Mechanical Verification of Total Correctness Through Diversion Verification Conditions

Peter V. Homeier[1] and David F. Martin[2]

[1] University of Pennsylvania Computer and Information Science Department
Philadelphia, Pennsylvania 19104-6389 USA
homeier@saul.cis.upenn.edu
[2] UCLA Computer Science Department, 4732 Boelter Hall
Los Angeles, California 90095-1596 USA
dmartin@cs.ucla.edu

Abstract. The total correctness of programs with mutually recursive procedures is significantly more complex than their partial correctness. Past methods of proving termination have suffered from being rigid, not general, non-intuitive, and *ad hoc* in structure, not suitable for mechanization.We have devised a new method for proving termination and mechanized it within an automatic tool called a Verification Condition Generator. This tool analyzes not only the program's syntax but also, uniquely, its procedure call graph, to produce verification conditions sufficient to ensure the program's total correctness. *Diversion verification conditions* reduce the labor involved in proving termination from infinite to finite. The VCG tool has itself been deeply embedded and proven sound within the HOL theorem prover with respect to the underlying structural operational semantics. Now proofs of total correctness of individual programs may be significantly automated with complete security.

1 Introduction

If a program is partially correct but not proven to terminate, then its correct answers may not ever be provided. Thus, the termination of programs is an essential element of their proper function. Even for programs which are not intended to terminate, such as operating systems or embedded reactive programs, significant portions *are* expected to terminate, such as a response to an external event. But assuring this termination is a complex task. Previous methods have required considerable work and skill to create proofs of termination, and each such proof has depended greatly on the *ad hoc* structure of the particular problem. What is needed is automation of the proof process; but these prior methods are not sufficiently regular to support mechanization.

In this paper we investigate the semi-automatic verification of the total correctness of programs with mutually recursive procedures, including termination. The automation is performed by a tool called a *Verification Condition Generator*, or VCG, which constructs the proof of a program's correctness, modulo a set of *verification conditions*, which are logical formulae left to the user to prove.

This twice simplifies the programmer's task, as it reduces both the volume and the level of the proof. These verification conditions do not contain any references to programming language constructs or concepts, such as assignment or recursion, but only involve relationships among the values used in the program.

In the past, VCG tools in general were not themselves verified [8]. This meant that the soundness of a proof of a program's correctness rested upon the soundness of an unverified tool. Most VCGs are based on an axiomatic semantics, and in particular for procedures, there is a history of axiomatic semantics proposed in the literature which were later found to be unsound [1]. The VCG tool we present is itself verified to be sound. This means that for any program and specification, if the verification conditions are proven, then the program must be totally correct with respect to its specification. The proof of soundness was conducted within and checked by the HOL mechanical theorem prover [3], based on the structural operational semantics for the programming language. The theorem of the soundness of the VCG forms the basis for practical, effective proofs of total correctness for individual programs, with complete security. We have previously verified a VCG for partial correctness [7]; the requirement of proving termination has added at least as much complexity as all of partial correctness.

In this investigation, we have discovered a more powerful method for verifying the termination of programs with mutually recursive procedures than those previously proposed in the literature, called the *diversion verification condition* method. Though counterintuitive at first glance, these diversion verification conditions reduce the labor involved in proving termination from infinite to finite. This method is both more general and more natural than prior ones, and is also suitable for mechanization in a VCG. The VCG we exhibit here implements this new method for proving termination and hence total correctness.

The organization of this paper is as follows. Section 2 discusses approaches to proving termination. In Section 3 we give the syntax of the language used, and in Section 4 its semantics. Section 5 defines the automatic VCG tool. Section 6 applies the VCG to an example with an interesting termination argument. Section 7 focuses on the graph analysis, and in Section 8 we conclude.

2 Termination

For normal while loops, termination can be assured through the specification of a "variant" expression, with values in a well-founded set, that can be shown to strictly decrease for each iteration of the loop. For programs with mutually recursive procedures, a new form of nontermination arises, where a procedure calls itself recursively, either directly or through a chain of intermediate procedures, where each recursive invocation continues calling itself deeper and deeper, without ever returning from any of these calls. This is known as "infinite recursive descent," and it must be eliminated for total correctness.

Originating with Sokołowski [11], and continuing with Apt [2], America and de Boer [1], and Pandya and Joseph [10], rules have been presented based on Hoare's rule for partial correctness [4]; but these have suffered from *ad hoc* organization, lack of generality and awkwardness for many real programs. The essential idea of these methods is the introduction of a recursion depth counter, an integer-valued expression which is required to decrease by exactly one upon each deeper procedure call. Since the expression is required to be non-negative, it cannot continue decreasing forever, and hence no infinite descent is possible.

This is effective for proving the termination of many programs whose code naturally matches this structure, but many real programs, like recursive descent parsers, do not easily fit within the rigor of requiring the recursion depth counter to decrease by exactly one upon each new call. A fast multiplication example was given by Pandya and Joseph [10]. They ameliorated the problem by reducing the number of procedures for which the depth counter needed to decrease, but retained the rigidity of the recursion depth counter for the remainder.

We have designed a more general and flexible approach, where we require progress not across a single procedure call, but around a *cycle* in the procedure call graph. Each procedure which is intended to recurse is given a well-founded expression which is expected to decrease between recursive calls of *that* procedure, irrespective of the number of other procedures called along the way. This decrease around each cycle can be demonstrated if the progress along each single arc of the cycle is known. This allows individual arcs to contribute no forward progress, or even to step backwards, as long as in the end, when the cycle is completely traversed, the accumulation of all the progress along the cycle suffices to ensure the well-founded decrease. Thus we free the termination argument from being artificially attached to the number of procedure calls, and return it to the actual problem of instances of recursion. Also, different procedures may use different well-founded expressions, according to the reason why each terminates.

We establish the decrease of a procedure's well-founded expression across every path of nested calls that leads to a recursive call in two parts. First, the progress of each individual procedure across a single call is specified and proven by analysis of the program's syntax. Then in the second part, the procedure call graph is analyzed, where the progress of all the procedures around a cycle in the procedure call graph is "added together." If the progress around each cycle implies the recursive progress, then termination of the procedure is assured.

Most programs which contain mutually recursive procedures will have an infinite number of possible paths of recursion, all of which need to be verified for the necessary recursive progress. This would be impossible, but we have discovered a counter-intuitive but effective means to simplify this task to a finite one, through *diversion verification conditions*. This is discussed in Section 7.

Our new method is more general and flexible than previous proposals. It supports more natural proofs and also may enable proofs of programs otherwise impractical. In addition, the proofs are more stable under program evolution, such as breaking out an interior code block into a new procedure, because this method is based on essential progress between recursive calls, rather than the fragile rigidity of the exact number of procedure calls involved, an artifact of the code rather than the problem being solved. Of ultimate practical value, the regular structure of this proof method supports mechanization in a VCG.

