A User's Guide to Proving Programs Correct with the Sunrise Verification System version 7.3

Peter Vincent Homeier
homeier@saul.cis.upenn.edu
http://www.cis.upenn.edu/~homeier

February 9, 2005

Contents

Chapter 1

Introduction

This document is a guide to verifying the correctness of imperative programs written in the Sunrise programming language, using the Sunrise package within the Higher Order Logic (HOL) theorem prover.

Various features of the Sunrise package are described, including the syntax and features of the Sunrise language, how to create terms in the HOL logic corresponding to Sunrise programs, how to establish program correctness goals to be proven, and how to invoke the automatic tools provided to help verify a given program. These tools are examples of *Verification Condition Generators* (VCGs), which construct a substantial portion of the proof of a program's correctness, leaving a residue of lemmas to be proven by the user using traditional HOL techniques. When these lemmas are proven, this completes the verification of the original program's correctness.

This is a great aid to the programmer, since it reduces significantly the effort and detail involved in finding a proof of a program's correctness. The remainder left for the programmer to prove is also simpler, in that it does not deal with any concepts arising from the programming language itself, such as loops or recursion, but only concerning the logics of the underlying datatypes, such as arithmetic.

This VCG technology is not new in its concepts, dating from the early 1970's. One problem with this VCG approach is that the VCG is itself a program, and subject to error, just as any other program. However, most previous verification condition generators were not themselves proven correct. This meant that the proofs of programs produced using such VCGs were cast into doubt, depending upon the correctness of the VCG tool which was unproven.

The Sunrise package provides VCG tools which have been rigorously proven completely sound. Not only are the tools sound, but the rigor of the soundness proof itself flows from the fact that the proof was constructed and mechanically checked within HOL. This establishes the trustworthiness of these VCG tools as a firm foundation for proving individual programs correct.

The theory on which the Sunrise system is built is Floyd/Hoare-style axiomatic semantics. This has been commonly taught for decades at the graduate and undergraduate levels. The Sunrise set of tools should be broadly familiar in style to both the teachers and students of such courses. Thus, these tools may serve the purposes of instructors wishing to demonstrate this established theory in the classroom.

1.1 Recommended Reading

This document does expect that the reader is familiar with the above theory, and with the HOL theorem prover. These prerequisites may be addressed by the following references.

In particular, the book by Francez is very well written for beginners, succinctly presenting the material in an understandable way. It is especially valuable for it's good definitions of the terminology of this field.

- Axiomatic Semantics and Program Proofs:
  - Nissim Francez, *Program Verification*, Addison Wesley, 1992.
  - Krzysztof R. Apt and Ernst-Rüdiger Olderog, *Verification of Sequential and Concurrent Programs*, Second Edition, Springer-Verlag, 1997.
  - David Gries, *The Science of Programming*, Springer-Verlag, 1981.
  - Suad Alagić and Michael A. Arbib, *The Design of Well-Structured and Correct Programs*, Springer-Verlag, 1978.
  - M. J. C. Gordon, *Programming Language Theory and its Implementation*, Prentice-Hall, 1988.
  - Glynn Winskel, *The Formal Semantics of Programming Languages: An Introduction*, MIT Press, 1993.
- The Higher Order Logic theorem prover:
  - M. J. C. Gordon and T. F. Melham, *Introduction to HOL: A theorem proving environment for higher order logic*, Cambridge University Press, 1993

In addition, the reader may wish to read some of the papers published about this work. The most complete and extensive description is contained within the author's dissertation.

- Papers:
  - Peter V. Homeier and David F. Martin, "Trustworthy Tools for Trustworthy Programs: A Verified Verification Condition Generator," in *Proceedings of the 7th International Workshop on Higher Order Logic Theorem Proving and its Applications*, eds. Thomas Melham and Juanito Camilleri, Valletta, Malta, September 19-22, 1994, Lecture Notes in Computer Science Vol 859, Springer-Verlag, pages 269-284.
    (also reprinted as) Peter V. Homeier and David F. Martin, "A Mechanically Verified Verification Condition Generator," *The Computer Journal*, Vol. 38, No. 2, July 1995, pages 131-141.
  - Peter V. Homeier and David F. Martin, "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, Rutgers University, New Brunswick, NJ, USA, July 30 - August 3, 1996, Lecture Notes in Artificial Intelligence Vol 1104, Springer-Verlag, pages 201-215.

  - Peter V. Homeier and David F. Martin, "Mechanical Verification of Total Correctness Through Diversion Verification Conditions," in *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics* (TPHOLs'98), eds. J. Grundy and M. Newey, The Australian National University (ANU), Canberra, Australia, September 28 - October 1, 1998. Lecture Notes in Computer Science Vol 1479, Springer-Verlag, pages 189-206.

- Dissertation:

  - Peter V. Homeier, *Trustworthy Tools for Trustworthy Programs: A Mechanically Verified Verification Condition Generator for the Total Correctness of Procedures* (Ph.D. Dissertation), Department of Computer Science, University of California, Los Angeles, June 1995.

The above papers and dissertation may be obtained from the following web pages:

```
http://www.cis.upenn.edu/~homeier/publications/
http://www.cis.upenn.edu/~homeier/phd.html
```

Chapter 2

Preliminaries

2.1 Getting and Building Sunrise

The Sunrise system may be retrieved from the following web page:

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

This web page provides links to versions of Sunrise for several versions of HOL. Sunrise version 7.3 works with HOL4 versions Kananaskis-1 through Kananaskis-3. Sunrise versions 7.1 and 7.2 work with HOL4 version Kananaskis-2. Sunrise version 7.0.1 works with HOL98 versions Taupo-5 and Taupo-6.

2.1.1 Sunrise version 7.3

This most current version of Sunrise, as of February 2005, works with any of the HOL4 versions Kananaskis-1 through Kananaskis-3. It is downloaded as the file `vcg-73.tar.gz`. Move this to the directory `{HOLDIR}/examples`, where `{HOLDIR}` stands for the root directory of the HOL4 system. Then uncompress the archive `vcg-73.tar.gz` with `gunzip` and unpack it with `tar`:

```
mv vcg-73.tar.gz {HOLDIR}/examples
cd {HOLDIR}/examples
gunzip vcg-73.tar.gz
tar xvf vcg-73.tar
```

This will create a new subdirectory called `sunrise` in the `{HOLDIR}/examples` directory. Then to build the Sunrise system, type

```
cd sunrise/src
Holmake
```

2.1.2 Sunrise versions 7.1 and 7.2

Version 7.1 of Sunrise was released in May 2002. Version 7.2 of Sunrise was released in January 2004. The only difference is that 7.2 evades a clock rollover bug in Moscow ML. Both work with HOL4 version Kananaskis-2, which also evades the bug. They are downloaded as either the file `vcg-71.tar.gz` or `vcg-72.tar.gz`. Move this file to the directory `{HOLDIR}/examples`, where `{HOLDIR}` stands for the root directory of the HOL4 system. Then uncompress the archive with `gunzip` and unpack it with `tar`:

```
mv vcg-72.tar.gz {HOLDIR}/examples
cd {HOLDIR}/examples
gunzip vcg-72.tar.gz
tar xvf vcg-72.tar
```

This will create a new subdirectory called `sunrise` in the `{HOLDIR}/examples` directory. Then to build the Sunrise system, type

```
cd sunrise/src
Holmake
```

2.1.3 Sunrise version 7.0.1

This version of Sunrise works with HOL98 versions Taupo-5 and Taupo-6. It is downloaded as the file `vcg-701.tar.gz`. Move this to the directory `{HOLDIR}/src`, where `{HOLDIR}` stands for the root directory of the HOL98 system. Then uncompress `vcg-701.tar.gz` with `gunzip` and unpack it with `tar`:

```
mv vcg-701.tar.gz {HOLDIR}/src
cd {HOLDIR}/src
gunzip vcg-701.tar.gz
tar xvf vcg-701.tar
```

This will create a new subdirectory called `vcg` in the `{HOLDIR}/src` directory. Then to build the Sunrise system, type

```
cd vcg/src
mosmlyac Parser.grm
mosmllex Lexer.lex
Holmake
```

2.2 Loading Sunrise

In order to load the Sunrise system, we first need to set the load path to include the Sunrise library.

For Sunrise versions 7.1, 7.2, or 7.3, start HOL4 and type

```
loadPath := HOLDIR ^ "/examples/sunrise/src" :: !loadPath;
```

or for Sunrise version 7.0.1, type

```
loadPath := HOLDIR ^ "/src/vcg/src" :: !loadPath;
```

Then, for all versions, type the following lines into an HOL98 session:

```
load "vcg_parser";
open vcg_parser;

load "vcg_prettyp";

load "vcg_tactics";
open vcg_tactics;
```

If you also wish to use the fast but insecure VCG tactics, then type the next lines:

```
load "fvcg_tactics";
open fvcg_tactics;
```

All of these (and some additional tools) may be loaded by reading a single SML file by the command

```
use (HOLDIR ^ "/examples/sunrise/examples/load_vcg.sml");
```

Chapter 3

Sunrise Syntax

The Sunrise system presents two new languages, the Sunrise programming language and an associated assertion language. The programming language is for writing programs, and is essentially a small subset of Pascal. The assertion language is for writing expressions that describe relationships between variables in the state at a particular moment in time, or to describe relationships between the values of variables at two different times.

The identifiers, numbers, and symbols of these languages are typical of imperative languages such as Pascal or C.

Whitespace, which is spaces, tabs, newlines, and carriage return characters, is ignored, except to delimit lexical tokens.

Comments are bracketed by `(*` and `*)`, and may be nested.

