THE EVOLUTION OF LIST-COPYING ALGORITHMS
and The Need for Structured Program Verification

STANLEY LEE
WILLEM P. DE ROEVER
SUSAN L. GERHART

RUU-CS-78-7

November 1978

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6
Postbus 80.012
3508 TA Utrecht
Telefoon 030–531454

The Netherlands

THE EVOLUTION OF LIST-COPYING ALGORITHMS
and The Need for Structured Program Verification

Stanley Lee
Computer Science Division
University of California
Berkeley CA 94720

Willem P. de Roever
Department of Computer Science
University of Utrecht
3508 TA Utrecht, the Netherlands

Susan L. Gerhart
USC/Information Sciences Inst.
4676 Admiralty Way
Marina del Rey CA 90291

Technical Report RUU-CS-78-7

November 1978

Department of Computer Science
University of Utrecht
3508 TA Utrecht, the Netherlands

THE EVOLUTION OF LIST-COPYING ALGORITHMS
and The Need for Structured Program Verification

Stanley Lee*
Computer Science Division
University of California
Berkeley CA 94720

Willem P. deRoever* **
Computer Science Division
University of California
Berkeley CA 94720

Susan L. Gerhart***
USC/Information Sciences Inst.
4676 Admiralty Way
Marina del Rey CA 90291

1. INTRODUCTION

How can one organize the understanding of complex algorithms? People have been thinking about this issue at least since Euclid first tried to explain his innovative greatest common divisor algorithm to his colleagues, but for current research into verifying state-of-the-art programs, some precise answers to the question are needed. Over the past decade the various verification methods which have been introduced (inductive assertions, structural induction, least-fixedpoint semantics, etc.) have established many *basic principles* of program verification (which we define as: establishing that a program text satisfies a given pair of input-output specifications). However, it is no coincidence that most published examples of the application of these methods have dealt with "toy programs" of carefully considered simplicity.

Experience indicates that these "first generation" principles, with which one can easily verify a three-line greatest common divisor algorithm, do not directly enable one to verify a 10,000 line operating system (or even a 50 line list-processing algorithm) in complete detail. To verify complex programs, additional techniques of organization, analysis and manipulation are required. (That a similar situation exists in the *writing* of large, correct programs has long been recognized -- structured programming being one solution.)

This paper examines the usefulness of correctness-preserving program transformations (see [6]) in structuring fairly complex correctness proofs. Using our approach one starts with a simple, high-level (or "abstract") algorithm which can be easily verified, then successively refines it by implementing the abstractions of the initial algorithm to obtain various final, detailed algorithms. In Section 2 we introduce the technique by deriving the Deutsch-Schorr-Waite list-marking algorithm [14]. Our main example is the more complex problem of verifying bounded-workspace list-copying algorithms: Section 3 defines the issues, Section 4 presents the key intermediate algorithm in detail and Section 5 considers three of the most complex (published) implementations of list-copying, one of which is discussed in detail. In Section 6 we make some general remarks on program verification and the relevance of our results to the (larger) field of program correctness; Section 7 mentions some related work.

*Partially supported by NSF grant MCS 78-00673

**Present affiliation: Dept. of Computer Science, Univ. of Utrecht, Budapestlaan 8, Postbus 80-012, 3508TA Utrecht, The Netherlands

***Research conducted in part under Defense Advanced Research Projects Agency contract DAHC-15-72-C0308; also partially supported by NSF grant MCS 75-08146

2. FIRST EXAMPLE: LIST MARKING

2.1 Problem specification and the initial algorithm

We wish to define list marking in general terms, applicable to any particular implementation. As list marking is a special case of computing the reflexive-transitive closure of a relation, we let

(1) Mem denote a non-empty set,

(2) R denote a binary relation between elements of Mem, and

(3) Z denote some element of Mem.

Then (2) and (3) define our input assertion as $Z \in Mem \wedge R \subseteq Mem \times Mem$, and the goal is to construct the set $R^*(Z)$, defined as the smallest set, m, satisfying $Z \in m \wedge R(m) \subseteq m$. Interpreting

Mem as the (finite) set of all memory cells,

R as: aRb $\Leftrightarrow$ b is directly reachable from a (a "points to" b), and

Z as the root cell of some list structure,

we conclude that

Input-M: $Z \in Mem \wedge R \subseteq Mem \times Mem$

Output-M: $m = R^*(Z) \wedge Z, R, Mem$ unchanged

are appropriate implementation-independent ("abstract") specifications of the task of constructing the set m of all cells reachable from cell Z. (As we use identifiers beginning with capital letters exclusively for constants, the second half of Output-M will be left implicit.)

Our initial marking algorithm is MA-0 (see top of next page). In MA-0 (and throughout this paper) we use R(p) to denote {q: pRq}, i.e. those nodes directly reachable from p. Note that the while

SIXTH ACM SYMPOSIUM ON
PRINCIPLES OF PROGRAMMING LANGUAGES

January 29-31, 1979
St. Anthony Hotel, San Antonio, Texas

MA-0:

```
Assert Input-M: Z ∈ Mem ∧ R ⊆ Mem × Mem
  m := {Z} ;
  loop asserting Invar-M0: Z ∈ m ∧ m ⊆ R*(Z)
     while  m ⊂ R*(Z)  do
        Select p in m satisfying not R(p) ⊆ m ;
        m := m ∪ R(p)
  endloop
Assert Output-M: m = R*(Z)
```

statement, delimited by loop ... endloop , includes its invariant assertion. The semantics of the Select statement are defined as follows:

$$\frac{P \Rightarrow \exists y \in S: Q(y)}{\{P\}\ \underline{\text{Select}}\ z\ \underline{\text{in}}\ S\ \underline{\text{satisfying}}\ Q(z)\ \{P \wedge z \in S \wedge Q(z)\}}$$

for predicates P and Q not containing z as a free variable. Thus the Select statement is *non-deterministic*, in that any element of set S satisfying Q may be assigned to z . When a transformation replaces a Select statement with a deterministic program segment, any implementation may be chosen which meets the above semantic definition. This, along with differing implementations of abstract data structures, will enable us to generate different final algorithms from a common ancestor.

A formal proof of (partial) correctness for MA-0 with respect to assertions Input-M, Output-M is obtained by proving that 1) Invar-M0 is invariant for the while loop, and

2) $[\text{Invar-M0} \wedge \sim(m \subset R^*(Z))] \Rightarrow m = R^*(Z)$.

Both proofs are straightforward, requiring the use of various properties of the domain in question, here finite sets. (2) trivially follows from

$$[a \subseteq b \wedge \sim(a \subset b)] \Rightarrow a = b ,$$

(1) requires several properties of sets, e.g.

$$a \in R^*(b) \Rightarrow R(a) \subseteq R^*(b) .$$

Termination of MA-0 is proven with the variant function $|R^*(Z) - m|$ whose value decreases at each iteration.

2.2 Transformations yielding the archetype

We apply our first correctness-preserving program transformation to MA-0 in order to remove the reference to $R^*(Z)$ from the loop exit test. By using the transformation schema TS1 (Figure 1) and the domain property:

$$[a \in b \wedge b \subseteq R^*(a)] \Rightarrow (R(b) \subseteq b) \equiv (b = R^*(a))$$

we can change "while $m \subset R^*(Z)$ do ..." in MA-0 to "while not $R(m) \subseteq m$ do ..." , as the above set property ensures that the premise of TS1 is satisfied.

Our next transformation introduces a new variable to increase the efficiency of the exit test. If u (for "unsure") denotes a subset of m satisfying $R(m - u) \subseteq m$, then only cells in u can (possibly) point to new cells not already in m, allowing us to ignore cells in (m - u) when evaluating not $R(m) \subseteq m$. Employing transformation schemata TS1, TS2 and TS3 (see Figure 1) yields a new algorithm, MA-1:

MA-1:

```
Assert Input-M
  m := {Z} ;  u := {Z} ;
  loop asserting Invar-M1: Invar-M0
                  ∧ u ⊆ m ∧ R(m - u) ⊆ m
     while not R(u) ⊆ m do
        Select p in u satisfying not R(p) ⊆ m ;
        m := m ∪ R(p) ;   u := "new u"
  endloop
Assert Output-M
```

where "new u" satisfies the premise of TS2 (i.e., maintains the invariance of Invar-M1).

We refer to MA-1 as an archetypal algorithm for list marking -- that is, different marking algorithms can be obtained from MA-1 (via transformations) depending on how set u is implemented. In the remainder of Section 2 we shall derive the Deutsch-Schorr-Waite (DSW) algorithm from MA-1; by derivations omitted here, one can also obtain Algorithms A, B and C of [10] from archetype MA-1.

2.3 Intermediate marking algorithms

In deriving DSW we interpret u as the set of just those cells in m with any unexamined pointers -- hence "u := new u" in MA-1 becomes

$$u := (u - \{p\}) \cup R(p) ,$$

which satisfies the premise of TS2. As u will be implemented with a data structure not allowing random access (e.g. a stack rather than an array), our next algorithm, MA-2, removes the condition on the Select statement:

MA-2:

```
Assert Input-M
  m := {Z} ;   u := {Z} ;
  loop asserting Invar-M1
     while  u ≠ φ do
        Select p in u ;
        if not R(p) ⊆ m
              then m := m ∪ R(p);
                   u := (u - {p}) ∪ R(p)
              else u :=  u - {p}
        fi
  endloop
Assert Output-M
```

where φ denotes the empty set and Select p in u abbreviates Select ... satisfying true. Note that while the loop invariant is unchanged from algorithm MA-1, the variant function necessary to prove termination is now $|R^*(Z) - (m - u)|$. The transformation schema used is TS4. As the system of basic transformation schemata that we use has been formally presented elsewhere [6], [7], we discuss in what follows only the intermediate algorithms, since in this paper they, and not the transformations themselves, are our chief interest.

