CHAPTER 8

Example Runs

"By their fruits you shall know them."

— Matthew 7:20

"Imitate those who through faith and patience inherit the promises."

— Hebrews 6:12

In this chapter we take the verification condition generator for the Sunrise programming language presented in the last chapter, and apply it to prove several example programs. We prove these programs totally correct within the HOL theorem prover, and thus complete soundness is assured.

Given the *vcg* function defined in the last chapter and its associated correctness theorem, proofs of program correctness may now be partially automated with security. This has been implemented as an HOL tactic, called `VCG_TAC`, which uses the VCG soundness theorem to transform a given program correctness goal to be proved into a set of subgoals which are the verification conditions returned by the *vcg* function. These subgoals are then proved within the HOL theorem proving system, using all the power and resources of that theorem prover, directed by the user's ingenuity. The reliance on the VCG soundness theorem is the "faith" referred to above, and the completion of the proofs within HOL by the programmer is the "patience." The "promise" is verified programs.

The `VCG_TAC` tactic has the ability to print a trace of its processing while it works, which provides both a running commentary on its construction of the implicit proof of the program's correctness, and also provides the expressions which serve as the annotations between commands in a skeleton of the program's proof. This trace may be turned on or off at the user's will, by setting a global flag. If it is turned off, nothing is printed until the verification condition subgoals are displayed.

8.1 Quotient/Remainder

As a first example, we consider a program to compute the integer quotient and remainder of a pair of numbers. We do not have division or remainder operators present in the Sunrise programming language, so we will simulate them by an algorithm of repeated subtraction. This example has no recursion; its purpose is to demonstrate the syntactic analysis of the VCG.

Here is an expression of the quotient/remainder procedure, offered as a goal for the VCG. The following is the actual text submitted to HOL:

```
g [[ program
        procedure quotient_remainder (var q,r; val x,y);
           pre  0 < y;
           post ^x = q * ^y + r /\ r < ^y;

           r := x;
           q := 0;
           assert ^x = q * ^y + r /\ 0 < y /\ ^y = y
             with r < ^r
           while ~(r < y) do
              r := r - y;
              q := ++q
           od
        end procedure;

        quotient_remainder(q,r;7,3)
     end program
     [ q = 2 /\ r = 1 ]
  ]];;
```

The double square brackets ("`[[`" and "`]]`") enclose program text which is parsed into an HOL term containing the syntactic constructors that form the program specification. This parser was made using the parser library of HOL.

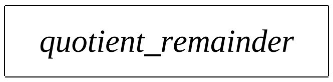


Figure 8.1: Procedure Call Graph for Quotient/Remainder Program.

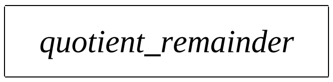


Figure 8.2: Procedure Call Tree for root procedure *quotient_remainder*.

This program's call graph is very simple, consisting of one procedure with no calls at all; it is shown in Figure 8.1. The call tree rooted at *quotient_remainder* is equally simple, shown in Figure 8.2.

Applying `VCG_TAC` to the program correctness goal with the tracing turned on produces the following.

```
#e(VCG_TAC);;
OK..
For procedure `quotient_remainder`,

By the "ASSIGN" rule, we have
   [[ {(^x = (q + 1) * ^y + r /\ 0 < y /\ ^y = y) /\ r < ^r}
       q := ++q
      {(^x = q * ^y + r /\ 0 < y /\ ^y = y) /\ r < ^r} ]]

By the "ASSIGN" rule, we have
   [[ {(^x = (q + 1) * ^y + (r - y) /\ 0 < y /\ ^y = y) /\ r - y < ^r}
       r := r - y
      {(^x = (q + 1) * ^y + r /\ 0 < y /\ ^y = y) /\ r < ^r} ]]

By the "SEQ" rule, we have
   [[ {(^x = (q + 1) * ^y + (r - y) /\ 0 < y /\ ^y = y) /\ r - y < ^r}
       r := r - y; q := ++q
      {(^x = q * ^y + r /\ 0 < y /\ ^y = y) /\ r < ^r} ]]

By the "WHILE" rule, we have
   [[ {^x = q * ^y + r /\ 0 < y /\ ^y = y}
       assert ^x = q * ^y + r /\ 0 < y /\ ^y = y
         with r < ^r
       while ~(r < y) do
          r := r - y; q := ++q
       od
      {^x = q * ^y + r /\ r < ^y} ]]
with verification conditions
   "[[[ {((^x = q * ^y + r /\ 0 < y /\ ^y = y) /\ ~(r < y)) /\
         r = ^r ==>
         (^x = (q + 1) * ^y + (r - y) /\ 0 < y /\ ^y = y) /\
         r - y < ^r} ]];
     [[ {(^x = q * ^y + r /\ 0 < y /\ ^y = y) /\ ~(~(r < y)) ==>
         ^x = q * ^y + r /\ r < ^y} ]]]"
```

```
By the "ASSIGN" rule, we have
   [[ {^x = 0 * ^y + r /\ 0 < y /\ ^y = y}
       q := 0
      {^x = q * ^y + r /\ 0 < y /\ ^y = y} ]]

By the "ASSIGN" rule, we have
   [[ {^x = 0 * ^y + x /\ 0 < y /\ ^y = y}
       r := x
      {^x = 0 * ^y + r /\ 0 < y /\ ^y = y} ]]

By the "SEQ" rule, we have
   [[ {^x = 0 * ^y + x /\ 0 < y /\ ^y = y}
       r := x; q := 0
      {^x = q * ^y + r /\ 0 < y /\ ^y = y} ]]

By the "SEQ" rule, we have
   [[ {^x = 0 * ^y + x /\ 0 < y /\ ^y = y}
       r := x; q := 0; assert ^x = q * ^y + r /\ 0 < y /\ ^y = y
                         with r < ^r
                       while ~(r < y) do
                          r := r - y; q := ++q
                       od
      {^x = q * ^y + r /\ r < ^y} ]]
with verification conditions
   "[[[ {((^x = q * ^y + r /\ 0 < y /\ ^y = y) /\ ~(r < y)) /\
         r = ^r ==>
         (^x = (q + 1) * ^y + (r - y) /\ 0 < y /\ ^y = y) /\
         r - y < ^r} ]];
     [[ {(^x = q * ^y + r /\ 0 < y /\ ^y = y) /\ ~(~(r < y)) ==>
         ^x = q * ^y + r /\ r < ^y} ]]]"

By precondition strengthening, we have
   [[ {(^q = q /\ ^r = r /\ ^x = x /\ ^y = y /\ true) /\ 0 < y}
       r := x; q := 0; assert ^x = q * ^y + r /\ 0 < y /\ ^y = y
                         with r < ^r
                       while ~(r < y) do
                          r := r - y; q := ++q
                       od
      {^x = q * ^y + r /\ r < ^y} ]]
with additional verification condition
```

```
   [[ {(^q = q /\ ^r = r /\ ^x = x /\ ^y = y /\ true) /\ 0 < y ==>
       ^x = 0 * ^y + x /\ 0 < y /\ ^y = y} ]]

Examining the structure of the procedure call graph:

Traversing the call graph back from the procedure quotient_remainder:

By the call graph progress from procedure quotient_remainder
to quotient_remainder, we have
   [[ {0 < y /\ true}
      quotient_remainder-<>->quotient_remainder
      {false} ]]

For the main body,

By the "CALL" rule, we have
   [[ {(0 < 3 /\ true) /\
       (!q r x1 y1. 7 = q * 3 + r /\ r < 3 ==> q = 2 /\ r = 1)}
       quotient_remainder(q,r;7,3)
      {q = 2 /\ r = 1} ]]

By precondition strengthening, we have
   [[ {true} quotient_remainder(q,r;7,3) {q = 2 /\ r = 1} ]]
with additional verification condition
   [[ {true ==> (0 < 3 /\ true) /\
                (!q r x1 y1. 7 = q * 3 + r /\ r < 3 ==>
                             q = 2 /\ r = 1)} ]]

4 subgoals
"0 < 3 /\
 (!q r x1 y1. (7 = (q * 3) + r) /\ r < 3 ==> (q = 2) /\ (r = 1))"

"!^x q ^y r y.
  ((^x = (q * ^y) + r) /\ 0 < y /\ (^y = y)) /\ r < y ==>
  (^x = (q * ^y) + r) /\ r < ^y"

"!^x q ^y r y ^r.
  (((^x = (q * ^y) + r) /\ 0 < y /\ (^y = y)) /\
    ~r < y) /\ (r = ^r) ==>
  ((^x = ((q + 1) * ^y) + (r - y)) /\ 0 < y /\ (^y = y)) /\
   (r - y) < ^r"
```

```
"!^q q ^r r ^x x ^y y.
  ((^q = q) /\ (^r = r) /\ (^x = x) /\ (^y = y)) /\ 0 < y ==>
  (^x = (0 * ^y) + x) /\ 0 < y /\ (^y = y)"

() : void
```

These four subgoals, in this order, roughly correspond to the following claims:

- The main body is partially correct.
- The loop invariant is sufficiently powerful.
- The loop invariant is maintained, and the progress expression decreases.
- The procedure's body is partially correct.

Of these four subgoals, all are readily solved. This proof has been completed in HOL, yielding the following theorem. There are slight differences with the original text, as this was prettyprinted according to a standard template.

```
|- [[ program
         procedure quotient_remainder(q,r;x,y);
            global ;
            pre  0 < y;
            post ^x = q * ^y + r /\ r < ^y;
            recurses with false;

            r := x; q := 0;
            assert ^x = q * ^y + r /\ 0 < y /\ ^y = y
              with r < ^r
            while ~(r < y) do
               r := r - y; q := ++q
            od
         end procedure;

         quotient_remainder(q,r;7,3)
      end program
      [q = 2 /\ r = 1] ]]
```

8.2 McCarthy's "91" Function

As a second example, we consider McCarthy's "91" function. The purpose of this example is to introduce recursion in a single procedure which calls itself, and also to show a nontrivial verification condition.

We define the function $f91$ as

$$f91 \;=\; \lambda y.\, y > 100 \;\; \texttt{=>} \;\; y - 10 \;\mid\; f91(f91(y + 11)).$$

We claim that the behavior of $f91$ is such that

$$f91 \;=\; \lambda y.\, y > 100 \;\; \texttt{=>} \;\; y - 10 \;\mid\; 91,$$

which is not immediately obvious. Not only is this an interesting partial correctness statement, but the termination of this function is also not easily transparent. We claim that the behavior of $f91$ is such that the value of the expression $101 - y$, where subtraction is restricted to yielding nonnegative values, strictly decreases for every (recursive) call, measured from the state at time of an entrance, to the state at time of recursive entrance.

Here is an expression of the "91" function as a procedure, offered as a goal for the VCG. The following is the actual text submitted to HOL:

```
g [[ program
        procedure p91(var x; val y);
           pre  true;
           post 100 < ^y  =>  x = ^y - 10  |  x = 91;
           calls p91 with 101 - y < 101 - ^y;
           recurses  with 101 - y < ^z;

           if 100 < y then x := y - 10
           else
              p91(x; y + 11);
              p91(x; x)
           fi
        end procedure;

        p91(a; 77)

     end program
     [ a = 91 ]
  ]];;
```

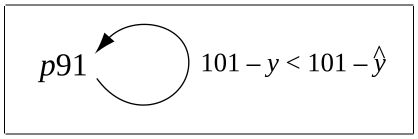


Figure 8.3: Procedure Call Graph for McCarthy's "91" Program.

Now the procedure call graph is given in Figure 8.3. Applying the graph traversal algorithm, beginning at the node $p91$, we generate the call tree in Figure 8.4, with the undiverted recursion verification condition VC1.

