:=s
         end;
      loop
         if b=∅
         then finished:=true;
         else
            begin f:=x | x∈b; b:=b−{f}
            end
      until finished or not R(f)⊆m
   end
{m=R*(n)}
```

FIGURE 5.2: Algorithm *lm4f*.

pointer. Then it is possible to replace the abstract test **not**$R(f)\subseteq m$ 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, if it exists. 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. This 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 the set b as a 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 nodes in the final copy list.

The correctness proof gives no problems when it is done in the style of Jones [10]. 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, $\mathcal{C}(q)$ is defined as:

$$\mathcal{C}(q) = \begin{cases} \varnothing & \text{if } q = nilq \\ \{deq(q)\} \cup \mathcal{C}(q') & \text{if } q \neq nilq \text{ and } q' \text{ is } q \text{ after } deq(q) \end{cases}$$

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 is via a stack.

```
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
   begin
      m:=m∪{f};
      while
         m:=m∪{f};
         f.r↑∉m
      do
         begin
            b:=b∪{f}; f:=f.r
         end;
      loop
         if b=∅
         then finished:=true;
         else
            begin f:=x|x∈b; b:=b−{f}
            end
      until finished or f.l↑∉m;
      if ¬ finished then f:=f.l↑
   end
{m=R*(n)}
```

FIGURE 5.3: Algorithm *lm5f.*

5.3. 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 the organization of memory allocation is beyond the scope of this text the various properties following from this demand will be treated abstractly. Also no allowance will be made for the possibility to run out of memory.

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 whether a node is in the copy part of memory. So *next*(Mem) is the first available location in the copy part of memory.

Then algorithm *flc'* (figure 5.5), the starting point of the derivation, can be introduced. Pointer *copy* is used to point to the root of the result structure. Queue *invq* is a queue in which noded fetched from queue *q* are stored in inverse order.

In the first loop node *c* is set to the next available memory space. An auxiliary structure 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 will point to the next field in the copy space.

The second traversal allocates the right pointer field in the copy structure to the task of maintaining the link between original and copy node. It also builds the queue *invq* in correct order. The queue-handling primitive needed for this is *addq*, which adds a node at the front of the queue. The correctness of this additional primitive will be assumed again.

The last loop empties the queue *invq* while setting the right pointer fields in the copy structure to

```
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
   begin
      while
         m:=m∪{f};
         f.r↑∉m
      do
         begin
            enq(q,f); f:=f.r
         end;
      loop
         if q=nilq
         then finished:=true;
         else f:=deq(q)
      until finished or f.l↑∉m;
      if ¬ finished then f:=f.l↑
   end
{m=R*(n)}
```

FIGURE 5.4: Algorithm *lm6f*.

their final value. 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*. They allow definition of two queues in memory, starting with a certain node qp.

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

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

The last space used in the part of memory reserved for the copy is pointed to by pointer 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 in which 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 the economy in memory space for the queues. Additionally 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 $f \in m$ is

equivalent to the test *iscopy*($f.r\uparrow$).

After the first loop the left pointers in both original and copy nodes are correctly filled in. The right pointer of every original node is pointing to its assigned copy. If the original right pointer was pointing forward in the spanning tree it 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 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 set in a different font. Deletion of these statements produces the original Fisher list-copying algorithm.

```
f:=n; m:=∅; q:=nilq; finished:=false; c:=Mem; copy:=next(c);
{LoopinvListmark 1' ∧ LoopinvListcopy 1'}
while not finished
do
  begin
    while
      m:=m∪{f}; c:=next(c); f.a:=(f,c,f.l,f.r);
      if f.r↑∈m then c.r:=f.r.a.cn elsec.r:=f.r;
      f.r↑∉m
    do
      begin
        enq(q,f); f:=f.r
      end;
    loop
      if q=nilq
      then finished:=true
      else
        begin
          f:=deq(q); c 1:=f.a.cn;
          if f.l∈m
          then c 1.l:=f.l↑.a.cn
          else
            if atom(f.l)
            then c.l:=f.l
            else c.l:=next(c)
        end
    until finished or f.l↑∉m;
    if ¬ finished then f:=f.l↑
  end;
{PostListcopy 1'}
```

```
f:=n; m:=∅; q:=nilq; invq:=nilq; finished:=false;
{LoopinvListmark 2' ∧ LoopinvListcopy 2'}
while not finished
do
   begin
      while
         m:=m∪{f}; c1:=f.a.cn; c1.r:=f;
         f.r↑∉m
      do
         begin
            enq(q,f); f:=f.r
         end;
      loop
         if q=nilq
         then finished:=true;
         else begin f:=deq(q); addq(invq,f) end
      until finished or f.l↑∉m;
      if ¬ finished then f:=f.l↑
   end;
{PostListcopy 2'}
loop
   f:=deq(invq); c1:=f.a.cn;
   if atom(f.r)
   then c1.r:=f.r
   else c1.r:=f.r.cn
until invq=nilq
{PostListcopy 3'}
```

FIGURE 5.5: Algorithm *flc'*.
The various assertions are described in figure 5.7