3 Programming and Assertion Languages Syntax

We deeply embed a simple imperative programming language called Sunrise in the HOL logic to illustrate the VCG with the new termination methodology. The syntax of the Sunrise programming and assertion languages is given in Table 1.

`exp`: $e ::= n \mid x \mid {+}{+}x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2$

`bexp`: $b ::= e_1 = e_2 \mid e_1 < e_2 \mid es_1 \ll es_2 \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \sim b$

`cmd`: $c ::=$ **skip** $\mid$ **abort** $\mid x := e \mid c_1 \,;\, c_2 \mid$ **if** b **then** c_1 **else** c_2 **fi** $\mid$
assert a **with** a_{pr} **while** b **do** c **od** $\mid p(x_1, \ldots, x_n \,;\, e_1, \ldots, e_n)$

`decl`: $d ::=$ **procedure** $p($**var** $x_1, \ldots, x_n$; **val** $y_1, \ldots, y_m)$;
global $z_1, \ldots, z_k$;
pre a_{pre}; (this will be represented later as
post a_{post}; **proc** $p\ vars\ vals\ glbs\ pre\ post\ enters\ rec\ c$)
enters p_1 **with** a_1;
$\vdots$
enters p_j **with** a_j;
recurses with a_{rec};
c
end procedure $\mid$
$d_1 \,;\, d_2 \mid$
empty

`prog`: $\pi ::=$ **program** d ; c **end program**

`vexp`: $v ::= n \mid x \mid v_1 + v_2 \mid v_1 - v_2 \mid v_1 * v_2$

`aexp`: $a ::=$ **true** $\mid$ **false** $\mid v_1 = v_2 \mid v_1 < v_2 \mid vs_1 \ll vs_2 \mid a_1 \wedge a_2 \mid a_1 \vee a_2 \mid$
$\sim a \mid a_1 \Rightarrow a_2 \mid a_1 = a_2 \mid a_1$ `=>` $a_2 \mid a_3 \mid$ **close** $a \mid \forall x.\, a \mid \exists x.\, a$

Table 1: Programming and Assertion Language Syntax

The notation $f[e/x]$ indicates the function f overridden so that

$$(f[e/x])(x) = e, \text{ and for } y \neq x,\ (f[e/x])(y) = f(y)$$

We will also use $f[es/xs]$ where es and xs are lists, to indicate a multiple override in order from right to left across the lists, so the right-most elements of es and xs make the first override, and the others are added on top of this.

Most of these constructs are standard. n is an unsigned integer ≥ 0 (`num`). x is a program variable, required to not begin with the single quote character (`'`); such names are reserved as "logical" variables. ++ is the increment operator, with a side effect as in C. $es_1 \ll es_2$ is the lexicographic ordering between two lists. **abort** causes an immediate abnormal termination. The **while** loop requires an invariant assertion a and a variant expression v (in $a_{pr} = (v < x)$) to be supplied. In the procedure call $p(xs; es)$, p is a string, xs is a list of variables, denoting the actual variable parameters (passed by call-by-name), and es is a list of `exp` expressions, denoting actual value parameters (call-by-value).

The procedure declaration specifies the procedure's name p, formal variable parameter names $x_1, \ldots, x_n$, formal value parameter names $y_1, \ldots, y_m$, global variables used in p (or any procedure p calls) $z_1, \ldots, z_k$, precondition a_{pre}, postcondition a_{post}, entrance progress expressions a_1 for p_1 through a_j for p_j, recursive progress expression a_{rec}, and body c. All parameter types are `num`. The *entrance* of a procedure is within its scope, just before the body. We refer to a typical procedure declaration as **proc** p $vars$ $vals$ $glbs$ pre $post$ $enters$ rec c, instead of the longer version in Table 1. Here $enters$ collects all the entrance progress clauses together, as an *entrance progress environment*, of type `prog_env` = `string -> aexp`, defined as $enters = (\lambda p.\mathbf{false})[a_1/p_1]\ldots[a_j/p_j]$. Procedures are mutually recursive, and may call each other irrespective of declaration order.

The syntax of the associated assertion language is also given in Table 1. Most of these constructs are standard. Note that $v_1 - v_2$ terminates at zero, e.g., $3-5=0$. $vs_1 \ll vs_2$ is the lexicographic ordering between two lists. a_1 `=>` a_2 `|` a_3 is a conditional expression, yielding the value of a_2 or a_3 depending on the value of a_1. **close** a forms the universal closure of a, quantifying all free variables.

The functions $FV_a\ a$ and $FV_v\ v$ yield the sets of all free variables in a and v.

Variables contain two parts, a string and a variant number (of type `num`), assembled into a variable by the constructor function $VAR\ s\ n$. The function *variant* x s produces a variable which is a variant of x, but which is guaranteed not to be within the set s. In addition, the closest possible variant is produced, so if x itself is not within s, then *variant* $x\ s = x$. More generally,

$$variant\ x\ s \;=\; (x \in s => variant\ (mk\_variant\ x\ 1)\ s \mid x)$$

where $mk\_variant\ (VAR\ s\ n)\ k = VAR\ s\ (n+k)$.

The function *variants* xs s applies *variant* repeatedly to each of the elements of the list xs to produce variants distinct from the set s and from each other.

$$\begin{array}{ll} variants\ []\ s & = [] \\ variants\ (\mathbf{cons}\ x\ xs)\ s & = \mathbf{let}\ x' = variant\ x\ s\ \mathbf{in} \\ & \quad \mathbf{cons}\ x'\ (variants\ xs\ (\{x'\} \cup s)) \end{array}$$

Variables are separated into two classes, program variables and logical variables, where logical variables are distinguished by beginning with a quote character (`'`). A program variable may be converted to a corresponding logical variable by prepending a quote; this is done by the function *logical* x. This is naturally extended to lists of variables by the function *logicals* xs. The variant functions above may be applied to either program or logical variables.

A simultaneous substitution of expressions for variables is considered an object, apart from applying the substitution to an expression. The substitution is represented by a function from variables to expressions; all but a finite number of variables map to themselves. Such a substitution is created by $[ys/xs]$, where ys and xs are lists of variables. (Note this overloads the notation for function override.) Then $a \triangleleft ss$ explicitly applies a substitution ss to an expression a, automatically renaming bound variables in a as necessary to avoid capture.

Because the programming and assertion languages are distinct, functions are provided to translate from the programming language to the assertion language: *VE* for numeric expressions, *VES* for lists of numeric expressions, and *AB* for boolean expressions. Since in this language expressions may have side effects, functions are also provided to yield the substitutions that represent those side effects: *VE_state* for numeric expressions, *VES_state* for lists of numeric expressions, and *AB_state* for boolean expressions.