Applying `VCG_TAC` to the program correctness goal with the tracing turned on produces the following.

```
#e(VCG_TAC);;
OK..
For procedure `p91`,
```

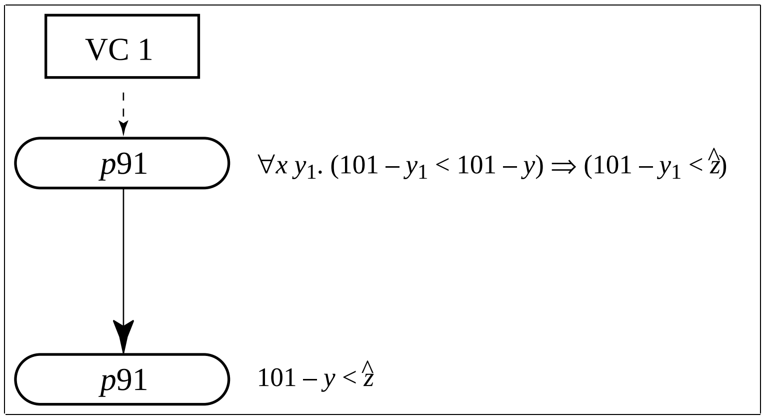

VC 1
$p91$
$\forall x\, y_1.\ (101 - y_1 < 101 - y) \Rightarrow (101 - y_1 < \hat{z})$
$p91$
$101 - y < \hat{z}$

Figure 8.4: Procedure Call Tree for root procedure $p91$.

```
By the "ASSIGN" rule, we have
   [[ {(100 < ^y => y - 10 = ^y - 10 | y - 10 = 91)}
       x := y - 10
      {(100 < ^y => x = ^y - 10 | x = 91)} ]]

By the "CALL" rule, we have
   [[ {(true /\ 101 - x < 101 - ^y) /\
       (!x1 y1. (100 < x => x1 = x - 10 | x1 = 91) ==>
                (100 < ^y => x1 = ^y - 10 | x1 = 91))}
      p91(x;x)
      {(100 < ^y => x = ^y - 10 | x = 91)} ]]

By the "CALL" rule, we have
   [[ {(true /\ 101 - (y + 11) < 101 - ^y) /\
       (!x y1. (100 < y + 11 => x = (y + 11) - 10 | x = 91) ==>
               (true /\ 101 - x < 101 - ^y) /\
               (!x1 y1. (100 < x => x1 = x - 10 | x1 = 91) ==>
                        (100 < ^y => x1 = ^y - 10 | x1 = 91)))}
      p91(x;y + 11)
      {(true /\ 101 - x < 101 - ^y) /\
       (!x1 y1. (100 < x => x1 = x - 10 | x1 = 91) ==>
                (100 < ^y => x1 = ^y - 10 | x1 = 91))} ]]
```

```
By the "SEQ" rule, we have
   [[ {(true /\ 101 - (y + 11) < 101 - ^y) /\
       (!x y1. (100 < y + 11 => x = (y + 11) - 10 | x = 91) ==>
               (true /\ 101 - x < 101 - ^y) /\
               (!x1 y1. (100 < x => x1 = x - 10 | x1 = 91) ==>
                        (100 < ^y => x1 = ^y - 10 | x1 = 91)))}
       p91(x;y + 11); p91(x;x)
      {(100 < ^y => x = ^y - 10 | x = 91)} ]]

By the "IF" rule, we have
   [[ {(100 < y => (100 < ^y => y - 10 = ^y - 10 | y - 10 = 91)
           | (true /\ 101 - (y + 11) < 101 - ^y) /\
             (!x y1. (100 < y + 11 => x = (y + 11) - 10 | x = 91) ==>
                     (true /\ 101 - x < 101 - ^y) /\
                     (!x1 y1. (100 < x => x1 = x - 10 | x1 = 91) ==>
                              (100 < ^y => x1 = ^y - 10 | x1 = 91))))}
       if 100 < y then x := y - 10 else p91(x;y + 11); p91(x;x) fi
      {(100 < ^y => x = ^y - 10 | x = 91)} ]]

By precondition strengthening, we have
   [[ {(^x = x /\ ^y = y /\ true) /\ true}
       if 100 < y then x := y - 10 else p91(x;y + 11); p91(x;x) fi
      {(100 < ^y => x = ^y - 10 | x = 91)} ]]
with additional verification condition
   [[ {(^x = x /\ ^y = y /\ true) /\ true ==>
       (100 < y => (100 < ^y => y - 10 = ^y - 10 | y - 10 = 91)
           | (true /\ 101 - (y + 11) < 101 - ^y) /\
             (!x y1.
               (100 < y + 11 => x = (y + 11) - 10 | x = 91) ==>
               (true /\ 101 - x < 101 - ^y) /\
               (!x1 y1. (100 < x => x1 = x - 10 | x1 = 91) ==>
                        (100 < ^y => x1 = ^y - 10 | x1 = 91))))} ]]

Examining the structure of the procedure call graph:

Traversing the call graph back from the procedure p91:

By the call graph progress from procedure p91 to p91, we have
   [[ {true /\ (!x y1. 101 - y1 < 101 - y ==> 101 - y1 < ^z)}
      p91-<>->p91
      {101 - y < ^z} ]]
```

```
Generating the undiverted recursion verification condition
   [[ {true /\ 101 - y = ^z ==>
       (!x y1. 101 - y1 < 101 - y ==> 101 - y1 < ^z)} ]]

For the main body,

By the "CALL" rule, we have
   [[ {(true /\ true) /\
       (!a y1. (100 < 77 => a = 77 - 10 | a = 91) ==> a = 91)}
       p91(a;77)
      {a = 91} ]]

By precondition strengthening, we have
   [[ {true} p91(a;77) {a = 91} ]]
with additional verification condition
   [[ {true ==> (true /\ true) /\
                (!a y1. (100 < 77 => a = 77 - 10 | a = 91) ==>
                        a = 91)} ]]

3 subgoals
"!a y1. (100 < 77 => (a = 77 - 10) | (a = 91)) ==> (a = 91)"

"!y ^z.
  (101 - y = ^z) ==> (!x y1. (101 - y1) < (101 - y) ==>
                             (101 - y1) < ^z)"

"!^x x ^y y.
  (^x = x) /\ (^y = y) ==>
  (100 < y =>
   (100 < ^y => (y - 10 = ^y - 10) | (y - 10 = 91)) |
   ((101 - (y + 11)) < (101 - ^y) /\
    (!x' y1.
      (100 < (y + 11) => (x' = (y + 11) - 10) | (x' = 91)) ==>
      (101 - x') < (101 - ^y) /\
      (!x1 y1'.
        (100 < x' => (x1 = x' - 10) | (x1 = 91)) ==>
        (100 < ^y => (x1 = ^y - 10) | (x1 = 91))))))"

() : void
```

These three subgoals, in this order, roughly correspond to the following claims:

- The main body is partially correct.
- The value of the recursion expression of the procedure strictly decreases across an undiverted recursion call (VC1).
- The procedure's body is partially correct.

Of these three subgoals, the first two are readily solved. The last verification condition is proven by taking four cases: $y < 90$; $90 \leq y < 100$, $y = 100$, and $y > 100$. This proof has been completed in HOL, yielding the following theorem:

```
|- [[ program procedure p91(x;y);
                global ;
                pre  true;
                post (100 < ^y => x = ^y - 10 | x = 91);
                calls p91 with 101 - y < 101 - ^y;
                recurses with 101 - y < ^z;

                if 100 < y
                   then x := y - 10
                   else p91(x;y + 11); p91(x;x)
                fi
             end procedure; p91(a;77) end program
      [a = 91] ]]
```

8.3 Odd/Even Mutual Recursion

As a third example, we consider the odd/even program presented originally in Table 6.1. The purpose of this example is to demonstrate mutual recursion. We have analyzed this program fairly extensively in the last two chapters in terms of its procedure call graph. Now we will prove it totally correct using `VCG_TAC`.

Here is the odd/even program as a goal for the VCG. The following is the actual text submitted to HOL:

```
g [[ program
        procedure odd(var a; val n);
           pre  true;
           post (?b.^n = 2*b + a) /\ a < 2 /\ n = ^n;
           calls odd  with n < ^n;
           calls even with n < ^n;
           recurses   with n < ^n;

           if n = 0 then a:=0
           else if n = 1 then even(a; n-1)
                         else odd (a; n-2)
                fi
           fi
        end procedure;

        procedure even(var a; val n);
           pre  true;
           post (?b.^n + 1 = 2*b + a) /\ a < 2 /\ n = ^n;
           calls even with n < ^n;
           calls odd  with n < ^n;
           recurses   with n < ^n;

           if n = 0 then a:=1
           else if n = 1 then odd (a; n-1)
                         else even(a; n-2)
                fi
           fi
        end procedure;

        odd(a; 5)

     end program
     [ a = 1 ]
  ]];;
```

Now the procedure call graph is given in Figure 8.5. Applying the graph traversal algorithm, beginning at the node *odd*, we generate the call tree in Figure 8.6, with two undiverted recursion verification conditions, VC1 and VC2, and one diversion verification condition, VC3.

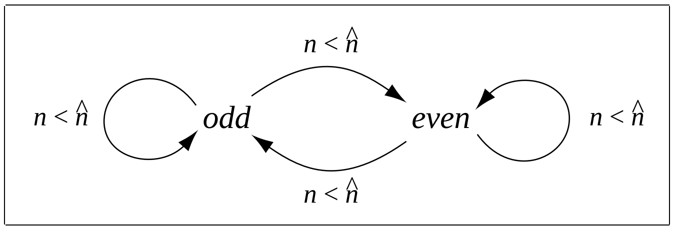


Figure 8.5: Procedure Call Graph for Odd/Even Program.

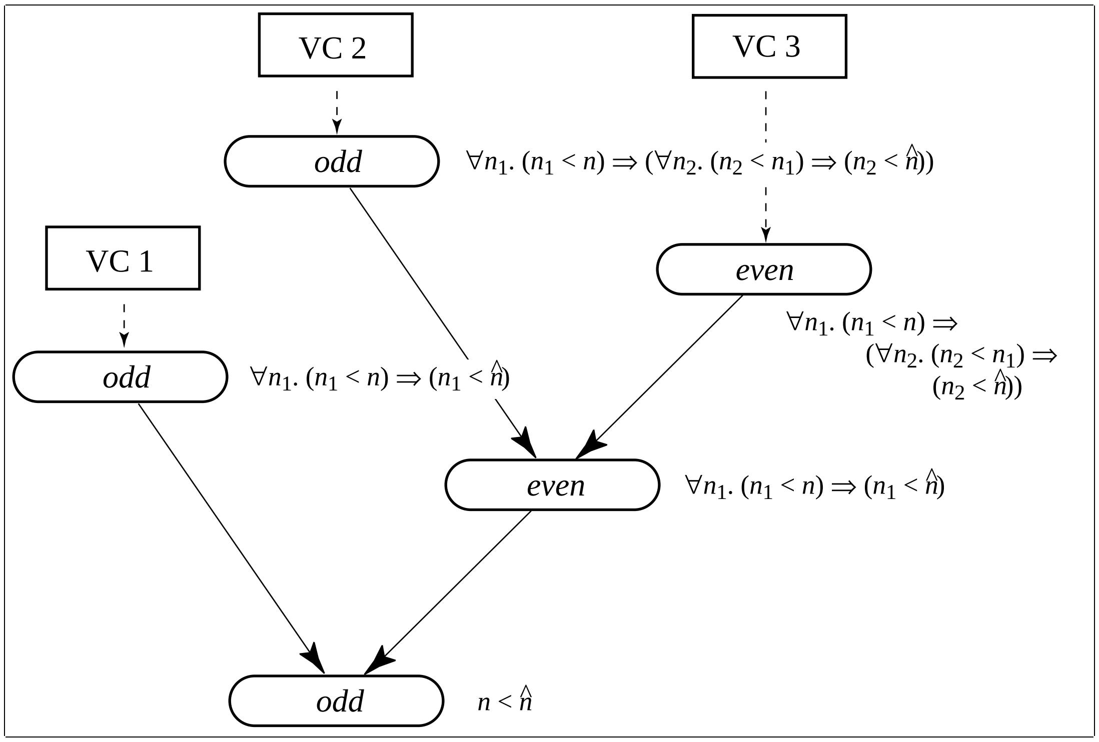

VC 2
VC 3
odd
$\forall n_1. (n_1 < n) \Rightarrow (\forall n_2. (n_2 < n_1) \Rightarrow (n_2 < \hat{n}))$
VC 1
even
$\forall n_1. (n_1 < n) \Rightarrow (\forall n_2. (n_2 < n_1) \Rightarrow (n_2 < \hat{n}))$
odd
$\forall n_1. (n_1 < n) \Rightarrow (n_1 < \hat{n})$
even
$\forall n_1. (n_1 < n) \Rightarrow (n_1 < \hat{n})$
odd
$n < \hat{n}$

Figure 8.6: Procedure Call Tree for root procedure *odd*.