Variables consist of a string of characters, where each character is either a letter (upper or lower case), a digit, an underscore (`_`), or a quote character (`'`). The variable must begin with an letter or a quote character. It may not be any of the reserved keywords, which for the Sunrise language are:

```
abort       global      suc
assert      if          then
calls       od          true
do          post        val
else        pre         var
empty       procedure   while
end         program     with
false       recurses
fi          skip
```

Variables which begin with a quote character are called *logical variables*, otherwise they are called *program variables*. Only program variables may be used in Sunrise program code; both program and logical variables may be used in the assertion language. Thus, the variable `'n` may appear in the invariant for a **while** command, but not in the test or the body. Logical variables may never be accessed for reading or writing by the program code. Instead, they are used to mark and

hold the values of the corresponding program variables (without the initial quote character) at some strategic moment and place. The fact that they cannot be written to by the program ensures that they never change from those initial values.

Numbers are strings of digits, without any $+$ or $-$ in front, and without any decimal points or exponents. Only non-negative integers may be represented, but there is no limit on their size.

Here is the BNF grammar for the Sunrise programming language:

`exp` :	e	$::=$	$n \mid x \mid \ ++x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid (e)$
`list` :	es	$::=$	$\langle\,\rangle \mid \langle e_1,\ \ldots,\ e_m\rangle$
`bexp` :	b	$::=$	$e_1 = e_2 \mid e_1 < e_2 \mid es_1 \ll es_2 \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \sim b \mid (b)$
`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** $\mid$ $p(x_1,\ \ldots,\ x_n\ ;\ e_1,\ \ldots,\ e_m)$
`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}; **post** a_{post}; **calls** p_1 **with** a_1; $\vdots$ **calls** p_j **with** a_j; **recurses with** a_{rec}; c **end procedure** $\mid$ $d_1\ ;\ d_2$ $\mid$ **empty**
`prog` :	π	$::=$	**program** $d\ ;\ c$ **end program**

Most of these language constructs should be familiar, but a few notes of explanation are in order. Currently there are no arrays; hopefully these will be included later.

In the grammar, variables are denoted by x, y, or z, or their variants, e.g. x_1 or z_k.

Numbers are denoted by n.

$++$ is the increment operator, with a side effect as in C. Subtraction terminates at zero, so $e_1 - e_2 = 0$ if $e_2 > e_1$. For lists, $es_1 \ll es_2$ is the lexicographic ordering.

abort causes an immediate abnormal termination. The **while** loop requires an invariant assertion a and a progress assertion a_{pr}, where a_{pr} is either **false** (the default if omitted) or of the form $(v < x)$, where x is a logical variable denoting the value of v for the prior loop iteration. **false** indicates no progress, which is not useful if total correctness is the goal.

In the procedure call $p(x_1, \ldots, x_n ; e_1, \ldots, e_m)$, p is an identifier, the x_i are program variables denoting the actual variable parameters (passed by call-by-name), and the e_i are numeric expressions denoting the actual value parameters (passed by call-by-value).

A 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 $z_1, \ldots, z_k$, precondition a_{pre}, postcondition a_{post}, entrance progress expressions a_1 for procedure p_1 through a_j for p_j, recursive progress expression a_{rec}, and body c. The global variables list must include all global variables used in p or in any procedure p calls directly or indirectly. All parameter and variable types are `num`. The *entrance* of a procedure is just before its body, but within its scope. The precondition denotes the state when control reaches the entrance of the procedure. It should contain only program variables. The postcondition denotes the state at the exit of the procedure. It may contain both logical and program variables, where logical variables denote the values of variables at the entrance of the procedure, and program variables denote the values at the exit.

The entrance procedure expression for an associated procedure describes calls from the body of the current procedure to the associated procedure. It denotes the state at the entrance of the associated procedure, just after a one-deep call from the current procedure. Logical variables denote the values of variables at the entrance of the calling procedure, and program variables denote the values of variables at the entrance of the called procedure.

The recursive progress expression describes the progress achieved between recursive calls of the same procedure. The expression must either be **false**, (the default,) which means that the procedure is *not* recursive, or it must be of the form $v < y$, where y is a logical variable and v is a numeric expression of the assertion language containing program variables. The logical variable y denotes the value of the expression v at the entrance of the procedure for the earlier call, and the recursive progress expression states that the value of v must have strictly decreased at the entrance of the deeper, nested call. The use of the progress annotations will be clarified through examples.

The **empty** declaration defines no new names. In a sequence of declarations, if the same procedure name is used twice, the latter declaration is used.

A program contains a declaration and a command, which is the main body. The declaration may contain multiple declarations of procedures, or it may be omitted.

Programs written in the above Sunrise programming language could be executed without any recourse to the assertion language. However, for proving programs correct, the assertions provide necessary information about the intended behavior of the program.

Here is the BNF grammar for the Sunrise assertion language:

`vexp` :	v	::=	$n \mid x \mid v_1 + v_2 \mid v_1 - v_2 \mid v_1 * v_2$
		|	$(a \ => \ v_1 \mid v_2) \mid f(v_1, \ \ldots, \ v_k) \mid (v)$
`list` :	vs	::=	$\langle\,\rangle \mid \langle v_1, \ \ldots, \ v_k \rangle$
`aexp` :	a	::=	$\mathbf{true} \mid \mathbf{false}$
		|	$v_1 = v_2 \mid v_1 < v_2 \mid vs_1 \ll vs_2$
		|	$a_1 \wedge a_2 \mid a_1 \vee a_2 \mid \sim a$
		|	$a_1 \Rightarrow a_2 \mid a_1 = a_2 \mid (a_1 \ => \ a_2 \mid a_3)$
		|	$\forall x.a \mid \exists x.a$

The meaning of most of these should be evident to the reader. Subtraction is terminated by zero, so that $v_1 - v_2 = 0$ if $v_2 > v_1$. $(a \ => \ v_1 \mid v_2)$ is a conditional expression, yielding the value of v_1 if a is true, otherwise yielding the value of v_2.

$vs_1 \ll vs_2$ is the lexicographic ordering between two lists. $a_1 \Rightarrow a_2$ is implication. $a_1 = a_2$ is "if and only if." $(a_1 \ => \ a_2 \mid a_3)$ is a conditional expression, yielding the value of a_2 if a_1 is true, otherwise yielding the value of a_3. $\forall x.a$ and $\exists x.a$ are the normal universal and existential quantifications. These make the assertion language of Sunrise into a first-order logic. Quantification can only take place over non-negative integers.

$f(v_1, \ \ldots, \ v_k)$ is a call to an "uninterpreted function." f should be the name of a function defined in the HOL logic. It may be either a standard function or one defined by the user, but it must be defined before it is used in this way. The type of the HOL function should be `:(num)list->num`. For example, the factorial function is predefined in HOL as `FACT`, with type `:num->num`. This cannot be used directly, but another function may be defined by the user, as

```
val fact = new_definition
 ("fact",
  (--‘fact ns  =  FACT (HD ns)‘--));
```

Then a Sunrise procedure to compute factorial could refer to `fact`, as in the postcondition below:

```
procedure factorial (var f; val n);
   pre  true;
   post f = fact(’n);
   calls factorial with n < ’n;
   recurses with n < ’n;
```

```
   if n = 0
   then f := 1
   else
      factorial(f; n-1);
      f := n * f
   fi
end procedure;
```

Both the grammar tables above use the "printed" version of the operators; there is sometimes a different way that the operators are actually typed in to HOL, as illustrated by examples in the next table.

Printed	**Typed**
$++x$	`++x`
$\langle\,\rangle$	`< >`
$\langle v_1,\ v_2,\ v_3\rangle$	`<v1, v2, v3>`
$es_1 \ll es_2$	`es1 << es2`
$b_1 \wedge b_2$	`b1 /\ b2`
$b_1 \vee b_2$	`b1 \/ b2`
skip	`skip`
$(a\ =>\ v_1 \mid v_2)$	`(a => v1 | v2)`
$\forall x.a$	`!x.a`
$\exists x.a$	`?x.a`

Chapter 4

Well-Formedness

Many programs which may be syntactically acceptable, which fit the BNF, are still meaningless, and cannot be considered as real programs. For example, suppose a procedure has three formal parameters, but they all have the same name; such a program simply does not make sense.

A more subtle problem occurs when, for example, a procedure has a variable parameter and also accesses a global variable; if it is ever called with that global variable *as* the actual parameter, then the body of that procedure will have two different names that both access the same real variable. This situation is called *aliasing*. While this program could run and generate some result, the behavior might be disjointedly different from normal, without aliasing. For the sake of simplicity and regularity, we wish to avoid aliasing.

We describe this sort of issue as *well-formedness*. The well-formedness of every program is a well-defined test, which in fact is not complex but can be quickly decided for any program which is syntactically correct. Only well-formed programs are suitable for analysis by the verification condition generator. It is only for such programs that the verification condition generator has been verified.

The following descriptions are terse, but should indicate the sources of non-well-formedness to avoid.

All expressions in the programming language, numeric (`:exp`), numeric lists (`:(exp)list`), and boolean (`:bexp`), must not contain any logical variables.

Other than **while** commands and procedure calls, commands (`:cmd`) are well-formed if their components are well-formed, whether expressions or other commands.

A **while** command (**assert** a **with** a_{pr} **while** b **do** c **od**) is well-formed if the boolean test b and the body c are well-formed, and if the progress condition a_{pr} is either **false** or of the form $v < x$, where v is a numeric assertion-language expression and x is a logical variable, and where x is not in either v or the invariant a. In addition, if the **while** command occurs within the body of a procedure, x must not occur in any of the "**calls** ... **with**" entrance progress conditions of the surrounding procedure.

