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], Kowaltowski [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
    $ptr = \uparrow node$;
    $node$ = record
        $info$: $SomeType$;
        $tag$: $boolean$; { for the DSW procedure only }
        $llink, rlink$: $ptr$
        end;
```

Figure 1: Type declarations for all the procedures

```
procedure $Recursive(p\colon ptr)$;
begin
{$Q_1$}
if $p \neq$ nil then
    with $p\uparrow$ do begin
        $PreOrderVisit(p)$; {$R_1$}
        $Recursive(llink)$;
        $InOrderVisit(p)$; {$R_2$}
        $Recursive(rlink)$;
        $PostOrderVisit(p)$
        end
{$Q_2$}
end;
```

Figure 2: Recursive traversal procedure

2 Recursive tree traversal

The procedure shown in Figure 2 cartainly 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 Q_1 holds then after a finite number of steps the procedure will reach its end in a state satisfying the assertion Q_2. 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

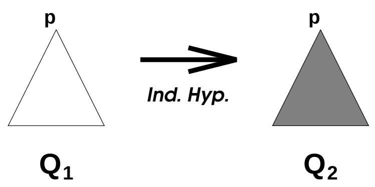

Figure 3: Inductive hypothesis for the recursive procedure

proof steps when $p \neq$ **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 *done* remains *false*.

The proof for $p =$ **nil** is again trivial and the inductive step for $p \neq$ **nil** is covered in Figure 7 where we assume that $p\uparrow.rlink \neq$ **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

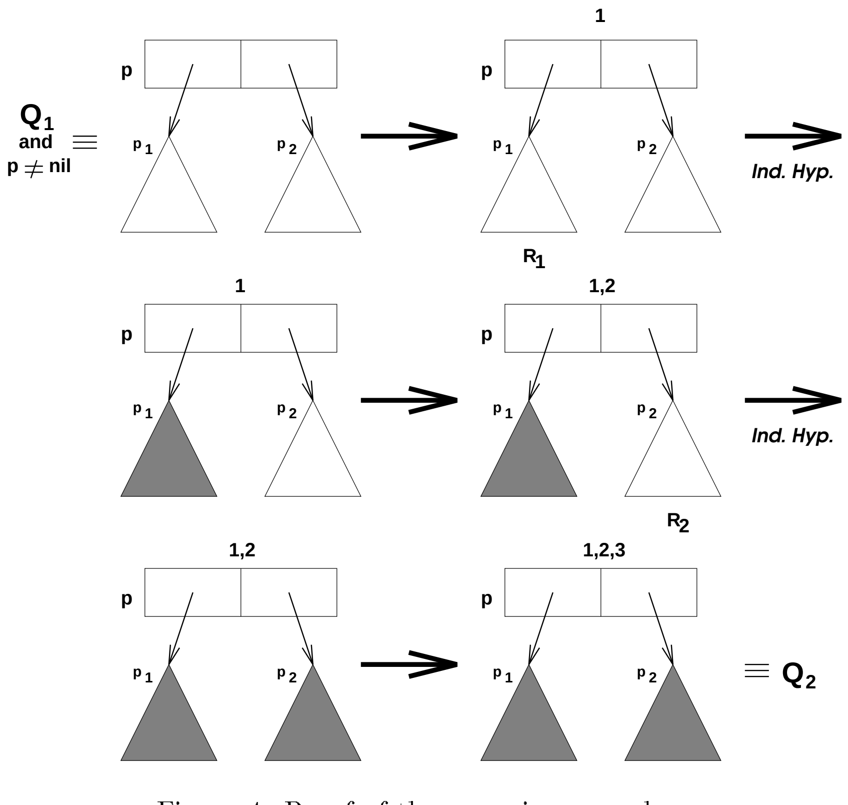

Figure 4: Proof of the recursive procedure

```
procedure $PreOrder(p{:}\ ptr)$;
  var $done$: $boolean$;
begin
$InitializeStack$; $done := false$;
repeat
$\{Q_1\}$
    if $p \neq$ nil
        then
          with $p\uparrow$ do begin
            $PreOrderVisit(p)$;
            if $rlink \neq$ 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