Name of transformation	Original program segment	Premise(s) for transformation	New program segment
TS1 -- exit test replacement	Assert P loop asrt. Invar while B do S endloop Assert Q	Invar ⇒ B ≡ B'	Assert P loop asrt. Invar while B' do S endloop Assert Q
TS2 -- addition of loop variable	Assert P loop asrt. Invar while B do S endloop Assert Q	{P} v:=e0 {P1} {Invar ∧ P1 ∧ B} S; v:=e1 {Invar ∧ P1} v not free in P,Q or Invar v doesn't appear in S	Assert P v := e0 ; loop asrt. Invar ∧ P1 while B do S ; v := e1 endloop Assert Q
TS3 -- Select stmt set restriction	Select x in a satisfying P(x)	b ⊆ a ∧ ∀ y: y ∈ (a-b) ⇒ ~P(y)	Select x in b satisfying P(x)
TS4 -- Select stmt condition elimination	Assert P loop asrt. Invar(a) while x∈a: B(x) do Select x in a satisfying B(x) ; S endloop Assert Q	[Invar(a) ∧ a ≠ ϕ ∧ y ∈ a ∧ ~B(y)] ⇒ Invar(a-{y})	Assert P loop asrt. Invar(a) while a ≠ ϕ do Select x in a ; if B(x) then S else a:= a-{x} fi endloop Assert Q

FIGURE 1 -- A Few Correctness-Preserving Program Transformation Schemata

Next we partition the set u into cell p currently under examination and 'all the rest', denoted by u1 -- substituting u1 ∪ {p} for u in MA-2 we obtain the new algorithm MA-3:

MA-3:

```
Assert Input-M
  m := {Z} ;  p := Z ;  u1 := ϕ ;
  loop asserting Invar-M3:
            p, Z ∈ m ∧ m ⊆ R*(Z)
            ∧ R(m - (u1 ∪ {p})) ⊆ m
            ∧ u1 ⊆ m - {p}
     while not (u1 = ϕ ∧ R(p) ⊆ m) do
       if not R(p) ⊆ m
          then Select q in R(p)
                       satisfying not q ∈ m ;
               m := m ∪ {q} ;
               u1 := u1 ∪ {p} ;
               p := q
          else $elect p in u1 ;
               u1 := u1 - {p}
       fi
  endloop
Assert Output-M
```

The set u1 can now be directly implemented as a stack, which we denote by t . We then separate the single while loop of MA-3 into two inner loops, each corresponding to one branch of the conditional statement, to get our next algorithm MA-4 (see top of next page). In the invariant Invar-M4, t* denotes the set of nodes contained in stack t .

Our next algorithm modifies MA-4 in two ways:
1) To avoid re-calculating R(p) for each node popped off the stack in loop 2, we make each stack element a pair: <node, set of nodes> , and in loop 1 push p together with its (possibly) unmarked descendants R(p) - {q} onto stack t .
2) To avoid unnecessary pop-push sequences we move the exit test for the main loop so that it follows loop 2, and unfold loop 1 once. (In the following algorithm the syntax

loop asserting A: S1; while B do S2 endloop

could equivalently be expressed

```
L1:  S1 ;
     if not B then goto L2 ;
     S2 ; goto L1
L2:  ...                          )
```

Applying the appropriate transformations yields algorithm MA-5 (Figure 2a).

2.4 Implementing Deutsch-Schorr-Waite

MA-5 is an archetype for stack-based marking algorithms (including DSW) which differ principally

MA-4:

```
Assert Input-M
  m := {Z} ;  p := Z ;  Create/Stack(t) ;
  loop asserting Invar-M4:
            p,Z ∈ m ∧ m ⊆ R*(Z)
            ∧ R((m - {p}) - t*) ⊆ m
            ∧ t* ⊆ m - {p}
    while not (empty(t) ∧ R(p) ⊆ m) do
      loop asserting Invar-M4        —loop 1
        while not R(p) ⊆ m do
          Select q in R(p)
                satisfying not q ∈ m ;
          m := m ∪ {q} ;
          push (t,p) ;
          p := q
      endloop ;
      loop asserting Invar-M4        —loop 2
        while R(p) ⊆ m and not empty(t) do
          p := pop(t)
      endloop
  endloop
Assert Output-M
```

in their implementation of stack t . The archetype has a "down phase" -- loop 1 -- which follows pointers to unmarked nodes, and a "backup phase" -- loop 2 -- which pops already-marked nodes off stack t in search of pointers to unmarked nodes. Obtaining DSW from MA-5 requires making four additions:

1) Specify the node structure by defining each node to contain four fields: mark, atom, a and b . For any node p, p.mark is the mark bit of the node; p.atom = false indicates p is non-atomic and p.a, p.b contain left- and right-link pointers, respectively (if p.atom = true then p.a, p.b are disregarded as p is an atom). Since memory cells corresponding to atoms are not marked in DSW, we implement R by interpreting R(p) as Ra(p) ∪ Rb(p),

Ra(p) = if p.a.atom then ϕ else {p.a}
Rb(p) = if p.b.atom then ϕ else {p.b} .

The atom and mark fields correspond to abstract functions Atom, Mark: Mem → {true, false} respectively.

2) Implement set m by interpreting p ∈ m as equivalent to p.mark = true.

3) Implement Select q in R(p) satisfying not q ∈ m (where not R(p) ⊆ m holds) by assigning the value

if not p.a.mark then p.a else p.b to q. Thus the left link of a node is followed in the traversal before its right link (if possible).

4) Implement stack t as a reversed-pointer linked list within the original list structure (see [12]).

Applying transformations to MA-5 to add the above four features produces the DSW marking algorithm MA-6 (Figure 2b). The following general observations about MA-6 also apply to the list-copying implementations of Section 5.

Two new types of assertions appear in the loop invariant Invar-DSW. SameStack(t,tp) and "m = {y: y.mark} " are equivalence assertions which define the correspondence between the abstract and implemented data structures. Note that m and t are, in MA-6, auxiliary or 'ghost' variables, as the equivalence assertions have enabled the loop tests to be expressed independently of the abstract variables. We can now remove the statements in italics from MA-6, and preface Invar-DSW with " ∃m, t: " ; the correctness of the resulting (non-italic) Deutsch-Schorr-Waite algorithm and invariant follows from the correctness of MA-6 by the Ghost Variable Theorem (see [8]).

"Mod(aptr,bptr,atombit,tp,p) = (Ra,Rb,Atom)" is a perturbation assertion which is necessary as a result of the in situ stack implementation. This assertion defines the changes made in the original list structure -- which pointers are reversed -- and makes it possible to guarantee at termination that all pointers (and atom flags) have been restored to their original values, in other words, "(aptr,bptr,atombit) = (Ra,Rb,Atom)". (See Appendix A1 for texts of assertions SameStack, Mod.)

3. LIST COPYING: SPECIFICATION AND INITIAL ALGORITHM

In a fashion similar to Sec. 2.1, we now construct a pair of input/output specifications for list copying by considering the general case of extending a relation to produce an isomorphic mapping between elements. With Mem, R as in Sec. 2.1, let

- S denote a subset of Mem,
- C denote a set disjoint from Mem, and equal in size to S, and
- D denote a pairing of elements from S and C;

then we wish to extend R to σ such that R ⊆ σ ∧ (σ - R) ⊆ C × C and σ(D(y)) = {D(x): x ∈ σ(y)} for every y in S. Then, interpreting

- S as R*(Z),
- C as a set of |R*(Z)| cells (disjoint from the original set Mem) used for the copy of R*(Z),
- D as a one-to-one mapping from cells in the original list structure to cells in the copy, and
- σ - R as the set of all pointers in the new list structure,

we obtain

Input-C: Input-M ∧ C ∩ Mem = φ ∧ D: R*(Z) → C ∧ D is a bijection

Output-C: R ⊆ σ ∧ (σ - R) ⊆ C × C ∧ ∀y ∈ R*(Z): σ(D(y)) = D(σ(y)) ∧ Z,R,Mem,C,D unchanged