For a procedure call ($p(xs; es)$), where xs is the list of actual variable parameters, and es is the list of actual value parameters, to be well formed, all the variables and expressions in xs and es must be well-formed. In addition, the number of actual variable and value parameters must

match the number of corresponding formal parameters in the declaration of procedure p. Finally, there must be no duplicate variables among the combination of xs and the declared globals of p. This last check ensures that there is no aliasing.

For a single procedure declaration (as opposed to a call), let x denote the combination of all formal parameters and declared global variables. Then let $\acute{x}$ denote the logical variables corresponding to x, formed by adding quotes to their beginnings. Then, for the declaration to be well-formed, all of the following must hold:

1. All the variables in x must be program variables, not logical.
2. There must be no duplicates in x.
3. The body of the procedure must be well-formed (as above).
4. For all procedures called from within the body, their declared globals must be contained within this procedure's declared globals.
5. The free variables of the body must be contained in x.
6. The free variables of the preconditon must be contained in x.
7. The free variables of the postcondition must be contained in the combination of x and $\acute{x}$.
8. For every entrance progress expression **calls** p_i **with** a_i, the free variables of a_i must be contained within the combination of:
   - $\acute{x}$
   - the formal parameters of p_i
   - the declared globals of p_i
9. The recursion expression a_{rec} must be either **false** (the default), which indicates that this procedure is *not* recursive, or else a_{rec} must be of the form $v < y$, where y is a logical variable, and v is a numeric expression in the assertion language, whose free variables must be contained within x.

A compound procedure declaration is well-formed if all of the single procedure declarations are well-formed, as above.

A program is well-formed if its declaration is well-formed and its main body is well-formed.

Chapter 5

Parsing and Prettyprinting

We can create Sunrise code by using the ||‘ and ‘|| brackets:

```
- ||‘ a+b*45 ‘||;
> val it = ‘‘a + b * 45‘‘ : term
- ||‘ if 0<1 then skip else abort fi ‘||;
> val it = ‘‘if 0 < 1 then skip else abort fi‘‘ : term
```

Notice how each expression is immediately pretty-printed back similar to the original input. Despite the closeness of the input and output, in fact these expressions are being parsed into HOL terms which would look strange if observed directly, so a prettyprinter is installed which makes their display far more friendly.

The phrase between the double-bar brackets may be any of the expressions of the programming language, or a command, a declaration, or an entire program.

We can write assertions in the Sunrise assertion language by using curly braces ({}) within the double-bar brackets:

```
- ||‘ {a+b*45} ‘||;
> val it = ‘‘a + b * 45‘‘ : term
- ||‘ {!m. ?n. m+m<n} ‘||;
> val it = ‘‘!m. (?n. m + m < n)‘‘ : term
- ||‘ {!m. !n. !p. m<n /\ n<p ==> m<p} ‘||;
> val it = ‘‘!m n p. m < n /\ n < p ==> m < p‘‘ : term
```

The pretty-printing of consecutively quantified variables is condensed as shown above. Similarly, the version 7.1 or later Sunrise parser can accept multiple bound variables at a time for the quantifiers. Here is an sample:

```
- ||‘ {!m n p. m<n /\ n<p ==> m<p} ‘||;
> val it = ‘‘!m n p. m < n /\ n < p ==> m < p‘‘ : term
```

Thus the condensed form may be used for input as well as output.

Unfortunately, though versions of Sunrise before 7.1 do handle the condensed pretty-printing, those versions cannot parse multiple bound variable in one quantifier, and the last term would raise an exception.

Here are two examples of parsing and pretty-printing full programs:

```
- ||‘ program
        x := 3;
        y := x * x;
        z := ++x * y
     end program ‘||;
> val it = ‘‘program
               empty;
               x := 3;
               y := x * x;
               z := ++x * y
            end program‘‘ :
  term

- ||‘ program
        procedure quotient_remainder (var q,r; val x,y);
           pre  0 < y;
           post ’x = q * ’y + r /\ r < ’y;

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

        quotient_remainder(q,r;7,3)
     end program
  ‘||;
> val it =
    ‘‘program
        procedure quotient_remainder(var q,r;val x,y);
           pre 0 < y;
           post ’x = q * ’y + r /\ r < ’y;

           r := x;
```

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

         quotient_remainder(q,r;7,3)
      end program'' : term
```

You can also enter program correctness statements, also known as "Hoare triples." There are several forms of these, but the four most important ones are the statements of the partial or total correctness of either a command with regards to its precondition and postcondition, or of a program with regards to its postcondition, for which the precondition is assumed to be **true**.

Here is a term which expresses the partial correctness of a command:

```
- ||' { 0 < y /\ 'x = x /\ 'y = y }
         r := x;
         q := 0;
         assert 'x = q * 'y + r /\ 0 < y /\ 'y = y
         while ~(r < y) do
            r := r - y;
            q := ++q
         od
      { 'x = q * 'y + r /\ r < 'y }
      /empty
  '||;
> val it =
    ''{0 < y /\ ('x = x /\ 'y = y)}
         r := x;
         q := 0;
         assert 'x = q * 'y + r /\ (0 < y /\ 'y = y)
         while ~(r < y) do r := r - y; q := ++q od
      {'x = q * 'y + r /\ r < 'y} /empty'' : term
```

In the above partial correctness statement, the precondition and postcondition are enclosed in curly braces, indicating partial correctness. The **empty** environment at the end (which is required) means that this is within a context where no procedures are defined. Note that the "with" clause of the while loop has been omitted, since it is not relevant to partial correctness.

Here is a term which expresses the total correctness of the same command:

```
- ||' [ 0 < y /\ 'x = x /\ 'y = y ]
         r := x;
         q := 0;
         assert 'x = q * 'y + r /\ 0 < y /\ 'y = y
```

```
          with r < 'r
        while ~(r < y) do
           r := r - y;
           q := ++q
        od
      [ 'x = q * 'y + r /\ r < 'y ]
      /empty
  ‘||;
> val it =
    ‘‘[0 < y /\ ('x = x /\ 'y = y)]
        r := x;
        q := 0;
        assert 'x = q * 'y + r /\ (0 < y /\ 'y = y) with r < 'r
        while ~(r < y) do r := r - y; q := ++q od
      ['x = q * 'y + r /\ r < 'y] /empty‘‘ : term
```

In the above total correctness statement, the precondition and postcondition are enclosed in square brackets, indicating total correctness, and the **empty** environment at the end (which is required) means that this is within a context where no procedures are defined.

Here is a term which expresses the partial correctness of a program:

```
- ||‘ program
        x := 3;
        y := x * x;
        z := ++x * y
      end program
      { z = y * x } ‘||;
> val it =
    ‘‘program
        empty;
        x := 3;
        y := x * x;
        z := ++x * y
      end program
      {z = y * x}‘‘ : term
```

As the program defined no procedures, it's declaration is **empty**. Next, here is a term which expresses the total correctness of the same program:

```
- ||‘ program
        x := 3;
        y := x * x;
        z := ++x * y
      end program
```

```
      [ z = y * x ] ‘||;
> val it =
    ‘‘program
        empty;
        x := 3;
        y := x * x;
        z := ++x * y
      end program
      [z = y * x]‘‘ : term
```

The above program has no procedure declarations, and a body of three assignment statements. The program correctness statement says that when the program is run, it will terminate in a state where $z = y * x$.

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

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

       p91(a; 77)

    end program
    [ a = 91 ]
  ‘||;
> val it =
    ‘‘program
        procedure p91(var x;val y);
           global ;
           pre true;
           post 100 < ’y => x = ’y - 10 | x = 91;
           calls p91 with 101 - y < 101 - ’y;
           recurses with 101 - y < ’z;

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

        p91(a;77)
      end program
      [a = 91]‘‘ : term
```

The above program contains a procedure which encodes McCarthy’s “91 function.” This function has a very simple behavior, as seen by the specification, which may be surprising given the code of its body.

The program correctness statement says that this program, which calls the 91 function with a value of 77, terminates and returns a value of 91.

Chapter 6

Establishing Program Correctness Goals

The program correctness statements given at the end of the last chapter are just the sort of thing we want to use Sunrise to prove, so the first step is to set this up as a goal for the interactive goal-stack interface.

However, there is a small problem. Goals are normally posited to the HOL system by use of the `g` command, as in

```
g ‘!x y. x < y ==> ~(y <= x)‘;
```

But the `g` command expects a quotation as its argument, and the Sunrise parser invoked by the double-bar brackets produces a term. A term and a quotation are different to HOL. To produce a quotation, we can use antiquotation to insert a term inside of a quotation, as follows:

```
g ‘^(||‘ program
            x := 3;
            y := x * x;
            z := ++x * y
         end program
         [ z = y * x ]
    ‘||)‘;
```

While perfectly usable, this begins to look messy. The goal can be posited without antiquotation by the `set_goal` command, as follows:

```
set_goal([],
     ||‘ program
            x := 3;
            y := x * x;
            z := ++x * y
         end program
         [ z = y * x ]
    ‘||);
```

Both methods yield the same result.

One can also posit goals which state the total correctness of a command (as opposed to a program), along with its precondition and postcondition. Here is an example.

```
g `^(||` [ 0 < y /\ 'x = x /\ 'y = y ]
           r := x;
           q := 0;
           assert 'x = q * 'y + r /\ 0 < y /\ 'y = y
             with r < 'r
           while ~(r < y) do
              r := r - y;
              q := ++q
           od
         [ 'x = q * 'y + r /\ r < 'y ]
         /empty
   `||)`;
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         [0 < y /\ ('x = x /\ 'y = y)]
            r := x;
            q := 0;
            assert 'x = q * 'y + r /\ (0 < y /\ 'y = y) with r < 'r
            while ~(r < y) do r := r - y; q := ++q od
         ['x = q * 'y + r /\ r < 'y] /empty


     : proofs
```

The square brackets signify total correctness, including termination, not only partial correctness. The `/empty` at the end is requried, and indicates that this command is to be considered in a context where there are no declared procedures.

