CHAPTER 7

Verification Condition Generator

"You will not need to fight in this battle. Position yourselves, stand still and see the salvation of the LORD, who is with you, O Judah and Jerusalem!"

— 2 Chronicles 20:17

In this chapter we present a verification condition generator for the Sunrise programming language. This is a function that analyzes programs with specifications to produce an implicit proof of the program's correctness with respect to its specification, modulo a set of verification conditions which need to be proven by the programmer. This reduces the problem of proving the program correct to the problem of proving the verification conditions. This is a partial automation of the program proving process, and significantly eases the task.

The many different correctness specifications and Hoare-style rules of the last chapter all culminate here, and contribute to the correctness of the VCG presented. All the rules condense into a remarkably small definition of the verification condition generator. The operations of the VCG are simple syntactic manipulations, which may be easily and quickly executed.

The correctness that is proven by the VCG is total correctness, including the termination of programs with mutually recursive procedures. Much of the content of the previous chapter was aimed at establishing the termination of programs. This is the part of the verification condition generator which is most novel. The partial correctness of programs is verified by the VCG producing a fairly standard set of verification conditions, based on the structure of the syntax of bodies of procedures and the main body of the program. Termination is verified by the VCG producing new kinds of verification conditions arising from the structure of the procedure call graph.

7.1 Definitions

In this section, we define the primary functions that make up the verification condition generator.

7.1.1 Verification of Commands

We begin with the analysis of the structure of commands. There are two VCG functions that analyze commands. The main function is the *vcgc* function. Most of the work of *vcgc* is done by a helper function, *vcg*1.

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.

$$vcg1\ (\mathbf{skip})\ calls\ q\ \rho\ =\ q,\ [\,]$$

$$vcg1\ (\mathbf{abort})\ calls\ q\ \rho\ =\ \mathbf{false},\ [\,]$$

$$vcg1\ (x := e)\ calls\ q\ \rho\ =\ q \lhd [x := e],\ [\,]$$

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

Figure 7.1: Definition of $vcg1$, helper VCG function for commands.

$$vcgc\ p\ c\ calls\ q\ \rho \;=\; \textbf{let}\ (a,h) = vcg1\ c\ calls\ q\ \rho\ \textbf{in}\ [\,p \Rightarrow a\,]\ \&\ h$$

Figure 7.2: Definition of *vcgc*, main VCG function for commands.

The *vcg*1 function is presented in Figure 7.1. This function has type `cmd` → `prog_env` → `aexp` → `env` → (`aexp` × (`aexp`)`list`). *vcg*1 takes a command, a calls progress environment, a postcondition, and a procedure environment, and returns 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. *vcg*1 is defined recursively, based on the structure of the command argument. Note that the procedure call clause includes *calls p*; this inclusion causes the verification conditions generated to verify not only the partial correctness of the command, but also the call progress claims present in *calls*.

The *vcgc* function is presented in Figure 7.2. This function has type `aexp` → `cmd` → `prog_env` → `env` → (`aexp`)`list`. *vcgc* takes a precondition, a command, a calls progress environment, a postcondition, and a procedure environment, and returns a list of verification conditions that must be proved in order to verify that command with respect to the precondition, postcondition, and environments.

7.1.2 Verification of Declarations

The verification condition generator function to analyze declarations is *vcgd*. The *vcgd* function is presented in Figure 7.3. This function has type `decl` → `env` → (`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

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

Figure 7.3: Definition of *vcgd*, VCG function for declarations.

declaration with respect to the procedure environment.

7.1.3 Verification of Call Graph

The next several functions deal with the analysis of 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*.

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 Figure 7.4. Each yields a list of verification conditions to verify progress across parts of the graph. In the definitions, SL converts a list to a set, and $CONS$ adds an element to a 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 level, a list of elements from all the lists.

The purpose of the graph analysis is to verify that the progress specified in the

```
extend_graph_vcs p ps p_0 q pcs ρ all_ps n p' =
    let q_1 = call_progress p' p q ρ in
    (q_1 = true => []
      | p' = p_0 =>
            let (vars, vals, glbs, pre, post, calls, rec, c) = ρ p_0 in
                [ pre ∧ induct_pre rec ⇒ q_1 ]
      | p' ∈ SL(CONS p ps) => [ pcs p' ⇒ q_1 ]
      | fan_out_graph_vcs p' (CONS p ps) p_0 q_1 (pcs[q_1/p']) ρ all_ps n
    )

fan_out_graph_vcs p ps p_0 q pcs ρ all_ps (n + 1) =
    FLAT (MAP (extend_graph_vcs p ps p_0 q pcs ρ all_ps n) all_ps)

fan_out_graph_vcs p ps p_0 q pcs ρ all_ps 0 = []
```

Figure 7.4: Definition of *extend_graph_vcs* and *fan_out_graph_vcs*.

recurses with clause for each procedure is achieved for every possible recursive call of the procedure. The general process is to begin at a particular node of the call graph, and explore *backwards* through the directed arcs of the graph. We associate with that starting node the recursion expression for that procedure, and this is the starting path expression. For each arc traversed backwards, the current path expression is transformed using the *call_progress* function defined in Table 6.11, and we associate the result yielded by *call_progress* with the new node reached along the arc. At each point we keep track of the path of nodes from the current node to the starting node. This backwards exploration continues recursively, until we reach a “leaf” node. A leaf node is one which is a duplicate of one already in the path of nodes to the starting node. This duplicate may match the starting node itself, or it may match one of the other nodes encountered in

the path of the exploration.

When a leaf node is reached, a verification condition is generated. These will be explained in more detail later; for now it suffices to note that there are two kinds of verification conditions generated, depending on which node the leaf node duplicated. If the leaf node matched the starting node, then we generate an *undiverted recursion* verification condition. If the leaf node matched any other node, then we generate a *diversion* verification condition.

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 arguments to these functions have the following types and meanings:

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

The depth counter n was a necessary artifact to be able to define these functions in HOL; first *fan_out_graph_vcs* was defined as a single primitive recursive function on n combining the functions of Figure 7.4. Then *extend_graph_vcs* was defined as a mutually recursive part of *fan_out_graph_vcs*, and *fan_out_graph_vcs* resolved to the remainder. For calls of *fan_out_graph_vcs*, n should be equal to the difference between the lengths of *all_ps* and ps. For calls of *extend_graph_vcs*, n should be equal to the difference between the lengths of *all_ps* and ps, minus one.

The definition of *fan_out_graph_vcs* maps *extend_graph_vcs* across all defined procedures, as listed in *all_ps*. It is expected that practically speaking, most programs will have relatively sparse call graphs, in that there will be many procedures in the program, but each individual procedure will only be called by a small fraction of all defined. Therefore it is important for the application of *extend_graph_vcs* described above to terminate quickly for applications across an arc which does not actually exist in the procedure call graph. The lack of an arc is represented by the lack of a corresponding **calls** ... **with** clause in the header of the procedure which would be the source of the arc. When assembing the calls progress environment *calls* from the **calls** ... **with** clauses of a procedure, each clause produces a binding onto an initial default calls progress environment. This default calls progress environment is $\lambda p.$ **false**. Then all references to target procedures *not* specified in the **calls** ... **with** clauses yield the default value of this default calls progress environment, **false**. This indicates that there is no relationship at all possible between the values in the states before and after such a call, and therefore signifies that such calls cannot occur. As a side benefit, this ensures that any omission of a **calls** ... **with** clause from the header of a procedure whose body does indeed contain a call to the target procedure will generate verification conditions that require proving **false**, and these will be quickly identified as untrue.