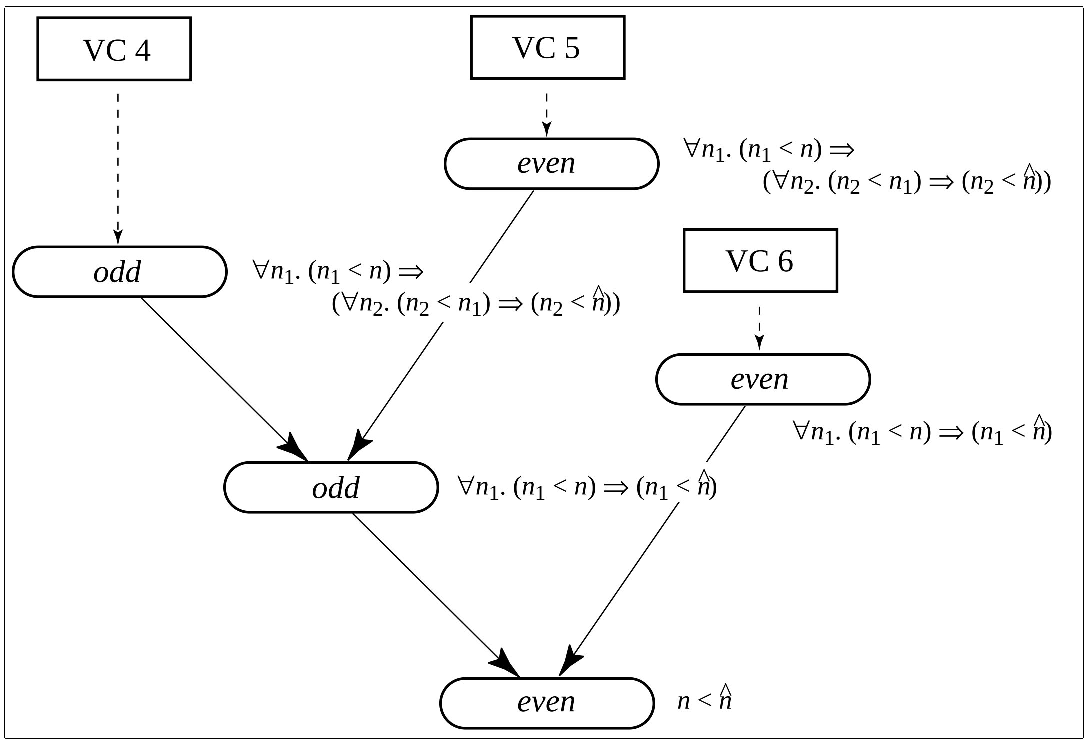

VC 4
VC 5
VC 6
even
odd
even
odd
even
$\forall n_1.\,(n_1 < n) \Rightarrow (\forall n_2.\,(n_2 < n_1) \Rightarrow (n_2 < \hat{n}))$
$\forall n_1.\,(n_1 < n) \Rightarrow (\forall n_2.\,(n_2 < n_1) \Rightarrow (n_2 < \hat{n}))$
$\forall n_1.\,(n_1 < n) \Rightarrow (n_1 < \hat{n})$
$\forall n_1.\,(n_1 < n) \Rightarrow (n_1 < \hat{n})$
$n < \hat{n}$

Figure 8.7: Procedure Call Tree for root procedure *even*.

Applying the graph traversal algorithm, beginning at the node *even*, we generate the call tree in Figure 8.7, with one diversion verification condition, VC4, and two undiverted recursion verification conditions, VC5 and VC6.

Applying `VCG_TAC` to the program correctness goal with the tracing turned on produces the following output. In this example, we are primarily interested in the proof of termination by analyzing the structure of the procedure call graph. This section of the trace follows the line "`Examining the structure of the procedure call graph:`" in the following transcript.

```
#e(VCG_TAC);;
OK..
For procedure `odd`,

By the "ASSIGN" rule, we have
   [[ {(?b. ^n = 2 * b + 0) /\ 0 < 2 /\ n = ^n}
       a := 0
      {(?b. ^n = 2 * b + a) /\ a < 2 /\ n = ^n} ]]

By the "CALL" rule, we have
   [[ {(true /\ n - 1 < ^n) /\
       (!a n2.
         (?b. (n - 1) + 1 = 2 * b + a) /\ a < 2 /\ n2 = n - 1 ==>
         (?b. ^n = 2 * b + a) /\ a < 2 /\ n = ^n)}
       even(a;n - 1)
      {(?b. ^n = 2 * b + a) /\ a < 2 /\ n = ^n} ]]

By the "CALL" rule, we have
   [[ {(true /\ n - 2 < ^n) /\
       (!a n2. (?b. n - 2 = 2 * b + a) /\ a < 2 /\ n2 = n - 2 ==>
               (?b. ^n = 2 * b + a) /\ a < 2 /\ n = ^n)}
       odd(a;n - 2)
      {(?b. ^n = 2 * b + a) /\ a < 2 /\ n = ^n} ]]
```

```
By the "IF" rule, we have
   [[ {(n = 1
          => (true /\ n - 1 < ^n) /\
             (!a n2.
               (?b. (n - 1) + 1 = 2 * b + a) /\
               a < 2 /\
               n2 = n - 1 ==> (?b. ^n = 2 * b + a) /\ a < 2 /\ n = ^n)
           | (true /\ n - 2 < ^n) /\
             (!a n2.
               (?b. n - 2 = 2 * b + a) /\ a < 2 /\ n2 = n - 2 ==>
               (?b. ^n = 2 * b + a) /\ a < 2 /\ n = ^n))}
       if n = 1 then even(a;n - 1) else odd(a;n - 2) fi
      {(?b. ^n = 2 * b + a) /\ a < 2 /\ n = ^n} ]]

By the "IF" rule, we have
   [[ {(n = 0 => (?b. ^n = 2 * b + 0) /\ 0 < 2 /\ n = ^n
           | (n = 1 => (true /\ n - 1 < ^n) /\
                       (!a n2. (?b. (n - 1) + 1 = 2 * b + a) /\
                               a < 2 /\
                               n2 = n - 1 ==>
                               (?b. ^n = 2 * b + a) /\ a < 2 /\ n = ^n)
                 | (true /\ n - 2 < ^n) /\
                   (!a n2. (?b. n - 2 = 2 * b + a) /\
                           a < 2 /\
                           n2 = n - 2 ==>
                           (?b. ^n = 2 * b + a) /\ a < 2 /\ n = ^n)))}
       if n = 0
          then a := 0
          else if n = 1 then even(a;n - 1) else odd(a;n - 2) fi
       fi
      {(?b. ^n = 2 * b + a) /\ a < 2 /\ n = ^n} ]]

By precondition strengthening, we have
   [[ {(^a = a /\ ^n = n /\ true) /\ true}
       if n = 0
          then a := 0
          else if n = 1 then even(a;n - 1) else odd(a;n - 2) fi
       fi
      {(?b. ^n = 2 * b + a) /\ a < 2 /\ n = ^n} ]]
```

```
with additional verification condition
   [[ {(^a = a /\ ^n = n /\ true) /\ true ==>
       (n = 0 => (?b. ^n = 2 * b + 0) /\ 0 < 2 /\ n = ^n
           | (n = 1
                => (true /\ n - 1 < ^n) /\
                   (!a n2. (?b. (n - 1) + 1 = 2 * b + a) /\
                           a < 2 /\
                           n2 = n - 1 ==>
                           (?b. ^n = 2 * b + a) /\ a < 2 /\ n = ^n)
                 | (true /\ n - 2 < ^n) /\
                   (!a n2. (?b. n - 2 = 2 * b + a) /\
                           a < 2 /\
                           n2 = n - 2 ==>
                           (?b. ^n = 2 * b + a) /\
                           a < 2 /\
                           n = ^n)))} ]]

For procedure `even`,

By the "ASSIGN" rule, we have
   [[ {(?b. ^n + 1 = 2 * b + 1) /\ 1 < 2 /\ n = ^n}
       a := 1
      {(?b. ^n + 1 = 2 * b + a) /\ a < 2 /\ n = ^n} ]]

By the "CALL" rule, we have
   [[ {(true /\ n - 1 < ^n) /\
       (!a n2. (?b. n - 1 = 2 * b + a) /\ a < 2 /\ n2 = n - 1 ==>
               (?b. ^n + 1 = 2 * b + a) /\ a < 2 /\ n = ^n)}
       odd(a;n - 1)
      {(?b. ^n + 1 = 2 * b + a) /\ a < 2 /\ n = ^n} ]]

By the "CALL" rule, we have
   [[ {(true /\ n - 2 < ^n) /\
       (!a n2.
         (?b. (n - 2) + 1 = 2 * b + a) /\ a < 2 /\ n2 = n - 2 ==>
         (?b. ^n + 1 = 2 * b + a) /\ a < 2 /\ n = ^n)}
       even(a;n - 2)
      {(?b. ^n + 1 = 2 * b + a) /\ a < 2 /\ n = ^n} ]]
```

```
By the "IF" rule, we have
   [[ {(n = 1 => (true /\ n - 1 < ^n) /\
                 (!a n2.
                   (?b. n - 1 = 2 * b + a) /\ a < 2 /\ n2 = n - 1 ==>
                   (?b. ^n + 1 = 2 * b + a) /\ a < 2 /\ n = ^n)
           | (true /\ n - 2 < ^n) /\
             (!a n2. (?b. (n - 2) + 1 = 2 * b + a) /\
                     a < 2 /\
                     n2 = n - 2 ==>
                     (?b. ^n + 1 = 2 * b + a) /\ a < 2 /\ n = ^n))}
       if n = 1 then odd(a;n - 1) else even(a;n - 2) fi
      {(?b. ^n + 1 = 2 * b + a) /\ a < 2 /\ n = ^n} ]]

By the "IF" rule, we have
   [[ {(n = 0 => (?b. ^n + 1 = 2 * b + 1) /\ 1 < 2 /\ n = ^n
           | (n = 1 => (true /\ n - 1 < ^n) /\
                       (!a n2.
                         (?b. n - 1 = 2 * b + a) /\
                         a < 2 /\
                         n2 = n - 1 ==>
                         (?b. ^n + 1 = 2 * b + a) /\ a < 2 /\ n = ^n)
                 | (true /\ n - 2 < ^n) /\
                   (!a n2.
                     (?b. (n - 2) + 1 = 2 * b + a) /\
                     a < 2 /\
                     n2 = n - 2 ==>
                     (?b. ^n + 1 = 2 * b + a) /\ a < 2 /\ n = ^n)))}
       if n = 0
          then a := 1
          else if n = 1 then odd(a;n - 1) else even(a;n - 2) fi
       fi
      {(?b. ^n + 1 = 2 * b + a) /\ a < 2 /\ n = ^n} ]]

By precondition strengthening, we have
   [[ {(^a = a /\ ^n = n /\ true) /\ true}
       if n = 0
          then a := 1
          else if n = 1 then odd(a;n - 1) else even(a;n - 2) fi
       fi
      {(?b. ^n + 1 = 2 * b + a) /\ a < 2 /\ n = ^n} ]]
```

```
with additional verification condition
   [[ {(^a = a /\ ^n = n /\ true) /\ true ==>
       (n = 0 => (?b. ^n + 1 = 2 * b + 1) /\ 1 < 2 /\ n = ^n
           | (n = 1 => (true /\ n - 1 < ^n) /\
                     (!a n2.
                       (?b. n - 1 = 2 * b + a) /\
                       a < 2 /\
                       n2 = n - 1 ==>
                       (?b. ^n + 1 = 2 * b + a) /\ a < 2 /\ n = ^n)
                 | (true /\ n - 2 < ^n) /\
                   (!a n2.
                     (?b. (n - 2) + 1 = 2 * b + a) /\
                     a < 2 /\ n2 = n - 2 ==>
                     (?b. ^n + 1 = 2 * b + a) /\ a < 2 /\ n = ^n)))} ]]

Examining the structure of the procedure call graph:

Traversing the call graph back from the procedure even:

By the call graph progress from procedure even to even, we have
   [[ {true /\ (!a n1. n1 < n ==> n1 < ^n)} even-<>->even {n < ^n} ]]

Generating the undiverted recursion verification condition
   [[ {true /\ n = ^n ==> (!a n1. n1 < n ==> n1 < ^n)} ]]

By the call graph progress from procedure odd to even, we have
   [[ {true /\ (!a n1. n1 < n ==> n1 < ^n)} odd-<>->even {n < ^n} ]]

By the call graph progress from procedure even to odd, we have
   [[ {true /\ (!a n1. n1 < n ==> (!a n2. n2 < n1 ==> n2 < ^n))}
      even-<>->odd
      {!a n1. n1 < n ==> n1 < ^n} ]]

Generating the undiverted recursion verification condition
   [[ {true /\ n = ^n ==>
       (!a n1. n1 < n ==> (!a n2. n2 < n1 ==> n2 < ^n))} ]]

By the call graph progress from procedure odd to odd, we have
   [[ {true /\ (!a n1. n1 < n ==> (!a n2. n2 < n1 ==> n2 < ^n))}
      odd-<>->odd
      {!a n1. n1 < n ==> n1 < ^n} ]]
```

```
Generating the diversion verification condition
   [[ {(!a n1. n1 < n ==> n1 < ^n) ==>
       (!a n1. n1 < n ==> (!a n2. n2 < n1 ==> n2 < ^n))} ]]

Traversing the call graph back from the procedure odd:


By the call graph progress from procedure even to odd, we have
   [[ {true /\ (!a n1. n1 < n ==> n1 < ^n)} even-<>->odd {n < ^n} ]]

By the call graph progress from procedure even to even, we have
   [[ {true /\ (!a n1. n1 < n ==> (!a n2. n2 < n1 ==> n2 < ^n))}
      even-<>->even
      {!a n1. n1 < n ==> n1 < ^n} ]]

Generating the diversion verification condition
   [[ {(!a n1. n1 < n ==> n1 < ^n) ==>
       (!a n1. n1 < n ==> (!a n2. n2 < n1 ==> n2 < ^n))} ]]

By the call graph progress from procedure odd to even, we have
   [[ {true /\ (!a n1. n1 < n ==> (!a n2. n2 < n1 ==> n2 < ^n))}
      odd-<>->even
      {!a n1. n1 < n ==> n1 < ^n} ]]

Generating the undiverted recursion verification condition
   [[ {true /\ n = ^n ==>
       (!a n1. n1 < n ==> (!a n2. n2 < n1 ==> n2 < ^n))} ]]

By the call graph progress from procedure odd to odd, we have
   [[ {true /\ (!a n1. n1 < n ==> n1 < ^n)} odd-<>->odd {n < ^n} ]]

Generating the undiverted recursion verification condition
   [[ {true /\ n = ^n ==> (!a n1. n1 < n ==> n1 < ^n)} ]]

For the main body,

By the "CALL" rule, we have
   [[ {(true /\ true) /\
       (!a n1. (?b. 5 = 2 * b + a) /\ a < 2 /\ n1 = 5 ==> a = 1)}
       odd(a;5)
      {a = 1} ]]
```

```
By precondition strengthening, we have
   [[ {true} odd(a;5) {a = 1} ]]
with additional verification condition
   [[ {true ==>
       (true /\ true) /\
       (!a n1. (?b. 5 = 2 * b + a) /\ a < 2 /\ n1 = 5 ==> a = 1)} ]]

9 subgoals
"!a n1. (?b. 5 = (2 * b) + a) /\ a < 2 /\ (n1 = 5) ==> (a = 1)"

"!n ^n. (n = ^n) ==> (!a n1. n1 < n ==> n1 < ^n)"

"!n ^n. (n = ^n) ==> (!a n1. n1 < n ==> (!a' n2. n2 < n1 ==> n2 < ^n))"

"!n ^n.
  (!a n1. n1 < n ==> n1 < ^n) ==>
  (!a n1. n1 < n ==> (!a' n2. n2 < n1 ==> n2 < ^n))"

"!n ^n.
  (!a n1. n1 < n ==> n1 < ^n) ==>
  (!a n1. n1 < n ==> (!a' n2. n2 < n1 ==> n2 < ^n))"

"!n ^n. (n = ^n) ==> (!a n1. n1 < n ==> (!a' n2. n2 < n1 ==> n2 < ^n))"

"!n ^n. (n = ^n) ==> (!a n1. n1 < n ==> n1 < ^n)"

"!^a a ^n n.
  (^a = a) /\ (^n = n) ==>
  ((n = 0) =>
   ((?b. ^n + 1 = (2 * b) + 1) /\ 1 < 2 /\ (n = ^n)) |
   ((n = 1) =>
    ((n - 1) < ^n /\
     (!a' n2.
       (?b. n - 1 = (2 * b) + a') /\ a' < 2 /\ (n2 = n - 1) ==>
       (?b. ^n + 1 = (2 * b) + a') /\ a' < 2 /\ (n = ^n))) |
    ((n - 2) < ^n /\
     (!a' n2.
       (?b. (n - 2) + 1 = (2 * b) + a') /\ a' < 2 /\ (n2 = n - 2) ==>
       (?b. ^n + 1 = (2 * b) + a') /\ a' < 2 /\ (n = ^n)))))"
```

```
"!^a a ^n n.
  (^a = a) /\ (^n = n) ==>
  ((n = 0) =>
   ((?b. ^n = (2 * b) + 0) /\ 0 < 2 /\ (n = ^n)) |
   ((n = 1) =>
    ((n - 1) < ^n /\
     (!a' n2.
       (?b. (n - 1) + 1 = (2 * b) + a') /\ a' < 2 /\ (n2 = n - 1) ==>
       (?b. ^n = (2 * b) + a') /\ a' < 2 /\ (n = ^n))) |
    ((n - 2) < ^n /\
     (!a' n2.
       (?b. n - 2 = (2 * b) + a') /\ a' < 2 /\ (n2 = n - 2) ==>
       (?b. ^n = (2 * b) + a') /\ a' < 2 /\ (n = ^n)))))"

() : void
```

