A Structure-directed Total Correctness Proof Rule for Recursive Procedure Calls

P. PANDYA* AND M. JOSEPH
Computer Science Group, Tata Institute of Fundamental Research, Homi Bhabha Road, Bombay 400 005, India

Recursive procedures have been extensively used for a long time, yet it was only in 1977 that Sokolowski gave a proof rule for the total correctness of recursive procedure calls. Unfortunately, even for simple programs his rule requires the use of complex predicates that encode information about the depth of recursion. Thus the rule can be very difficult to use for practical programs. In this paper we propose a new rule for this purpose. This rule makes use of the structure of the recursion which is discovered by carrying out an interval analysis of the procedure call graph in the proof. Proofs using this rule are simpler to carry out, and it is shown that Sokolowski's rule is in fact a special case of the new rule.

Received March 1985

1.0 INTRODUCTION

In the past fifteen years, proof rules have been defined for almost every program construct and several methodologies based on these proof rules have been proposed for program development.[1] There is evidence that programmers in industry are beginning to use such formalisms in the design and verification of their programs. Hence it is crucial to examine these rules for their convenience of use. In this paper, we discuss the pragmatics of proving the properties of one fairly common program feature: the use of mutually recursive procedure calls.

Harel[2] has classified program proof techniques as either data-directed or syntax-directed. A data-directed technique uses reasoning (like induction) about the state of the data manipulated by the program; the structure of the control flow of the program is not directly used in the proof. In syntax-directed techniques, programs are looked upon as composed of syntactic entities, such as the **while** or **if** constructs, and assertions are attached to these entities at well-defined points. Rules[3] are given to compose the proofs of component entities into the proof of the whole program.

Syntax-directed methods offer many advantages. The complexity of the proof is reduced, as a program may be broken into syntactic components whose separate proofs can be combined to give the proof for the whole program. Thus the proof technique is compositional. Proofs follow the structure of the program, and are usually easy to understand; moreover, such proofs and programs can be developed hand in hand.[4, 1]

An important characteristic of a program proof is the extent to which it makes use of abstractions, i.e. the degree to which assertions used in the proof disregard those aspects of the working of the program which do not affect the property being proved. Abstraction is difficult to characterise formally, and therefore has hardly been studied; but it is an important pragmatic consideration. Proof rules can differ in their capacity to admit abstraction. For example, there are two well-known forms of the total correctness proof rule for **while** loops[2], one requiring the value of the bound function to decrease by exactly 1 and the other only requiring the value to decrease (by an arbitrary amount), on each iteration. Both rules are consistent and arithmetically complete. But the first forces the programmer to encode information about the number of iterations into the loop invariant. For many programs this can be extremely inconvenient, and the second rule is generally preferred for use in the proofs of real programs.

Sokolowski[5] has given a proof rule for the total correctness of recursive procedure calls. By Harel's classification this rule would be described as data-directed. As shown later, this rule can be difficult to use in practice because of its dependence on details of the depth of recursion. This results in the use of predicates that are often complex and non-intuitive, even for simple programs. Since recursion plays a significant role in sequential programs as diverse as compilers and mathematical software, we have been investigating other techniques for proving the total correctness of recursive procedure calls. This paper describes a structure-directed rule for this purpose. The proof involves showing that no cycle in the procedure call graph can cause infinite recursion. By carrying out interval analysis of the procedure call graph, it is possible to identify a set of procedures that 'cover' all the cycles in the call graph. We call such a set a header set. The new rule is based on induction over the number of calls to procedures in the header set and thus does not require analysis of the depth of recursion. Proofs using this rule are relatively simple to carry out. Moreover, it can be shown that Sokolowski's rule is a special case of the new rule.

2.0 TOTAL CORRECTNESS PROOF RULES FOR RECURSIVE PROCEDURE CALLS

A procedure P is said to be directly recursive if in its body it contains calls to itself, and indirectly recursive if there exist procedures $P_1, P_2, \ldots, P_n$ $(n > 1)$ such that each P_i, $1 \leqslant i \leqslant n$, contains a call to P_{i+1} and P_n has a call to P_1. An activation of a recursive procedure may give rise to an arbitrarily long activation sequence at run-timer, even infinitely long if the recursion does not terminate. Procedure P is said to terminate with respect to a given precondition if any activation of P satisfying its precondition results in only a finite activation sequence.