An invocation of *extend_graph_vcs* will at its beginning call the *call_progress* function. According to its definition in the last chapter, *call_progress* will evaluate *calls* p_2 to extract the progress expression. For a nonexistent arc, this will be **false**, as described above. The definition of *call_progress* then tests whether the progress expression is equal to **false**. For such a nonexistent arc in the procedure

call graph, it is, and *call_progress* then immediately terminates with value **true**.

The invocation of *extend_graph_vcs* then receives **true** as the current path condition. The next step of *extend_graph_vcs* is to test whether the path condition is equal to **true**. Since it is, the definition of *extend_graph_vcs* then immediately terminates, yielding an empty list with no verification conditions as its result.

In theory, these functions could have been designed more simply and homogeneously to yield equivalent results just using the parts of each definition which handle the general case. However, this would not have been a practical solution. All these functions are designed with particular attention to as quickly as possible dismiss all nonexistent arcs of the procedure call graph. This is critical in practice, because of the potentially exponential growth of the time involved in exploring a large graph. 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.

$$
\begin{array}{l}
\mathit{graph_vcs}\ \mathit{all_ps}\ \rho\ p\ \ = \\
\qquad \mathbf{let}\ (\mathit{vars}, \mathit{vals}, \mathit{glbs}, \mathit{pre}, \mathit{post}, \mathit{calls}, \mathit{rec}, c) = \rho\ p\ \mathbf{in} \\
\qquad \mathit{fan_out_graph_vcs}\ p\ [\,]\ p\ \mathit{rec}\ (\lambda p'.\ \mathbf{true})\ \rho\ \mathit{all_ps}\ (\mathit{LENGTH}\ \mathit{all_ps})
\end{array}
$$

Figure 7.5: Definition of *graph_vcs*.

The *fan_out_graph_vcs* function is called initially by the function *graph_vcs*. *graph_vcs* is presented in Figure 7.5. It 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 designated in its recursion expression declared in the procedure's header.

$$vcgg\ all_ps\ \rho\ =\ FLAT\ (MAP\ (graph_vcs\ all_ps\ \rho)\ all_ps)$$

Figure 7.6: Definition of *vcgg*, the VCG function to analyze the call graph.

The *graph_vcs* function is called by the function *vcgg*. *vcgg* is presented in Figure 7.6. It 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*.

7.1.3.1 Example of Verification of Call Graph

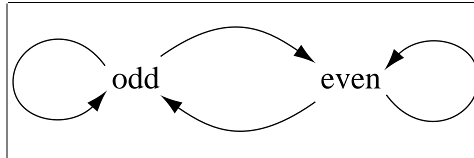

Figure 7.7: Procedure Call Graph for Odd/Even Example.

As an example of this graph traversal algorithm, consider the odd/even program in Table 6.1. We repeat its procedure call graph in Figure 7.7. We wish to explore this call graph, beginning at the node corresponding to procedure *even*. In this process, we will trace part of the structure of the procedure call tree rooted at *even*, which is given in Figure 7.8. We take the recursion expression of *even*, $n < \hat{n}$, and attach that to the *even* node. This becomes the current path expression. Examining the call graph, we see that there are two arcs coming into the *even* node, one from *odd* and one from *even* itself, as a self-loop. These will form two paths, which we will explore as two cases.

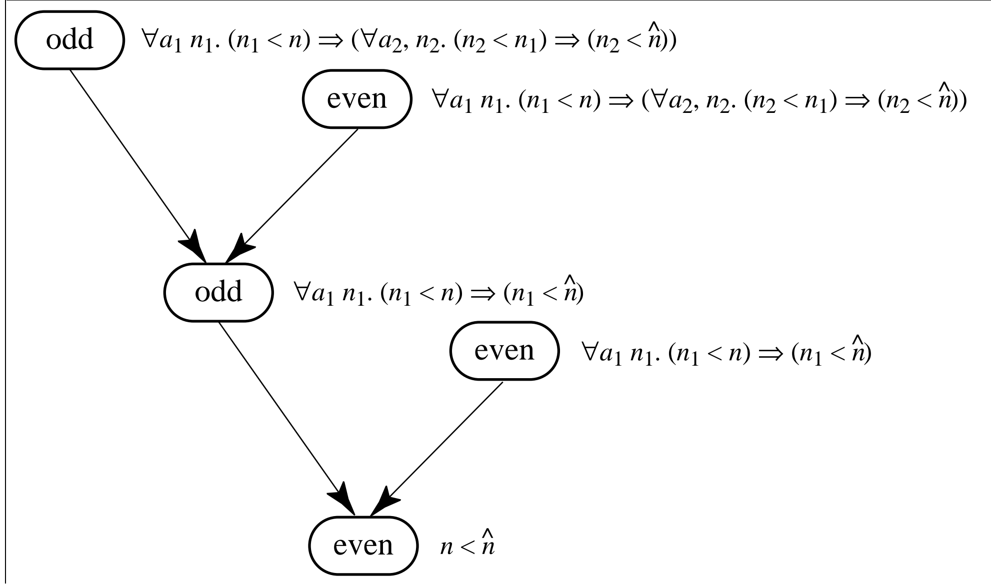

Figure 7.8: Procedure Call Tree for Odd/Even Example.

Case 1: Path $odd \rightarrow even$.

The call graph arc goes from *odd* to *even*. We push the current path expression backwards across the arc from *even* to *odd*, using the function *call_progress*. We previously described that

$$\begin{aligned} & call_progress\ odd\ even\ (n < \hat{n})\ \rho \\ = \quad & \forall a_1, n_1.\ (n_1 < n) \Rightarrow (n_1 < \hat{n}) \end{aligned}$$

We attach this path expression to the *odd* node. According to the definition of *extend_graph_vcs*, we then go through a series of tests. We first test to see if this path expression is **true**, which it clearly is not. If, however, there had been no arc in the procedure call graph from *odd* to *even*, then the *call_progress* function would have returned **true**, and *extend_graph_vcs* would terminate, yielding an empty list of verification conditions for this path.

The second test we encounter in the definition of *extend_graph_vcs* is whether the node just reached backwards across the arc is the same as the starting node. In this case, the node just reached is *odd* and the starting node is *even*, so this test is not satisfied.

The third test we encounter is whether the node just reached, *odd*, is a duplicate of one of the nodes in the path to the starting node. In this case the path only consists of the starting node *even* itself, and *odd* is not a duplicate of any member.

The choice finally arrived at in the definition of *extend_graph_vcs* is to continue the graph exploration recursively, by calling *fan_out_graph_vcs*. Considering the node *odd* in the procedure call graph in Figure 7.7, we see there are two arcs of the procedure call graph which enter the node *odd*, one from *odd* itself and one from *even*. These will form two paths, which we will explore as two cases.

Case 1.1: Path $odd \rightarrow odd \rightarrow even$.

We push the current path expression backwards across the arc from *odd* to *odd*, using the function *call_progress*. We previously described that

$$\begin{aligned} & \mathit{call_progress}\ \mathit{odd}\ \mathit{odd}\ (\forall a_1, n_1.\ (n_1 < n) \Rightarrow (n_1 < \hat{n}))\ \rho \\ = \ \ & \forall a_1, n_1.\ (n_1 < n) \Rightarrow (\forall a_2, n_2.\ (n_2 < n_1) \Rightarrow (n_2 < \hat{n})) \end{aligned}$$

This becomes the current path expression. We then go through the series of tests in the definition of *extend_graph_vcs*. We first test to see if this path expression is **true**, which it clearly is not.

The second test is whether the node just reached backwards across the arc is the same as the starting node. In this case, the node just reached is *odd* and the

starting node is *even*, so this test is not satisfied.