as our general copying specifications. (In Output-C D(σ(y)) denotes {D(x): x ∈ σ(y)} and "D is a bijection" abbreviates "D(y) = D(x) ⇒ x = y ∧ range(D) = C"). We use "original list structure" and "old cell" in referring to R*(Z), and "copy list structure" and "new cell" to refer to (σ - R)*(D(Z)).

```
Assert Input-M: Z ∈ Mem ∧ R ⊆ Mem × Mem

   m := {Z} ;   p := Z ;
   Create/Stack(t) ;
   loop asserting Invar-M5:
                p,Z ∈ m ∧ m ⊆ R*(Z)
                  ∧ R((m-{p})-t*) ⊆ m
                  ∧ DefStack(t)
      loop asserting Invar-M5
         while not R(p) ⊆ m do

            Select q in R(p)
                  satisfying not q ∈ m ;

            m := m ∪ {q} ;
            push(t, <p,R(p) - {q} >) ;

            p := q
      endloop
      loop asserting Invar-M5^{m-{p}}_{m}
         while (not empty(t)) cand
                              t2 ⊆ m do
            pop(t)

      endloop
      while not empty(t) do
         Select q in t2
               satisfying not q ∈ m ;
         m := m ∪ {q} ;
         Replace Top-of t with  <t1, t2-{q} > ;

         p := q
   endloop
Assert Output-M: m = R*(Z)
```

```
Assert Input-DSW: Input-M ∧ R = Ra ∪ Rb
                  ∧ aptr = Ra ∧ bptr = Rb
                  ∧ atombit = Atom ∧ nil.atom
                  ∧ markbit = Mem × {false}
   m := {Z}; Z.mark := true ;   p := Z ;
   Create/Stack(t) ;   tp := nil ;
   loop asserting Invar-DSW:
        Invar-M5 ∧ m = {y: y.mark}
        ∧ SameStack(t,tp)
        ∧ Mod(aptr,bptr,atombit,tp,p)=(Ra,Rb,Atom)
      loop asserting Invar-DSW
         while not((p.a.mark or p.a.atom) and
                   (p.b.mark or p.b.atom))   do
            if not (p.a.mark or p.a.atom)
              then q := p.a else q := p.b  fi ;
            m := m ∪ {q} ;  q.mark := true ;
            push(t, <p,R(p)-{q}> ) ;
            if  q = p.a  then
                    p.atom,p.a,tp := true,tp,p
              else          p.b,tp := tp,p  fi ;
            p := q
      endloop
      loop asserting Invar-DSW^{m-{p}}_{m}
         while tp ≠ nil cand ((not tp.atom) or
                   (tp.b.mark or tp.b.atom))  do
            pop(t) ;
            if tp.atom then
                    tp.atom := false ;
                    p,tp.a,tp := tp,p,tp.a
              else  p,tp.b,tp := tp,p,tp.b  fi
      endloop
      while tp ≠ nil do
         q := tp.b ;

         m := m ∪ {q} ;   q.mark := true ;
         Replace Top-of t with  <t1, t2-{q} > ;
         tp.atom := false ;
         p,tp.b,tp.a := tp.b,tp.a,p ;
         p := q
   endloop
Assert Output-DSW: Output-M
          ∧ {q: q.mark} = R*(Z)
          ∧ (aptr,bptr,atombit) = (Ra,Rb,Atom)
```

Notation: $tn \triangleq (top(t))_n$ for n=1,2

$t^* \triangleq$ if empty(t) then ϕ else t1 $\cup$ (pop(t))* fi

DefStack(t) $\equiv$ empty(t) $\vee$ [t1 $\in$ m $\wedge$ t2 $\subseteq$ R(t1) $\subseteq$ t2 $\cup$ m
$\wedge$ DefStack(pop(t))]

FIGURE 2a -- Algorithm MA-5

cptr $\triangleq$ {<y,y.c>: y in memory $\wedge$ ~y.atom
$\wedge$ ~y.c.atom} for c = a,b

fbit $\triangleq$ {<y,y.f>} for f = atom, mark

(see App. A1 for SameStack, Mod)

FIGURE 2b -- Algorithm MA-6

Although $R^*(Z)$ appears in the assertion Input-C, the final copying algorithms start with only Z and R given, and must traverse the original graph (i.e. construct $R^*(Z)$) in order to copy it. If we assume for the moment that values of the function D are known a priori, we can obtain our initial list-copying algorithm CA-0:

CA-0:

```
   Assert Input-C: Input-M
(1a)               ∧ C ∩ Mem = φ ∧  D:R*(Z) → C
(1b)               ∧ D a bijection

      m := {Z} ;

(2)   σ := R ∪ copy(Z) ;

      loop asserting Invar-C0: Invar-M0 ∧  R ⊆ σ
(3)              ∧ (σ - R) ⊆ C × C ∧  σ - R = copy(m)
         while  m ⊂ R*(Z)  do
            Select p in m
                 satisfying not R(p) ⊆ m ;
            m := m ∪ R(p) ;
(4)         σ := σ ∪ copy(R(p))
      endloop

   Assert Output-C: Output-M
(5a)               ∧ R ⊆ σ  ∧ (σ - R) ⊆ C × C
(5b)               ∧  y ∈ R*(Z): σ(D(y))=D(σ(y))
```

by applying a transformation to MA-0 in order to add the lines numbered above.
In line (2) copy(Z) denotes $\{\langle D(Z),D(y)\rangle: y \in R(Z)\}$ i.e. the set of copies of edges from node Z (speaking of the list structure as a graph); in lines (3) and (4), copy(w) is $\cup$ copy(y): $y \in w$ for w equal to m or R(p) respectively. Hence in CA-0, the edges from each node y are copied as y is added to m (encountered in the traversal). Given a verification of MA-0, to verify CA-0 with respect to Input-C, Output-C we need only check that the premises of the applied transformation are satisfied, by 1) establishing the invariance of line (3) and 2) noting that Invar-C0 ∧ Output-M ⇒ Output-C. The proof of termination uses the same variant function as for MA-0.

4. THE ARCHETYPAL LIST-COPYING ALGORITHM (ALCA)

4.1 Motivation of the archetype

Three new issues relevant to list copying (but not list marking) influence our derivation of the copying archetype ALCA from CA-0.

4.1.1 Generating the bijection D

In the final "target" copying algorithms a pre-existent D, or old-cell/new-cell pairing, is not supplied; thus ALCA constructs a mapping which satisfies CA-0's Input-C assertion. This results in a multi-pass structure for the archetypal (and final) algorithms: one traversal of the graph to define a D (and identify $R^*(Z)$), followed by a second traversal to do the copying.

4.1.2 Specifying traversal/copy order

The target algorithms of interest to us operate under a bounded workspace constraint -- i.e. only a small, fixed amount of additional storage is available to the algorithm apart from the original and copy list structures. As a result, implementing the abstract data structures of CA-0 (and the copying archetype) will involve considerable re-arrangement of the original list contents, which must nonetheless be restored to their initial values once the copying is complete. Verifying that this is accomplished requires specification of the exact traversal order in the invariants, and to this end the archetypal algorithm defines a spanning tree for the original digraph, greatly simplifying the assertions necessary for defining the traversal and copying order.

4.1.3 Edge-oriented copying and traversal

In CA-0, all edges from any node p are copied simultaneously (e.g. at line (4), for $p \neq Z$). In the final algorithms of Section 5, however, edges (pointers) from a given cell are in general copied at different times. This suggests an edge-oriented copying process for ALCA, since once copying is postponed for some edges, the set m of visited *nodes* can no longer fully characterize how much copying has been done, as it does in CA-0. Instead, the assertions in Pass 2 of ALCA, where the copying is done, define the set of edges copied independently of m, and all loop tests and traversal assertions are expressed in terms of edges rather than nodes.

4.2 The copying archetype in detail

As Section 2 presented an extended example of the use of correctness-preserving program transformations, we omit here the intermediate algorithms (several of them similar to MA-1 through MA-5) lying between CA-0 and ALCA. Pass 1 of the archetype (Figure 3a) is a proper extension of MA-5; Pass 2 (Figure 3b), which is derived from CA-0 independently of Pass 1, has a very similar structure. The additions and changes to MA-5 (reflecting the comments of Sec. 4.1) are discussed below.

4.2.1 Bijection generation (Pass 1)

The copy-area assertion Input-ALCA guarantees that the set avail -- those cells initially available for constructing the copy -- contains a subset which can satisfy Input-C of algorithm CA-0. For each node in the original graph encountered during the Pass 1 traversal, a new cell is transferred from avail to c via a Select/Move statement (an abbreviation for Select n in avail; avail := avail - {n}; c := c ∪ {n}) and the pairing is added to the *constructed* bijection μ (now referred to as (the lowercase) μ to distinguish it from the constant D of CA-0). A new clause in the invariant -- Bijection (μ,m,c) -- asserts μ to be a bijection from m to c; thus upon termination of Pass 1, μ maps $R^*(Z)$ onto c, satisfying the a priori bijection assertion of Input-C.