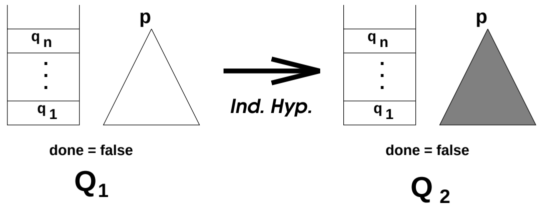

Figure 6: Inductive hypothesis for the preorder procedure

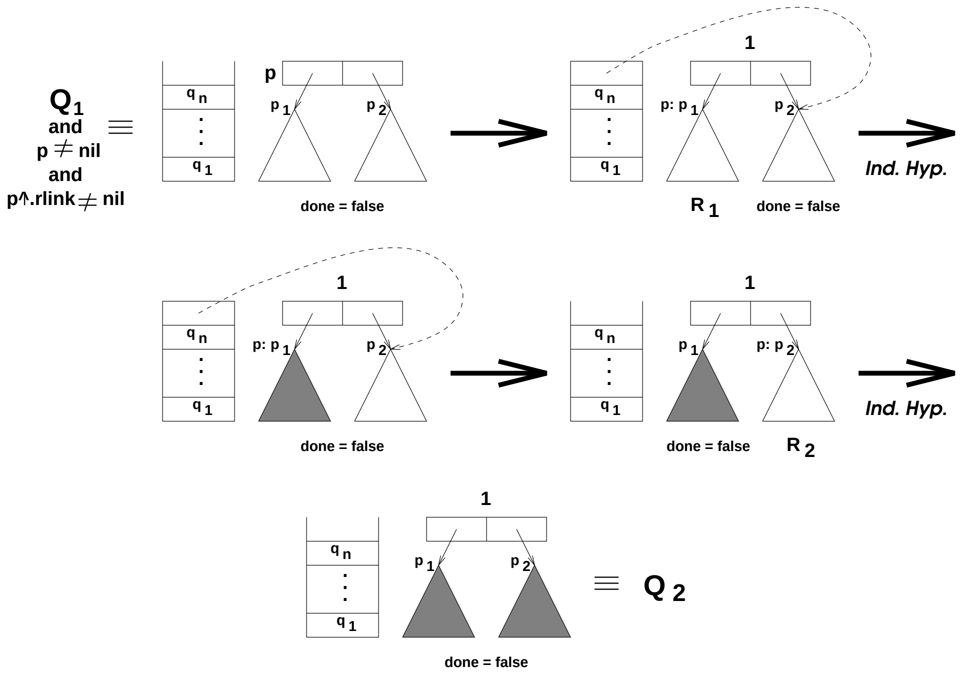

Figure 7: Proof of the preorder procedure

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 $PostOrder(p\colon ptr)$;
    var
        $up\colon boolean$;
        $last\colon ptr$;
begin
$InitializeStack$;
repeat
    while $\{Q_1\}$ $p \neq$ nil do begin { go left }
        $Push(p)$;
        $p := p\uparrow.llink$ $\{R_1\}$
        end;
    $up := true$;
    $last :=$ nil;
    while $\{Q_2\}$ $up$ and (not $EmptyStack$) do
        begin
        $Pop(p)$;
        if $p\uparrow.rlink \neq last$
            then begin { go right }
                $Push(p)$;
                $p := p\uparrow.rlink$;
                $up := false$ $\{R_2\}$
                end
            else begin { go up }
                $PostOrderVisit(p)$;
                $last := p$
                end
        end
until $EmptyStack$
end;
```

Figure 8: Postorder traversal with an explicit stack

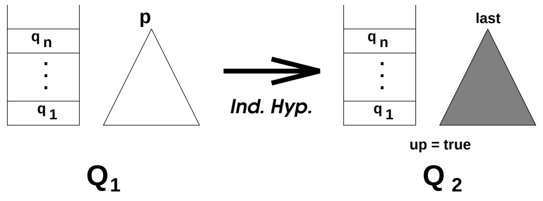

Figure 9: Inductive hypothesis for the postorder procedure