The third test is whether the node just reached, *odd*, is a duplicate of one of the nodes in the path to the starting node. In this case this path is $odd \rightarrow even$, so *odd is* a duplicate, and this test succeeds.

According to the definition of *extend_graph_vcs*, for satisfying this test, we generate a verification condition of the form $pcs\ p' \Rightarrow q_1$, which in this case is

$$\begin{array}{l}(\forall a_1, n_1.\ (n_1 < n) \Rightarrow (n_1 < \hat{n})) \Rightarrow \\ (\forall a_1, n_1.\ (n_1 < n) \Rightarrow (\forall a_2, n_2.\ (n_2 < n_1) \Rightarrow (n_2 < \hat{n})))\end{array}$$

We call this kind of verification condition a *diversion verification condition*, which we will describe more later.

This terminates this exploration of this path (Case 1.1) through the procedure call graph.

Case 1.2: Path $even \rightarrow odd \rightarrow even$.

We push the current path expression backwards across the arc from *odd* to *even*, using the function *call_progress*. We previously described that

$$\begin{array}{rl} & call_progress\ even\ odd\ (\forall a_1, n_1.\ (n_1 < n) \Rightarrow (n_1 < \hat{n}))\ \rho \\ = & \forall a_1, n_1.\ (n_1 < n) \Rightarrow (\forall a_2, n_2.\ (n_2 < n_1) \Rightarrow (n_2 < \hat{n}))\end{array}$$

This becomes the current path expression. We then go through the series of tests in the definition of *extend_graph_vcs*. We first test to see if this path expression is **true**, which it clearly is not.

The second test is whether the node just reached backwards across the arc is the same as the starting node. In this case, the node just reached is *even* and the starting node is *even*, so this test succeeds.

According to the definition of *extend_graph_vcs*, for satisfying this test, we generate a verification condition of the form

$$\begin{array}{l}\textbf{let } (vars, vals, glbs, pre, post, calls, rec, c) = \rho\ p_0 \textbf{ in} \\ \quad [\ pre \wedge induct_pre\ rec \Rightarrow q_1\ ],\end{array}$$

which in this case is

$$\begin{array}{l}(\textbf{true} \ \wedge\ n = \widehat{n}) \Rightarrow \\ (\forall a_1, n_1.\ (n_1 < n) \Rightarrow (\forall a_2, n_2.\ (n_2 < n_1) \Rightarrow (n_2 < \widehat{n}))).\end{array}$$

We call this kind of verification condition an *undiverted recursion verification condition*, which we will describe more later.

This terminates this exploration of this path (Case 1.2) through the procedure call graph. Since this is also the last case for expanding the path of Case 1, this also terminates the exploration of that path.

Case 2: Path $even \rightarrow even$.

The call graph arc goes from *even* to *even*. We push the current path expression backwards across the arc from *even* to *even*, using the function *call_progress*. We previously described that

$$\begin{array}{rl} & call_progress\ even\ even\ (n_1 < \widehat{n})\ \rho \\ = & \forall a_1, n_1.\ (n_1 < n) \Rightarrow (n_1 < \widehat{n})\end{array}$$

This becomes the current path expression. We then go through the series of tests in the definition of *extend_graph_vcs*. We first test to see if this path expression is **true**, which it clearly is not.

The second test is whether the node just reached backwards across the arc is the same as the starting node. In this case, the node just reached is *even* and the starting node is *even*, so this test succeeds.

According to the definition of *extend_graph_vcs*, for satisfying this test, we generate a verification condition of the form

$$\begin{array}{l}\textbf{let}\ (vars, vals, glbs, pre, post, calls, rec, c) = \rho\ p_0\ \textbf{in} \\ \quad\quad [\ pre \wedge induct_pre\ rec \Rightarrow q_1\ ],\end{array}$$

which in this case is

$$\begin{array}{l}(\mathbf{true}\ \wedge\ n = \widehat{n}) \Rightarrow \\ (\forall a_1, n_1.\ (n_1 < n) \Rightarrow (n_1 < \widehat{n})).\end{array}$$

This is another *undiverted recursion verification condition.*

This terminates this exploration of this path (Case 2) through the procedure call graph. Since this is also the last case, this also terminates the exploration of the procedure call graph for paths rooted at *even*.

This ends the example.

7.1.4 Verification of Programs

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

Figure 7.9: Definition of *mkenv*.

The *mkenv* function is presented in Figure 7.9. This function has type `decl` → `env` → `env`. *mkenv* takes a declaration and an environment, and returns a new environment containing all of the declarations of procedures present in the declaration argument, overriding the declarations of those procedures already present in the environment.

$$
\begin{array}{l}
\mathit{proc_names}\ (\textbf{proc}\ p\ \mathit{vars}\ \mathit{vals}\ \mathit{glbs}\ \mathit{pre}\ \mathit{post}\ \mathit{calls}\ \mathit{rec}\ c) \ = \ [p] \\
\mathit{proc_names}\ (d_1\,;\, d_2) \ = \ \mathit{proc_names}\ d_1\ \&\ \mathit{proc_names}\ d_2 \\
\mathit{proc_names}\ (\textbf{empty}) \ = \ [\,]
\end{array}
$$

Figure 7.10: Definition of *proc_names*.

The *proc_names* function is presented in Figure 7.10. This function has type `decl` $\rightarrow$ `(string)list`. *proc_names* takes a declaration, and returns the list of procedure names that are declared in the declaration.

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

Figure 7.11: Definition of *vcg*, the main VCG function.

vcg is the main VCG function, presented in Figure 7.11. *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 whose proofs are sufficient to prove the program totally correct with respect to the given postcondition. *mkenv* creates the procedure environment that corresponds to a declaration using the empty procedure environment ρ_0 (with all procedures undeclared), and g_0 is the "empty" call progress environment $\lambda p.\ \textbf{true}$.

7.2 Verification Conditions

In the functions presented above, the essential task is constructing a proof of the program, but this proof is implicit and not actually produced as a result. Rather, the primary results are verification conditions, whose proof verifies the construct analyzed.

In [Gri81], Gries gives an excellent presentation of a methodology for developing programs and proving them correct. He lists many principles to guide and strengthen this process. The first and primary principle he lists is

> **Principle**: A program and its proof should be developed hand-in-hand, with the *proof* usually leading the way.

In [AA78], Alagić and Arbib establish the following method of top-down design of an algorithm to solve a given problem:

> **Principle**: Decompose the overall problem into precisely specified subproblems, and prove that if each subproblem is solved correctly *and these solutions are fitted together in a specified way* then the original problem will be solved correctly. Repeat the process of "decompose and prove correctness of the decomposition" for the subproblems; and keep repeating this process until reaching subproblems so simple that their solution can be expressed in a few lines of a programming language.

We would like to summarize these in our own principle:

Principle: The structure of the proof should match the structure of the program.

In the past, verification condition generators have concentrated exclusively on the structure of the syntax of the program, decomposing commands into their subcommands, and constructing the proof with the same structure based on the syntax, so that the proof and the program mirror each other.

We continue that tradition in this work, but we also recognize that an additional kind of structure exists in programs with procedures, the structure of the procedure call graph. This is a perfectly valid kind of structure, and it provides an opportunity to structure part of the proof of a program's correctness. In particular, it is the essential structure we use to prove the recursive progress claims of procedures.

In our opinion, wherever a natural and inherent kind of structure is recognized in a class of programs, it is worth examining to see if it may be useful in structuring proofs of properties about those programs. Such structuring regularizes proofs and reduces their *ad hoc* quality. In addition, it may provide opportunities to prove general results about all programs with that kind of structure, moving a part of the proof effort to the meta-level, so that it need not be repeated for each individual program being proven.