As a product, we may now define the simultaneous substitution that corresponds to a single or multiple assignment statement, overriding the expression's state change with the change of the assignment:

$$[x := e] = (\mathit{VE\_state}\ e)[(\mathit{VE}\ e)\ /\ x]$$
$$[xs := es] = (\mathit{VES\_state}\ es)[(\mathit{VES}\ es)\ /\ xs]$$

Also, we define the function *ab_pre*, which given a boolean expression b and a desired postcondition q, yields an appropriate precondition which if true, ensures that after executing b that the postcondition q holds.

$$\mathit{ab\_pre}\ b\ q \;=\; q \triangleleft (\mathit{AB\_state}\ b)$$

This brief description of the translation functions is detailed more in [6], [7].

4 Operational Semantics

This language is almost exactly the same as that given by Homeier and Martin [7], except for the additional annotations for termination. That paper presents a structural operational semantics for the programming language and a denotational semantics for the assertion language, which apply here almost without modification. In that paper, tables 2 and 3 give the structural operational semantics of the Sunrise programming language, as rules inductively defining the six relations E, B, ES, C, D, and P. These relations (except for ES) are defined within HOL using Tom Melham's excellent rule induction package [9].

For example, the relation $C\ c\ \rho\ s_1\ s_2$ expresses how a command c may operate on a state s_1 of type `state = var->num` (binding non-negative integer values to variables), in the presence of an environment ρ (containing all information about all declared procedures), to produce a resulting state s_2. The procedure environment ρ is represented as a function from procedure names to tuples; we define the type `env` as

```
string ->  (var list # var list # var list # aexp # aexp
             # prog_env # aexp # cmd),
```

The tuple contains, in order, the variable parameter list, value parameter list, global variables list, the precondition, the postcondition, the entrance progress environment, the recursive progress expression, and the body.

The semantics is changed from that described in [7] as follows. The multiplication operator is added, analogous to the addition operator. The lexicographic ordering operator is added, analogous to the less-than operator, where the two arguments are evaluated using the relation ES instead of E. The empty declaration is added, with the semantics of not producing any modification in the environment ρ. Other than these, where loops or declarations have additional annotations, consider the semantics rules to be adapted without any new effect.

For defining P, we use the empty environment $\rho_0 = \lambda p.\langle\ [\,],\ [\,],\ [\,],\ \mathbf{false},$ $\mathbf{true},\ (\lambda p.\ \mathbf{false}),\ \mathbf{false},\ \mathbf{abort}\rangle$, and the initial state $s_0 = \lambda x.\ 0$. We may construct an environment ρ from a declaration d as $\rho = mkenv\, d\, \rho_0$, where

$$\begin{aligned}
&mkenv\,(\mathbf{proc}\ p\ vars\ vals\ glbs\ pre\ post\ enters\ rec\ c)\ \rho \\
&\qquad\qquad\qquad\qquad = \rho[\langle vars, vals, glbs, pre, post, enters, rec, c\rangle\,/\,p] \\
&mkenv\,(d_1;\, d_2)\,\rho \quad = mkenv\, d_2\,(mkenv\, d_1\,\rho) \\
&mkenv\,(\mathbf{empty})\,\rho = \rho
\end{aligned}$$

The semantics of the assertion language is almost the same as that given in table 4 of [7] by recursive functions V, VS, and A defined on the structure of assertion-language expressions, in a denotational fashion. The only change is the addition of the lexicographic relation. For example, the function $A\ a\ s$ evaluates the boolean expression a in the state s, yielding a boolean value. Similarly, V and VS evaluate numeric expressions and lists of numeric expressions, respectively.

The Hoare-style total correctness of programs ($\pi[q]$) can now be defined as

$$\pi[q] \;=\; (\forall s.\ P\ \pi\ s \Rightarrow A\ q\ s) \wedge (\exists s.\ P\ \pi\ s)$$

5 Verification Condition Generator

In this section we give the definition of the Verification Condition Generator as new functions within the HOL logic for proving the total correctness of Sunrise programs. There are two general classes of functions, those which analyze the structure of the program's syntax, and those which analyze the structure of the program's procedure call graph.

5.1 Verification of Commands and Declarations

We begin with the analysis of commands. The VCG functions for this are $vcgc$, the main function, and $vcg1$, which does most of the work. These are presented in Table 2. In the definitions of these functions, comma (,) makes a pair of two items, square brackets ([ ]) delimit lists, semicolon (;) within a list separates elements, and ampersand (&) appends two lists. In addition, the function $dest_<$ is a destructor function, breaking an assertion language expression of the form $v_0 < v_1$ into a pair of its constituent subexpressions, v_0 and v_1.