In addition, the user can posit goals which are the partial correctness versions of the above goals, by simply replacing the square brackets (`[]`) with curly braces ({}).

Thus the Sunrise system can be used to prove either partial or total correctness, and either of commands or of entire programs.

Chapter 7

Verification Condition Generator Tactics

There are twelve tactics in the Sunrise system for analyzing program correctness goals and reducing them to subgoals, where the subgoals correspond to the verification conditions generated by the verified Sunrise VCG. These tactics are differentiated by whether they analyze either commands or complete programs, by whether they attempt to prove partial or total correctness, and by whether they calculate the verification conditions by secure proof, by fast ML code, or both. These differences specify the $2 \times 2 \times 3 = 12$ tactics.

VCG TACTICS	Partial Correctness	Total Correctness
Command	VCGCP_TAC FAST_VCGCP_TAC TEST_VCGCP_TAC	VCGC_TAC FAST_VCGC_TAC TEST_VCGC_TAC
Program	VCGP_TAC FAST_VCGP_TAC TEST_VCGP_TAC	VCG_TAC FAST_VCG_TAC TEST_VCG_TAC

Essentially they all do the same thing, but they do it different ways. They all take the goals of partial or total correctness of particular Sunrise programs with respect to their specifications (expressed as postconditions), suitably annotated, and convert these correctness goals into a set of subgoals, where each subgoal corresponds to one of the verification conditions generated by the VCG analysis. This has the same structure as any tactic, which in general reduces any goal to be solved to a set of sufficient subgoals.

These tactics have the benefit of converting a goal concerning programs, specifications, and correctness into a set of subgoals which are free of any mention of program code or correctness specifications. Rather, the subgoals are in the logic of "pure" HOL, without any Sunrise constructs, and can be then proven by the user using all of the normal HOL tools.

The partial correctness VCG tactics are not available in versions of Sunrise before version 7.1, that is, not in version 7.0.1.

7.1 VCGCP_TAC

The VCGCP_TAC tactic takes a goal which is the partial correctness of a command, and reduces the goal to its VCG subgoals in a process which is completely secure, without any chance of unsoundness in the process. All computations are performed solely by secure proof within the HOL theorem prover, so the results are guarranteed to be logically consistent with the original goal. This means that if all the subgoals are proven by the user, then the HOL subgoal package will return a fully verified HOL theorem stating the partial correctness goal which was originally posited.

Once the goal is entered into HOL and presented to the user as the top goal on the stack, it can then be analyzed by the Sunrise VCG by the tactic VCGCP_TAC. By default, this tactic prints out a trace as it works of the rules of the Floyd/Hoare-style axiomatic semantics that it is using as it analyzes the program and specification.

Overall, there are three phases of operation of VCGCP_TAC:

1. Well-formedness analysis of the command
   - Syntax of command is checked to make sure well-formedness conditions are not violated— any violation here causes the tactic to fail
2. Syntactic analysis of the command
   - VCs are generated to confirm the partial correctness of the command, given the properties from the previous phase
3. Translation of VCs to HOL logic
   - VCs generated previously are now translated into equivalent statements in the HOL logic, according to the semantics of the assertion language

The middle phase generates verification conditions, and the trace describes this work in detail. If desired, the trace can be omitted by typing

```
print_vcg := false;
```

However, in order to clarify the work of the VCG, we will assume that tracing is left on. How this trace is calculated is described in detail in chapter 8.

7.1.1 Well-formedness Analysis of Commands

The first step of analysis checks to see that the command is well-formed. This includes a variety of simple checks, for example that no logical variables are used in program code. These checks are straightforward and simple, and this analysis takes only a fraction of the total time. If everything is fine, analysis proceeds silently to the next phase. If any problems are found, the VCGCP_TAC tactic fails here.

Sunrise version 7.1 for the first time provides error messages to aid the user in identifying and isolating the source of a well-formedness error. In prior versions these were not produced. The

error messages should describe a single well-formedness error, and perhaps give the command or other identifying text where it occurred. Chapter 4 on well-formedness gives the definitions of what is a well-formed command or program.

7.1.2 Syntactic Analysis of Program Code

The VCG analyzes the command according to its abstract syntax tree, generating verification conditions to confirm that it meets the partial correctness specified by its precondition and postcondition, and all internal annotations, such as for while loops. The VCG efficiently generates all needed verification conditions for all these claims in a single pass over the syntax of the body.

The VCG recursively descends the tree of the body's syntax, pushing the postcondition backwards through the commands, generating necessary preconditions for each command. As commands are composed recursively of commands, the results for constituent commands are used in the VCG's computations for aggregate commands. At the end of processing the entire original command, a precondition is produced which is reconciled with the specified precondition by means of one additional verification condition.

Here is an example of this analysis for a command which computes the quotient and remainder of integral division. All phases of the analysis are shown.

```
- g `^(||` { 0 < y /\ 'x = x /\ 'y = y }
               r := x;
               q := 0;
               assert 'x = q * 'y + r /\ 0 < y /\ 'y = y
               while ~(r < y) do
                  r := r - y;
                  q := ++q
               od
         { 'x = q * 'y + r /\ r < 'y } /empty
  `||)`;
```

The HOL system repeats this back to us, and we then issue the VCGCP_TAC tactic. As the tactic runs, it produces a trace of the Floyd/Hoare triples proven. These are printed as

$$\{p\}c\{q\}/r$$

where p is the precondition, c is the command, q is the postcondition, and r is the environment of procedures. Since this environment r does not change and is also bulky, it is abbreviated in the trace just by the variable r, but the other arguments are pretty-printed in full.

```
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         {0 < y /\ ('x = x /\ 'y = y)}
```

```
            r := x;
            q := 0;
            assert 'x = q * 'y + r /\ (0 < y /\ 'y = y)
            while ~(r < y) do r := r - y; q := ++q od
          {'x = q * 'y + r /\ r < 'y} /empty


     : proofs
- et(VCGCP_TAC);
By the ASSIGN rule, we have
{'x = (q + 1) * 'y + r /\ (0 < y /\ 'y = y)}
   q := ++q
{'x = q * 'y + r /\ (0 < y /\ 'y = y)} /r

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

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

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

By the ASSIGN rule, we have
{'x = 0 * 'y + r /\ (0 < y /\ 'y = y)}
   q := 0
{'x = q * 'y + r /\ (0 < y /\ 'y = y)} /r

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

```
 'x = (q + 1) * 'y + (r - y) /\ (0 < y /\ 'y = y),
 ('x = q * 'y + r /\ (0 < y /\ 'y = y)) /\ ~(~(r < y)) ==>
 'x = q * 'y + r /\ r < 'y]

By the ASSIGN rule, we have
{'x = 0 * 'y + x /\ (0 < y /\ 'y = y)}
   r := x
{'x = 0 * 'y + r /\ (0 < y /\ 'y = y)} /r

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

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


CPU: usr: 1.760 s    sys: 0.000 s    gc: 0.320 s    theorems: 15155
> val it =
    !'x q 'y r y.
      (('x = q * 'y + r) /\ 0 < y /\ ('y = y)) /\ r < y ==>
      ('x = q * 'y + r) /\ r < 'y


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


    !y 'x x 'y.
      0 < y /\ ('x = x) /\ ('y = y) ==>
      ('x = 0 * 'y + x) /\ 0 < y /\ ('y = y)
```

```
    : goalstack
```

After proving these three subgoals by normal means, we see HOL print:

```
> val it =
    Initial goal proved.
    |- {0 < y /\ (’x = x /\ ’y = y)}
          r := x;
          q := 0;
          assert ’x = q * ’y + r /\ (0 < y /\ ’y = y)
          while ~(r < y) do r := r - y; q := ++q od
       {’x = q * ’y + r /\ r < ’y} /empty : goalstack
```

7.1.3 Translation into HOL logic

The last stage of the above calculation is where each verification condition is translated into the HOL logic according to the semantics of the assertion language, which is deeply embedded within the HOL logic. Thus every assertion has a corresponding term in the HOL logic, whose truth is equivalent to that of the assertion.

The results are seen in the example above, after the line with the CPU timings and theorem count.

For the most part, care is taken to translate variables from the assertion language to a similarly-named variable in the HOL logic, so that there is continuity between the two languages.

This process is completely automatic and cannot fail.

7.2 VCGC_TAC

The VCGC_TAC tactic is similar to VCGCP_TAC, as just discussed, but analyzes the command for total correctness, including termination, instead of only partial correctness. This tactic takes a goal which states the total correctness of a command, and reduces the goal to its VCG subgoals in a process which is completely secure, without any chance of unsoundness in the process.

The verification conditions generated include logic to ensure the termination of any while loops in the command, according to the progress annotations "$v < x$" in the while commands. These should specify expressions v which will strictly decrease by at least one for each iteration of the loop. The verification conditions generated require this decrease to be proven.

If all the subgoals are proven by the user, then the HOL subgoal package will return a fully verified HOL theorem stating the total correctness goal which was originally posited.

Overall, there are the same three phases of operation of VCGC_TAC as there were for VCGCP_TAC:

1. Well-formedness analysis of the command
   - Syntax of command is checked to make sure well-formedness conditions are not violated— any violation here causes the tactic to fail
2. Syntactic analysis of the command
   - VCs are generated to confirm the total correctness of the command, given the properties from the previous phase
3. Translation of VCs to HOL logic
   - VCs generated previously are now translated into equivalent statements in the HOL logic, according to the semantics of the assertion language

The middle phase generates verification conditions, and the trace describes this work in detail.

7.2.1 Syntactic Analysis of Program Code

Here is an example of this analysis for a command which computes the quotient and remainder of integral division. All phases of the analysis are shown.