```
f:=n; m:=∅; q:=nilq; finished:=false;
c:=Mem; copy:=next(c); qp:=copy;
{LoopinvListmark 1 ∧ LoopinvListcopy 1}
while not finished
do
   begin
      while
         m:=m∪{f}; c:=next(c); f.a:=(f,c,f.l,f.r);
         c 1:=f.r; f.r:=c; c.l:=f.l;
         if iscopy(c 1.r↑) then c.r:=c 1.r else c.r:=c 1;
         iscopy(c 1.r↑)
      do f:=c 1;
      bool:=false;
      loop
         if q=nilq
         then finished:=true
         else
            begin
               c 1:=qp; qp:=next(qp);
               if notatom(c 1.l)
               then
                  if iscopy(c 1.l↑.r)
                  then c 1.l:=c 1.l↑.r
                  else begin c 1.l:=next(c); bool:=true end
      until finished or bool;
      if ¬ finished then f:=f.l↑
   end;
{PostListcopy 1}
f:=n; m:=∅; finished:=false; qp:=copy;
{LoopinvListmark 2 ∧ LoopinvListcopy 2}
while not finished
do
   begin
      bool:=true;
      while
         m:=m∪{f}; c 1:=f.r; f.r:=c 1.r; c 1.r:=f;
         if atom(f.r↑)
         then bool:=false
         else bool:=iscopy(f.r↑.r);
         bool
      do f:=f.r;
      bool:=false;
      loop
         if qp=next(c)
         then finished:=true
         else
            begin
               c 1:=qp; qp:=next(qp);
               if not atom(c 1.l↑) then bool:=iscopy(c 1.l↑.r)
            end
      until finished or bool;
```

```
        if ¬ finished then f:=c1.l↑
    end;
{PostListcopy 2}
loop
    qp:=prev(qp); c1:=qp; f:=c1.r;
    if atom(f.r)
    then c1.r:=f.r
    else
        begin
            if iscopy(f.r↑)
            then begin c:=f.r; f.r:=c1.r; c1.r:=c end
            else c1.r:=next(c1)
until qp=copy
{PostListcopy 3}
```

FIGURE 5.6: Fisher's list-copying algorithm.
The various assertions are described in figure 5.7

$$
\begin{aligned}
\mathit{LoopinvListmark}\,1' &= n \in m \cup \{f\} \ \wedge\ m \cup \{f\} \subseteq R^*(n) \ \wedge\ R(m-(\mathcal{C}(q)\cup\{f\})) \subseteq m \cup \{f\} \\
&\quad \wedge\ \mathcal{C}(q) \subseteq m-\{f\} \ \wedge\ \mathit{finished} \rightarrow m = R^*(n) \\
\mathit{LoopinvListcopy}\,1' &= \forall g \in m\ [(g.r\uparrow.a.cn = \mathit{next}(g.a.cn) \rightarrow g.a.cn.r = g.r) \wedge \\
&\quad (g.r\uparrow.a.cn \neq \mathit{next}(g.a.cn) \rightarrow g.a.cn.r = g.r\uparrow a.cn) \wedge \\
&\quad g \notin \mathcal{C}(q) \rightarrow g.a.cn.l = g.a.l\uparrow.cn)] \\
\mathit{PostListcopy}\,1' &= \forall g \in R^*(n)\ [(g.r\uparrow.a.cn = \mathit{next}(g.a.cn) \rightarrow g.a.cn.r = g.r) \wedge \\
&\quad (g.r\uparrow.a.cn \neq \mathit{next}(g.a.cn) \rightarrow g.a.cn.r = g.r\uparrow a.cn) \wedge \\
&\quad g.a.cn.l = g.a.l\uparrow.cn)] \\
\mathit{LoopinvListmark}\,2' &= \mathit{LoopinvListmark}\,1' \ \wedge\ \forall g \in \mathcal{C}(\mathit{invq})\ \forall m > 1 \\
&\quad (g = \mathit{deq}^m(\mathit{invq}) \rightarrow \mathit{next}(g.a.cn) = [\mathit{deq}^{m-1}(\mathit{invq})].cn) \\
\mathit{LoopinvListcopy}\,2' &= \forall g \in R^*(n)[g.a.cn.l = g.a.l\uparrow.cn \ \wedge \\
&\quad g \notin m \rightarrow \\
&\quad\quad [(g.r\uparrow.a.cn = \mathit{next}(g.a.cn) \rightarrow g.a.cn.r = g.r) \wedge \\
&\quad\quad (g.r\uparrow.a.cn \neq \mathit{next}(g.a.cn) \rightarrow g.a.cn.r = g.r\uparrow a.cn)] \wedge \\
&\quad g \in m \rightarrow [g.r\uparrow.a.cn = g \ \wedge\ g \in \mathcal{C}(\mathit{invq})]\ ] \\
\mathit{PostListcopy}\,2' &= \forall g \in R^*(n)[g.a.cn.l = g.a.l\uparrow.cn \ \wedge \\
&\quad g \in \mathcal{C}(\mathit{invq}) \rightarrow \\
&\quad\quad (g.a.cn.r = g \ \wedge \\
&\quad\quad\quad \forall m > 1\ (g = \mathit{deq}^m(\mathit{invq}) \rightarrow \mathit{next}(g.a.cn) = [\mathit{deq}^{m-1}(\mathit{invq})].cn)) \wedge \\
&\quad g \notin \mathcal{C}(\mathit{invq}) \rightarrow (g.a.cn.r = g.r\uparrow.a.cn)\ ] \\
\mathit{PostListcopy}\,3' &= \forall g \in R^*(n)\ [g.a.cn.l = g.l\uparrow.a.cn \ \wedge \\
&\quad g.a.cn.r = g.r\uparrow.a.cn\ ] \\
\mathcal{C}(cq) &= \{g \mid \exists c \in \mathcal{C}(cq)\ g.a.cn = c\} \\
\mathit{LoopinvListmark}\,1 &= \mathit{LoopinvListmark}\,1'[\mathcal{C}(Qu(qn))\,/\,\mathcal{C}(q)] \\
\mathit{LoopinvListcopy}\,1 &= \mathit{LoopinvListcopy}\,1'[\mathcal{C}'(Qu(qn))\,/\,\mathcal{C}(q)] \wedge \\
&\quad \forall g \in m\ (g.r = g.a.cn) \\
\mathit{PostListcopy}\,1 &= \mathit{PostListcopy}\,1' \wedge \\
&\quad \forall g \in R^*(n)\ (g.r = g.a.cn)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{LoopinvListmark}\,2 \;=\; & \mathit{LoopinvListmark}\,2'[\mathcal{C}'(Qu(qn)) \,/\, \mathcal{C}(q)] \\
\mathit{LoopinvListcopy}\,2 \;=\; & \mathit{LoopinvListcopy}\,2'[\mathcal{C}'(Iq(qn)) \,/\, \mathcal{C}(invq)] \;\wedge \\
& \forall g \in R^*(n) \\
& \quad [g \notin m \rightarrow g.r = g.a.cn \;\; \wedge \\
& \quad\; g \in m \rightarrow \\
& \quad\quad [g.a.r\uparrow.a.cn = next(g.a.cn) \rightarrow g.r = g.a.r \;\wedge \\
& \quad\quad\; g.a.r\uparrow.a.cn \neq next(g.a.cn) \rightarrow g.r = g.a.r\uparrow.a.cn\,] \\
& \quad] \\
\mathit{PostListcopy}\,2 \;=\; & \\
& \mathit{PostListcopy}\,2'[\mathcal{C}'(Iq(qn)) \,/\, \mathcal{C}(invq)] \;\wedge \\
& \forall g \in \mathcal{C}'(Iq(qn)) \\
& \quad [g.a.r\uparrow.a.cn = next(g.a.cn) \rightarrow g.r = g.a.r \;\wedge \\
& \quad\; g.a.r\uparrow.a.cn \neq next(g.a.cn) \rightarrow g.r = g.a.r\uparrow.a.cn\,] \;\wedge \\
& \forall g \notin \mathcal{C}'(Iq(qn))\; g.r = g.a.r \\
\\
\mathit{PostListcopy}\,3 \;=\; & \mathit{PostListcopy}\,3'
\end{aligned}
$$

FIGURE 5.7: Assertions from figures 5.5 and 5.6.

6. Clark's List-Copying Algorithm

6.1. Introduction

The fastest list-copying algorithm known to date has been published by Douglas W. Clark [5]. 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.

6.2. 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. This algorithm will be derived from *lm2*.

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 invalidated.

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 it might be possible to reach one child via another. While 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 $b := b \cup \{f\}$ in *lm2* by **if** $R(f) - (m \cup \{s\}) \neq \varnothing$ **then** $b := b \cup \{f\}$. Node f will only be placed in b if there is another son besides s which is not yet marked. The second refinement consists of replacing $f := x \mid x \in b$; $b := b - \{f\}$ by a loop that repeats this action until a node with an unmarked sibling is found or b is exhausted. 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 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 to a node 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 nodes with two unmarked siblings upon first encounter (thus appearing of type FF) can have both children marked when it is fetched from set b. Then a node is noted to be of type BF or FB, depending on the direction of the first descent. Such a node is called *inscrutable*.

The next transformation depends on the availability of a function *type*, which returns the type of a node as given above. The exact implementation of this function will depend on the amount of information given in the original structure about the spanning tree. In the first pass of Clark's list-copying algorithm all true FF-nodes will be marked to distinguish them from inscrutable nodes. Hence the type-check of the second pass will differ from the first. At the moment correctness of type-checking will be assumed, with the exception of inscrutable nodes on first encounter.

