Examples of Informal but Rigorous
Correctness
Proofs for Tree Traversing Algorithms

Tomasz Kowaltowski

Relatório Técnico DCC–10/92

Novembro de 1992
Examples of Informal but Rigorous Correctness
Proofs for Tree Traversing Algorithms

Tomasz Kowaltowski

Abstract

Correctness of several tree traversing algorithms is proved in an informal but quite rigorous way by using induction and a convenient graphical representation for the state of computation. These proofs are much simpler than their formal counterparts and provide an intuitive insight for the ideas behind the algorithms.

A preliminary version of this paper was written in 1986 while the author was on leave visiting the School of Information and Computer Science of the Georgia Institute of Technology. Partial support was provided by the grant 85/155-7 from the Fundação de Amparo de Pesquisa do Estado de São Paulo (FAPESP).

Author's Address: Tomasz Kowaltowski, Department of Computer Science, University of Campinas, Caixa Postal 6065, 13081-970 Campinas, SP, Brazil.

E-mail: tomasz@dcc.unicamp.br.
1 Introduction

Formal proofs of correctness of algorithms manipulating data structures can become quite lengthy and unreadable, even though the basic ideas behind those proofs may be simple and intuitive. For examples of such proofs see for instance Burstall [1], Gries [3], Kowalowski [5] and Topor [10]. In many cases it is possible however to provide simple and informal but quite rigorous proofs by using a convenient graphical representation of the state of computation. This is particularly true in the case of tree manipulating algorithms where mathematical induction on the structure of the trees is often applied.

We illustrate these ideas through a series of examples of algorithms for traversing binary trees in preorder, inorder and/or postorder as defined by Knuth [4]. For the sake of uniformity we present all these algorithms as Pascal procedures, and the type declarations of Figure 1 are assumed to be global throughout the paper. The state of computation is described by drawings where triangles represent (possibly empty) binary trees, rectangles represent explicit nodes, and separate triangles and rectangles denote disjoint parts of the data structures. Dark triangles denote trees which have already been traversed in the specified order (or orders). Inductive assertions are referred to as comments within the procedures by their names $Q_i$ or $R_i$, and it should be understood that whenever we state that an assertion holds, it implies that it holds at the corresponding point within the procedure. Additional conventions will be introduced with specific examples. Proofs are carried out by representing the inductive hypothesis, supplemented with some informal explanation, and then by showing how to carry out the inductive step.

It is interesting to notice that these proofs, besides showing correctness, seem to be very helpful in understanding the ideas behind the algorithms and thus make it easier to modify them and to adapt them to particular situations.
type
  ptr = |node;
node = record
  info: SomeType;
  tag: boolean; { for the DSW procedure only }
  llink, rlink: ptr
end;

Figure 1: Type declarations for all the procedures

procedure Recursive(p: ptr);
begin
  {Q1}
  if p/=nil then
    with p do begin
      PreOrderVisit(p); {R1}
      Recursive(llink);
      InOrderVisit(p); {R2}
      Recursive(rlink);
      PostOrderVisit(p)
    end
  {Q2}
end;

Figure 2: Recursive traversal procedure

2 Recursive tree traversal

The procedure shown in Figure 2 certainly does not require any proof, but it is included here as a convenient introductory example. Figure 3 shows the inductive hypothesis which should be read as: if the assertion Q1 holds then after a finite number of steps the procedure will reach its end in a state satisfying the assertion Q2. It is also implicit in our hypothesis that the final tree is identical to the original one. This fact is obvious in this case since the procedure does not modify the tree, but will have to be shown in some later examples.

The hypothesis is trivially true when p = nil. Figure 4 shows the
proof steps when $p \neq \text{nil}$ and is obtained by following the execution of the procedure and applying the inductive hypothesis to the two subtrees of $p$; $R_1$ and $R_2$ are auxiliary assertions. Notice that $R_1$ and $R_2$ imply $Q_1$ for the trees $p_1$ and $p_2$ respectively. Digits 1, 2 and 3 over the node $p$ show that the node has been already visited in preorder, inorder and postorder; an arrow marked with ‘Ind. Hyp.’ denotes an application of the inductive hypothesis. We shall use these conventions throughout this paper.