7.2.1 Program Structure Verification Conditions

The functions $vcg1$, $vcgc$, and $vcgd$ are defined recursively, based on the recursive syntactic structure of the program constructs involved. An examination of the definitions of $vcg1$, $vcgc$, and $vcgd$ (Figures 7.1, 7.2, 7.3) reveals several instances

where verification conditions are generated in this analysis of the syntactic structure. The thrust of the work done by $vcg1$ is to transform the postcondition argument into an appropriate precondition, but it also generates two verification conditions for the iteration command. $vcgc$ takes the verification conditions generated by $vcg1$, and adds one new one, making sure the given precondition implies the precondition computed by $vcg1$. $vcgd$ invokes $vcgc$ on the body of each procedure declared, and collects the resulting verification conditions into a single list. All of these verification conditions were generated at appropriate places in the syntactic structure of the program.

Principally, the purpose of these verification conditions is to establish the partial correctness of the constructs involved, with respect to the preconditions and postconditions present. In addition, however, a careful examination of the procedure call clause in the definition of $vcg1$ in Figure 7.1 discloses that the $pre \wedge calls\ p$ phrase occuring there ensures that both pre and $calls\ p$ must be true upon entry to the procedure being called. Thus the preconditions generated by $vcg1$, and incorporated by $vcgc$ and $vcgd$, carry the strength of being able to ensure both that the preconditions of any called procedures are fulfilled, and that the call progress specified in the $calls$ argument is fulfilled. For $vcgd$, this means that that the preconditions of declared procedures are fulfilled, and the call progress claimed in the header of each procedure declared has been verified. From the partial correctness that they imply, it is then possible to prove for each of these VCG functions that the command involved terminates if all of its immediate calls terminate. Thus it is possible to reason simply from the verification conditions generated by this syntactic analysis and conclude four essential properties of the procedure environment ρ:

$WF_{envp}\ \rho$	ρ is well-formed for partial correctness
$WF_{env_pre}\ \rho$	ρ is well-formed for preconditions
$WF_{env_calls}\ \rho$	ρ is well-formed for calls progress
$WF_{env_term}\ \rho$	ρ is well-formed for conditional termination

7.2.2 Call Graph Structure Verification Conditions

In this dissertation, we have introduced functions as part of the verification condition generator to analyze the structure of the procedure call graph. The goal of this graph analysis is to prove that every recursive call, reentering a procedure that had been called before and has not yet finished, demonstrates some measurable degree of progress. This progress is quantified in the **recurses with** clause in the procedure declaration's header. The expression given in this clause is either **false**, signifying that no recursion is allowed, or $v < x$, where v is an assertion language numeric expression, and where x is a logical variable. The exact choice of x is not vital, merely that it serve as a name for the prior value of v at the first call of the procedure, so that it may be compared with the eventual value of v at the recursive call.

The progress described by $v < x$ is the decrease of an integer expression. In the Sunrise language, this is restricted to nonnegative integer values. The nonnegative integers form a well-founded set with $<$ as its ordering. By the definition of well-founded sets, there does not exist any infinite decreasing sequence of values from a well-founded set. Hence there cannot be an infinite number of times that the expression v decreases before it reaches 0, and thus we will eventually be able to argue that any call of the procedure must terminate. However, at this point we are only trying to establish the recursive progress between recursive invocations of the procedure, that v has strictly decreased.

To prove this recursive progress, we need to consider every possible path of procedure calls from the procedure to itself. Given the possible presence of cycles in the procedure call graph, there may be an infinite number of such paths, all of which cannot be examined in finite time. However, in our research, we have discovered a small, finite number of verification conditions which together cover every possible path, even if the paths are infinite in number. These verification conditions are of two kinds, which we call *undiverted recursion verification conditions* and *diversion verification conditions*.

To understand the intent of these verification conditions, as a first step consider the possibility of exploring the procedure call graph to find paths that correspond to recursive calls. Starting from a designated procedure and exploring backwards across arcs in the graph yields an expanding tree of procedure calls, where the root of the tree is the starting procedure. If cycles are present in the graph, this tree will grow to be infinite in extent. An example of such a tree is presented in Figure 7.12.

Now examine this infinite tree of procedure calls. Some of the nodes in the tree duplicate the root node, that is, they refer to the same procedure. We call these occurrences instances of *recursion*. Of these duplicate nodes, consider the paths from each node to the root. Some of these paths will themselves contain internally another duplicate of the root, and some will not. Those that do not contain another duplicate of the root we call instances of *single recursion*. The other paths, that do contain additional duplicates of the root, we call instances of *multiple recursion*. Observe that each instance of multiple recursion is a chaining together of multiple instances of single recursion. In addition, if the progress

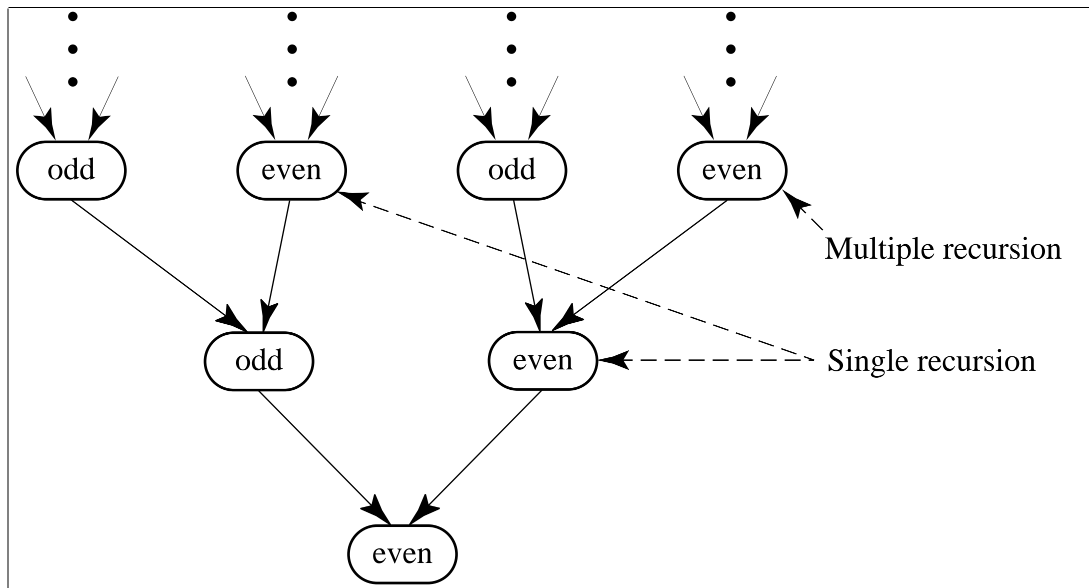

Figure 7.12: Procedure Call Tree for Recursion for Odd/Even Example.

claimed by the recursion expression for the root procedure is achieved for each instance of single recursion, then the progress achieved for each instance of multiple recursion will be the accumulation of the progresses of each constituent instance of single recursion, and thus should also satisfy the progress claim even more easily.

So the problem of proving the recursive progress for all recursive paths simplifies to proving it for all singly recursive paths. Now, there still may be an infinite number of singly recursive paths in the procedure call tree. For instance, in the odd/even program example, if we consider all singly recursive paths with root at *even*, the presence of the self-loop at *odd* means that there are an infinite number of paths with different numbers of times around that self-loop involved. This tree is presented in Figure 7.13. Singly recursive paths traverse the call graph from

even to *odd*, then to *odd* via an indefinite number of times around the self-loop, and finally to *even*.

Figure 7.13: Procedure Call Tree for Single Recursion for Odd/Even Example.