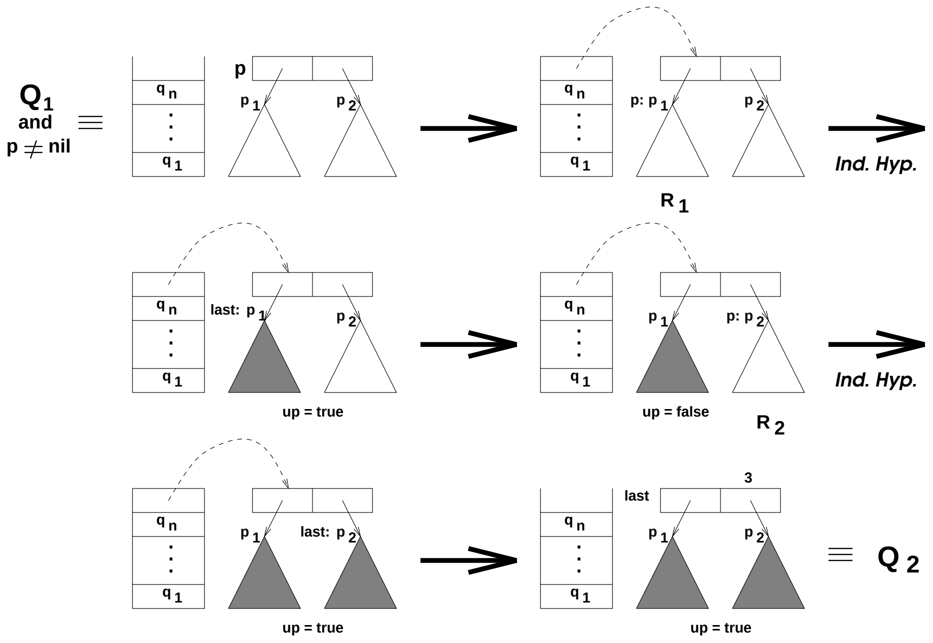

Figure 10: Proof of the postorder procedure

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 $p \neq \textbf{nil}$. 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 *true* and by `F` for *false*) do not change. It is obvious that the inductive hypothesis implies the correctness of the procedure since initially $t = \textbf{nil}$.

```
procedure $DSW(p\colon ptr)$;
    var
        $t, q\colon ptr$; $up\colon boolean$;
begin
$t$ := nil;
repeat
    while $\{Q_1\}$ $p \neq$ nil do { go left }
        with $p\uparrow$ do begin
            $PreOrderVisit(p)$; $tag := true$;
            $q := llink$; $llink := t$;
            $t := p$; $p := q$ $\{R_1\}$
            end
    $up := true$;
    while $\{Q_2\}$ $up$ and ($t \neq$ nil) do
        with $t\uparrow$ do
            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 $t$=nil
end;
```

Figure 11: Deutsch-Schorr-Waite procedure

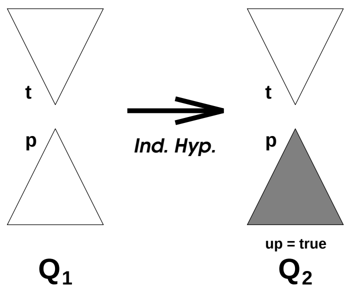

Figure 12: Inductive hypothesis for the DSW procedure

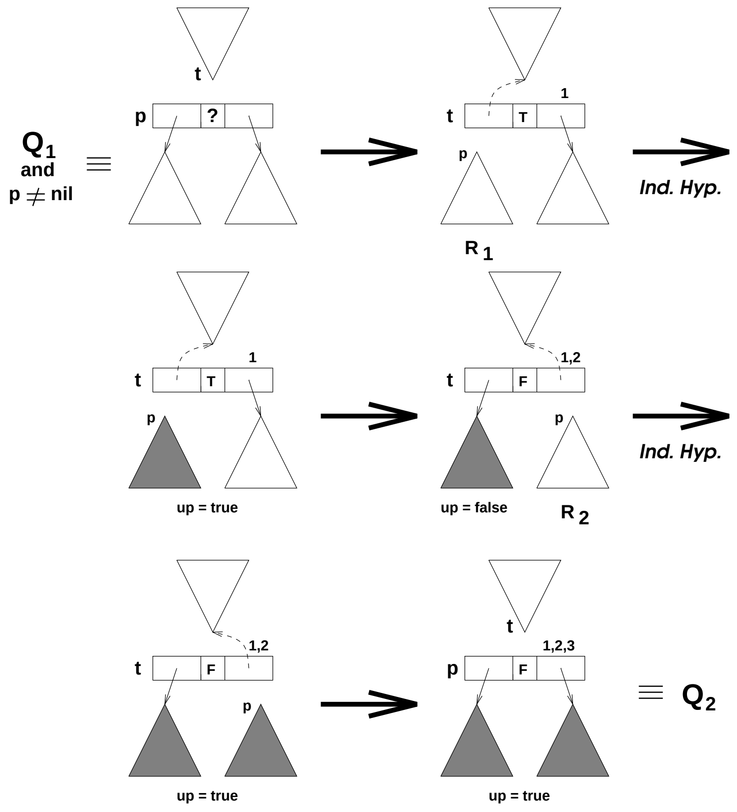

Figure 13: Proof of the DSW procedure