```
- g `^(||` [ 0 < y /\ 'x = x /\ 'y = y ]
              r := x;
              q := 0;
              assert 'x = q * 'y + r /\ 0 < y /\ 'y = y
                with r < 'r
              while ~(r < y) do
                 r := r - y;
                 q := ++q
              od
```

```
        [ 'x = q * 'y + r /\ r < 'y ] /empty
  `||)`;
```

Note the inclusion of the "with" clause, stating that the expression r should strictly decrease on each iteration. The HOL system repeats this back to us, and we then issue the VCGC_TAC tactic. As the tactic runs, it produces a trace of the Floyd/Hoare triples proven. As before, these are printed as $\{p\}c\{q\}/r$.

```
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         [0 < y /\ ('x = x /\ 'y = y)]
            r := x;
            q := 0;
            assert 'x = q * 'y + r /\ (0 < y /\ 'y = y) with r < 'r
            while ~(r < y) do r := r - y; q := ++q od
         ['x = q * 'y + r /\ r < 'y] /empty


     : proofs
- et(VCGC_TAC);
By the ASSIGN rule, we have
{('x = (q + 1) * 'y + r /\ (0 < y /\ 'y = y)) /\ r < 'r}
   q := ++q
{('x = q * 'y + r /\ (0 < y /\ 'y = y)) /\ r < 'r} /r

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

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

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

```
By the ASSIGN rule, we have
{'x = 0 * 'y + r /\ (0 < y /\ 'y = y)}
   q := 0
{'x = q * 'y + r /\ (0 < y /\ 'y = y)} /r

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

By the ASSIGN rule, we have
{'x = 0 * 'y + x /\ (0 < y /\ 'y = y)}
   r := x
{'x = 0 * 'y + r /\ (0 < y /\ 'y = y)} /r

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

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


CPU: usr: 2.380 s    sys: 0.000 s    gc: 0.390 s    theorems: 27817
> val it =
```

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


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


!y 'x x 'y.
  0 < y /\ ('x = x) /\ ('y = y) ==>
  ('x = 0 * 'y + x) /\ 0 < y /\ ('y = y)

 : goalstack
```

Comparing these with the subgoals generated to prove partial correctness by the VCGCP_TAC tactic, we observe that the middle subgoal is different, with new conditions attached both to its antecedent ($r = 'r$) and its consequent ($r - y < 'r$).

After proving these three subgoals by normal means, we see HOL print:

```
> val it =
    Initial goal proved.
    |- [0 < y /\ ('x = x /\ 'y = y)]
          r := x;
          q := 0;
          assert 'x = q * 'y + r /\ (0 < y /\ 'y = y) with r < 'r
          while ~(r < y) do r := r - y; q := ++q od
       ['x = q * 'y + r /\ r < 'y] /empty : goalstack
```

The square brackets indicate that this is a total correctness statement, as compared with the earlier partial correctness proven by VCGCP_TAC.

7.3 VCGP_TAC

The VCGP_TAC tactic reduces a goal of the partial correctness of a given program to its VCG subgoals, in a process which is completely secure, without any chance of unsoundness in the process.

Once the goal is entered into HOL and presented to the user as the top goal on the stack, it can then be analyzed by the Sunrise VCG by the tactic VCGP_TAC. By default, this tactic prints out a trace as it works of the rules of the Floyd/Hoare-style axiomatic semantics that it is using as it analyzes the program and specification.

Overall, there are four phases of operation of VCGP_TAC:

1. Well-formedness analysis of entire program
   - Syntax of entire program checked to make sure well-formedness conditions are not violated—
     any violation here causes the tactic to fail
2. Syntactic analysis of procedure declarations
   - VCs generated to confirm the partial correctness and entrance progress conditions of each procedure in turn
3. Syntactic analysis of main program commands
   - VCs generated to confirm the partial correctness of the main body of the program, given the properties from the previous phases
4. Translation of VCs to HOL logic
   - VCs generated previously are now translated into equivalent statements in the HOL logic, according to the semantics of the assertion language

Each of the middle two phases generate verification conditions, and the trace describes this work in detail.

7.3.1 Well-formedness Analysis of the Program

The first step of analysis checks to see that the program is well-formed. This includes a variety of simple checks, for example that no logical variables are used in program code, or in the preconditions for procedures. Also, it checks the lists of global variable declarations for completeness, and makes sure that there are no duplicates among the formal variables and the global variables accessable for each procedure. Significantly, this phase also checks that no aliasing of variables happens in any procedure call. These checks are straightforward and simple, and this analysis takes only a fraction of the total time. If everything is fine, analysis proceeds silently to the next phase. If any problems are found, the VCGP_TAC tactic fails here.

Sunrise version 7.1 for the first time provides error messages to aid the user in identifying and isolating the source of a well-formedness error. In prior versions these were not produced. The

error messages should describe a single well-formedness error, and perhaps give the command or other identifying text where it occurred. Chapter 4 on well-formedness gives the definitions of what is a well-formed command or program.

7.3.2 Syntactic Analysis of Program Code

For each procedure, the VCG analyzes the body of the procedure according to its abstract syntax tree, generating verification conditions to confirm that it meets the partial correctness specified by its declared precondition and postcondition. These verification conditions are constructed in such a way that they also confirm the entrance progress conditions specified in the procedure's header. The VCG efficiently generates all needed verification conditions for all these claims in a single pass over the syntax of the body.

The VCG recursively descends the tree of the body's syntax, pushing the postcondition backwards through the commands, generating necessary preconditions for each command. As commands are composed recursively of commands, the results for constituent commands are used in the VCG's computations for aggregate commands. At the end of processing the entire body, a precondition is produced which is reconciled with the declared precondition by means of one additional verification condition.

The same kind of syntactic analysis is performed later on the main body of the program, except that the precondition to reconcile with is simply the assertion **true**.

Here is an example of this analysis for a program with only one procedure, which calculates McCarthy's "91 function,"

$$f_{91}(y) = \textbf{if } 100 < y \textbf{ then } y - 10 \textbf{ else } f_{91}(f_{91}(y + 11))$$

which has the interesting and non-obvious behavior that for all inputs less than or equal to 100, the function returns the constant 91.

In the following trace, all phases of the analysis are shown.

```
- g '^(||' program
           procedure p91(var x; val y);
              pre  true;
              post 100 < 'y  =>  x = 'y - 10  |  x = 91;

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

           p91(a; 77)
```

```
        end program
        { a = 91 }
  '||)';
```

Note that there are no **calls** ... **with** or **recurses with** clauses, since these contribute to proving termination, and the goal here is only partial correctness.

The HOL system repeats this back to us, and we then issue the VCGP_TAC tactic. As the tactic runs, it produces a trace of the Floyd/Hoare triples proven. As before, these are printed as $\{p\}c\{q\}/r$.

```
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         program
            procedure p91(var x;val y);
               pre true;
               post 100 < 'y => x = 'y - 10 | x = 91;

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

            p91(a;77)
         end program
         {a = 91}


     : proofs
- et(VCGP_TAC);
For procedure p91,

By the ASSIGN rule, we have
{(100 < 'y => y - 10 = 'y - 10 | y - 10 = 91)}
   x := y - 10
{(100 < 'y => x = 'y - 10 | x = 91)} /r

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

By the CALL rule, we have
{true /\
 (!x y1.
```

```
    (100 < y + 11 => x = (y + 11) - 10 | x = 91) ==>
    true /\
    (!x1 y1.
       (100 < x => x1 = x - 10 | x1 = 91) ==>
       (100 < 'y => x1 = 'y - 10 | x1 = 91)))}
   p91(x;y + 11)
{true /\
 (!x1 y1.
    (100 < x => x1 = x - 10 | x1 = 91) ==>
    (100 < 'y => x1 = 'y - 10 | x1 = 91))} /r

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

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

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

```
which is transformed in the context of the equality of
logical and program variables (at the procedure's entrance) to
true ==>
(100 < y => (100 < y => y - 10 = y - 10 | y - 10 = 91) |
   true /\
   (!x y1.
      (100 < y + 11 => x = (y + 11) - 10 | x = 91) ==>
      true /\
      (!x1 y1.
         (100 < x => x1 = x - 10 | x1 = 91) ==>
         (100 < y => x1 = y - 10 | x1 = 91))))


For the main body,

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

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


CPU: usr: 6.680 s    sys: 0.010 s    gc: 1.150 s    theorems: 164323
> val it =
    !a. (if 100 < 77 then a = 77 - 10 else a = 91) ==> (a = 91)


    !y.
      (if 100 < y then
         (if 100 < y then T else y - 10 = 91)
       else
         !x.
           (if 100 < y + 11 then x = y + 11 - 10 else x = 91) ==>
           !x1.
             (if 100 < x then x1 = x - 10 else x1 = 91) ==>
             (if 100 < y then x1 = y - 10 else x1 = 91))

     : goalstack
```

After proving these two subgoals by normal means, we see HOL print:

```
> val it =
    Initial goal proved.
```

```
|- program
      procedure p91(var x;val y);
         pre true;
         post 100 < 'y => x = 'y - 10 | x = 91;

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

      p91(a;77)
   end program
   {a = 91} : goalstack
```

7.3.3 Translation into HOL logic

In this phase, each verification condition is translated into the HOL logic according to the semantics of the assertion language, which is deeply embedded within the HOL logic. Thus every assertion has a corresponding term in the HOL logic, whose truth is equivalent to that of the assertion.

The results are seen in the example above, after the line with the CPU timings and theorem count.

For the most part, care is taken to translate variables from the assertion language to a similarly-named variable in the HOL logic, so that there is continuity between the two languages.

This process is completely automatic and cannot fail.