These nine subgoals, in this order, roughly correspond to the following claims:

- The main body is partially correct.
- The value of the recursion expression of the procedure *odd* strictly decreases across the undiverted recursion path $odd \rightarrow odd$ (VC1).
- The value of the recursion expression of the procedure *odd* strictly decreases across the undiverted recursion path $odd \rightarrow even \rightarrow odd$ (VC2).
- The diversion of *even* in $even \rightarrow even \rightarrow odd$ does not interfere with the recursive progress of the procedure *odd* (VC3).
- The diversion of *odd* in $odd \rightarrow odd \rightarrow even$ does not interfere with the recursive progress of the procedure *even* (VC4).
- The value of the recursion expression of the procedure *even* strictly decreases across the undiverted recursion path $even \rightarrow odd \rightarrow even$ (VC5).

- The value of the recursion expression of the procedure *even* strictly decreases across the undiverted recursion path $even \rightarrow even$ (VC6).

- The body of procedure *even* is partially correct.

- The body of procedure *odd* is partially correct.

Of these nine subgoals, three have to do with syntactic structure partial correctness, four have to do with undiverted recursion, and two have to do with diversions.

All of these subgoals are readily solved. This proof has been completed in HOL, yielding the following theorem:

```
|- [[ program
        procedure odd(a;n);
           global ;
           pre  true;
           post (?b. ^n = 2 * b + a) /\ a < 2 /\ n = ^n;
           calls odd with n < ^n;
           calls even with n < ^n;
           recurses with n < ^n;

           if n = 0
              then a := 0
              else if n = 1
                      then even(a;n - 1)
                      else odd(a;n - 2)
                   fi
           fi
        end procedure;
        procedure even(a;n);
           global ;
           pre  true;
           post (?b. ^n + 1 = 2 * b + a) /\ a < 2 /\ n = ^n;
           calls even with n < ^n;
           calls odd with n < ^n;
           recurses with n < ^n;

           if n = 0
              then a := 1
              else if n = 1
                      then odd(a;n - 1)
                      else even(a;n - 2)
                   fi
           fi
        end procedure;

        odd(a;5)
     end program
     [a = 1] ]]
```

8.4 Pandya and Joseph's Product Procedures

In 1986, Pandya and Joseph described a new rule for the total correctness of procedure calls, improving on the earlier proposal of Sokołowski. Sokołowski used a recursion depth counter to track the current depth of each call, and required the counter to decrease by exactly one for every call of every procedure. This supported the proof of the termination of procedures, because it did not allow infinite recursive descent. However, Pandya and Joseph showed how even for simple programs, the use of Sokołowski's rule could lead to the use of predicates which were complex and non-intuitive. They eased Sokołowski's requirement that the recursion depth counter decrease by one for *every* call, by choosing a subset of the procedures as "header" procedures. Then the recursion depth counter was required to decrease by one only for calls of header procedures, not the others.

Pandya and Joseph state that this leads to proofs which are simpler and more intuitive, reducing the programmer's burden of encoding information about the number of iterations into the recursion depth counter. This does not eliminate the burden, however, but simply reduces the number of procedures whose calls must be counted.

The new rule they proposed they classified as syntax-directed, as opposed to data-directed. A data-directed rule reasons about the full semantics of the state of the program, and the values of all variables. A syntax-directed rule, on the other hand, reasons about an object which is syntactically built of subcomponents by assembling the proofs about the components. Syntax-directed reasoning is significantly simpler than data-directed reasoning, if it is semantically valid.

We have taken this idea further, and have introduced rules that deal with the structure of the procedure call graph, and not only the syntax of the program. This provides even more structure to organize the proof of termination of the procedures, and eliminates the need for recursion depth counters.

To illustrate their arguments, Pandya and Joseph have presented an algorithm using three procedures to compute the product of two numbers. In this section, we will present the program (see Figure 8.8) and their proof, and then show how we would prove the program in our system with equal ease. Actually, the proof they present is not complete, but takes the form of a proof skeleton, where the program is shown annotated with assertions between commands that show the conditions that are true at each point in the control structure. We will likewise present such a proof skeleton. We had originally hoped to present an automated proof like the other examples in this chapter, but the example program that Pandya and Joseph present contains several operators, predicates *even* and *odd* and binary operator **div** to compute integer division, which we have not yet included in the Sunrise language. In the future we expect to add these, and then run the example completely through. For now we offer a proof skeleton constructed by hand.

In Figure 8.8 we see the three procedures of this program. The purpose of this program is to multiply two numbers a and b and leave the result in variable z. None of these procedures takes any parameters, but instead they communicate through global variables, as Pandya and Joseph designed them. The procedure *product* tests y to see if it is even or odd, and calls *evenproduct* or *oddproduct* accordingly to perform the multiplication. *oddproduct* reduces the problem to an "even" situation by subtracting one from y and simultaneously adding x to z,

```
procedure $product(;)$;
    global  $x, y, z, a, b$;
    pre     $z + x * y = a * b$;
    post    $z = a * b$;

    if $even(y)$  then $evenproduct(;)$
                  else  $oddproduct(;)$
    fi
end procedure;

procedure $oddproduct(;)$;
    global  $x, y, z, a, b$;
    pre     $z + x * y = a * b \wedge odd(y)$;
    post    $z = a * b$;

    $y := y - 1$;
    $z := z + x$;
    $evenproduct(;)$
end procedure;

procedure $evenproduct(;)$;
    global  $x, y, z, a, b$;
    pre     $z + x * y = a * b \wedge even(y)$;
    post    $z = a * b$;

    if $y = 0$ then skip
    else  $x := 2 * x$;
          $y := y$ div $2$;
          $product(;)$
    fi
end procedure;
```

Figure 8.8: Pandya and Joseph's Product Procedures.

and then calls *evenproduct* to complete the multiplication. *evenproduct* in turn tests y to see if it is zero; if it is, then the multiplication is complete and the procedure terminates; otherwise, if y is not zero, then *evenproduct* reduces the problem to a "lesser" situation by dividing y by 2 and multiplying x by 2, at which point *evenproduct* calls *product* on the "lesser" situation.