```
f:=n; m:=∅; b:=∅;
{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
   begin
      m:=m∪{f};
      if not R(f)⊆m
      then
         begin
            s:=x|x∈R(f)−m;
            if R(f)−(m∪{s})≠∅ then b:=b∪{f};
            f:=s;
         end
      else
         begin
            loop
               f:=x|x∈b; b:=b−{f}
            until b=∅ ∨ ¬R(f)⊆m;
            if ¬R(f)⊆m
            then
               begin
                  s:=x|x∈R(f)−{m};
                  if R(f)−{m}=s then f:=s
               end
         end
   end;
m:=m∪{f}
{m=R*(n)}
```

FIGURE 6.1: Algorithm *lm3c*.

A second transformation is the replacement of the set b by a stack called *kst*. The procedures *push* and *pop* are standard and the notation $\mathcal{C}(st)$ for the content of stack *st* has been encountered before.

Lastly, a simplification is possible when a node of type FF is fetched from *kst*. Since it has two siblings and one at least has been marked before it need not be put on the stack again. The application of these three transformations will result in algorithm *lm4c* (figure 6.2).

Next defining *pop*(*nilst*) to be *nil* makes it possible to use transformation rule $L2$. Execution of the body of the loop while $kst=nilst \wedge R(f)\subseteq m\cup\{f\}$ results in f being *nil*, which doesn't happen earlier during execution, so the test $f\neq nil$ is a candidate for the new loop-test. Since the set m is not changed in the **if**-statement the postcondition is not affected. 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 $f=nil\rightarrow kst=nilst$ has to be added to the invariant, which is clearly valid. The old loop-test $\neg(kst=nilst \wedge R(f)\subseteq m\cup\{f\})$ and the new assertion imply $f\neq nil$ since $R(nil)=\varnothing$.