We notice finally that the inductive hypothesis implies the correctness of the procedure when applied to the initial value of $p$.

### 3 Traversals with an explicit stack

Figure 5 shows a procedure for preorder traversal, and Figure 6 shows the corresponding inductive hypothesis which should be read as: if the assertion $Q_1$ holds, then after a finite number of steps the assertion $Q_2$ will hold. It is implicit also that the stack contains exactly the same values at the points corresponding to $Q_1$ and $Q_2$, and that the value of the variable $\text{done}$ remains false.

The proof for $p = \text{nil}$ is again trivial and the inductive step for $p \neq \text{nil}$ is covered in Figure 7 where we assume that $p_1.rlink \neq \text{nil}$; the other case is even simpler. It should be noticed that $R_1$ implies $Q_1$ for $p = p_1$ and $R_2$ implies $Q_1$ for $p = p_2$. It is easy to see that the initial state of the procedure with an empty stack satisfies $Q_1$, and that by applying the inductive hypothesis we get the corresponding $Q_2$, showing
Figure 4: Proof of the recursive procedure
procedure PreOrder\((p: \text{ptr})\);

var done: boolean;

begin
  InitializeStack; done := false;
  repeat
    \(\{Q_1\}\)
    if \(p \neq \text{nil}\) then
      with \(p \uparrow\) do begin
        PreOrderVisit\((p)\);
        if rlink \neq \text{nil} then Push\((rlink)\);
        p := llink \{R_1\}
      end
    else
      \(\{Q_2\}\)
      if EmptyStack then done := true
      else Pop\((p)\)
    \{R_2\}
  until done
end;

Figure 5: Preorder traversal with an explicit stack

\[
\begin{array}{c}
\text{Figure 6: Inductive hypothesis for the preorder procedure} \\
\end{array}
\]
Figure 7: Proof of the preorder procedure

correctness of the procedure.

Figures 8 through 10 show the procedure for postorder traversal and its proof which is similar. Notice that the procedure discovers whether it should go up or down to the right subtree by comparing the right link of the current node with the pointer last to the most recently visited node. In this way it avoids pushing on the stack an extra tag together with the pointer to the current node. An almost identical procedure and its proof for the inorder are left to the reader.
procedure PostOrder(p: ptr);
var
    up: boolean;
    last: ptr;
begin
    InitializeStack;
repeat
    while {Q₁} p ≠ nil do begin {go left}
        Push(p);
        p := p\.llink {R₁}
    end;
    up := true;
    last := nil;
    while {Q₂} up and (not EmptyStack) do begin
        Pop(p);
        if p\.rlink ≠ last then begin {go right}
            Push(p);
            p := p\.rlink;
            up := false {R₂}
        end
        else begin {go up}
            PostOrderVisit(p);
            last := p
        end
    end
until EmptyStack
end;

Figure 8: Postorder traversal with an explicit stack
Figure 9: Inductive hypothesis for the postorder procedure

Figure 10: Proof of the postorder procedure
4 Deutsch-Schorr-Waite algorithm

The procedure shown in Figure 11 implements the algorithm which according to Knuth [4] was discovered independently by Deutsch and by Schorr and Waite [9]; see also Gries [3] and Topor [10]. It can perform any of the three traversals and it uses an explicit stack which is kept in the tree itself by reversing the pointers on the path from the root to the current node, and restoring them on the way back up the tree. Each node on this stack has its tag field set conveniently in order for the procedure to know whether the left or right link was reversed. The algorithm is really more general and was originally formulated for traversing arbitrary data structures and not only trees.