$$
\begin{array}{l}
\mathit{vcg1}\ (\mathbf{skip})\ \mathit{enters}\ q\ \rho\ =\ q,\ [\,] \\
\mathit{vcg1}\ (\mathbf{abort})\ \mathit{enters}\ q\ \rho\ =\ \mathbf{false},\ [\,] \\
\mathit{vcg1}\ (x := e)\ \mathit{enters}\ q\ \rho\ =\ q \lhd [x := e],\ [\,] \\
\mathit{vcg1}\ (c_1\ ;\ c_2)\ \mathit{enters}\ q\ \rho\ = \\
\qquad\qquad\qquad \mathbf{let}\ (s, h_2) = \mathit{vcg1}\ c_2\ \mathit{enters}\ q\ \rho\ \mathbf{in} \\
\qquad\qquad\qquad \mathbf{let}\ (p, h_1) = \mathit{vcg1}\ c_1\ \mathit{enters}\ s\ \rho\ \mathbf{in} \\
\qquad\qquad\qquad\qquad p,\ h_1 \mathbin{\&} h_2 \\
\mathit{vcg1}\ (\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi})\ \mathit{enters}\ q\ \rho\ = \\
\qquad\qquad\qquad \mathbf{let}\ (r_1, h_1) = \mathit{vcg1}\ c_1\ \mathit{enters}\ q\ \rho\ \mathbf{in} \\
\qquad\qquad\qquad \mathbf{let}\ (r_2, h_2) = \mathit{vcg1}\ c_2\ \mathit{enters}\ q\ \rho\ \mathbf{in} \\
\qquad\qquad\qquad\qquad (AB\ b => \mathit{ab\_pre}\ b\ r_1 \mid \mathit{ab\_pre}\ b\ r_2),\ h_1 \mathbin{\&} h_2 \\
\mathit{vcg1}\ (\mathbf{assert}\ a\ \mathbf{with}\ a_{pr}\ \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od})\ \mathit{enters}\ q\ \rho\ = \\
\qquad\qquad\qquad \mathbf{let}\ (v_0, v_1) = \mathit{dest}_{<}\ a_{pr}\ \mathbf{in} \\
\qquad\qquad\qquad \mathbf{let}\ (p, h) = \mathit{vcg1}\ c\ \mathit{enters}\ (a \wedge a_{pr})\ \rho\ \mathbf{in} \\
\qquad\qquad\qquad\qquad a,\ [\ a \wedge AB\ b \wedge (v_0 = v_1) \Rightarrow \mathit{ab\_pre}\ b\ p\ ; \\
\qquad\qquad\qquad\qquad\quad a \wedge \sim\!(AB\ b) \Rightarrow \mathit{ab\_pre}\ b\ q\ ] \mathbin{\&} h \\
\mathit{vcg1}\ (\mathbf{call}\ p\ (xs\,;\, es))\ \mathit{enters}\ q\ \rho\ = \\
\qquad\qquad\qquad \mathbf{let}\ (\mathit{vars}, \mathit{vals}, \mathit{glbs}, \mathit{pre}, \mathit{post}, \mathit{enters}', \mathit{rec}, c) = \rho\ p\ \mathbf{in} \\
\qquad\qquad\qquad \mathbf{let}\ \mathit{vals}' = \mathit{variants}\ \mathit{vals}\ (FV_a\ q \cup \mathbf{sl}(xs \mathbin{\&} \mathit{glbs}))\ \mathbf{in} \\
\qquad\qquad\qquad \mathbf{let}\ u = xs \mathbin{\&} \mathit{vals}'\ \mathbf{in} \\
\qquad\qquad\qquad \mathbf{let}\ v = \mathit{vars} \mathbin{\&} \mathit{vals}\ \mathbf{in} \\
\qquad\qquad\qquad \mathbf{let}\ x = u \mathbin{\&} \mathit{glbs}\ \mathbf{in} \\
\qquad\qquad\qquad \mathbf{let}\ y = v \mathbin{\&} \mathit{glbs}\ \mathbf{in} \\
\qquad\qquad\qquad \mathbf{let}\ x_0 = \mathit{logicals}\ x\ \mathbf{in} \\
\qquad\qquad\qquad \mathbf{let}\ y_0 = \mathit{logicals}\ y\ \mathbf{in} \\
\qquad\qquad\qquad \mathbf{let}\ x_0' = \mathit{variants}\ x_0\ (FV_a\ q)\ \mathbf{in} \\
\qquad\qquad\qquad\qquad (\ (\ (\mathit{pre} \wedge \mathit{enters}\ p) \lhd [u/v]\ ) \wedge \\
\qquad\qquad\qquad\qquad\quad (\ (\forall x.\ (\mathit{post} \lhd [u \mathbin{\&} x_0'/v \mathbin{\&} y_0]) \Rightarrow q) \lhd [x/x_0']\ ) \\
\qquad\qquad\qquad\qquad) \lhd [\mathit{vals}' := es],\ [\,]
\end{array}
$$

$$
\begin{array}{l}
\mathit{vcgc}\ p\ c\ \mathit{enters}\ q\ \rho\ =\ \mathbf{let}\ (a, h) = \mathit{vcg1}\ c\ \mathit{enters}\ q\ \rho\ \mathbf{in} \\
\qquad\qquad\qquad\qquad [\ p \Rightarrow a\ ]\ \mathbin{\&}\ h
\end{array}
$$

$$
\begin{array}{l}
\mathit{vcgd}\ (\mathbf{proc}\ p\ \mathit{vars}\ \mathit{vals}\ \mathit{glbs}\ \mathit{pre}\ \mathit{post}\ \mathit{enters}\ \mathit{rec}\ c)\ \rho\ = \\
\qquad\qquad\qquad \mathbf{let}\ x = \mathit{vars} \mathbin{\&} \mathit{vals} \mathbin{\&} \mathit{glbs}\ \mathbf{in} \\
\qquad\qquad\qquad \mathbf{let}\ x_0 = \mathit{logicals}\ x\ \mathbf{in} \\
\qquad\qquad\qquad\qquad \mathit{vcgc}\ (x_0 = x\ \wedge\ \mathit{pre})\ c\ \mathit{enters}\ \mathit{post}\ \rho \\
\mathit{vcgd}\ (d_1\ ;\ d_2)\ \rho\ =\ \mathbf{let}\ h_1 = \mathit{vcgd}\ d_1\ \rho\ \mathbf{in} \\
\qquad\qquad\qquad\quad \mathbf{let}\ h_2 = \mathit{vcgd}\ d_2\ \rho\ \mathbf{in} \\
\qquad\qquad\qquad\qquad h_1 \mathbin{\&} h_2 \\
\mathit{vcgd}\ (\mathbf{empty})\ \rho\ =\ [\,]
\end{array}
$$

Table 2: Definition of VCG functions for commands and declarations.

The $vcg1$ function has type `cmd` $\rightarrow$ `prog_env` $\rightarrow$ `aexp` $\rightarrow$ `env` $\rightarrow$ (`aexp` $\times$ `aexp list`). $vcg1$ takes a command, an entrance progress environment, a postcondition, and a procedure environment, and returns a pair, consisting of a precondition and a list of verification conditions that must be proved in order to verify that command with respect to the precondition, postcondition, and environments. $vcg1$ is defined recursively, based on the structure of the command. Note that the procedure call clause includes the expression *enters* p; this strengthens the precondition generated to verify not only the partial correctness of the command, but also the entrance progress claims in *enters*.

The *vcgc* function is similar to $vcg1$, but takes an additional parameter, a precondition of type `aexp`, and returns only a list of verification conditions.

The verification condition generator function to analyze declarations is *vcgd*. The *vcgd* function is also presented in Table 2. This function has type `decl` $\rightarrow$ `env` $\rightarrow$ `aexp list`. *vcgd* takes a declaration and a procedure environment, and returns a list of verification conditions that must be proved in order to verify that declaration with respect to the procedure environment.

5.2 Verification of Recursion

The next several functions analyze the structure of the procedure call graph. We will begin with the lowest level functions, and build up to the main VCG function for the procedure call graph, *vcgg*.

The purpose of the graph analysis is to verify that the progress specified in the **recurses with** clause for each procedure is achieved for every possible recursive call of the procedure. This key process is justified in Section 7.

The fundamental building block for the graph analysis is the *call_progress* function. Just as weakest precondition functions compute the appropriate precondition to establish a given postcondition for a partial correctness specification, *call_progress* computes the appropriate precondition when starting execution from the entrance of procedure p_1 to establish a given entrance condition q at the entrance of procedure p_2, using the entrance progress declared in p_1 for p_2.