```
f:=n; m:=∅; kst:=nilst;
{n∈m∪{f} ∧ m∪{f}⊆R*(n) ∧
 R(m−(Q(kst)∪{f}))⊆m∪{f} ∧ Q(kst)⊆m−{f}}
while not (kst=nilst and R(f)⊆m∪{f})
do
   begin
      m:=m∪{f};
      if type(f)∈{AF,BF,FA,FB,FF}
      then
         begin
            s:=x|x∈R(f)−m;
            if type(f)=FF then push(kst,f);
            f:=s;
         end
      else { type(f)∈{AA,AB,BA,BB} }
         begin
            loop
               f:=pop(kst)
            until kst=nilst ∨ type(f)=FF;
            if type(f)=FF
            then f:=x|x∈R(f)−{m};
         end
   end;
m:=m∪{f}
{m=R*(n)}
```

FIGURE 6.2: Algorithm *lm4c.*

This new assertion can be used again to replace

```
loop
   f:=pop(kst)
until kst=nilst ∨ type(f)=FF;
```

with

```
loop
   f:=pop(kst);
   b:=f≠nil;
   if b then b:=type(f)≠FF
until not b;
```

The latter loop pops one 'element' more off *kst*. The whole resulting **else**-clause will be replaced by a function named *popS*.

Furthermore it is desired to be more explicit about which son is selected while descending into the list structure. To achieve this all node-types have to be divided into four classes: no forward pointer, one left forward pointer, one right forward pointer and two forward pointers. Since lateron every type needs a separate treatment it is convenient to split up these classes in their separate types. A **case**-statement is used instead of the equivalent repeated **if**-statements.

In the case of two forward pointers (the FF-type) the right son will be selected. Then for all nodes on *kst* the right son will be marked if it is not an atom. Thus we can be sure that the left son should be selected when a node is popped off *kst*. The resulting algorithm *lm5c* can be found in figure 6.3. Function *popS* is given in figure 6.4.

```
f:=n; m:=∅; kst:=nilst;
{n∈m∪{f} ∧ m∪{f}⊆R*(n) ∧
 R(m−(𝒬(kst)∪{f}))⊆m∪{f} ∧ 𝒬(kst)⊆m−{f} ∧
 ∀x∈𝒬(kst) x.r↑∈m∪{f}}
while f≠nil
do
   begin
      m:=m∪{f};
      case type(f) of
         AA: f:=popS(kst);
         AB: f:=popS(kst);
         AF: f:=f.r;
         BA: f:=popS(kst);
         BB: f:=popS(kst);
         BF: f:=f.r;
         FA: f:=f.l;
         FB: f:=f.l;
         FF: begin push(kst,f); f:=f.r end
      end
   end
{m=R*(n)}
```

FIGURE 6.3: Algorithm *lm5c.*

```
function popS (var s: stack): pointer;
   var f: pointer; b: boolean;
begin
   loop
      f:=pop(kst);
      b:=f≠nil;
      if b then b:=type(f)≠FF
   until not b;
   if type(f)=FF then f:=f.l;
   popS:=f
end
```

FIGURE 6.4: Algorithm *popS.*

6.3. Some auxiliary functions

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

The Clark algorithm demands the placement of the copy structure into a contiguous part of memory. Hence Clark can use the test *iscopy* encountered with Fisher. 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 machine-level action will be represented abstractly by the function *next*. Actually the size of a node will be added to the address of a copy node c to give the address of $next(c)$.

Two stacks are used by Clark's algorithm. Since the algorithm aims at using bounded workspace these stacks must be implemented using the actual nodes in either the original or the copy structure. The choice made is to link nodes by their right pointer fields. Two rather trivial changes are necessary to adapt the usual stack operators *push* and *pop*. The modified versions *pushC* and *popC* are given in figure 6.5.

```
procedure pushC (var st: stack; var n: pointer);
begin
  n.r:=st; st:=n
end

function popC (var st: stack): pointer;
begin
  if st=nilst
  then popC:=nil
  else begin popC:=st; st:=st.r end
end
```

FIGURE 6.5: Algorithms *pushC* and *popC*.

6.4. 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. Hence they will be treated as such. When an inscrutable node is popped off *kst* it can be recognized as beying of type BF. Then it will be treated as a regular node of type BF. To avoid confusion and improve efficiency every true FF-node will be marked by means of the pointer field left over by the stack.

The other type giving problems are the BB-nodes. Normally five fields are necessary to store all essential pointers (two original and two new back pointers and a pointer from original 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 nodes in a special stack called *bst*. This stack is emptied between the two passes through the entire structure and all pointer fields in both original and copy nodes are then 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 old inscrutable nodes are distinguishable as BF-type nodes the only nodes on stack *kst* will be of type FF. The mark will not be needed after a node is recognized as such and put on *kst*, hence this field can be used to implement *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'* in figure 6.6. A new pointer *c* is introduced to point to the copy of the node *f*, the node 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, its siblings and the matching copy node are stored in the auxiliary field of every node of the original structure.

The case-clause in the first pass is essentially the same as in algorithm *lm5c*. Preceding the statements of the latter algorithms the fields of the newly assigned copy nodes are filled. In the case of a BB-type node a stack is filled. 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 is not used, hence the copy nodes of these 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 to 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* needs a slight modification, since the treatment of all inscrutable nodes must be changed to the treatment of type BF. 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 represented by the function *next* from the previous section.