Using one of the more traditional approaches such as Sokołowski's, Pandya and Joseph have shown that one would need to encode the depth of recursion in a predicate which was quite complex, even for this simple example. They could only find a recursive form for it, and even that was only an approximate estimate of the depth of recursion. They then presented their method of only requiring the depth counter to decrease for header procedures. Taking in this example the header procedures to consist solely of the procedure *product*, they present the proof skeleton given in Figures 8.9 and 8.10, with boxes enclosing assertions.

```
procedure $product(;)$;
    global  $x, y, z, a, b$;
    pre     $q_p(i): \; z + x * y = a * b \;\wedge\; y \leq i$;
    post    $z = a * b$;

    if $even(y)$  then  $\boxed{z + x * y = a * b \;\wedge\; y \leq i \;\wedge\; even(y)} \ldots \boxed{q_e(i)}$
                        $evenproduct(;)$
                        $\boxed{z = a * b}$
                  else  $\boxed{z + x * y = a * b \;\wedge\; y \leq i \;\wedge\; odd(y)} \ldots \boxed{q_o(i)}$
                        $oddproduct(;)$
                        $\boxed{z = a * b}$
    fi
end procedure;
```

Figure 8.9: Pandya and Joseph's Proof Skeleton for procedure *product*.

```
**procedure** $oddproduct(;);$
    **global**  $x, y, z, a, b;$
    **pre**     $q_o(i):\ z + x * y = a * b \ \wedge\ y \le i \ \wedge\ odd(y);$
    **post**    $z = a * b;$

    $y := y - 1;$
    $z := z + x;$
    $\boxed{z + x * y = a * b \ \wedge\ y \le i \ \wedge\ even(y)} \ldots \boxed{q_e(i)}$
    $evenproduct(;)$
    $\boxed{z = a * b}$
**end procedure**;

**procedure** $evenproduct(;);$
    **global**  $x, y, z, a, b;$
    **pre**     $q_e(i):\ z + x * y = a * b \ \wedge\ y \le i \ \wedge\ even(y);$
    **post**    $z = a * b;$

    **if** $y = 0$
    **then**  $\boxed{z + x * y = a * b \ \wedge\ y = 0}$
          **skip**
          $\boxed{z = a * b}$
    **else**  $\boxed{z + x * y = a * b \ \wedge\ 0 < y \le i \ \wedge\ even(y)}$

          $\boxed{z + x * y = a * b \ \wedge\ (y \textbf{ div } 2) \le i - 1}$
          $x := 2 * x;$
          $y := y \textbf{ div } 2;$
          $\boxed{z + x * y = a * b \ \wedge\ y \le i - 1} \ldots \boxed{q_p(i - 1)}$
          $product(;)$
          $\boxed{z = a * b}$
    **fi**
**end procedure**;
```

Figure 8.10: Pandya and Joseph's Proof Skeletons for procedures *oddproduct* and *evenproduct*.

To motivate this proof, Pandya and Joseph state

> 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.

Pandya and Joseph leave it to the reader to verify this annotated proof skeleton, and we will do the same. They presented a proof of its termination, using a mathematical induction argument based on the value of i. Their rule depended on the existence of predicates $q_k(i)$, for which the variable i is the recursion depth counter, here only counting calls to the header procedure *product*. Pandya and Joseph's argument is that one can prove $y \leq i$, which is far more natural and a great improvement over the expression which would arise from making i a counter of all procedure calls. However, in our version we can eliminate the use of i entirely, and thus our system is even simpler and more natural, and at the same time more general.

We present our annotated proof skeleton of this program in Figures 8.11 and 8.12. Since this is a hand proof, we have performed some obvious simplifications to clarify the formulas.

```
procedure $product(;)$;
        global   $x, y, z, a, b$;
        pre      $z + x * y = a * b$;
        post     $z = a * b$;
        calls  $evenproduct$   with $y = \widehat{y}$;
        calls  $oddproduct$    with $y = \widehat{y}$;
        recurses               with $y < \widehat{y}$;

       $\boxed{\begin{array}{lcl} even(y) & => & z + x * y = a * b \ \wedge \ even(y) \ \wedge \ y = \widehat{y} \\ & | & z + x * y = a * b \ \wedge \ odd(y) \ \wedge \ y = \widehat{y} \end{array}}$

        if $even(y)$  then  $\boxed{z + x * y = a * b \ \wedge \ even(y) \ \wedge \ y = \widehat{y}}$
                            $evenproduct(;)$
                            $\boxed{z = a * b}$

                      else  $\boxed{z + x * y = a * b \ \wedge \ odd(y) \ \wedge \ y = \widehat{y}}$
                            $oddproduct(;)$
                            $\boxed{z = a * b}$
        fi
end procedure;

procedure $oddproduct(;)$;
        global   $x, y, z, a, b$;
        pre      $z + x * y = a * b \ \wedge \ odd(y)$;
        post     $z = a * b$;
        calls  $evenproduct$   with $y < \widehat{y}$;
        recurses               with $y < \widehat{y}$;

        $\boxed{(z + x) + x * (y - 1) = a * b \ \wedge \ even(y - 1) \ \wedge \ y - 1 < \widehat{y}}$
        $y := y - 1$;
        $z := z + x$;
        $\boxed{z + x * y = a * b \ \wedge \ even(y) \ \wedge \ y < \widehat{y}}$
        $evenproduct(;)$
        $\boxed{z = a * b}$
end procedure;
```

Figure 8.11: Sunrise Proof Skeletons for procedures *product* and *oddproduct*.

```
procedure $evenproduct(;);$
        global   $x, y, z, a, b;$
        pre      $z + x * y = a * b \ \wedge \ even(y);$
        post     $z = a * b;$
        calls  $product$   with $y < \widehat{y};$
        recurses           with $y < \widehat{y};$

        $\boxed{\begin{array}{lll} y = 0 & => & z = a * b \\ & | & z + (2 * x) * (y \textbf{ div } 2) = a * b \ \wedge \ y \textbf{ div } 2 < \widehat{y} \end{array}}$
        if $y = 0$
        then  $\boxed{z = a * b}$
              skip
              $\boxed{z = a * b}$
        else  $\boxed{z + (2 * x) * (y \textbf{ div } 2) = a * b \ \wedge \ y \textbf{ div } 2 < \widehat{y}}$
              $x := 2 * x;$
              $y := y$ div $2;$
              $\boxed{z + x * y = a * b \ \wedge \ y < \widehat{y}}$
              $product(;)$
              $\boxed{z = a * b}$
        fi
end procedure;
```

Figure 8.12: Sunrise Proof Skeleton for procedure *evenproduct*.

The analysis of the syntax of these three procedures generates three verification conditions, for the partial correctness of each body. These verification conditions are

1. $z + x * y = a * b \ \wedge \ y = \widehat{y} \ \Rightarrow$
   $(\ even(y) \quad => \quad z + x * y = a * b \ \wedge \ even(y) \ \wedge \ y = \widehat{y}$
   $\qquad\qquad\quad | \qquad z + x * y = a * b \ \wedge \ odd(y) \ \wedge \ y = \widehat{y} \ \ )$

2. $z + x * y = a * b \ \wedge \ odd(y) \ \wedge \ y = \widehat{y} \ \Rightarrow$
   $(z + x) + x * (y - 1) = a * b \ \wedge \ even(y - 1) \ \wedge \ y - 1 < \widehat{y}$

3. $z + x * y = a * b \ \wedge \ even(y) \ \wedge \ y = \widehat{y} \ \Rightarrow$
   $(\ y = 0 \quad => \quad z = a * b$
   $\qquad\qquad | \qquad z + (2 * x) * (y \ \mathbf{div} \ 2) = a * b \ \wedge \ y \ \mathbf{div} \ 2 < \widehat{y} \ \ )$

for the partial correctness of the bodies of *product*, *oddproduct*, and *evenproduct*, respectively.

The procedure call graph for Pandya and Joseph's product program is given in Figure 8.13.

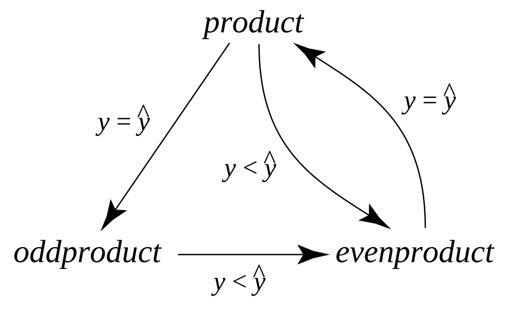
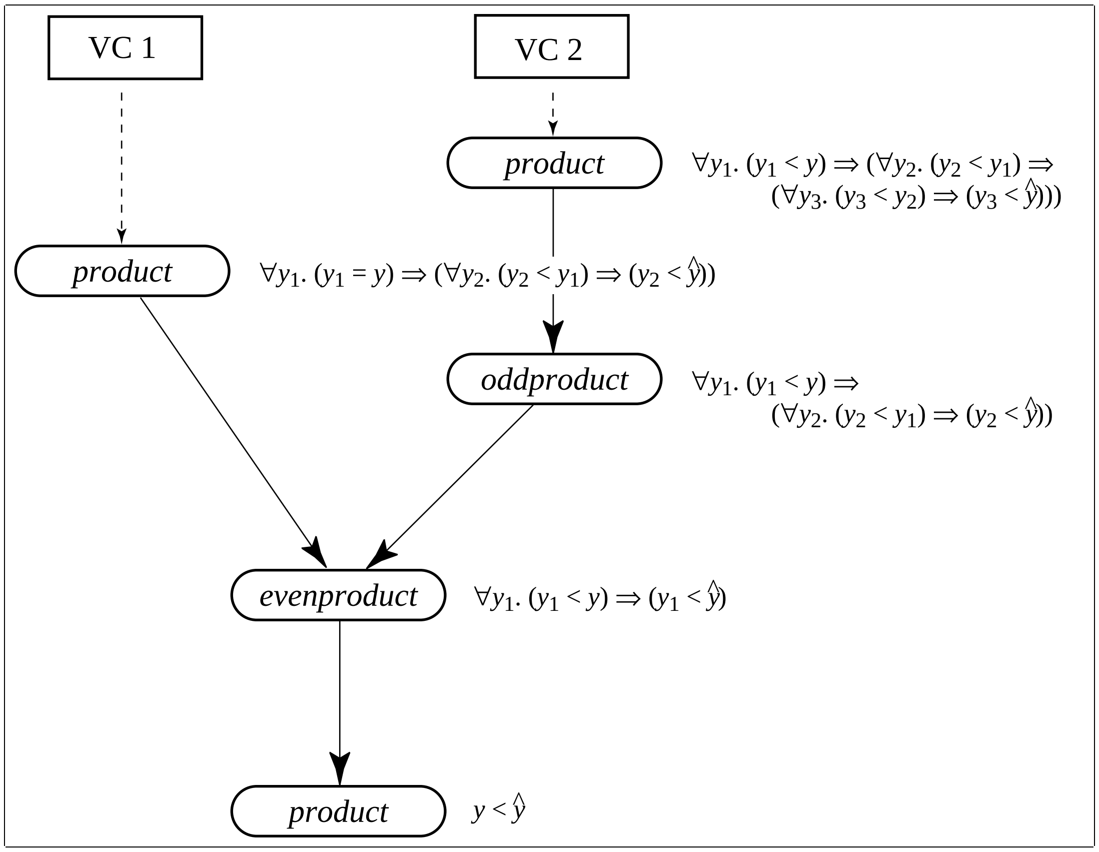


Figure 8.13: Procedure Call Graph for Pandya and Joseph's Product Program.

Applying the graph traversal algorithm, beginning at the node *product*, we generate the call tree in Figure 8.14, with the following two undiverted recursion verification conditions, VC1 and VC2.



VC 1
VC 2
product
$\forall y_1. (y_1 < y) \Rightarrow (\forall y_2. (y_2 < y_1) \Rightarrow (\forall y_3. (y_3 < y_2) \Rightarrow (y_3 < \hat{y})))$
product
$\forall y_1. (y_1 = y) \Rightarrow (\forall y_2. (y_2 < y_1) \Rightarrow (y_2 < \hat{y}))$
oddproduct
$\forall y_1. (y_1 < y) \Rightarrow (\forall y_2. (y_2 < y_1) \Rightarrow (y_2 < \hat{y}))$
evenproduct
$\forall y_1. (y_1 < y) \Rightarrow (y_1 < \hat{y})$
product
$y < \hat{y}$

Figure 8.14: Procedure Call Tree for root procedure *product*.

