Another variant of the Schorr-Waite algorithm and its correctness proof -- an exercise in formulation

M.M. Fokkinga, P.G. Jansen

Twente University of Technology
Department of Informatics
Enschede, Netherlands

Abstract
The Schorr-Waite algorithm is of interest for garbage collection: it marks all cells of the store reachable from a given pointer, using only a few simple global varibales and two bits in each cell. We present another variant of this well-known algorithm and prove it correct in an intuitively appealing way.

1 Introduction

The Schorr-Waite algorithm, independently found by [Schorr, Waite 1967] and Deutch [Knuth 1968], is of practical interest because it can be used in a garbage collector to detect the garbage without needing additonal storage apart from two bits in each cell (one of which being the mark bit) and a few global variables containing a simple value each. The algorithm and especially its correctness proof has received considerable attention in the literature: [Gries 1979], [Topor 1979], [Morris 1982], [de Roever 1978], [Kowaltowski 1979], [Jonkers 1982] and others. The algorithm has been called a testbed for proof techniques by [Morris 1982] and a challenge for any proof procedure by [Topor 1979], apparently because it doesn't seem to admit an easy formal correctness proof.

We present another variant of the algorithm, found by P.G. Jansen. The program text is simpler than anyone of those discussed in the literature, although it very much resembles the program given by [Gries 1979]. The structure of the data to be manipulated by the program is clearly reflected in the formal correctness proof; the proof method may be called data directed, but is generally known as the method of intermittent assertions, [Manna 1978]. [Gries 1979] and others give a correctness proof which reflects the structure of the program text; the method may be called syntax directed, but is usually known as the axiomatic method [Hoare 1969]. We argue that for this algorithm the data directed proof is easier to formulate, comprehend and perform than a syntax directed proof. In particular we show that the proof is just a minor adaptation of the correctness proof of a recursive marking algorithm and is therefore in some sense the best possible.

One might say that our proof is merely a reformulation of the proof given by [Topor 1979]. From an abstract point of view this is certainly true. Our work differs from Topor's in two respects. First, we use a notation that has more appeal to the average programmer and seems suitable for more applications in the field of pointer manipulating programs. Secondly, we show that the proof of the Schorr-Waite algorithm really is a minor updating of the proof of the recursive algorithm. Thirdly, we consider a more structured version of the algorithm, thus meeting Gries' complaint that "in all the published literature on the intermittent assertion method, that I have seen so far, the algorithms used to illustrate the method have been quite "unstructured" and therefore difficult to comprehend. It is as if the authors felt that the method gave them licence to dish up raw spaghetti to the reader to digest, with the feeling that the sauce (the proof method) poured over the spaghetti would compensate" [Gries 1979]. Actually we could easily have taken Gries' algorithm; we feel however that our version is worth to be recorded and moreover it has for the correctness proof a slight advantage over Gries' version.

In the next sections we present the algorithm (Section 2), introduce some notation in order to clearly express the task of the algorithm formally (Section 3), give the formal correctness proof of the recursive and iterative version (Section 4 and 5) and discuss the merits of our proof (in Section 6).

2 The Schorr-Waite algorithm

Let us assume that each cell of the store contains exactly two references to other cells and two bits which are used exclusively for marking, and possibly some other fields which are not of interest here. We use the two mark bits to count the number of visits payed to the cell during the marking phase; this is possible because the number of visits to each cell is exactly three, as in a normal tree traversal. Thus the cell, from now on called nodes, are discribed by the following types.

```
types pnode = ↑ node
      node = record c: 0..3; l,r: pnode; ... end
```

Initially all c-fields have the value 0 and it is the task of the algorithm to set the c-field to 3 of each node reachable via the l- and r-fields from a given pnode Root. During this marking the c-field may assume any value in the range 0..3; (one of the bits of the binary representation of the c-value may be termed the mark bit and the other is then an auxiliary flag bit). For ease of presentation we assume that nil is not used; its role may be played by some particular node with both the l- and r-field pointing to that very node itself.