```
procedure $LD(p\colon ptr)$;
    var
        $empty,\ q,\ t\colon ptr$;
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 }
            $Visit(p)$;
            $q := t$; $t := p$;
            $p := p\uparrow.llink$;
            $t\uparrow.llink := t\uparrow.rlink$;
            $t\uparrow.rlink := q$
            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 discribed 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

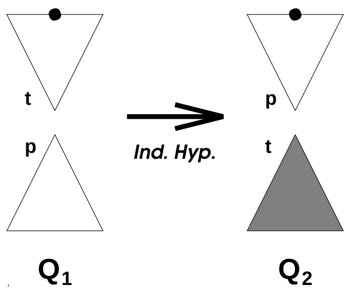

Figure 15: Inductive hypothesis for the LD procedure

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 *rlink* fields, and the *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 *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 *top* points to the rest of the stack. The variable *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.

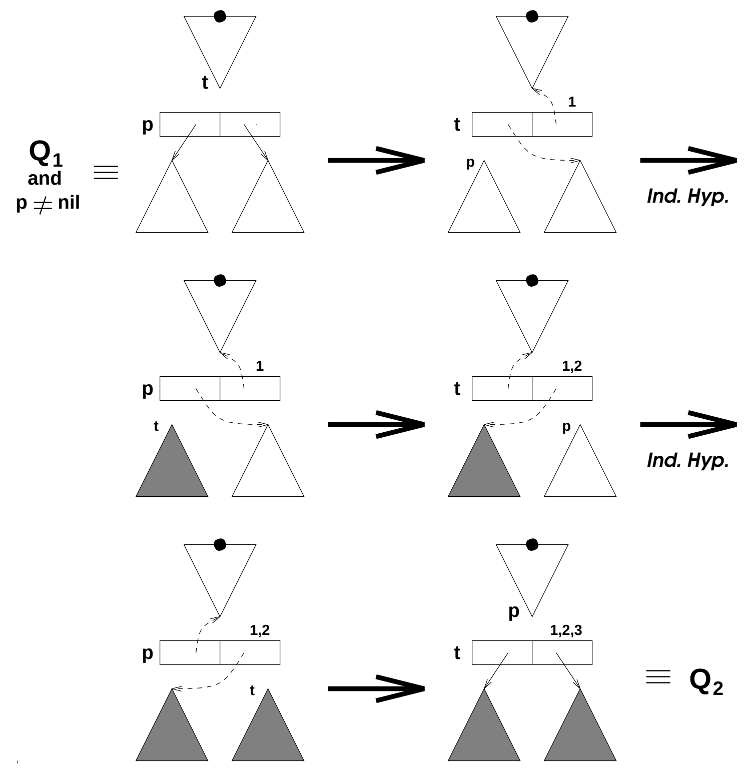

Figure 16: Proof of the LD procedure

```
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 {$Q_1$} { going down }
    PreOrderVisit(p);
    if (p↑.llink=nil) and (p↑.rlink=nil)
      then begin { leaf: go up }
          InOrderVisit(p);
          up := true
          end
      else
    if p↑.llink=nil then begin { go right }
        InOrderVisit(p);
        q := p↑.rlink;
        p↑.rlink := t;
        t := p; p := q {$R_1$}
        end
      else begin { go left }
        q := p↑.llink;
        p↑.llink := t;
        t := p; p := q {$R_2$}
        end
    until up;
    av := p;
```

Figure 17: Robson's procedure (continues)

```