```
f:=n; m:=∅; kst:=nilst; c:=next(c); copy:=c; bst:=nilst;
{Loopinvlistmark' ∧ Loopinvlistcopy 1'}
while f≠nil   (*pass 1*)
do
   begin
      m:=m∪{f}; oldl:=l.f; oldr:=r.f; f.a:=(f,c,oldl,oldr);
      case type(f) of
         AA:  begin c.l:=oldl; c.r:=oldr;                          f:=pop'(kst)   end;
         AB:  begin c.l:=oldl; c.r:=oldr.a.cn;                     f:=pop'(kst)   end;
         AF:  begin c.l:=oldl; c.r:=next(c);                       f:=f.r         end;
         BA:  begin c.l:=oldl.a.cn; c.r:=oldl;                     f:=pop'(kst)   end;
         BB:  begin c.l:=oldl; c.r:=oldr; push(bst,f);             f:=pop'(kst)   end;
         BF:  begin c.l:=oldl.a.cn; c.r:=oldl;                     f:=f.r         end;
         FA:  begin c.l:=oldl; c.r:=oldr;                          f:=f.l         end;
         FB:  begin c.l:=oldl; c.r:=oldr.a.cn;                     f:=f.l         end;
         FF:  begin c.l:=oldl;                  push(kst,f);       f:=f.r         end
      end;
      c:=next(c);
   end;
{Postlistcopy 1'}
while bst≠nilst  (*B-stack processing*)
do
   begin
      f:=pop(bst); c:=f.a.cn;
      oldl:=c.l; oldr:=c.r;
      c.l:=oldl.a.cn; c.r:=oldr.a.cn;
      f.l:=oldl; f.r:=oldr
   end;
f:=n; m':=∅; kst:=nilst; c:=copy;
{Loopinvlistmark'[m'/m] ∧ Loopinvlistcopy 2'}
while f≠nil   (*pass 2*)
do
   begin
      m:=m∪{f};
      case type(f) of
         AA:  begin                  f:=pop''(kst)   end;
         AB:  begin                  f:=pop''(kst)   end;
         AF:  begin                  f:=f.r          end;
         BA:  begin c.r:=f.r;        f:=pop''(kst)   end;
         BB:  begin                  f:=pop''(kst)   end;
         BF:  begin c.r:=next(c);    f:=f.r          end;
         FA:  begin c.r:=next(c);    f:=f.l          end;
         FB:  begin c.r:=next(c);    f:=f.l          end;
         FF:  begin     push(kst,f); f:=f.r          end
      end;
      c:=next(c);
   end
{Postlistcopy 2'}
```

FIGURE 6.6: Algorithm *clc'*.
The assertions are described in figure 6.14.

```
function pop' (var s: stack): pointer;
  var f, c: pointer; b: boolean;
begin
  while
    f:=pop(kst);
    b:=f≠nil;
    if b then b:=type(f)≠FF;
    b
  do
    begin
      c:=f.a.cn; c.r:=c.l;
      c.l:=c.l↑.a.cn
    end;
  if type(f)=FF then f:=f.l;
  pop':=f
end
```

FIGURE 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 and the original pointers can be restored.

In pass two through the entire structure all remaining pointers in the copy structures 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 function *pop''* is presented in figure 6.8. Note that there are no inscrutable nodes on *kst*.

```
function pop'' (var s: stack): pointer;
  var f, a: pointer;
begin
  f:=pop(kst);
  if f≠nil
  then
    begin
      a:=f.a.cn; f:=f.l;
      a.l:=next(c); (*the copy of f*)
      a.r:=next(a)
    end;
  pop'':=f
end
```

FIGURE 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 left child of the node popped off *kst*, and this node is *next*(*c*). The copy of the right ponter 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 one 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. *kst* will be realized by linking the copy nodes by their right pointer fields. Functions

pop' and *pop''* have to be modified to accommodate for these changes. Figures 6.9 and 6.10 contain the new procedures *pop* 1 and *pop* 2.

```
function pop1 (var s: stack): pointer;
  var f, c: pointer; b: boolean;
begin
  while
    c:=popC(kst);
    b:=c≠nil;
    if b
    then b:=iscopy(c.l↑.l↑)
    else f:=nil;
    b
  do
    begin
      c.r:=c.l; c.l:=c.l↑.l
    end;
  if c≠nil
  then
    begin
      f:=c.l; f.r:=markFF
    end;
  pop1:=f
end
```

FIGURE 6.9: Algorithm *pop* 1.

```
function pop2 (var s: stack): pointer;
  var f, a: pointer;
begin
  a:=popC(kst);
  if a≠nil
  then
    begin
      f:=a.l;
      a.l:=next(c);
      a.r:=next(a)
    end
  else f:=nil;
  pop2:=f
end
```

FIGURE 6.10: Algorithm *pop* 2.

During *pop* 1 every node popped off *kst* was either a scrutable or an inscrutable 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 on *type*(*f*)≠FF can be replaced by the test *iscopy*(*c.l*↑.*l*). Then a redundant assignation to *c* is deleted and an initial assignation to *f* is introduced in the case of an empty stack. Similar transformations allow one to derive *pop* 2 from *pop''*.