The following recursive algorithm is quite straightforward. Its only drawback is that its execution requires additonal storage for house-keeping of the stack of recursive invocations.

```
procedure Mark (P: pnode);

  if P↑.c ≠ 0 --> skip

  [] P↑.c = 0 --> P↑.c := P↑.c + 1;

                  Mark (P↑.l);

                  P↑.c := P↑.c + 1;

                  Mark (P↑.r);

                  P↑.c := P↑.c + 1;

  fi
```

Of course, we could have collected the three assignments to P↑.c := 3. The above form, however, resembles the Schorr-Waite algorithm more closely. Procedure Mark traverses a spanning tree of the graph to be marked, that is the graph consisting of those nodes N reachable from P along a path all of whose nodes (including P↑ and N) have c-field zero. There is no danger of infinite recursion because Mark immediately returns from nodes P↑ which have already been visited before (P↑.c ≠ 0).

For the Schorr-Waite algorithm, henceforth called SW, we assume that there is one particular pnode VirtualRoot which is not used in any of the l- and r-fields (of nodes initially reachable from Root). This assumption both simplifies the text and the tests, but is inessential in all other aspects. The algorithm uses three global variables

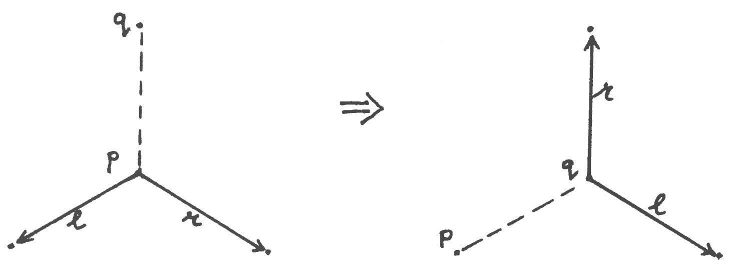
```
var p, q: pnode; b: boolean.
```

The program text of SW reads as follows; an explanation is given below.

```
      p,q,b := Root,VirtualRoot,false;

   L: do p ≠ VirtualRoot -->

        if p↑.c ≠ 0 ∧ ¬ b --> p,q,b := q,p,true

        [] p↑.c = 0 ∨ b --> p↑.c := p .c+1; b := P↑.c=3;

                                rotate (q, p, p↑.l, p↑.r)

        fi

      od
```

Here, rotate (a,b,c,d) abbreviates a,b,c,d:=b,c,d,a.
The pair (p,q) traverses the same spanning tree as traversed by Mark (Root); p is always the head and q the tail, i.e. at each step q gets the value of p and p is advanced forwards. Boolean variable b indicates whether vector (p,q) is currently on its way <u>back</u> (along a path of the spanning tree already traversed before). In the first branch of the if-construct vector (p,q) is about to leave the tree, so vector (p,q) is reversed and b

is set to true. In the second branch of the if-construct another visit is payed to node p↑, b records whether this is the third visit, and the l- and r- fields are rotated as follows.



So after three visits (p,q) has entered both the left and the right subtree and has returned back, and the l- and r- fields are restored to their original values. As this happens for all nodes, the original graph structure has been restored upon termination! (In order to make this argument into a proof one should also show that vector (p,q) indeed returns to the node for a second and third visit. Notice also that there is no danger of nontermination due to cycles in the graph: as soon as a node has been visited (c ≠ 0) it will not be visited anymore except when (p,q) is on its way back (b=true).

3 Notation

Let us first remark that we everywhere use identifiers consisting of small letters for objects whose value depends on the state, like variables, whereas identifiers beginning with a capital letter denote fixed, mathematical, state independent values. For instance, p is a pointer variable and P is a pointer value, possibly the value of p in a particular state.

Next we remark that in dealing with proofs about programs involving pointers, there is one major notational problem, namely how to denote the storage area or the cells into which the pointers point. Indeed, something does change by executing p↑.c := 3 but it is certainly not p that gets the value 3. We need a name for the cell to which p points, or rather a name for the whole collection of such cells.

Definition. The storage area into which pointers point is called h (from: heap). Let P be a pointer, f some field name and V a value. Dereferencing P, i.e. P↑, is actually subscripting h with index P. So, using array notation, P↑ actually means h[P], P↑.f actually means h[P].f and P↑.f:= V actually means h[P].f := V.

Thus the little arrow ↑ in a program text may be viewed as a synonym for h, written after the index and without brackets rather than before it and with brackets.

Having now a name for the heap at hand, we may relate its initial and final value to each other by putting h=H in the precondition and h=H' in the postcondition; here H' should be a heap value constructed from H but with some specific changes. Our notation for selective updating of such values is as follows.

Definition. Let H, P, V be appropriate values and f a field name. Then

H[P:=V] is that value H' such that

H'[P'] = V for P'= P

= H'[P'] otherwise

Thus H[P:=V] is identical to H everywhere except at location P.