 **repeat** {$Q_2$} { going up }
 $PostOrderVisit(p)$;
 **if** $t$=$empty$ **then** $up$ := $false$ { finished }
   **else with** $t\uparrow$ **do**
     **if** $llink$=**nil** **then begin** { coming from right }
         $q$ := $rlink$; $rlink$ := $p$;
         $p$ := $t$; $t$ := $q$
         **end**
       **else**
     **if** $rlink$=**nil** **then begin** { coming from left }
         $q$ := $llink$; $llink$ := $p$;
         $p$ := $t$; $t$ := $q$;
         $InOrderVisit(p)$;
         **end**
       **else**
     **if** $t$=$r$ **then begin** { coming from right }
         $q$ := $top$; $r$ := $q\uparrow.llink$; $top$ := $q\uparrow.rlink$; { pop }
         $q\uparrow.llink$ := **nil**;
         $q\uparrow.rlink$ := **nil**;
         $q$ := $rlink$; $rlink$ := $p$;
         $p$ := $t$; $t$ := $q$
         **end**
       **else begin** { coming from left }
         $av\uparrow.llink$ := $r$; $av\uparrow.rlink$ := $top$; $top$ := $av$; { push }
         $r$ := $t$; $q$ := $llink$;
         $llink$ := $p$; $p$ := $rlink$;
         $rlink$ := $q$; $up$ := $false$;
         $InOrderVisit(t)$ {$R_3$}
         **end**
 **until not** $up$
**until** $t$=$empty$
**end**;
```

Figure 17: Robson's procedure (cont.)

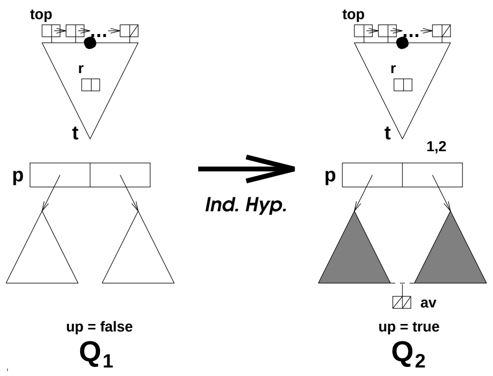

Figure 18: Inductive hypothesis for the Robson procedure

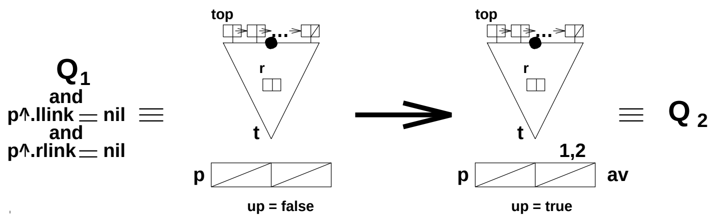

Figure 19: Proof of the Robson procedure – part I

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 $p\uparrow.llink$ and $p\uparrow.rlink$. 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