VC1: $z + x * y = a * b \ \wedge \ y = \widehat{y} \ \Rightarrow$
$(\forall y_1.\ y_1 = y \Rightarrow (\forall y_2.\ y_2 < y_1 \Rightarrow y_2 < \widehat{y}))$

VC2: $z + x * y = a * b \ \wedge \ y = \widehat{y} \ \Rightarrow$
$(\forall y_1.\ y_1 = y \Rightarrow (\forall y_2.\ y_2 < y_1 \Rightarrow (\forall y_3.\ y_3 < y_2 \Rightarrow y_3 < \widehat{y})))$

Applying the graph traversal algorithm, beginning at the node *oddproduct*, we generate the call tree in Figure 8.15, generating one diversion verification condition, VC3, and one undiverted recursion verification condition, VC4:

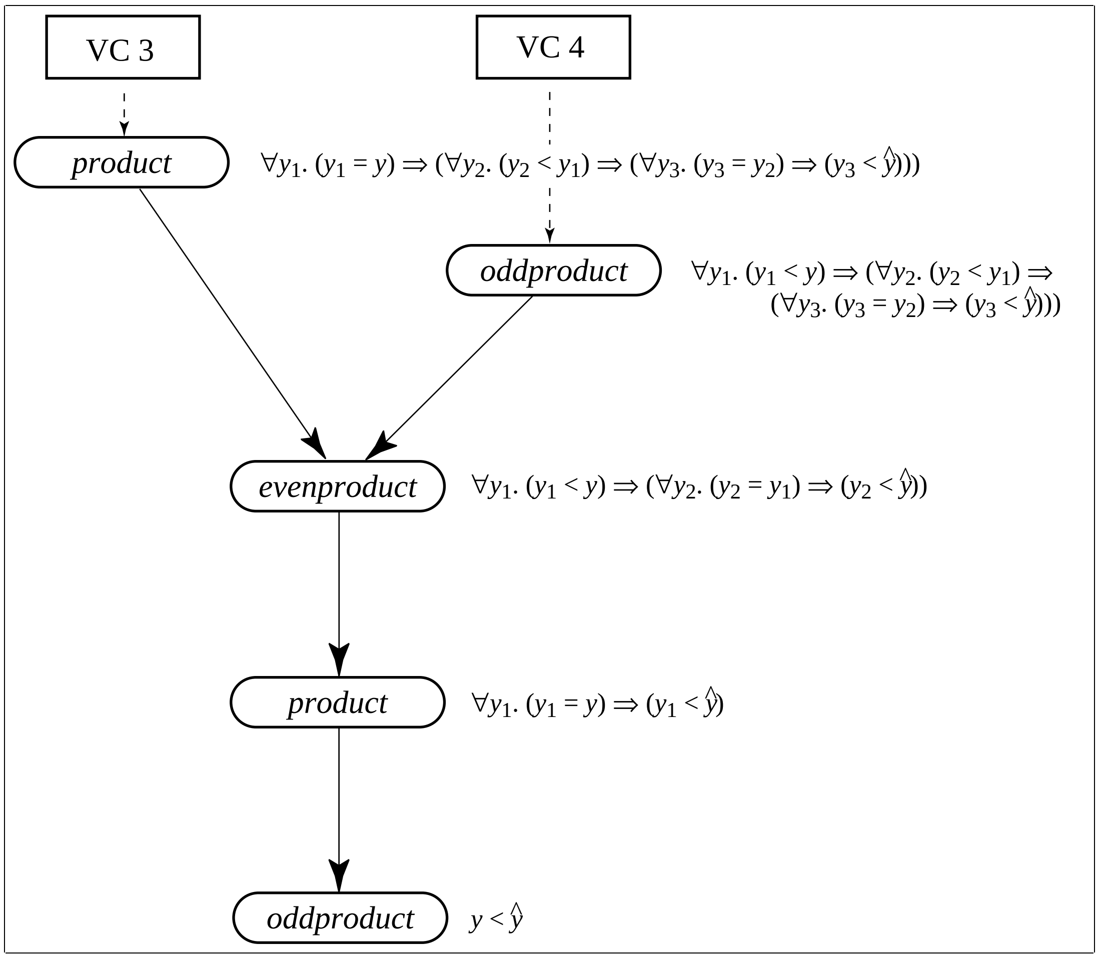
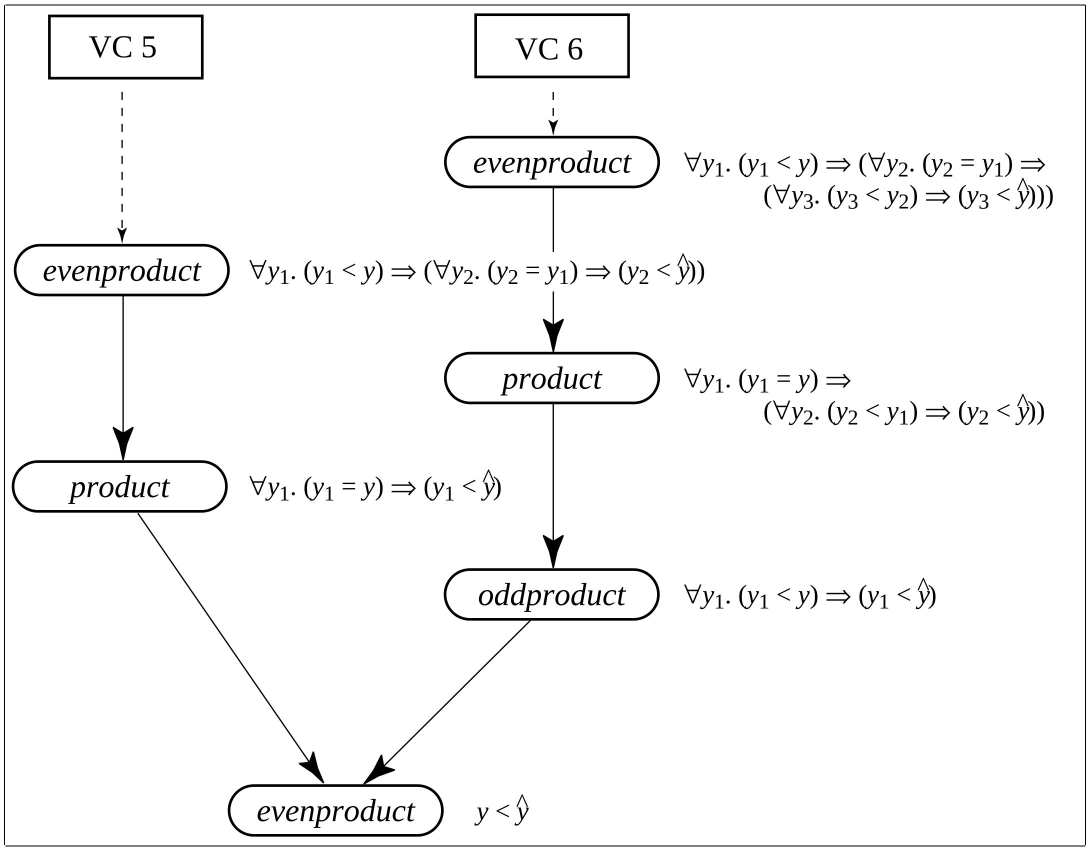

VC 3
VC 4
product
$\forall y_1.\ (y_1 = y) \Rightarrow (\forall y_2.\ (y_2 < y_1) \Rightarrow (\forall y_3.\ (y_3 = y_2) \Rightarrow (y_3 < \hat{y})))$
oddproduct
$\forall y_1.\ (y_1 < y) \Rightarrow (\forall y_2.\ (y_2 < y_1) \Rightarrow (\forall y_3.\ (y_3 = y_2) \Rightarrow (y_3 < \hat{y})))$
evenproduct
$\forall y_1.\ (y_1 < y) \Rightarrow (\forall y_2.\ (y_2 = y_1) \Rightarrow (y_2 < \hat{y}))$
product
$\forall y_1.\ (y_1 = y) \Rightarrow (y_1 < \hat{y})$
oddproduct
$y < \hat{y}$

Figure 8.15: Procedure Call Tree for root procedure *oddproduct*.

$$\text{VC3:}\quad (\forall y_1.\ y_1 = y \Rightarrow y_1 < \hat{y}) \ \Rightarrow$$
$$(\forall y_1.\ y_1 = y \Rightarrow (\forall y_2.\ y_2 < y_1 \Rightarrow (\forall y_3.\ y_3 = y_2 \Rightarrow y_3 < \hat{y})))$$

$$\text{VC4:}\quad z + x * y = a * b \ \wedge \ odd(y) \ \wedge \ y = \hat{y} \ \Rightarrow$$
$$(\forall y_1.\ y_1 < y \Rightarrow (\forall y_2.\ y_2 < y_1 \Rightarrow (\forall y_3.\ y_3 = y_2 \Rightarrow y_3 < \hat{y})))$$

Applying the graph traversal algorithm, beginning at the node *evenproduct*, we generate the call tree in Figure 8.16, generating the following two undiverted recursion verification conditions, VC5 and VC6.



VC 5
VC 6
evenproduct
∀y1. (y1 < y) ⇒ (∀y2. (y2 = y1) ⇒ (∀y3. (y3 < y2) ⇒ (y3 < ŷ)))
evenproduct
∀y1. (y1 < y) ⇒ (∀y2. (y2 = y1) ⇒ (y2 < ŷ))
product
∀y1. (y1 = y) ⇒ (∀y2. (y2 < y1) ⇒ (y2 < ŷ))
product
∀y1. (y1 = y) ⇒ (y1 < ŷ)
oddproduct
∀y1. (y1 < y) ⇒ (y1 < ŷ)
evenproduct
y < ŷ

Figure 8.16: Procedure Call Tree for root procedure *evenproduct*.

VC5: $z + x * y = a * b \ \wedge \ even(y) \ \wedge \ y = \widehat{y} \ \Rightarrow$
$(\forall y_1.\ y_1 < y \Rightarrow (\forall y_2.\ y_2 = y_1 \Rightarrow y_2 < \widehat{y})))$

VC6: $z + x * y = a * b \ \wedge \ even(y) \ \wedge \ y = \widehat{y} \ \Rightarrow$
$(\forall y_1.\ y_1 < y \Rightarrow (\forall y_2.\ y_2 = y_1 \Rightarrow (\forall y_3.\ y_3 < y_2 \Rightarrow y_3 < \widehat{y})))$

All of these verification conditions are readily proved. This completes our proof of Pandya and Joseph's Product Procedures example.

8.5 Cycling Termination

As a fifth example, we choose a program specifically to show the strengths of our approach to proving programs correct. The program has two mutually recursive procedures, like the odd/even program, but here there is a difference in the measurable progress across the various arcs of the call graph. In particular, across one of the arcs of the call graph, there is no progress at all, in that the state does not change. This would pose difficulties for the other methods of proving termination, because they expect that a recursion depth counter would decrease for every call. Even Pandya and Joseph's system, which we believe to be the strongest of the previous systems, would not help here, as there is no identifiable set of header procedures as a proper subset of all procedures. In Pandya and Joseph's system, we must then take all procedures as the header procedures, and thus we would devolve essentially to Sokołowski's method.

We call this example "Cycling Termination," first because the only issue is termination (no interesting result is computed), and second because the structure of the call graph reminds us of a bicycle, with its two wheels and the chain that transfers power from the pedals to the rear wheel. This is not an inappropriate analogy for this program, if one might imagine a bicycle with one pedal damaged so that it could not support any pressure. When pedaling such a bicycle, one would need to thrust hard when the good pedal was moving downward, but then would exert no force while it was moving upwards again, and in fact would coast during this period, depending solely on the momentum generated by the other phase to propel you to the goal. [1] This corresponds to the progress we will see

[1] We are grateful to Prof. D. Stott Parker for his recollection of such a damaged bicycle.

attached to the various arcs of the procedure call graph for this program.