There are two mutually recursive functions at the core of the algorithm to analyze the procedure call graph, *extend_graph_vcs* and *fan_out_graph_vcs*. They are presented together in Table 3. Each yields a list of verification conditions to verify recursive progress across parts of the graph. In the definitions, **sl** converts a list to a set, and **cons** adds an element to a list. **length** simply returns the length of the list. **map** applies a function to each element of a list, and gathers the results of all the applications into a new list which is the value yielded. **flat** takes a list of lists and appends them together, to "flatten" the structure into a single list of elements from all the lists.

extend_graph_vcs performs the task of tracing backwards across a particular arc of the procedure call graph. *fan_out_graph_vcs* traces backwards across all incoming arcs of a particular node in the graph. The types and meanings of the arguments to these functions are given at the bottom of Table 3.

The depth counter n was a necessary artifact to be able to define these functions in HOL. Originally, the function *fan_out_graph_vcs* was defined as

$call\_progress\ p_1\ p_2\ q\ \rho\ =$
  let $\langle vars, vals, glbs, pre, post, enters, rec, c\rangle = \rho\ p_1$ **in**
  let $x = vars\ \&\ vals\ \&\ glbs$ **in**
  let $x_0 = logicals\ x$ **in**
  let $x_0' = variants\ x_0\ (FV_a\ q)$ **in**
  let $\langle vars', vals', glbs', pre', post', enters', rec', c'\rangle = \rho\ p_2$ **in**
  let $y = vars'\ \&\ vals'\ \&\ glbs'$ **in**
  let $a = enters\ p_2$ **in**
  ($a =$ **false** => **true**
      | $(\forall y.\ (a \lhd [x_0'/x_0]) \Rightarrow q) \lhd [x/x_0']$)

$induct\_pre$ **false** = **true**
$induct\_pre\ (v < x)\ =\ (v = x)$

$extend\_graph\_vcs\ p\ ps\ p_0\ q\ pcs\ \rho\ all\_ps\ n\ p'\ =$
  let $q_1 = call\_progress\ p'\ p\ q\ \rho$ **in**
  ($q_1 =$ **true** => [ ]
    | $p' = p_0$ =>
      let $(vars, vals, glbs, pre, post, enters, rec, c) = \rho\ p_0$ **in**
        [$pre \wedge induct\_pre\ rec \Rightarrow q_1$]
    | $p' \in$ **sl**(**cons** $p\ ps$) => [$pcs\ p' \Rightarrow q_1$]
    | $fan\_out\_graph\_vcs\ p'$ (**cons** $p\ ps$) $p_0\ q_1\ (pcs[q_1/p'])\ \rho\ all\_ps\ n$
)

$fan\_out\_graph\_vcs\ p\ ps\ p_0\ q\ pcs\ \rho\ all\_ps\ (n+1)\ =$
  flat (**map** $(extend\_graph\_vcs\ p\ ps\ p_0\ q\ pcs\ \rho\ all\_ps\ n)\ all\_ps$)

$fan\_out\_graph\_vcs\ p\ ps\ p_0\ q\ pcs\ \rho\ all\_ps\ 0\ =\ [\,]$

Types and meanings of arguments:

p	: `string`	: current node (procedure name)
ps	: `string list`	: path (list of procedure names)
p_0	: `string`	: starting node (procedure name)
q	: `aexp`	: current path condition
pcs	: `string` $\rightarrow$ `aexp`	: prior path conditions
ρ	: `env`	: procedure environment
all_ps	: `string list`	: all declared procedures (list of names)
n	: `num`	: depth counter
p'	: `string`	: source node of arc being explored

Table 3: Definition of core VCG functions for graph analysis.

$graph\_vcs\ all\_ps\ \rho\ p\ =$ $\quad\quad$ **let** $(vars, vals, glbs, pre, post, enters, rec, c) = \rho\ p$ **in** $\quad\quad$ $fan\_out\_graph\_vcs\ p\ []\ p\ rec\ (\lambda p'.\ \textbf{true})\ \rho\ all\_ps\ (\textbf{length}\ all\_ps)$
$vcgg\ all\_ps\ \rho\ =\ \textbf{flat}\ (\textbf{map}\ (graph\_vcs\ all\_ps\ \rho)\ all\_ps)$

Table 4: Definition of top-level VCG functions for graph analysis.

a single primitive recursive function on n combining the definitions of both *fan_out_graph_vcs* and *extend_graph_vcs*. Then *extend_graph_vcs* was defined as a part of *fan_out_graph_vcs*, and *fan_out_graph_vcs* resolved to the remainder. For calls of *extend_graph_vcs*, n should be $\textbf{length}\ all\_ps - \textbf{length}\ ps - 1$. For calls of *fan_out_graph_vcs*, the argument should be $\textbf{length}\ all\_ps - \textbf{length}\ ps$.

The definition of *fan_out_graph_vcs* maps *extend_graph_vcs* across all defined procedures, in *all_ps*. This leads to exponential time complexity. To minimize this, it is important for the application of *extend_graph_vcs* to terminate quickly for arcs which do not exist. These are indicated by the lack of an **enters ... with** clause in the header of the procedure which is the source of the arc, leading to the default value for the progress expression, **false**. If the call of *call_progress* (from *extend_graph_vcs*) retrieves **false** for the arc, then it returns **true** immediately to *extend_graph_vcs*, which in turn immediately returns the empty list of no verification conditions. This rapid dismissal limits the exponential growth to a factor depending more on the average number of incoming arcs for nodes in the graph, than on the total number of declared procedures.

The *fan_out_graph_vcs* function is called initially by the function *graph_vcs*, given in Table 4. *graph_vcs* analyzes the procedure call graph beginning at a particular node, and generates verification conditions for paths in the graph to that node to verify its recursive progress, as declared for the procedure.

The *graph_vcs* function is called by the function *vcgg*, given in Table 4. *vcgg* analyzes the entire procedure call graph, beginning at each node in turn, and generates verification conditions for paths in the graph, to verify the recursive progress declared for each procedure in *all_ps*.

5.3 Verification of Programs

The main VCG function is *vcg*, presented in Table 5. *vcg* calls *vcgd* to analyze the declarations, *vcgg* to analyze the call graph, and *vcgc* to analyze the main body of the program. *vcg* takes a program and a postcondition as arguments, analyzes the entire program, and generates verification conditions which are sufficient to prove the program totally correct with respect to the given postcondition. In the definition of *vcg*, *proc_names* returns the list of procedure names declared in a declaration, and g_0 is the "empty" call progress environment $\lambda p.\ \textbf{true}$.

$$
\begin{array}{l}
vcg\ (\textbf{program}\ d\,;\, c\ \textbf{end program})\ q\ \ = \\
\quad \textbf{let}\ \rho = mkenv\ d\ \rho_0\ \textbf{in} \\
\quad \textbf{let}\ h_1 = vcgd\ d\ \rho\ \textbf{in} \\
\quad \textbf{let}\ h_2 = vcgg\ (proc\_names\ d)\ \rho\ \textbf{in} \\
\quad \textbf{let}\ h_3 = vcgc\ \textbf{true}\ c\ g_0\ q\ \rho\ \textbf{in} \\
\qquad\quad h_1\ \&\ h_2\ \&\ h_3
\end{array}
$$

Table 5: Definition of *vcg*, the main VCG function.

$$\texttt{vcg\_THM}:\ \vdash\ \forall \pi q.\ \ WF_p\ \pi\ \ \wedge\ \ \textbf{every close}\ (vcg\ \pi\ q)\ \ \Rightarrow\ \ \pi[q]$$

Table 6: Verification Condition Generator Verification Theorem.