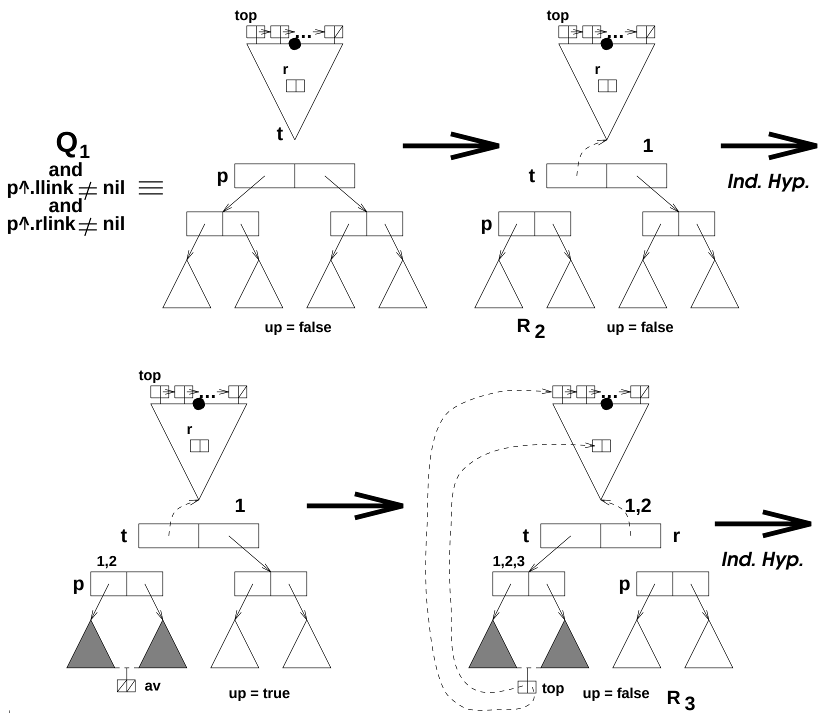

Figure 20: Proof of the Robson procedure – part II (continues)

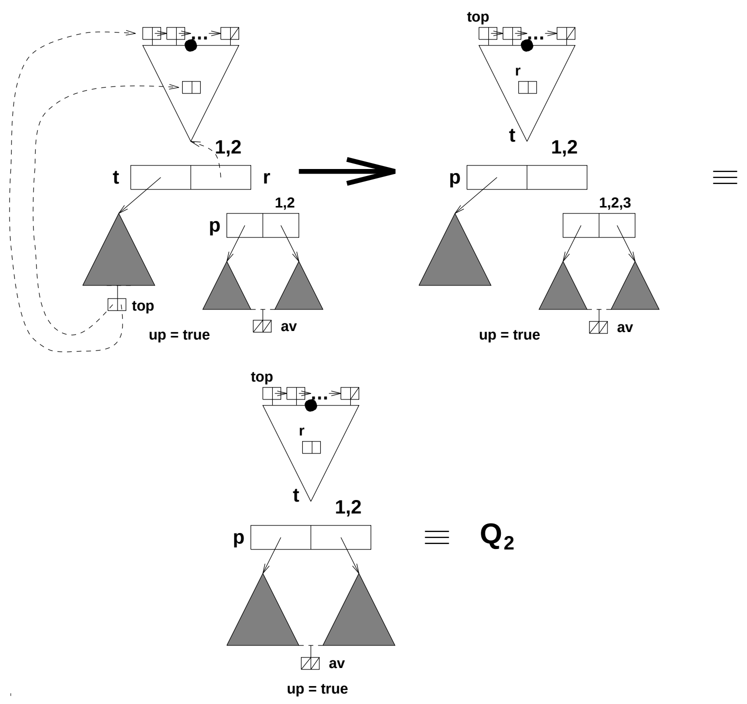

Figure 20: Proof of the Robson procedure – part II (cont.)