7.4 VCG_TAC

The VCG_TAC tactic reduces the given program total correctness goal to its VCG subgoals in a process which is completely secure, without any chance of unsoundness in the process. All computations are performed solely by secure proof within the HOL theorem prover, so the results are guarranteed to be logically consistent with the original goal. This means that if all the subgoals are proven by the user, then the HOL subgoal package will return a fully verified HOL theorem stating the total correctness goal which was originally posited.

Once the goal is entered into HOL and presented to the user as the top goal on the stack, it can then be analyzed by the Sunrise VCG by the tactic VCG_TAC. By default, this tactic prints out a trace as it works of the rules of the Floyd/Hoare-style axiomatic semantics that it is using as it analyzes the program and specification.

Overall, there are five phases of operation of VCG_TAC:

1. Well-formedness analysis of entire program
   - Syntax of entire program checked to make sure well-formedness conditions are not violated—
     any violation here causes the tactic to fail
2. Syntactic analysis of procedure declarations
   - VCs generated to confirm the partial correctness and entrance progress conditions of each procedure in turn
3. Graph analysis of procedure call graph
   - VCs generated to confirm the termination of all procedures in community with each other, given the properties from the previous phase
4. Syntactic analysis of main program commands
   - VCs generated to confirm the total correctness of the main body of the program, given the properties from the previous phases
5. Translation of VCs to HOL logic
   - VCs generated previously are now translated into equivalent statements in the HOL logic, according to the semantics of the assertion language

Each of the middle three phases generate verification conditions, and the trace describes this work in detail.

This tactic is very similar to VCGP_TAC discussed earlier, and so we will stress only the differences here for total correctness, where the VCGP_TAC tactic analyzed only partial correctness.

The primary difference is that there is a new phase of analysis, where the procedure call graph is traced to generate verification conditions to ensure the termiation of all recursive procedure calls, eliminating infinite recursive descent.

In the following, we will discuss only this new phase of graph analysis.

7.4.1 Graph Analysis of Procedure Call Graph

During the earlier phase of syntactic analysis of the procedure definitions, the VCG generated verification conditions sufficient to verify the correctness of the "**calls** ... **with**" entrance progress conditions specified in the header of each procedure. Now we can use that knowledge to establish the correctness of the recursive progress conditions specified in the header of each procedure. In the earlier phase, each procedure's specifications were checked independently of the others. In this phase, all the "**recurses with**" recursion progress conditions of all procedures are verified simultaneously. This establishes the termination of all procedure calls.

In the earlier phase, the analysis was structured according to the syntactic structure of the body of each procedure. In this phase, all the procedures are analyzed by the structure of the procedure call graph, as described in the "**calls** ... **with**" entrance progress conditions in the procedure headers. This process and the theory is described in full in the sources mentioned in the first chapter. To use this tool, it is sufficient to understand that each procedure is checked in turn, and its recursive progress condition is pushed backwards through the procedure call graph along all possible paths until a necessary verification condition is generated on each tip of the exploration. Then the union of all these necessary conditions is sufficient to justify the termination of all procedures.

We will demonstrate this with a program that calculates whether a number is odd or even, using two mutually recursive procedures. We are not here interested in the partial correctness part of the analysis, so the syntactic analysis will be elided from the transcript.

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

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

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

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

```
                    fi
               fi
            end procedure;

            odd(a; 5)

         end program
         [ a = 1 ]
      '||)';
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         program
            procedure odd(var a;val n);
               global ;
               pre true;
               post (?b. 'n = 2 * b + a) /\ (a < 2 /\ n = 'n);
               calls odd with n < 'n; calls even with n < 'n;
               recurses with n < 'n;

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

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

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

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


     : proofs
- et(VCG_TAC);
OK..
```

[*... syntactic analysis of procedures omitted ...*]

```
Exploring the structure of the procedure call graph:

Traversing the call graph back from the procedure odd:

odd
n < 'n
|
<-- odd
    !a n1. n1 < n ==> n1 < 'n
    |
    Undiverted recursion verification condition for the path
    odd->odd:
    true /\ n = 'n ==> (!a n1. n1 < n ==> n1 < 'n)
|
<-- even
    !a n1. n1 < n ==> n1 < 'n
    |
    <-- odd
        !a n1. n1 < n ==> (!a n2. n2 < n1 ==> n2 < 'n)
        |
        Undiverted recursion verification condition for the path
        odd->even->odd:
        true /\ n = 'n ==>
        (!a n1. n1 < n ==> (!a n2. n2 < n1 ==> n2 < 'n))
    |
    <-- even
        !a n1. n1 < n ==> (!a n2. n2 < n1 ==> n2 < 'n)
        |
        Diversion verification condition for the path
        even->even->odd:
        (!a n1. n1 < n ==> n1 < 'n) ==>
        (!a n1. n1 < n ==> (!a n2. n2 < n1 ==> n2 < 'n))

Traversing the call graph back from the procedure even:

even
n < 'n
|
<-- odd
    !a n1. n1 < n ==> n1 < 'n
    |
    <-- odd
        !a n1. n1 < n ==> (!a n2. n2 < n1 ==> n2 < 'n)
        |
        Diversion verification condition for the path
        odd->odd->even:
        (!a n1. n1 < n ==> n1 < 'n) ==>
        (!a n1. n1 < n ==> (!a n2. n2 < n1 ==> n2 < 'n))
```

```
    |
    <-- even
        !a n1. n1 < n ==> (!a n2. n2 < n1 ==> n2 < 'n)
        |
        Undiverted recursion verification condition for the path
        even->odd->even:
        true /\ n = 'n ==>
        (!a n1. n1 < n ==> (!a n2. n2 < n1 ==> n2 < 'n))
|
<-- even
    !a n1. n1 < n ==> n1 < 'n
    |
    Undiverted recursion verification condition for the path
    even->even:
    true /\ n = 'n ==> (!a n1. n1 < n ==> n1 < 'n)

For the main body,

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

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

9 subgoals:

CPU: usr: 23.210 s    sys: 0.050 s    gc: 2.720 s
> val it =
    !a n1. (?b. 5 = 2 * b + a) /\ a < 2 /\ (n1 = 5) ==> (a = 1)


    !n 'n. (n = 'n) ==> !n1. n1 < n ==> n1 < 'n


    !n 'n. (n = 'n) ==> !n1. n1 < n ==> !n2. n2 < n1 ==> n2 < 'n


    !n 'n.
      (!n1. n1 < n ==> n1 < 'n) ==> !n1. n1 < n ==> !n2. n2 < n1 ==> n2 < 'n
```

```
!n 'n.
  (!n1. n1 < n ==> n1 < 'n) ==> !n1. n1 < n ==> !n2. n2 < n1 ==> n2 < 'n


!n 'n. (n = 'n) ==> !n1. n1 < n ==> !n2. n2 < n1 ==> n2 < 'n


!n 'n. (n = 'n) ==> !n1. n1 < n ==> n1 < 'n


!n.
  (if n = 0 then
     (?b. n + 1 = 2 * b + 1) /\ 1 < 2
   else
     (if n = 1 then
        n - 1 < n /\
        !a n2.
          (?b. n - 1 = 2 * b + a) /\ a < 2 /\ (n2 = n - 1) ==>
          (?b. n + 1 = 2 * b + a) /\ a < 2
      else
        n - 2 < n /\
        !a n2.
          (?b. n - 2 + 1 = 2 * b + a) /\ a < 2 /\ (n2 = n - 2) ==>
          (?b. n + 1 = 2 * b + a) /\ a < 2))


!n.
  (if n = 0 then
     (?b. n = 2 * b + 0) /\ 0 < 2
   else
     (if n = 1 then
        n - 1 < n /\
        !a n2.
          (?b. n - 1 + 1 = 2 * b + a) /\ a < 2 /\ (n2 = n - 1) ==>
          (?b. n = 2 * b + a) /\ a < 2
      else
        n - 2 < n /\
        !a n2.
          (?b. n - 2 = 2 * b + a) /\ a < 2 /\ (n2 = n - 2) ==>
          (?b. n = 2 * b + a) /\ a < 2))

 : goalstack
```

After proving these nine subgoals by normal means, we see HOL print the total correctness result:

```
> val it =
    Initial goal proved.
    |- program
```

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

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

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

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

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

7.5 FAST_VCG*_TAC

The four VCG*_TAC tactics described above (VCGCP_TAC, VCGC_TAC, VCGP_TAC, VCG_TAC) are completely secure, but at the price of conducting all calculations by secure proof within HOL. This may be too slow for practical use in some cases. Therefore we also provide faster versions of the VCG*_TAC tactics, called (correspondingly) FAST_VCG*_TAC.

These tactics performs essentially the same tasks as VCG*_TAC, but many times faster. In tradeoff for this speed, we give up security. The calculation of the well-formedness test and the calculation of the verification conditions are the two most time-intensive parts of VCG*_TAC. In FAST_VCG*_TAC, these are replaced by calls to ML code to compute the same functions, as an oracle (DESCRIPTION 3.10).

These ML functions have been coded carefully, and should compute the same results as the secure proof versions, but *this cannot be assurred with mathematical certainty*, so **the user is warned that some (small) possibility of error exists when using these faster tactics.**

Tests have shown that these recoded portions may run up to or beyond three orders of magnitude faster than the secure versions. However, the user will not see this factor of speedup since there are other portions of the FAST_VCG*_TAC tactic that are still done by secure proof.

It is envisioned that in a normal software development process, that the fast tactic may be used many times during development and modification of the software, and only at the end when the software is stable and the proofs are complete, will it be necessary to run the completely secure version once.

The FAST_VCG*_TAC tactics do not provide the tracing output available from the VCG*_TAC tactics.