Assume that there is a set of procedures $S = \{P_1, \ldots, P_n\}$ closed upon mutual calls, i.e. these procedures do not call any procedure not belonging to S. The procedures are defined by

* To whom correspondence should be addressed.

{Pre: q_1}
{Post: r_1}
proc P_1: B_1 (call P_1, ..., call P_n) end;
⋮
{Pre: q_n}
{post: r_n}
proc P_n: B_n (call P_1, ..., call P_n) end;

Where B_i is the body of procedure P_i which may contain calls to procedures P_1, ..., P_n. Sokolowski's rule can be defined as follows.

Definition

valid-call (Procset, depth) Δ

$$\forall_j: P_j \in \text{Procset}: \{q_j\ (\text{depth})\}\ \text{call}\ P_j\ \{r_j\}$$

Rule 1 (Sokolowski)

$\forall k, P_k \in S,$

$$\vdash \{q_k(0)\}\ B_k\ \{r_k\} \qquad \text{(base step)}$$

$$\text{valid-call (S, i)} \vdash \{q_k\ (i+1)\}\ B_k\ \{r_k\} \qquad \text{(induction step)}$$

$\exists i \geqslant 0$: valid-call (S, i)

The intuition behind this rule has been described by Apt.[6]

(i) A computation is said to be (r, i)-deep if at any time no more than i calls can be active in its execution, and if it terminates in a state satisfying r.

(ii) If any execution of the statement S starting in a state satisfying q(i) is (r, i)-deep then {q(i)} S {r} holds.

(iii) If execution of any procedure $P_j \in$ Procset starting in a state satisfying q_j (depth) is (r_j, depth) deep then valid-call (Procset, depth) holds.

(iv) Assuming valid-call (S, i), we must establish valid-call (S, i+1). This is done by proving $\{q_k\ (i+1)\}\ B_k\ \{r_k\}$ for all $P_k \in S$ (induction step).

(v) We also need to show valid-call(S, 0) i.e. a procedure P_j starting in a state satisfying $q_j(0)$ does not make any procedure calls (base step). The conclusion of the rule then follows by induction.

Rule 1 requires the properties of all mutually recursive procedures to be derived from the same recursion depth counter. Sokolowski has proved the validity and completeness of this rule, and Apt[2] has given a complete proof system which includes this rule.

Rule 1 is based on induction over the depth of recursion. Essentially, we have to define a well-founded order over the state whose value decreases for each procedure call. The well-founded order is defined by predicates $q_j(i)$ relating the state and the recursion depth counter i.

Application of Rule 1 requires assertions that encode information about the depth of recursion to be attached to all the procedures, disregarding the structure of the flow of control between them, so Rule 1 is a data-directed rule (for a similar reason, Floyd's system[8] for the proof of flow-chart programs has also been described by Harel as data-directed). The structure of the control flow between procedures cannot explicitly be used in the proof and needs to be encoded into the definition of the predicates q(i). These can then become quite complex.

Consider the following example of a program for multiplying two numbers a and b with the result contained in variable z.

```
P: z+x*y = a*b /\ a,b,x,y ≥ 0
{pre: P}
{post: z = a*b}
procedure PRODUCT;
  begin
    if
      even(y) ⇒ call EVENPRODUCT;
    []
      Odd(y) ⇒ call ODDPRODUCT;
    fi
  end;
{pre: P /\ odd(y)}
{post: z = a*b}
procedure ODDPRODUCT;
  begin
    y,z: = y−1, z+x;
    call EVENPRODUCT
  end;
{pre: P /\ even(y)}
{post: z = a*b}
procedure EVENPRODUCT;
  begin
    if
      y = 0 ⇒ skip;
    []
      y ≠ 0 ⇒ x,y:= 2*x, y div 2;
        call PRODUCT
    fi
  end;
```

In order to prove the total correctness of this program using Rule 1, we must find predicates $q_p(i)$, $q_E(i)$ and $q_0(i)$, the preconditions to the procedures PRODUCT, EVENPRODUCT and ODDPRODUCT, which relate the state to the recursion depth counter i. The predicates must be such that with every successive procedure call the value of i decreases.