4.2.2 Defining the spanning tree (Pass 1)

The only other extension of Pass 1 beyond MA-5 is a standard partitioning of the edges of the list structure into sets (s and b) of spanning-tree and back edges, respectively, as asserted by SpanTree(s,b,m). While not all final copying algorithms implement sets s and b, their presence (if only as 'ghost' variables) greatly simplifies our proofs of correctness, since the spanning tree (defined by) s is a recursively-describable struc-

```
Assert Input-ALCA: Input-M ∧ avail ∩ Mem = φ
                   ∧ |avail| ≥ |R*(Z)|
  m := {Z} ;  p := Z ;  Create/Stack(t) ;
  c := φ ;  Select/Move n from avail to c ;
  μ := {<Z, n>} ;  s := φ ;  b := φ ;
  loop asserting Invar-ALCA1:
        Invar-M5 ∧ |avail| ≥ |R*(Z) - m|
        ∧ Bijection(μ,m,c) ∧ SpanTree(s,b,m)
    loop asserting Invar-ALCA1
      while not R(p) ⊆ m do
        Select q in R(p)
             satisfying not q ∈ m ;
        m := m ∪ {q} ;
        push(t, <p,R(p) - {q}>) ;
        s := s ∪ {<p,q>} ;
        Select/Mpve n from avail to c ;
        μ := μ ∪ {<q,n>} ;
        p := q
    endloop
    b := b ∪ {<p,ql>: ql ∈ R(p)} ;
    loop asserting Invar-ALCA1^{m-{p}}_{m}
      while (not empty(t)) cand t2 ⊆ m do
        b := b ∪ {<tl,ql>: ql ∈ t2} ;
        pop(t)
    endloop
    while not empty(t) do
      Select p in t2 satisfying not p ∈ m ;
      m := m ∪ {p} ;
      Replace Top-of t with <tl,t2 - {p}> ;
      s := s ∪ {<tl,p>} ;
      Select/Move n from avail to c ;
      μ := μ ∪ {<p,n>} ;
  endloop
Assert Output-ALCA1: Output-M
                     ∧ Bijection(μ,R*(Z),c)
                     ∧ SpanTree(s,b,R*(Z))
```

(see Appendix A2 for Bijection, SpanTree)

FIGURE 3a -- Pass 1 of ALCA

ture with respect to which traversal order can be statically defined: e.g., "Pass 1 traverses the spanning tree in preorder", to paraphrase a typical assertion from Section 5. Such invariants become more complex when they must be expressed solely in terms of traversal order over a (possibly cyclic) digraph.

4.2.3 Edge-copying and -traversal (Pass 2)

In parallel with the 'node' notation used so far we now define

R as also a set of edges, each $e \triangleq \langle e_1, e_2 \rangle$ in R representing a directed edge from (node) e_1 to (node) e_2

```
Assert Input-ALCA2: Output-ALCA1 ∧ Ze_2 = Z
  et := {Ze} ;  e := Ze ;  Create/Stack(tt) ;
  σ := R ;
  loop asserting Invar-ALCA2:
        Ze, e ∈ et ∧ et ⊆ I*(Ze)
        ∧ I((et-{e}) - tt*) ⊆ et
        ∧ DefttStack(e,tt)
        ∧ Input-ALCA2
        ∧ σ - R = copyof(et - {Ze})
    loop asserting Invar-ALCA2
      while I(e) ∩ s ≠ φ do
        Select e' in I(e) satisfying e' ∈ s ;
        et := et ∪ {e'} ;
        σ := σ ∪ copyof(e') ;
        push(tt, <e,I(e) - {e'}>) ;
        e := e'
    endloop
    et := et ∪ I(e) ;
    σ := σ ∪ copyof(I(e)) ;
    loop asserting Invar-ALCA2^{et-{e}}_{et}
      while (not empty(tt)) cand
            tt2 ∩ s = φ do
        et := et ∪ tt2 ;
        σ := σ ∪ copyof(tt2) ;
        pop(tt)
    endloop
    while not empty(tt) do
      Select e in tt2 satisfying e ∈ s ;
      Replace Top-of tt with <ttl,tt2 - {e}>) ;
      et := et ∪ {e} ;
      σ := σ ∪ copyof(e)
  endloop
Assert Output-ALCA2: σ - R = copyof(I+(Ze))
```

Notation: copyof(e) is $\{\langle \mu(e_1), \mu(e_2) \rangle\}$

copyof(eset) is $\bigcup$ copyof(e ∈ eset)

(see App. A2 for DefttStack)

FIGURE 3b -- Pass 2 of ALCA

I as a relation between edges defined by: eIe' ⇔ e is incident upon e' (or $e_2 = e'_1$ in terms of nodes); hence $I \subseteq R \times R$, and

Ze as an auxiliary initial edge incident upon node Z, the root; thus I(Ze) = edges out of the root of the list structure.

Like Pass 1, the second pass is a transitive closure algorithm: it copies $I^+(Ze)$, the set of all edges of the graph (not I*(Ze) since Ze is fictitious), just as Pass 1 traverses R*(Z), all the nodes of the graph. Pass 2 has a down and a backup loop just as MA-5 and Pass 1 do. The loop-exit tests in Pass 2 do not refer to membership in et, the set of edges traversed (compare "while not R(p) ⊆ m ..." in Pass 1), but use only the spanning tree defined in Pass 1 to guide traversal: tree

edges are copied and followed, back edges are merely copied. Showing that this results in only uncopied edges being added to et in loop 1 (hence termination) requires a more complex stack assertion, DefttStack, than in Pass 1 (see Appendix A2). Since edges are pushed onto the stack in Pass 2 rather than nodes, we now refer to the stack as "tt".

5. THE ROBSON LIST-COPYING ALGORITHM [11]

Below we present the Robson copying algorithm (RCA) as one implementation of our copying archetype.

5.1 Overview of implementation of abstract data structures

5.1.1 Node structure and relation R

Each node p considered by RCA contains two fields p.L, p.R which contain either pointers to other nodes, or the nil pointer. So R = R1 ∪ R2,

R1(p) = if p.L = nil then ϕ else {p.L}
R2(p) = if p.R = nil then ϕ else {p.R} .

5.1.2 Set m

RCA uses four pointer values -- called "MARK-i flags" for i=0,1,2,3 -- which are distinguishable from any pointer in the original list structure. The Rlink of each original node is overwritten with one of these flags when the node is first encountered, so we interpret

p ∈ m ⇔ p.R = MARK-i,
i.e. m = {y: y.R = MARK-i for i=0,1,2,3}.

5.1.3 Stack t

RCA implements t just as DSW does, with a reversed-pointer linked list. Thus if the current node p lies in the left (right) subtree of some node y on the stack, then the Llink (Rlink) field of y's left (right) descendant in the spanning tree points back to y.

5.1.4 Set avail

RCA obtains cells for the copy list structure from a free-list (i.e. heap allocation) -- so Select/Move n from avail to c is implemented by a statement such as "new(n)", to use Pascal terminology.

5.1.5 Function μ

RCA puts a "forwarding address", or FAddr, which points to the corresponding new cell, into the Llink of each old cell - thus ALCA's "μ := μ ∪ {<p,n>}" is implemented by "p.L := n".

5.1.6 Sets s and b

RCA uses the four values of the MARK-i flags to indicate which of a node's two fields contain tree- or back-edge pointers (nil is considered a back-edge pointer): odd-value MARK-i flags indicate tree-edge Rlinks, flags valued 2 or 3 indicate tree-edge Llinks. So in the implementation

Llinks in s = {<p,p.L>: p contains a MARK-2 or -3 flag},
Rlinks in s = {<p,p.R>: p contains a MARK-1 or -3 flag},

where p.L, p.R denote the original pointer values.

5.2 Overview of program execution

We now describe in general terms the two passes of RCA -- details are covered when the invariant assertions for the Robson algorithm are considered in Sections 5.3 - 5.6 . The discussion below refers to intermediate algorithms RCA1-A and RCA2 (Figures 4a, 6 respectively), which roughly correspond in their degree of refinement to algorithm MA-6 except that stack t remains unimplemented. The reversed-pointer stack implementation appears in algorithm RCA1-B (Figure 4b), as discussed in Section 5.7 .

5.2.1 Pass 1 (Figure 4a)

As each old node, p, is first encountered in loop 1, a new cell n = μ(p) is obtained (as in 5.1.4 above) and p's original pointer values are stored in the Llink and Rlink fields of n. Fields p.L and p.R are then overwritten with a FAddr and MARK-0 flag respectively (5.1.5, 5.1.2).

Figure 5b illustrates this stage in the processing of p. Traversal then continues just as in DSW (MA-6), with the addition that whenever a pointer in p is followed to an unmarked node, the MARK-i flag in p.R is incremented by one (two) indicating that p's Rlink (Llink) is a tree edge (5.1.6). No other processing (e.g. copying) is performed during Pass 1, so at its conclusion we have:

a) defined μ (via FAddrs) for every node in the original list,

b) classified every edge as in s or b (via MARK-i flags),

c) not copied any edges, and all original pointers are accessible (stored in corresponding new cells),

exactly as in ALCA Pass 1. In addition, a depth-first numbering of the nodes of the original list structure is defined by means of an abstract function called df#. This is an auxiliary function introduced solely to facilitate the correctness proofs -- see Section 5.6.2 .

5.2.2 Pass 2 (Figure 6)

The main traversal tests of Pass 2 -- whether a given edge is a spanning-tree edge -- are implemented using the MARK-i flags. In general pointers are copied in loop 2, during backup phase, by placing the appropriate FAddr (of the cell pointed to) into the copy cell (see Fig. 5f). The corresponding old pointer is restored to its original cell field at the same time; thus a FAddr (MARK-i flag respectively) is lost each time a Llink (Rlink) pointer is copied. In loop 1 Rlinks are followed before Llinks when possible, in opposite order to Pass 1 -- this is crucial to the algorithm's success, as explained in Section 5.6.