```
f:=n; m:=∅; kst:=nilst; c:=next(c); copy:=c; bst:=nilst;
{Loopinvlistmark ∧ Loopinvlistcopy 1}
while f≠nil   (*pass 1*)
do
  begin
    m:=m∪{f}; oldl:=l.f; oldr:=r.f; f.a:=(f,c,oldl,oldr);
    case type1(f) of
      AA: begin c.l:=oldl; c.r:=oldr;                          f:=pop1(kst)  end;
      AB: begin c.l:=oldl; c.r:=oldr.l;                        f:=pop1(kst)  end;
      AF: begin c.l:=oldl; c.r:=next(c);                       f:=oldr       end;
      BA: begin c.l:=oldl.l; c.r:=oldl;                        f:=pop1(kst)  end;
      BB: begin c.l:=oldl; c.r:=oldr; pushC(bst,f);            f:=pop1(kst)  end;
      BF: begin c.l:=oldl.l; c.r:=oldl;                        f:=oldr       end;
      FA: begin c.l:=oldl; c.r:=oldr;                          f:=oldl       end;
      FB: begin c.l:=oldl; c.r:=oldr.l;                        f:=oldl       end;
      FF: begin c.l:=oldl;                pushC(kst,c);        f:=oldr       end
    end;
    c:=next(c);
  end;
{Postlistcopy 1}
while bst≠nilst  (*B-stack processing*)
do
  begin
    f:=popC(bst); c:=f.l;
    oldl:=c.l; oldr:=c.r;
    c.l:=oldl.l; c.r:=oldr.l;
    f.l:=oldl; f.r:=oldr
  end;
f:=n; m':=∅; kst:=nilst; c:=copy;
{Loopinvlistmark[m'/m] ∧ Loopinvlistcopy 2}
while f≠nil   (*pass 2*)
do
  begin
    m:=m∪{f};
    case type2(f) of
      AA: begin f.l:=c.l;                                      f:=pop2(kst)  end;
      AB: begin f.l:=c.l;                                      f:=pop2(kst)  end;
      AF: begin f.l:=c.l;                                      f:=f.r        end;
      BA: begin f.l:=c.r; c.r:=f.r;                            f:=pop2(kst)  end;
      BB: begin                                                f:=pop2(kst)  end;
      BF: begin f.l:=c.r; c.r:=next(c);                        f:=f.r        end;
      FA: begin f.l:=c.l; c.l:=next(c);                        f:=f.l        end;
      FB: begin f.l:=c.l; c.l:=next(c);                        f:=f.l        end;
      FF: begin f.l:=c.l;               pushC(kst,f);          f:=f.r        end
    end;
    c:=next(c);
  end
{Postlistcopy 2}
```

FIGURE 6.11: algorithm *clc*, Clark's list-copying algorithm.
The assertions are described in figure 6.14.

The final list-copying algorithm according to Clark is shown in figure 6.11. The algorithm proper excludes the statements dealing with auxiliary variables (set in a different font). Transformations from *clc'* to *clc* fall into three categories. The introduction of the special stack-operators *pop* 1, *pop* 2, *popC* (included in the former two) and *pushC* was already explained. The variables *oldl* and *oldr* are used in the first pass, since the old values are sometimes overwritten. And the route to the copy node via the auxiliary structure is replaced by a direct pointer stored in the left pointer field of the original node.

This last transformation guides one to store the BB-type nodes in a stack. The first BB-type node encountered cannot point to any other BB-type node, since none of them were encountered as yet. Similarly the second BB-type node encountered can only point to the first one, not to others. Hence by induction the removal of the pointer to the copy in the last BB-type node encountered does not lose information about pointers in the copy nodes of the others. Next the last node but one can be removed, et cetera.

6.5. Nodetypes

One part of the transformation series from list-marking algorithm *lm4c* to the final algorithm was treated rather abstractly: how to recognize the type of a node. The main use of the type-check is to decide which branch to take in the two case-clauses. The functions necessary differ of course since the original structure is altered at the instant of invocation in the second pass. The two functions *type* 1 and *type* 2 are given in figures 6.12 and 6.13.

Function *type* 1 is obviously correct. The correctness of *type* 2 follows straightforward from the results of the first pass and the processing of stack *bst*.

```
function type1 (n: node): nodetype;
  var l, r: pointer;
begin
  l:=n.l; r:=n.r;
  if atom(l)
  then
    if atom(r) then type1:=AA
    else
      if iscopy(r.l↑) then type1:=AB
      else type1:=AF
  else
    if iscopy(l.l↑)
    then
      if atom(r) then type1:=BA
      else
        if iscopy(r.l↑) then type1:=BB
        else type1:=BF
    else
      if atom(r) then type1:=FA
      else
        if iscopy(r.l↑) then type1:=FB
        else type1:=FF
end
```

FIGURE 6.12: Algorithm *type1*.