Unfortunately, the necessary predicates are quite difficult to discover, as finding them involves an analysis of the depth of recursion. One way of proceeding is to define a function f(y) to represent the depth of recursion for a given value of y. However, this function turns out to be quite complex, and we could only find a recursive formulation for it and had even then to use an approximate estimate of the depth of recursion! We give below an annotated version of the program PRODUCT using predicates $q_p(i)$, $q_E(i)$ and $q_O(i)$ which are defined using f(y). The proof of the program can be reconstructed from this using Rule 1. (The reader is invited to appreciate the complexity of the proof by finding better predicates!).

```
P: z+x*y = a*b /\ a,b,x,y ≥ 0
let f(y) = 2+f(y div 2)+y mod 2; f(0) = 2;
{q_P(i): P /\ f(y) ≤ i}
{post: z = a*b}
procedure PRODUCT;
  begin
    if
      even(y) ⇒
        {P /\ even(y) /\ f(y) ≤ i}
        {P /\ even(y) /\ (2+f(y div 2)+y mode 2 ≤ i)}
        {P /\ even(y) /\ (f(y div 2)+1 ≤ i−1)}...
                                          q_E(i−1)
        call EVENPRODUCT;
    []
```

```
      Odd(y) ⇒
        {P /\ odd(y) /\ (2+f(y div 2)+y mod 2 ⩽ i)}
        {P /\ odd(y) /\ (2+f(ydiv2) ⩽ i−1)}...q_O(i−1)
        call ODDPRODUCT;
    fi
  end;
{q_O(i): P /\ odd(y) /\ (f(y div 2)+2 ⩽ i)}
{post: z = a*b}
procedure ODDPRODUCT;
  begin
    {P /\ odd(y) /\ (f((y−1) div 2)+1 ⩽ i−1)}
      ...(as odd(y) ⇒ y div 2 = (y−1) div 2)
    y,z:= y−1,z+x;
    {P /\ even(y) /\ (f(y div 2)+1 ⩽ i−1)}...q_E(i−1)
    call EVENPRODUCT
  end;
{q_E(i): P /\ even(y) /\ (f(y div 2)+1 ⩽ i)}
{post: z = a*b}
procedure EVENPRODUCT;
  begin
    if
      y = 0 ⇒ skip;
    []
      y ≠ 0 ⇒
        {P /\ f(y div 2) ⩽ i−1}
        x,y: = 2*x, y div 2;
        {P /\ f(y) ⩽ i−1}...q_P(i−1)
        call PRODUCT
    fi
  end;
```

Using Rule 1 we can conclude

$\{\exists i \geqslant 0: q_P(i)\}$ **call** PRODUCT $\{z = a*b\}$

This simplifies to

{P} **call** PRODUCT $\{z = a*b\}$.

It is fairly clear that this proof has a measure of artificiality and that the predicates $q_j(i)$ offer little insight into the structure of the program. A reader going through this proof is not likely to appreciate easily why the program terminates. And this is despite the fact that there is a very simple and natural argument for its termination. For the program PRODUCT, calls go cyclically either PRODUCT → ODDPRODUCT → EVENPRODUCT → PRODUCT or PRODUCT → EVENPRODUCT → PRODUCT. In either case, proving the termination of PRODUCT is sufficient to prove the termination of the recursion.

On each successive call to the procedure PRODUCT the value of y becomes y div 2. Thus we can argue that the procedure PRODUCT terminates because on each successive call to PRODUCT, the value of y decreases, and if a call to PRODUCT is made with y = 0 then no further recursive call to PRODUCT is made. It is possible to give a simple total correctness proof based on the above argument using induction over the number of calls to PRODUCT active at any instant. Unfortunately, Rule 1 does not allow such induction.