5.3 Structuring the assertions

One important benefit of using the transformational method of program proving is the assistance it provides in organization of the invariant assertions. That these assertions will be of considerable length for any but the simplest of programs, cannot be denied -- but this need not be fatal to verification efforts, provided assertions

```
Assert Input-RCA1-A: Input-ALCA ∧ R = R1 ∪ R2
                     ∧ Rj ⊆ Σj ⊆ Mem × (Mem ∪ {nil})
                     ∧ MARKset ∩ Mem = φ

  m := {Z} ;  p := Z ;  Create/Stack(t) ;
  σc1,σc2,c := φ ;  μ := {<nil,nil>} ;
  s1,s2,b1,b2 := φ ; σ1,σ2 := Σ1,Σ2 ;
  df# := { <Z,1>} ; dnum := 2 ;

  loop asserting Invar-RCA1-A

    loop asserting Invar-RCA1-A

      Select/Move n from avail to c ;

      μ := μ ∪ { <p,n>} ;

      σc1(n),σc2(n) := σ1(p),σ2(p) ;

      σ1(p),σ2(p) := n,MARK-0 ;

      while not (marked(σc1(n)) and
                 marked(σc2(n)))  do

        if not marked(σc1(n)) then

          q := σc1(n) ; addmark(p,2) ;

          push(t, <p,σc2(n)>) ;

          s1 := s1 ∪ { <p,q>}

        else  q := σc2(n) ; addmark(p,1) ;

              push(t, <p,φ>) ;

              s2 := s2 ∪ { <p,q>} ;

              b1 := b1 ∪ { <p,σc1(n)>}  fi ;

        m := m ∪ {q} ;

        df# := df# ∪ {<q,dnum>}; dnum:=dnum+1 ;

        p := q
      endloop

      bj := bj ∪ { <p,σcj(n)>}          --for j=1,2

      loop asserting Invar-RCA1-A^{m-{p}}_m

        while (not empty(t)) cand
              (t2 = φ or marked(t2))  do

          if t2 ≠ φ then b2 := b2 ∪ {<t1,t2>} fi ;

          pop(t)
      endloop

      while not empty(t) do

        p := t2 ; m := m ∪ {p} ; addmark(t1,1) ;

        Replace Top-of t with  <t1, φ > ;

        s2 := s2 ∪ { <t1,t2>}
    endloop
```

```
Assert Output-RCA1-A: Output-ALCA-1
                      ∧ R*(Z) = {y: 3-Enc(y)}
                      ∧ RCA-1-j        for j=i to v

  Notation:  MARKset ≜ {MARK-i: i = 0,1,2,3}

             marked(y) ≡ if y=nil then true
                         else σ2(y) ∈ MARKset

           addmark(y,n)   increments the MARK-i flag
                          in σ2(y) into a MARK-(i+n) flag

             f(p) := x    for any function f denotes
                          f := (f-{<p,f(p)>}) ∪ {<p,x>}

          see Figure 7a for invariants
```

FIGURE 4a -- Algorithm RCA1-A

```
Assert Input-RCA1-B: Input-RCA1-A
                                  --without MARKset
                                    assertion

  f := Z ;  gf := nil ;

  new(MARK-0) ;  new(MARK-1) ;
  new(MARK-2) ;  new(MARK-3) ;

  loop asserting Invar-RCA1-B

    loop asserting Invar-RCA1-B

      new(newf) ;


      newf.L,newf.R := f.L,f.R ;

      f.L,f.R := newf,MARK-0 ;

      while not (marked(newf.L) and
                 marked(newf.R))  do

        if not marked(newf.L) then

          s := newf.L ; addmark(f,2) ;

          newf.L,gf := gf,f ;


        else  s := newf.R ; addmark(f,1) ;

              newf.R,gf := gf,f ;


        fi ;



        f := s
    endloop


    loop asserting Invar-RCA1-B^{m-{f}}_m

      while gf ≠ nil cand marked(gf.L.R)  do

        if odd(MARK(gf)) then
              gf,newf.R := newf.R,s
          else gf,newf.L := newf.L,s  fi ;

        s,f,newf := f,gf,gf.L
    endloop

    while  gf ≠ nil  do

      f := newf.R ;  addmark(gf,1) ;

      newf.L,newf.R := s,newf.L

  endloop
```

```
Assert Output-RCA1-B: Output-RCA1-A
```

variables in RCA1-A	-B	
p	f	father
q	s	son
t1	gf	grandfather
n	newf	μ(p)

see Appendix A3 for invariants

FIGURE 4b -- Algorithm RCA1-B

FIGURE 4

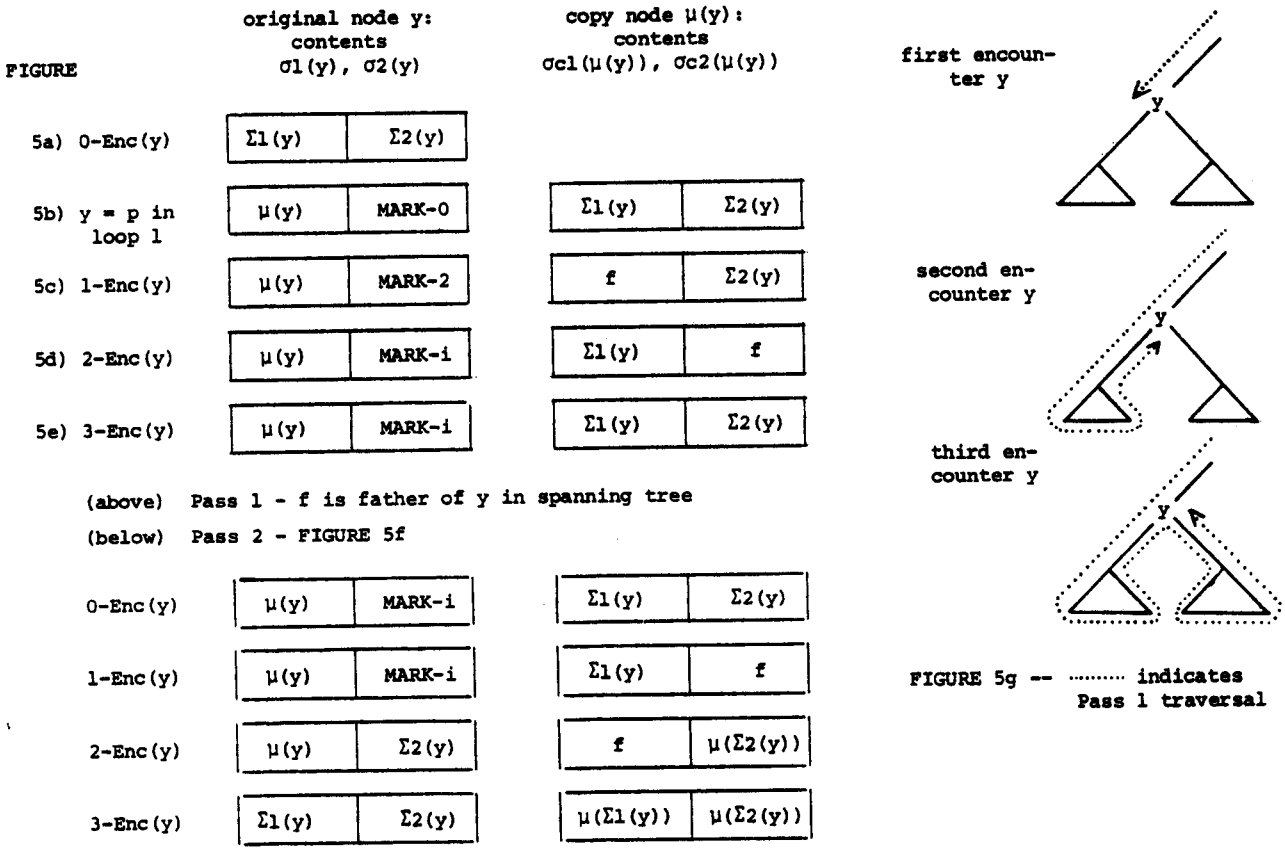

FIGURE 5g -- ..... indicates Pass 1 traversal

FIGURE 5

are not thought of and produced as formless expressions in first-order predicate logic. Assertions become much more manageable when their structure clearly reflects (one's understanding of) the functioning of the program which they describe. We feel this is a property shared by our invariants, by virtue of their incremental construction in the context of increasingly-refined algorithms.

The concept we use to organize the assertions needed for the final copying algorithms is node status. A natural way to describe, in the course of Pass 1 or Pass 2, how much processing a node has undergone is to refer to how many times the node has been "encountered" thus far in the traversal. Since RCA uses Deutsch-Schorr-Waite traversal, the possibilities are 0, 1, 2 or 3 encounters (see Figure 5g). Correspondingly we employ four predicates to partition the nodes of the original graph -- referring to the abstract traversal stack t, we define

0-Enc(y) as $\sim(y \in m')$

1-Enc(y) as $y \in t^* \wedge t2\text{-of}(y,t) \neq \phi$

2-Enc(y) as $y \in t^* \wedge t2\text{-of}(y,t) = \phi$

3-Enc(y) as $y \in m' \wedge \sim(y \in t^*)$

where m' denotes m - {p} in loop 1 of Pass 1, m in loop 2. (For Pass 2 interpret m' as $\{Z\} \cup \{e_2: e \in (et \cap s)\}$, and replace t with tt -- see Appendix A5). Given that node y is on stack t, t2-of(y, t) is the set of nodes (in Pass 2, edges) pushed onto t along with y. Thus, intuitively, n-Enc(y) is true just when y has been encountered (in the DSW sense) n times.