Figure 12 shows the inductive hypothesis. The upside down triangle denotes the tree formed by the stack nodes and their subtrees (as the proof shows it is a tree). The hypothesis also states implicitly that the trees pointed to by \( p \) and \( t \) remain identical to the original ones, except possibly for the tag fields in the tree \( p \). Figure 13 shows again the proof steps when \( p \neq \text{nil} \). Notice that in this case it is also proved that the original trees are restored, and that the tag fields in the tree \( t \) (denoted by \( T \) for \( \text{true} \) and by \( F \) for \( \text{false} \)) do not change. It is obvious that the inductive hypothesis implies the correctness of the procedure since initially \( t = \text{nil} \).
procedure DSW (p: ptr);
  var
    t, q: ptr; up: boolean;
  begin
    t := nil;
  repeat
    while \{Q_1\} p \neq nil do \{ go left \}
      with p do begin
        PreOrderVisit (p); tag := true;
        q := llink; llink := t;
        t := p; p := q \{R_1\}
      end
    up := true;
    while \{Q_2\} up and (t \neq \text{nil}) do
      with t do begin
        case tag of
          true: begin \{ go right \}
            InOrderVisit (t); up := false;
            tag := false; q := p; p := rlink;
            rlink := llink; llink := q \{R_2\}
          end;
          false: begin \{ go up \}
            PostOrderVisit (t);
            q := rlink; rlink := p;
            p := t; t := q
          end
        end
      until t = \text{nil}
  end;
Figure 11: Deutsch-Schorr-Waite procedure
Figure 12: Inductive hypothesis for the DSW procedure

Figure 13: Proof of the DSW procedure
procedure $LD(p: \text{ptr})$;

var
  empty, q, t: \text{ptr};

begin
  new(empty); t := empty;
  repeat
    \{$Q_1$\}
    if $p$ = nil
      then begin \{ swap $p$ and $t$ \}
        $p$ := $t$;
        $t$ := nil
      end
    else begin \{ rotate four pointers \}
      Visit($p$);
      $q$ := $t$; $t$ := $p$;
      $p$ := $p$.$\text{link}$;
      $t$.$\text{link}$ := $t$.$r\text{link}$;
      $t$.$r\text{link}$ := $q$
    end
  \{$Q_2$\}
  until $p$ = empty
end;

Figure 14: Lindstrom-Dwyer procedure

5 Lindstrom-Dwyer algorithm

The procedure of Figure 14 implements the algorithm discovered by Lindstrom [6] and Dwyer [2] which is similar to that described in the previous section, but does not require the extra tag field in each node. The price to be paid is that the algorithm is “blind”, i.e., does not produce one of the canonical traversals but a “merge” of the three: each node is visited three times and the algorithm cannot distinguish between the three visits. Instead of reversing alternately the two link fields, a rotation of four values is performed: the two link fields, the current node pointer and the previous node pointer (the stack pointer).

The procedure must distinguish however between the three visits
to the root node in order to terminate properly. Instead of using a small counter, we represent the empty stack by a pointer value which is different from any pointer in the tree (we call it empty). The small black dot denotes the occurrences of this pointer which actually corresponds to an empty subtree. The proof is similar to that of the previous section.

6 Robson’s algorithm

The procedure of Figure 17 implements the algorithm devised by Robson [8] which is another clever modification of the Deutsch-Schorr-Waite algorithm of Section 4. Instead of using the extra tag in each node to know whether the left or right link field was reversed, the algorithm keeps an additional stack (besides the one formed by the reversed pointers). This stack does not require any additional space since it is kept temporarily in the leaves of the tree. The leaves are linked through their \textit{rlink} fields, and the \textit{llink} fields are used to keep pointers to the nodes whose two subtrees are not empty and whose right links have been reversed. It is not necessary to store the pointers in other cases. The variable \textit{r} points to the topmost value of this stack (i.e. to the last node whose right link was reversed, if any; otherwise it is nil), and the variable \textit{top} points to the rest of the stack. The variable \textit{av} points to the next leaf which can be used to extend the stack. The variable empty is used in the same way as in the Lindstrom-Dwyer procedure.
Figure 16: Proof of the LD procedure
procedure Robson (p: ptr);

var
  r, q, t, top, av, empty: ptr;
  up: boolean;

begin
  r := nil; top := nil; up := false;
  new (empty); t := empty;
repeat
  repeat {Q₁} { going down }
    PreOrderVisit (p);
  if (p.link = nil) and (p.rlink = nil)
    then begin { leaf: go up }
      InOrderVisit (p);
      up := true
    end
  else if p.link = nil then begin { go right }
    InOrderVisit (p);
    q := p.rlink;
    p.rlink := t;
    t := p; p := q {R₁}
  end
  else begin { go left }
    q := p.link;
    p.link := t;
    t := p; p := q {R₂}
  end
until up;
  av := p;