```
function type2 (n: node): nodetype;
  var c: pointer;
begin
  c:=n.l;
  if not iscopy(c↑)
  then type2:=BB
  else
    if atom(c.l)
    then
      if atom(n.r) then type2:=AA
      else
        if c.r↑=next(c) then type2:=AF
        else type2:=AB
    else
      if atom(n.r)
      then
        if atom(c.r) then type2:=FA
        else type2:=BA
      else
        if iscopy(f.l↑) then type2:=BF
        else type2:=FB
end
```

FIGURE 6.13: Algorithm *type2*.

$$
\begin{aligned}
\mathit{Loopinvlistmark}' = {} & n \in m \cup \{f\} \ \wedge \ m \cup \{f\} \subseteq R^*(n) \ \wedge \ R(m - (\mathcal{C}(kst) \cup \{f\})) \subseteq m \cup \{f\} \\
& \wedge \ \mathcal{C}(kst) \subseteq m - \{f\} \ \wedge \ \forall g \in \mathcal{C}(kst)\ g.r \in m \cup \{f\}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Loopinvlistcopy}\,1' = {} & \forall g \in m\ \exists c\ [c = g.a.cn \ \wedge \\
& \mathit{type}(g) = \mathrm{AA} \rightarrow (c.l = g.a.l \ \wedge \ c.r = g.a.r) \ \wedge \\
& \mathit{type}(g) = \mathrm{AB} \rightarrow (c.l = g.a.l \ \wedge \ c.r = g.a.r\uparrow.a.cn) \ \wedge \\
& \mathit{type}(g) = \mathrm{AF} \rightarrow (c.l = g.a.l \ \wedge \ c.r = \mathit{next}(c)) \ \wedge \\
& \mathit{type}(g) = \mathrm{BA} \rightarrow (c.l = g.a.l\uparrow.a.cn \ \wedge \ c.r = g.a.l) \ \wedge \\
& \mathit{type}(g) = \mathrm{BB} \rightarrow (c.l = g.a.l \ \wedge \ c.r = g.a.r \ \wedge \ g \in \mathcal{C}(bst)) \ \wedge \\
& \mathit{type}(g) = \mathrm{BF} \rightarrow (g \notin \mathcal{C}(kst) \rightarrow (c.l = g.a.l\uparrow.a.cn \ \wedge \ c.r = g.a.l) \ \wedge \\
& \qquad\qquad g \in \mathcal{C}(kst) \rightarrow c.l = g.a.l) \ \wedge \\
& \mathit{type}(g) = \mathrm{FA} \rightarrow (c.l = g.a.l \ \wedge \ c.r = g.a.r) \ \wedge \\
& \mathit{type}(g) = \mathrm{FB} \rightarrow (c.l = g.a.l \ \wedge \ c.r = g.a.r\uparrow.a.cn) \ \wedge \\
& \mathit{type}(g) = \mathrm{FF} \rightarrow (c.l = g.a.l)\ ]
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Postlistcopy}\,1' = {} & \forall g \in m\ \exists c\ [c = g.a.cn \ \wedge \\
& \mathit{type}(g) = \mathrm{AA} \rightarrow \ \cdots \\
& \cdots \\
& \mathit{type}(g) = \mathrm{BB} \rightarrow (g \in \mathcal{C}(bst) \rightarrow (c.l = g.a.l \ \wedge \ c.r = g.a.r) \ \wedge \\
& \qquad\qquad g \notin \mathcal{C}(bst) \rightarrow \\
& \qquad\qquad\quad (c.l = g.a.l\uparrow.a.cn \ \wedge \ c.r = g.a.r\uparrow.a.cn)) \ \wedge \\
& \cdots\ ]
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Loopinvlistcopy}\,2' \;=\; & \forall g \in R^*(n)\ \exists c\ [c = g.a.cn \ \wedge \\
& \mathit{type}(g) = \mathrm{AA} \rightarrow \cdots \\
& \cdots \\
& \mathit{type}(g) = \mathrm{BA} \rightarrow (c.l = g.a.l\uparrow.a.cn \ \wedge \\
& \qquad\qquad g \notin m' \rightarrow c.r = g.a.l \ \wedge \\
& \qquad\qquad g \in m' \rightarrow c.r = g.a.r) \ \wedge \\
& \cdots \\
& \mathit{type}(g) = \mathrm{BF} \rightarrow (c.l = g.a.l\uparrow.a.cn \ \wedge \\
& \qquad\qquad g \notin m' \rightarrow c.r = g.a.l \ \wedge \\
& \qquad\qquad g \in m' \rightarrow c.r = \mathit{next}(c)) \ \wedge \\
& \mathit{type}(g) = \mathrm{FA} \rightarrow (c.r = g.a.r \ \wedge \\
& \qquad\qquad g \notin m' \rightarrow c.l = g.a.l \ \wedge \\
& \qquad\qquad g \in m' \rightarrow c.l = \mathit{next}(c)) \ \wedge \\
& \mathit{type}(g) = \mathrm{FB} \rightarrow (c.r = g.a.r\uparrow.a.cn \ \wedge \\
& \qquad\qquad g \notin m' \rightarrow c.l = g.a.l \ \wedge \\
& \qquad\qquad g \in m' \rightarrow c.l = \mathit{next}(c)) \ \wedge \\
& \mathit{type}(g) = \mathrm{FF} \rightarrow (g \notin m' - \mathcal{C}(\mathit{kst}) \rightarrow c.l = g.a.l \ \wedge \\
& \qquad\qquad g \in m' - \mathcal{C}(\mathit{kst}) \rightarrow (c.l = c.a.l\uparrow.a.cn \ \wedge \\
& \qquad\qquad\qquad\qquad c.r = c.a.r\uparrow.a.cn))\ ]
\end{aligned}
$$

$$
\mathit{Postlistcopy}\,2' \;=\; \forall g \in R^*(n)\ \exists c\ [c = g.a.cn \ \wedge\ c.l = \mathit{copy}(g.l) \ \wedge\ c.r = \mathit{copy}(g.r)]
$$

$$
\mathit{copy}(p) \;=\; \begin{cases} p & \mathit{if\ atom}(p) \\ p.a.cn & \mathit{otherwise} \end{cases}
$$

$$
\mathit{Loopinvlistmark} \;=\; \mathit{Loopinvlistmark'}
$$

$$
\begin{aligned}
\mathit{Loopinvlistcopy}\,1 \;=\; & \forall g \in m\ \exists c\ [c = g.a.cn \ \wedge\ g.l = c \ \wedge \\
& \mathit{type}(g) = \mathrm{AA} \rightarrow \cdots \\
& \cdots \\
& \mathit{type}(g) = \mathrm{FF} \rightarrow (c.l = g.a.l \ \wedge\ c \notin \mathcal{C}(\mathit{kst}) \rightarrow g.r = \mathit{markFF})\ ]
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Postlistcopy}\,1 \;=\; & \forall g \in R^*(n)\ \ [(\mathit{type}(g) \neq \mathrm{BB} \ \vee\ g \in \mathcal{C}(\mathit{bst})) \rightarrow g.l = g.a.cn] \ \wedge \\
& \mathit{Postlistcopy}\,1'
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Loopinvlistcopy}\,2 \;=\; & \forall g \in R^*(n) - m'\ \ g.l = g.a.cn \ \wedge \\