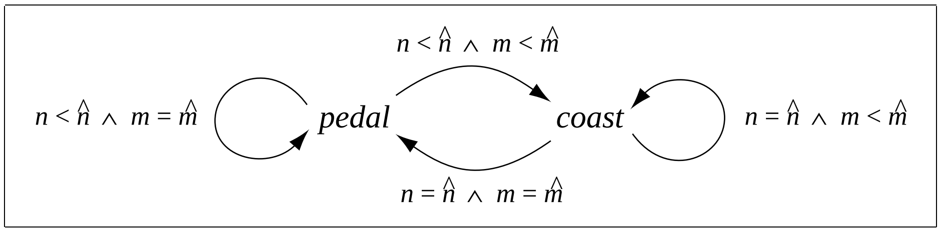
Here is the text of the Cycling Termination program as a goal for the VCG.

```
g [[ program

        procedure pedal (;val n,m);
           pre  true;
           post true;
           calls pedal with n < ^n /\ m = ^m;
           calls coast with n < ^n /\ m < ^m;
           recurses    with n < ^n;

           if 0 < n then
              if 0 < m then
                 coast(;n - 1,m - 1)
              else skip
              fi;
              pedal(;n - 1,m)
           else skip
           fi
        end procedure;

        procedure coast (;val n,m);
           pre  true;
           post true;
           calls pedal with n = ^n /\ m = ^m;
           calls coast with n = ^n /\ m < ^m;
           recurses    with m < ^m;

           pedal(;n,m);
           if 0 < m then
              coast(;n,m - 1)
           else skip
           fi
        end procedure;

        pedal(;7,12)

     end program
     [ true ]
  ]];;
```

Like the odd/even program, the two procedures of this program call each other and themselves recursively. However, unlike the odd/even program, the progress across each of the four arcs of the graph is different. In particular, the progress across a call from *coast* to *pedal* does not change any variables in the program.

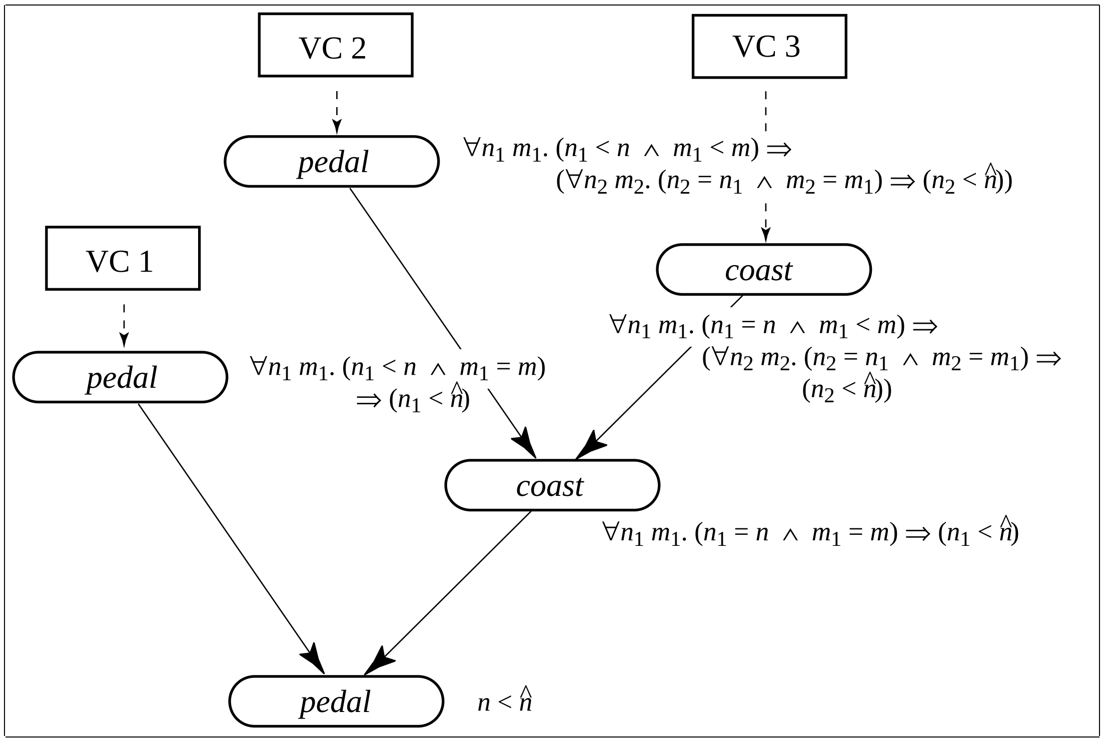
We do not mean to imply that this program could not be proven by prior methods. We only suggest that our system can generate a more natural proof, easier to create and understand. This program's termination can be proven using, say, Sokołowski's method, by creating a new value parameter p which is passed in each call, where $p = 1$ if the call is to *coast*, and where $p = 0$ if the call is to *pedal*. Then the expression $n + m + p$ becomes a workable recursion depth counter, and it reliably decreases by exactly one for every call. However, we feel that this solution is not truly natural. The introduction of a new variable unrelated to the program's purpose draws the user into a search for artifacts to prove termination. This variable p essentially serves as a kind of program counter, determining which procedure we are in at any moment. This represents control using data, an inherent confusion of concepts. Finally, the introduction of p means adding a quantity of new code to the program, concerned with maintaining the proper value of p. This code is unrelated to the original purpose of the program, and obscures that purpose on surface reading of the code.

The procedure call graph is given in Figure 8.17. Applying the graph traversal algorithm, beginning at the node *pedal*, we generate the call tree in Figure 8.18, with two undiverted recursion verification conditions, VC1 and VC2, and one diversion verification condition, VC3.



$n < \hat{n} \wedge m < \hat{m}$
$n < \hat{n} \wedge m = \hat{m}$
pedal
coast
$n = \hat{n} \wedge m < \hat{m}$
$n = \hat{n} \wedge m = \hat{m}$

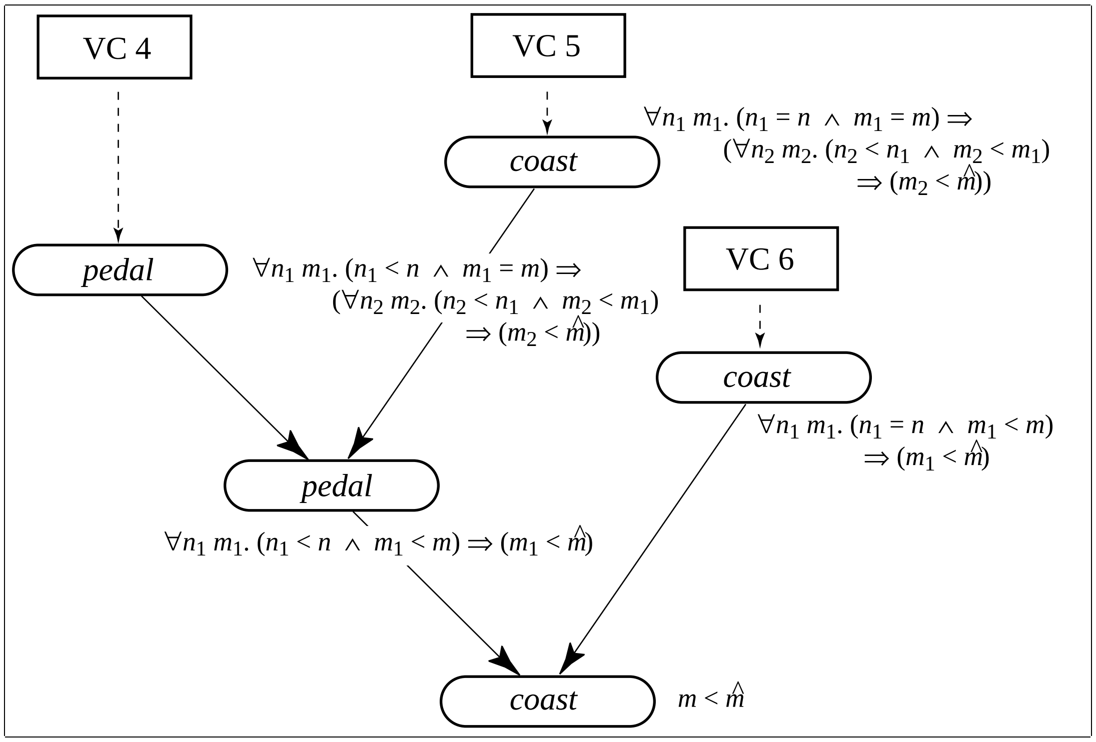
Figure 8.17: Procedure Call Graph for Cycling Termination Program.



VC 2
VC 3
pedal
$\forall n_1\, m_1.\, (n_1 < n \wedge m_1 < m) \Rightarrow$
$(\forall n_2\, m_2.\, (n_2 = n_1 \wedge m_2 = m_1) \Rightarrow (n_2 < \hat{n}))$
VC 1
coast
$\forall n_1\, m_1.\, (n_1 = n \wedge m_1 < m) \Rightarrow$
$(\forall n_2\, m_2.\, (n_2 = n_1 \wedge m_2 = m_1) \Rightarrow$
$(n_2 < \hat{n}))$
pedal
$\forall n_1\, m_1.\, (n_1 < n \wedge m_1 = m)$
$\Rightarrow (n_1 < \hat{n})$
coast
$\forall n_1\, m_1.\, (n_1 = n \wedge m_1 = m) \Rightarrow (n_1 < \hat{n})$
pedal
$n < \hat{n}$

Figure 8.18: Procedure Call Tree for root procedure *pedal*.

Applying the graph traversal algorithm, beginning at the node *coast*, we generate the call tree in Figure 8.19, with one diversion verification condition, VC4, and two undiverted recursion verification conditions, VC5 and VC6.



VC 4
VC 5
VC 6
coast
pedal
coast
pedal
coast
$\forall n_1\, m_1.\ (n_1 = n \wedge m_1 = m) \Rightarrow (\forall n_2\, m_2.\ (n_2 < n_1 \wedge m_2 < m_1) \Rightarrow (m_2 < \hat{m}))$
$\forall n_1\, m_1.\ (n_1 < n \wedge m_1 = m) \Rightarrow (\forall n_2\, m_2.\ (n_2 < n_1 \wedge m_2 < m_1) \Rightarrow (m_2 < \hat{m}))$
$\forall n_1\, m_1.\ (n_1 = n \wedge m_1 < m) \Rightarrow (m_1 < \hat{m})$
$\forall n_1\, m_1.\ (n_1 < n \wedge m_1 < m) \Rightarrow (m_1 < \hat{m})$
$m < \hat{m}$

Figure 8.19: Procedure Call Tree for root procedure *coast*.