7.6 TEST_VCG*_TAC

If there is ever any doubt about the equality of the results of the VCG*_TAC and FAST_VCG*_TAC tactics, then the tactics TEST_VCG*_TAC are provided to automatically run both tactics, compare the results, and identify any discrepancies. In addition, they provide timing results, comparing the secure and insecure tactics. Other than this, they perform the same as the VCG*_TAC tactics.

7.7 Global Flags

The printing and operation of all twelve VCG tactics is controlled by a set of global flags which the user can set as he wishes. The flags, with their default values, are summarized below.

Flag	Default	Effect when true
print_vcg	true	causes tracing of the VCG's work.
print_subst	false	causes tracing of substitutions.
debug_vcg	false	causes deep tracing of the VCG.
caching	true	causes caching of some theorems.
debug_cache	false	causes deep tracing of the cache.
timing	false	causes timing of the VCG's work.
new_graph	true	prints 'diagram' of graph analysis.

"print_vcg" is useful for turning off the trace when it is not of interest. Except for the flag "print_vcg," however, these flags are not recommended for use by people other than the Sunrise system designer, as the output is cryptic and volumuous. However, "caching" may affect the speed performance for some programs. For one of the larger included examples, caching improves the speed by a factor of 3. Generally caching should help, but for some cases it is possible it may slow the VCG.

Chapter 8

Verification Condition Generator Trace Explained

The purpose of this chapter is to explain how the Floyd/Hoare-style rules of the axiomatic semantics of the Sunrise programming language are used to prove the partial correctness of a program by the verification condition generator.

This will be shown in the context of a simple example, the "91 function" invented by John McCarthy. These rules will be found in the papers and in the dissertation mentioned in Chapter 1.

The trace of the execution of the Verification Condition Generator is given at the end of this chapter. We will here examine some pieces of the output, and justify them so that the reader can see how they were computed.

Central in this will be the concept of substitution. We will use the infix triangle notation $a \lhd [t_1, \ldots, t_n/x_1, \ldots, x_n]$ to denote the application of the substitution $[t_1, \ldots, t_n/x_1, \ldots, x_n]$ to the expression a. The effect of this is that across the expression a, every free occurrence of the variables $x_1, \ldots, x_n$ is replaced (simultaneously) by an occurence of $t_1, \ldots, t_n$, so x_1 is replaced by t_1, x_2 by t_2, and so on, respectively.

The first element of the trace is the application of the Assignment rule to the command $x := y - 10$, with postcondition $(100 < {}'y => x = {}'y - 10 \mid x = 91)$.

The Assignment rule is repeated here for clarity:

$$\overline{\{q \lhd [x := e]\}\ x := e\ \{q\}/\rho}$$

Now, we apply the Assignment rule to this example by associating

$$\begin{aligned} x &= x \\ e &= y - 10 \\ q &= (100 < {}'y => x = {}'y - 10 \mid x = 91) \end{aligned}$$

Then we compute the precondition $q \lhd [x := e]$ as

$$\begin{aligned} q \lhd [x := e] &= (100 < {}'y => x = {}'y - 10 \mid x = 91) \lhd [y - 10/x] \\ &= (100 < {}'y => (y - 10) = {}'y - 10 \mid (y - 10) = 91) \end{aligned}$$

which gives the same Hoare triple as in the trace:

```
By the ASSIGN rule, we have
{(100 < 'y => y - 10 = 'y - 10 | y - 10 = 91)}
   x := y - 10
{(100 < 'y => x = 'y - 10 | x = 91)} /r
```

Now, that was pretty simple. By contast, the procedure call rule shows an interesting depth of complexity. The Procedure Call rule is given at the bottom of the table titled, "Hoare Logic for Partial Correctness," Table 6.6 on page 107 of the dissertation.

$$\frac{\begin{array}{c} \mathit{WF}_{envp}\ \rho, \quad \mathit{WF}_c\ (\textbf{call}\ p(xs; es))\ g\ \rho \\ \rho\ p = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle \\ vals' = \mathit{variants}\ vals\ (\mathit{FV}_a\ q \cup \mathit{SL}(xs\ \&\ glbs)), \quad y = vars\ \&\ vals\ \&\ glbs \\ u = xs\ \&\ vals', \quad v = vars\ \&\ vals, \quad x = xs\ \&\ vals'\ \&\ glbs \\ x_0 = \mathit{logicals}\ x, \quad y_0 = \mathit{logicals}\ y, \quad x'_0 = \mathit{variants}\ x_0\ (\mathit{FV}_a\ q) \end{array}}{\begin{array}{c} \{(pre \lhd [u/v]\ \wedge\ (\forall x.\ (post \lhd [u, x'_0/v, y_0] \Rightarrow q)) \lhd [x/x'_0])) \lhd [vals' := es]\} \\ \textbf{call}\ p(xs; es)\{q\}\ /\rho \end{array}}$$

The next element of the trace is a procedure call, so we can see how the Procedure Call rule is applied. The particular procedure call is $p91(x; x)$, with the same postcondition $(100 < {}'y => x = {}'y - 10 \mid x = 91)$.

We assume that in the procedure environment ρ, that the name $p91$ is associated with the following tuple:

$$\rho\ p91 = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle$$

where

$$\begin{array}{rcl} vars & = & [x] \\ vals & = & [y] \\ glbs & = & [\,] \\ pre & = & true \\ post & = & (100 < {}'y => x = {}'y - 10 \mid x = 91) \\ calls & = & p91 \mapsto (101 - y < 101 - {}'y) \\ rec & = & 101 - y < {}'z \end{array}$$

and where c is the body of the procedure,

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

We apply the Procedure Call rule by associating

$$\begin{array}{rcl} p & = & p91 \\ xs & = & [x] \\ es & = & [x] \\ q & = & (100 < {}'y => x = {}'y - 10 \mid x = 91) \end{array}$$

According to the rule, we first compute some auxilliary variables:

$$\begin{array}{rcl} vals' & = & [y] \\ x & = & [x, y] \\ y & = & [x, y] \\ u & = & [x, y] \\ v & = & [x, y] \\ x_0 & = & [{}'x, {}'y] \\ y_0 & = & [{}'x, {}'y] \\ x_0' & = & [{}'x, {}'y_1] \end{array}$$

Then we can compute the preconditon in stages as follows:

$$\begin{array}{rcl} pre \lhd [u/v] & = & true \lhd [x, y/x, y] \\ & = & true \end{array}$$

$$\begin{array}{rcl} post \lhd [u, x_0'/v, y_0] & = & (100 < {}'y => x = {}'y - 10 \mid x = 91) \lhd [x, y, {}'x, {}'y_1/x, y, {}'x, {}'y] \\ & = & (100 < {}'y_1 => x = {}'y_1 - 10 \mid x = 91) \end{array}$$

So let $post1 = post \lhd [u, x_0'/v, y_0]$. Then we compute

$$\begin{array}{l} (\forall x.\ post1 \Rightarrow q) \lhd [x/x_0'] \\ \quad = \quad (\forall x, y.\ (100 < {}'y_1 => x = {}'y_1 - 10 \mid x = 91) \Rightarrow (100 < {}'y => x = {}'y - 10 \mid x = 91)) \\ \qquad \lhd [x, y/{}'x, {}'y_1] \\ \quad = \quad (\forall x_1, y_1.\ (100 < y => x_1 = y - 10 \mid x_1 = 91) \Rightarrow (100 < {}'y => x_1 = {}'y - 10 \mid x_1 = 91)) \end{array}$$

Note that the quantifications $\forall x, y$ have shifted to $\forall x_1, y_1$ automatically, in order to avoid capturing the free variables x and y which the substitution could have introduced.

Now let $post2 = (\forall x.\ post1 \Rightarrow q) \lhd [x/x_0']$, just computed. Next we need $post2 \lhd [vals' := es]$. To compute this, we must first determine the substitution represented by $[vals' := es]$.

By the definitions on pages 248, 249, and 251 of the dissertation,

$$\begin{array}{rcl} [vals' := es] & = & (\mathit{VES\_state}\ es)[(\mathit{VES}\ es)/vals'] \\ \mathit{VES}\ es & = & \mathit{FST}\ (\mathit{VES1}\ es\ \iota) \\ \mathit{VES\_state}\ es & = & \mathit{SND}\ (\mathit{VES1}\ es\ \iota) \\ \iota\ x & = & x \end{array}$$

so we need to compute *VES*1 $es\ \iota$ by the definitions on pages 247-8:

$$\begin{aligned}
\mathit{VES}1\ es\ \iota &= \mathit{VES}1\ [x]\ \iota \\
&= (\mathit{VE}1\ x \rightarrow \lambda v.\ (\mathit{VES}1\ [\,] \rightarrow \lambda vs\ ss_2.\ (\mathit{CONS}\ v\ vs,\ ss_2)))\ \iota \\
&= \textbf{let}\ (v', ss') = \mathit{VE}1\ x\ \iota\ \textbf{in}\ (\lambda v.\ (\mathit{VES}1\ [\,] \rightarrow \lambda vs\ ss_2.\ (\mathit{CONS}\ v\ vs,\ ss_2)))\ v'\ ss' \\
&= \textbf{let}\ (v', ss') = (\iota\ x,\ \iota)\ \textbf{in}\ (\mathit{VES}1\ [\,] \rightarrow \lambda vs\ ss_2.\ (\mathit{CONS}\ v'\ vs,\ ss_2))\ ss' \\
&= (\mathit{VES}1\ [\,] \rightarrow \lambda vs\ ss_2.\ (\mathit{CONS}\ x\ vs,\ ss_2))\ \iota \\
&= \textbf{let}\ (vs', ss') = \mathit{VES}1\ [\,]\ \iota\ \textbf{in}\ (\lambda vs\ ss_2.\ (\mathit{CONS}\ x\ vs,\ ss_2))\ vs'\ ss' \\
&= \textbf{let}\ (vs', ss') = ([\,],\ \iota)\ \textbf{in}\ (\mathit{CONS}\ x\ vs',\ ss') \\
&= (\mathit{CONS}\ x\ [\,],\ \iota) \\
&= ([x],\ \iota)
\end{aligned}$$

In this example, the expressions es did not have any side effects, and so we have the resulting identity substitution ι in the result above. In general, any side effects produced by es will be represented in the substitution returned as the second element of the pair computed here.