The VCG has been verified within the Higher Order Logic (HOL) mechanical theorem prover [3], based on the given structural operational semantics. The ultimate theorem of the verification of the VCG is given in Table 6. This theorem states that for any program π and postcondition q, if the program is well-formed ($WF_p\ \pi$) and if all of the verification conditions yielded by the VCG ($vcg\ \pi\ q$) are true, then the program must be totally correct with respect to the postcondition ($\pi[q]$). For the vital definition of program well-formedness, please see [5].

The proof of this theorem involved 8 new types, 217 definitions, and 906 major theorems proved within HOL, using over 57,000 lines of proof. These types, definitions, and theorems were organized into 22 HOL theories, on subjects including syntax, structural operational semantics, axiomatic semantics, recursive progress, termination, and the VCG. This proof shows that the VCG is sound, that the correctness of the verification conditions it produces suffice to establish the total correctness of the annotated program. This soundness result is quite useful, as we may directly apply the verification theorem in proving individual programs totally correct within HOL, with the amount of work involved reduced significantly by the VCG while maintaining complete security, and with the full power of HOL available to the user to prove the verification conditions.

Unfortunately, space precludes giving many interesting details of the proof of this theorem, which can be found in the first author's dissertation [5]. However, Figure 7 displays the overall structure of this proof. Each box indicates a property about the entire program, with the arrows indicating logical consequence. The boxes are numbered in the order the properties are proven. The preceeding paper [7] described the proof of partial correctness; this is accomplished by the proofs of boxes 1–4. The addition in this paper of the proof of termination is accomplished by the proofs of boxes 5–9. Then the partial correctness and termination properties are combined to prove total correctness in boxes 10–12. Despite the altitude of this proof structure, it does reasonably depict the increase in conceptual difficulty in adding termination, as a factor between 2 and 3.

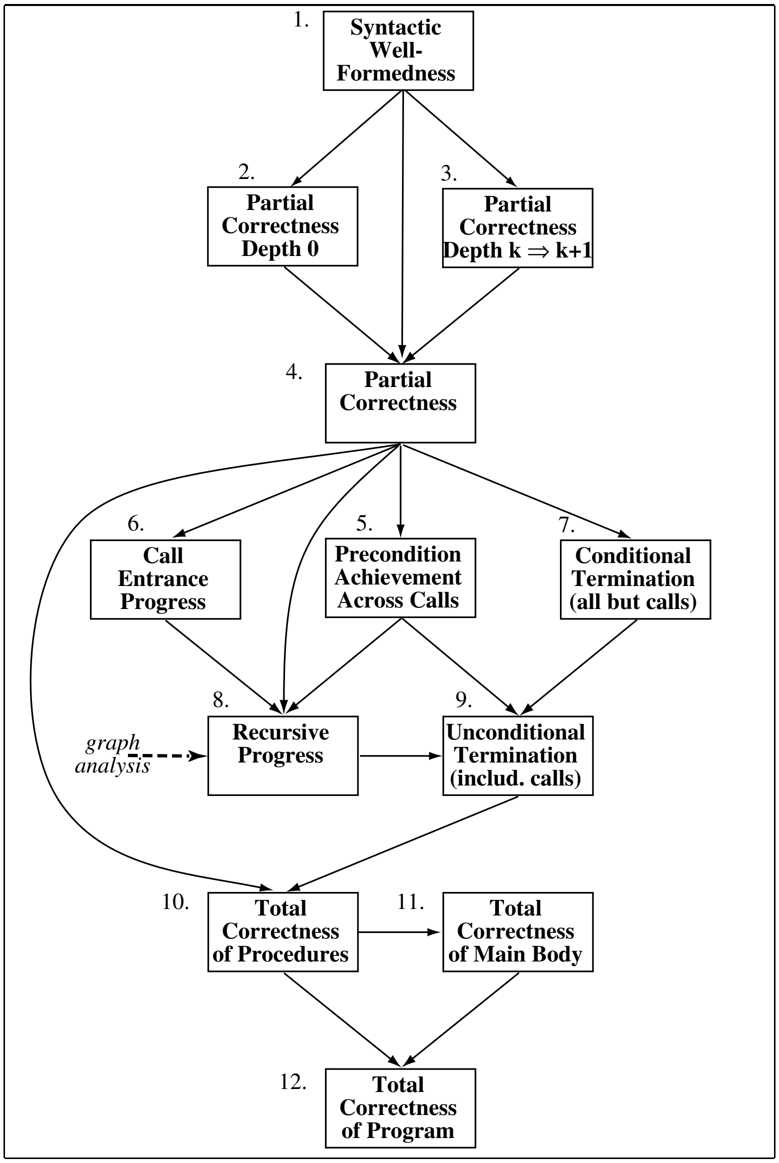

Figure 7: Structure of VCG verification proof

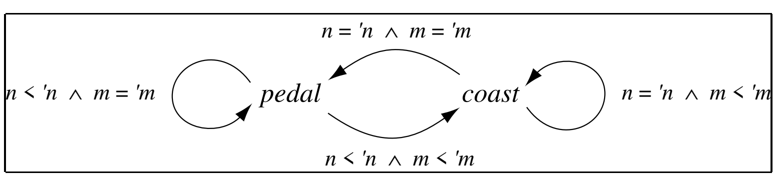

Figure 8: Bicycling Example Call Graph.

```
program

    procedure pedal(; val n, m);
        global a, b, c;
        pre     n * m + c = a * b;
        post    c = a * b;
        enters pedal with n < 'n ∧ m = 'm;
        enters coast with n < 'n ∧ m < 'm;
        recurses     with n < 'n;

        if n = 0 ∨ m = 0 then
            skip
        else
            c := c + m;
            if n < m then
                coast(; n − 1, m − 1)
            else
                pedal(; n − 1, m)
            fi
        fi
    end procedure;

    procedure coast(; val n, m);
        global a, b, c;
        pre     n * (m + 1) + c = a * b;
        post    c = a * b;
        enters pedal with n = 'n ∧ m = 'm;
        enters coast with n = 'n ∧ m < 'm;
        recurses     with m < 'm;

        c := c + n;
        if n < m then
            coast(; n, m − 1)
        else
            pedal(; n, m)
        fi
    end procedure;

    a := 7; b := 12; c := 0;
    pedal(; a, b)
end program

[ c = 7 * 12 ]
```

Table 9: Bicycling Example Program.