2.1. A structure-directed rule for the total correctness of recursive procedure calls

For most programs with recursive procedure calls, we can select a subset of procedures whose termination implies the termination of the recursion. Such a set is called a header set. Clearly, the selection of the header set depends upon the structure of recursion. Often, there is an easily expressible, natural relationship between the state changes and the calls to the procedures in the header set that can be used to give a simple proof of total correctness.

We shall analyse the static structure of recursion in a program by constructing its call graph. Let $\{P_1, \ldots, P_n\}$ be the set of procedures in a program. The call graph G of the program is constructed as follows.

The main body and each procedure $P_1 . . P_n$ is represented by a node of the graph. If there is a call from procedure P_i to procedure P_j, add a directed edge from the node P_i to the node P_j. Further, add directed edges from the main node to all the procedures called by the main body. The main node has no incoming arc and is called the initial node of the graph.

A procedure P is nonrecursive if no cycle of G passes through its node. It is directly recursive if the only cycle passing through its node is a self-loop. A set of procedures $S = \{P_1, \ldots, P_n\}$ is mutually recursive if for all $P_i, P_j \in S$ there is a cycle in G passing through the nodes of P_i and P_j.

Definition

Let $S = \{P_1, \ldots, P_n\}$ be the set of procedures in a program. [S1,S2] is said to be a termination partition of S, iff

(1) $S = S1 \cup S2$
(2) Every cycle in the call graph of the program passes through at least one node whose associated procedure is contained in S1. The set S1 is called a header set.

Given S, which is a subset of the procedures of a program, an activation of procedure P_i by a procedure call from P_j is called an activation internal to S if both P_i and P_j are in S. The termination partition [S1,S2] has the property that no activation of P_i in S2 satisfying $\{Pre P_i\}$ results in an infinite sequence of activations internal to S2 (since all cycles pass through the nodes of S1).

An activation of P_i may still result in an infinite activation sequence if there are calls to procedures in S1 in the bodies of the procedures of S2. For example let S = {P,Q,R,S} have a termination partition [{P},{Q,R,S}]; the activation sequence (P-Q-R-S) does not violate the definition of the termination partition and yet is infinite.

Thus, proving the termination of the procedures of a program reduces to showing that there can be only a finite number of recursive activations of the procedures in the header set. (Naturally, we assume that the proof of termination of all the loops in the program has been established using some total correctness rule,[7] so that loops do not cause nontermination.) This is achieved by the procedure call proof Rule 2.

Rule 2

If [S1,S2] is a termination partition of S,
for all $P_k \in S$,

valid-call(S2,0) $\vdash \{q_k(0)\}\ B_k\ \{r_k\}$ (base step)
valid-call(S1,i), valid-call(S2,i+1) $\vdash$
$\{q_k(i+1)\}\ B_k\ \{r_k\}$ (induction step)
$\exists i \geqslant 0$: valid-call(S,i)

By analogy with the explanation for Sokolowski's rule, the intuition behind this rule can be explained as follows.

(i) A computation is ([S1,S2],r,i)-deep if at any time no more than i calls of procedures in S1 are active in its execution and if it terminates in a state satisfying r. No limit is placed on the activation of the procedures in S2.

(ii) If any execution of the statement S starting in a state satisfying q(i) is ([S1,S2],r,i)-deep, then {q(i)} S {r} holds.

(iii) If execution of any procedure $P_j \in$ Procset starting in a state satisfying q_j(depth) is ([S1,S2],r_j,i)-deep then valid-call(Procset,depth) holds.

(iv) The rule is based on induction over the number of calls to procedures in the header set. Thus each call to a procedure in S1 increases the induction depth while a call to a procedure in S2 does not do so. Hence, assuming valid-call(S1,i) and valid-call (S2,i+1), we must establish valid-call (S,i+1). This can be shown by proving $\{q_k(i+1)\}\ B_k\{r_k\}$ for all procedures $P_k \in S$ (induction step).