This notation is generalized to updating of fields:

H[P:(f)=V] is that value H' such that

H'[P'].f' = V for P'= P and f'= f

= H[P].f otherwise

and to several fields, for example

H[P:(c,l,r)=(0,Pl,Pr)]

and also to multiple updating, for example for a set R of pointers

H[R:(c)=3]

denotes the same heap value as H except that all nodes pointed to by members of R have been "marked".

Finally we need a notation for the "reachable" nodes, so that we can formulate the task of the algorithm properly.

Definition. Let H be some heap value, P a pointer value and R a set of pointer values. We set

$\text{reach}_H(P)$ = the set of pointers reachable in H, from P, via the l- and r-fields, along a path (beginning with P) all of which pnodes P' satisfy H[P'].c=0

$\text{reach}_H(P,R)$ = similarly but with the additonal requirement that the pnodes P' along the path satisfy $P' \notin R$.

Thus, setting $h=H \wedge R=\text{reach}_H(\text{Root})$ initially, the task of the algorithm is to establish $h=H[R:(c)=3]$ upon termination. In particular this means that for any P the final value of $P\uparrow.l$ equals the initial value, for we deduce from the postcondition $h=H[R:(c)=3]$ that upon termination

$P\uparrow.l = h[P].l = H[R:(c)=3]\ [P].l = H[P].l =$ the original value of $P\uparrow.l$

Similarly we deduce that the r-fields (and all other fields except the c-field) have their original value. So, the original structure of the heap is restored in spite of intermediate destructions. We consider the invention of "h" as a great advantage; it allows us to name all values of the storage area at once. In particular, by setting H=h initially, we have in H[P].l a notation for the initial value of $P\uparrow.l$, for any P! [Gries 1979] introduces a whole collection of constants for this purpose.

At last, we shall abbreviate the formulae slightly by writing P rather than {P} (the singleton set with sole member P).

We are now prepared to formulate and prove the correctness of the algorithms and of the repetition of SW in particular.

4 The correctness of Mark

Presumably no one would doubt the correctness of procedure Mark. We give an explicit proof in order to show that our proof of SW indeed closely parallels this one.

The theorem to be proved reads

(1) $\{h=H \wedge R=\text{reach}_H(P)\}$ Mark (P) $\{h=H[R:(c)=3]\}$

for arbitrary H, R and P. From now on we interpret a formula {A}stmnt{B} as expressing termination of stmnt (in a state satisfying B, provided the initial state satisfies A). So the claim of termination is included in (1). The theorem will be proved by induction on $|R|$, the number of elements in R, that is to say, we may use (1) as an induction hypothesis for recursive calls, with suitable choices for H, P and R provided the choice for R has a smaller cardinality than R itself. The proof is presented in the form of an annotation of its body; the justification of various steps is given afterwards.

Here is the annotated body.

```
{h=H ∧ R=$reach_H$(P)}
if P↑.c ≠ 0 -->
   {R=∅ so h=H=H[R:(c)=3]}
   skip
   {h=H[R:(c)=3]}
 [] P↑.c = 0 -->
 -- Let Pl=P↑.l=H[P].l and Pr=H[P].r                          (a)
 -- and Rl=$reach_H$(Pl,P) and Rr=$reach_H$(Pr, P ∪ Rl).       (b)
 -- Conclude R=P ∪ RL∪ Rr                                     (c)
 -- and P, Rl and Rr are disjoint                             (d)
 -- and, because P∉ Rl⊂R and P∉ Rr∪R, also
 -- that |Rl|<|R| and |Rr|<|R|.                               (e)
 {h = H[P:(c)=0]}
 P↑.c := P↑.c +1;
 {h = H[P:(c)=1]}
 Mark (P↑.l);
 {h = H[Rl:(c)= 3][P:(c)=1]}
 P↑.c := P↑.c + 1;
 {h = H[Rl:(c)=3][P:(c)=2]}
 Mark (P↑.r);
 {h = H[Rl:(c)=3][P:(c)=2]}
 P↑.c := P↑.c + 1;
 {h = H[Rl:(c)=3][Rr:(c)=3][P:(c)=2]}
 {h = H[Rl Rr P:(c)=3]}
 {h = H[R:(c)=3]}
fi
{h = H[R:(c)=3]}
```

We now justify various "steps" in the proof. It is simple to verify each step around $P\uparrow.c:=P\uparrow.c+1$ separately, but we could also use the following (rather trivial) lemma three times.