Consider the procedure call tree as before but limited now in its expansion to singly recursive paths, so that the only occurrences of the root node are at the root and as leaves. None of the internal nodes of the tree duplicate the root node. However, for any particular leaf node and the path from that leaf to the root, there may be duplicates within that list, not involving the root node. If there are duplicates, say two occurences of a procedure p not the root, then we call this

an instance of *diverted recursion*, and we call the part of the path between the two occurrences of p a *diversion*. Intuitively this name suggests that the search for recursive paths from the root procedure to itself became diverted from that goal when the search reached p. For a while the search followed the cycle from p to p, and only when it returned to p did it resume again to head for the root procedure. In contrast, we call a path from a leaf to the root which does not have any examples of diversion an instance of *undiverted recursion*. These instances of undiverted recursion would be the occasions of generating verification conditions to verify the recursion expression claim, except that the tree is still infinite.

Now, given a diversion involving the procedure p, we observe that the subtrees of the procedure call tree rooted at the two instances of p are identical in their branching structure. The only things that change are the path conditions attached to the various nodes. Except for these, one could copy one of the subtrees, move it so that it was superimposed on the other subtree, and the two would look identical. This provides the motivation for the final simplification here, the introduction of *diversion verification conditions*. We can implicitly cover the infinite expansion of the procedure call tree for single recursion by looking for cases of diversion as we expand the tree, and then for each case, *bending* the endpoint of the diversion farthest from the root around and *connecting* it to the near endpoint of the diversion. The connection we establish is the generation of a verification condition, that the path condition at the near endpoint implies the path condition at the far endpoint. Compare Figures 7.13 and 7.14 to see an example of this for the odd/even program.

At first, this may seem counter-intuitive, or even bizzare, and we confess this

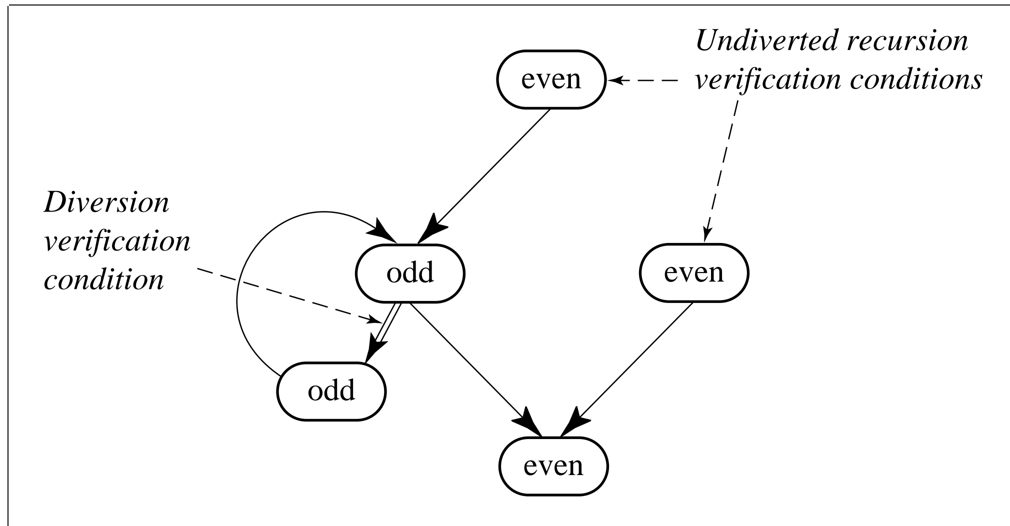

Figure 7.14: Diverted and Undiverted Verification Conditions for Odd/Even.

was how the idea struck us initially. Since the far endpoint is previous in time to the near endpoint, one would normally expect any implication to flow from the prior to the later. However, in this case what the diversion verification condition is saying is that the changes to the path expressions imposed by moving around the diversion cycle in the graph do not interfere with justifying the recursive progress claim for the root procedure. In other words, we do not lose ground by going around a diversion cycle, but instead the cycle either has no effect or a positive effect. In terms of the procedure call tree, making this connection between the endpoints of a diversion is tantamount to copying the entire subtree rooted at the nearer endpoint and attaching the root of the copy at the farther endpoint. Since the copied subtree includes the farther endpoint within it, this creates an infinite expansion, fully covering the infinite singly recursive procedure call tree. However, since there is only one verification condition per diversion required to achieve this, we have reduced the proof burden imposed on the programmer

to a finite number of verification conditions, which now consist of a mixture of undiverted recursion verification conditions for leaves of the expansion which match the root, and diversion verification conditions for leaves of the expansion which match another node along the path to the root.

7.3 VCG Soundness Theorems

The verification condition generator functions defined in the first section of this chapter are simple syntactic manipulations of expressions as data. For this to have any reliable use, we must establish the semantics of these syntactic manipulations. We have done this in this dissertation by proving theorems within the HOL system that describe the relationship between the verification conditions produced by these functions and the correctness of the programs with respect to their specifications. These theorems are proven at the meta-level, which means that they hold for all programs that may be submitted to the VCG.

The VCG theorems that have been proven related to the *vcg*1 function are listed in Table 7.1. There are seven theorems listed, which correspond to seven ways that the results of the *vcg*1 function are used to prove various kinds of correctness about commands. `vcg1_0_THM` and `vcg1_k_THM` are the proof of *staged* versions of the partial correctness of commands, necessary steps in proving the full partial correctness. These stages and the process of proving the partial correctness of every procedure, $WF_{envp}\ \rho$, is described in Section 10.5 on Semantic Stages. Given these two theorems, it is possible to prove `vcg1p_THM`, which verifies that if the verification conditions produced by *vcg*1 are true, then the partial correctness of the command analyzed follows. Furthermore, it is possible to prove `vcg1_PRE_PROGRESS` and `vcg1_BODY_PROGRESS`, which state that if the verification conditions are true, then the preconditions of all called procedures hold, and the progress conditions contained in *calls* also hold. Beyond this, `vcg1_TERM` shows that the command conditionally terminates if all immediate calls terminate. Finally, if the environment ρ has been shown to be completely well formed, then

vcg1_0_THM	$\forall c\ calls\ q\ \rho.\ \ WF_{env_syntax}\ \rho\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\mathbf{let}\ (p,h) = vcg1\ c\ calls\ q\ \rho\ \mathbf{in}$ $(\mathbf{all_el\ close}\ h \Rightarrow \{p\}\ c\ \{q\}\ /\rho,\ 0)$
vcg1_k_THM	$\forall c\ calls\ q\ \rho\ k.\ WF_{envk}\ \rho\ k\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\mathbf{let}\ (p,h) = vcg1\ c\ calls\ q\ \rho\ \mathbf{in}$ $(\mathbf{all_el\ close}\ h \Rightarrow \{p\}\ c\,\{q\}/\rho, k+1)$
vcg1p_THM	$\forall c\ calls\ q\ \rho.\ \ WF_{envp}\ \rho\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\mathbf{let}\ (p,h) = vcg1\ c\ calls\ q\ \rho\ \mathbf{in}$ $(\mathbf{all_el\ close}\ h \Rightarrow \{p\}\ c\ \{q\}\ /\rho)$
vcg1_PRE_PROGRESS	$\forall c\ calls\ q\ \rho.\ \ WF_{envp}\ \rho\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\mathbf{let}\ (p,h) = vcg1\ c\ calls\ q\ \rho\ \mathbf{in}$ $(\mathbf{all_el\ close}\ h \Rightarrow \{p\}\ c \rightarrow \mathbf{pre}\ /\rho)$
vcg1_BODY_PROGRESS	$\forall c\ calls\ q\ \rho.\ \ WF_{envp}\ \rho\ \wedge\ WF_{calls}\ calls\ \rho$ $\wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\mathbf{let}\ (p,h) = vcg1\ c\ calls\ q\ \rho\ \mathbf{in}$ $(\mathbf{all_el\ close}\ h \Rightarrow \{p\}\ c \rightarrow calls\ /\rho)$
vcg1_TERM	$\forall c\ calls\ q\ \rho.\ \ WF_{envp}\ \rho\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\mathbf{let}\ (p,h) = vcg1\ c\ calls\ q\ \rho\ \mathbf{in}$ $(\mathbf{all_el\ close}\ h \Rightarrow [p]\ c \downarrow /\rho)$
vcg1_THM	$\forall c\ calls\ q\ \rho.\ \ WF_{env}\ \rho\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\mathbf{let}\ (p,h) = vcg1\ c\ calls\ q\ \rho\ \mathbf{in}$ $(\mathbf{all_el\ close}\ h \Rightarrow [p]\ c\ [q]/\rho)$