(v) We must also prove valid-call(S,0) assuming valid-call(S2,0); i.e. any execution of a procedure P_j starting in a state satisfying $q_j(0)$ terminates without making calls to the procedures of the header set (base step). The conclusion of the rule follows by induction.

This rule is based on induction over the number of calls to the procedures in the header set that are active at any instant. It admits simpler proofs than Rule 1 as it allows the well-founded order over the state to be defined such that it decreases only on calls to the procedures in the header set, and not necessarily on calls to other procedures (this condition is weaker than the one required by Rule 1). Thus it is possible to exploit any natural relationship between the state changes and the calls to header procedures. No such relationship may exist between the state changes and calls to all the procedures. Usually, a program has many termination partitions and the programmer may choose one that gives the simplest proof. It is our claim that such a proof then closely mirrors a programmer's informal reasoning.

Result

If we select the termination partition of S to be [S,Null] then Rule 2 reduces to Sokolowski's rule.

Using Rule 2, we now give a proof of the program PRODUCT. Fig. 1 gives the call graph for the program. An obvious termination partition is [{PRODUCT} , {EVENPRODUCT,ODDPRODUCT}]. The proof for the procedure PRODUCT can easily be constructed from the following annotated program, using Rule 2.

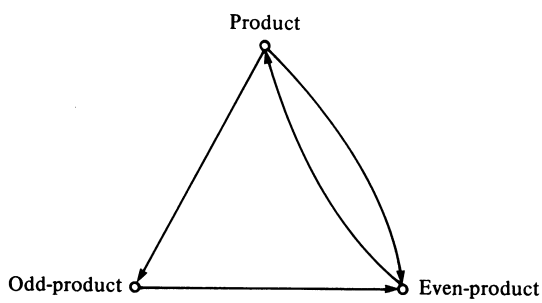

Figure 1.

```
{{q_P(i): P/\y ≤ i}
{z = a*b}
procedure PRODUCT;
  begin
    if even(y) ⇒ {P/\y ≤ i/\even(y)}...q_E(i)
                 call EVENPRODUCT;...(step A)
                 {z = a*b}
    [] Odd(y) ⇒ {P/\y ≤ i/\odd(y)}...q_O(i)
                 call ODDPRODUCT;...(step B)
                 {z = a*b}
    fi
  end;
{q_O(i): p/\y ≤ i/\odd(y)}
{z = a*b}
procedure ODDPRODUCT;
  begin
    y,z:= y-1,z+x;
    {P/\y ≤ i/\even(y)}...q_E(i)
    call EVENPRODUCT...(step C)
    {z = a*b}
  end
{q_E(i): p/\y ≤ i/\even(y)}
{z = a*b}
procedure EVENPRODUCT;
  begin
    if y = 0 ⇒ {p/\y = 0}
               skip;
               {z = a*b}
    [] y ≠ 0 ⇒ {p/\0 < y ≤ i/\even(y)}
               {P/\(y div 2) ≤ i-1}
               x,y:= 2*x, y div 2;
               {p/\y ≤ i-1}...q_P(i-1)
               call PRODUCT...(step D)
               {z = a*b}
    fi
  end;
```

Steps (A),(B) and (C) follow from the assumption validcall(S2,i−1). Step (D) follows from the assumption validcall(S1,i). (We leave it to the reader to verify this annotated program.) Using Rule 2 we can conclude:

$$\{\exists i \leq 0\colon P/\backslash y \leq i\}\ \text{call PRODUCT}\ \{z = a*b\}.$$

This simplifies to {P} call PRODUCT {z = a*b}.

It is instructive to compare this proof with the earlier proof using Rule 1. For the program PRODUCT, the relationship between the state and the number of activations of PRODUCT i is given by y ≤ i. No such simple relationship exists between the state and the depth of recursion. In the above proof we have abstracted information from the full details of the state changes in the program and focused attention only on how the state changes on a call to the procedure PRODUCT. It is the ability to allow such abstractions in proofs that makes Rule 2 convenient to use.