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 **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 **nil**, t 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 **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 **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 **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 $p =$ **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 $Morris(p\colon ptr)$;
    var $q\colon ptr$;
begin
while $\{Q_1\}$ $\{Q_2\}$ $p \neq$ nil do
    with $p\uparrow$ do
        if $llink$=nil
            then
                begin { empty left – go right }
                $PreOrderVisit(p)$;
                $InOrderVisit(p)$;
                $p := rlink$ $\{R_1\}$
                end
            else
                begin
                $q := llink$;
                while ($q\uparrow.rlink \neq$ nil) and ($q\uparrow.rlink \neq p$) do
                    $q := q\uparrow.rlink$;
                if $q\uparrow.rlink$=nil
                    then
                        begin { first time at $p$ – go down }
                        $PreOrderVisit(p)$;
                        $q\uparrow.rlink := p$; $p := llink$ $\{R_2\}$
                        end
                    else
                        begin { second time at $p$ – go right }
                        $q\uparrow.rlink :=$ nil;
                        $InOrderVisit(p)$;
                        $p := rlink$ $\{R_3\}$
                        end
                end
end;
```

Figure 21: Morris's procedure

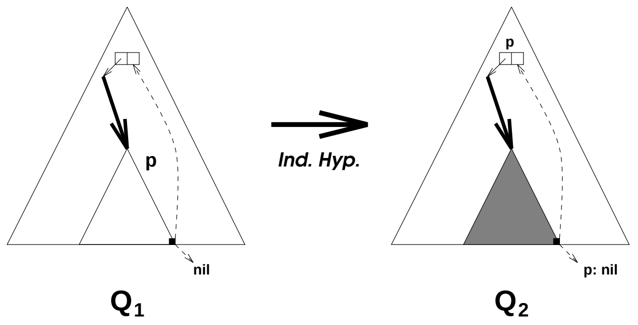

Figure 22: Inductive hypothesis for the Morris's procedure

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 = **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.