Figure 17: Robson’s procedure (continues)
repeat \{Q_2\} \{ going up \}
PostOrderVisit(p);
if \(t=\text{empty}\) then up := false \{ finished \}
else with \(t\) do
    if \(l\link=\text{nil}\) then begin \{ coming from right \}
        \(q := r\link; r\link := p;\)
        \(p := t; t := q\)
    end
else
    if \(r\link=\text{nil}\) then begin \{ coming from left \}
        \(q := l\link; l\link := p;\)
        \(p := t; t := q;\)
        InOrderVisit(p);
    end
else
    if \(t=r\) then begin \{ coming from right \}
        \(q := t\top; r := q.l\link; t\top := q.r\link;\) \{ pop \}
        \(q.l\link := \text{nil};\)
        \(q.r\link := \text{nil};\)
        \(q := r\link; r\link := p;\)
        \(p := t; t := q\)
    end
else begin \{ coming from left \}
        \(a\top.l\link := r; a\top.r\link := t\top; t\top := a\top;\) \{ push \}
        \(r := t; q := l\link;\)
        \(l\link := p; p := r\link;\)
        \(r\link := q; up := false;\)
        InOrderVisit(t) \{ R_3 \}
    end
until not up
until \(t=\text{empty}\)
end;

Figure 17: Robson’s procedure (cont.)
Figure 18: Inductive hypothesis for the Robson procedure

Figure 19: Proof of the Robson procedure – part I

Figure 18 shows the inductive hypothesis for this procedure and should be read in the usual way. Some facts implicit in this hypothesis are: (a) the trees in $Q_1$ and $Q_2$ are exactly the same; (b) the values of the variables $r$ and $top$ remain the same; (c) the stack formed by the leaves represented on top of the upside down triangles remains the same; (d) the new leaf found by the algorithm and pointed to by $av$ is contained in the subtree $p$.

The procedure assumes that the initial tree is not empty and the proof can be carried out by considering the four possible cases for the values of $p$.l.link and $p$.r.link. We show in Figures 19 and 20 only the cases when both subtrees of $p$ are empty (when a new leaf is found) or when both trees are not empty (when the stack is extended). Other cases
Figure 20: Proof of the Robson procedure – part II (continues)
Figure 20: Proof of the Robson procedure – part II (cont.)
are similar.

Notice that the proof also implies that the reversed pointers are restored to their original values, that the link fields of the leaves used to keep the stack are restored to \texttt{nil} and that a new leaf is always available when it is necessary to extend the stack. By applying the inductive hypothesis to the initial state when the variables \( r \) and \( top \) are \texttt{nil}, \( r \) is empty and \( up \) is false, and following the execution from \( Q_2 \) (when \( PostOrderVisit(p) \) is executed), we conclude that the whole tree is traversed and the procedure ends.

7 Morris's algorithm

Figure 21 shows yet another procedure for traversing binary trees in preorder and/or inorder. The algorithm devised by Morris [7] creates temporary right threads as in threaded binary trees (see for instance Knuth [4]). Whenever a non-empty tree is about to be traversed and its left subtree is empty, then the root is visited and the procedure advances to the right subtree. If the left subtree is not empty then the procedure searches for this subtree's last node in inorder, and sets its right thread, i.e. replaces a \texttt{nil} by a pointer to the current root. Subsequently the traversal proceeds to the left subtree. Due to the existence of the thread, \( p \) will end up pointing to the same initial node after the traversal of the left subtree is completed. At this point, the thread is eliminated and the \texttt{nil} pointer restored. After visiting the current root, the procedure moves to the right subtree.