<u>lemma 1</u> Let H, P and i be arbitrary. Then
$\{h = H[P:(c)=i]\}\ P\uparrow.c:=P\uparrow.c+1\ \{h = H[P:(c)=i+1]\}$

The steps around the recursive calls really need some justification. Consider the first call. In its precondition we should prove that $Rl = reach_H(Pl,P) = reach_{H}(Pl,P)$ where $H' = H[P:(c)=1]$ = the current value of h. The induction hypothesis can then be applied, because of conclusion (e). The postcondition should actually read $h = H[P:(c)=1][Rl:(c)=3]$, but one easily verifies that disjoint updatings commute. A similar reasoning applies to the second call. At the last three assertions, just before <u>fi</u>, commutativety of updatings is used again. Finally, one should prove (or already be convinced of) conclusions (c) and (d). We omit these proofs; they are of a pure mathematical, graph theoretic nature and any mathematician should be able to provide the details. Anyway, [Topor 1979] gives a very precise treatment of these and similar properties.

(It is only for completeness' sake that we list here some general properties from which the above ones are easily dirived. Let H, P, Pl, Pr, R and R' be arbitrary. Then
(i) R and P disjoint ==>
$H[R:(c)=3]P:---] = H[P:---][R:(c)=3]$ and
$H[R:(c)=3][R':(c)=3] = H[R \cup R':(c)=3]$.
(ii) $R=P \cup Rl \cup Rr$ where $Rl = reach_H(Pl)$ and $Rr = reach_H(Pr)$ <==>
$R=P \cup Rl' \cup Rr'$ where $Rl = reach_H(Pl,P)$ and $Rr = reach_H(Pr,P \cup Rl)$.
(iii) $reach_H(P,R) = reach_{H'}(P,R)$ where $H'=H[R:...]$.
)

<u>5 The correctness of SW</u>

Although the net effect of SW and Mark are the same, we cannot expect the assertions in the proof the same, because SW uses p, q, b, VirtualRoot and the l- and r-fields for house-keeping purposes. However, apart from simple additions for these variables and values, the assertions are really the same.

First we give a lemma that corresponds to lemma 1 in the proof of Mark; the lemma gives a general form of the net effect of one visit to a node, i.e. one increment of a c-field. The function rot rotates a 3-tuple:
rot (A, B, C) = (B, C, A)
and for any 3-tuple X we let X_i denote the i-th component of X, but for typographical reasons we write Xi instead of X_i.

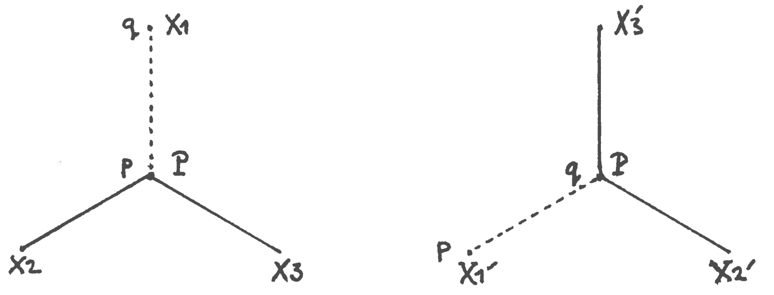
<u>Lemma 1'</u>

$\{h = H[P:(c,l,r)=(i,X2,X3)] \wedge p,q,b=P,X1,F \text{ where } X=rot^i(Y)\}$

$p\uparrow.c:= p\uparrow.c+1;\ b:= p\uparrow.c=3;$ rotate $(q, p, p\uparrow.l, p\uparrow.r)$

$\{h = H[P:(c,l,r)=(i+1,X2,X3)] \wedge q,p,b=P,X1,(i=2) \text{ where } X=rot^{i+1}(Y)\}$

for arbitrary H, P, i and 3-tuple Y. Pictorially, a part of the pre- and postcondition looks like



pre; $X = rot^i(Y)$ post; $X' = rot^{i+1}(Y)$

Proof

One may easily prove this by whatever method one is used to. After all, there are just three (multiple) assignments involved. It is obvious that q and b get the values P and (i=2). Further, one easily sees that

p gets the value of p↑.l, i.e. the 2nd of $rot^i(Y)$,
p↑.l gets the value of p↑.r, i.e. the 3rd of $rot^i(Y)$,
p↑.r gets the value of q, i.e. the 1st of $rot^i(Y)$.