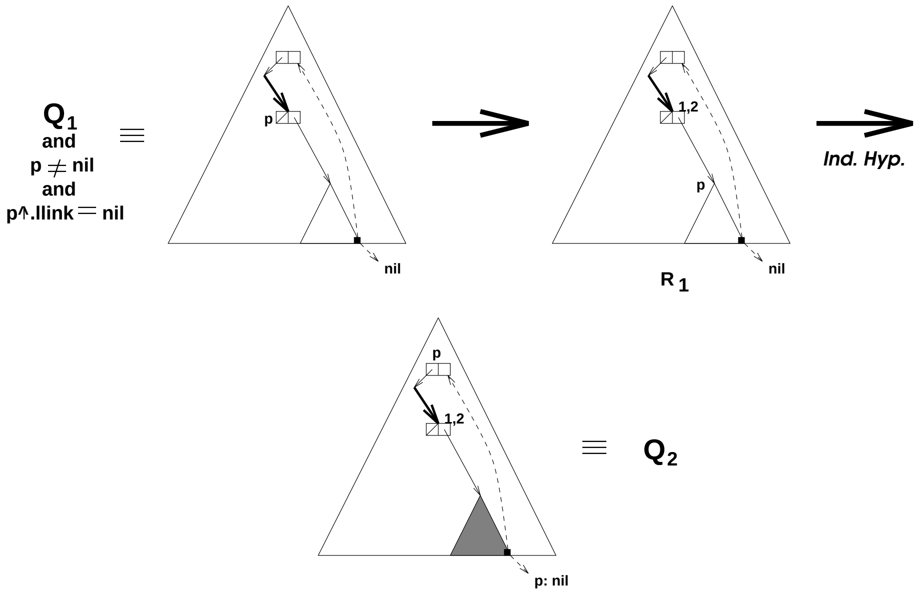

Figure 23: Proof of the Morris's procedure – part I

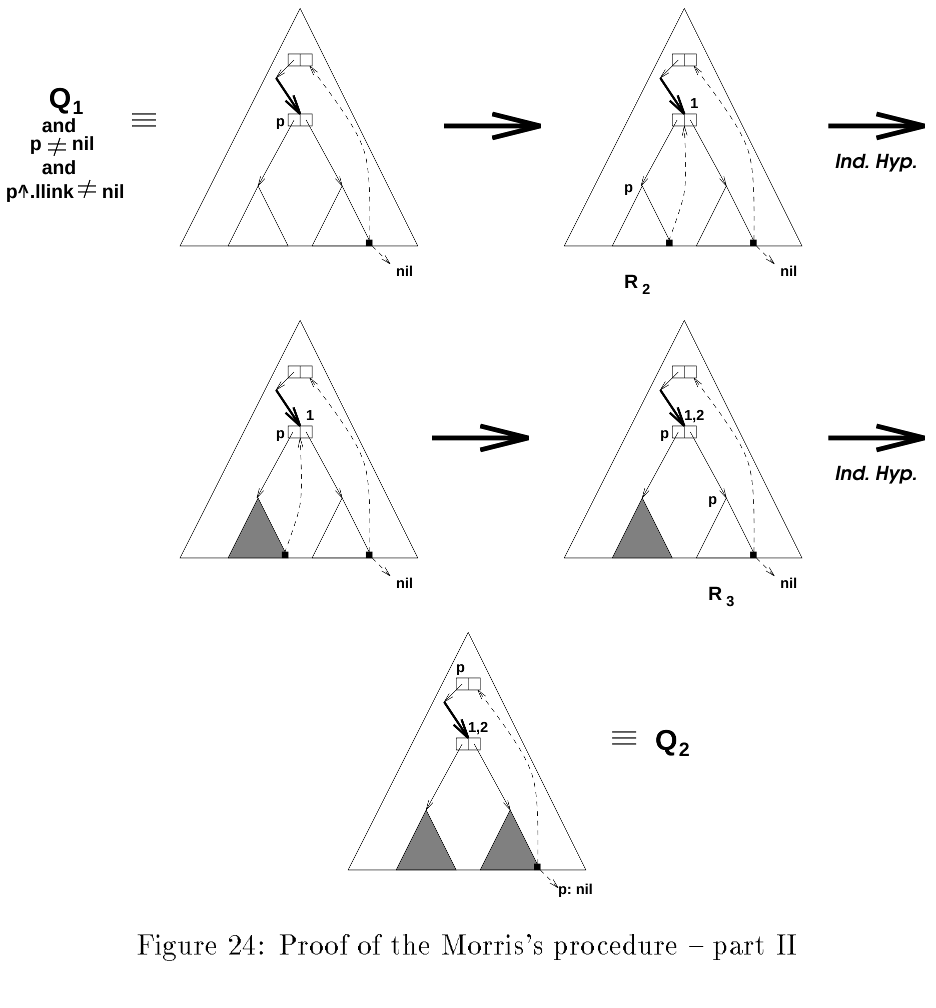

Figure 24: Proof of the Morris's procedure – part II

References

[1] R. M. Burstall, 'Some techniques for proving correctness of programs which alter data structures', in *Machine Intelligence* **7**, D. Michie (ed.), American Elsevier, 1972, 23-50.

[2] B. Dwyer, 'Simple algorithm for traversing a tree without auxiliary stack', *Information Processing Letters* **2** (1974) 143-145.

[3] D. Gries, 'The Schorr-Waite Graph Marking Algorithm', *Information Processing Letters* **11** (1979) 223-232.

[4] D. E. Knuth, *The Art of Computer Programming*, vol. 1, (Addison-Wesley, Reading, Mass. 1973).

[5] T. Kowaltowski, 'Data structures and correctness of programs', *Journal of the ACM* **26**,2 (1979) 283-301.

[6] G. Lindstrom, 'Scanning list structures without stacks or tag bits', *Information Processing Letters* **2** (1973) 47-51.

[7] J. M. Morris,'Traversing binary trees simply and cheaply', *Information Processing Letters* **9** (1979) 197-200.

[8] J. M. Robson, 'An improved algorithm for traversing binary trees without auxiliary stack', *Information Processing Letters* **2** (1973) 12-14.

[9] H. Schorr and W. M. Waite, 'An efficient machine-independent procedure for garbage collection in various structures', *Communications of the ACM* **10**,8 (1967) 501-506.

[10] R. W. Topor, 'The Correctness of the Schorr-Waite List Marking Algorithm', *Information Processing Letters* **11** (1979) 211-221.

Relatórios Técnicos

Departamento de Ciência da Computação — IMECC
Caixa Postal 6065
Universidade Estadual de Campinas
13081-970 – Campinas – SP
BRASIL

`reltec@dcc.unicamp.br`