Table 7.1: Theorems of verification of commands using the $vcg1$ function.

vcg1_THM states that if all the verification conditions are true, then the command is totally correct with respect to the computed precondition and the given postcondition.

The VCG theorems that have been proven related to the *vcgc* function are listed in Table 7.2. These are similar to the theorems proven for $vcg1$. There are seven theorems listed, which correspond to seven ways that the results of the *vcgc* function are used to prove various kinds of correctness about commands. vcgc_0_THM and vcgc_k_THM are the proof of *staged* versions of the partial correctness of commands, necessary steps in proving the full partial correctness. These stages and the process of proving the partial correctness of every procedure, $WF_{envp}\ \rho$, is described in Section 10.5 on Semantic Stages. Given these two theorems, it is possible to prove vcgcp_THM, which verifies that if the verification conditions produced by *vcgc* are true, then the partial correctness of the command analyzed follows. Furthermore, it is possible to prove vcgc_PRE_PROGRESS and vcgc_BODY_PROGRESS, which state that if the verification conditions are true, then the preconditions of all called procedures hold, and the progress conditions contained in *calls* also hold. Beyond this, vcgc_TERM shows that the command conditionally terminates if all immediate calls terminate. Finally, if the environment ρ has been shown to be completely well formed, then vcgc_THM states that if all the verification conditions are true, then the command is totally correct with respect to the given precondition and postcondition.

`vcgc_0_THM`	$\forall c\ p\ calls\ q\ \rho.\ \ WF_{env_syntax}\ \rho\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\quad\mathbf{all_el\ close}\ (vcgc\ p\ c\ calls\ q\ \rho) \Rightarrow$ $\quad\{p\}\ c\ \{q\}\ /\rho,\ 0$
`vcgc_k_THM`	$\forall c\ p\ calls\ q\ \rho\ k.\ WF_{envk}\ \rho\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\quad\mathbf{all_el\ close}\ (vcgc\ p\ c\ calls\ q\ \rho) \Rightarrow$ $\quad\{p\}\ c\ \{q\}\ /\rho,\ k+1$
`vcgcp_THM`	$\forall c\ p\ calls\ q\ \rho.\ \ WF_{envp}\ \rho\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\quad\mathbf{all_el\ close}\ (vcgc\ p\ c\ calls\ q\ \rho) \Rightarrow$ $\quad\{p\}\ c\ \{q\}\ /\rho$
`vcgc_PRE_PROGRESS`	$\forall c\ p\ calls\ q\ \rho.\ \ WF_{envp}\ \rho\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\quad\mathbf{all_el\ close}\ (vcgc\ p\ c\ calls\ q\ \rho) \Rightarrow$ $\quad\{p\}\ c \rightarrow \mathbf{pre}\ /\rho$
`vcgc_BODY_PROGRESS`	$\forall c\ p\ calls\ q\ \rho.\ \ WF_{envp}\ \rho\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\quad\mathbf{all_el\ close}\ (vcgc\ p\ c\ calls\ q\ \rho) \Rightarrow$ $\quad\{p\}\ c \rightarrow calls\ /\rho$
`vcgc_TERM`	$\forall c\ p\ calls\ q\ \rho.\ \ WF_{envp}\ \rho\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\quad\mathbf{all_el\ close}\ (vcgc\ p\ c\ calls\ q\ \rho) \Rightarrow$ $\quad[p]\ c\downarrow\ /\rho$
`vcgc_THM`	$\forall c\ p\ calls\ q\ \rho.\ \ WF_{env}\ \rho\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\quad\mathbf{all_el\ close}\ (vcgc\ p\ c\ calls\ q\ \rho) \Rightarrow$ $\quad[p]\ c\ [q]\ /\rho$

Table 7.2: Theorems of verification of commands using the *vcgc* function.

The VCG theorems that have been proven related to the *vcgd* function for declarations are listed in Table 7.3. These are similar in purpose to the theorems proven for *vcg*1 and *vcgc*. There are seven theorems listed, which correspond to seven ways that the results of the *vcgd* function are used to prove various kinds of correctness about declarations. `vcgd_syntax_THM` shows that if a declaration is well-formed syntactically and the verification conditions returned by *vcgd* are true, then the corresponding procedure environment is well-formed syntactically. `vcgd_0_THM` and `vcgd_k_THM` are the proof of *staged* versions of the partial correctness of declarations, necessary steps in proving the full partial correctness. These stages and the process of proving the partial correctness of every procedure, $WF_{envp}\ \rho$, is described in Section 10.5 on Semantic Stages. Given these two theorems, it is possible to prove `vcgd_THM`, which verifies that if the verification conditions produced by *vcgd* are true, then the partial correctness of the environment follows. Furthermore, it is possible to prove `vcgd_PRE_PROGRESS` and `vcgd_BODY_PROGRESS`, which state that if the verification conditions are true, then the environment is well-formed for preconditions and for calls progress. Finally, `vcgd_TERM` shows that if all the verification conditions are true, then every procedure in the environment conditionally terminates if all immediate calls from its body terminate.

The VCG theorems that have been proven related to the graph exploration functions for the procedure call graph are given in the following tables. The theorem about $fan_out_graph_vcs$ is listed in Table 7.4. It essentially states that if the verification conditions returned by $fan_out_graph_vcs$ are true, then for every possible extension of the current path to a leaf node, if it is a leaf corresponding to an instance of undiverted recursion, then the undiverted recursion verifica-

vcgd_syntax_THM	$\forall d\ \rho.\ \ \rho = mkenv\ d\ \rho_0\ \wedge\ WF_d\ d\ \rho\ \wedge$ **all_el close** $(vcgd\ d\ \rho) \Rightarrow$ $WF_{env_syntax}\ \rho$
vcgd_0_THM	$\forall d\ \rho.\ \ \rho = mkenv\ d\ \rho_0\ \wedge\ WF_d\ d\ \rho\ \wedge$ **all_el close** $(vcgd\ d\ \rho) \Rightarrow$ $WF_{envk}\ \rho\ 0$
vcgd_k_THM	$\forall d\ \rho\ k.\ \rho = mkenv\ d\ \rho_0\ \wedge\ WF_d\ d\ \rho\ \wedge$ **all_el close** $(vcgd\ d\ \rho) \Rightarrow$ $WF_{envk}\ \rho\ k \Rightarrow$ $WF_{envk}\ \rho\ (k+1)$
vcgd_THM	$\forall d\ \rho.\ \ \rho = mkenv\ d\ \rho_0\ \wedge\ WF_d\ d\ \rho\ \wedge$ **all_el close** $(vcgd\ d\ \rho) \Rightarrow$ $WF_{envp}\ \rho$
vcgd_PRE_PROGRESS	$\forall d\ \rho.\ \ \rho = mkenv\ d\ \rho_0\ \wedge\ WF_d\ d\ \rho\ \wedge$ **all_el close** $(vcgd\ d\ \rho) \Rightarrow$ $WF_{env_pre}\ \rho$
vcgd_BODY_PROGRESS	$\forall d\ \rho.\ \ \rho = mkenv\ d\ \rho_0\ \wedge\ WF_d\ d\ \rho\ \wedge$ **all_el close** $(vcgd\ d\ \rho) \Rightarrow$ $WF_{env_calls}\ \rho$
vcgd_TERM	$\forall d\ \rho.\ \ \rho = mkenv\ d\ \rho_0\ \wedge\ WF_d\ d\ \rho\ \wedge$ **all_el close** $(vcgd\ d\ \rho) \Rightarrow$ $WF_{env_term}\ \rho$