So we compute $[vals' := es]$ as

$$\begin{aligned}
[vals' := es] &= (\mathit{VES_state}\ es)[(\mathit{VES}\ es)/vals'] \\
&= (\mathit{SND}\ (\mathit{VES}1\ es\ \iota))\ [\ (\mathit{FST}\ (\mathit{VES}1\ es\ \iota))\ /\ [y]\ ] \\
&= (\iota)[\ [x]/[y]\ ] \\
&= \iota[x/y] \\
&= [x/y]
\end{aligned}$$

Now we are ready to compute $post2 \lhd [vals' := es]$.

$$\begin{aligned}
&post2 \lhd [vals' := es] \\
&= (\forall x_1, y_1.\ (100 < y \Rightarrow x_1 = y - 10 \mid x_1 = 91) \Rightarrow (100 < {}'y \Rightarrow x_1 = {}'y - 10 \mid x_1 = 91)) \\
&\quad \lhd [x/y] \\
&= (\forall x_1, y_1.\ (100 < x \Rightarrow x_1 = x - 10 \mid x_1 = 91) \Rightarrow (100 < {}'y \Rightarrow x_1 = {}'y - 10 \mid x_1 = 91))
\end{aligned}$$

The remaining part comes from the Procedure Call rule from the table entitled "Entrance Logic," Table 6.8 on page 112 in the dissertation. This shows how to prove that the "calls ... with" declarations are satisfied by a procedure's body.

$$\frac{\begin{array}{c}\mathit{WF}_{env\_syntax}\ \rho,\quad \mathit{WF}_c\ (\textbf{call}\ p(xs; es))\ g\ \rho \\ \rho\ p = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle \\ vals' = \mathit{variants}\ vals\ (\mathit{SL}(xs\ \&\ glbs))\end{array}}{\begin{array}{c}\{(q \lhd [xs\ \&\ vals'/vars\ \&\ vals]) \lhd [vals' := es]\} \\ \textbf{call}\ p(xs; es) \rightarrow p\ \{q\}\ /\rho\end{array}}$$

This Procedure Call rule uses a postcondition q which is taken from the expression specified in the "calls ... with" clause for this procedure $p91$. In this example, the clause is

$$q = 101 - y < 101 - {}'y$$

Here we need to compute

$$\begin{aligned} & q \lhd [xs \ \& \ vals'/vars \ \& \ vals] \\ = \ & (101 - y < 101 - {}'y) \lhd [x, y/x, y] \\ = \ & (101 - y < 101 - {}'y) \end{aligned}$$

and then for the last substitution, let $q1 = q \lhd [xs \ \& \ vals'/vars \ \& \ vals]$, and then:

$$\begin{aligned} & q1 \lhd [vals' := es] \\ = \ & (101 - y < 101 - {}'y) \lhd [x/y] \\ = \ & (101 - x < 101 - {}'y) \end{aligned}$$

In fact, the Verification Condition Generator economically combines both of these two Procedure Call rules into one rule, which calculates the combined precondition as

$$pre \lhd [u/v] \wedge ((q1 \wedge post2) \lhd [vals' := es])$$

This is combination is shown in the definition of the *vcg*1 function in Figure 7.1 on page 137 of the dissertation.

This is sufficient to ensure both partial correctness and that the entrance condition is satisfied.

Putting the pieces computed earlier together, this precondition matches what was computed in the verificaton condition generator's trace:

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

This completes the explanation of the CALL rule trace.

In a similar manner, the preconditions of calls across the procedure call graph may be found using the Call Progress Rule, given on Page 119 of the dissertation. This rule is evaluated using the definition of the *call_progress* function, defined in Table 6.11 on page 120. The mechanization of this graph analysis is primarily performed by two mutually recursive functions, *extend_graph_vcs* and *fan_out_graph_vcs*, as given in Figure 7.4 on page 140. This graph analysis is described with examples in those parts of the dissertation, so it will not be further elaborated here.

This ends our discussion of the trace of the "91 function" example. The trace itself is given beginning on the next page.

This shows a taste of the calculations which are automatically computed by the VCG, without any possibility of error or omission.

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

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

        p91(a; 77)

     end program
     [ a = 91 ]
  ‘||)‘;
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         program
            procedure p91(var x;val y);
               global ;
               pre true;
               post 100 < ’y => x = ’y - 10 | x = 91;
               calls p91 with 101 - y < 101 - ’y;
               recurses with 101 - y < ’z;

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

            p91(a;77)
         end program
         [a = 91]


     : proofs
- et(VCG_TAC);
OK..
For procedure "p91",

By the ASSIGN rule, we have
{(100 < ’y => y - 10 = ’y - 10 | y - 10 = 91)}
   x := y - 10
{(100 < ’y => x = ’y - 10 | x = 91)} /r
```

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

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

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

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

By precondition strengthening, we have
{true}
```

```
   if 100 < y then x := y - 10 else p91(x;y + 11); p91(x;x) fi
{(100 < 'y => x = 'y - 10 | x = 91)} /r
with additional verification condition
true ==>
(100 < y => (100 < 'y => y - 10 = 'y - 10 | y - 10 = 91) |
   (true /\ 101 - (y + 11) < 101 - 'y) /\
   (!x y1.
      (100 < y + 11 => x = (y + 11) - 10 | x = 91) ==>
      (true /\ 101 - x < 101 - 'y) /\
      (!x1 y1.
         (100 < x => x1 = x - 10 | x1 = 91) ==>
         (100 < 'y => x1 = 'y - 10 | x1 = 91))))

which is transformed in the context of the equality of
logical and program variables (at the procedure's entrance) to
true ==>
(100 < y => (100 < y => y - 10 = y - 10 | y - 10 = 91) |
   (true /\ 101 - (y + 11) < 101 - y) /\
   (!x y1.
      (100 < y + 11 => x = (y + 11) - 10 | x = 91) ==>
      (true /\ 101 - x < 101 - y) /\
      (!x1 y1.
         (100 < x => x1 = x - 10 | x1 = 91) ==>
         (100 < y => x1 = y - 10 | x1 = 91))))

Exploring the structure of the procedure call graph:

Traversing the call graph back from the procedure p91:

p91
101 - y < 'z
|
<-- p91
    !x y1. 101 - y1 < 101 - y ==> 101 - y1 < 'z
    |
    Undiverted recursion verification condition for the path
    p91->p91:
    true /\ 101 - y = 'z ==>
    (!x y1. 101 - y1 < 101 - y ==> 101 - y1 < 'z)

For the main body,

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

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

3 subgoals:

CPU: usr: 8.620 s    sys: 0.040 s    gc: 0.770 s
> val it =
    !a. (if 100 < 77 then a = 77 - 10 else a = 91) ==> (a = 91)


    !y 'z. (101 - y = 'z) ==> !y1. 101 - y1 < 101 - y ==> 101 - y1 < 'z


    !y.
      (if 100 < y then
         (if 100 < y then T else y - 10 = 91)
       else
         101 - (y + 11) < 101 - y /\
         !x.
           (if 100 < y + 11 then x = y + 11 - 10 else x = 91) ==>
           101 - x < 101 - y /\
           !x1.
             (if 100 < x then x1 = x - 10 else x1 = 91) ==>
             (if 100 < y then x1 = y - 10 else x1 = 91))

     : goalstack
- et(ARITH_TAC);
OK..

Goal proved.
|- !y.
     (if 100 < y then
        (if 100 < y then T else y - 10 = 91)
      else
        101 - (y + 11) < 101 - y /\
        !x.
          (if 100 < y + 11 then x = y + 11 - 10 else x = 91) ==>
          101 - x < 101 - y /\
          !x1.
            (if 100 < x then x1 = x - 10 else x1 = 91) ==>
            (if 100 < y then x1 = y - 10 else x1 = 91))

Remaining subgoals:

CPU: usr: 24.260 s    sys: 0.000 s    gc: 1.520 s
```

```
> val it =
    !a. (if 100 < 77 then a = 77 - 10 else a = 91) ==> (a = 91)


    !y 'z. (101 - y = 'z) ==> !y1. 101 - y1 < 101 - y ==> 101 - y1 < 'z

     : goalstack
- et(ARITH_TAC);
OK..

Goal proved.
|- !y 'z. (101 - y = 'z) ==> !y1. 101 - y1 < 101 - y ==> 101 - y1 < 'z

Remaining subgoals:

CPU: usr: 0.590 s    sys: 0.000 s    gc: 0.040 s
> val it =
    !a. (if 100 < 77 then a = 77 - 10 else a = 91) ==> (a = 91)

     : goalstack
- et(ARITH_TAC);
OK..

Goal proved.
|- !a. (if 100 < 77 then a = 77 - 10 else a = 91) ==> (a = 91)

CPU: usr: 0.250 s    sys: 0.000 s    gc: 0.000 s
> val it =
    Initial goal proved.
    |- program
          procedure p91(var x;val y);
             global ;
             pre true;
             post 100 < 'y => x = 'y - 10 | x = 91;
             calls p91 with 101 - y < 101 - 'y;
             recurses with 101 - y < 'z;

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

          p91(a;77)
       end program
       [a = 91] : goalstack
```