2.2 Termination partitions

The termination partition of a set of procedures can be found by analysing the call graph. Broadly, the method involves identifying the loops (intervals) within the call graph, working outwards from the innermost loops. Techniques for interval analysis of flow graphs have been extensively used in compilers for loop-optimisation.[9]

Definitions

For a call graph G having the initial node n_0,

(*a*) for any node n of the graph G, the interval with header n, denoted by I(n), is defined as:

(1) n is in I(n);
(2) if $m \neq n_0$ and all the predecessors of m are in I(n), then m is in I(n);
(3) only nodes that satisfy (1) and (2) are in I(n).

(*b*) the interval partition P(G) of the call graph G is defined as:

(1) $I(n_0)$ is in P(G);
(2) If n has a predecessor in an interval of P(G) but n is not in any interval of P(G) then I(n) is in P(G);
(3) only nodes that satisfy (1) and (2) are in I(n).

(*c*) Each interval of P(G) can be collapsed into a single node to give the reduced graph I(G). If G has an edge $n \rightarrow m$, and n and m are in different intervals of G (m will necessarily be an interval header), then I(G) has an edge from the interval containing n to the interval containing m. The initial node of I(G) is the interval containing n.

(*d*) A graph is called reducible if by a sequence of reductions it can be reduced to a single node.

(*e*) An hset is a set with one of its elements designated as the head element.

Using the notation $I^0(G) = G$, $I^k(G) = I(I^{k-1}(G))$, with each node of the graph I (G) we associate an hset of procedures:

(i) If n is in $I^0(G)$ then hset(n) = {P}, where node n represents the procedure P in the call graph G.

(ii) Let $S = n_1, \ldots, n_P$ be an interval of $I^{k-1}(G)$ with the header node n_1 and let m be a node in $I^k(G)$ such that m = I(S); then

$hset(m) = \cup_{1 \leq i \leq P}$:hset($n_i$)
$head(m) = head(n_1)$.

An interval has the property that all its cycles pass through the header node. This leads to the following method for the construction of a termination partition. Construct the call graph G of the program;

```
S1:= {head(n)|node n has a self-loop in G};
loop
  for each interval I(n) in P(G) do
    if I(n) contains a cycle then S1:= S1 ∪ {head(n)}
  end for;
  if reducible(G) then G:= I(G)
                  else exit-loop
end loop;
if G is not a single node then
  for each node n in G do S1:= S1 ∪ {head(n)}
  end for;
end if;
```

Example

Consider a parser for recognising an assignment statement of a Pascal-S program. Typically such a parser would be a part of the Pascal-S compiler. The statement may contain Pascal-like array and record references and is defined by the grammar below.

⟨STATEMENT⟩::= ⟨ASSIGNMENT⟩
⟨ASSIGNMENT⟩::= ⟨IDENTIFIER⟩ [⟨SELECTOR⟩] ':=' ⟨EXPRESSION⟩
⟨EXPRESSION⟩::= ⟨SIMPLE_EXPR⟩ { ('>'|'<'|'=') ⟨SIMPLE_EXPR⟩}
⟨SIMPLE_EXPR⟩::= ['+'|'−'] ⟨TERM⟩ { ('+'|'−'|'OR') ⟨TERM⟩ }
⟨TERM⟩::= ⟨FACTOR⟩{('*'|'/'|'AND')⟨FACTOR⟩ }
⟨FACTOR⟩::= NUMBER | '('⟨EXPRESSION⟩')' | ⟨IDENTIFIER⟩ [⟨SELECTOR⟩] | ⟨IDENTIFIER⟩ ['(' ⟨EXPRESSION⟩ {','⟨EXPRESSION⟩} ')'] | 'not' ⟨FACTOR⟩
⟨SELECTOR⟩::= ('.'IDENTIFIER | '['⟨EXPRESSIONLIST⟩']') { '.'IDENTIFIER | '['⟨EXPRESSIONLIST⟩']' }
⟨EXPRESSIONLIST⟩::= ⟨EXPRESSION⟩ {',' ⟨EXPRESSION⟩}