Table 7.3: Theorems of verification of declarations using the *vcgd* function.

tion condition is true, and if the leaf corresponds to an instance of diversion, then the diversion verification condition is true. In brief, this theorem says that $fan_out_graph_vcs$ produces all the verification conditions previously described as arising from the current point on in the exploration of the call graph.

The theorem about the verification of $graph_vcs$ is listed in Table 7.5. It essentially states that if the verification conditions returned by $graph_vcs$ are true, then for every instance of undiverted recursion, the undiverted recursion verification condition is true, and for every instance of diversion, the diversion verification condition is true. In brief, this theorem says that $graph_vcs$ produces all the verification conditions previously described as arising from a particular starting node in the exploration of the call graph.

Given that $graph_vcs$ collects the proper set of verification conditions, we can now prove that for all instances of single recursion, if the verification conditions returned by $graph_vcs$ are true, then the initial value of the recursion expression for a procedure implies the precondition computed by the $call_path_progress$ function (defined in Table 6.12), as shown in Table 7.6. The proof proceeds by well-founded induction on the length of the path ps.

Now, in the previous chapter a rule was presented that $call_path_progress$ returned appropriate preconditions for path entrance specifications. We can now prove path entrance specifications for all possible paths starting from a procedure to a recursive call of the same procedure, where the precondition at the original entrance of the procedure is $induct_pre\ rec$, and the entrance condition at all the eventual recursive entrances of the procedure is rec. If rec is of the form $v < x$, then $induct_pre\ rec$ is $v = x$, and these path entrance specifications declare

$$
\begin{array}{l}
\forall\ n\ p\ ps\ p_0\ q\ pcs\ \rho\ all_ps\ y\ z \\
\quad vars\ vals\ glbs\ pre\ post\ calls\ rec\ c \\
\quad vars'\ vals'\ glbs'\ pre'\ post'\ calls'\ rec'\ c'\ . \\
WF_{env_syntax}\ \rho\ \wedge \\
WF_{env_pre}\ \rho\ \wedge \\
WF_{env_calls}\ \rho\ \wedge \\
\rho\ p_0 = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle\ \wedge \\
p_0 \in SL\ all_ps\ \wedge \\
(\forall p'.\ (p' \notin SL\ all_ps) \Rightarrow (\rho\ p' = \rho_0\ p'))\ \wedge \\
LENGTH\ all_ps = LENGTH\ ps + n\ \wedge \\
p_0 \in SL\ (CONS\ p\ ps)\ \wedge \\
SL\ (CONS\ p\ ps) \subseteq SL\ all_ps\ \wedge \\
DL\ (CONS\ p\ ps)\ \wedge \\
\rho\ p = \langle vars', vals', glbs', pre', post', calls', rec', c'\rangle\ \wedge \\
y = vars'\ \&\ vals'\ \&\ glbs'\ \wedge \\
FV_a\ q \subseteq SL\ (y\ \&\ logicals\ z)\ \wedge \\
((ps = [\,]) \Rightarrow (q = rec))\ \wedge \\
(\forall ps'.\ (ps = ps'\ \&\ [p_0]) \Rightarrow \\
\qquad\quad (q = call_path_progress\ p\ ps'\ p_0\ rec\ \rho))\ \wedge \\
(\forall p'.\ p' \in SL\ (CONS\ p\ ps)\ \ \wedge\ \ p' \neq p_0 \Rightarrow \\
\qquad (\forall ps_2\ ps_1.\ (CONS\ p\ ps = ps_2\ \&\ (CONS\ p'\ (ps_1\ \&\ [p_0]))) \Rightarrow \\
\qquad\qquad\qquad (pcs\ p' = call_path_progress\ p'\ ps_1\ p_0\ rec\ \rho)))\ \wedge \\
\textbf{all_el close}\ (fan_out_graph_vcs\ p\ ps\ p_0\ q\ pcs\ \rho\ all_ps\ n) \\
\Rightarrow \\
\\
(\forall ps'. \\
\qquad DL\ ps'\ \ \wedge\ \ DISJOINT\ (SL\ ps')\ (SL\ (CONS\ p\ ps))\ \ \Rightarrow \\
\qquad \textbf{close}\ (pre \wedge induct_pre\ rec \Rightarrow \\
\qquad\qquad call_path_progress\ p_0\ ps'\ p\ q\ \rho))\ \wedge \\
(\forall p_1\ ps'\ ps_1\ ps_2. \\
\qquad (p_1 \neq p_0)\ \wedge \\
\qquad DL\ ps'\ \wedge \\
\qquad DISJOINT\ (SL\ ps')\ (SL\ (CONS\ p\ ps))\ \wedge \\
\qquad (ps'\ \&\ (CONS\ p\ ps) = ps_2\ \&\ (CONS\ p_1\ (ps_1\ \&\ [p_0])))\ \ \Rightarrow \\
\qquad \textbf{close}\ (call_path_progress\ p_1\ ps_1\ p_0\ rec\ \rho \Rightarrow \\
\qquad\qquad call_path_progress\ p_1\ (ps_2\ \&\ (CONS\ p_1\ ps_1))\ p_0\ rec\ \rho))
\end{array}
$$

Table 7.4: Theorem of verification condition collection by $fan_out_graph_vcs$.

$$
\begin{array}{l}
\forall p\ \rho\ all_ps\ vars\ vals\ glbs\ pre\ post\ calls\ rec\ c. \\
\quad WF_{env_syntax}\ \rho\ \wedge \\
\quad WF_{env_pre}\ \rho\ \wedge \\
\quad WF_{env_calls}\ \rho\ \wedge \\
\quad (\forall p'.\ (p' \notin SL\ all_ps) \Rightarrow (\rho\ p' = \rho_0\ p'))\ \wedge \\
\quad p \in SL\ all_ps\ \wedge \\
\quad \rho\ p = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle\ \wedge \\
\quad \textbf{all_el close}\ (graph_vcs\ all_ps\ \rho\ p) \\
\quad \Rightarrow \\
\\
\quad (\forall ps. \\
\qquad\quad DL\ (ps\ \&\ [p])\ \ \Rightarrow \\
\qquad\quad \textbf{close}\ (pre \wedge induct_pre\ rec \Rightarrow \\
\qquad\qquad\quad call_path_progress\ p\ ps\ p\ rec\ \rho))\ \wedge \\
\quad (\forall p'\ ps\ ps_1\ ps_2. \\
\qquad\quad (p' \neq p)\ \wedge \\
\qquad\quad DL\ ps\ \wedge \\
\qquad\quad p \notin\ SL\ ps\ \wedge \\
\qquad\quad ps = ps_2\ \&\ (CONS\ p'\ ps_1)\ \ \Rightarrow \\
\qquad\quad \textbf{close}\ (call_path_progress\ p'\ ps_1\ p\ rec\ \rho \Rightarrow \\
\qquad\qquad\quad call_path_progress\ p'\ (ps_2\ \&\ (CONS\ p'\ ps_1))\ p\ rec\ \rho))
\end{array}
$$