Hence (p, p↑.l, p↑.r) gets the value $rot^{i+1}(Y)$.
(End of proof)

Next we consider the repetition. We claim that some consecutive number of iterations have the same net effect as one call of procedure Mark. This cannot be formulated in the format {A}repetition{A'} because this would mean that it is only after termination (of all iterations) that A' holds and this is not what we intend to say (and it is not true either). To express our claim we use the format

L: A leads in SW to L: A'

meaning that if control is at program point L (about to test the do-guard) and A holds true in that state, then sometime later (after a suitable number of iterations) control will be at L again (about to test the do-guard), this time A' being satisfied. L: A is called the initial configuration of that formula and L: A' is the final configuration.

The theorem to be proved reads

(1') L: $h=H \wedge R=reach_H(P) \wedge$ p,q,b=P,Q,F
leads in SW to
L: $h=H[R:(c)=3] \wedge$ p,q,b=Q,P,T

for arbitrary H, P, Q and R satisfying $V \notin R \cup P$. (V is an abbreviation for VirtualRoot). Compare this with (1): the only difference is the additional conjunct about the additional variables p, q and b. The theorem is proved by induction on $|R|$; so we may use (1') as an induction hypothesis with suitable choices for H, P, Q and R provided the choice for R has a smaller cardinality than R itself. The proof is presented as a sequence of configurations, each one leading in SW to the next one; the justification of various steps is given afterwards.

Here is the proof.

```
L: $h=H \wedge R=\mathrm{reach}_H(P) \wedge p,q,b=P,Q,F$
if $P\uparrow.c \neq 0$ then
   --$R=\emptyset$ so $h = H = H[R:(c)=3]$
   --the do-guard and 1st if-guard are passed
   L: $h=H[R:(c)=3] \wedge p,q,b=Q,P,T$
if $P\uparrow.c=0$ then
   --Let $Pl=P\uparrow.l=H[P].l$ and $Pr=H[P].r$
   --and $Rl=\mathrm{reach}_H(Pl,P)$ and $Rr=\mathrm{reach}_H(Pr,Rl \cup P)$ and $Y=(Q,Pl,Pr)$.
   --Conclude $R=P \cup Rl \cup Rr$, P and Rl and Rl are
   --disjoint, $|Rl|<|R|$ and $|Rr|<|R|$, and $V \notin Rl \cup P$
   --and $V \notin Rr \cup P$.
   L: $h=H[P:(c,l,r)=(0,X2,X3)] \wedge p,q,b=P,X1,F$ where $X=rot^0(Y)$
   --the do- and 2nd if-guard are passed, apply lemma 1'
   L: $h=H[P:(c,l,r)=(1,X2,X3)] \wedge q,p,b=P,X1,F$ where $X=rot^1(Y)$
   --apply induction hypothesis(1')
   L: $h=H[Rl:(c)=3][P:(c,l,r)=(1,X2,X3)] \wedge p,q,b=P,X1,T$ where $X=rot^1(Y)$
   --the do- and 2nd if-guard are passed, apply lemma 1'
   L: $h=H[Rl:(c)=3][P:(c,l,r)=(2,X2,X3)] \wedge q,p,b=P,X1,T$ where $X=rot^2(Y)$
   -- apply induction hypothesis (1')
   L: $h=H[Rl:(c)=3][Rr:(c)=3][P:(c,l,r)=(2,X2,X3)] \wedge p,q,b=P,X1,T$ where
     $X=rot^3(Y)$
   --the do- and 2nd if-guard are passed, apply lemma 1'
   L: $h=H[Rl:(c)=3][Rr:(c)=3][P:(c,l,r)=(3,X2,X3)] \wedge q,p,b=P,X1,T$ where
     $X=rot^3(Y)$
   i.e.: $h=H[Rl \cup Rr \cup P:(c)=3] \wedge q,p,b=P,Q,T$
   i.e.: $h=H[R:(c)=3] \wedge q,p,b=P,Q,T$
fi, so in both cases
L: $h=H[R:(c)=3] \wedge q,p,b=P,Q,T$
```

We can be very brief with respect to the justification of the various steps in this proof: they are exactly the same as in the proof of procedure Mark, and are therefore omitted here.