Virtually all of the new invariant clauses for the Robson algorithm not present in ALCA are of the form "n-Enc(y) ⇒ ...". This format for the assertions clarifies the description of exactly how RCA first builds up (in Pass 1), then dismantles (in Pass 2) its in situ implementations of ALCA's copy-related abstractions μ, s and b. In the DSW implementation (MA-6) of MA-5, in contrast, only the invariant "Mod(...)=(Ra,Rb,Atom)" is needed to define the progression of the single in-place implementation (of stack t) which must be removed before termination of the algorithm.

5.4 On verifying Pass 1

The invariant clauses for Passes 1 and 2 of RCA appear in tabular form in Figures 7a, 7b respectively. This figure is read as, the n-Enc(y) predicate beginning each row implies (the conjunction of) all the entries in that row. Each column comprises a single assertion, of one of the types (perturbation, equivalence) introduced in Section 2.4. Thus reading across a row defines the state of, e.g., all 1-Enc nodes; reading down a column illustrates the stages that, say, MARK-i flags go through in a given pass. Note in particular that since at the end of Pass 1 every node in R*(Z) is a 3-Enc node (obvious since by the outer-loop exit test empty(t) holds, and also m = R*(Z), an old friend by now), reading across the bottom row of Figure 7a yields the copying part of the RCA Pass 1

Output assertion. As every node of the graph is again 0-Enc at the start of Pass 2 by definition of n-Enc, the top line of Figure 7b is identical to the bottom line of Figure 7a, i.e. Output-RCA1 ⇔ Input-RCA2, as desired.

For the most part the copying assertions (columns Invar-RCA1-i through -v) for Pass 1 of RCA simply reiterate Figures 5a-e. RCA1-i says original pointer values are stored in the corresponding copy cell. We use (the constant) $\Sigma 1(y)$, $\Sigma 2(y)$ for the original Llink, Rlink values of node y, rather than Ra, Rb as in the DSW assertions, as a reminder that the Robson pointer relations are subsets of Mem × (Mem ∪ {nil}), and extensions of ALCA's $R \subseteq Mem \times Mem$. This is why $\sigma ci(y)$ is used to denote the pointer value of a copy node y (rather than σ - R as in ALCA); σi is used to refer to current pointer values in the original list structure (similarly to aptr, bptr of DSW). Note that if, for example, both of p's pointers are nil, loop 1 of the implemented algorithm is exited. This is one case of the abstract condition $R(p) \subseteq m$, since R(p) (as defined in 5.1.1) is empty when p.L = p.R = nil.

Of the four remaining equivalence assertions, m-MARK and μ-FAddr(RCA1-ii, -iii) are self-explanatory; the sb-MARK assertion (RCA1-iv) indicates how the information content of the MARK-i flag increases as first a node's Llink, then Rlink are examined during Pass 1 traversal; and the t-MARK assertion (RCA1-v) defines the possible values of the MARK-i flags for each n-Enc group. This assertion is needed in Pass 1 since the flag's value is *incremented* each time a tree edge is discovered - it is unnecessary in Pass 2 and does not appear in Figure 7b.

5.5 The need for availability assertions

We use the term availability assertions in referring to Pass 2's equivalence assertions, to indicate the role played by these invariant clauses in establishing the correctness of the second pass. The need for these invariants is a straightforward consequence of the bounded-workspace constraint on the Robson algorithm -- since FAddrs and MARK-i flags are removed during Pass 2, our assertions must be strong enough to prove that at every appearance of a "μ(y)" or "e ∈ s" in algorithm RCA2 the necessary FAddr or MARK-i flag is still present.

Availability assertions are unnecessary for the copying archetype, since in ALCA the abstract data structures μ, s and b are present throughout the course of the algorithm. Note also that in the implementation of DSW (algorithm MA-6) availability assertions are not used because stack t can only be accessed through the variable tp (cf. comments made about MA-3). There is, however, no simple restriction on when any given node's FAddr or MARK-i flag will be needed in Pass 2 -- thus μ, s and b are represented as sets in ALCA and must be implemented as (approximately) random-access data structures. The next section explains, using availability assertions, the precise extent of that approximation.

5.6 On verifying Pass 2

Note that instead of referring to a 'current' edge e in the traversal, as in Pass 2 of ALCA, we write p and q in RCA2 for (the head and tail nodes respectively of) ALCA's e . Correspondingly

```
Assert Input-RCA2: Output-RCA1-A
   q,p := nil, Z ;
   et1,et2 := { <q,p>} , φ ;  Create/Stack(tt) ;
   loop asserting Invar-RCA2          --see App. A4
      loop asserting Invar-RCA2
*-1      while  E(p) ∩ s ≠ φ  do
*-1         if  E2(p) ⊆ s2  then
               et2 := et2 ∪ E2(p) ;
               push(tt, <p,E1(p)>) ;
               q,p := p,Σ2(p)
            else etj := etj ∪ Ej(p) ;
               push(tt, <p,φ>) ;
**-1           σ2(p),σc2(μ(p)) := Σ2(p),μ(Σ2(p)) ;
               q,p := p,Σ1(p)
         fi
      endloop
      etj := etj ∪ Ej(p) ;   push(tt, <p,φ>) ;
      cson := μ(Σ1(p)) ;
**-2  σ2(p),σc2(μ(p)) := Σ2(p),μ(Σ2(p)) ;
      loop asserting Invar-RCA2^{et-{e}}_{e}
                     ∧ cson = μ(rson(ttl))
         while  (not empty(tt)) cand
*-2                 tt2 ⊆ b1  do
            if tt2 ≠ φ then
**-3           σ2(ttl),σc2(μ(ttl)) :=
                        Σ2(ttl),μ(Σ2(ttl)) fi ;
            et1 := et1 ∪ E1(ttl) ; cson := μ(ttl) ;
**-4        σ1(ttl),σc1(μ(ttl)):=Σ1(ttl),μ(Σ1(ttl));
            pop(tt)
      endloop
      while  not empty(tt)  do
         et1 := et1 ∪ E1(ttl) ;
**-5     σ2(ttl),σc2(μ(ttl)) := Σ2(ttl),μ(Σ2(ttl));
         q,p := ttl,Σ1(ttl) ;
         Replace Top-of tt with  <ttl, φ >
   endloop
Assert Output-RCA2:   y ∈ R*(Z): σj(y) = Σj(y)
                       ∧ σcj(μ(y)) = μ(Σj(y))
  Notation: rson(ttl) ≜ if 1-Enc(ttl) then Σ2(ttl)
                                      else Σ1(ttl)
     each assignment stmt S containing a j
                                  denotes S^j_1 ; S^j_2
```

FIGURE 6 -- Algorithm RCA2

the first component of each stack element in RCA2 is written as a node.

This essentially notational change in stack tt is made in order to simplify the implementation assertions, since in RCA2 it is more convenient to think of implementation values (FAddrs, etc.) as stored in the current node rather than in (the head of) the current edge. Recall that in Pass 2 of ALCA, nodes are not mentioned to emphasize that

	Perturbation assertion	m-MARK equivalence	μ-FAddr equivalence	sb-MARK equivalence	t-MARK equivalence
	RCA1-i	RCA1-ii	RCA1-iii	RCA1-iv	RCA1-v
0-Enc(y) ⇒	$\Sigma j(y) = \sigma j(y)$	~MARKed(y)	--	$\sim(Ej(y) \subseteq s \cup b)$	~MARKed(y)
1-Enc(y) ⇒	$\Sigma j(y) = \sigma cj(\sigma 1(y))$	MARKed(y)	$\sigma 1(y) = \mu(y)$	$(E1(y) \subseteq sl \Leftrightarrow MARK(y) \geq 2) \wedge \sim(E2(y) \subseteq s \cup b)$	MARK(y) ∈ {2}
2-Enc(y) ⇒				$E1(y) \subseteq sl \Leftrightarrow MARK(y) \geq 2 \wedge E2(y) \subseteq s2 \Leftrightarrow$ odd MARK(y)	MARK(y) ∈ {1,3}
3-Enc(y) ⇒					MARK(y) ∈ {0,1,2,3}

$$\text{Invar-RCA1-A} \equiv \text{Invar-ALCA1}^{\text{DefStack(t)}}_{\text{DefStackR1(p,t)}} \wedge \bigwedge_{k=i}^{v} \text{RCA1-k} \wedge \text{Depth-First(df\#,m)}$$

FIGURE 7a -- Robson Pass 1 invariants

	Perturbation assertion	MARK availability	FAddr availability	sb-MARK availability	Edges Copied assertion
	RCA2-i	RCA2-ii	RCA2-iii	RCA2-iv	RCA2-v
0-Enc(y) ⇒	$\Sigma j(y) = \sigma cj(\sigma 1(y))$	MARKed(y)	$\sigma 1(y) = \mu(y)$	$E1(y) \subseteq sl \Leftrightarrow MARK(y) \geq 2 \wedge E2(y) \subseteq s2 \Leftrightarrow$ odd MARK(y)	$\mu(E(y)) \cap \sigma c = \phi$
1-Enc(y) ⇒					
2-Enc(y) ⇒	$\Sigma 1(y) = \sigma c1(\sigma 1(y)) \wedge \Sigma 2(y) = \sigma 2(y))$	~MARKed(y)		--	$\mu(E1(y)) \cap \sigma c1 = \phi \wedge \mu(\Sigma 2(y)) = \sigma c2(\mu(y))$
3-Enc(y) ⇒	$\Sigma j(y) = \sigma j(y)$		--	--	$\mu(\Sigma j(y)) = \sigma cj(\mu(y))$

$$\text{Invar-RCA2} \equiv \text{Invar-ALCA2}^{\text{DefttStack(e,tt)}}_{\text{DefStackR2(p,tt)}} \wedge \bigwedge_{k=i}^{v} \text{RCA2-k} \wedge \text{Depth-First(df\#,R*(Z))} \wedge \text{CopyOrder}$$

FIGURE 7b -- Robson Pass 2 invariants

Notation: $A \equiv A_1^j \wedge A_2^j$ for any assertion A containing j

$\text{MARKed(y)} \equiv \sigma 2(y) \in \{\text{MARK-0, -1, -2, -3}\}$

$Ei(y) \triangleq \{<y, \Sigma i(y)>\}$ for i = 1,2

$E(y) \triangleq E1(y) \cup E2(y)$

$\text{MARK}(y) = n$ iff $\sigma 2(y) = \text{MARK-}n$ (n = 0,1,2,3)

FIGURE 7

(for definitions of n-Enc, stack assertions, Depth-First and CopyOrder see Appendix)

the transitive closure of the *edge*-incidence relation is being copied.