& \mathit{Loopinvlistcopy}\,2'
\end{aligned}
$$

$$
\mathit{Postlistcopy}\,2 \;=\; \mathit{Postlistcopy}\,2'
$$

FIGURE 6.14: Assertions from figures 6.6 and 6.11.

7. Related work

The introduction of Hoare Logic made it relatively easy to prove the correctness of small conventional programs. Correctness proofs of larger programs can be given in this formalism, however, they tend to expand to impractical sizes. Thus some more structure is needed in the proof of a large program in Hoare 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 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 [10].

Experience is also available on the technique of control transfer between program variables and auxiliary variables. Blikle [4] gives a very thorough example of this technique on a rather simple program: calculation of the integer square root.

Scherlis [19] 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. On the other hand termination can easily be proved.

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 termination of the resulting program is the first aim of the transformation. Hence the rationale for the transformation is the first step towards a proof. A more formal treatment of termination criteria is possible. Apt and Delporte [1] worked on this topic using a version of *temporal logic* as proof system.

An interesting development is also the introduction of a new primitive programming tool for graph algorithms by Suzuki [21]. While his pointer rotation technique has some nice advantages it does not 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 ancestral paper by Lee, De Roever and Gerhart [11] 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 list-copying algorithms through transformations starting with a list-marking algorithm, is retained.

Acknowledgement

We thank J. Heering, P. Klint, M. Sintzoff and the referees for their valuable comments on an earlier version of this paper, and J. Heering and H. Noot for their assistance with preparation of the manuscript.

REFERENCES

1. K.R. APT and C. DELPORTE (1983). *An axiomatization of the intermittent assertion method (extended abstract),* Report 82-70, Laboratoire Informatique Thèorique et Programmation, Paris.
2. R.J.R. BACK (1980). *Correctness preserving program refinements: proof theory and applications,* MC tracts 131, Amsterdam.
3. R.S. BIRD (October 1984). The promotion and accumulation strategies in transformational programming, *ACM Transactions on Programming Languages and Systems*, 6.4, 487-504.
4. A. BLIKLE (1978). *Specified programming,* ICS PAS Reports 333, Warszawa.
5. D.W. CLARK (1978). A fast algorithm for copying list structures, *Comm. ACM.*, 21.5, 351-357.
6. J. DARLINGTON (1980). A synthesis of several sorting algorithms, *Acta Informatica*, 11, 1-30.
7. D.A. FISHER (1975). Copying cyclic list structures in linear time using bounded workspace, *Comm. ACM.*, 18.5, 251-252.
8. S.L. GERHART (1975). Correctness preserving program transformations, in *Proceedings Second Principles of Programming Languages Symposium*, 54-66, Palo Alto.
9. S.L. GERHART (1976). Proof theory of partial correctness verification systems, *SIAM J. Comput.*, 5.3, 355-377.
10. C.B. JONES (1980). *Software development: a rigorous approach,* Series in Computer Science, Prentice/Hall, London.
11. S. LEE, W.P. DE ROEVER, and S.L. GERHART (1979). The evolution of list copying algorithms and the need for structured program verification, in *Conf. Rec. sixth ann. ACM Symp. on Principles Of Progr. Languages*, 53-67, San Antonio.
12. G. LINDSTROM (1974). Copying list structures using bounded workspace, *Comm. ACM*, 17.4, 198-202.
13. C. LIVERCY (1978). *Théorie des Programmes. Schémas, preuves, sémantique*, Paris.
14. Z. MANNA and R. WALDINGER (1981). Deductive synthesis of the unification algorithm, *Science of Computer Programming*, 1, 5-48, North-Holland.
15. P.G. NEUMANN (1978). Computer system security evaluation, in *1978 National Computer Conference*, Anaheim, CA.
16. H. PARTSCH (April 1984). Structuring transformational developments: a case study based on Earley's recognizer, *Science of Computer Programming*, 4.1, 17-44, North-Holland.
17. J.H. REIF and W.L. SCHERLIS (1982). *Deriving efficient graph algorithms,* Report CMU-CS-82-155.
18. J.M. ROBSON (1977). A bounded storage algorithm for copying cyclic list structures, *Comm. ACM*, 20.6, 431-433.
19. W.L. SCHERLIS (1981). Program improvement by internal specialization, in *Conf. Rec. eight ann. ACM Symp. on Principles Of Progr. Languages*, 41-49.
20. H. SCHORR and W.M. WAITE (1967). An efficient machine-independent procedure for garbage collection in various list structures, *Comm. ACM*, 10, 501-506.
21. N. SUZUKI (1982). Analysis of pointer 'rotation', *Comm. ACM*, 25.5, 330-335.