It is now easy to prove the correctness of the algorithm. Let the initial state satisfy $h=H \wedge R=reach_H(Root)$ and notice that we have already assumed $V \notin R \cup Root$. The initialisation then leads to
   L: $h=H \quad R=reach_H(Root) \wedge p,q,b=Root,V,F$
so that we may apply theorem (1') and infer that this configuration leads to
   L: $h=H[R:(c)=3] \wedge p,q,b=V,Root,T$
so that the do-guard cannot be passed and the required postassertion has been established. This reasoning corresponds to the correctness proof of the main call Mark (Root) which we ought to have given in the previous section as well.

6 Discussion

It is clear that our proof of SW very closely resembles the proof of Mark; not only in the structure of the proof but also in the assertions used. Assuming that the proof of Mark is in some sense the best of most economic proof and noting that for a proof of SW we have at the least to say something in the assertions about p, q, b and the current l- and r-field, we might conclude that we also have the best or most economic proof of SW!

Let us compare the structures of the proofs and the programs. The structure of both proofs coincides with the structure of the data to be manipulated. In particular, for the data we have the fact (suppressing H everywhere) that

$$reach\ (P) = \emptyset \qquad \text{if } P\uparrow.c \neq 0$$

$$reach\ (P) = P \cup Rl \cup Rr \qquad \text{if } P\uparrow.c = 0$$

$$\text{where } Rl = reach\ (Pl,\ P)$$

$$Rr = reach\ (Pr,\ P \cup Rl)$$

$$Pl,\ Pr = P\uparrow.l,\ P\uparrow.r$$

In both proofs the two applications of the induction hypothesis correspond to the treatment of Rl and Rr respectively. In procedure Mark this structure is also present in its syntactic form. The syntactic structure of SW, however, deviates from this one: it is not at all clear at first sight what one step of the repetition corresponds to in terms of the data. Of course, it is possible to define a corresponding structuring of the data. [Gries 1979] performs this excercise. He introduces an auxiliary notion of stack, encoded in a clever way in the currently modified heap. Further he uses invariant assertions which, apart from necessitating natural language (in order to be comprehensible by humans), take several lines to write down and require extension case-analysis for their verification.

The reason of the large difference between our proof (or that of [Topor 1979]) and axiomatic proofs (like that of [Gries 1979]) may be explained as follows. What we are after when proving a program correct, is proving a relation between the initial and final configuration of the computation (= sequence of configurations) evoked by the program text. This is done by induction: the computation is split into subcomputations and the theorem is generalized to a relaton between the initial and final configuration of any subcomputaton. In our proof the division of the computation corresponds to the structure of the data. In an axiomatic proof it is the structure of the program text that determines the division of the computation: the computation evoked by a repetition is divided into parts, each part being the computaton evoked by one step of the repetition. The invariant assertion relates the initial and final configurations of those subcomputations to each other. More often than not a program is constructed so that its syntactic structure correspond to the structure of the data. In those cases there is no difference between the data directed proof method and the syntax directed, axiomatic method. However, when a program is obtained from another by several semantic preserving transformations, then the data directedness may got lost underway. This is indeed the case for the Schorr-Waite algorithm; [Jonkers 1982] formally derives the Schorr-Waite algorithm from its recursive counterpart.

References

de Roever, W.P.: On backtracking and greatest fixpoints. In Formal Descriptions of Programming concepts, (ed E.J. Neuhold), North-Holland Publ Co, Amsterdam, 1978, pp 621-636.

Gries, D.: The Schorr-Waite graph marking algorithm. Acta Informatica 11, (1979), pp 223-232.

Hoare, C.A.R.: An axiomatic basis for computer programming. Comm ACM 12 (1969) pp 576-580.

Jonkers, H.B.M.: Abstraction, Specification and Implementation Techniques. Doctoral Dissertation, Amsterdam, 1982.

Kowaltowski, T.: Data structures and correctness of programs. Journal of the ACM 26 (1979), pp 283-301.

Manna Z.: Is 'Sometime' sometimes Better than 'Always'? Comm ACM 21 (1978), pp 159-172.

Morris, J.M.: A proof of the Schorr-Waite Algorithm. In Theoretical Foundations of Programming Methodology (Eds M. Broy and G. Schmidt). D. Reidel Publ Co, 1982, pp 43-41.

Topor, R.W.: The correctness of the Schorr-Waite list marking algorithm. Acta Informatica 11 (1979) pp 211-221.