In algorithm RCA2, loop tests and assignment-statement right hand sides are still expressed abstractly; the next transformation to be applied will replace "E(p) ∩ s ≠ ϕ " with "MARK(p) > 0 ", "p := Σ2(p)" with "p := p.L.R", etc. In the discussion below the intermediate algorithm referred to may be either RCA2 or its successor (omitted due to lack of space) -- the availability assertions for the two are the same.

5.6.1 MARK-i flag availability

Every implementation of an "e ∈ s" test in Pass 2 (marked by (*) in Figure 6) is applied to an edge from either a 0-Enc node (node p at (*-1)), or a 1-Enc node (node tt1 in statement (*-2), since the test is only made if tt2 ≠ ϕ, implying 1-Enc(tt1)). Thus by the availability assertion Invar-RCA2-iv (which could be paraphrased as

0-Enc(y) ∨ 1-Enc(y) ⇒ "y.R is a valid MARK-i flag"),

the implemented version of the tree-edge test is correct. The invariance of RCA2-iv is an immediate consequence of the fact that the only edge possibly copied in loop 1 ("going down") is the back-edge Rlink of a node y (if it has one), after which y is a 2- or 3-Enc node (depending on whether or not y's Llink is found to be a tree edge).

5.6.2 FAddr availability

Here the assertions must imply that when any edge, e, is copied by a statement marked (**) in RCA2, that the needed μ-value for e -- i.e. μ(Σi(p)) or μ(Σi(tt1)) -- is available. For tree edges (**-3,4,5) this is accomplished by using in loop 2 an extra variable cson (copy son), along with a corresponding invariant clause stating that cson stores the FAddr of the node up from which we are returning. Since all tree edges are copied in "backup phase", and each node has a unique tree ancestor, the single variable suffices.

For back edges the case is more complex, as an arbitrary number of back edges, from any node in the list, may point to a given node y. As the FAddr-availability assertion (RCA2-iii) states:

0-Enc(y) ∨ 1-Enc(y) ∨ 2-Enc(y) ⇒ σ1(y) = μ(y),

we may (indirectly) demonstrate the validity of (**-1,2,4) by establishing

(1) 3-Enc(y) ⇒ μ({e ∈ b: e_2 = y}) ⊆ σc

where σc denotes the set σc1 ∪ σc2 of edges in the copy list structure. The invariance of (1) is a consequence of Pass 2's traversing the list structure in reverse order (Rlink before Llink) from Pass 1. A detailed argument of (1)'s invariance is greatly facilitated by the presence of df# and the Depth-First assertion (see Appendix A5), which enable a simple case argument (omitted here for lack of space) to be made.

Note that we certainly do not claim the above discussions to be proofs of the invariance of RCA2-iv or assertion (1) above; they are merely intuitive arguments to that end. In this article we emphasize the construction and nature of the invariant assertions themselves, and have deliberately omitted formal proofs of their invariance. For that reason this article should not be considered to be an example of formal verification, but rather an exposition on some methods useful in formal verification -- see Section 7.1 .

5.7 The final transformations

Applying a transformation to algorithm RCA1-A to implement stack t (just as with the MA-5/MA-6 transformation) yields RCA1-B (Figure 4b), in which we revert to Robson's original variable names. (The correspondence with RCA1-A's variables is noted in the accompanying table.) This is done both to facilitate comparison with the program as originally presented in [11] and because Robson's mnemonics are more suggestive once the DSW stack is introduced.

Given the assertions of RCA1-B, together with some elementary path analysis, one can then apply to RCA1-B the final transformation, obtaining Pass 1 of the Robson algorithm exactly as originally presented. That version bears approximately the relationship to RCA1-B that MA-3 has to MA-4, since each pass of Robson's program is written as a conditional statement within a single while loop. Pass 2 of the Robson program is obtained from RCA2 via an identical sequence of transformations.

5.8 Beyond the Robson algorithm

One advantage of the Robson copying algorithm is its adaptability to any system of copy-cell allocation. In some applications, this is an important consideration -- if, in the presence of garbage collection, say, all free cells must be organized as a free-list. However, more efficient algorithms can be obtained by implementing ALCA's storage pool by means of a contiguous region of memory, addressed by values greater than Amin, for example. This seemingly minor implementation restriction has a significant effect on the efficiency of the resulting algorithms, as it eliminates the need for traversal flags. (Forwarding addresses are recognizable as pointers whose values exceed Amin, and thus by their presence mark visited cells.) The extra workspace made available by this double use of the forwarding addresses is utilized in two ways.

1) All copying need no longer be postponed until Pass 2. Consider Figure 5b -- once the MARK flag becomes unnecessary, only the node's Llink must be saved in the copy cell; thus one new pointer for the copy list structure can now be inserted in Pass 1.

2) Different types of traversal may be used, as follows. The Fisher copying algorithm, presented in [5], can be otained from algorithm CA-0 by a derivation which, at the transformation corresponding to the one producing algorithm MA-4 in Section 2.2, organizes the set, ul, of cells containing unexamined pointers, as a queue rather than as a stack. Subsequent transformations then employ the contiguous copy region as an array in order to implement the traversal queue. The principal intermediate algorithm in this derivation is, apart from its use of a queue, essentially the same as ALCA. The Clark list-copying algorithm, from [3], is the current state-of-the-art in terms of execution time. As its traversal method is stack-oriented, its derivation parallels that of RCA up to and including the archetypal algorithm.

However, in subsequent refinements, both pointers of the current node are always examined in loop 1. Only those nodes containing two tree edges are pushed onto the stack, thus speeding up traversal of the original list structure.

Our unpublished work on the Fisher and Clark algorithm adds support to the opinion that the refinement approach is particularly advantageous when verifying a family of related algorithms.

6. PHILOSOPHY

In assessing the significance of program-correctness research, we think it important to present some personal philosophy. By "demonstrating program correctness" we mean "establishing that a given program does what we want it to do", or more precisely, "increasing our confidence that a given program does what we want it to do". *This* is the desired end, which can be attained by a variety of means:

1. There are empirical methods, such as program testing, by which we increase our confidence in a program's behavior. Since this increase results from an inductive inference based on a particular experimental result, testing by itself is not the ultimate solution to problems of program correctness: "testing cannot detect the absence of bugs, only their presence" as the proverb goes.

2. There are logical methods, such as program verification, which employ (either formal or informal) arguments that a program "does what we intend". The advantage of an informal method is that is deals directly with our intuitive notions of what the program should accomplish; however, a verbal analysis cannot take full advantage of current logical techniques of program verification, such as fixedpoint induction or our transformational approach. Once we express our arguments completely within a formal system (e.g. predicate calculus), however, we face the problem of translating our original intentions as to "what we want the program to do" into mathematically precise formal specifications. This problem of formal specification of real-world programs is a concern separate from the issues which we address in the current article. Despite our optimism about the progress being made (by others and ourselves) in the area of machine-checkable proofs of large programs, we grant that further progress in formal specification techniques is necessary before The Day arrives (if ever) that formal verification, *alone*, can claim to "solve" the problem of establishing program correctness.

7. CONCLUSION AND RELATED WORK

7.1 Conclusion

We see our version of the transformational method as making three contributions:

1. As presented in this paper, it can be used as an aid to informal verification: Sections 3, 4 and 5 argue the correctness of the final copying algorithm by using our framework of abstractions and implementations to motivate the invariance of the loop assertions. That (verbal) reasoning does not formally verify the Robson algorithm, but we feel that explaining an algorithm in terms of its high-level structure is a more effective way of persuading oneself of the algorithm's correctness than is considering only the final program text. Also, the use of intermediate algorithms reduces the complexity (and hence increases the credibility) of the individual components of the correctness argument.

2. The axiomatic definition of our transformation schemata as presented in full in [7] establishes the correctness-preserving property of the transformations -- thus the correctness of algorithm MA-6 is a function solely of the validity of the application of the transformations used in its derivation, and the correctness of our initial algorithm, MA-0. This fact may be used to formally verify MA-6 without ever having to independently establish the invariance of any loop assertions (except that of the initial algorithm). In verifying large programs, this may be an easier approach than the traditional method of proving, in one stroke, the invariance of the final, extremely complex loop assertion.