Applying `VCG_TAC` to the program correctness goal with the tracing turned on produces the following output. In this example, we are primarily interested in the proof of termination by analyzing the structure of the procedure call graph. This section of the trace follows the line "`Examining the structure of the procedure call graph:`" in the following transcript.

```
#e(VCG_TAC);;
OK..
For procedure `pedal`,

By the "CALL" rule, we have
   [[ {(true /\ n - 1 < ^n /\ m = ^m) /\ (!n m. true ==> true)}
       pedal(;n - 1,m)
      {true} ]]

By the "CALL" rule, we have
   [[ {(true /\ n - 1 < ^n /\ m - 1 < ^m) /\
       (!n1 m1.
         true ==>
         (true /\ n - 1 < ^n /\ m = ^m) /\ (!n m. true ==> true))}
       coast(;n - 1,m - 1)
      {(true /\ n - 1 < ^n /\ m = ^m) /\ (!n m. true ==> true)} ]]

By the "SKIP" rule, we have
   [[ {(true /\ n - 1 < ^n /\ m = ^m) /\ (!n m. true ==> true)}
       skip
      {(true /\ n - 1 < ^n /\ m = ^m) /\ (!n m. true ==> true)} ]]

By the "IF" rule, we have
   [[ {(0 < m => (true /\ n - 1 < ^n /\ m - 1 < ^m) /\
                 (!n1 m1. true ==> (true /\ n - 1 < ^n /\ m = ^m) /\
                                   (!n m. true ==> true))
           | (true /\ n - 1 < ^n /\ m = ^m) /\ (!n m. true ==> true))}
       if 0 < m then coast(;n - 1,m - 1) else skip fi
      {(true /\ n - 1 < ^n /\ m = ^m) /\ (!n m. true ==> true)} ]]

By the "SEQ" rule, we have
   [[ {(0 < m => (true /\ n - 1 < ^n /\ m - 1 < ^m) /\
                 (!n1 m1. true ==> (true /\ n - 1 < ^n /\ m = ^m) /\
                                   (!n m. true ==> true))
           | (true /\ n - 1 < ^n /\ m = ^m) /\ (!n m. true ==> true))}
       if 0 < m then coast(;n - 1,m - 1) else skip fi; pedal(;n - 1,m)
      {true} ]]

By the "SKIP" rule, we have
   [[ {true} skip {true} ]]
```

```
By the "IF" rule, we have
   [[ {(0 < n
          => (0 < m
                 => (true /\ n - 1 < ^n /\ m - 1 < ^m) /\
                    (!n1 m1. true ==> (true /\ n - 1 < ^n /\ m = ^m) /\
                                      (!n m. true ==> true))
                  | (true /\ n - 1 < ^n /\ m = ^m) /\
                    (!n m. true ==> true)) | true)}
       if 0 < n then if 0 < m then coast(;n - 1,m - 1) else skip fi;
                     pedal(;n - 1,m) else skip fi
      {true} ]]

By precondition strengthening, we have
   [[ {(^n = n /\ ^m = m /\ true) /\ true}
       if 0 < n then if 0 < m then coast(;n - 1,m - 1) else skip fi;
                     pedal(;n - 1,m) else skip fi
      {true} ]]
with additional verification condition
   [[ {(^n = n /\ ^m = m /\ true) /\ true ==>
       (0 < n => (0 < m => (true /\ n - 1 < ^n /\ m - 1 < ^m) /\
                           (!n1 m1.
                             true ==> (true /\ n - 1 < ^n /\ m = ^m) /\
                                      (!n m. true ==> true))
                        | (true /\ n - 1 < ^n /\ m = ^m) /\
                          (!n m. true ==> true)) | true)} ]]

For procedure ‘coast‘,

By the "CALL" rule, we have
   [[ {(true /\ n = ^n /\ m - 1 < ^m) /\ (!n m. true ==> true)}
       coast(;n,m - 1)
      {true} ]]

By the "SKIP" rule, we have
   [[ {true} skip {true} ]]

By the "IF" rule, we have
   [[ {(0 < m => (true /\ n = ^n /\ m - 1 < ^m) /\
                 (!n m. true ==> true) | true)}
       if 0 < m then coast(;n,m - 1) else skip fi
      {true} ]]
```

```
By the "CALL" rule, we have
   [[ {(true /\ n = ^n /\ m = ^m) /\
       (!n1 m1. true ==> (0 < m => (true /\ n = ^n /\ m - 1 < ^m) /\
                                    (!n m. true ==> true) | true))}
       pedal(;n,m)
      {(0 < m => (true /\ n = ^n /\ m - 1 < ^m) /\
                 (!n m. true ==> true) | true)} ]]

By the "SEQ" rule, we have
   [[ {(true /\ n = ^n /\ m = ^m) /\
       (!n1 m1. true ==> (0 < m => (true /\ n = ^n /\ m - 1 < ^m) /\
                                    (!n m. true ==> true) | true))}
       pedal(;n,m); if 0 < m then coast(;n,m - 1) else skip fi
      {true} ]]

By precondition strengthening, we have
   [[ {(^n = n /\ ^m = m /\ true) /\ true}
       pedal(;n,m); if 0 < m then coast(;n,m - 1) else skip fi
      {true} ]]
with additional verification condition
   [[ {(^n = n /\ ^m = m /\ true) /\ true ==>
       (true /\ n = ^n /\ m = ^m) /\
       (!n1 m1. true ==> (0 < m => (true /\ n = ^n /\ m - 1 < ^m) /\
                                    (!n m. true ==> true) | true))} ]]

Examining the structure of the procedure call graph:

Traversing the call graph back from the procedure coast:

By the call graph progress from procedure coast to coast, we have
   [[ {true /\ (!n1 m1. n1 = n /\ m1 < m ==> m1 < ^m)}
      coast-<>->coast
      {m < ^m} ]]

Generating the undiverted recursion verification condition
   [[ {true /\ m = ^m ==> (!n1 m1. n1 = n /\ m1 < m ==> m1 < ^m)} ]]

By the call graph progress from procedure pedal to coast, we have
   [[ {true /\ (!n1 m1. n1 < n /\ m1 < m ==> m1 < ^m)}
      pedal-<>->coast
      {m < ^m} ]]
```

```
By the call graph progress from procedure coast to pedal, we have
   [[ {true /\ (!n1 m1. n1 = n /\ m1 = m ==>
                        (!n2 m2. n2 < n1 /\ m2 < m1 ==> m2 < ^m))}
      coast-<>->pedal
      {!n1 m1. n1 < n /\ m1 < m ==> m1 < ^m} ]]

Generating the undiverted recursion verification condition
   [[ {true /\ m = ^m ==>
       (!n1 m1. n1 = n /\ m1 = m ==>
                (!n2 m2. n2 < n1 /\ m2 < m1 ==> m2 < ^m))} ]]

By the call graph progress from procedure pedal to pedal, we have
   [[ {true /\ (!n1 m1. n1 < n /\ m1 = m ==>
                        (!n2 m2. n2 < n1 /\ m2 < m1 ==> m2 < ^m))}
      pedal-<>->pedal
      {!n1 m1. n1 < n /\ m1 < m ==> m1 < ^m} ]]

Generating the diversion verification condition
   [[ {(!n1 m1. n1 < n /\ m1 < m ==> m1 < ^m) ==>
       (!n1 m1. n1 < n /\ m1 = m ==>
                (!n2 m2. n2 < n1 /\ m2 < m1 ==> m2 < ^m))} ]]

Traversing the call graph back from the procedure pedal:

By the call graph progress from procedure coast to pedal, we have
   [[ {true /\ (!n1 m1. n1 = n /\ m1 = m ==> n1 < ^n)}
      coast-<>->pedal
      {n < ^n} ]]

By the call graph progress from procedure coast to coast, we have
   [[ {true /\ (!n1 m1. n1 = n /\ m1 < m ==>
                        (!n2 m2. n2 = n1 /\ m2 = m1 ==> n2 < ^n))}
      coast-<>->coast
      {!n1 m1. n1 = n /\ m1 = m ==> n1 < ^n} ]]

Generating the diversion verification condition
   [[ {(!n1 m1. n1 = n /\ m1 = m ==> n1 < ^n) ==>
       (!n1 m1. n1 = n /\ m1 < m ==>
                (!n2 m2. n2 = n1 /\ m2 = m1 ==> n2 < ^n))} ]]
```

```
By the call graph progress from procedure pedal to coast, we have
   [[ {true /\ (!n1 m1. n1 < n /\ m1 < m ==>
                     (!n2 m2. n2 = n1 /\ m2 = m1 ==> n2 < ^n))}
      pedal-<>->coast
      {!n1 m1. n1 = n /\ m1 = m ==> n1 < ^n} ]]

Generating the undiverted recursion verification condition
   [[ {true /\ n = ^n ==>
       (!n1 m1. n1 < n /\ m1 < m ==>
                (!n2 m2. n2 = n1 /\ m2 = m1 ==> n2 < ^n))} ]]

By the call graph progress from procedure pedal to pedal, we have
   [[ {true /\ (!n1 m1. n1 < n /\ m1 = m ==> n1 < ^n)}
      pedal-<>->pedal
      {n < ^n} ]]

Generating the undiverted recursion verification condition
   [[ {true /\ n = ^n ==> (!n1 m1. n1 < n /\ m1 = m ==> n1 < ^n)} ]]

For the main body,

By the "CALL" rule, we have
   [[ {(true /\ true) /\ (!n m. true ==> true)} pedal(;7,12) {true} ]]

By precondition strengthening, we have
   [[ {true} pedal(;7,12) {true} ]]
with additional verification condition
   [[ {true ==> (true /\ true) /\ (!n m. true ==> true)} ]]

8 subgoals
"!m ^m n. (m = ^m) ==> (!n1 m1. (n1 = n) /\ m1 < m ==> m1 < ^m)"

"!m ^m n.
  (m = ^m) ==>
  (!n1 m1.
    (n1 = n) /\ (m1 = m) ==> (!n2 m2. n2 < n1 /\ m2 < m1 ==> m2 < ^m))"

"!n m ^m.
  (!n1 m1. n1 < n /\ m1 < m ==> m1 < ^m) ==>
  (!n1 m1.
    n1 < n /\ (m1 = m) ==> (!n2 m2. n2 < n1 /\ m2 < m1 ==> m2 < ^m))"
```

```
"!n m ^n.
  (!n1 m1. (n1 = n) /\ (m1 = m) ==> n1 < ^n) ==>
  (!n1 m1.
    (n1 = n) /\ m1 < m ==> (!n2 m2. (n2 = n1) /\ (m2 = m1) ==>
                                   n2 < ^n))"

"!n ^n m.
  (n = ^n) ==>
  (!n1 m1.
    n1 < n /\ m1 < m ==> (!n2 m2. (n2 = n1) /\ (m2 = m1) ==> n2 < ^n))"

"!n ^n m. (n = ^n) ==> (!n1 m1. n1 < n /\ (m1 = m) ==> n1 < ^n)"

"!^n n ^m m.
  (^n = n) /\ (^m = m) ==>
  ((n = ^n) /\ (m = ^m)) /\
  (!n1 m1. (0 < m => ((n = ^n) /\ (m - 1) < ^m) | T))"

"!^n n ^m m.
  (^n = n) /\ (^m = m) ==>
  (0 < n =>
   (0 < m =>
    (((n - 1) < ^n /\ (m - 1) < ^m) /\
     (!n1 m1. (n - 1) < ^n /\ (m = ^m))) |
    ((n - 1) < ^n /\ (m = ^m))) |
   T)"

() : void
```

These eight subgoals, in this order, roughly correspond to the following claims:

- The value of the recursion expression of the procedure *pedal* strictly decreases across the undiverted recursion path *pedal* $\rightarrow$ *pedal* (VC1).
- The value of the recursion expression of the procedure *pedal* strictly decreases across the undiverted recursion path *pedal* $\rightarrow$ *coast* $\rightarrow$ *pedal* (VC2).
- The diversion of *coast* in *coast* $\rightarrow$ *coast* $\rightarrow$ *pedal* does not interfere with

the recursive progress of the procedure *pedal* (VC3).

- The diversion of *pedal* in *pedal* $\rightarrow$ *pedal* $\rightarrow$ *coast* does not interfere with the recursive progress of the procedure *coast* (VC4).
- The value of the recursion expression of the procedure *coast* strictly decreases across the undiverted recursion path *coast* $\rightarrow$ *pedal* $\rightarrow$ *coast* (VC5).
- The value of the recursion expression of the procedure *coast* strictly decreases across the undiverted recursion path *coast* $\rightarrow$ *coast* (VC6).
- The body of procedure *coast* is partially correct.
- The body of procedure *pedal* is partially correct.

Of these eight subgoals, two have to do with syntactic structure partial correctness, four have to do with undiverted recursion, and two have to do with diversions.

All of these subgoals are readily solved. This proof has been completed in HOL, yielding the following theorem:

```
|- [[ program
         procedure pedal(;n,m);
            global ;
            pre  true;
            post true;
            calls pedal with n < ^n /\ m = ^m;
            calls coast with n < ^n /\ m < ^m;
            recurses with n < ^n;

            if 0 < n
               then if 0 < m then coast(;n - 1,m - 1) else skip fi;
                    pedal(;n - 1,m)
               else skip
            fi
         end procedure;
         procedure coast(;n,m);
            global ;
            pre  true;
            post true;
            calls pedal with n = ^n /\ m = ^m;
            calls coast with n = ^n /\ m < ^m;
            recurses with m < ^m;

            pedal(;n,m);
            if 0 < m then coast(;n,m - 1) else skip fi
         end procedure;

         pedal(;7,12)
      end program
      [true] ]]
```