A simple recursive descent parser for recognising the assignment statement would consist of one procedure for each nonterminal, with mutually recursive calls. We give an outline of the procedures of the parser. We shall not give the bodies for these well-known procedures, but the interested reader is referred to Wirth.[10]

```
procedure STATEMENT
  procedure SELECTOR
    procedure EXPRESSION_LIST
  procedure EXPRESSION
    procedure SIMPLE_EXPRESSION
      procedure TERM
        procedure FACTOR
  procedure ASSIGNMENT_STATEMENT
```

The call graph of this parser is given in Fig. 2. The figure also shows the interval analysis of the call graph giving the following obvious termination partition

S1 = {expression, factor}
S2 = {assignment,selector,simple-expression,term, functioncall}

The termination of the parser is established by noting that on each successive call to the procedure EXPRESSION or FACTOR, the length of the remaining input string to be parsed decreases (and the input string is of finite length). This argument can be directly formalised using Rule 2 with the predicate {i < scannedlen}, where i is the number of activations of the procedures EXPRESSION and FACTOR and scannedlen is the length of input string processed by the parser (including the look-ahead). We shall not give the complete proof.

3.0 DISCUSSION

Programmers tend to use proof rules informally, so it is important that formal proofs mirror the informal reasoning closely. Proofs with complex and non-intuitive predicates are thus generally unsatisfactory. In some cases, the need for such proofs can point to the inelegance of language constructs. However, it can also point to the inadequacy of a proof rule in permitting abstraction.

Recursion is certainly not an inelegant construct, and its widespread use stresses the need for a sound an simple means of reasoning about correctness and termination. For many recursive programs, our structure-directed

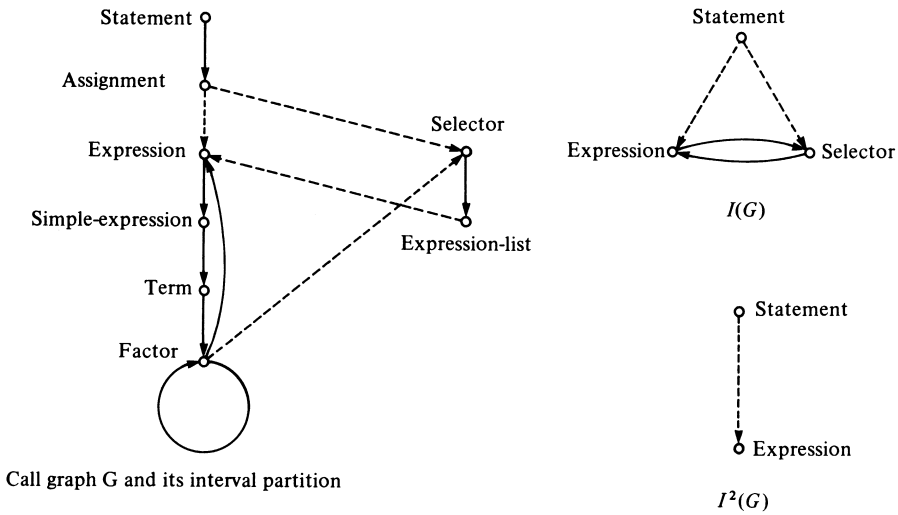

Figure 2.

approach yields simple and intuitively natural proofs. The method does involve the overhead of interval analysis, but most cases of recursion found in practice have quite simple structures, and the termination partitions for such programs are fairly obvious. Ideally, the termination partition for a set of recursive procedures should evolve along with their design.

It should be noted that Rule 2 uses simple induction, i.e. for proving $\{q(i+1)\}$ call P $\{r\}$ we assume $\{q(i)\}$ call P $\{r\}$. Sometimes it is more convenient to use generalised induction, i.e. we may instead assume $\forall_j: 0 \leqslant j \leqslant i: \{q(j)\}$ call P $\{r\}$. Rule 2 can thus be modified to the following rule.