Table 7.5: Theorem of verification condition collection by *graph_vcs*.

$$
\begin{array}{l}
\forall n\ ps\ p\ \rho\ all_ps\ vars\ vals\ glbs\ pre\ post\ calls\ rec\ c. \\
\quad WF_{env_syntax}\ \rho\ \wedge \\
\quad WF_{env_pre}\ \rho\ \wedge \\
\quad WF_{env_calls}\ \rho\ \wedge \\
\quad (\forall p'.\ (p' \notin SL\ all_ps) \Rightarrow (\rho\ p' = \rho_0\ p'))\ \wedge \\
\quad LENGTH\ ps = n\ \wedge \\
\quad p \in SL\ all_ps\ \wedge \\
\quad p \notin SL\ ps\ \wedge \\
\quad \rho\ p = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle\ \wedge \\
\quad \textbf{all_el close}\ (graph_vcs\ all_ps\ \rho\ p) \\
\quad \Rightarrow \\
\quad \textbf{close}\ (pre \wedge induct_pre\ rec \Rightarrow \\
\qquad\qquad call_path_progress\ p\ ps\ p\ rec\ \rho)
\end{array}
$$

Table 7.6: Theorem of verification of single recursion by *call_path_progress*.

that the recursive expression v strictly decreases across every possible instance of single recursion of that procedure. This theorem is shown in Table 7.7.

Using the transitivity of $<$, we can now prove the verification of all recursion, single and multiple, by well-founded induction on the length of the path *ps*. This theorem is shown in Table 7.8.

We can now describe the verification of recursion given the verification conditions returned by *graph_vcs*, in Table 7.9.

This allows us to verify the recursion of all declared procedures by the main call graph analysis function, *vcgg*, as described in Table 7.10.

Finally, this allows us to verify the main call graph analysis function, *vcgg*, as described in Table 7.11.

We will show later how the progress described in the recursive progress claims enables the proof of the termination of procedures. This is a particularly inter-

$$
\begin{array}{l}
\forall ps\ p\ \rho\ all_ps\ vars\ vals\ glbs\ pre\ post\ calls\ rec\ c. \\
\quad WF_{env_syntax}\ \rho\ \wedge \\
\quad WF_{env_pre}\ \rho\ \wedge \\
\quad WF_{env_calls}\ \rho\ \wedge \\
\quad (\forall p'.\ (p' \notin SL\ all_ps) \Rightarrow (\rho\ p' = \rho_0\ p'))\ \wedge \\
\quad p \in SL\ all_ps\ \wedge \\
\quad p \notin SL\ ps\ \wedge \\
\quad \rho\ p = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle\ \wedge \\
\quad \textbf{all_el close}\ (graph_vcs\ all_ps\ \rho\ p) \\
\quad \Rightarrow \\
\quad \{pre \wedge induct_pre\ rec\}\ p - ps \rightarrow p\ \{rec\}\ /\rho
\end{array}
$$

Table 7.7: Theorem of verification of all single recursion.

$$
\begin{array}{l}
\forall n\ ps\ p\ \rho\ all_ps\ vars\ vals\ glbs\ pre\ post\ calls\ rec\ c. \\
\quad WF_{env_syntax}\ \rho\ \wedge \\
\quad WF_{env_pre}\ \rho\ \wedge \\
\quad WF_{env_calls}\ \rho\ \wedge \\
\quad (\forall p'.\ (p' \notin SL\ all_ps) \Rightarrow (\rho\ p' = \rho_0\ p'))\ \wedge \\
\quad LENGTH\ ps = n\ \wedge \\
\quad p \in SL\ all_ps\ \wedge \\
\quad \rho\ p = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle\ \wedge \\
\quad \textbf{all_el close}\ (graph_vcs\ all_ps\ \rho\ p) \\
\quad \Rightarrow \\
\quad \{pre \wedge induct_pre\ rec\}\ p - ps \rightarrow p\ \{rec\}\ /\rho
\end{array}
$$

Table 7.8: Theorem of verification of all recursion, single and multiple.

$$
\begin{array}{l}
\forall p\ \rho\ all_ps\ vars\ vals\ glbs\ pre\ post\ calls\ rec\ c.\\
\quad WF_{env_syntax}\ \rho\ \wedge\\
\quad WF_{env_pre}\ \rho\ \wedge\\
\quad WF_{env_calls}\ \rho\ \wedge\\
\quad (\forall p'.\ (p' \notin SL\ all_ps) \Rightarrow (\rho\ p' = \rho_0\ p'))\ \wedge\\
\quad p \in SL\ all_ps\ \wedge\\
\quad \rho\ p = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle\ \wedge\\
\quad \textbf{all_el close}\ (graph_vcs\ all_ps\ \rho\ p)\\
\quad \Rightarrow\\
\quad \{pre \wedge induct_pre\ rec\}\ p \hookleftarrow \{rec\}\ /\rho
\end{array}
$$

Table 7.9: Theorem of verification of recursion by $graph_vcs$.

$$
\begin{array}{l}
\forall \rho\ all_ps.\\
\quad WF_{env_syntax}\ \rho\ \wedge\\
\quad WF_{env_pre}\ \rho\ \wedge\\
\quad WF_{env_calls}\ \rho\ \wedge\\
\quad (\forall p'.\ (p' \notin SL\ all_ps) \Rightarrow (\rho\ p' = \rho_0\ p'))\ \wedge\\
\quad \textbf{all_el close}\ (vcgg\ all_ps\ \rho)\\
\quad \Rightarrow\\
\quad (\forall p.\ \textbf{let}\ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle = \rho\ p\ \textbf{in}\\
\qquad\quad \{pre \wedge induct_pre\ rec\}\ p \hookleftarrow \{rec\}\ /\rho)
\end{array}
$$

Table 7.10: Theorem of verification of recursion by $vcgg$.

$$
\begin{array}{l}
\forall d\ \rho.\\
\quad \rho = mkenv\ d\ \rho_0\ \wedge\\
\quad WF_{env_syntax}\ \rho\ \wedge\\
\quad WF_{env_pre}\ \rho\ \wedge\\
\quad WF_{env_calls}\ \rho\ \wedge\\
\quad \textbf{all_el close}\ (vcgg\ (proc_names\ d)\ \rho)\ \Rightarrow\\
\quad WF_{env_rec}\ \rho
\end{array}
$$

Table 7.11: Theorem of verification of $vcgg$.

esting part of the verification of the VCG, and possibly the deepest theoretically. It is described in Section 11.2.

At last, we come to the main theorem of the correctness of the verification condition generator. This is our ultimate theorem and our primary result. It is given in Table 7.12.

$$\forall \pi\ q.\ \ WF_p\ \pi\ \wedge\ \mathbf{all_el\ close}\ (vcg\ \pi\ q)\ \Rightarrow\ \pi\ [q]$$

Table 7.12: Theorem of verification of verification condition generator.

This verifies the verification condition generator. It shows that the *vcg* function is *sound*, that the correctness of the verification conditions it produces suffice to establish the total correctness of the annotated program. This does not show that the *vcg* function is *complete*, namely that if a program is correct, then the *vcg* function will produce a set of verification conditions sufficient to prove the program correct from the axiomatic semantics. However, this soundness result is quite useful, in that we may directly apply these theorems in order to prove individual programs totally correct within HOL, as seen in the next chapter.