6 Example

We onsider an artificial but simple example, the Bicycling program in Table 9. Its procedure call graph is given in Figure 8. The effect of this program is to multiply two numbers $(a * b)$ by repeated additions, and leave the result in c. Despite the surprising nature of the partial correctness of this program, however, our primary interest is in its termination.

We call this example "Bicycling" 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. 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 while it was moving upwards would exert no force, and would coast, depending solely on the momentum generated by the other phase to propel one to the goal.[1] This corresponds to the entrance progress achieved across the arcs of the call graph.

We believe that it is difficult to prove termination for the Bicycling program using either Sokołowski's or Pandya and Joseph's methods. Sokołowski's does not easily apply, since for the call from *coast* to *pedal*, neither n nor m change, and if $n = 0$, not even c changes. Pandya and Joseph's method relies on finding a smaller set of procedures, but since both procedures are self-recursive, neither can be eliminated, devolving to Sokołowski's method, with the difficulties above. Nevertheless there is a very natural and simple argument that this program terminates, namely that the variable n decreases in any recursion involving *pedal*, and m decreases in any recursion involving *coast*. This argument suggests the arc labels in Figure 8, and the progress annotations in Table 9.

The VCG has been implemented as a secure HOL tactic. In the Bicycling example, applying *vcgd* to the two procedure declarations generates two VC's:

VC1: Partial correctness and entrance progress for *pedal*:

$$
\begin{array}{l}
({}'n = n \;\wedge\; {}'m = m \;\wedge\; n * m + c = a * b) \Rightarrow \\
((n = 0 \;\vee\; m = 0 \;\;=> \\
\quad c = a * b \\
\;\;|\;\; (n < m \;\;=> \\
\qquad (n-1) * ((m-1)+1) + c + m = a * b \;\wedge \\
\qquad\quad n - 1 < {}'n \;\wedge\; m - 1 < {}'m \\
\quad\;\;|\;\; (n-1) * m + c + m = a * b \;\wedge\; n - 1 < {}'n \;\wedge\; m = {}'m)))
\end{array}
$$

VC2: Partial correctness and entrance progress for *coast*:

$$
\begin{array}{l}
({}'n = n \;\wedge\; {}'m = m \;\wedge\; n * (m+1) + c = a * b) \Rightarrow \\
((n < m \;\;=> \\
\quad n * ((m-1)+1) + c + n = a * b \;\wedge\; n = {}'n \;\wedge\; m - 1 < {}'m \\
\;\;|\;\; n * m + c + n = a * b \;\wedge\; n = {}'n \;\wedge\; m = {}'m))
\end{array}
$$

Then applying *vcgg* to the procedure call graph generates six VC's:

VC3: Undiverted recursion verification condition for the path *pedal* $\rightarrow$ *pedal*:

$$n * m + c = a * b \;\wedge\; n = {}'n \;\Rightarrow\; (\forall n_1 m_1 .\; n_1 < n \;\wedge\; m_1 = m \;\Rightarrow\; n_1 < {}'n)$$

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

VC4: Undiverted recursion verification condition for the path *pedal* $\rightarrow$ *coast* $\rightarrow$ *pedal*:
$n * m + c = a * b \ \wedge \ n = {}'n \ \Rightarrow$
$(\forall n_1 m_1 .\ n_1 < n \ \wedge \ m_1 < m \ \Rightarrow \ (\forall n_2 m_2 .\ n_2 = n_1 \ \wedge \ m_2 = m_1 \ \Rightarrow \ n_2 < {}'n))$

VC5: Diversion verification condition for the path *coast* $\rightarrow$ *coast* $\rightarrow$ *pedal*:
$(\forall n_1 m_1 .\ n_1 = n \ \wedge \ m_1 = m \ \Rightarrow \ n_1 < {}'n) \ \Rightarrow$
$(\forall n_1 m_1 .\ n_1 = n \ \wedge \ m_1 < m \ \Rightarrow \ (\forall n_2 m_2 .\ n_2 = n_1 \ \wedge \ m_2 = m_1 \ \Rightarrow \ n_2 < {}'n))$

VC6: Diversion verification condition for the path *pedal* $\rightarrow$ *pedal* $\rightarrow$ *coast*:
$(\forall n_1 m_1 .\ n_1 < n \ \wedge \ m_1 < m \ \Rightarrow \ m_1 < {}'m) \ \Rightarrow$
$(\forall n_1 m_1 .\ n_1 < n \ \wedge \ m_1 = m \ \Rightarrow \ (\forall n_2 m_2 .\ n_2 < n_1 \ \wedge \ m_2 < m_1 \ \Rightarrow \ m_2 < {}'m))$

VC7: Undiverted recursion verification condition for the path *coast* $\rightarrow$ *pedal* $\rightarrow$ *coast*:
$n * (m + 1) + c = a * b \ \wedge \ m = {}'m \Rightarrow$
$(\forall n_1 m_1 .\ n_1 = n \ \wedge \ m_1 = m \Rightarrow (\forall n_2 m_2 .\ n_2 < n_1 \ \wedge \ m_2 < m_1 \Rightarrow m_2 < {}'m))$

VC8: Undiverted recursion verification condition for the path *coast* $\rightarrow$ *coast*:
$n * (m + 1) + c = a * b \ \wedge \ m = {}'m \ \Rightarrow \ (\forall n_1 m_1 .\ n_1 = n \ \wedge \ m_1 < m \ \Rightarrow \ m_1 < {}'m)$

Finally, applying *vcgc* to the main body of the program generates one VC:

VC9: Total correctness for main body:
$\mathbf{true} \Rightarrow (7 * 12 + 0 = 7 * 12)$

The reader is invited to verify that these verification conditions are true. We have applied the VCG to this example, proved the resulting VC's in HOL, and hence proven the total correctness of the Bicycling example as an HOL theorem.

7 Graph Analysis

To justify the recursive progress claims, we must prove that the recursive decrease is implied by the cumulative progress across *every* possible path in the procedure call graph that is an instance of recursion. A path is an instance of recursion if it begins and ends with the same procedure (the "root"). There is a potentially infinite number of such paths, but the task may be simplified by a series of steps.

First, if an instance of recursion includes another instance of the root interior to the path, then we call this an instance of *multiple recursion*; otherwise we call this *single recursion*. Fortunately, to prove the progress claims for instances of multiple recursion, it sufficies to prove them for instances of single recursion.