Figure 22 shows the inductive hypothesis which covers two possible cases: (a) if at \( Q_1 \) \( p \) points to a tree whose last inorder node does not contain a thread (i.e. its right link field is \texttt{nil}) then the procedure will end up at \( Q_2 \) (at the same place in the program) with the tree traversed in preorder and/or inorder, and \( p = \texttt{nil} \); or (b) if at \( Q_1 \) \( p \) points to a tree whose last inorder node contains a thread then the procedure ends up at \( Q_2 \) with the tree traversed in preorder and/or inorder with \( p \) at the node pointed to by the thread. It should be noticed that the inductive hypothesis also implies that at \( Q_1 \) there is at most one thread within the
procedure Morris(p: ptr);
var q: ptr;
begin
while \(Q_1\) \(\&\) \(Q_2\) \(p\neq nil\) do
  with \(p\) do
    if \(l\_link=\text{nil}\) then
      \begin{align*}
      \text{begin } \{ \text{empty left } \text{ go right } \}\end{align*}
      PreOrderVisit(p);
      InOrderVisit(p);
      \(p:=r\_link\ \{R_1\}\end{align*}
    \text{end}
  else
    \begin{align*}
    q:=l\_link;
    \text{while } (q\.r\_link\neq\text{nil}) \text{ and } (q\.r\_link\neq p) \text{ do}
    q:=q\.r\_link;
    \text{if } q\.r\_link=\text{nil} \text{ then}
    \begin{align*}
    \text{begin } \{ \text{first time at } p \text{ go down } \}\end{align*}
    PreOrderVisit(p);
    q\.r\_link:=p; \(p:=l\_link\ \{R_2\}\end{align*}
    \text{end}
    else
    \begin{align*}
    \text{begin } \{ \text{second time at } p \text{ go right } \}\end{align*}
    q\.r\_link:=\text{nil};
    InOrderVisit(p);
    \(p:=r\_link\ \{R_3\}\end{align*}
    \text{end}
  \end{align*}
\text{end};

Figure 21: Morris’s procedure
tree $p$ (shown explicitly), and that after the traversal all fields within the data structure are the same as at $Q_1$. A bold arrow denotes a (possibly empty) chain of right links. Figures 23 and 24 show the proof steps for the cases when the left subtree is empty or not. The case $p = \text{nil}$ is trivial. As before it is easy to see that the inductive hypothesis implies correctness of the procedure.

The algorithm can be extended to traverse the tree in postorder as suggested in [7]; the proof is left to the reader.
Figure 23: Proof of the Morris’s procedure – part I
\[ Q_1 \quad \equiv \quad \begin{array}{l}
\text{Ind. Hyp.} \\
\text{and} \\
p \neq \text{nil} \\
\text{and} \\
p \text{.link} \neq \text{nil}
\end{array} \]

\[ R_2 \quad \rightarrow \quad \rightarrow \quad \text{Ind. Hyp.} \]

\[ Q_2 \quad \equiv \]

Figure 24: Proof of the Morris's procedure – part II
References


Relatórios Técnicos

01/92 Applications of Finite Automata Representing Large Vocabularies, C. L. Lucchesi, T. Kowalowski
02/92 Point Set Pattern Matching in d-Dimensions, P. J. de Rezende, D. T. Lee
03/92 On the Irrelevance of Edge Orientations on the Acyclic Directed Two Disjoint Paths Problem, C. L. Lucchesi, M. C. M. T. Giglio
04/92 A Note on Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams, W. Jacometti
05/92 An \((l, u)\)-Transversal Theorem for Bipartite Graphs, C. L. Lucchesi, D. H. Younger
06/92 Implementing Integrity Control in Active Databases, C. B. Medeiros, M. J. Andrade
07/92 New Experimental Results For Bipartite Matching, J. C. Setubal
08/92 Maintaining Integrity Constraints across Versions in a Database, C. B. Medeiros
09/92 On Clique-Complete Graphs, C. L. Lucchesi, C. P. Mello, J. Szwarcfiter

Departamento de Ciência da Computação — IMECC
Caixa Postal 6065
Universidade Estadual de Campinas
13081-970 — Campinas — SP
BRASIL
reltec@dcc.unicamp.br