Rule 3

for all k, $P_k \in S$,

valid-call(S2,0) $\vdash \{q_k(0)\}\ B_k\ \{r_k\}$ (base step)

$\forall_j: 0 \leqslant j \leqslant i$: valid-call (S,j), valid-call(S2,i+1) $\vdash$ $\{q_k(i+1)\}\ B_k\ \{r_k\}$ (induction step)

$\exists i \geqslant 0$: valid-call (S,i)

Although Rule 3 (and Rule 2) is for parameterless procedures, it can be used for procedures with parameters. There have been many proposals for proof rules for procedure calls with parameters, disallowing recursion. These rules specify how actual parameters must be substituted for the formal parameters in the pre- and postconditions of the procedure bodies to obtain the pre- and postconditions of the call; e.g. the Gries/Levin rule[11] allows value-result and variable parameters along with global variable references, with the restriction that actual variable parameters and global variables are disjoint. We shall call such rules the parameter substitution rules.[12]

In total correctness proofs, the preconditions to the procedures q(i) contain, besides the formal parameters, the induction depth counter i as a free variable. Rule 2 specifies how the variable i must be substituted by i−1 or i in q(i) to obtain the precondition of the call. Thus the parameter substitution rules and Rule 2 act on disjoint sets of variables (provided procedure parameters are disallowed) and hence are orthogonal. So Rule 2 can be used with any parameter substitution rule to prove the total correctness of procedures with parameters.

A major advantage of syntax-directed proof rules is that they lead to compositional proof systems. Unfortunately, total correctness proof rules for recursive procedures cannot be syntax-directed because there are no syntactic constructs restricting the dependencies between procedure calls. In fact, such rules cannot even be modular, because a change in any procedure may require the entire proof to be reconstructed (e.g. for a call graph which is complete). Our rules attempt to preserve some of the advantages of a syntax-directed rule because they allow attention to be focused on some procedures (i.e. those in the header set) rather than all the procedures. Hence, changes in non-header procedures will not affect the termination properties of the proof.

We have not dealt with the issues of validity and completeness of our proof rule. Since Sokolowski's complete rule is a special case of our rule, we expect our proof rule also to be complete (when used in conjunction with the other axioms given by Apt).

It seems possible to apply similar structural analysis techniques to proofs of deadlock freedom and termination of data-flow and distributed programs.

Acknowledgement

The authors thank Kamal Lodaya and S. Mahadevan for their helpful comments.

REFERENCES

1. D. Gries, *The Science of Programming*. Springer-Verlag, Heidelberg (1981).
2. D. Harel, Proving the correctness of regular deterministic programs: a unifying survey using dynamic logic. *Theoretical Computer Science* **12** (1), 61–79 (1980).
3. C. A. R. Hoare, An axiomatic basis for computer programming. *Communications of the ACM* **12** (10), 576–583 (1969).
4. E. W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey
5. S. Sokolowski, Total correctness for procedures. *Proc. 6th*

Symp. on the Mathematical Foundations of Computer Science, LNCS **53**, 475–492, Springer-Verlag, Heidelberg (1977).
6. K. R. Apt, 'Ten years of Hoare's logic: survey – part 1. *ACM Transations on Programming Languages and Systems* **3** (4), 431–483 (1981).
7. Z. Manna and A. Pneuli, Axiomatic approach to total correctness of programs. *Acta Informatica* **3** (3), 243–263 (1974).
8. R. W. Floyd, 'Assigning meanings to programs. In *Proceedings of the AMS Symposium on Applied Mathematics* **19**, 19–31. American Mathematical Society, Providence, R.I. (1967).
9. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*. Addison Wesley, London (1977).
10. N. Wirth, Pascal-S: a subset and its implementation. In *Pascal – The Language and its Implementation*, edited by D. W. Barron, pp. 199–260. J. Wiley, Chichester (1981).
11. D. Gries and G. Levin, Assignment and procedure call proof rules. *ACM Transactions on Programming Languages and Systems* **2** (4), 564–579 (1980).
12 S. A. Cook, Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing* **7** (1), 70–90 (1978).