This simplifies the task, but it may still involve an infinite number of paths due to the presence of other cycles in the call graph. This is because a path which is an instance of single recursion may contain duplicate nodes other than the root. We call such an occurrence of duplicate nodes other than the root a *diversion*, as it seems to temporarily divert the path from its goal of the root procedure. To simplify the task further, we can collapse the cases for paths which differ only in how many times the diversion is traversed, if we can prove that traversing the diversion does not "lose ground" logically.

Every node in every path must be annotated with it's "path condition." This is computed using the *call_progress* function backwards across each arc, starting from the end of the path, which is initially annotated with the recursion expression for the root procedure. Paths with the same root procedure at the end may be collected into a *call tree* (see Figure 10 for *coast*), since they share some of the same path conditions. For each path, if it is an instance of recursion, the path condition at the beginning is the precondition for satisfying the recursion progress at the end. If this precondition is implied by the *induct_pre* value of the recursion expression of that procedure, this path shows the needed progress. We call such an implication a *undiverted recursion verification condition.*

Consider such a path marked with path conditions that contains a diversion. Let the earlier node of the diversion have path condition pc_1, and the later node pc_2 (see Figure 10). The similar but shorter path with the diversion removed is verified above. This verification can apply to the current path if traversing the diversion "loses no ground" logically. Surprisingly, this is assured if $pc_2 \Rightarrow pc_1$. This may seem counterintuitive, since pc_1 is earlier in time than pc_2. But in fact this guarantees that traversing the diversion continues to ensure achieving the desired recursive progress. We call $pc_2 \Rightarrow pc_1$ a *diversion verification condition.*

Proving such diversion verification conditions allows us to collapse the set of paths we must consider to a finite number, because any path with a diversion can then be reduced to a path without the diversion, and any path of single recursion without any diversions can include all non-root procedures at most once. With a finite number of declared procedures, each such path must be finite, and even with a completely connected procedure call graph, the number of possible such paths must be finite. The diversion verification condition method for proving total correctness is currently U.S. Patent Pending.

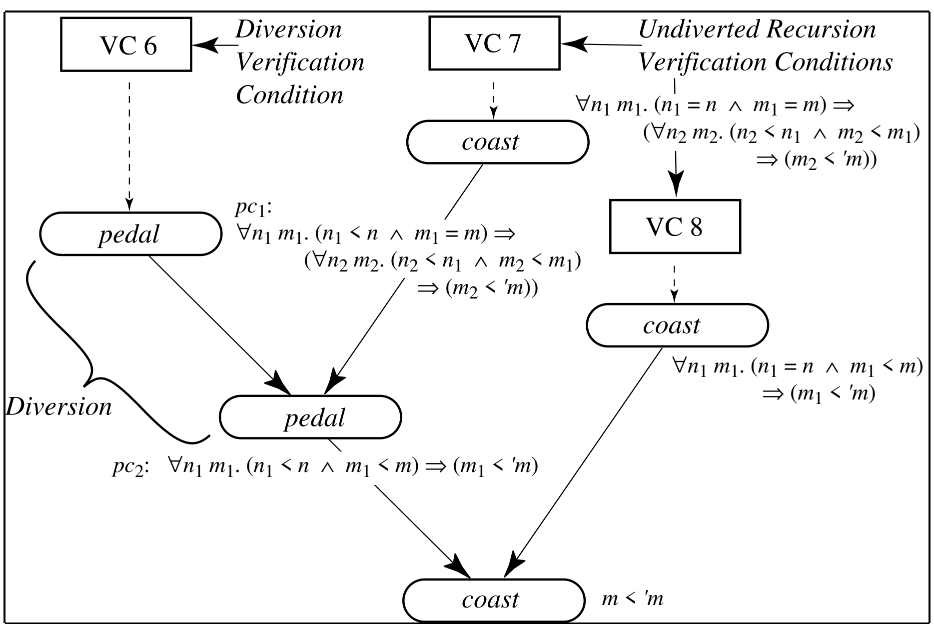

Figure 10: Call Tree for *coast*, with path conditions and verification conditions.

8 Summary and Conclusions

We have defined concisely and verified within HOL a verification condition generator which effectively implements the diversion verification condition method. This new method is more general and flexible than prior proposals, and more powerful while remaining intuitive. It is compositional and readily mechanized.

The real security of this work is in the mechanical verification of the VCG. In such subtle areas as mutual recursion and graph analysis, our personal intuition was insufficient to guarantee soundness. HOL made this security possible.

This VCG substantially decreases the difficulty of proving programs totally correct, and does so with a very high level of security. A VCG itself must be trustworthy for the proofs to be trustworthy. This level of trusworthiness is now demonstrated to be feasible, by the example of this mechanically verified VCG.

For more information on the Sunrise system, please see

`http://www.cis.upenn.edu/~hol/sunrise`.

Soli Deo Gloria.

References

1. America, P. and de Boer, F.: Proving Total Correctness of Recursive Procedures. *Information and Computation* **84** No. 2 (1990) 129–162
2. Apt, K. R.: Ten Years of Hoare logic: A Survey—Part 1. *ACM TOPLAS* **3** No. 4 (1981) 431–483
3. Gordon, M., Melham, T.: *Introduction to HOL, A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, Cambridge (1993)
4. Hoare, C. A. R.: Procedures and Parameters: an axiomatic approach, in *Proceedings of Symposium on Semantics of Algorithmic Languages*, ed. E. Engeler, Lecture Notes in Mathematics **188** (1971) 102–116
5. Homeier, P.: *Trustworthy Tools for Trustworthy Programs: A Mechanically Verified Verification Condition Generator for the Total Correctness of Procedures.* Ph.D. Dissertation, UCLA Computer Science Department (1995)
6. Homeier, P., Martin, D.: A Mechanically Verified Verification Condition Generator. *The Computer Journal* **38** No. 2 (1995) 131–141
7. Homeier, P., Martin, D.: Mechanical Verification of Mutually Recursive Procedures, in *Proceedings of the 13th International Conference on Automated Deduction (CADE-13)*, eds. M. A. McRobbie and J. K. Slaney. Lecture Notes in Arificial Intelligence **1104** Springer-Verlag (1996) 201–215
8. Igarashi, S., London, R. L., Luckham, C.: Automatic program verification: A logical basis and its implementation. *Acta Informatica* **4** (1975) 145–182
9. Melham, T.: A Package for Inductive Relation Definitions in HOL, in *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*, Davis, August 1991, ed. Archer, M., Joyce, J., Levitt, K., Windley, P. IEEE Computer Society Press (1992) 350–357
10. Pandya, P. and Joseph, M.: A Structure-directed Total Correctness Proof Rule for Recursive Procedure Calls. *The Computer Journal* **29** No. 6 (986) 531–537
11. Sokolowski, S.: Total Correctness for Procedures, in *Proceedings, 6th Symposium on the Mathematical Foundations of Computer Science*, ed. J. Gruska, Springer-Verlag, LNCS **53** (1977) 475–483