3. Any formal verification should closely reflect one's understanding of why an algorithm works. Our methodology both reflects and supports that opinion, and provides another "counterexample to much-propagated and hence commonly held belief that there is an antagonism between rigour and formality on the one hand and 'understandability' on the other" as Dijkstra says in [4].

7.2 Related work

While in this summary of our initial efforts with list-copying algorithms we have spoken in terms of larger-scale program transformations than those described in [6], [7], this larger scale is particularly important when the intermediate algorithms are first formulated for complex programs. Blikle [1] considers some correctness-preserving program transformations defined in terms of an algebra of binary relations, and uses them to derive a small but highly-optimized square root algorithm.

Topor [15] was the first to prove correctness of the Deutsch-Schorr-Waite algorithm. Yelowitz was among the first to apply the refinement technique to verifying list-marking algorithms -- see Yelowitz and Duncan [16] for an alternate, more formal definition of the abstractions of Section 2. Various implementations of an abstract backtracking algorithm are considered by Gerhart and Yelowitz in [9]. The very elegant program transformation work of Burstall and Darlington in [2] is comparable in spirit to our approach; however, their technique appears best suited to (algorithms processing) recursive data structures such as trees, in a simple applicative framework. The imperative features required to handle pointer manipulation and digraphs dictate a more complex memory formalism -- obtained in our method by starting out with an abstract memory representation (cf. the σ function of algorithm CA-0), and then implementing it in later transformations.

De Roever in [12] presents proofs of correctness, employing (both least and greatest) fixed-point techniques, of a group of bounded-workspace traversal and backtracking algorithms, including that of Deutsch-Schorr-Waite, and focuses on the similarity in their proofs without using trans-

formations (see [13] for termination proofs for the algorithms). It was the attempt to extend this work to some difficult list-copying algorithms which brought out the importance of using the explicitly structured techniques of the current paper.

ACKNOWLEDGEMENTS

We wish to warmly thank the University of Utrecht for its generous support of the international collaboration involved in the writing of this paper. The Tuesday Afternoon Club, especially E. W. Dijkstra, provided many valuable comments on an earlier version of this paper. Thanks also to Emmy Busch and Sandy for their typing; and to Stephanie for moral support.

APPENDIX

Discussion of assertions omitted for lack of space.

A1. Alg. MA-6 SameStack(t,tp) ≡ empty(t)∧tp=nil
∨ [tl=tp∧ [(tp.atom∧t2=tp.b∧SameStack(pop(t),tp.a))
∨ (t2= ϕ ∧SameStack(pop(t),tp.b))]]
Mod(aptr,bptr,atombit,tp,p) =def
if tp=nil then (aptr,bptr,atombit) else if tp.atom
then Mod((aptr-{<tp,tp.a>})∪{<tp,p>},bptr,
(atombit-{<tp,tp.atom>})∪{<tp,false>},tp.a,tp)
else Mod(aptr,(bptr-{<tp,tp.b>})∪{<tp,p>},
atombit, tp.b,tp) fi

A2. ALCA Bijection(μ,w,c) ≡ μ:w→c ∧ μ(a)=μ(b)⇒
a=b ∧ {y: ∃v∈w: μ(v)=y} = c where w = m,R*(Z)
SpanTree(s,b,w) ≡ {e2:e∈s}={e2:e∈s∪b}=w-{Z} ∧
b∩s= ϕ ∧|s|=|w| -1 ∧ I(E(m-{p})-t*)⊆ (s∪b)⊆ I+(Ze)
DefttStack(e,tt) ≡ empty(tt) ∨ [ttl ∈ et
∧ Ib(ttl)⊆ tt2⊆ I(ttl)⊆ et ∪tt2
where Ia(e)= ∧ (tt2∪IIs*(tt2∩s)) ∩ et = ϕ
I(e) ∩ a for ∧ e ∈ Is(ttl)
a = s,b ∧ DefttStack(ttl,pop(tt))]

A3. Alg. RCA1 DefStackR1(p,t) ≡ empty(t)∨[tl∈m
∧ [(t2=Σ2(tl) ∧ <tl,p>∈sl)
∨(t2= ϕ ∧ <tl,p>∈s2 ∧ Σ1(tl)∈m)]
∧ DefStackR1(tl,pop(t))]
Invar-RCA1-B ≡ ∃m,t,μ,s,b: Invar-RCA1-A ∧ Same-
StackRCA(t,gf) where i) σcj in RCA-i is replaced by σc'j, def'd. by (σc'1,σc'2)=ModRCA(σc1,σc2,gf,f)
ii) SameStackRCA, ModRCA are virtually identical to DSW assertions in App. A1.

A4. Alg. RCA2 DefStackR2(p,tt) ≡ empty(tt) ∨
[<sinv(ttl),ttl>∈et
∧[(tt2=E1(p) ∧ <ttl,p>∈s2∩et ∧~(tt2⊆ et)
∧ tt2⊆ sl=IIs*(tt2)∩et=ϕ)
∨(tt2= ϕ ∧ <ttl,p>∈sl∩et ∧ E2(ttl)⊆ et)]
∧ DefStackR2(ttl,pop(tt))]
where <sinv(ttl),ttl>∈s.

A5. Robson invariants
0-Enc(y) ≡ ~(y ∈ m')
1-Enc(y) ≡ y ∈ stk* ∧ t2-of(y,stk) ≠ ϕ
2-Enc(y) ≡ y ∈ stk* ∧ t2-of(y,stk) = ϕ
3-Enc(y) ≡ y ∈ m' ∧~(y ∈ stk*) , where
m', stk = m-{p}, t in Pass 1
= {Z}∪{e2: e∈(et-{e})∩s}, tt
in Pass 2
and t2-of(y,stk) = if y=(top(stk))$_1$ then
(top(stk))$_2$ else t2-of(y,pop(stk))

Depth-First(df#,w) ≡ Bijection(df#,w,{j: 1≤j≤|w|})
∧ df#(Z) = 1
∧ <x,y>∈sl ⇒ df#(y) = df#(x) + 1
∧ <x,y>∈s2 ⇒ df#(y) = df#(x)+|s*(Σ1(x))|+1
∧ <x,y>∈b1 ⇒ df#(y) ≤ df#(x)
∧ <x,y>∈b2 ⇒ df#(y) ≤ df#(x) ∨ y∈s*(Σ1(x))
where w = m, R*(Z) and s*(q) is the set of nodes in the subtree rooted at the node q (including q).
CopyOrder ≡ df#(y) >df#(p) ∧ ~(y∈s*(p)) ⇒
μ(E(y)) ⊆ σc
≡ df#(y) >df#(ttl) ⇒ μ(E(y)) ⊆ σc ,
in loops 1 and 2 respectively.

REFERENCES

[1] Blikle, A. Towards Mathematical Structured Programming, in *Formal Descriptions of Programming Concepts*, E.J. Neuhold (ed.), North-Holland Publishing Co., 1978.

[2] Burstall, R.M. and J. Darlington. A Transformation System for Developing Recursive Programs. JACM 24 (Jan. 1977), pp. 44-67.

[3] Clark, D.W. A Fast Algorithm for Copying List Structures. CACM 21 (May 1978), pp. 351-7.

[4] Dijkstra, E.W. Finding the Correctness Proof of a Concurrent Program. Proc. Koninklijke Nederlandse Akademie van Wetenschappen, Amsterdam, A 81 (June 9, 1978), pp. 207-15.

[5] Fisher, D.A. Copying Cyclic List Structures in Linear Time Using Bounded Workspace. CACM 18 (May 1975), pp. 251-2.

[6] Gerhart, S.L. Correctness-Preserving Program Transformations. Proc. Second POPL Symp., Palo Alto (1975), pp. 54-66.

[7] Gerhart, S.L. Proof Theory of Partial Correctness Verification Systems. SIAM J. Comp. 5 (Sept. 1976), pp. 355-77.

[8] Gerhart, S.L. Two Proof Techniques for Transferral of Program Correctness, (forthcoming).

[9] Gerhart, S.L. and L. Yelowitz. Control Structure Abstractions of the Backtracking Programming Technique. Proc. Second Intl. Conf. on Software Eng., San Francisco (Oct. 1976).

[10] Knuth, D.E. *The Art of Computer Programming, vol. 1: Fundamental Algorithms*, Addison-Wesley, 1973, Section 2.3.5.

[11] Robson, J.M. A Bounded Storage Algorithm for Copying Cyclic Structures. CACM 20 (June 1977), pp. 431-3.

[12] deRoever, W.P. On Backtracking and Greatest Fixedpoints, in *Proc. Fourth Intl. Conf. on Automata, Languages and Programming*, A. Salomaa (ed.), Springer-Verlag, 1977.

[13] deRoever, W.P. An Essay on Trees and Iteration. Report RUU-CS-78-6, Dept. of Comp. Sci., University of Utrecht, 1978.

[14] Schorr, H. and W.M. Waite. An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures. CACM 10 (Aug. 1967), pp. 501-6.

[15] Topor, R. Correctness of the Schorr-Waite List Marking Algorithm. Memo MIP-R-104, School of Artificial Intelligence, Univ. of Edinburgh, 1974.

[16] Yelowitz, L. and A.G. Duncan. Abstractions, Instantiations, and Proofs of Marking Algorithms. Proc. Symp. on Artificial Intelligence and Prog. Lang., SIGPLAN 12 (Aug. 1977), pp. 13-21.